## Table of Contents

Chapter 14. Data-Bound and Templated Controls

- Designing a Data-Bound Control .................................................. 1
- Building a Simple Data-Bound Control .......................................... 2
- Building a List Data-Bound Control .............................................. 10
- Building a Composite Templated Control ...................................... 21
- Conclusion ..................................................................................... 33

Conclusion ..................................................................................... 55
Chapter 14. Data-Bound and Templated Controls

All sample custom controls we have built thus far are effective and functional but lack a few key features. More often than not, real-world controls need to be data-bindable and support templates. The importance of a control being data-bindable surfaced clearly in Chapter 11, in which we thoroughly discussed and explored the Repeater and DataList controls. The topic was covered even more extensively in Part 2 of Programming Microsoft ASP.NET 2.0: Core Reference (Microsoft Press 2005), the companion book to this one.

For an ASP.NET programmer, a control that can be bound to a collection of data is an essential aid that smooths development and significantly increases both personal and team productivity. Similarly helpful is built-in support for templates, which makes the control much more flexible and reduces by an order of magnitude the need for further customization. Often when you buy a third-party feature-rich control, you’re not completely satisfied with its user interface and overall set of capabilities and functions. Many software houses buy only components that come with full (and commented) source code, primarily so that they can solve any issues that derive from the integration between the component and the core application. Templates provide a way to customize controls. Templates don’t ensure that you’ll never want something more or different from a component;
however, templates have been introduced mainly as a way to make portions of the user interface as generic and parametric as possible.

In this chapter, we’ll take the gauge control introduced in Chapter 13 to the next level by adding data-binding capabilities. We’ll also design and implement a couple more sample controls to provide a broader view of data-bound controls. In particular, we’ll discuss list controls and composite data-bound controls with template support.

**Designing a Data-Bound Control**

A data-bound control is mainly characterized by a data source property and a bunch of string properties to define mappings between fields in the bound data source and parts of the control’s user interface. In ASP.NET 1.x, the data source property can point only to an object of type `IEnumerable` (or any derived type) or `IListSource`. ASP.NET 2.0 adds a new data source model. It consists of a bunch of new user interface–less controls that bridge the gap between visual parts of data-bound controls and data containers. Basically, the vast majority of code that developers were required to write in ASP.NET 1.x, properly factored and authored, is now embedded in a new family of controls—data source controls. Data source controls are covered in great detail in Chapter 9 of *Programming Microsoft ASP.NET 2.0: Core Reference* (Microsoft Press 2005).

There are many benefits to using data source controls in ASP.NET pages. First and foremost, we have the possibility of a fully declarative data-binding model. The new model reduces the loose code inserted inline in `.aspx` files or scattered through code-behind classes. The new data-binding architecture forces developers to play by strict rules. Moreover, it inherently changes the quality of the code to be written. Long blocks of code attached to events tend to disappear and are replaced by components that just plug into the existing framework. These components (for example, data source controls) derive from abstract classes, implement well-known interfaces, and overall signify a higher level of reusability.

How does this affect control developers? In ASP.NET 2.0, any data-bound control must be able to support both forms of binding—against enumerable objects and data source controls. With ASP.NET 2.0, you get a completely redesigned graph of classes that add more specific data-binding capabilities as you scroll the tree from base to leaf classes. The new hierarchy of data-bound controls makes it easier for everybody to pick up the right class to inherit from for building their own custom data-bound control.
Types of Data-Bound Controls

Any server control can be designed to automatically retrieve some of its data from an external data source. Based on the interaction between the data source and the control, you can identify three distinct forms of data binding and, subsequently, data-bound controls—simple binding, list controls, and complex binding.

Simple Data Binding

Simple data binding consists of binding an object and one or more control properties. The data source binds to an individual item as opposed to a list of items. The internal structure of a control that uses simple binding is nearly identical to that of a complex bound control—only simpler.

The *SimpleGaugeBar* control that we discussed in the previous chapter is a good example of a simple data-bound control. Later, we’ll extend the *SimpleGaugeBar* control to make the *Value* and *FormatString* properties bindable to a field on a data source object or data source control.

List Controls

List controls are controls that display a list of data items through a fixed and immutable user interface. Popular examples of list controls are *RadioButtonList*, *CheckBoxList* and, new in ASP.NET 2.0, *BulletedList*. The main trait of list controls is the strict association between data items and child controls—such as a radio button, check box, or bullet point.

As we’ll see later in the chapter, a custom list control can be built by defining a control to repeat and initialize the control with the data contained in each bound data item. Most list controls also feature advanced layout capabilities such as rendering data in a given number of columns or rows. We’ll see how to add that capability to custom controls, too.

Complex Data Binding

Complex data-bound controls are typically composite controls that display a list of items with no limitation at all on the rendering mechanism. A good example of complex data-bound control is the *DataGrid* control.

In this chapter, we’ll design and implement a *BarChart* control as a real-world example of complex binding and template support. The *BarChart* control renders as an HTML table where each row
represents a horizontal bar in a chart. Needless to say, information for the chart comes from a bound data source.

**Important**

In ASP.NET 1.x, data binding works only in one direction, meaning that the control has typically no way to update the data source. Each data-bound control generally reads data from the data source and fires events when something happens that requires updates on the bound source. In ASP.NET 2.0, data source controls form the underlying machinery for two-way data binding. In the whole ASP.NET 2.0 control toolbox, only three controls support two-way binding—`GridView`, `FormView`, and `DetailsView`. Two-way data binding requires additional logic in the control code to invoke proper methods on the bound data source, which has to be a data source control. All examples in this book refer to the most common scenario—one-way data binding.

**The Data-Binding Mechanism**

Figure 14-1 presents the list of data-bound controls available in ASP.NET 2.0. As you can see, all classes derive from `BaseDataBoundControl`—the root class for ASP.NET 2.0 data-bound controls.
The `BaseDataBoundControl` class defines the machinery through which the data binding occurs and validates any bound data. Common data source properties are defined on this class—`DataSource` for enumerable data, and `DataSourceID` for data source controls.

```csharp
public virtual object DataSource { get; set; }
public virtual string DataSourceID { get; set; }
```

Each property activates a distinct binding mechanism. The two properties are mutually exclusive; if both are set simultaneously, you run into an exception. Because `DataSource` and `DataSourceID` refer to distinct mechanisms, as a control developer, should you be ready to incorporate both in your code? To find an effective answer to this central question, we need to delve deeper in the internals of `BaseDataBoundControl` and its derivatives.

### Binding to a Data Source

The `DataSource` property accepts objects that implement the `IEnumerable` (for example, collections) or `IListSource` (for example, `DataTable`) interface. The assignment occurs just as in ASP.NET 1.x.

```csharp
ctl.DataSource = dataTable;
ctl.DataBind();
```
In addition to setting the **DataSource** property, you have to call the **DataBind** method on the control (or on the control’s container object—say, the page). Only when **DataBind** is invoked is the data physically loaded into the runtime image of the control.

The **DataSourceID** property is a string and refers to the ID of a bound data source control. Once a control is bound to a data source, any further interaction between the two (in both reading and writing) is out of your control and hidden from view. This is good and bad news at the same time. It is good (actually, it is great) news because you cut a large quantity of code from your pages. The ASP.NET framework guarantees that correct code executes and is written against recognized best practices. You’re more productive because you author pages faster with the inherent certainty of having no subtle bugs in the middle. If you don’t like this situation, you can stick to the ASP.NET 1.x programming style.

The internal architecture of data-bound controls changed quite a bit in the transition to ASP.NET 2.0. Most of the changes are incorporated in the **BaseDataBoundControl** class. Let’s start looking at what happens when you set **DataSource** and **DataSourceID** on any data-bound controls. The following code shows the **setter** of **DataSource**:

```csharp
// Pseudo-code for setting DataSource
set (object value)
{
    if (value != null)
        ValidateDataSource(value);
    _dataSource = value;
    OnDataPropertyChanged();
}
```

The passed data source object is first validated to check whether it is of an acceptable type and then cached in a private member. Finally, the internal **OnDataPropertyChanged** method is invoked. Let’s look at the **setter** of **DataSourceID** also:

```csharp
// Pseudo-code for setting DataSourceID
set (string value)
{
    ViewState["DataSourceID"] = value;
    OnDataPropertyChanged();
}
```

The value of **DataSourceID** is cached in the view state, and then a call to **OnDataPropertyChanged** is made. What happens within the **OnDataPropertyChanged** method? The method sets a protected Boolean variable named **RequiresDataBinding**. The protected member is defined on **BaseDataBoundControl** and inherited by all derived controls—it is set to **true** whenever the control requires binding to data.

Let’s take a deeper look at some key members of the **BaseDataBoundControl** class.
The **BaseDataBoundControl** Class

Most of the behavior of the **BaseDataBoundControl** class revolves around a few built-in event handlers, the override of **DataBind** (a method originally defined on **Control**), and a pair of virtual methods.

The class handles the **Init** and **PreRender** events. In the **OnInit** method, it automatically sets **RequiresDataBinding** to true if the host page is posting back. This trick guarantees that data-bound controls are correctly refreshed even when bound to data source controls. In the **OnPreRender** method, the class ensures that data binding is executed if required. Another internal method—named **EnsureDataBound**—takes care of this task. It checks **RequiresDataBinding** and calls **DataBind** if the property returns true.

As mentioned, the **DataBind** method is defined on the **Control** class and gets overridden in **BaseDataBoundControl**. Here’s the pseudo-code:

```csharp
public override void DataBind()
{
    PerformSelect();
}
```

Interestingly enough, **PerformSelect** is marked as abstract on **BaseDataBoundControl** so its implementation is left to derived classes. The ultimate goal of the method, though, is quite unequivocal—performing any operation aimed at retrieving the bound data. The second abstract method is **ValidateDataSource**, which is invoked when you set the **DataSource** property.

