CHAPTER 13

Website Navigation

You've already learned several simple ways to send a website visitor from one page to another. The most straightforward approach is to add ordinary HTML links, using either the <a> element or the HyperLink web control. These links let users click their way through the pages of your site. Or, if you want to trigger a page change in response to some other action, your code can call the handy Response.Redirect() method or the Server.Transfer() method at any time. Both methods are detailed in Chapter 5.

All these techniques are fundamental parts of ASP.NET web design. But as your website becomes more sophisticated, it gets more-complex navigational requirements—ones that can't be satisfied with a pile of ordinary links. Instead, professional applications need a complete navigation *system* that allows users to surf through a hierarchy of pages, without forcing you to copy a large block of markup or write the same tedious navigation code in every page.

Fortunately, ASP.NET includes a navigation model that makes it easy to let users surf through your web applications. Before you can use this model, you need to determine the hierarchy of your website—in other words, how pages are logically organized into groups. You then *define* that structure in a dedicated file and *bind* that information to specialized navigation controls. Best of all, these navigation controls include nifty widgets such as the TreeView and Menu.

In this chapter, you'll learn everything you need to know about the site map model and the navigation controls that work with it.

Site Maps

If your website has more than a handful of pages, you'll probably want some sort of navigation system to let users move from one page to the next. Obviously, you can use the ASP.NET toolkit of controls to implement almost any navigation system, but this requires that *you* perform all the hard work. Fortunately, ASP.NET has a set of navigation features that can simplify the task dramatically.

As with all the best ASP.NET features, ASP.NET navigation is flexible, configurable, and pluggable. It consists of three components:

- A way to define the navigation structure of your website. This part is the XML site map, which is (by default) stored in a file.
- A convenient way to read the information in the site map file and convert it to an object model. The SiteMapDataSource control and the XmlSiteMapProvider perform this part.
- A way to use the site map information to display the user's current position and give the user the ability to easily move from one place to another. This part takes place through the navigation controls you bind to the SiteMapDataSource control, which can include breadcrumb links, lists, menus, and trees.

You can customize or extend each of these ingredients separately. For example, if you want to change the appearance of your navigation controls, you simply need to bind different controls to the SiteMapDataSource. On the other hand, if you want to read site map information from a different type of file or from a different location, you need to change your site map provider.

Figure 13-1 shows how these pieces fit together.

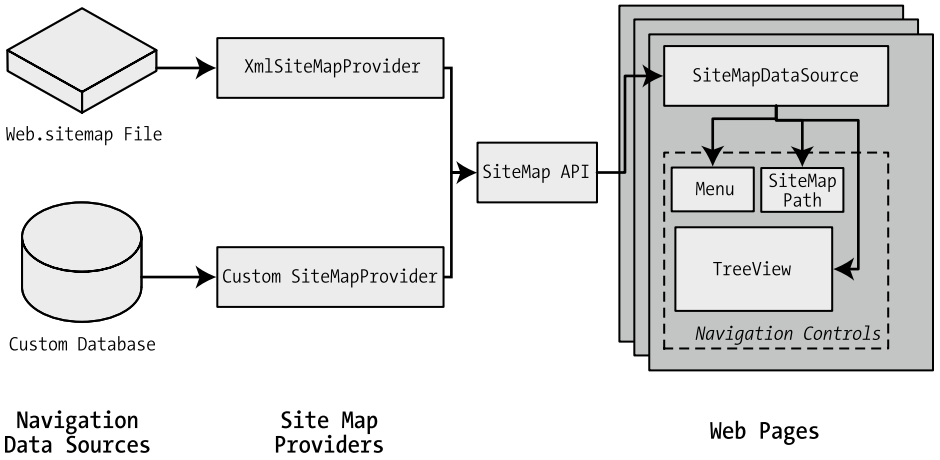

Figure 13-1. *ASP.NET navigation with site maps*

Defining a Site Map

The starting point in site map–based navigation is the site map provider. ASP.NET ships with a single site map provider, named XmlSiteMapProvider, which is able to retrieve site map information from an XML file. If you want to retrieve a site map from another location or in a custom format, you'll need to create your own site map provider or look for a third-party solution on the Web.

The XmlSiteMapProvider looks for a file named Web.sitemap in the root of the virtual directory. Like all site map providers, the task of the XmlSiteMapProvider is to extract the site map data and create the corresponding SiteMap object. This SiteMap object is then made available to the SiteMapDataSource, which you place on every page that uses navigation. The SiteMapDataSource provides the site map information to navigation controls, which are the final link in the chain.

■ **Tip** To simplify the task of adding navigation to your website, you can use master pages, as described in Chapter 12. That way, you simply need to place the SiteMapDataSource and navigation controls on the master page, rather than on all the individual pages in your website. You'll use this technique in this chapter.

You can create a site map by using a text editor such as Notepad, or you can create it in Visual Studio by choosing Website ➤ Add New Item and then choosing the Site Map option. Either way, it's up to you to enter all the site map information by hand. The only difference is that if you create it in Visual Studio, the site map will start with a basic structure that consists of three siteMap nodes.

Before you can fill in the content in your site map file, you need to understand the rules that all ASP.NET site maps must follow. The following sections break these rules down piece by piece.

■ **Note** Before you begin creating site maps, it helps to have a basic understanding of XML, the format that's used for the site map file. You should understand what an element is, how to start and end an element, and why exact capitalization is so important. If you're new to XML, you may find that it helps to refer to Chapter 18 for a quick introduction before you read this chapter.

Rule 1: Site Maps Begin with the <siteMap> Element

Every Web.sitemap file begins by declaring the <siteMap> element and ends by closing that element. You place the actual site map information between the start and end tags (where the three dots are shown here):

```
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0">
    ...
</siteMap>
```

The xmlns attribute is required, and must be entered exactly as shown here. This tells ASP.NET that the XML file uses the ASP.NET site map standard.

Rule 2: Each Page Is Represented by a <siteMapNode> Element

So, what does the site map content look like? Essentially, every site map defines an organization of web pages. To insert a page into the site map, you add the <siteMapNode> element with some basic information. Namely, you need to supply the title of the page (which appears in the navigation controls), a description (which you may or may not choose to use), and the URL (the link for the page). You add these three pieces of information by using three attributes—named title, description, and url, as shown here:

```
<siteMapNode title="Home" description="Home" url="~/default.aspx" />
```

Notice that this element ends with the characters />. This indicates it's an *empty element* that represents a start tag and an end tag in one. Empty elements (an XML concept described in Chapter 18) never contain other nodes.

Here's a complete, valid site map file that uses this page to define a website with exactly one page:

```
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0">
    <siteMapNode title="Home" description="Home" url="~/default.aspx" />
</siteMap>
```

Notice that the URL for each page begins with the ~/ character sequence. This is quite important. The ~/ characters represent the root folder of your web application. In other words, the URL ~/default.aspx points to the default.aspx file in the root folder. This style of URL isn't required, but it's strongly recommended, because it makes sure you always get the right page. If you were to simply enter the URL default.aspx without the ~/ prefix, ASP.NET would look for the default.aspx page in the *current* folder. If you have a web application with pages in more than one folder, you'll run into a problem.

For example, if the user browses into a subfolder and clicks the default.aspx link, ASP.NET will look for the default.aspx page in that subfolder instead of in the root folder. Because the default.aspx page isn't in this folder, the navigation attempt will fail with a 404 Not Found error.

Rule 3: A <siteMapNode> Element Can Contain Other <siteMapNode> Elements

Site maps don't consist of simple lists of pages. Instead, they divide pages into groups. To represent this in a site map file, you place one <siteMapNode> inside another. Instead of using the empty element syntax shown previously, you'll need to split your <siteMapNode> element into a start tag and an end tag:

```
<siteMapNode title="Home" description="Home" url="~/default.aspx">
    ...
</siteMapNode>
```

Now you can slip more nodes inside. Here's an example of a Home group that contains two more pages:

```
<siteMapNode title="Home" description="Home" url="~/default.aspx">
    <siteMapNode title="Products" description="Our products"
      url="~/products.aspx" />
    <siteMapNode title="Hardware" description="Hardware choices"
      url="~/hardware.aspx" />
</siteMapNode>
```

Essentially, this represents the hierarchical group of links shown in Figure 13-2.

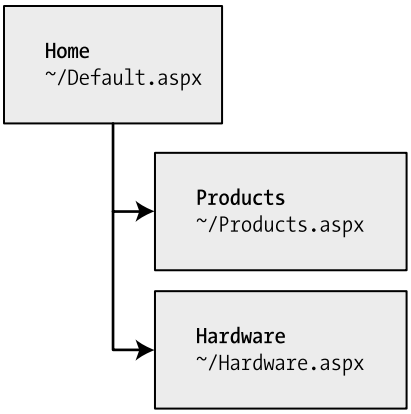

Figure 13-2. *Three nodes in a site map*

In this case, all three nodes are links. This means the user could surf to one of three pages. However, when you start to create more-complex groups and subgroups, you might want to create nodes that serve only to organize other nodes but aren't links themselves. In this case, just omit the url attribute, as shown here with the Products node:

```
<siteMapNode title="Products" description="Products">
    <siteMapNode title="In Stock" description="Products that are available"
      url="~/inStock.aspx" />
    <siteMapNode title="Not In Stock" description="Products that are on order"
      url="~/outOfStock.aspx" />
</siteMapNode>
```

When you show this part of the site map in a web page, the Products node will appear as ordinary text, not a clickable link.

No limit exists for how many layers deep you can nest groups and subgroups. However, it's a good rule to go just two or three levels deep; otherwise, it may be difficult for users to grasp the hierarchy when they see it in a navigation control. If you find that you need more than two or three levels, you may need to reconsider how you are organizing your pages into groups.

Rule 4: Every Site Map Begins with a Single <siteMapNode>

Another rule applies to all site maps. A site map must always have a single root node. All the other nodes must be contained inside this root-level node.

