See This in MSDN Library

Examining the skmMenu Server Control

Scott Mitchell
4GuysFromRolla.com

December 2003

Applies to:
Microsoft® ASP.NET

Summary: Learn about the skmMenu ASP.NET menu server control source code and about ASP.NET server control development. (28 printed pages)

Page Options

Contents

Introduction

In a previous article, Building an ASP.NET Menu Server Control, we looked at using skmMenu, an open-source Microsoft® ASP.NET menu server control. The complete source code is available at the skmMenu GotDotNet Workspace. There are also some simple live demos of skmMenu in use:

- A simple menu with no style settings specified
- A demo illustrating styles
- A demo illustrating a horizontally laid out menu

Building an ASP.NET Menu Server Control focused on examining the client-side CSS and JavaScript mix needed to get interactive menus working, studying the object model used by skmMenu, and illustrating how to use skmMenu in an ASP.NET Web page. This article serves as a continuation, focusing on the code for skmMenu. (If you've yet to read Building an ASP.NET Menu Server Control, I highly recommend you do so before proceeding with this article.)

Before examining the skmMenu source code, we'll first do a quick review of the features and the object model. We'll then look at the code for the MenuItem and MenuItemCollection classes, which are the two key objects in the skmMenu object model. We'll also study how skmMenu recursively constructs this object model through databinding. Following this, we'll see how skmMenu provides ASP.NET page developers the ability to customize the stylistic appearance (background colors, borders, fonts, and so on). Finally, we'll turn our attention to how skmMenu maintains state between postbacks and how it ties server events to client-side events (namely, how skmMenu fires a server-side **MenuItemClick** event when the Web visitor clicks on a menu item. To fully understand these concepts, we'll take a high-level look at the life cycle all ASP.NET server controls go through each time the ASP.NET Web page they reside in is visited.

This article should provide informative and useful to readers who are either not very experienced with ASP.NET

server control development, and wish to learn more, or for those who are interested in working with skmMenu.

An Overview of skmMenu's Features and Object Model

As discussed in Building an ASP.NET Menu Server Control, a menu can be defined recursively like so: a menu is a set of menu items, where each menu item has a text, a specified action to perform when clicked, and an optional menu itself (as a submenu). Therefore, a menu can have menu items with submenus, and each of these submenus can have menu items with submenus, which themselves can have menu items with submenus, and on and on.

The object model for skmMenu encapsulates this recursive nature. Specifically, skmMenu employs three classes, as discussed in the previous article:

- Menu—the actual server control class, an instance of this class concretely represents the top-level menu.
- MenuItemCollection—represents a collection of menu items.
- MenuItem—represents a menu item in a menu. It contains properties like **Text**, **Url**, **CommandName**, and **SubItems**. Here, **SubItems** is a MenuItemCollection instance, and represents the menu item's (optional) submenu.

The recursion is made possible by the MenuItem's **SubMenu** property. Figure 1 shows this object model.

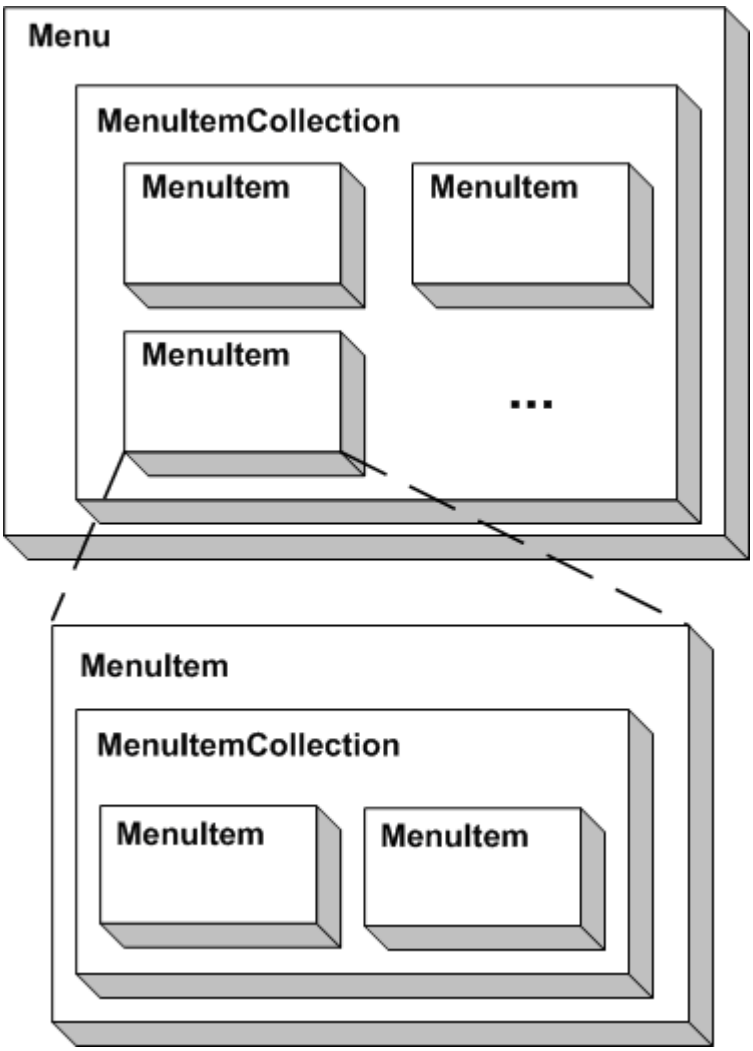

Figure 1. The skmMenu Object Model

To specify the menu's structure, the page developer can simply bind an XML document to skmMenu. When the Menu class's **DataBind()** method is called the appropriate recursive object model is built up based on the XML data provided in the **DataSource** property.

In addition to easily allowing the page developer to specify the menu's structure, skmMenu makes it easy to customize the menu's appearance by offering a number of `Style` properties. Specifically, the Menu class provides two `TableItemStyle` properties, **UnselectedMenuItemStyle** and **SelectedMenuItemStyle**, for specifying the style for the menu items when they are not selected and selected, respectively, and a **TableStyle MenuStyle**, for specifying the style for the HTML `<table>` element for the menu and its submenus. In addition to these `Style` properties, the Menu class provides a **Layout** property that can be set to either **MenuLayout.Vertical** or **MenuLayout.Horizontal**, specifying how the top-level menu should be laid out on the screen.