The **DataBoundControl** Class

**ValidateDataSource** is overridden, care of the **DataBoundControl** class. Here’s the related pseudo-code:

```csharp
protected override void ValidateDataSource(object dataSource)
{
    if (dataSource != null)
    {
        if (!dataSource is IListSource) &&
        !dataSource is IEnumerable) &&
        !dataSource is IDataSource)
            throw new InvalidOperationException(...);
    }
}
```
As you can see, the `DataSource` object can either be null or one of the following types—`IListSource`, `IEnumerable`, or `IDataSource`. `IDataSource` is the interface that characterizes data source controls. In other words, you can bind a control to a data source control in two ways—declaratively using the `DataSourceID` property, or programmatically by setting the `DataSource` property with the instance of the data source control.

`DataBoundControl` also overrides the `PerformSelect` method. Here’s how:

```csharp
protected override void PerformSelect()
{
    // Get the data-source-view object associated to the control
    DataSourceView view = GetData();

    // Invoke the SELECT operation on the bound data source control
    // _arguments is an internal member that references a data
    // structure filled with arguments for the SELECT operation
    _arguments = CreateDataSourceSelectArguments();
    view.Select(_arguments, OnDataSourceViewSelectCallback);

    // bound data is processed through the internal callback function
}
```

It connects to the data source object and obtains the default view. Next, it prepares a `SELECT` command to execute on the data source control, such as `SqlDataSource` or `ObjectDataSource`.

Even though control developers are offered two distinct ways of binding (`DataSource` vs. `DataSourceID`), under the hood of data-bound controls data retrieval occurs in just one way—through data source view objects. If the control is bound to a data source control, the incorporated data source view object is retrieved via the members of the `IDataSource` interface. If the control is bound to an enumerable object, a data source view object is dynamically built and returned by `GetData`. If the `DataSource` is non-empty, the bound object is wrapped in a dynamically created data source view object of type `ReadOnlyDataSource`—an internal and undocumented class.

By design, a `DataSourceView` object features the `Select` method and returns the bound data through it. As shown in the preceding pseudo-code, the `Select` method accepts some input arguments and a callback function. The callback function receives an enumerable collection of data—the items to bind to the control.

The bottom line is that whatever data source object you bind to the control, a data source view object is created. A data source view object is a class that can perform SELECT, INSERT, DELETE, and UPDATE operations on a bound object. `DataBoundControl` performs a SELECT operation on the data source view and obtains an enumerable collection of data. This collection contains the data to show in the control’s user interface regardless of whether `DataSource` or `DataSourceID` was used.

As a control developer, how can you access this bindable collection of data?
The **HierarchicalDataBoundControl** class has an internal structure that is nearly identical to **DataBoundControl** except that data source view objects are used with a hierarchical nature.

### The **PerformDataBinding** Overridable Method

The callback function that gets to process the results of the SELECT operation on the data source view ends up calling a protected overridable method—**PerformDataBinding**.

```csharp
protected virtual void PerformDataBinding(IEnumerable data)
{
    // data is the collection of data to show in the data-bound
    // control's user interface
}
```

The **PerformDataBinding** method defined on **DataBoundControl** contains no executable code. In derived classes, you override this method and load the bound data in internal structures as necessary.

In summary, data-bound controls support a larger set of features in ASP.NET 2.0 than in any previous versions. Building custom data-bound controls, though, has never been easier. In most cases, all that you have to do is create a new class from **DataBoundControl** and override the **PerformDataBinding** method.

Armed with this knowledge, let’s see how to build a few typical custom data-bound controls. Along the way, we’ll explore in more depth other data-bound control classes that appear in Figure 14-1.

### Note

**DataMember** was a pretty popular property for data-bound controls in ASP.NET 1.x and was often used in conjunction with **DataSource**. **DataMember** is a string property that selects a data member in the specified source. In ASP.NET 2.0, the two properties have been separated. **DataSource** is defined on **BaseDataBoundControl**, whereas **DataMember** is defined on **DataBoundControl**.
Building a Simple Data-Bound Control

In the previous chapter, we created the SimpleGaugeBar control as a composite control to display a notched indicator of a given quantity. By setting the Value and Maximum properties on the control, you can graphically represent a value on the proper scale. The SimpleGaugeBar control is not data bound, meaning that no elements in its programming interface can be automatically and declaratively bound to external data. Derived from CompositeControl, the SimpleGaugeBar control doesn’t incorporate any of the features listed previously regarding data-bound controls.

The goal of this section is to extend the SimpleGaugeBar control to make it support data binding through enumerable objects and data source controls.

Key Features

A data-bound version of SimpleGaugeBar is a form of simple binding. A couple of existing properties—Value and FormatString—can be automatically filled with external data according to the classic data-binding pattern of ASP.NET. A data source object specified through either DataSource or DataSourceID and with bindable properties is mapped to public fields on the data source object through mapper properties. In simple binding, the bound data source object is an individual object that contains just one logical piece of information—no items, no lists.

The key features of a data-bound control can be summarized as follows:

- Additional properties to represent mappings between control properties and data source fields
- An additional property to represent and persist the data source object
- Additional view-state management to persist the data source object
- Modified rendering to take bound data into account

Let’s dig out more.
Adding Data-Bound Properties

When you bind data to, say, a DropDownList control, you first set the data source and then specify which fields on the data source should be used to display the text and the value of the resulting list. The DropDownList control features a pair of DataTextField and DataValueField string properties.

The former is set to the name of the public field on the data source that will render the text of displayed list items. The latter is set to the name of the field on the bound data source object that will render the unique value associated with each displayed list item.

On a brand new data-bound control, you need to define similar properties to specify any required mapping between data source fields and bindable control properties. All these properties are usually string properties stored in the view state; the name is arbitrary, but it generally follows the pattern DataXxxField, where Xxx indicates the role of the bindable control property.

Adding a Data Item Property

By design, the bound data source object must be an object that implements any of the following interfaces: IEnumerable (collections), IListSource (ADO.NET objects), orIDataSource (data source controls). Let’s suppose you bind a control to one row of a DataTable. Do you really need to persist the whole data row? If yes, what if the data row contains a couple of large binary object (BLOB) fields?

The recommended approach entails that you extract a subset of information from the originally bound data source object and copy that to a control-specific data item object. This object is an instance of a custom class that typically has as many public properties as there are bindable properties on the control. For example, the DropDownList control has two bindable properties—Text and Value. Subsequently, the data item object—named ListItem—has two properties—Text and Value. (Naming is arbitrary, though.)

In a brand new data-bound control, you define a data item class that will be filled with any necessary information contained in the bound data source. This data item object must be persisted through the view state to guarantee that the control refreshes properly across postbacks. For performance reasons, the data item class must be able to serialize itself to the view state without resorting to the binary formatter. Put another way, it means that the data item class must implement IStateManager just like style classes do.
The data item class will be a collection of single data item classes if the data binding involves the association of a list of elements to a control.

Extra View-State Management

A custom data-bound control that supports data items must also provide extra view-state management code to ensure that the data item class is saved and restored. The data-bound control must override its `LoadViewState` and `SaveViewState` methods to persist any useful objects that implement `IStateManager`, such as styles and data items.

Note that in the `SimpleGaugeBar` control (discussed in Chapter 13), we haven’t overridden `LoadViewState` and `SaveViewState` methods for the custom style object. That was an arguable implementation choice. To stay on the safe side of development, you must override `LoadViewState` and `SaveViewState` whenever you have a custom control that works with `IStateManager` objects. If you omit this, as we did in Chapter 13, the risk is that any dynamically modified style attributes (or attributes on `IStateManager` objects) are lost across postbacks. Consider the following code snippet:

```csharp
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
        SimpleGaugeBar1 TextStyle.BackColor = Color.Yellow;
}
```

The `BackColor` attribute is set only the first time the page displays. If the `TextStyle` property is not explicitly persisted to the view state (as in the `SimpleGaugeBar` control of Chapter 13), the color yellow will be irreversibly lost the first time the page posts back.

```csharp
protected void Page_Load(object sender, EventArgs e)
{
    // The style is dynamically changed any time the page
    // is invoked. In this case, there's no need to persist the style
    // object to the viewstate.
    SimpleGaugeBar1 TextStyle.BackColor = Color.Yellow;
}
```

If you can assume that users of your control will always set styles at design time or don’t check the `IsPostBack` property, you should save extra view-state management for styles as a safeguard.
Overriding the PerformDataBinding Method

The final key feature for a custom data-bound control—that is, a control that inherits from DataBoundControl—is overriding the PerformDataBinding method. The method receives the contents of the bound data source object in the form of an enumerable object. As a control developer, you must read any required data from the source and cache it in the data item object.

Finally, you modify the rendering engine of the control to display bound data.

**Note**

Unless you need a data-bound control that behaves in a particular way (for example, a list control or a composite data-bound control), deriving your control from DataBoundControl is the most reasonable thing to do most of the time. If you need to start from a lower level, though, you can inherit from BaseDataBoundControl and override PerformSelect and ValidateDataSource. Needless to say, you might want to take this route only if you need to change the way a data source is validated and/or retrieved.

The GaugeBar Control

Let’s apply all the steps outlined so far to a new version of the SimpleGaugeBar control, aptly named GaugeBar. The new control will still be a composite control, but it will inherit from DataBoundControl to gain standard data-binding capabilities.

```csharp
public class GaugeBar : DataBoundControl
{
    //
    //
    //
}
```

To be precise, ASP.NET 2.0 features a class that incorporates both composition and data binding—CompositeDataBoundControl. We’ll tackle that in the next example.
Mapping Data Source Fields to Control Properties

The new **GaugeBar** control uses the same code of **SimpleGaugeBar** that we carefully examined in Chapter 13 and extends it in the way we discuss here. You need to have a string property for each bindable property on the control’s programming interface. The **GaugeBar** control has two bindable properties—**Value** and **FormatString**. This choice of bindable properties is arbitrary. You define a pair of **DataXxxField** properties—one for **Value** and one for **FormatString**. These string properties contain the name of the data source fields mapped to the **Value** and **FormatString**. In particular, **DataValueField** indicates that the field mapped to **Value** and **DataTextField** specifies the field linked to **FormatString**. Once again, note that the names used here are arbitrary.

```csharp
public virtual string DataValueField
{
    get
    {
        object o = ViewState["DataValueField"];
        if (o == null)
            return String.Empty;
        return (string) o;
    }
    set { ViewState["DataValueField"] = value; }
}

public virtual string DataTextField
{
    get
    {
        object o = ViewState["DataTextField"];
        if (o == null)
            return String.Empty;
        return (string) o;
    }
    set { ViewState["DataTextField"] = value; }
}
```

As you can see, both properties use the **ViewState** as the storage medium and are set to the empty string by default. Other popular data-bound properties available on the **GaugeBar** class are **DataSource**, **DataSourceID**, and **DataMember**, all of which are inherited from parent classes.