That means the following is *not* a valid site map, because it contains two top-level nodes:

```
<siteMapNode title="Products" description="Our products"
  url="~/products.aspx" />
<siteMapNode title="Hardware" description="Hardware choices"
  url="~/hardware.aspx" />
```

The following site map is valid, because it has a single top-level node (Home), which contains two more nodes:

```
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0">
  <siteMapNode title="Home" description="Home" url="~/default.aspx">
    <siteMapNode title="Products" description="Our products"
      url="~/products.aspx" />
    <siteMapNode title="Hardware" description="Hardware choices"
      url="~/hardware.aspx" />
  </sitemapNode>
</siteMap>
```

As long as you use only one top-level node, you can nest nodes as deep as you want in groups as large or as small as you want.

Rule 5: Duplicate URLs Are Not Allowed

You cannot create two site map nodes with the same URL. This might seem to present a bit of a problem when you want to have the same link in more than one place—and it does. However, it's a requirement because the default SiteMapProvider included with ASP.NET stores nodes in a collection, with each item indexed by its unique URL.

This limitation doesn't prevent you from creating more than one URL with minor differences pointing to the same page. For example, consider the following portion of a site map. These two nodes are acceptable, even though they lead to the same page (products.aspx), because the two URLs have different query string arguments at the end:

```
<siteMapNode title="In Stock" description="Products that are available"
  url="~/products.aspx?stock=1" />
<siteMapNode title="Not In Stock" description="Products that are on order"
  url="~/products.aspx?stock=0" />
```

This approach works well if you have a single page that will display different information, depending on the query string. Using the query string argument, you can add both "versions" of the page to the site map. Chapter 8 describes the query string in more detail.

■ **Note** The URL in the site map is not case sensitive.

Seeing a Simple Site Map in Action

A typical site map can be a little overwhelming at first glance. But if you keep the previous five rules in mind, you'll be able to sort out exactly what's taking place.

The following is an example that consists of seven nodes. (Remember, each *node* is either a link to an individual page, or a heading used to organize a group of pages.) The example defines a simple site map for a company named RevoTech.

```
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0">

    <siteMapNode title="Home" description="Home" url="~/default.aspx">

        <siteMapNode title="Information" description="Learn about our company">
            <siteMapNode title="About Us"
              description="How RevoTech was founded"
              url="~/aboutus.aspx" />
            <siteMapNode title="Investing"
              description="Financial reports and investor analysis"
              url="~/financial.aspx" />
        </siteMapNode>

        <siteMapNode title="Products" description="Learn about our products">
            <siteMapNode title="RevoStock"
              description="Investment software for stock charting"
              url="~/product1.aspx" />
            <siteMapNode title="RevoAnalyze"
              description="Investment software for yield analysis"
              url="~/product2.aspx" />
        </siteMapNode>

    </siteMapNode>

</siteMap>
```

In the following section, you'll bind this site map to the controls in a page, and you'll see its structure emerge.

Binding an Ordinary Page to a Site Map

Once you've defined the Web.sitemap file, you're ready to use it in a page. First, it's a good idea to make sure you've created all the pages that are listed in the site map file, even if you leave them blank. Otherwise, you'll have trouble testing whether the site map navigation actually works.

The next step is to add the SiteMapDataSource control to your page. You can drag and drop it from the Data tab of the Toolbox. It creates a tag like this:

```
<asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server" />
```

The SiteMapDataSource control appears as a gray box on your page in Visual Studio, but it's invisible when you run the page.

The last step is to add controls that are linked to the SiteMapDataSource. Although you can use any of the data controls described in Part 3, in practice you'll find that you'll get the results you want only with the three controls that are available in the Navigation tab of the Toolbox. That's because these controls support hierarchical

data (data with multiple nested levels), and the site map is an example of hierarchical data. In any other control, you'll see only a single level of the site map at a time, which is impractical.

These are the three navigation controls:

TreeView: The TreeView displays a "tree" of grouped links that shows your whole site map at a glance.

Menu: The Menu displays a multilevel menu. By default, you'll see only the first level, but other levels pop up (thanks to some nifty JavaScript) when you move the mouse over the subheadings.

SiteMapPath: The SiteMapPath is the simplest navigation control—it displays the full path you need to take through the site map to get to the current page. For example, it might show Home > Products > RevoStock if you're at the product1.aspx page. Unlike the other navigation controls, the SiteMapPath is useful only for moving up the hierarchy.

To connect a control to the SiteMapDataSource, you simply need to set its DataSourceID property to match the name of the SiteMapDataSource. For example, if you added a TreeView, you should tweak the tag so it looks like this:

```
<asp:TreeView ID="TreeView1" runat="server" DataSourceID="SiteMapDataSource1" />
```

Figure 13-3 shows the result—a tree that displays the structure of the site, as defined in the website. (The text at the bottom is from the current page—in this case, default.aspx, which contains the site map.)

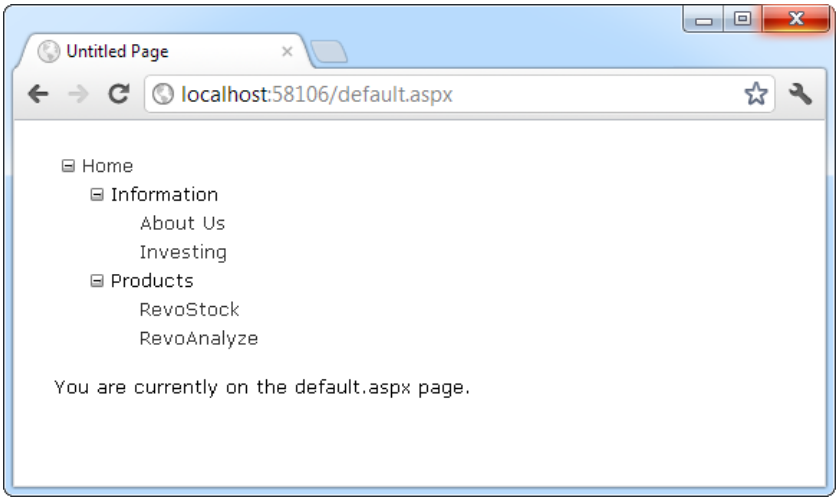

Figure 13-3. *A site map in the TreeView*

When using the TreeView, the description information doesn't appear immediately. Instead, it's displayed as a tooltip when you hover over an item in the tree.

Best of all, this tree is created automatically. As long as you link it to the SiteMapDataSource control, you don't need to write any code.

When you click one of the nodes in the tree, you'll automatically be taken to the page you defined in the URL. Of course, unless that page also includes a navigation control such as the TreeView, the site map will disappear from sight. The next section shows a better approach.

Binding a Master Page to a Site Map

Website navigation works best when combined with another ASP.NET feature—master pages. That's because you'll usually want to show the same navigation controls on every page. The easiest way to do this is to create a master page that includes the SiteMapDataSource and the navigation controls. You can then reuse this template for every other page on your site.

Here's how you might define a basic structure in your master page that puts navigation controls on the left:

```
<%@ Master Language="C#" AutoEventWireup="true"
  CodeFile="MasterPage.master.cs" Inherits="MasterPage" %>
<html>
<head runat="server">
  <title>Navigation Test</title>
</head>
<body>
<form id="form1" runat="server">
  <table>
    <tr>
      <td style="width: 226px;vertical-align: top;">
        <asp:TreeView ID="TreeView1" runat="server"
         DataSourceID="SiteMapDataSource1" />
      </td>
      <td style="vertical-align: top;">
        <asp:ContentPlaceHolder id="ContentPlaceHolder1" runat="server" />
      </td>
    </tr>
  </table>
  <asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server" />
</form>
</body>
</html>
```

Then create a child with some simple static content. Here's the code for the default.aspx page in this example:

```
<%@ Page Language="C#" MasterPageFile="~/MasterPage.master"
AutoEventWireup="true"
  CodeFile="default.aspx.cs" Inherits="_default" Title="Home Page" %>
<asp:Content ID="Content1" ContentPlaceHolderID="ContentPlaceHolder1"
  runat="Server">
    <br />
    <br />
    You are currently on the default.aspx page (Home).
</asp:Content>
```

In fact, while you're at it, why not create a second page so you can test the navigation between the two pages?

```
<%@ Page Language="C#" MasterPageFile="~/MasterPage.master"
AutoEventWireup="true" CodeFile="product1.aspx.cs"
Inherits="product1" Title=" RevoStock Page" %>
<asp:Content ID="Content1" ContentPlaceHolderID="ContentPlaceHolder1"
  runat="Server">
    <br />
    <br />
    You are currently on the product1.aspx page (RevoStock).
</asp:Content>
```

Now you can jump from one page to another by using the TreeView (see Figure 13-4). The first picture shows the home page as it initially appears, while the second shows the result of clicking the RevoStock link in the TreeView. Because both pages use the same master, and the master page includes the TreeView, the site map always remains visible.