Finally, skmMenu allows developers to specify the behavior of a menu item when it is clicked. It may do nothing, it may whisk the user to a specified URL, or it may cause a postback, thereby raising the server-side event **MenuItemClicked**. An event handler can be associated with this event and when the event handler executes in response to the event, it is passed information on the menu item that was clicked.

Examining the Code for the MenuItemCollection and MenuItem Classes

The MenuItemCollection and MenuItems are used to represent a collection of menu items and a menu item, respectively. The MenuItemCollection class implements the `System.Collections.ICollection` interface, which means MenuItemCollection must implement the **GetEnumerator()** and **CopyTo()** methods, as well as **Count** and other properties. MenuItemCollection also provides a number of methods to add, remove, and edit the contained MenuItem instances. These methods include:

- **Add(menuItem)**
- **Insert(int, menuItem)**
- **AddRange(menuItemCollection)**
- **Clear()**
- **Remove(menuItem)**
- **RemoveAt(int)**

MenuItemCollection contains a number of properties, the two germane ones being **Count**, which returns the number of MenuItems in the collection, and an indexer, which allows the menu items to be accessed using the familiar indexing notation:

```
// C#
menuItemCollectionInstance[i]

' VB.NET
menuItemCollectionInstance(i)
```

Internally, the MenuItemCollection uses an ArrayList to maintain its collection of MenuItem instances.

> **Note** If you are currently examining the MenuItemCollection you may notice that MenuItemCollection also implements the `IStateManager` interface, and therefore has methods like **SaveViewState()**, **LoadViewState(object)**, and so on. This interface and these methods are needed for preserving changes to the MenuItemCollection during postback. We'll be discussing how changed state is preserved across postbacks in The ASP.NET Server Control Life Cycle section. (Note that the MenuItem class also implements `IStateManager`, which will also not be discussed until the later section.)

The MenuItem class is fairly simple. It needs no methods, just a number of properties. The germane properties are: **Text**, **Url**, **CommandName**, and **SubItems**. The **Text** property specifies the text to display in the menu item. Both the **Url** and **CommandName** properties are optional. If **Url** is specified, then when the menu item is

clicked the user is whisked to the specified URL. If **CommandName** is specified instead, then clicking the menu item causes the Web Form to postback, and the Menu class's **MenuItemClicked** event to fire, passing along the value of the menu item's **CommandName** property. Finally, if neither of these properties is set, then no action is performed when the menu item is clicked. As discussed earlier in this article, `SubItems` is an optional MenuItemCollection instance, and represents the menu item's submenu.

Constructing the skmMenu Object Model with DataBinding

skmMenu offers an easy interface for ASP.NET page developers to construct the menu system. Rather than having to programmatically fiddle with the MenuItem and MenuItemCollection classes, an ASP.NET page developer can simply build an XML file that represents the structure of the menu. The precise XML format was discussed in detail in Building an ASP.NET Menu Server Control, so we won't bother discussing the format in this article. As we saw in the previous article, like with binding data to one of the data Web controls, for the page developer binding the XML data to skmMenu can be accomplished in just two lines of code:

```
menuID.DataSource = Server.MapPath(xmlFileName);
menuID.DataBind();
```

The Menu class has a **DataSource** property of type `object`, but in actuality the property only accepts values of either one of two types: a string, which indicates the physical path to the XML file to use, or an `XmlDocument` instance. The following property statement, found in the Menu class, illustrates this:

```
private object dataSource = null;
public object DataSource
{
   get
   {
      return this.dataSource;
   }
   set
   {
      if (value is string || value is XmlDocument)
         this.dataSource = value;
      else
         throw new ArgumentException("DataSource must be a string or
           XmlDocument instance.");
   }
}
```

The get accessor simply returns the value of the **dataSource** property, but the set accessor checks to make sure the entered value is of type string or type `XmlDocument` by using the C# `is` statement. If the value is not of either of these types, then an `ArgumentException` is thrown.

Realize that the Menu class is derived from the `System.Web.UI.Control` class, which all ASP.NET server controls must be derived from either directly or indirectly. This class provides the base set of methods, properties, and events needed for a server control. One such method is the **DataBind()** method, which calls the protected method **OnDataBinding()**. The Menu class overrides the **OnDataBinding()** method (shown below, some content omitted for brevity), which constructs the object model recursively.

```
protected override void OnDataBinding(EventArgs e)
{
   // Start by resetting the Control's state
   this.Controls.Clear();

   // load the datasource either as a string or XmlDocuemnt
   XmlDocument xmlDoc = new XmlDocument();

   if (this.DataSource is String)
      // Load the XML document specified by DataSource as a filepath
      xmlDoc.Load((string) this.DataSource);
```

```
   else if (this.DataSource is XmlDocument)
      xmlDoc = (XmlDocument) DataSource;
   else
      throw new ArgumentException("DataSource either null or not of the
        correct type.");

   // Clear out the MenuItems and build them according to the
     XmlDocument
   this.items.Clear();
   this.items = GatherMenuItems(xmlDoc.SelectSingleNode("/menu"),
     this.ClientID);
   BuildMenu();

   this.ChildControlsCreated = true;
}
```

This method begins by clearing out all of the children controls. Recall from the previous article that skmMenu constructs each menu as an HTML <table> element. Therefore, if there is a top-level menu with three menu items, where one menu item has a submenu, there will be two <table> elements—one for the top-level menu and one for the submenu. Next, an XmlDocument instance, xmlDoc, is created. If the DataSource is a string (specifying the file location of the XML document), then xmlDoc's **Load()** method is called to load the XML file. Otherwise, the DataSource is an XmlDocument instance, which is assigned to xmlDoc.