The **GaugeBar** Data Item Object

Once the **GaugeBar** control is bound to some external data, you need to track and cache any bound data. For this purpose, you need a data item object. As mentioned, a data item object is a custom class with as many public properties as there are bindable properties in the control’s interface. The data item class for the **GaugeBar** control is named **GaugeBarDataItem** (again, an arbitrary name) and is defined as follows:
public class GaugeBarDataItem : IStateManager
{
    private string _text;
    private float _value;
    private bool _marked;

    public GaugeBarDataItem()
    {
    }
    public GaugeBarDataItem(float value, string text)
    {
        _text = text;
        _value = value;
    }
    public string Text
    {
        get { return _text; }
        set { _text = value; }
    }
    public float Value
    {
        get { return _value; }
        set { _value = value; }
    }
    public bool IsTrackingViewState
    {
        get { return _marked; }
    }
    public void LoadViewState(object state)
    {
        if (state != null)
        {
            Pair p = (Pair)state;
            _value = (float)p.First;
            _text = (string)p.Second;
        }
    }
    public object SaveViewState()
    {
        return new Pair(_value, _text);
    }
    public void TrackViewState()
    {
        _marked = true;
    }
}

The class has two public properties—Text and Value—persisted through local members, and it implements the IStateManager interface to save its contents to the view state across postbacks.

The SaveViewState method returns a Pair object (a sort of simplified array of two elements) filled with the current values of Text and Value properties. The Pair object returned by SaveViewState becomes the input argument of LoadViewState, which unpacks the Pair object and initializes the Text and Value properties.
The **GaugeBar** control will need to expose a read-only property of type **GaugeBarDataItem**. You can use any name for this variable—I’m using **DataItem** here. The name of the property is not as important as its implementation. Take a look at the following code:

```csharp
private GaugeBarDataItem _dataItem;
.
.
private GaugeBarDataItem DataItem
{
    get
    {
        if (_dataItem == null)
        {
            _dataItem = new GaugeBarDataItem();
            if (base.IsTrackingViewState)
                _dataItem.TrackViewState();
        }
        return _dataItem;
    }
}
```

Unlike other control properties that are persisted directly in the **ViewState** collection object, the **DataItem** property uses a private member (\_dataItem) to persist its value. A private member, though, is not persistent and doesn’t survive postbacks. For this reason, in the **get** accessor of the property you need to check \_dataItem for nullness and create a new instance if it is null.

The code contained in the **get** accessor of a property runs whenever that property is invoked. As we’ll see in a moment, the preceding code ensures that no access to **DataItem** results in a null object exception, and that the state of the object is restored correctly after each postback.

**Data Item and View State**

Most of the control properties we’ve considered thus far use the **ViewState** container to persist the values. Why should we not store **DataItem** or style properties in the same way? Is there anything wrong with the following code?

```csharp
// NB: for this code to work, GaugeBarDataItem must be
// a serializable type
public virtual GaugeBarDataItem DataItem
{
    get
    {
        object o = ViewState["DataItem"]; // NB: for this code to work, GaugeBarDataItem must be
        if (o == null)
            return new GaugeBarDataItem(); // a serializable type
        return (string) o;
    }
}
Actually, nothing is “wrong” with the code per se—but consider for a moment view-state size and performance. Saving a class type directly in the ViewState container results in the object being serialized using the binary formatter. The BinaryFormatter class—the standard way to serialize managed objects in .NET applications—is not particularly fast and is designed to save the entire state of the object, including both public and private members, both simple and complex. The use of the BinaryFormatter increases the response time for each request and generates a larger view-state output. By customizing the view-state serialization, you obtain much faster code and save exactly the information you need to save.

As a rule of thumb, you should use the ViewState container to store property values if the type of the property is primitive—string, numbers, Boolean values, colors, dates, bytes, and arrays of any of these types. Reference types (for example, custom classes) should be serialized by implementing IStateManager and exposing the property via a get accessor like the one shown previously. As far as control development is concerned, this is commonly required for styles and data item properties.

**Ad Hoc View-State Management**

A control that has properties that take advantage of custom view-state serialization must override the SaveViewState and LoadViewState protected methods. These methods are defined on the Control class, and they indicate how to save and restore the state of the control to and from the view state. The default implementation of both methods takes care only of the contents of the ViewState container object.

```csharp
protected override object SaveViewState()
{
    // Get the standard state object—ViewState container
    object baseState = base.SaveViewState();
    // Get the state object for the DataItem property
    object itemState = DataItem.SaveViewState();
    // Get the state object for the TextStyle object
    object styleState = TextStyle.SaveViewState();
    // Pack everything into a unique object
    return new Triplet(baseState, itemState, styleState);
}
```

The SaveViewState method of the GaugeBar control needs to save three objects—the standard view state, the DataItem property, and the TextStyle property. You get the standard view-state output by calling SaveViewState on the base class, and other state objects by calling SaveViewState on the IStateManager implementation of DataItem and TextStyle. The SaveViewState method on the
control needs to return a single object, so you just group all data to return in a single object—typically an array or a combination of \(\text{Pair}\) and \(\text{Triplet}\) objects.

The object returned by \(\text{SaveViewState}\) is received by \(\text{LoadViewState}\), which extracts and assigns data back to the original objects.

```csharp
protected override void LoadViewState(object savedState)
{
    if (savedState != null)
    {
        Triplet t = (Triplet) savedState;
        base.LoadViewState(t.First);
        DataItem.LoadViewState(t.Second);
        TextStyle.LoadViewState(t.Third);
    }
    else
    {
        base.LoadViewState(null);
    }
}
```

The \(\text{IStateManager}\) implementation of \(\text{LoadViewState}\) on the serialized objects determines how each object (for example, styles or data items) restores its own data.

Note that when \(\text{DataItem.LoadViewState}\) is called, the \textit{get} accessor of \(\text{DataItem}\) is invoked and initializes the internal \_\textit{dataItem} member on the first call.

### Getting Bound Data

In ASP.NET 2.0, a bound control obtains bound data through the \(\text{PerformDataBinding}\) method. Overriding this method is mandatory for any data-bound control, as the standard implementation of the method does nothing. It is important to recall that the \(\text{IEnumerable}\) argument passed to \(\text{PerformDataBinding}\) represents the collection of bound data regardless of the format of the originally bound data source—whether it is an ADO.NET object, a collection, or a data source control.

Here’s the implementation of \(\text{PerformDataBinding}\) for the \(\text{GaugeBar}\) control:

```csharp
protected override void PerformDataBinding(IEnumerable data)
{
    // Argument data is a single object, not a list to enumerate.
    // Need to get an enumerator and call MoveNext once to get to data
    if (data == null)
    { return; }
    IEnumerator e = data.GetEnumerator();
    e.MoveNext();
```
    // Set default values for bindable properties
    float displayValue = 0;
    string displayText = String.Empty;

    // Read the value for the Value property
    if (!String.IsNullOrEmpty(DataValueField))
        displayValue = (float) DataBinder.GetPropertyValue(e.Current, DataValueField);

    // Read the value for the FormatString property
    if (!String.IsNullOrEmpty(DataTextField))
        displayText = (string) DataBinder.GetPropertyValue(e.Current, DataTextField);

    // Fill the DataItem property
    DataItem.Value = displayValue;
    DataItem.Text = displayText;
}

In this particular case, the IEnumerable object passed to PerformDataBinding contains just one element. The IEnumerable interface, though, doesn’t distinguish between a single element and a list of elements. In other words, to get the data object, you need to get the enumerator and move to the first item.

    // data is of type IEnumerable
    IEnumerator e = data.GetEnumerator();
    e.MoveNext();
    // Use e.Current to get the physical data object

The e.Current expression returns the data object bound to the control—that is, the container from which you extract the fields mapped to bindable properties. If you know the control is bound to, say, a DataRow object, you could retrieve the value for the Value property through the following code:

    displayValue = ((DataRow) e.Current)[DataValueField];

Using the DataBinder class adds greater flexibility to your code and makes your code independent from the type of the bound data source. The GetPropertyValue method on the DataBinder class uses reflection to query the object to see whether it contains a public property with the specified name:

    displayText = (string) DataBinder.GetPropertyValue(e.Current, DataTextField);

GetPropertyValue returns an object and requires a cast to the proper type.

The remaining step is updating the rendering engine so that it accesses the DataItem object whenever it requires bound data. The BuildLabel method shown next displays the descriptive text around the gauge. (See Chapter 13 for a full explanation of the logic behind this method.)
void BuildLabel(TableCell container)
{
    // Calculate the value to represent
    float buf = GetValueToRepresent();

    // Get the string to display on the label
    string msg = GetTextToRepresent();

    Label lbl = new Label();
    if (container is TableCell)
        container.Controls.Add(lbl);
    else
        Controls.Add(lbl);
    lbl.Text = String.Format(msg, buf, Maximum);
}

The BuildLabel method adds a Label control to the control hierarchy under construction. The text displayed through the label is composed using the value and the format string of the gauge. Both Value and FormatString can be either data-bound or statically assigned. For this reason, you should use a get function that checks the current binding, if any, and returns the bound value or the assigned value. Note that the bound value is returned in favor of an assigned value, if both are present.

float GetValueToRepresent()
{
    float f = 0;
    if (DataItem.Value >=0)
        f = DataItem.Value;
    else
        f = Value;
    return f;
}

string GetTextToRepresent()
{
    string msg = "";
    if (!String.IsNullOrEmpty(DataItem.Text))
        msg = DataItem.Text;
    else
        msg = FormatString;
    return msg;
}

No other changes are required to enhance the SimpleGaugeBar control of Chapter 13 and make it data-bound.

The following code shows the Load handler of a sample page that uses the GaugeBar control and binds it to a dynamically generated DataTable object:

protected void Page_Load(object sender, EventArgs e)
{
    // Uses a random number as the value of the GaugeBar.
    // The value is stored in a DataTable object
Random rnd = new Random();
DataTable dt = new DataTable();
dt.Columns.Add("Numbers", typeof(float));
dt.Columns.Add("Label", typeof(string));
DataRow row = dt.NewRow();
row[0] = rnd.Next(0, 100);
row[1] = "{0} out of {1}";
dt.Rows.Add(row);
dt.AcceptChanges();

// Binds the DataTable to the GaugeBar
GaugeBar1.DataValueField = "Numbers";
GaugeBar1.DataTextField = "Label";
GaugeBar1.DataSource = dt;
GaugeBar1.DataBind();

The DataTable has two columns—Numbers and Label—of type float and string, respectively. The table contains one data row. If the table contained multiple rows, only the first would be taken into account according to the code in PerformDataBinding.

Note that you can also use the DataItem property to bind data to the GaugeBar control.

GaugeBar1.DataItem.Value = 12;
GaugeBar1.DataItem.Text = "{0} %";

Note that no call to DataBind is required to trigger the process and update the control’s user interface.

Building a List Data-Bound Control

Data-bound controls are often list controls. A list control builds its own user interface by repeating a fixed template for each bound data item within the boundaries of the control’s mainframe. For example, a CheckBoxList control just repeats a CheckBox control for each bound data item. Likewise, a DropDownList control iterates through its data source and creates a new <option> element within a parent <select> tag.

In ASP.NET 2.0, all list controls inherit from ListControl—the only class in Figure 14-1 to be defined already in ASP.NET 1.x. ListControl adds quite a few new members to the interface of its parent class—DataBoundControl. In addition, list controls often also feature some layout capabilities such as vertical or horizontal rendering.
Generalities of List Controls

As shown in Figure 14-1, five frequently used controls inherit from `ListControl`. They are `DropDownList`, `CheckBoxList`, `RadioButtonList`, `ListBox`, and `BulletedList`. `BulletedList` exists only in ASP.NET 2.0. All these controls have a common programming interface that goes significantly beyond the programming interface of `DataBoundControl`. Let’s review the new members.

Additional Members of `ListControl`

Table 14-1 lists properties and methods that are specific to the `ListControl` class and all derived list controls.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>AppendDataBoundItems</code></td>
<td>Indicates whether the <code>Items</code> collection can be populated both programmatically and through data binding.</td>
</tr>
<tr>
<td><code>AutoPostBack</code></td>
<td>Indicates whether the control has to post back automatically when a change in the selection occurs.</td>
</tr>
<tr>
<td><code>CausesValidation</code></td>
<td>Indicates whether validation should occur before the control posts back.</td>
</tr>
<tr>
<td><code>DataTextField</code></td>
<td>Name of the data source field to use as the display text of each bound item.</td>
</tr>
<tr>
<td><code>DataTextFormatString</code></td>
<td>Format string for the display text.</td>
</tr>
<tr>
<td><code>DataValueField</code></td>
<td>Name of the data source field to use as the value of each bound item.</td>
</tr>
<tr>
<td><code>Items</code></td>
<td>Collection of bound items. Depending on the value of <code>AppendDataBoundItems</code>, the collection also can be filled programmatically. Elements in the <code>Items</code> collection are of type <code>ListItem</code>.</td>
</tr>
<tr>
<td><code>SelectedIndex</code></td>
<td>Returns the index of the currently selected item.</td>
</tr>
<tr>
<td><code>SelectedItem</code></td>
<td>Returns the <code>ListItem</code> object that corresponds to the currently selected item.</td>
</tr>
<tr>
<td><code>SelectedValue</code></td>
<td>Returns the value of the currently selected item.</td>
</tr>
<tr>
<td>Member</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Text</strong></td>
<td>String property that indicates the text of all list items.</td>
</tr>
<tr>
<td><strong>ValidationGroup</strong></td>
<td>Indicates the name of the validation group this control belongs to.</td>
</tr>
</tbody>
</table>

Some properties—such as AutoPostBack, CausesValidation and ValidationGroup—clearly have little to do with the data-binding process.

For all list controls, the data item type is ListItem. Because list controls display a list of items, instead of using a DataItem property (discussed in the previous section) that points to a single object, you need a collection object in this case. The Items property is just what you need, and it is of type ListItemCollection. When you bind a list control to its data, the Items collection gets filled, typically within the PerformDataBinding method. However, if AppendDataBoundItems returns true (the default value), you can add ListItem elements to Items programmatically.

### A Very Simple List Control

When it comes to building a custom list control, you have two possible routes to take. In the simplest one, you derive your class directly from ListControl. As we’ll see in a moment, the amount of code to write is minimal, but there are a few limitations to be aware of. Let’s suppose we need to create a list of hyperlinks. The control to repeat is the HyperLink control, and we want to specify both the URL and the text of each link through bindable properties.

```csharp
public class SimpleHyperLinkList : ListControl
{
    private HyperLink _controlToRepeat;
    private HyperLink ControlToRepeat
    {
        get
        {
            if (_controlToRepeat == null)
                _controlToRepeat = new HyperLink();
            return _controlToRepeat;
        }
    }

    protected override void Render(HtmlTextWriter writer)
    {
        for (int i = 0; i < Items.Count; i++)
        {
            HyperLink ctl = ControlToRepeat;
            ctl.ApplyStyle(ControlStyle);
            ctl.Text = Items[i].Text;
```
The `SimpleHyperLinkList` control revolves around the private `ControlToRepeat` property and the `Render` method. `ControlToRepeat` indicates the control tree that will be repeated for each bound item. You instantiate it the first time the control is used in the request and reuse it all the way through.

The `Render` method is based on a loop that renders out a `HyperLink` control for each bound item. The `Text` and `NavigateUrl` properties are set with bound fields. `RenderControl` generates the markup for the `HyperLink` control; the markup is accumulated in the text writer object and output as shown in Figure 14-2.

What’s wrong with this custom list control? Functionally speaking, there’s nothing wrong, but there are some limitations of which you should be aware. Deriving your control from `ListControl` forces you to adopt `ListItem` as the data item object (you can’t build your own) and `ListItemCollection` as the data item collection type. The names of the existing mapping properties cannot be changed, though new mapping properties can be added.
To further illustrate data-bound list controls, let’s write an enhanced version of the SimpleHyperLinkList control that uses a made-to-measure data item collection and a bunch of custom mapping properties.

**The HyperLinkList Control**

The expected behavior of the new HyperLinkList control is nearly the same as the behavior we obtained by inheriting a control from ListControl. The new class declaration is as follows:

```csharp
public class HyperLinkList : DataBoundControl, IRepeatInfoUser
{
    .
    .
    .
}
```

The methods of the IRepeatInfoUser interface serve to express the layout capabilities of the new control and render the hyperlinks on a fixed number of rows and columns. We’ll return to this topic later.

**Mapping Data Source Fields to Control Properties**

The HyperLinkList control has the following three field-mapping properties: DataTextField, DataTooltipField, and DataUrlField.

```csharp
public virtual string DataTextField
{
    get
    {
        object o = ViewState["DataTextField"];
        if (o == null)
            return String.Empty;
        return (string) o;
    }
    set { ViewState["DataTextField"] = value; }
}
```

```csharp
public virtual string DataTooltipField
{
    get
    {
        object o = ViewState["DataTooltipField"];
        if (o == null)
            return String.Empty;
        return (string) o;
    }
    set { ViewState["DataTooltipField"] = value; }
}
```
The properties map three key attributes of a hyperlink to fields on a data source object—display text, ToolTip and, of course, the URL. The choice of these properties affects the structure of the data item object.

### The HyperLinkList Data Item Object

The data item object is a class that represents a bound item. In this case, it will be the instance of a class that contains link information—URL, text, and ToolTip.

```csharp
public class HyperLinkItem
{
    private string _text;
    private string _url;
    private string _tooltip;

    public HyperLinkItem()
    {
    }
    public HyperLinkItem(string url, string text, string tooltip)
    {
        _text = text;
        _url = url;
        _tooltip = tooltip;
    }
    public string Text
    {
        get { return _text; }
        set { _text = value; }
    }
    public string Tooltip
    {
        get { return _tooltip; }
        set { _tooltip = value; }
    }
    public string Url
    {
        get { return _url; }
        set { _url = value; }
    }
}
```
The class defines three public members—`Url`, `Tooltip`, and `Text`—stored in local variables and a couple of constructors. The `HyperLinkItem` class represents the `n`th bound item; to represent the whole data set bound to a list control, you need a collection of `HyperLinkItem` objects.

To build a collection in ASP.NET 2.0, you use generics. The use of generics normally reduces the amount of necessary collection code you have to write to nearly zero. However, in this case, you also need to provide an implementation for the `IStateManager` interface.

class HyperLinkItemCollection : Collection<HyperLinkItem>, IStateManager
{
    private bool _marked;

    public HyperLinkItemCollection()
    {
        _marked = false;
    }

    public bool IsTrackingViewState
    {
        get { return _marked; }
    }

    public void TrackViewState()
    {
        _marked = true;
    }

    public void LoadViewState(object state)
    {
        if (state != null)
        {
            Triplet t = (Triplet) state;

            // This is required to ensure that the collection is empty
            Clear();

            string[] rgUrl = (string[])t.First;
            string[] rgText = (string[])t.Second;
            string[] rgTooltip = (string[])t.Third;

            for (int i = 0; i < rgUrl.Length; i++)
                Add(new HyperLinkItem(rgUrl[i], rgText[i], rgTooltip[i]));
        }
    }

    public object SaveViewState()
    {
        int numOfItems = Count;
        object[] rgTooltip = new string[numOfItems];
        object[] rgText = new string[numOfItems];
        object[] rgUrl = new string[numOfItems];

        return new Triplet(rgUrl, rgText, rgTooltip);
    }
}
for (int i = 0; i < numOfItems; i++)
{
    rgTooltip[i] = this[i].Tooltip;
    rgText[i] = this[i].Text;
    rgUrl[i] = this[i].Url;
}

return new Triplet(rgUrl, rgText, rgTooltip);

The real data item public property that matches the DataItem property of the GaugeBar control is a collection named Items. (The name is arbitrary.)

private HyperLinkItemCollection _items;
.
.
public virtual HyperLinkItemCollection Items
{
    get
    {
        if (_items == null)
        {
            _items = new HyperLinkItemCollection();
            if (base.IsTrackingViewState)
                _items.Track ViewState();
        }
        return _items;
    }
}

It is now interesting to take a look at the code of SaveViewState. The object returned by this method must contain a URL, text, and a ToolTip for each bound item. I opted for a triplet of arrays. Each array contains all URLs, text, and ToolTips for items listed by the control. Once filled, each array is added to a Triplet and returned. LoadViewState unpacks this information and refills the collection.

In this way, HyperLinkItemCollection is endowed with the capabilities of saving to, and restoring from, the view state. This code alone is not sufficient, though, to persist the collection through the view state. You also need, in fact, ad hoc view-state management code:

protected override object SaveViewState()
{
    object baseState = base.SaveViewState();
    object itemState = Items.SaveViewState();
    return new Pair(baseState, itemState);
}

protected override void LoadViewState(object savedState)
{
    if (savedState != null)
    {
        Pair p = (Pair) savedState;
        base.LoadViewState(p.First);
        Items.LoadViewState(p.Second);
    }
The `SaveViewState` and `LoadViewState` methods on the `HyperLinkList` control are invoked when the control is going to save itself to the view state and restore from it. The call to the same pair of methods on the `Items` property ensures that the collection is persisted and restored correctly.

Getting Bound Data

A data-bound control receives its data within the `PerformDataBinding` method. The data bound control is a collection that represents the data source. The goal of the `PerformDataBinding` method is to extract information from the data source and cache it in an intermediate data structure that is more lightweight for persisting across postbacks.