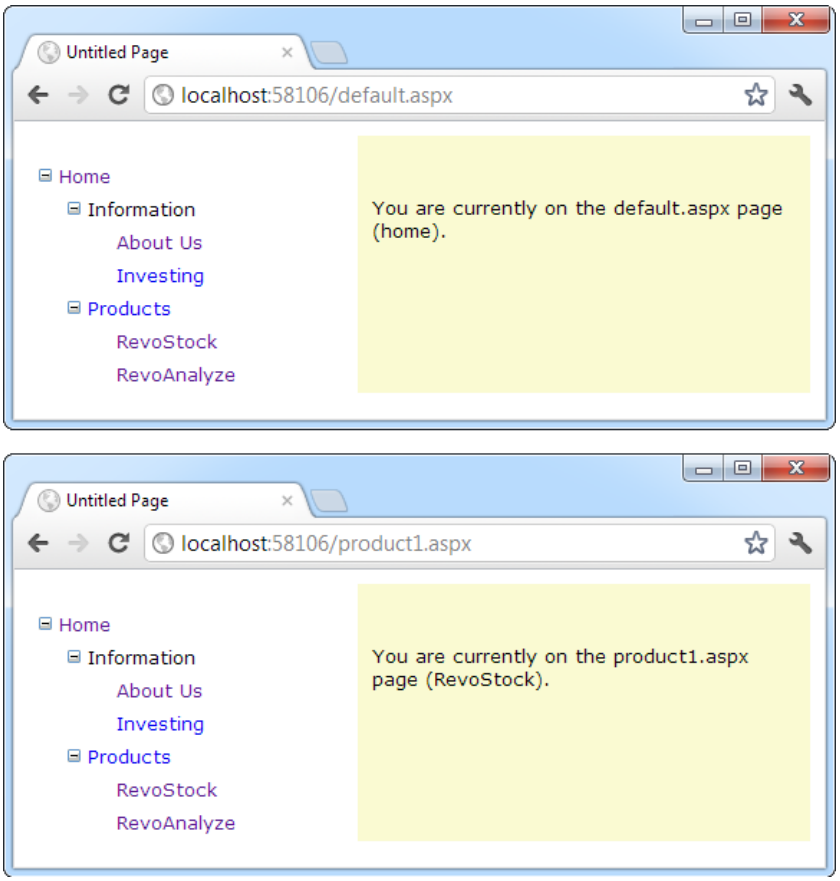

Figure 13-4. *Navigating from page to page with the TreeView*

You can do a lot more to customize the appearance of your pages and navigation controls. You'll consider these topics in the following sections.

Binding Portions of a Site Map

In the previous example, the TreeView shows the structure of the site map file *exactly*. However, this isn't always what you want. For example, you might not like the way the Home node sticks out because of the XmlSiteMapProvider rule that every site map must begin with a single root.

One way to clean this up is to configure the properties of the SiteMapDataSource. For example, you can set the ShowStartingNode property to false to hide the root node:

```
<asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server"
  ShowStartingNode="False" />
```

Figure 13-5 shows the result.

Figure 13-5. *A site map without the root node*

This example shows how you can hide the root node. Another option is to show just a portion of the complete site map, starting from the current node. For example, you might use a control such as the TreeView to show everything in the hierarchy starting from the current node. A user who wants to move up a level could use another control (such as the SiteMapPath).

Showing Subtrees

By default, the SiteMapDataSource shows a full tree that starts with the root node. However, the SiteMapDataSource has several properties that can help you configure the navigation tree to limit the display to just a specific branch. Typically, this is useful if you have a deeply nested tree. Table 13-1 describes the full set of properties.

Table 13-1. *SiteMapDataSource Properties*

Property	Description
ShowStartingNode	Set this property to false to hide the first (top-level) node that would otherwise appear in the navigation tree. The default is true.
StartingNodeUrl	Use this property to change the starting node. Set this value to the URL of the node that should be the first node in the navigation tree. This value must match the url attribute in the site map file exactly. For example, if you specify a StartingNodeUrl of "~/home.aspx", then the first node in the tree is the Home node, and you will see nodes only underneath that node.
StartFromCurrentNode	Set this property to true to set the current page as the starting node. The navigation tree will show only pages beneath the current page (which allows the user to move down the hierarchy). If the current page doesn't exist in the site map file, this approach won't work.

(*continued*)

Table 13-1. (*continued*)

Property	Description
StartingNodeOffset	Use this property to shift the starting node up or down the hierarchy. It takes an integer that instructs the SiteMapDataSource to move from the starting node down the tree (if the number is positive) or up the tree (if the number is negative). The actual effect depends on how you combine this property with other SiteMapDataSource properties. For example, if StartFromCurrentNode is false, you'll use a positive number to move down the tree, from the starting node toward the current node. If StartFromCurrentNode is true, you'll use a negative number to move up the tree, away from the current node and toward the starting node.

Figuring out these properties can take some work, and you might need to do a bit of experimenting to decide the right combination of SiteMapDataSource settings you want to use. To make matters more interesting, you can use more than one SiteMapDataSource on the same page. This means you could use two navigation controls to show different sections of the site map hierarchy.

Before you can see this in practice, you need to modify the site map file used for the previous few examples into something a little more complex. Currently, the site map has three levels, but only the first level (the Home node) and the third level (the individual pages) have URL links. The second-level groupings (Information and Products) are just used as headings, not links. To get a better feel for how the SiteMapDataSource properties work with multiple navigation levels, modify the Information node as shown here:

```
<siteMapNode title="Information" description="Learn about our company"
  url="~/information.aspx">
```

and change the Products node:

```
<siteMapNode title="Products" description="Learn about our products"
  url="~/products.aspx">
```

Next create the products.aspx and information.aspx pages.

The interesting feature of the Products node is that not only is it a navigable page, but it's a page that has other pages both above it and below it in the navigation hierarchy. This makes it ideal for testing the SiteMapDataSource properties. For example, you can create a SiteMapDataSource that shows only the current page and the pages below it, like this:

```
<asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server"
   StartFromCurrentNode="True" />
```

And you can create one that always shows the Information page and the pages underneath it, like this:

```
<asp:SiteMapDataSource ID="SiteMapDataSource2" runat="server"
  StartingNodeUrl="~/information.aspx" />
```

■ **Note** For this technique to work, ASP.NET must be able to find a page in the Web.sitemap file that matches the current URL. Otherwise, ASP.NET won't know where the current position is, and it won't provide any navigation information to the bound controls.

Now just bind two navigation controls. In this case, one TreeView is linked to each SiteMapDataSource in the markup for the master page:

```
Pages under the current page:
<asp:TreeView ID="TreeView1" runat="server"
  DataSourceID="SiteMapDataSource1" />
<br />
The Information group of pages:<br />
<asp:TreeView ID="TreeView2" runat="server"
  DataSourceID="SiteMapDataSource2" />
```

Figure 13-6 shows the result as you navigate from default.aspx down the tree to products1.aspx. The first TreeView shows the portion of the tree under the current page, and the second TreeView is always fixed on the Information group.

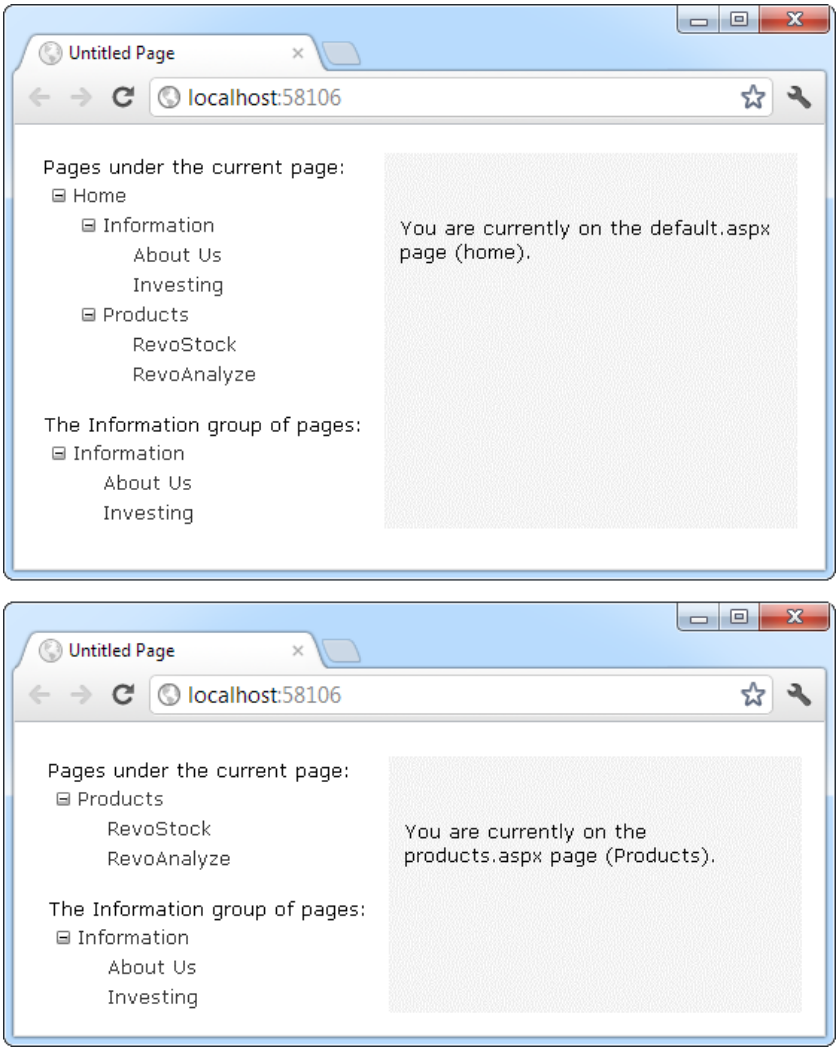

Figure 13-6. *Showing portions of the site map*

You'll need to get used to the SiteMapDataSource.StartingNodeOffset property. It takes an integer that instructs the SiteMapDataSource to move that many levels down the tree (if the number is positive) or up the tree (if the number is negative). An important detail that's often misunderstood is that when the SiteMapDataSource moves down the tree, it moves *toward* the current node. If it's already at the current node, or your offset takes it beyond the current node, the SiteMapDataSource won't know where to go, and you'll end up with a blank navigation control.

To understand how this works, it helps to consider an example. Imagine you're at this location in a website:

```
Home > Products > Software > Custom > Contact Us
```

If the SiteMapDataSource is starting at the Home node (the default) and you apply a StartingNodeOffset of 2, it will move down the tree two levels and bind to the tree of pages that starts at the Software node.