In the next two lines, the private member variable **items** is first cleared out, and then constructed. Recall from Building an ASP.NET Menu Server Control and our discussions earlier in this article that the Menu class contains a single MenuItemCollection that represents the top-level menu. This is precisely what items is—the MenuItemCollection instance. The **Clear()** method clears out any menu items that may be present in items, while the **GatherMenuItems()** method recursively populates the MenuItemCollection with the appropriate MenuItem instances. Note that the **GatherMenuItems()** method (shown below), accepts as an input the XmlNode retrieved by the XPath expression /menu and the control's ClientID. The XPath expression /menu will get the XmlNode corresponding to the XML documents root node (recall that the root element for the XML document is <menu>).

```
private MenuItemCollection GatherMenuItems(XmlNode itemsNode, string
  parentID)
{
   // Make sure we have an XmlNode instance - it should never be null,
     else the
   // XML document does not have the expected structure
   if (itemsNode == null)
      throw new ArgumentException("The XML data for the Menu control is
        in an invalid format.");

   MenuItemCollection myItems = new MenuItemCollection();

   // iterate through each MenuItem
   XmlNodeList menuItems = itemsNode.SelectNodes("menuItem");
   for (int i = 0; i < menuItems.Count; i++)
   {
      XmlNode menuItem = menuItems[i];

      // Create the menu item
      myItems.Add(BuildMenuItem(menuItem, parentID, i));
}

   return myItems;
}
```

The **GatherMenuItems()** method's job is to take in an XmlNode and a parentID string and generate a set of MenuItem instances, returned in a MenuItemCollection instance. This is accomplished in the following steps:

1. Check to ensure that the XmlNode instance is not null. If the XmlNode instance is null then the expected element was not found in the XML document. This means that the XML document specified by the user is not according to the specifications laid out previously. In such a case an ArgumentException is thrown.

2. Create a new MenuItemCollections instance.
3. Iterate through each of the current <menuItem> child element of the passed-in XmlNode. For each such element, the **BuildMenuItem()** method is called, passing in the XmlNode corresponding to the <menuItem> element, the parentID, and the node's ordinal position in the node list. The **BuildMenuItem()** method returns a MenuItem instance, which is then added (Add()) to the MenuItemCollection.

The **BuilddMenuItem()**, shown below, parses through the XmlNode instance corresponding to the <menuItem> element, building and returning a MenuItem instance.

```
private MenuItem BuildMenuItem(XmlNode menuItem, string parentID, int
  indexValue)
{
   MenuItem mi = new MenuItem();

   XmlNode textTextNode = menuItem.SelectSingleNode("text/text()");
   XmlNode urlTextNode = menuItem.SelectSingleNode("url/text()");
   XmlNode commandNameTextNode =
     menuItem.SelectSingleNode("commandName/text()");

// Format the indexValue so its three-digits (allows for 1,000 menu
  items per (sub)menu
   mi.ID = parentID + "-menuItem" + indexValue.ToString("d3");

   if (textTextNode == null)
      // whoops, the <text> element is required
      throw new ArgumentException("The XML data for the Menu control is
      in an invalid format: missing the <text> element for a
        <menuItem>.");

   mi.Text = textTextNode.Value;

   if (urlTextNode != null)
      mi.Url = urlTextNode.Value;

   if (commandNameTextNode != null)
      mi.CommandName = commandNameTextNode.Value;

   // see if there is a submenu
   XmlNode subMenu = menuItem.SelectSingleNode("subMenu");
   if (subMenu != null)
   {
      // Recursively processes the <menuItem>'s <subMenu> node, if
        present
      mi.SubItems.AddRange(GatherMenuItems(subMenu, mi.ID + "-
        subMenu"));
   }

   return mi;
}
```

Note that the **ID** property of the created MenuItem instance mi is set to the passed-in parentID concatenated with the string –menuItem, and then concatenated with the passed-in ordinal value of the XmlNode in the node list. This constructs the ID according to the specifications discussed in the previous article. For example, for a Menu class with a ClientID of Menu1, the first top-level menu item will have an ID value of **Menu1-menuItem000**, while the second top-level menu item will have an ID value of **Menu1-menuItem001**.

Next, the <text>, <url>, and <commandName> XML elements' text values are read and assigned to the MenuItem's properties, as needed. Finally, a check is made to see if the <menuItem> element has a <subMenu> element. If it does, the MenuItem's **SubItems** property's **AddRange()** method is called, adding the items in the MenuItemCollection instance returned by GatherMenuItems(). When GatherMenuItems() is called from this method, the parentID passed in is the current MenuItem's ID concatenated with the string –subMenu. Therefore, if the first top-level menu item has a submenu, the first menu item in the submenu will have an ID of Menu1-menuItem000-subMenu-menuItem000.

Notice that this algorithm for building the menu's structure is recursive in nature, allowing for an arbitrary number of submenus. Specifically, the menu is constructed using a depth-first traversal.

The **GatherMenuItems()** and **BuildMenuItem()** methods work together to construct the object model from the XML document specified by the ASP.NET page developer. Two other methods—**BuildMenu()** and **AddMenu()**—actually add the <table>, <tr>, and <td> elements to the Menu class's Controls collection based upon the object model. **BuildMenu()** is called from the **OnDataBinding()** method and constructs the menu for either a horizontal or vertically laid out top-level menu. **AddMenu()** is called to create the submenu, if needed, for each MenuItem in the top-level menu. **AddMenu()** is quite similar to **BuildMenu()**, but only concerns itself with laying out the menu in a vertical orientation, since only the top-level menu can have a horizontal orientation. The code in these methods is fairly straightforward, so we'll bypass an in-depth examination.

Specifying Style Information

Most often ASP.NET server controls are rendered as a single HTML element or a series of HTML elements. For example, skmMenu renders as a series of HTML <table> elements, one for each submenu, with each <table> element having inner <td> elements for each menu item. As server control developers, we are tasked with making it easy for the user to specify the style for the HTML element(s) generated by our control.

The .NET Framework provides a number of classes especially designed to specify style information, starting with the Style class. The Style class contains the minimum set of stylistic information that can be specified for a Web control. This includes properties like **BackColor, ForeColor, Font, Width, Height,** and so on. The Style class serves as a base class for two other classes: TableStyle and TableItemStyle. These classes represent the style information for an HTML <table> element and a <td> element, respectively. They extend the functionality of Style by adding <table>-specific style properties, such as **HorizontalAlign, VerticalAlign,** and **Wrap**.