```csharp
protected override void PerformDataBinding(IEnumerable dataSource)
{
    base.PerformDataBinding(dataSource);

    string urlField = DataUrlField;
    string textField = DataTextField;
    string tooltipField = DataTooltipField;

    if (dataSource != null)
    {
        foreach (object o in dataSource)
        {
            HyperLinkItem item = new HyperLinkItem();
            item.Url = DataBinder.GetPropertyValue(o, urlField, null);
            item.Text = DataBinder.GetPropertyValue(o, textField, null);
            item.Tooltip = DataBinder.GetPropertyValue(o, tooltipField, null);
            Items.Add(item);
        }
    }
}
```

As you can see, for each item in the bound data source, the code creates a new `HyperLinkItem` object, fills it with the value of mapped fields, and finally adds it to the `Items` collection.

The `IRepeatInfoUser` Interface

So far the `HyperLinkList` control is not much different from its `SimpleHyperLinkList` predecessor, which we created in the previous section, except for a slightly different set of members. Let’s add a brand new feature—the ability to render items in a given direction and within a fixed row/column scheme. If you ever used the ASP.NET built-in `CheckBoxList` control, you perhaps know about the `RepeatDirection`, `RepeatLayout`, and `RepeatColumns` properties. When set, these properties
determine the way the output of the list control is laid out. Table 14-2 details the role of each property.

Table 14-2. IRepeatInfoUser Specific Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RepeatColumns</td>
<td>Gets or sets the number of columns to display in the control.</td>
</tr>
<tr>
<td>RepeatDirection</td>
<td>Gets or sets a value that indicates whether the control displays vertically or horizontally.</td>
</tr>
<tr>
<td>RepeatLayout</td>
<td>Gets or sets the layout of the check boxes (table or flow).</td>
</tr>
</tbody>
</table>

By using a combination of these properties, a page author can decide the number of rows and columns that must be used to render the hyperlinks. Let’s see how to code this feature.

The first step consists of adding public properties to HyperLinkList to let page developers specify the desired layout.

```csharp
public virtual RepeatDirection RepeatDirection
{
    get
    {
        object o = ViewState["RepeatDirection"];
        if (o != null)
            return (RepeatDirection) o;
        return RepeatDirection.Vertical;
    }
    set { ViewState["RepeatDirection"] = value; }
}

public virtual int RepeatColumns
{
    get
    {
        object o = ViewState["RepeatColumns"];
        if (o != null)
            return (int) o;
        return 0;
    }
    set { ViewState["RepeatColumns"] = value; }
}

public virtual RepeatLayout RepeatLayout
{
    get
    {
        object o = ViewState["RepeatLayout"];
        if (o != null)
```
A control that intends to support special layout capabilities needs a special code fragment in the `Render` method. Here’s an example:

```csharp
protected override void Render(HtmlTextWriter writer)
{
    if (Items.Count > 0)
    {
        RepeatInfo ri = new RepeatInfo();
        Style controlStyle = (base.ControlStyleCreated
            ? base.ControlStyle : null);
        ri.RepeatColumns = RepeatColumns;
        ri.RepeatDirection = RepeatDirection;
        ri.RepeatLayout = RepeatLayout;
        ri.RenderRepeater(writer, this, controlStyle, this);
    }
}
```

The `RepeatInfo` structure exists since ASP.NET 1.x just to allow controls to output their contents according to different rendering rules. The second parameter of the `RenderRepeater` method references the object (`this`, in the preceding code snippet) that implements the `IRepeatInfoUser` interface. The interface implementation is required for the feature to work on a given control. The interface members are presented in **Table 14-3**.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HasFooter</td>
<td>Indicates whether the control has a footer</td>
</tr>
<tr>
<td>HasHeader</td>
<td>Indicates whether the control has a header</td>
</tr>
<tr>
<td>HasSeparators</td>
<td>Indicates whether the control has a separator between items</td>
</tr>
<tr>
<td>RepeatedItemCount</td>
<td>Indicates the number of items to render</td>
</tr>
</tbody>
</table>

In addition, the interface has a couple of methods—`GetItemStyle` and `RenderItem`. The former returns the style object for each rendered item. The latter generates the markup. The following code snippet shows how we might implement this interface for our hyperlink control:
bool IRepeatInfoUser.HasFooter
{
    get { return false; }
}

bool IRepeatInfoUser.HasHeader
{
    get { return false; }
}

bool IRepeatInfoUser.HasSeparators
{
    get { return false; }
}

int IRepeatInfoUser.RepeatedItemCount
{
    get { return this.Items.Count; }
}

Style IRepeatInfoUser.GetItemStyle(ListItemType itemType, int repeatIndex)
{
    return null;
}

void IRepeatInfoUser.RenderItem(ListItemType itemType,
int repeatIndex, RepeatInfo repeatInfo, HtmlTextWriter writer)
{
    HyperLink ctl = ControlToRepeat;
    int i = repeatIndex;
    ctl.ID = i.ToString();
    ctl.Text = Items[i].Text;
    ctl.NavigateUrl = Items[i].Url;
    ctl.ToolTip = Items[i].Tooltip;
    ctl.RenderControl(writer);
}

The **RenderItem** method is automatically invoked for each bound item—precisely once for each item in the **Items** collection—and is expected to render the output for the control to repeat. The control to repeat can be an instance of an individual control (for example, as in this case, a **HyperLink**) as well as the root of a control tree. The public properties of the control being repeated can be set using bound data. The control is then rendered out using the **RenderControl** method. **Figure 14-3** shows the control in action.
Building a Composite Templated Control

The *CompositeDataBoundControl* class is the starting point for building rich, complex, and data-bound composite controls. A composite data-bound control must do the following:

- Act as a naming container.
- Create its own user interface through the `CreateChildControls` method.
- Implement the necessary logic to restore its hierarchy of child elements after postback.

The last point is a subtle one that many developers cut their teeth on in ASP.NET 1.x. If you don’t fully understand the third point yet, the good news is that you can now forget about it entirely. Everything that is needed is now hard-coded in the *CompositeDataBoundControl* class.

Generalities of Composite Data-Bound Controls

In ASP.NET 2.0, the main aspect you care about when building a composite data-bound control is designing the internal hierarchy of your control. The method to override for this purpose is an overloaded version of `CreateChildControls`. In addition, you typically add styles and templates.
In a real-world composite control, the internal control tree is usually quite complex. The outermost container is often a multirow HTML table, but what’s in the various cells and rows can vary quite a bit and result in a pretty sophisticated combination of child controls and literals.

Creating a Hierarchy of Child Controls

You should know by now that composite controls build their own interface by composing controls in the override of the `CreateChildControls` method. Defined on the `Control` class, the method has the following prototype:

```csharp
protected override void CreateChildControls()
```

In the `CompositeDataBoundControl` class, the method is overridden and overloaded. In particular, the overridden version accomplishes a few interesting tasks. Here's its pseudo-code:

```csharp
protected override void CreateChildControls()
{
    Controls.Clear();
    object o = ViewState["_!ItemCount"];
    if ((o == null) && RequiresDataBinding)
        EnsureDataBound();
    else
    {
        int numOfItems = (int) o;
        object[] rg = new object[numOfItems];
        CreateChildControls(rg, false);
        base.ClearChildViewState();
    }
}
```

The method first empties the `Controls` collection so that no pending child controls are left around. Next, it retrieves a value from a particular (and internally managed) view-state entry named `_!ItemCount`. The view-state entry caches the number of items that form the composite control. The code that actually builds the control tree is responsible for storing this value in the view state.

It is important to know the number of items that form the control hierarchy if you want to optimize the data-binding process. In ASP.NET, complex controls showing a long list of data items are implemented as composite data-bound controls. In what way is this different from list and simple-bound controls?

List controls and simple-bound controls like the `GaugeBar` we considered earlier cache the data item or items in the view state. In addition, they can receive data either from the data-binding process or programmatically through the `Items` collection and the `DataItem` property, respectively. Composite data-bound controls (such as `DataGrid`, `DataList`, and `GridView`) work on the
assumption that they receive data exclusively from data binding and, for this reason, don’t persist bound data in any form. Consider now the following scenario.

Imagine a page that contains a rich control such as the DataView and some button controls. One of the button controls, when clicked, executes no code that involves the DataView but still refreshes the page. Without some special tricks in the control’s code, you can be sure that the composite data-bound control would be empty upon postback. Why is it so? If the postback event handler doesn’t bind data back to the composite control, the control has no way to figure it out and refresh properly. In ASP.NET, by design, composite data-bound controls take their data only from data binding and don’t cache any bound data. Therefore, a special workaround is required to handle postback events.

For composite data-bound controls, the CreateChildControls method works in either of two modes—binding or nonbinding. When working in binding mode, the control tree is created as usual. When working in nonbinding mode, the control calls an overloaded version of CreateChildControls. The method is defined as abstract on the CompositeDataBoundControl and must be overridden in any derived class.

The Overloaded CreateChildControls

The overloaded version of CreateChildControls that is defined on the CompositeDataBoundControl class is shown here:

```csharp
protected abstract int CreateChildControls(
    IEnumerable dataSource, bool dataBinding);
```

The first parameter is the collection of bound data. The second parameter indicates whether the control is being bound to fresh data (that is, it is working in binding mode) or is being refreshed after a postback. The return value indicates the number of items added to the control tree. This value will then be stored in the view state during the call to PerformDataBinding. The following code snippet shows the pseudo-code of PerformDataBinding on the CompositeDataBoundControl class:

```csharp
protected internal override void PerformDataBinding(IEnumerable data)
{
    base_PerformDataBinding(data);
    Controls.Clear();
    base.ClearChildViewState();
    TrackViewState();
    int numOfItems = CreateChildControls(data, true);
    base.ChildControlsCreated = true;
    ViewState["_!ItemCount"] = numOfItems;
}
```

Note that PerformDataBinding calls into the new overload of CreateChildControls and passes it true as the second argument, indicating that a binding operation is taking place. This makes sense,
because executing `PerformDataBinding`, by definition, means we are performing a binding operation.

What kind of code should you place in the overloaded `CreateChildControls`? Basically, you’d choose your own control builder method (typically, `CreateControlHierarchy`) and return its return value. We’ll return to this point later when discussing the sample `BarChart` control.

The overloaded `CreateChildControls` method is invoked in binding mode from within `PerformDataBinding`, and it’s invoked in nonbinding mode from within the other `CreateChildControls` method:

```csharp
// o is the value read from ViewState
int numOfItems = (int) o;
object[] rg = new object[numOfItems];
CreateChildControls(rg, false);
```

In this case, the bound data passed to the method is an empty array of objects of a well-known size. The goal of this array is to force the control builder method (typically, `CreateControlHierarchy`) to loop the right number of times and build an outermost container with the right configuration—for example, a table with the right number of rows and columns. As we’ll see in detail for the sample `BarChart` control, a composite data-bound control neatly separates hierarchy from data. If the Boolean parameter of `CreateChildControls is false`, no data is added to the hierarchy. How can the control show up as it did the last time? The ASP.NET postback mechanism guarantees that child controls are restored with all their values. In other words, if a composite data-bound control displays bound data through, say, a `Label` control, after a postback, the composite control doesn’t restore its bound data directly. However, it asks any child control, including the `Label`, to restore itself from the view state. In doing so, the `Label` restores the bound data from its `Text` property.

The bottom line is that the amount of extra data that flows in the view state for a composite control is limited to the number of constituent items, and the control refreshes correctly after a postback. (Of course, child controls put the usual amount of data in the view state.)

**The Control Item**

It should be clear from the previous discussion that the ASP.NET team had excellent arguments to dictate that composite data-bound controls get their data exclusively from the data-binding process. This fact eliminates the need of having a kind of `Items` property on composite data-bound controls that works like the `Items` property of list controls. This said, feel free to add support for data item objects and collections to your composite controls if you need to.

Most composite controls feature a collection of items, but not a collection of data items. Each item represents a control item—that is, a logical building block of the control’s user interface. For a
DataGrid, it is a DataGridItem object that represents a table row. For a sample BarChart control that displays a bar chart, the control item will be a class derived from TableRow that contains all the information needed to handle a single bar. The number of items that composite controls store in the view state is exactly the number of “control” items.

Let’s see how these concepts apply to a sample composite data-bound control such as BarChart.

**The BarChart Control**

The BarChart control inherits from CompositeDataBoundControl and defines the properties in Table 14-4.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataTextField</td>
<td>Name of the data field to use as the label of each bar.</td>
</tr>
<tr>
<td>DataTextFormatString</td>
<td>Format string for the display text.</td>
</tr>
<tr>
<td>DataValueField</td>
<td>Name of the data field to use as the value of each bar.</td>
</tr>
<tr>
<td>DataValueFormatString</td>
<td>Format string for the value to display on top of each bar.</td>
</tr>
<tr>
<td>Items</td>
<td>Collection of BarChart items. Each element represents a bar in the chart.</td>
</tr>
<tr>
<td>Maximum</td>
<td>Gets and sets the maximum value that can be represented in the chart.</td>
</tr>
<tr>
<td>SubTitle</td>
<td>Gets and sets the subtitle of the final chart.</td>
</tr>
<tr>
<td>Title</td>
<td>Gets and sets the title of the bar chart.</td>
</tr>
</tbody>
</table>

The final markup for the control is a horizontal bar chart such as the one illustrated in Figure 14-4.
Figure 14-4. The BarChart control in action.

Each bar is fully represented by an element in the Items collection. In addition, the BarChart control features a few style properties, as Table 14-5 details.

Table 14-5. BarChart Style Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BarStyle</td>
<td>The style of the whole row that contains the bar</td>
</tr>
<tr>
<td>LabelStyle</td>
<td>The style of the label</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------------------------------------------------------</td>
</tr>
<tr>
<td>SubTitleStyle</td>
<td>The style of the subtitle in the control’s header</td>
</tr>
<tr>
<td>TitleStyle</td>
<td>The style of the title in the control’s header</td>
</tr>
<tr>
<td>ValueStyle</td>
<td>The style of the element displaying the value rendered</td>
</tr>
</tbody>
</table>

The attributes of all style properties are applied in the `Render` method, as in other data-bound controls.

**The BarChart Item Object**

The user interface of the BarChart control is created in the overloaded version of the `CreateChildControls`.

```csharp
protected override int CreateChildControls(
    IEnumerable dataSource, bool dataBinding)
{
    return CreateControlHierarchy(dataSource, dataBinding);
}
```

Both input arguments are passed down to an internal `CreateControlHierarchy` method, which is ultimately responsible for the creation of the bar chart.

```csharp
int CreateControlHierarchy(IEnumerable dataSource, bool dataBinding)
{
    // Get the data to display (either from data source or viewstate)
    if (dataSource == null)
    {
        RenderEmptyControl();
        return 0;
    }

    // Start building the hierarchy of controls
    Table t = new Table();
    Controls.Add(t);

    // Add the header row with the caption
    CreateTitle(t);

    // Add the subtitle row
    CreateSubTitle(t);

    // Add bars
    int totalItems = CreateAllItems(t, dataSource, dataBinding);
```
return totalItems;
}

The control hierarchy is a table with two rows for the title and subtitle and other rows for the bars of the chart. CreateAllItems adds bar chart items and counts their number. This number is then returned and ends up in the view state.

```csharp
int CreateAllItems(Table t, IEnumerable data, bool useDataSource)
{
    // Count how many items we add
    int itemCount = 0;

    // Clears the Items collection (creates it, if null)
    Items.Clear();

    // Scroll data items and create table items
    foreach (object o in data)
    {
        // Create the match item object
        BarChartItemType itemType = BarChartItemType.Item;
        BarChartItem item = CreateBarChartItem(t,
            itemType, o, useDataSource);

        // Add the newly created object to the Items collection
        _items.Add(item);

        // Increase the counter
        itemCount++;
    }

    // Return how many items we have into the viewstate (for postbacks)
    return itemCount;
}
```

For each bound item, the method creates a BarChartItem object and adds it to the Items collection. We’ll discuss the BarChartItem class in a moment.

Note that we use Items.Clear to clear the collection and _items.Add to add a new bar chart item to the collection. The Items property is implemented as follows:

```csharp
private BarChartItemCollection _items;
.
.
public virtual BarChartItemCollection Items
{
    get
    {
        if (_items == null)
            _items = new BarChartItemCollection();

        return _items;
    }
}
```
The property `Items` uses the `_items` variable as its storage medium. The first call to `Items.Clear` ensures that the collection is properly initialized. The second call to the same collection can go through the local variable to save a call to the `get` accessor of the `Items` property.

The `BarChartItem` class represents a bar in the chart and is defined as follows:

```csharp
public class BarChartItem : TableRow
{
    private object _dataItem;
    private BarChartItemType _itemType;

    public BarChartItem(BarChartItemType itemType)
    {
        _itemType = itemType;
    }

    public object DataItem
    {
        get {return _dataItem;}
        set {_dataItem = value;}
    }

    public BarChartItemType ItemType
    {
        get {return _itemType;}
    }
}
```

The class inherits from `TableRow` (actually, a bar in the chart is a table row) and defines a couple of properties—`DataItem` and `ItemType`. The `DataItem` property references the data item in the bound data source associated with the corresponding item. For example, if the `BarChart` is bound to a `DataTable`, `DataItem` is bound to the `DataRow` that corresponds to a given bar. `ItemType`, on the other hand, indicates the type of table row—title, subtitle, item. The item types are defined through an enumerated type:

```csharp
public enum BarChartItemType
{
    Title,
    SubTitle,
    Item
}
```

The `Items` property groups a bunch of `BarChartItem` objects in a collection. The collection type is `BarChartItemCollection`:

```csharp
public class BarChartItemCollection : Collection<BarChartItem>
{
}
```
Because bar chart item objects don’t go to the view state, there’s no need to implement IStateManager and add extra view-state management methods as we did previously for the hyperlink control.

Adding Bound Data

With a composite data-bound control, you don’t need to override the PerformDataBinding method. However, you should pay some attention to keeping the code that builds the structure of the control and the code that adds data neatly separated.

The CreateBarItem method creates a new table row and enriches it with a DataItem property. What’s the content of the row? Looking at Figure 14-4, you can see that each table row has a cell for the label and a cell for the progress bar.