For skmMenu, then, it makes sense to allow the page developer to specify information both about the <table> elements and the <td> elements. Furthermore, when specifying style information for the <td> elements, the page designer should be able to specify information for when the menu item is selected (that is, the mouse is currently hovering over the menu item), and when the menu item is unselected (when the mouse is *not* over the menu item). To accomplish this, we'll need three private member variable style properties:

```
// styles for the Menu, and unselected & selected menu items...
private TableItemStyle unselectedMenuItemStyle = new TableItemStyle();
private TableItemStyle selectedMenuItemStyle = new TableItemStyle();
private TableStyle menuStyle = new TableStyle();
```

Now, to let the page developer specify these values, we need to provide public properties for each of the three styles. These public properties should only expose a get accessor. Additionally, these properties can include optional attributes that inform the Microsoft Visual Studio® .NET Designer how to display the property settings in the Web control. These attributes are optional, and skmMenu will work without specifying them. However, omitting them will greatly reduce the design-time experience in Visual Studio .NET. The three public properties are shown below:

```
[
   PersistenceMode(PersistenceMode.InnerProperty),
   DesignerSerializationVisibility(DesignerSerializationVisibility.
     Content),
]
public TableItemStyle SelectedMenuItemStyle
{
   get
   {
      return this.selectedMenuItemStyle;
   }
}
```

```
[
   PersistenceMode(PersistenceMode.InnerProperty),
   DesignerSerializationVisibility(DesignerSerializationVisibility.
     Content),
]
public TableItemStyle UnselectedMenuItemStyle
{
   get
   {
      return this.unselectedMenuItemStyle;
   }
}


[
   PersistenceMode(PersistenceMode.InnerProperty),
   DesignerSerializationVisibility(DesignerSerializationVisibility.
     Content),
]
public TableStyle MenuStyle
{
   get
   {
      return this.menuStyle;
   }
}
```

Each style property here specifies two attributes, **PersistenceMode** and **DesignerSerializationVisibility**. The first attribute specifies how the property should be rendered in the control's syntax. The options here are `PersistenceMode.InnerProperty` or `PersistenceMode.Attribute`. With `PersistenceMode.InnerProperty`, the style information is specified in a nested tag within the control's outer tag. With `PersistenceMode.Attribute`, the style information is specified as an attribute of the control's tag using hyphenation. For example, consider the case where the `MenuStyle`'s **BackColor** property is set to Silver in the Visual Studio .NET Designer. Since `MenuStyle`'s **PersistenceMode** attribute is set to `PersistenceMode.InnerProperty`, Visual Studio .NET will render the style information like so in the control's syntax:

```
<skm:Menu runat="server" ...>
  <MenuStyle BackColor="Silver"></MenuStyle>
</skm:Menu>
```

However, had the **PersistenceMode** attribute been set to `PersistenceMode.InnerProperty`, the output would be:

```
<skm:Menu runat="server" MenuStyle-BackColor="Silver" ...>
</skm:Menu>
```

The **DesignerSerializationVisibility** attribute tells the designer to step into the subproperties of the style property and serialize their values in the control's tag. Omitting this attribute setting will not cause changes in the Visual Studio .NET Designer to be reflected in the control's syntax.

With these three style properties and attribute settings, page developers can specify style information in the Visual Studio .NET Designer via the Properties pane, as shown in **Figure 2**.

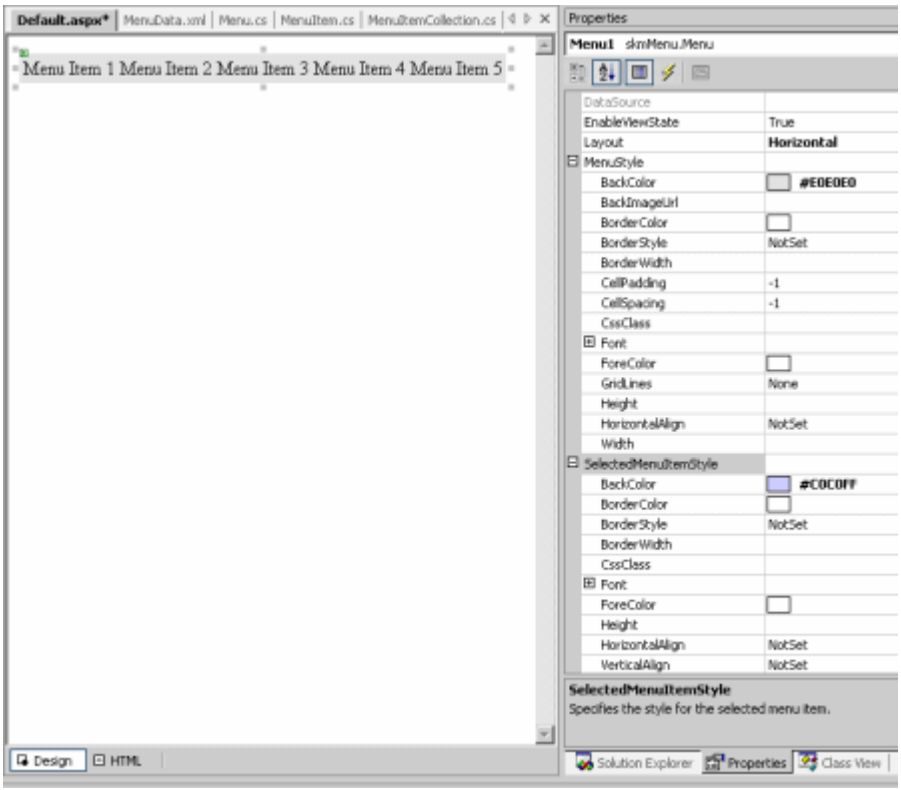

Figure 2. The Style properties can be set in the Visual Studio .NET Designer, and are persisted to the skmMenu control's syntax in the HTML portion.

The ASP.NET Server Control Life Cycle

All ASP.NET server controls go through a similar sequence of steps each time the ASP.NET Web page they reside on is requested. These steps are important to understand when developing controls, especially when building controls that can have server-side events fire in response to client-side events (for example, having the **MenuItemClicked** event fire when the Web visitor clicks a menu item that has a **CommandName** property value).

Before we delve into the various stages, let's take a moment to describe how a control's life cycle gets started in the first place. First, realize that each ASP.NET Web page is represented by a class instance that derives either directly or indirectly from `System.Web.UI.Page`. The Page class contains a method called **ProcessRequest()**, which is called when a page is requested. This method (and the methods it calls) adds the list of page controls to the Page class' `Controls` collection, and starts off the control life cycle for each control in the page. For more information on the Page class and its life cycle, refer to Dino Esposito's article, The ASP.NET Page Object Model or Solomon Shaffer's The ASP.NET Page Life Cycle.

The control stages occur in the order outlined below:

- **Initialize**—Each control in the Page class's Controls collection has its **OnInit** method called, which fires its **Init** event. Any properties that were declared in the control's syntax in the HTML portion of the ASP.NET Web page are set during this stage. That is, if you have a Label Web control defined like so:

  ```
  <asp:Label runat="server" ForeColor="Navy" Text="Foo"></asp:Label>
  ```

  In this stage the Label Web control instance will have its **ForeColor** property set to **Color.Navy** and its **Text** property set to **"Foo."**

- **ViewState Tracking Begins**—The last thing performed in the **Initialize** stage is the control's **TrackViewState()** method is called. The **TrackViewState()** method simply serves as a notification to the control that any changes to its state must be tracked.
- **LoadViewState()**—Next, the control's **LoadViewState()** method is called. This method's task is to restore the state of the control from the end of the previous page request.
- **Load Postback Data**—This stage only occurs if there has been a postback. During this stage the control must process the postback data to determine if any of its properties have been changed by user interaction. A typical example is the TextBox Web control—upon postback it must inspect the postback data to see if the user entered a value into its corresponding textbox HTML element, and set its **Text** property accordingly. Since skmMenu does not collect any user input, we do not need to add any code to handle this stage.
- **Load**—The control's **OnLoad()** method is called in this stage. At this point in the life cycle, all controls in the Page class's `Controls` collection can be accessed, and all of the state from the previous Web page request has been restored.
- **Raise Changed Events**—This stage only happens during postback and only if the Load Postback Data stage is also used. Essentially, it offers an opportunity for controls to raise an event signifying its state has changed across postback due to user input. Returning to the TextBox Web control example, in this stage the TextBox would raise the **TextChanged** event if the **Text** property had changed across postback due to the user entering a different value.
- **Raise Postback Event**—During this stage the control can fire an event based on some client-side action. With skmMenu, this stage only executes if the user has clicked a menu item that causes a postback. If this is the case, during this stage the **MenuItemClicked** event is raised.
- **PreRender**—This stage is useful for performing any tasks that need to be completed before the control is rendered. skmMenu uses this stage to build the client-side JavaScript that is needed for the interactive menus (specifics of the client-side JavaScript were discussed in Building an ASP.NET Menu Server Control).
- **SaveViewState()**—In this stage, the control's **SaveViewState()** method is called, which provides an opportunity for the control to store its state in the Web Form's hidden `VIEWSTATE` form field. This is the same data that, if the Web Form is posted back, will be loaded and parsed in the control's **LoadViewState()** method.
- **Render**—In this stage the control generates the markup that will be emitted to the Web visitor's browser.

Note The **Unload** and **Dispose** stages in the control life cycle have been omitted for brevity.

The server control life cycle is important to understand when working with controls with complex properties and controls that allow user interaction in the form of either user input or responding to client-side actions. In the remaining sections we'll see how skmMenu progresses through these steps, playing particularly close attention to the **LoadViewState(), SaveViewState(),** and **Raise Changed Events** stages.

Saving State: Examining the SaveViewState() Methods

Near the end of the control's life cycle, it is imperative that it save the state specified programmatically so that this information can be reloaded if the Web Form the control resides on is posted back. State that is specified declaratively in the control's tag syntax need not be saved as this information will be automatically reloaded in the **Init** stage upon postback. To make this concept clearer, consider a TextBox Web control on an ASP.NET Web page with the following declaration:

```
<asp:TextBox runat="server" id="myTextBox"></asp:TextBox>
```

When the ASP.NET page is visited, a TextBox control is added to the Page's `Controls` collection and the **ID** property of the control is set to **myTextBox**. Now, image in that Page_Load event handler you have the following lines of code:

```
if (!Page.IsPostBack)
  myTextBox.Columns = 3;
```

This sets the TextBox's **Columns** property to **3**. Later, in the **Render** stage, the TextBox will be rendered as an `<input type="text">` HTML element with the attribute `size="3"`. Imagine also that there is a Button Web

control on the page. Now, if the TextBox does not save its programmatically added state—namely that its **Columns** property has been assigned the value 3—then when the user clicks the button and posts back the Web Form, this **Columns** value will have been lost. To see why it would be lost, simply step through the sequence of events again:

When the ASP.NET page is visited, a TextBox control is added to the Page's `Controls` collection and the **ID** property of the control is set to `myTextBox`. Now, the **Columns** property in the `Page_Load` event handler is not set this time (since `Page.IsPostBack` is true), so the TextBox has the default **Columns** property value, **0**. So the TextBox will be rendered *without* the attribute `size="3"`.

Controls allow information to be saved across postbacks by providing a `StateBag` object named `ViewState`. In ASP.NET page development, you may have used the Page class's `ViewState` to save simple values across postbacks. The ViewState is optimized to store scalar values of type **Int32, Boolean, String, Unit,** and **Color** and non-scalar values of type **Array, ArrayList,** and **Hashtable**.

In order to maintain state across postbacks, it is vital that the control uses the `ViewState` object to store the value in the set accessors for properties, and to access the stored value from the `ViewState` in the get accessors. Consider the Menu class's **Layout** property, which has the following code:

```
public MenuLayout Layout
{
   get
   {
      object o = ViewState["MenuLayout"];
      if (o == null)
         return MenuLayout.Vertical;
      else
         return (MenuLayout) o;
   }
   set
   {
      ViewState["MenuLayout"] = value;
   }
}
```

The **Layout** property specifies if the top-level menu is laid out horizontally or vertically. `MenuLayout` is an enumeration with the following definition:

```
public enum MenuLayout { Horizontal, Vertical }
```

Note that in the **Layout** property's get accessor, the `ViewState` is consulted to retrieve the value of the property. If the `ViewState` does not contain a value, then the property has not been specified, so the default is returned (`MenuLayout.Vertical`). Else, the value in the `ViewState` is cast to a `MenuLayout` and returned. In the set accessor, the appropriate **ViewState** value is assigned the value specified.

This template needs to be used for all properties whose value can be easily expressed as a simple string. For complex properties, such as properties that are class instances like **MenuStyle** and **Items**, the task of state saving is left to these classes. The property syntax for such complex properties provides just a get accessor. In the get accessor, simply check to see if the control's **IsTrackingViewState** variable is true—if it is, call the **TrackViewState()** method of the complex property. The code for the **MenuStyle** property shown below illustrates this:

```
public TableStyle MenuStyle
{
   get
   {
      if (this.IsTrackingViewState)
         ((IStateManager) this.menuStyle).TrackViewState();

      return this.menuStyle;
```

```
    }
}
```

> **Note** This `if` statement was omitted for brevity when we earlier examined the **MenuStyle** property.