```csharp
BarChartItem CreateBarItem(Table t, BarChartItemType itemType, object dataItem, bool useDataSource)
{
    // Create a new row for the outermost table
    BarChartItem item = new BarChartItem(itemType);

    // Create cells for label and value
    TableCell labelCell = CreateLabelCell(item);
    TableCell valueCell = CreateValueCell(item);

    // Add the row to the table
    t.Rows.Add(item);

    // Handle the data object binding
    if (useDataSource)
    {
        // Get the data source object
        item.DataItem = dataItem;

        // Data bind the team labels
        BindLabelCell(labelCell, dataItem);
        BindValueCell(valueCell, dataItem);
    }

    // Return the fully configured row item
    return item;
}
```

CreateLabelCell and CreateValueCell add cells to the table row. Here is their implementation:

```csharp
private TableCell CreateLabelCell(BarChartItem item)
{
    // Create and add the cell
    TableCell cell = new TableCell();
    item.Cells.Add(cell);
    return cell;
}
```
The colored bar is represented with a label whose width is a percentage of the maximum value possible on the chart.

As you can see in the code of `CreateBarChartItem`, an `if` statement separates the creation of required child controls from the data binding. If the method is working in binding mode, the `DataItem` property is set on each bar chart item and the following two methods are called to add data to the child controls of the `BarChart` control:

```csharp
private void BindLabelCell(TableCell cell, object dataItem)
{
    if (!String.IsNullOrEmpty(DataTextField))
    {
        string txt = DataBinder.GetPropertyValue(
            dataItem, DataTextField, DataTextFormatString);
        cell.Text = txt;
    }
}

private void BindValueCell(TableCell cell, object dataItem)
{
    // Bind the label for the graph
    Label lblGraph = (Label) cell.Controls[0];
    object o = null;
    if (!String.IsNullOrEmpty(DataValueField))
    {
        o = DataBinder.GetPropertyValue(dataItem, DataValueField);
    }
    else
    {
        return;
    }
    float val = Convert.ToSingle(o);
    float valueToRepresent = 100 * val / Maximum;
    lblGraph.Width = Unit.Percentage(valueToRepresent);
    // Bind the label for the text
    Label lblText = (Label) cell.Controls[2];
    lblText.Text = DataBinder.GetPropertyValue(
        dataItem, DataValueField, DataValueFormatString);
}
```
The data-binding process works in a way that is no different from what we’ve seen earlier for other types of data-bound controls. The trickiest part here is the calculation of the width of the label that, when properly styled, generates the horizontal bar.

**Note**

As you can see, no style properties are assigned when the control hierarchy is being built. Just as for other data-bound controls, style attributes are applied later in the control life cycle in the *Render* method, immediately before generating the control’s markup.

### Events of the *BarChart* Control

The *BarChart* control also features a couple of events—*BarChartCreated* and *BarChartDataBound*. It is not coincidental that these two events mimic analogous events on the *DataGrid* control. Although far simpler, the *BarChart* is a control designed along the same guidelines that inspired the creation of the *DataGrid* control.

```csharp
public event EventHandler<BarChartItemEventArgs> BarChartItemCreated;
public event EventHandler<BarChartItemEventArgs> BarChartItemDataBound;
protected virtual void OnBarChartCreated(BarChartItemEventArgs e)
{
    if (BarChartItemCreated != null)
    BarChartItemCreated(this, e);
}
protected virtual void OnBarChartItemDataBound(BarChartItemEventArgs e)
{
    if (BarChartItemDataBound != null)
    BarChartItemDataBound(this, e);
}
```

The *BarChartItemCreated* event is fired whenever a new table row is added to represent a bar. The *BarChartItemDataBound* event fires when a newly added table row is bound to its data. The former event fires regardless of the working mode of the control. The latter fires only when the control is created in binding mode.

The data carried out with the event is grouped in the *BarChartItemEventArgs* class:

```csharp
public class BarChartItemEventArgs : EventArgs
{
    private BarChartItem _item;
    public BarChartItemEventArgs(BarChartItem item)
    {
        _item = item;
    }
    public BarChartItem Item
    {
        get { return _item; }
    }
    public event EventHandler<BarChartItemEventArgs> BarChartItemCreated;
    public event EventHandler<BarChartItemEventArgs> BarChartItemDataBound;
}
```
Both events are fired from within the `CreateBarChartItem` method.

```csharp
BarChartItem CreateBarChartItem(Table t, BarChartItemType itemType, object dataItem, bool useDataSource)
{
    // Create a new row for the outermost table
    BarChartItem item = new BarChartItem(itemType);

    // Create cells for label and value
    TableCell labelCell = CreateLabelCell(item);
    TableCell valueCell = CreateValueCell(item);

    BarChartItemEventArgs argsCreated = new BarChartItemEventArgs(item);
    OnBarChartItemCreated(argsCreated);

    if (useDataSource)
    {
        BarChartItemEventArgs argsData = new BarChartItemEventArgs(item);
        OnBarChartItemDataBound(argsData);
    }
}
```

**Using the `BarChart` Control**

Let’s see how to consume these events from within a host page. The following markup enables a `BarChart` control in an ASP.NET page:

```html
<exo:BarChart runat="server" id="BarChart1"
    Maximum="100" SubTitle="Subtitle" Title="Title"
    OnBarChartDataBound="BarChart1_BarChartDataBound">

    .
    .
    .
</exo:BarChart>
```

Nothing in the preceding markup indicates the data source. In the `Page_Load` event, the control is bound to its data—a SQL Server query that returns a resultset with two columns that show the
amount of sales for an employee in 1997 and the name of that employee. The source of the data is the Northwind database in SQL Server 2000.

protected void Button1_Click(object sender, EventArgs e)
{
    DataTable data = ExecuteQuery(1997);
    BarChart1.Maximum = 150000;
    BarChart1.Title = "Northwind Sales";
    BarChart1.SubTitle = "(Year 1997)";
    BarChart1.DataSource = data;
    BarChart1.DataTextField = "Employee";
    BarChart1.DataValueField = "Sales";
    BarChart1.DataBind();
}

The bar chart shown in Figure 14-4 is obtained by running the preceding code.

The sample page handles the BarChartDataBound event through the following code:

void BarChart1_BarChartDataBound(object sender, ProAspNet20.Advanced.BarChartItemEventArgs e)
{
    // Get the amount of sales for the current bar
    Decimal sales = (Decimal) DataBinder.GetPropertyValue(e.Item.DataItem, "sales");

    // Add a tooltip
    string tip = sales.ToString();
    e.Item.Attributes["title"] = tip;

    // Highlight bar where sales > 50000
    if (sales > 50000)
        e.Item.Cells[1].BackColor = Color.LightGreen;
}

The amount of sales for the current employee is retrieved and added to the row as a ToolTip. In addition, if the sales are larger than $50,000, the cell is highlighted by using a different background color. (See Figure 14-5.)
Figure 14-5. The output of a BarChart control modified by page-level event handlers.

Northwind Sales
(Year 1997)

Davoli $97,533.58
Fuller $74,958.60
Leverling $111,788.61
Peacock $139,477.70
Buchanan $32,995.05
Suyama $45,992.00
King $66,689.14
Callahan $59,776.52
Dodsworth $29,577.55

Note

All data-bound controls feature a couple of common events—DataBinding and DataBound. Also supported by ASP.NET 1.x, the former event fires before the data-binding process begins. The DataBound event has been introduced in ASP.NET 2.0 and signals that the data-binding phase has terminated.
Adding Templates Support

The *BarChart* control accepts two strings to display as the title and subtitle of the chart. Likewise, you could define a third similar property for the footer. Title, subtitle, and footer are just distinct items in the *BarChart* control hierarchy. What are you allowed to display in these items? As long as the properties are implemented as plain strings, there’s not much more than static text that can show up through the items.

A bit more flexibility can be added with format strings. A format string is a string that contains a predefined number of placeholders that the control machinery fills with internal data. For example, the *FormatString* property of the *GaugeBar* defaults to `{0}/{1}`—namely, a format string with two placeholders. The string is resolved as follows:

```csharp
// First placeholder gets the Value to represent
// Second placeholder gets the Maximum value that can be represented
String.Format(FormatString, Value, Maximum);
```

You can enrich the format string with HTML tags to obtain more appealing results but, in the long run, this approach would result in unmanageable code. A much better route to deep customizations of the user interface of controls would be to use templates.

Templates and User Controls

In ASP.NET, you can import templates in two ways—through properties of type *ITemplate* or by dynamically loading user controls. As we saw in Chapter 12, a Web user control is a custom component that can be used wherever a server control is valid. You can import such a user-defined control into the layout of the main control and make the interface more flexible and generic. You put a *PlaceHolder* control in the location in which you want custom contents to be injected, and then at run time, you create an instance of the user control and add it to the *Controls* collection of the placeholder.

```csharp
placeholder.Controls.Add(Page.LoadControl("usercontrol.ascx"));
```

The right time to call this code is early in the control life cycle—that is, in an *Init* event handler. Using the *LoadControl* method, the code of the template is insulated in a separate file. This can be a good thing or a bad thing, depending on the context. If the template you want to implement is complex, keeping it off the main page is positive. Otherwise, it would certainly add a layer of unnecessary complexity. Having the template directly available in the source code of the page makes authoring the page much more intuitive and fast because you don’t have to follow code into a separate file. There’s a sort of compromise between the two approaches. You could define an
ITemplate property in the control and leave the page author free to decide how to set it—with statically defined markup or using the contents of an .ascx file.

Defining a Template Property

A template property represents a collection of text and controls that is hosted within a container. The container is also responsible for exposing properties that page authors can use to create data-bound expressions. The following code snippet shows how to define a template property named TitleTemplate:

```csharp
[PersistenceMode(PersistenceMode.InnerProperty)]
[TemplateContainer(typeof(TitleTemplateContainer))]
public ITemplate TitleTemplate
{
    get { return _titleTemplate; }
    set { _titleTemplate = value; }
}
```

The storage of the template is guaranteed by the private member _titleTemplate, defined as follows:

```csharp
private ITemplate _titleTemplate = null;
```

A template property is characterized by a couple of attributes—PersistenceMode and TemplateContainer.

The PersistenceMode attribute indicates how a control property is persisted declaratively in a host page. Table 14-6 lists possible modes of persistence.

### Table 14-6. Persistence Modes for Control Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
<td>The property persists as an encoded HTML attribute in the final markup.</td>
</tr>
<tr>
<td>EncodedInnerTextProperty</td>
<td>The property persists as the only inner text of the control. The property value is HTML encoded. Only a string can be given this designation.</td>
</tr>
<tr>
<td>InnerInnerTextProperty</td>
<td>The property persists in the control as inner text and is the element’s default property. Only one property can be designated the default property.</td>
</tr>
<tr>
<td>InnerProperty</td>
<td>The property persists in the control as a nested tag. This is commonly used for complex objects with templates and styles.</td>
</tr>
</tbody>
</table>
### Property | Description
--- | ---

The most common setting is *InnerProperty*, which instructs Microsoft Visual Studio 2005 to save the contents of the template as a nested tag named after the property.