The `Control` class has a protected **IsTrackingViewState** Boolean property that indicates whether or not the control's `ViewState` is being tracked. This property is false by default, and made true when the control's **TrackViewState()** method is called. Recall from the previous section that **TrackViewState()** is called at the end of the **Init** stage, and indicates to the control that any changes to its properties henceforth need to be stored in the `ViewState`. The purpose for this is so that the declaratively specified properties are not needlessly stored in the `ViewState`.

So, if the control's `ViewState` is being tracked, then we want to make sure that the complex property's `ViewState` is also being tracked. That's what this `if` statement in the get accessor accomplishes.

By default, in the `SaveViewState()` cycle a control will save just the items in its `ViewState`. This means that if our control's properties are all simple properties that can be stored in the `ViewState`, then we don't need to bother overriding the **SaveViewState()** method. However, the Menu class has complex properties—**MenuStyle, SelectedMenuItemStyle, UnselectedMenuItemStyle,** and **Items**—and therefore we need to override this method and ensure not only the `ViewState` is saved, but also the `ViewStates` of these complex properties.

The **SaveViewState()** method has no input parameters, but needs to return an object that represents the state. When overriding this method, typically you'll want to return an `object` array, storing into the various indices of the array the base `ViewState` along with the complex properties' `ViewStates`. Each index of the array can be populated via a call to the appropriate **SaveViewState()** method, as shown below:

```
protected override object SaveViewState()
{
   Object [] state = new object[5];
   state[0] = base.SaveViewState();
   state[1] = ((IStateManager)
     this.selectedMenuItemStyle).SaveViewState();
   state[2] = ((IStateManager)
     this.unselectedMenuItemStyle).SaveViewState();
   state[3] = ((IStateManager) this.menuStyle).SaveViewState();
   state[4] = ((IStateManager) this.items).SaveViewState();

   return state;
}
```

Here a five-element array is used because we need to store the base `ViewState` and the `ViewState` of the four complex properties.

The `TableStyle` and `TableItemStyle` classes provide **SaveViewState()** methods, so we can simply delegate that saving of the state of these style properties to these classes' **SaveViewState()** methods. For the **Items** property, however, which is of type MenuItemCollection, we need to provide a **SaveViewState()** method. The MenuItemCollection's **SaveViewState()** method will in turn call the **SaveViewState()** method of (potentially) each of its MenuItem instances, so we'll need to provide a **SaveViewState()** method for the MenuItem class as well. We'll examine adding this method to both the MenuItemCollection and MenuItem classes later on in this article.

Loading State: Examining the LoadViewState() Method

One of the stages a control goes through when the page is reside on has been posted back is the **LoadViewState()** stage. The **LoadViewState()** method serves as an inverse to the **SaveViewState()** method—its job is to take the state saved by `SaveViewState()` and reload the state of the control. Not surprisingly, **LoadViewState()** returns no parameters, but accepts a single input parameter of type *object*. This passed-in *object* parameter is

precisely the state saved in the **SaveViewState()** method on the previous page visit.

Like with **SaveViewState()**, if all of your control's properties are simple properties, you don't need to override **LoadViewState()**. If you have complex properties then you need to override **SaveViewState()** (as we saw in the previous section) and **LoadViewState()** as well. The code for **LoadViewState()** for the Menu class is fairly straightforward—we simply cast the passed-in *object* parameter to an array of `objects` and then pass in the appropriate array elements into the appropriate **LoadViewState()** methods of the `Control` class and complex properties:

```
protected override void LoadViewState(object savedState)
{
   object [] state = null;

   if (savedState != null)
   {
      state = (object[]) savedState;

      base.LoadViewState(state[0]);
      ((IStateManager)
        this.selectedMenuItemStyle).LoadViewState(state[1]);
      ((IStateManager)
        this.unselectedMenuItemStyle).LoadViewState(state[2]);
      ((IStateManager) this.menuStyle).LoadViewState(state[3]);
      ((IStateManager) this.items).LoadViewState(state[4]);
   }
}
```

As aforementioned, the TableStyle and TableItemStyle classes already implement the **LoadViewState()** method, but we'll need to add the **LoadViewState()** method to the MenuItemsCollection class ourselves, which we'll see in the next section.

State Management of MenuItems and MenuItemCollection

In the Menu class we stored the state of our various simple properties by using the `ViewState` object, which is a protected member variable of the `Control` class. Since the Menu class is derived from the `Control` class, it has access to `ViewState`. For complex properties, we needed to delegate saving and loading of their state to the objects themselves, by calling their **SaveViewState()** and **LoadViewState()** methods, respectively.

In addition to the Menu class, the MenuItemCollection and MenuItems need to store and load their state as well. To accomplish this they must implement the IStateManager interface, which requires them to provide an **IsTrackingViewState** property and three methods: **SaveViewState(), LoadViewState(),** and **TrackingViewState()**. Both MenuItem and MenuItemCollection will need to implement this interface and provide this property and these methods.

Ideally, we'd like to handle state management in the MenuItem class in the exact same way handled it in the Menu class. The problem is that MenuItem doesn't have a **ViewState** property. The solution? Add one manually. The `ViewState` in the `Control` class is of type `StateBag`, so we create a private `StateBag` class instance in MenuItem and provide a protected property named **ViewState**, as shown below:

```
protected StateBag stateBag = new StateBag();
protected StateBag ViewState
{
   get
   {
      return this.stateBag;
   }
}
```

Then, in MenuItem's simple properties—**ID, Text, Url, CommandName,** and **Layout**—we read the value from the `ViewState` in the get accessor and write it to the `ViewState` in the set accessor. Below the **Text** property is

shown (the other simple properties are omitted for brevity):

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

The MenuItem contains a complex property in addition to its simple properties. This complex property, **SubItems**, is a MenuItemCollection instance. Again, as with the Menu class, with complex properties we simply delegate the management of state information to the property's class itself. The MenuItem's **SaveViewState()** and **LoadViewState()** methods, shown below, simply store and load the ViewState and the SubItems state in and from a Pair object.

```
object IStateManager.SaveViewState()
{
   object baseState = ((IStateManager) this.ViewState).SaveViewState();
   object subItemsState = ((IStateManager)
     this.subItems).SaveViewState();

   if (baseState == null && subItemsState == null)
      return null;
   else
      return new Pair(baseState, subItemsState);
}

void IStateManager.LoadViewState(object savedState)
{
   if (savedState != null)
   {
      Pair p = (Pair) savedState;
      if (p.First != null)
         ((IStateManager) this.ViewState).LoadViewState(p.First);
      if (p.Second != null)
         ((IStateManager) this.subItems).LoadViewState(p.Second);
   }
}
```

> **Note** The Pair class is designed specifically to hold two items in an ASP.NET server control's view state. For more information refer to the .NET Framework Class Library: Pair Class. Also, there is a Triplet class handy for storing three items.

MenuItem also needs to implement the **TrackingViewState()** method. Recall that this method is used to inform the class that it should store any changes made to its properties henceforth. The **TrackingViewState()** method simply sets a flag to true (the private isTrackingViewState member variable), and then calls the **TrackingViewState()** method of the **ViewState** and **SubItems** properties.

```
private bool isTrackingViewState = false;
void IStateManager.TrackViewState()
{
   isTrackingViewState = true;

   if (this.stateBag != null)
```

```
      ((IStateManager) this.stateBag).TrackViewState();

   if (subItems != null)
      ((IStateManager) subItems).TrackViewState();
}
```

MenuItem needs an additional state management helper function, one that indicates that all of the MenuItem's properties should be saved into the ViewState. This scenario can arise in the following sequence of steps transpire:

- An ASP.NET Web page is created with a skmMenu control and a Button Web control.
- The ASP.NET Web page's Page_Load event handler binds an XML file to skmMenu when Page.IsPostBack is false.
- The Button's Click event handler creates and adds a new MenuItem to the menu.

Now, when the page is first visited, the **SaveViewState()** method runs and saves the state of the menu control correctly. When the Button is clicked, the Web Form posts back. First, the **LoadViewState()** method runs, which loads the object model back from the preserved state correctly. Then, a new MenuItem is created and added. Since this was added after the **LoadViewState()** method, the MenuItemCollection has had its **TrackViewState()** method called. But this newly created MenuItem has not had its **TrackViewState()** method called, so its state won't be preserved when **SaveViewState()** is called later.

In order to forcibly require that this new MenuItem's state be saved, we need to create a helper method in the MenuItem class that, when called, marks all of the items in the ViewState as dirty. Then, this method needs to be called from the MenuItemCollection's methods that allow for a MenuItem to be added to the collection (Add() and Insert(), specifically). This helper method is named **SetDirty()**, and is shown below:

```
internal void SetDirty()
{
   if (this.stateBag != null)
   {
      ICollection keys = stateBag.Keys;
      foreach (string key in keys)
         stateBag.SetItemDirty(key, true);
   }
}
```

Like the MenuItem class, the MenuItemCollections class also implements the IStateManager interface. MenuItemCollections, however, does not have any simple properties, so it does not need to add a **ViewState** property. Rather, its only property whose state we're interested in is the private **ArrayList** that stores the set of MenuItem instances.

The **SaveViewState()** method saves the state of the MenuItems by creating an object array and, for each element in the MenuItemCollection, the item's **SaveViewState()** method is called and stored in the corresponding object array element. The **LoadViewState()** method simply performs the inverse of this task, looping through the passed-in object array, creating a new MenuItem for each element, adding it to the private ArrayList, and loading the MenuItem's state with a call to **LoadViewState()**.

```
object IStateManager.SaveViewState()
{
   bool isAllNulls = true;
   object [] state = new object[this.menuItems.Count];
   for (int i = 0; i < this.menuItems.Count; i++)
   {
      // Save each item's viewstate...
      state[i] = ((IStateManager) this.menuItems[i]).SaveViewState();
      if (state[i] != null)
         isAllNulls = false;
   }
```

```
   // If all items returned null, simply return a null rather than the
     object array
   if (isAllNulls)
      return null;
   else
      return state;
}

void IStateManager.LoadViewState(object savedState)
{
   if (savedState != null)
   {
      object [] state = (object[]) savedState;

      // Create an ArrayList of the precise size
      menuItems = new ArrayList(state.Length);

      for (int i = 0; i < state.Length; i++)
      {
         MenuItem mi = new MenuItem();
           ((IStateManager) mi).TrackViewState();

// Add the MenuItem to the collection
         menuItems.Add(mi);

         if (state[i] != null)
         {
            // Load its state via LoadViewState()
            ((IStateManager) menuItems[i]).LoadViewState(state[i]);
         }
      }
   }
}
```

Looking at the MenuItemCollection's **Add()** method you can see that **Add()** not only adds the MenuItem to the collection, but also, if the view state is being tracked, starts tracking the MenuItem's view state *and* calls the MenuItem's **SetDirty()** method.

```
public int Add(MenuItem item)
{
   int result = menuItems.Add(item);

   if (this.isTrackingViewState)
   {
      ((IStateManager) item).TrackViewState();
      item.SetDirty();
   }

   return result;
}
```

Responding to a Client-Side Event with a Server-Side Event

ASP.NET server controls are much more powerful and useful to page developers when a server control can raise a server-side event in response to a client-side action. For example, the Button Web control renders as a submit button in the user's browser. When clicked, it causes a postback and raises the Button Web control's **Click** event, to which page developers can create a server-side event handler.

skmMenu provides similar functionality. A menu item with its **CommandName** property set is marked to raise a server-side event when clicked. When a user clicks such a menu item, the Web Form containing the menu item causes a postback and the **MenuItemClicked** event is fired. As we'll see in this section, firing a server-side event in response to a client-side event is accomplished in the **Raise Postback Event** stage of a control's life cycle, and takes only a few lines of code.

There are five steps to adding a server-side event and having it fire when an associated client-side event executes. These five steps, in summary, are:

1. Create an appropriate delegate. This might entail creating a custom class derived from `EventArgs` if there does not already exist such a class with the properties needed.
2. Create the event in the server control.
3. Create an **OnEventName** protected method that raises the event.
4. Create the **IPostBackEventHandler.RaisePostBackEvent** method in your server control, and have it call the appropriate **OnEventName** method.
5. Add the appropriate JavaScript code to the ASP.NET server control to cause a postback when the desired client-side event transpires.