```xml
<expo:BarChart runat="server" ID="BarChart1" ... >
    <TitleTemplate>
        ...
        ...
    </TitleTemplate>
</expo:BarChart>
```

If you choose *InnerDefaultProperty*, you can have only one nested tag; by opting for *InnerProperty*, you can have as many nested tags as needed. This is good for rich controls with multiple templates and styles.

The *TemplateContainer* attribute declares the type of the naming container that will contain the template once it is created. As mentioned, a template is hosted by a container that, in turn, is appended to the control’s *Controls* collection. The *TemplateContainer* attribute references a type that, as a control developer, you’re responsible for declaring.

**Defining a Template Container**

A template container type is a simple Web control decorated with the *INamingContainer* interface. This control can be given any public members you like. However, it will typically expose the host control as a whole with a bunch of quick-access properties. Here’s a sample container type for the *TitleTemplate* property:

```csharp
public class TitleTemplateContainer : WebControl, INamingContainer
{
    private BarChart _parent;
    public TitleTemplateContainer(BarChart parent)
    {
        _parent = parent;
    }
    public string Title
    {
        get { return _parent.Title; }
    }
    public string SubTitle
    {
        get { return _parent.SubTitle; }
    }
}
```
Once again, it is important to note that there are no constraints or special guidelines to influence the set of members of the class. The class needs to have a reference to the parent control—the `BarChart` in this case. Normally, you create this class for a particular control (or set of controls) and don’t reuse it beyond that. It is up to you to expose the parent control through a direct property (`BarChart` in the preceding code) or to filter the control’s programming interface with a subset of properties (for example, `Title` and `SubTitle`). You can also do both things.

The programming interface of the template container class is important because it defines the information that page authors have access to when creating a template for the property. The template container is made accessible through the `Container` property.

**Setting a Template Property**

Figure 14-6 shows the templated version of the `BarChart` control in action in Visual Studio 2005. The `TitleTemplate` property shows up through Microsoft IntelliSense and greatly simplifies the assignment of a template property.

**Figure 14-6. Visual Studio 2005 recognizes and supports a template property.**

[View full size image]
You can use any combination of controls and literals to populate a template. To access external information, though, you will need to use data-bound expressions. Here’s an example:

```html
<TitleTemplate>
    <img src="Title.gif" /> 
    <%# Container.Title %>
</TitleTemplate>
```

The code snippet demonstrates a **BarChart** title that displays an image in addition to the text set through the **Title** property. Here’s another example:

```html
<TitleTemplate>
    <%# Container.Title %>
    <small>(<%# DateTime.Now.ToString() %>)</small>
</TitleTemplate>
```

**Figure 14-7** shows a templated title item where the originally set **Title** property is displayed side by side with the current time. The current time is rendered with a smaller font and within parentheses.
Figure 14-7. A *BarChart* control with a templated title.

Note that any style attributes set through the *TitleStyle* property are maintained in the template.

The *Container* keyword references an instance of the template container type. You use the *Container* keyword to access any control properties exposed through the template container class. Nonstatic information requires a `<%# ... %>` data-bound expression, just like in the templates of ASP.NET built-in controls.
Rendering a Template

So far we've seen how to define a template property in a server control. But what other changes to the code are required to host a template? In summary, to define a template property, you need to do the following:

- Define a property of type `ITemplate`, and use a private variable as its storage medium.
- Decorate the property with the `PersistenceMode` attribute.
- Define a template container class.
- Decorate the property with the `TemplateContainer` attribute.

These steps only define the public interface of the template; more is needed to embed the template in the control’s hierarchy. In particular, you need to tweak the code that creates the portion of the control tree where you want the template to display. For example, the `TitleTemplate` property refers to the title item, so the internal method to modify is `CreateTitle`. Here’s the updated version:

```csharp
private void CreateTitle(Table t)
{
    // Create the table row
    BarChartItem item = new BarChartItem(BarChartItemType.Title);
    t.Rows.Add(item);

    // Add the title cell
    TableCell cell = new TableCell();
    cell.ColumnSpan = BarChart.ColumnsCount;
    item.Cells.Add(cell);

    // Decide between plain string and template
    if (TitleTemplate != null)
    {
        _titleTemplateContainer = new TitleTemplateContainer(this);
        TitleTemplate.InstantiateIn(_titleTemplateContainer);
        cell.Controls.Add(_titleTemplateContainer);
    }
    else
    {
        cell.Text = Title;
    } // Must call DataBind to enable #-expression on templates
    item.DataBind();
}
```

You must check whether a template for the title item is defined; if it is not, you just set the `Text` property of the title cell with the contents of the `Title` property. Otherwise, you get an instance of the template container type and use it as the input argument of the `InstantiateIn` method—the only method on the `ITemplate` interface. When done, you add the template container to the control hierarchy—in this case, to the `Controls` collection of the title cell.

A fundamental further step is required to enable the template to successfully process data-bound expressions. You must place a call to `DataBind` on the title item. Data-bound expressions, in fact,
are evaluated only after a call to `DataBind` is made that involves the parent control that hosts the expression. Without the `DataBind` call, templates will work correctly but won’t display any `<% # ... %>` expression.

## How Data-Bound Controls Resolve Data Sources

Control developers using ASP.NET 1.x have an additional issue to solve. They need to figure out a way to normalize the data source to `IEnumerable`. Array, collection, and `DataView` objects implement `IEnumerable`, but the same isn’t true for popular ADO.NET objects such as `DataTable` and `DataSet`. These two classes, in fact, implement `IListSource`. Things get even more complicated if the control supports the `DataMember` property. This piece of code is required by virtually any data-bound control, including built-in ASP.NET controls. And, in fact, built-in ASP.NET controls use an internal method to do the job. Unfortunately, this method is defined on an internal class that is not accessible outside the `system.web` assembly. The class is named `DataSourceHelper` and is defined in the `System.Web.UI` namespace. The class features just one static method named `GetResolvedDataSource`.

```csharp
static IEnumerable GetResolvedDataSource(
    object dataSource, string dataMember);
```

If you’re writing an ASP.NET 1.x data-bound control, you might want to consider taking a look at this code. There are two possible ways to view it. One entails that you use a bit of reflection to call it into the method. The other is that you use a decompiler tool to snoop into the source code of the method, grab its logic, and rewrite it into your own project.

Thankfully, this is no longer a problem in ASP.NET 2.0. Data-bound controls are designed to override `PerformDataBinding`, and this method receives a ready-to-use `IEnumerable` collection, no matter what the originally bound data source was.

## Conclusion

Like it or not, real-world controls result from the composition of child controls and are data-bound. In addition, controls must be flexible enough to let people customize the user interface. When you
resort to visual properties, a rich set of properties, styles, and perhaps themes can do the job. Another, and deeper, level of customization can be achieved through templates.

In this chapter, we’ve analyzed the ASP.NET support for building data-bound controls. In ASP.NET 1.x, developers first needed to learn about best practices and then code their way to excellence. Books and articles were the only help; no support came from the framework itself. For example, you had to experience on your own the pain of having pages post back and having data-bound controls disappear or render almost empty. Once you’d figured out the whys and the wherefores, you had to update the control properly in a sort of trial-and-error scheme.

In ASP.NET 2.0, things go much better because a number of well-designed intermediate classes have been introduced to incorporate best practices and to let you choose exactly the level of support you want from the framework. As a result, you have a number of classes to build data-bound controls with minimal or richer base capabilities. We discussed the internal implementation of all these classes—BaseDataBoundControl, DataBoundControl, ListControl, and CompositeDataBoundControl. If you ever endured the pain of developing custom controls with ASP.NET 1.x, with these classes it might look like child’s play in ASP.NET 2.0.

We recognized and thoroughly examined three types of data-bound controls—simple binding, list controls, and complex binding. Along the way, we got into styles, rendering, composition, view-state management, data items, persistence, and templates.

Once you learn about data-bound and templated controls, you realize that all the powerful programmatic features you might cook into the control count for little if there’s no decent support from the Visual Studio design-time environment. Setting up a control for declarative programming is the next (and final) step in server control development and is the topic of the next chapter.

Just the Facts

- A data-bound control is characterized by a data source property and a bunch of string properties to define mappings between fields in the data source and parts of the control’s user interface.
- A data-bound control can be bound through either DataSource or DataSourceID properties. The former points to an IEnumerable object; the latter references a data source control.
- A simple data-bound control typically inherits from DataBoundControl and overrides PerformDataBinding.
- PerformDataBinding passes your code a ready-to-use enumerable collection of bound data, no matter what the bound data source was.
- The collection of bound data must be shrunk to a collection of data item objects and persisted to the view state to survive page postbacks. You create a data item class (and a collection of data item objects if your control binds to a list) and make it capable of serializing efficiently to the view state.
- To persist data items to the view state, you also override methods on the control class that save and restore the state to and from the view state.
• *ListControl* is a ready-to-use class that can be easily extended to “repeat” any control or tree of controls. To build a completely custom list control, though, you follow the steps outlined in this chapter and inherit from *DataBoundControl*.

• To extend list controls with advanced layout capabilities, you implement the *IRepeatInfoUser* interface and modify the rendering engine accordingly.

• *CompositeDataBoundControl* is the base class to use for building complex controls that result from the composition of child data-bound controls.

• Templated properties let page authors modify the standard user interface of a control by adding additional markup.