The first step in associating a server-side event with a client-side action is to create the delegate for the event. A thorough discussion on delegates and event firing and processing is beyond the scope of this article; for more information refer to Events and Event Handling in C#. Essentially a delegate allows a function to be passed as a parameter to a method. The delegate indicates the signature the function that is passed must implement.

Our delegate, `MenuItemClickedEventHandler`, specifies the signature for the Menu class's **MenuItemClicked** event. Like all event delegates in the .NET Framework, the **MenuItemClicked** event delegate needs to define two parameters: an *object*, representing the control that raised the event, and a *class* derived from `EventArgs`, which specifies information about the event.

```
public delegate void MenuItemClickedEventHandler(object sender,
  MenuItemClickEventArgs e);
```

Notice that this delegate defines a function signature that accepts two parameters: an *object* and a *MenuItemClickEventArgs*. The `MenuItemClickEventArgs` class, shown below, provides a single string property, **CommandName**. When a menu item that causes a postback is clicked, the **MenuItemClickEventArgs's CommandName** property is set to the **CommandName** property of the clicked menu item. Therefore, the page developer can determine what menu item was clicked in skmMenu's `MenuItemClicked` event handler.

```
public class MenuItemClickEventArgs : EventArgs
{
   private string commandName;

   public MenuItemClickEventArgs(string name)
   {
      commandName = name;
   }

   /// <summary>
   /// Readonly access to commandName parameter of EventArgs class
   /// </summary>
   public string CommandName
   {
      get
      {
         return commandName;
      }
   }
}
```

The second step in associating a server-side event with a client-side event is to add a public event to the server control class of the delegate type defined in the first step, as shown below:

```
public event MenuItemClickedEventHandler MenuItemClick;
```

With this step completed, your control now has an event.

Step three is adding an **OnEventName** method that, when called, will raise the event. This method should accept a single input parameter, a *MenuItemClickEventArgs* instance, and raise the event passing along this passed-in instance.

```
protected virtual void OnMenuItemClick(MenuItemClickEventArgs e)
{
   if (MenuItemClick != null)
      MenuItemClick(this, e);
}
```

The fourth step is to add the **IPostBackEventHandler.RaisePostBackEvent** method to the control. This method is executed in the **Raise Postback Event** stage of the control's life cycle if. Realize that this stage only occurs if the page the control resides upon has been posted back and the control implements the `IPostBackEventHandler` interface. The `IPostBackEventHandler.RaisePostBackEvent` method accepts a string input, which is an optional event argument that can be specified by a client-side event. For our purposes, we'll have the menu item, when clicked, pass in through the event argument the value of its **CommandName** property. Therefore, the **IPostBackEventHandler.RaisePostBackEvent** method will receive the **CommandName** property to assign to the `MenuItemClickEventArgs` instance through its sole input parameter.

```
void IPostBackEventHandler.RaisePostBackEvent(string eventArgument)
{
   OnMenuItemClick(new MenuItemClickEventArgs(eventArgument));
}
```

All the **IPostBackEventHandler.RaisePostBackEvent** method does is call the **OnMenuItemClick** event passing in a new `MenuItemClickEventArgs` instance. Note that the *eventArgument* input parameter—which, again, contains the `CommandName` of the menu item clicked—is passed into the `MenuItemClickEventArgs` constructor, which sets the `MenuItemClickEventArgs`'s **CommandName** property.

The fifth and final step is to add the JavaScript code needed to the desired client-side event to trigger a postback. Specifically, when the menu item is clicked, we want to cause a postback and pass in the menu item's `CommandName` value.

Fortunately, we do not have to author any client-side JavaScript ourselves to cause a postback. Rather, we can simply use the **Page.GetPostBackClientHyperlink()** method, which takes in two parameters: the control causing the postback and the string event argument to pass along. Realize that the value of this second parameter is the value that is passed into the **IPostBackEventHandler.RaisePostBackEvent** method.

Recall that skmMenu can have menu items that, when clicked, either do nothing, redirect the user to a specified URL, or cause a postback. Which course of action depends upon the values of the **Url** and **CommandName** properties of the MenuItem instance. Furthermore, recall that each menu item is rendered as a `<td>` element. Based on what properties are set, the menu item's `<td>` element has its client-side **onclick** event set to take the appropriate course of action, as the following code snippet from the **BuildMenu()** method illustrates:

```
if (mi.Url != String.Empty)
   td.Attributes.Add("onclick", "javascript:location.href='" + mi.Url +
     "'");
else if (mi.CommandName != String.Empty)
   td.Attributes.Add("onclick", Page.GetPostBackClientHyperlink(this,
     mi.CommandName));
```

Here `mi` is a MenuItem instance and `td` is a `TableCell` instance. First, if the MenuItem's **Url** property is not an empty string, then the `td` has its **onclick** attribute set to client-side JavaScript code that will redirect the user to the specified URL. If the **Url** property is an empty string and the **CommandName** property is not, then the **onclick** event is set to cause a postback, passing in the **CommandName** property value. Finally, if both **Url** and **CommandName** are empty strings, then no **onclick** event is associated with this `td`.

Note If your server control can raise many events, the five steps discussed above can be fine tuned to provide better performance. Refer to *Developing Microsoft ASP.NET Server Controls and Components* (ISBN – 0735615829) for more information.

Conclusion

In this article we examined the source code of the classes that make up skmMenu: Menu, MenuItemCollection, and MenuItem. We saw how these classes worked in tandem to provide databinding, state management, and responding to client-side events. More importantly, we examined a number of common issues facing ASP.NET server control developers, and looked at various solutions to these problems.

As mentioned earlier, the most up to date version of skmMenu can be found at the skmMenu GotDotNet Workspace. You are more than welcome to download the compiled assemblies and complete source code, or work on the code base to help improve skmMenu.

Finally, I would like to heartily recommend Nikhil Kothari and Vandana Datye's book *Developing Microsoft ASP.NET Server Controls and Components* (ISBN – 0735615829). This book is a definite must have for all ASP.NET server control developers.

Happy Programming!

About the Author

Scott Mitchell, author of five ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies for the past five years. An active member in the ASP and ASP.NET community, Scott is passionate about ASP and ASP.NET and enjoys helping others learn more about these exciting technologies. For more on the DataGrid, DataList, and Repeater controls check out Scott's book *ASP.NET Data Web Controls Kick Start* (ISBN: 0672325012).

Top of Page