On the other hand, if you're currently at the Products node, you won't see anything. That's because the starting node is Home, and the offset tries to move it down two levels. However, you're only one level deep in the hierarchy. Or, to look at it another way, no node exists between the top node and the current node that's two levels deep.

Now, what happens if you repeat the same test but set the site map provider to begin on another node? Consider what happens if you set StartFromCurrentNode to true and surf to the Contact Us page. Once again, you won't see any information, because the site map provider attempts to move two levels down from the current node—Contact Us—and it has nowhere to go. On the other hand, if you set StartFromCurrentNode to true and use a StartingNodeOffset of –2, the SiteMapDataSource will move *up* two levels from Contact Us and bind the subtree starting at Software.

Overall, you won't often use the StartingNodeOffset property. However, it can be useful if you have a deeply nested site map and you want to keep the navigation display simple by showing just a few levels up from the current position.

■ **Note** All the examples in this section filtered out higher-level nodes than the starting node. For example, if you're positioned at the Home > Products > RevoStock page, you've seen how to hide the Home and Products levels. You haven't seen how to filter out lower-level nodes. For example, if you're positioned at the Home page, you'll always see the full site map, because you don't have a way to limit the number of levels you see below the starting node. You have no way to change this behavior with the SiteMapDataSource; but later, in "The TreeView Control" section, you'll see that the TreeView.MaxDataBindDepth property serves this purpose.

Using Different Site Maps in the Same File

Imagine you want to have a dealer section and an employee section on your website. You might split this into two structures and define them both under different branches in the same file, like this:

```
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0">
  <siteMapNode title="Root" description="Root" url="~/default.aspx">
    <siteMapNode title="Dealer Home" description="Dealer Home"
     url="~/default_dealer.aspx">
      ...
    </siteMapNode>
    <siteMapNode title="Employee Home" description="Employee Home"
      url="~/default_employee.aspx">
      ...
    </siteMapNode>
  </siteMapNode>
</siteMap>
```

To bind the SiteMapDataSource to the dealer view (which starts at the Dealer Home page), you simply set the StartingNodeUrl property to "~/default_dealer.aspx". You can do this programmatically or, more likely, by creating an entirely different master page and implementing it in all your dealer pages. In your employee pages, you set the StartingNodeUrl property to "~/default_employee.aspx". This way, you'll show only the pages under the Employee Home branch of the site map.

You can even make your life easier by using the siteMapFile attribute to break a single site map into separate files, like this:

```
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0">
  <siteMapNode title="Root" description="Root" url="~/default.aspx">
    <siteMapNode siteMapFile="Dealers.sitemap" />
    <siteMapNode siteMapFile="Employees.sitemap" />
  </siteMapNode>
</siteMap>
```

Even with this technique, you're still limited to a single site map tree, and it always starts with the Web.sitemap file. But you can manage your site map more easily because you can factor some of its content into separate files.

However, this seemingly nifty technique is greatly limited because the site map provider doesn't allow duplicate URLs. This means you have no way to reuse the same page in more than one branch of a site map. Although you can try to work around this problem by creating different URLs that are equivalent (for example, by adding query string parameters on the end), this raises more headaches. Sadly, this problem has no solution with the default site map provider that ASP.NET includes.

Working with the SiteMap Class

You aren't limited to no-code data binding in order to display navigation hierarchies. You can interact with the navigation information programmatically. This allows you to retrieve the current node information and use it to configure details such as the page heading and title. All you need to do is interact with the objects that are readily available through the Page class.

The site map API is remarkably straightforward. To use it, you need to work with two classes from the System.Web namespace. The starting point is the SiteMap class, which provides the static properties CurrentNode (the site map node representing the current page) and RootNode (the root site map node). Both of these properties return a SiteMapNode object. Using the SiteMapNode object, you can retrieve information from the site map, including the title, description, and URL values. You can branch out to consider related nodes by using the navigation properties in Table 13-2.

Table 13-2. *SiteMapNode Navigation Properties*

Property	Description
ParentNode	Returns the node one level up in the navigation hierarchy, which contains the current node. On the root node, this returns a null reference.
ChildNodes	Provides a collection of all the child nodes. You can check the HasChildNodes property to determine whether child nodes exist.
PreviousSibling	Returns the previous node that's at the same level (or a null reference if no such node exists).
NextSibling	Returns the next node that's at the same level (or a null reference if no such node exists).

■ **Note** You can also search for nodes by using the methods of the current SiteMapProvider object, which is available through the SiteMap.Provider static property. For example, the SiteMap.Provider.FindSiteMapNode() method allows you to search for a node by its URL.

To see this in action, consider the following code, which configures two labels on a page to show the heading and description information retrieved from the current node:

```
protected void Page_Load(object sender, EventArgs e)
{
    lblHead.Text = SiteMap.CurrentNode.Title;
    lblDescription.Text = SiteMap.CurrentNode.Description;
}
```

If you're using master pages, you could place this code in the code-behind for your master page, so that every page is assigned its title from the site map.

The next example is a little more ambitious. It implements a Next link, which allows the user to traverse an entire set of subnodes. The code checks for the existence of sibling nodes, and if there aren't any in the required position, it simply hides the link:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (SiteMap.CurrentNode.NextSibling != null)
    {
        lnkNext.NavigateUrl = SiteMap.CurrentNode.NextSibling.Url;
        lnkNext.Visible = true;
    }
    else
    {
        lnkNext.Visible = false;
    }
}
```

Figure 13-7 shows the result. The first picture shows the Next link on the product1.aspx page. The second picture shows how this link disappears when you navigate to product2.aspx (either by clicking the Next link or the RevoAnalyze link in the TreeView).

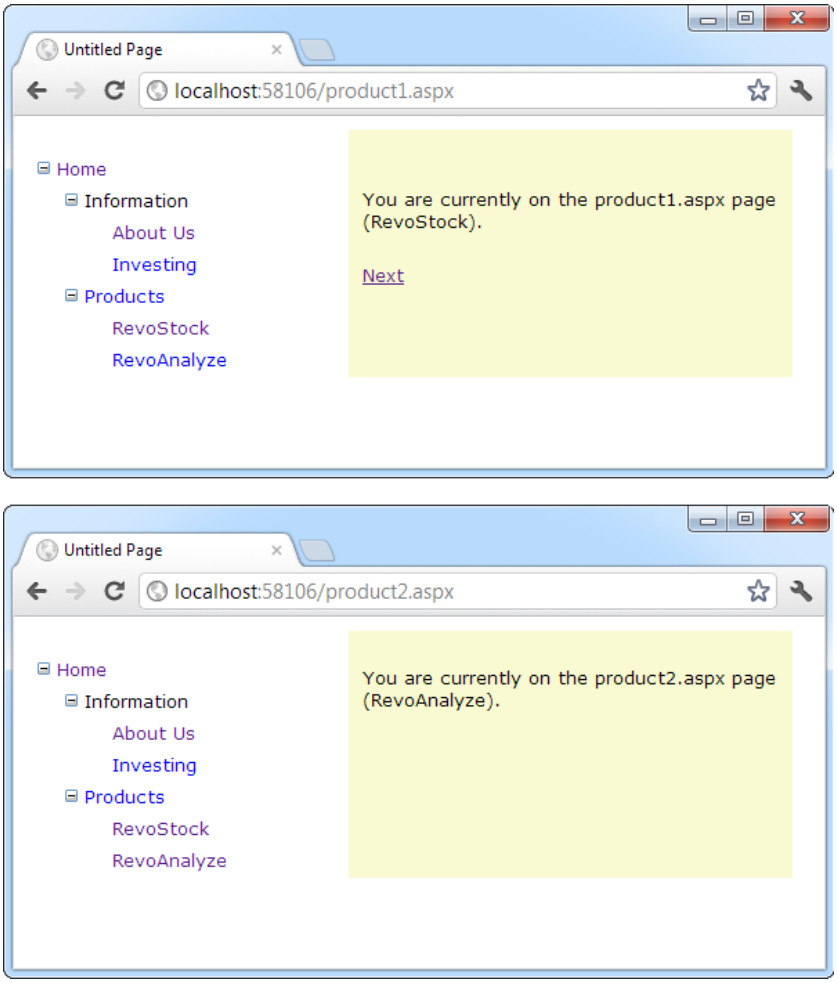

Figure 13-7. *Creating a Next page link*

URL Mapping and Routing

The site map model is designed around a simple principle: each entry has a separate URL. Although you can distinguish URLs by adding query string arguments, in many websites there is one-to-one correspondence between web forms and site map entries.

When this doesn't suit, ASP.NET has two tools that may be able to help you. The first is URL mapping, which is a clean, no-nonsense way to map one URL to another. The second is URL routing, which is a slightly more elaborate but much more flexible system that performs the same task. URL mapping is an ideal way to deal with "one-off" redirection. For example, mapping is a quick way to deal with old or recently moved pages or to allow extra entry points for a few popular pages. On the other hand, URL routing can serve as the basis for a more sophisticated redirection system that deals with many more pages. For example, you could use it to replace long, complex product page URLs with a simpler syntax and implement that across your entire website. Routing is particularly useful if you want to offer cleaner URLs so that search engines can index your website more easily and comprehensively.

URL Mapping

In some situations, you might want to have several URLs lead to the same page. This might be the case for a number of reasons—maybe you want to implement your logic in one page and use query string arguments but still provide shorter and easier-to-remember URLs to your website users (often called *friendly* URLs). Or maybe you have renamed a page, but you want to keep the old URL functional so it doesn't break user bookmarks. Although web servers sometimes provide this type of functionality, ASP.NET includes its own URL-mapping feature.

The basic idea behind ASP.NET URL mapping is that you map a request URL to a different URL. The mapping rules are stored in the web.config file, and they're applied before any other processing takes place. Of course, for ASP.NET to apply the remapping, it must be processing the request, which means the request URL must use a file type extension that's mapped to ASP.NET (such as .aspx).

You define URL mapping in the <urlMappings> section of the web.config file. You supply two pieces of information—the request URL (as the url attribute) and the new destination URL (mappedUrl). Here's an example:

```
<configuration>
  <system.web>
    <urlMappings enabled="true">
      <add url="~/category.aspx"
       mappedUrl="~/default.aspx?category=default" />
      <add url="~/software.aspx"
       mappedUrl="~/default.aspx?category=software" />
    </urlMappings>
    ...
  </system.web>
</configuration>
```

In order for ASP.NET to make a match, the URL that the browser submits must match the URL specified in the web.config file almost exactly. However, there are two exceptions. First, the matching algorithm isn't case sensitive, so the capitalization of the request URL is always ignored. Second, any query string arguments in the URL are disregarded. Unfortunately, ASP.NET doesn't support advanced matching rules, such as wildcards or regular expressions.

When you use URL mapping, the redirection takes place in the same way as the Server.Transfer() method, which means no round-trip happens and the URL in the browser will still show the original request URL, not the new page. In your code, the Request.Path and Request.QueryString properties reflect the new (mapped) URL. The Request.RawUrl property returns the original, friendly request URL.

This can introduce some complexities if you use it in conjunction with site maps—namely, does the site map provider try to use the original request URL or the destination URL when looking for the current node in the site map? The answer is both. It begins by trying to match the request URL (provided by the Request.RawUrl property), and if no value is found, it then uses the Request.Path property instead. This is the behavior of the XmlSiteMapProvider, so you could change it in a custom provider if desired.

URL Routing

URL routing was originally designed as a core part of ASP.NET MVC, an alternative framework for building web pages that doesn't use the web form features discussed in this book. However, the creators of ASP.NET realized that routing could also help web form developers tame sprawling sites and replace convoluted URLs with cleaner alternatives (which makes it easier for people to type them in and for search engines to index them). For all these reasons, they made the URL-routing feature available to ordinary ASP.NET web form applications.

■ **Note** To learn more about ASP.NET MVC, which presents a dramatically different way to think about rendering web pages, check out *Pro ASP.NET MVC 4* (Apress).

Unlike URL mapping, URL routing doesn't take place in the web.config file. Instead, it's implemented using code. Typically, you'll use the Application_Start() method in the global.asax file to register all the routes for your application.

To register a route, you use the RouteTable class from the System.Web.Routing namespace. To make life easier, you can start by importing that namespace:

```
using System.Web.Routing;
```

The RouteTable class provides a static property named Routes, which holds a collection of Route objects that are defined for your application. Initially, this collection is empty, but you can create custom routes by calling the MapPageRoute() method, which takes three arguments:

routeName: This is a name that uniquely identifies the route. It can be whatever you want.

routeUrl: This specifies the URL format that browsers will use. Typically, a route URL consists of one or more pieces of variable information, separated by slashes, which are extracted and provided to your code. For example, you might request a product page by using a URL such as */products/4312*.

physicalFile: This is the target web form—the place where users will be redirected when they use the route. The information from the original routeUrl will be parsed and made available to this page as a collection through the Page.RouteData property.

Here's an example that adds two routes to a web application when it first starts:

```
protected void Application_Start(object sender, EventArgs e)
{
    RouteTable.Routes.MapPageRoute("product-details",
      "product/{productID}", "~/productInfo.aspx");
    RouteTable.Routes.MapPageRoute("products-in-category",
      "products/category/{categoryID}", "~/products.aspx");
}
```

The route URL can include one or more parameters, represented by a placeholder in curly brackets. For example, the first route shown here includes a parameter named productID. This piece of information will be pulled out of the URL and passed along to the target page.

Here's a URL that uses this route to request a product with the ID FI_00345:

```
http://localhost:[PortNumber]/Routing/product/FI_00345
```

The ASP.NET routing infrastructure then redirects the user to the productInfo.aspx page. All the parameters are provided through the Page.RouteData property. Technically, Page.RouteData provides a RouteData object. Its most useful property is the Values collection, which provides all the parameters from the original request, indexed by name.

Here's an example that shows how the productInfo.aspx page can retrieve the requested product ID from the original URL:

```
protected void Page_Load(object sender, EventArgs e)
{
    string productID = (string)Page.RouteData.Values["productID"];
    lblInfo.Text = "You requested " + productID;
}
```

Similarly, the second route in this example accepts URLs in this form:

```
http://localhost:[PortNumber]/Routing/products/category/342
```

Although you can hard-code this sort of URL, there's a Page.GetRouteUrl() helper method that does it for you automatically, avoiding potential mistakes. Here's an example that looks up a route (using its registered name), supplies the parameter information, and then retrieves the corresponding URL.

```
hyperLink.NavigateUrl = Page.GetRouteUrl("product-details", new {productID = "FI_00345" });
```

The slightly strange syntax that passes the parameter information uses a language feature called *anonymous types*. It allows you to supply as few or as many parameters as you want. Technically, the C# compiler automatically creates a class that includes all the parameters you supply and submits that object to the GetRouteUrl() method. The final result is a routed URL that points to the FI_00345 product, as shown in the first example.

The SiteMapPath Control

The TreeView shows the available pages, but it doesn't indicate where you're currently positioned. To solve this problem, it's common to use the TreeView in conjunction with the SiteMapPath control. Because the SiteMapPath is always used for displaying navigation information (unlike the TreeView, which can also show other types of data), you don't even need to explicitly link it to the SiteMapDataSource:

```
<asp:SiteMapPath ID="SiteMapPath1" runat="server" />
```

The SiteMapPath provides *breadcrumb navigation*, which means it shows the user's current location and allows the user to navigate up the hierarchy to a higher level by using links. Figure 13-8 shows an example with a SiteMapPath control when the user is on the product1.aspx page. Using the SiteMapPath control, the user can return to the default.aspx page. (If a URL were defined for the Products node, you would also be able to click that portion of the path to move to that page.) Once again, the SiteMapPath has been added to the master page, so it appears on all the content pages in your site.

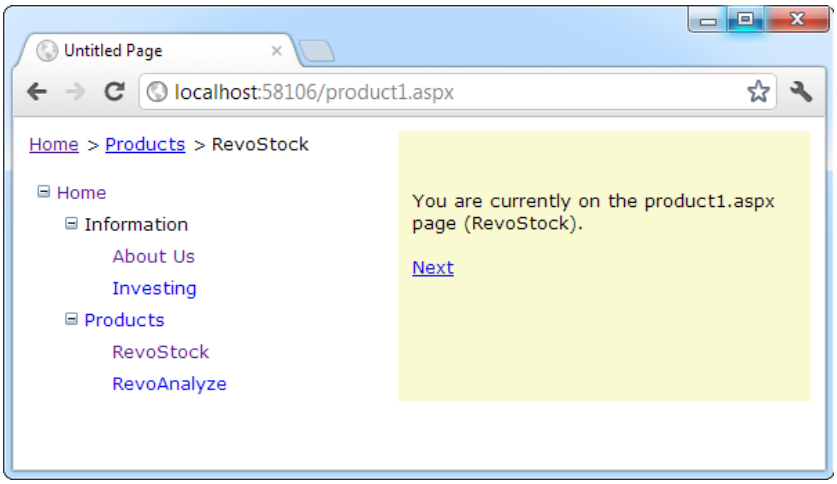

Figure 13-8. *Breadcrumb navigation with SiteMapPath*

The SiteMapPath control is useful because it provides both an at-a-glance view that shows the current position and a way to move up the hierarchy. However, you always need to combine it with other navigation controls that let the user move down the site map hierarchy.

Customizing the SiteMapPath

The SiteMapPath has a subtle but important difference from other navigation controls such as the TreeView and Menu. Unlike these controls, the SiteMapPath works directly with the ASP.NET navigation model—in other words, it doesn't need to get its data through the SiteMapDataSource. As a result, you can use the SiteMapPath on pages that don't have a SiteMapDataSource, and changing the properties of the SiteMapDataSource won't affect the SiteMapPath. However, the SiteMapPath control provides quite a few properties of its own that you can use for customization. Table 13-3 lists some of its most commonly configured properties.

Table 13-3. *SiteMapPath Appearance-Related Properties*

Property	Description
ShowToolTips	Set this to false if you don't want the description text to appear when the user hovers over a part of the site map path.
ParentLevelsDisplayed	This sets the maximum number of levels above the current page that will be shown at once. By default, this setting is –1, which means all levels will be shown.
RenderCurrentNodeAsLink	If true, the portion of the page that indicates the current page is turned into a clickable link. By default, this is false because the user is already at the current page.
PathDirection	You have two choices: RootToCurrent (the default) and CurrentToRoot (which reverses the order of levels in the path).
PathSeparator	This indicates the characters that will be placed between each level in the path. The default is the greater-than symbol (>). Another common path separator is the colon (:).

Using SiteMapPath Styles and Templates

For even more control, you can configure the SiteMapPath control with styles or even redefine the controls and HTML with templates. Table 13-4 lists all the styles and templates that are available in the SiteMapPath control; and you'll see how to use both sets of properties in this section.

Table 13-4. *SiteMapPath Styles and Templates*

Style	Template	Applies To
NodeStyle	NodeTemplate	All parts of the path except the root and current node.
CurrentNodeStyle	CurrentNodeTemplate	The node representing the current page.
RootNodeStyle	RootNodeTemplate	The node representing the root. If the root node is the same as the current node, the current node template or styles are used.
PathSeparatorStyle	PathSeparatorTemplate	The separator between each node.

Styles are easy enough to grasp—they define formatting settings that apply to one part of the SiteMapPath control. Templates are a little trickier, because they rely on data-binding expressions. Essentially, a *template* is a bit of HTML (that you create) that will be shown for a specific part of the SiteMapPath control. For example, if you want to configure how the root node displays in a site map, you could create a SiteMapPath with <RootNodeTemplate> as follows:

```
<asp:SiteMapPath ID="SiteMapPath1" runat="server">
  <RootNodeTemplate>
    <b>Root</b>
  </RootNodeTemplate>
</asp:SiteMapPath>
```

This simple template does not use the title and URL information in the root node of the sitemap node. Instead, it simply displays the word *Root* in bold. Clicking the text has no effect.

Usually, you'll use a data-binding expression to retrieve some site map information—chiefly, the description, text, or URL that's defined for the current node in the site map file. Chapter 15 covers data-binding expressions in detail, but this section will present a simple example that shows you all you need to know to use them with the SiteMapPath.

Imagine you want to change how the current node is displayed so that it's shown in italics. To get the name of the current node, you need to write a data-binding expression that retrieves the title. This data-binding expression is bracketed between <%# and %> characters and uses a method named Eval() to retrieve information from a SiteMapNode object that represents a page. Here's what the template looks like:

```
<asp:SiteMapPath ID="SiteMapPath1" runat="server">
  <CurrentNodeTemplate>
    <i><%# Eval("Title") %></i>
  </CurrentNodeTemplate>
</asp:SiteMapPath>
```

Data binding also gives you the ability to retrieve other information from the site map node, such as the description. Consider the following example:

```
<asp:SiteMapPath ID="SiteMapPath1" runat="server">
  <PathSeparatorTemplate>
    <asp:Image ID="Image1" ImageUrl="~/arrowright.gif"
     runat="server" />
  </PathSeparatorTemplate>
  <RootNodeTemplate>
    <b>Root</b>
  </RootNodeTemplate>
  <CurrentNodeTemplate>
    <%# Eval("Title") %> <br />
    <small><i><%# Eval("Description") %></i></small>
  </CurrentNodeTemplate>
</asp:SiteMapPath>
```

This SiteMapPath uses several templates. First it uses the PathSeparatorTemplate to define a custom arrow image that's used between each part of the path. This template uses an Image control instead of an ordinary HTML <img> tag because only the Image understands the ~/ characters in the image URL, which represent the application's root folder. If you don't include these characters, the image won't be retrieved successfully if you place your page in a subfolder.

Next the SiteMapPath uses the RootNodeTemplate to supply a fixed string of bold text for the root portion of the site map path. Finally, the CurrentNodeTemplate uses two data-binding expressions to show two pieces of information—both the title of the node and its description (in smaller text, underneath). Figure 13-9 shows the final result.

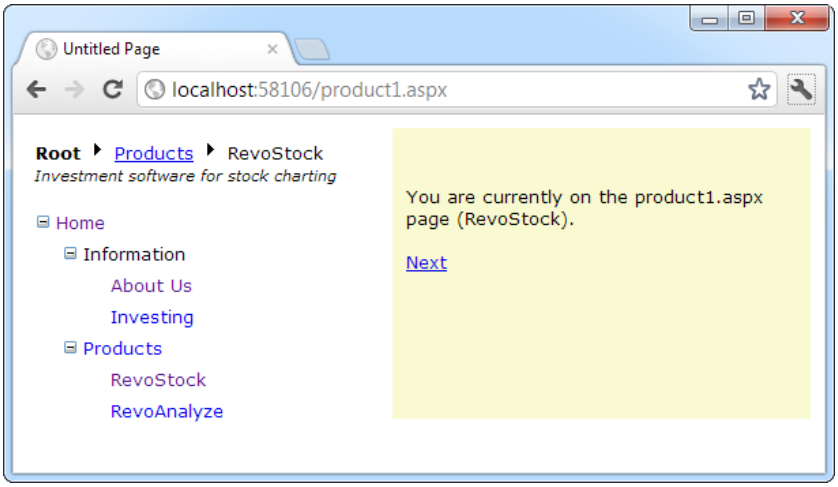

Figure 13-9. *A SiteMapPath with templates*

Keen eyes will notice that the template-based SiteMapPath not only shows more information but also is more interactive. Now you can click any of the page elements that fall between the root item and the current page. In Figure 13-9, that means you can click Products to move up a level to the products.aspx page.

Interestingly, the templates in the SiteMapPath don't contain any elements that provide these links. Instead, the SiteMapPath automatically determines what items should be clickable (by checking if they're linked to a page in the site map). If an item should be clickable, the SiteMapPath wraps the entire CurrentNodeTemplate for that item inside a link.

If you don't want links (or you want to link in a different way, or with a different control), you can change this behavior. The trick is to modify the NodeTemplate. You'll learn how to do this in the next section.

Adding Custom Site Map Information

In the site maps you've seen so far, the only information that's provided for a node is the title, description, and URL. This is the bare minimum of information you'll want to use. However, the schema for the XML site map is open, which means you're free to insert custom attributes with your own data.

You might want to insert additional node data for a number of reasons. This additional information might be descriptive information that you intend to display, or contextual information that describes how the link should work. For example, you could add an attribute specifying that the link should open in a new window. The only catch is that it's up to you to act on the information later. In other words, you need to configure your user interface so it uses this extra information.

For example, the following code shows a site map that uses a target attribute to indicate the frame where the link should open. In this example, one link is set with a target of _blank so it will open in a new browser window:

```
<siteMapNode title="RevoStock"
  description="Investment software for stock charting"
  url="~/product1.aspx" target="_blank" />
```

Now in your code, you have several options. If you're using a template in your navigation control, you can bind directly to the new attribute. Here's an example with the SiteMapPath from the previous section:

```
<asp:SiteMapPath ID="SiteMapPath1" runat="server" Width="264px" Font-Size="10pt">
    <NodeTemplate>
      <a href='<%# Eval("Url") %>' target='<%# Eval("[target]") %>'>
        <%# Eval("Title") %>
      </a>
    </NodeTemplate>
</asp:SiteMapPath>
```

This creates a link that uses the node URL (as usual) but also uses the target information. There's a slightly unusual detail in this example—the square brackets around the word [target]. You need to use this syntax to look up any custom attribute you add to the Web.sitemap file. That's because this value can't be retrieved directly from a property of the SiteMapNode class—instead, you need to use the SiteMapNode indexer to look it up by name.

If your navigation control doesn't support templates, you'll need to find another approach. For example, the TreeView doesn't support templates, but it fires a TreeNodeDataBound event each time an item is bound to the tree. You can react to this event to customize the current item. To apply the new target, use this code:

```
protected void TreeView1_TreeNodeDataBound(object sender, TreeNodeEventArgs e)
{
    SiteMapNode node = (SiteMapNode)e.Node.DataItem;
    e.Node.Target = node["target"];
}
```

As in the template, you can't retrieve the custom attribute from a strongly typed SiteMapNode property. Instead, you use the SiteMapNode indexer to retrieve it by name.

The TreeView Control

You've already seen the TreeView at work for displaying navigation information. As you've learned, the TreeView can show a portion of the full site map or the entire site map. Each node becomes a link that, when clicked, takes the user to the new page. If you hover over a link, you'll see the corresponding description information appear in a tooltip.

In the following sections, you'll learn how to change the appearance of the TreeView. In later chapters, you'll learn how to use the TreeView for other tasks, such as displaying data from a database.

■ **Note** The TreeView is one of the most impressive controls in ASP.NET. Not only does it allow you to show site maps, but it also supports showing information from a database and filling portions of the tree on demand (and without refreshing the entire page). But most important, it supports a wide range of styles that can transform its appearance.

TreeView Properties

The TreeView has a slew of properties that let you change how it's displayed on the page. One of the most important properties is ImageSet, which lets you choose a predefined set of node icons. (Each set includes three icons: one for collapsed nodes, one for expanded nodes, and one for nodes that have no children and therefore can't be expanded or collapsed.) The TreeView offers 16 possible ImageSet values, which are represented by the TreeViewImageSet enumeration.

For example, Figure 13-10 shows the same RevoStock navigation page you considered earlier, but this time with an ImageSet value of TreeViewImageSet.Faq. The result is help-style icons that show a question mark (for nodes that have no children) or a question mark superimposed over a folder (for nodes that do contain children).

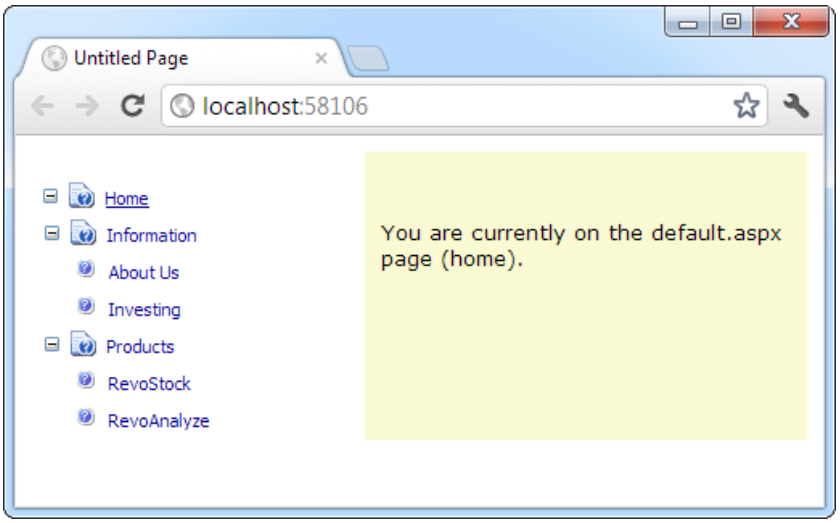

Figure 13-10. *A TreeView with fancy node icons*

You'll notice that this TreeView makes one more change. It removes the indentation between different levels of nodes, so all the sitemap entries fit in the same narrow column, no matter how many levels deep they are. This is accomplished by setting the NodeIndent property of the TreeView to 0.

Here's the complete TreeView markup:

```
<asp:TreeView ID="TreeView1" runat="server"
  DataSourceID="SiteMapDataSource1" ImageSet="Faq" NodeIndent="0" >
</asp:TreeView>
```

The TreeViewImageSet values are useful if you don't have a good set of images handy. Figure 13-11 shows a page with several TreeViews, each of which represents one of the options in the Auto Format window.

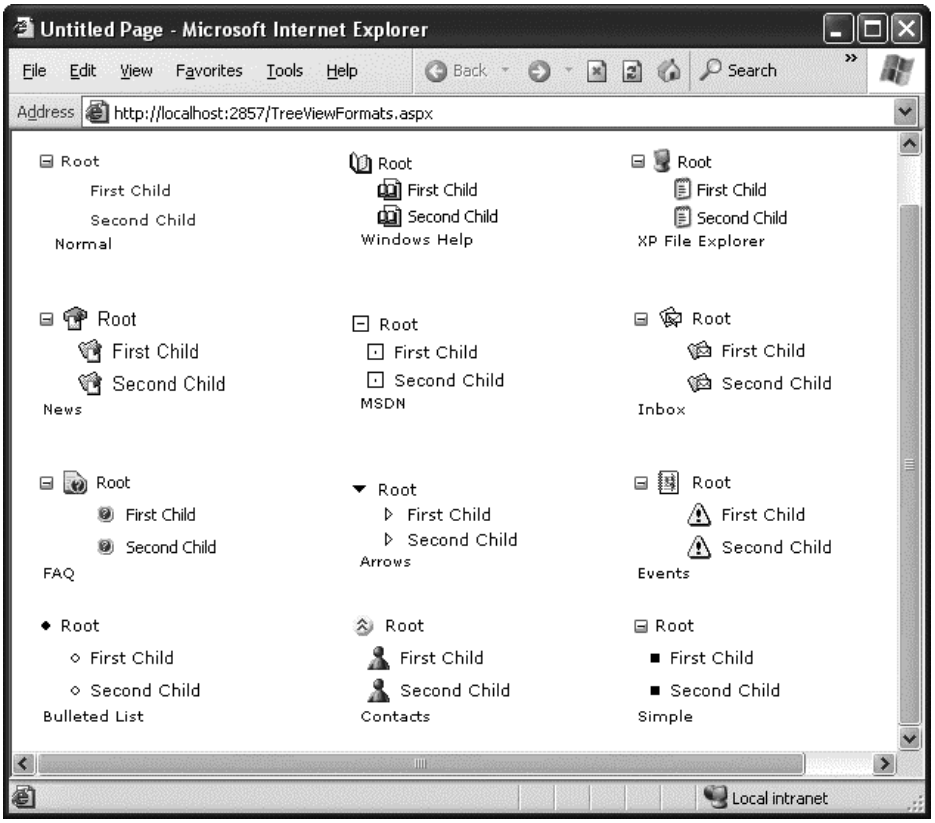

Figure 13-11. *Different looks for a TreeView*

Although the ImageSet and NodeIndent options can have a dramatic effect on their own, they aren't the only choices when configuring a TreeView. Table 13-5 lists some of the most useful properties of the TreeView.

Table 13-5. *Useful TreeView Properties*

Property	Description
MaxDataBindDepth	Determines how many levels the TreeView will show. By default, MaxDataBindDepth is –1, and you'll see the entire tree. However, if you use a value such as 2, you'll see only two levels under the starting node. This can help you pare down the display of long, multileveled site maps.
ExpandDepth	Lets you specify how many levels of nodes will be visible at first. If you use 0, the TreeView begins completely closed. If you use 1, only the first level is expanded, and so on. By default, ExpandDepth is set to the constant FullyExpand (-1), which means the tree is fully expanded and all the nodes are visible on the page.
NodeIndent	Sets the number of pixels between each level of nodes in the TreeView. Set this to 0 to create a nonindented TreeView, which saves space. A nonindented TreeView allows you to emulate an in-place menu (see, for example, Figure 13-12).

(*continued*)

Table 13-5. (*continued*)

Property	Description
ImageSet	Lets you use a predefined collection of node images for collapsed, expanded, and nonexpandable nodes. You specify one of the values in the TreeViewImageSet enumeration. You can override any node images you want to change by setting the CollapseImageUrl, ExpandImageUrl, and NoExpandImageUrl properties.
CollapseImageUrl, ExpandImageUrl, and NoExpandImageUrl	Sets the pictures that are shown next to nodes for collapsed nodes (CollapseImageUrl) and expanded nodes (ExpandImageUrl). The NoExpandImageUrl is used if the node doesn't have any children. If you don't want to create your own custom node images, you can use the ImageSet property instead to use one of several built-in image collections.
NodeWrap	Lets a node text-wrap over more than one line when set to true.
ShowExpandCollapse	Hides the expand/collapse boxes when set to false. This isn't recommended, because the user won't have a way to expand or collapse a level without clicking it (which causes the browser to navigate to the page).
ShowLines	Adds lines that connect every node when set to true.
ShowCheckBoxes	Shows a check box next to every node when set to true. This isn't terribly useful for site maps, but it is useful with other types of trees.

Properties give you a fair bit of customizing power, but one of the most interesting formatting features comes from TreeView styles, which are described in the next section.

TreeView Styles

Styles are represented by the TreeNodeStyle class, which derives from the more conventional Style class. As with other rich controls, the styles give you options to set background and foreground colors, fonts, and borders. Additionally, the TreeNodeStyle class adds the node-specific style properties shown in Table 13-6. These properties deal with the node image and the spacing around a node.

Table 13-6. *TreeNodeStyle-Added Properties*

Property	Description
ImageUrl	The URL for the image shown next to the node.
NodeSpacing	The space (in pixels) between the current node and the node above and below.
VerticalPadding	The space (in pixels) between the top and bottom of the node text and border around the text.
HorizontalPadding	The space (in pixels) between the left and right of the node text and border around the text.
ChildNodesPadding	The space (in pixels) between the last child node of an expanded parent node and the following node (for example, between the Investing and Products nodes in Figure 13-10).

Because a TreeView is rendered using an HTML table, you can set the padding of various elements to control the spacing around text, between nodes, and so on. One other property that comes into play is TreeView.NodeIndent, which sets the number of pixels of indentation (from the left) in each subsequent level of the tree hierarchy. Figure 13-12 shows how these settings apply to a single node.

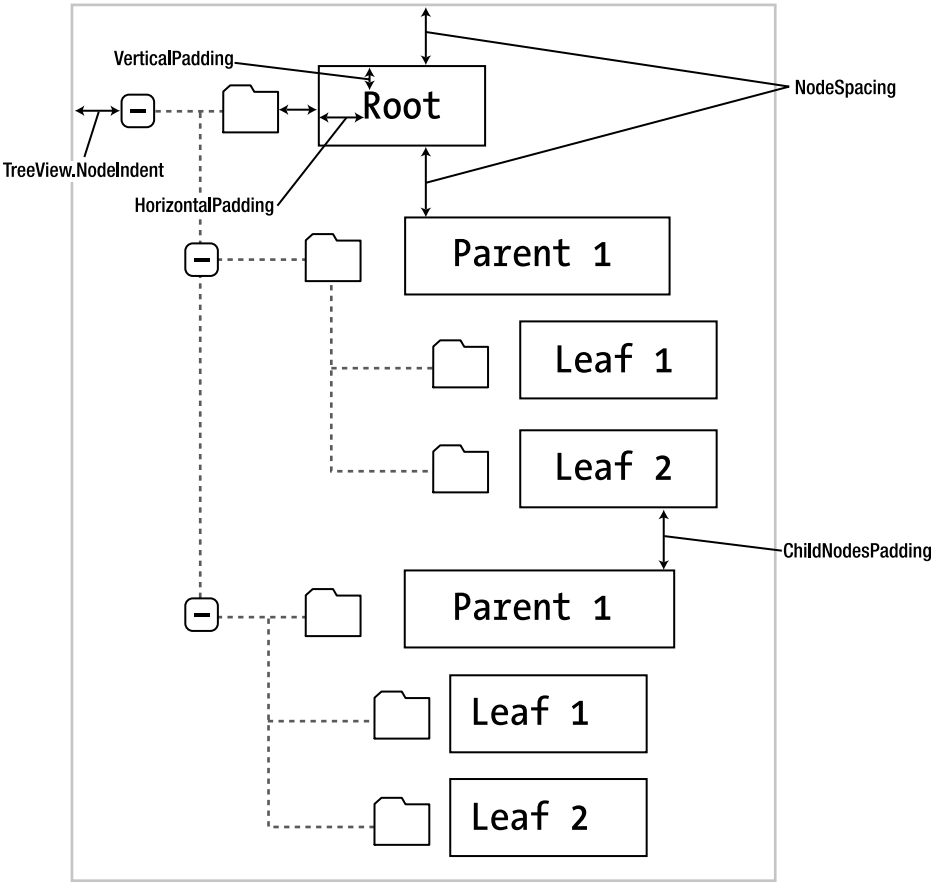

Figure 13-12. *Node spacing*

Clearly, styles give you a lot of control over how different nodes are displayed. To apply a simple TreeView makeover, and to use the same style settings for each node in the TreeView, you apply style settings through the TreeView.NodeStyle property. You can do this directly in the control tag or by using the Properties window.

For example, here's a TreeView that applies a custom font, font size, text color, padding, and spacing:

```
<asp:TreeView ID="TreeView1" runat="server" DataSourceID="SiteMapDataSource1">
    <NodeStyle Font-Names="Tahoma" Font-Size="10pt" ForeColor="Blue"
     HorizontalPadding="5px" NodeSpacing="0px" VerticalPadding="0px" />
</asp:TreeView>
```

Usually, this approach doesn't provide enough fine-tuning. Instead, you'll want to tweak a specific part of the tree. In this case, you need to find the style object that applies to the appropriate part of the tree, as explained in the following two sections.

Applying Styles to Node Types

The TreeView allows you to individually control the styles for types of nodes—for example, root nodes, nodes that contain other nodes, selected nodes, and so on. Table 13-7 lists different TreeView styles and explains what nodes they affect.

Table 13-7. *TreeView Style Properties*

Property	Description
NodeStyle	Applies to all nodes. The other styles may override some or all of the details that are specified in the NodeStyle property.
RootNodeStyle	Applies only to the first-level (root) node.
ParentNodeStyle	Applies to any node that contains other nodes, except root nodes.
LeafNodeStyle	Applies to any node that doesn't contain child nodes and isn't a root node.
SelectedNodeStyle	Applies to the currently selected node.
HoverNodeStyle	Applies to the node that the user is hovering over with the mouse. These settings are applied only in up-level clients that support the necessary dynamic script.

Here's a sample TreeView that first defines a few standard style characteristics by using the NodeStyle property, and then fine-tunes different sections of the tree by using the properties from Table 13-7:

```
<asp:TreeView ID="TreeView1" runat="server" DataSourceID="SiteMapDataSource1">
  <NodeStyle Font-Names="Tahoma" Font-Size="10pt" ForeColor="Blue"
   HorizontalPadding="5px" NodeSpacing="0px" VerticalPadding="0px" />
  <ParentNodeStyle Font-Bold="False" />
  <HoverNodeStyle Font-Underline="True" ForeColor="#5555DD" />
  <SelectedNodeStyle Font-Underline="True" ForeColor="#5555DD" />
</asp:TreeView>
```

Styles are listed in Table 13-7 in order of most general to most specific. This means the SelectedNodeStyle settings override any conflicting settings in a RootNodeStyle, for example. (If you don't want a node to be selectable, set the TreeNode.SelectAction to None.) However, the RootNodeStyle, ParentNodeStyle, and LeafNodeStyle settings never conflict, because the definitions for root, parent, and leaf nodes are mutually exclusive. You can't have a node that is simultaneously a parent and a root node, for example—the TreeView simply designates this as a root node.

Applying Styles to Node Levels

Being able to apply styles to different types of nodes is interesting, but often a more useful feature is being able to apply styles based on the node *level*. That's because many trees use a rigid hierarchy. (For example, the first level of nodes represents categories, the second level represents products, the third represents orders, and so on.) In this case, it's not so important to determine whether a node has children. Instead, it's important to determine the node's depth.

The only problem is that a TreeView can have a theoretically unlimited number of node levels. Thus, it doesn't make sense to expose properties such as FirstLevelStyle, SecondLevelStyle, and so on. Instead, the TreeView has a LevelStyles collection that can have as many entries as you want. The level is inferred from the position of the style in the collection, so the first entry is considered the root level, the second entry is the second node level, and so on. For this system to work, you must follow the same order, and you must include an empty style placeholder if you want to skip a level without changing the formatting.

For example, here's a TreeView that differentiates levels by applying different amounts of spacing and different fonts:

```
<asp:TreeView runat="server" HoverNodeStyle-Font-Underline="True"
 ShowExpandCollapse="False" NodeIndent="3" DataSourceID="SiteMapDataSource1">
  <LevelStyles>
    <asp:TreeNodeStyle ChildNodesPadding="10" Font-Bold="True" Font-Size="12pt"
     ForeColor="DarkGreen"/>
    <asp:TreeNodeStyle ChildNodesPadding="5" Font-Bold="True" Font-Size="10pt" />
    <asp:TreeNodeStyle ChildNodesPadding="5" Font-UnderLine="True"
     Font-Size="10pt" />
  </LevelStyles>
</asp:TreeView>
```

If you apply this to the category and product list shown in earlier examples, you'll see a page like the one shown in Figure 13-13.

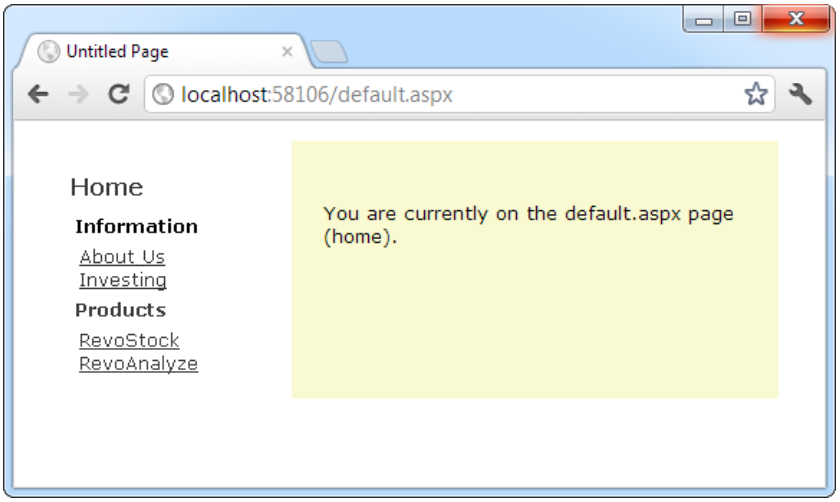

Figure 13-13. *A TreeView with styles*

TREEVIEW AUTO FORMAT

Using the right combination of styles and images can dramatically transform your TreeView. However, for those less artistically inclined, it's comforting to know that Microsoft has made many classic designs available through the TreeView's Auto Format feature.

To use it, start by selecting the TreeView on the design surface. Then click the arrow icon that appears next to the top-right corner of the TreeView to show its smart tag. In the smart tag, click the Auto Format link to show the Auto Format dialog box. In the Auto Format dialog box, you can pick from a variety of preset formats, each with a small preview. Click Apply to try the format out on your TreeView, Cancel to back out, and OK to make it official and return to Visual Studio.

The different formats correspond loosely to the different TreeViewImageSet values. However, the reality is not quite that simple. When you pick a TreeView format, Visual Studio sets the ImageSet property and applies a few matching style settings, to help you get that perfect final look.

The Menu Control

The Menu control is another rich control that supports hierarchical data. Like the TreeView, you can bind the Menu control to a data source, or you can use MenuItem objects to fill it by hand.

To try the Menu control, remove the TreeView from your master page, and add the following Menu control tag:

```
<asp:Menu ID="Menu1" runat="server" DataSourceID="SiteMapDataSource1" />
```

Notice that this doesn't configure any properties—it uses the default appearance. The only step you need to perform is setting the DataSourceID property to link the menu to the site map information.

When the Menu first appears, you'll see only the starting node, with an arrow next to it. When you move your mouse over the starting node, the next level of nodes will pop into display. You can continue this process to drill down as many levels as you want, until you find the page you want to click (see Figure 13-14). If you click a menu item, you'll be transported to the corresponding page, just as you are when you click a node in the TreeView. But unlike the TreeView, each time you click your way to a new page, the menu collapses itself back to its original appearance. It doesn't expand to show the current page.

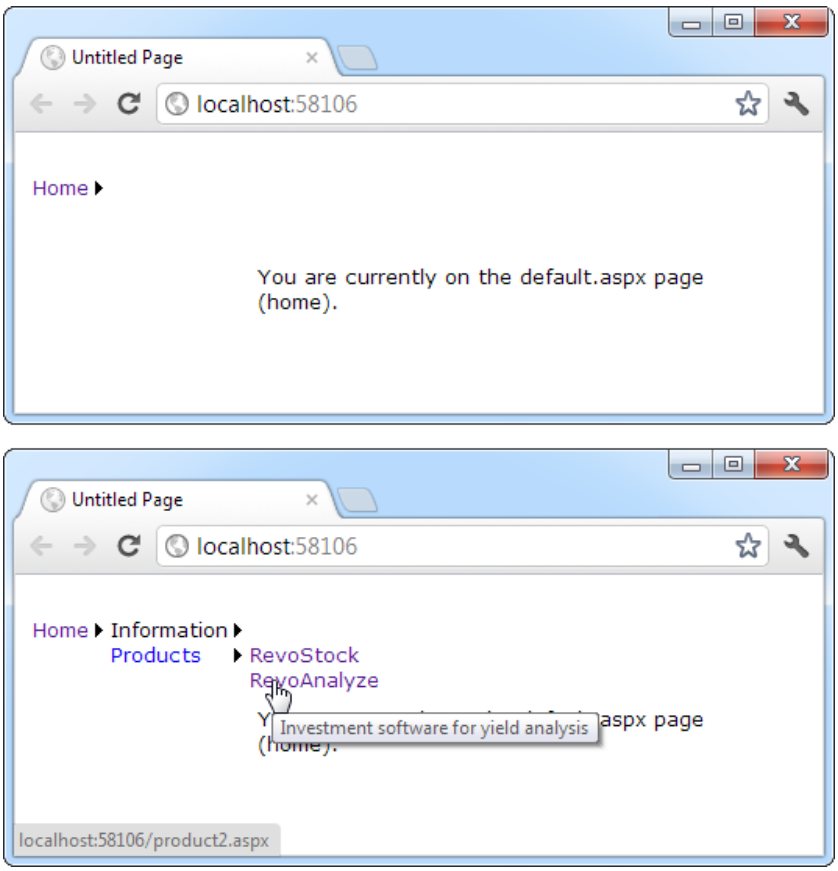

Figure 13-14. *Navigating through the menu*

Overall, the Menu and TreeView controls expose strikingly similar programming models, even though they render themselves quite differently. They also have a similar style-based formatting model. But a few noteworthy differences exist:

- The Menu displays a single submenu. The TreeView can expand an arbitrary number of node branches at a time.
- The Menu displays a root level of links in the page. All other items are displayed using fly-out menus that appear over any other content on the page. The TreeView shows all its items inline in the page.
- The Menu supports templates. The TreeView does not. (Menu templates are discussed later in this section.)
- The TreeView supports check boxes for any node. The Menu does not.
- The Menu supports horizontal and vertical layouts, depending on the Orientation property. The TreeView supports only vertical layout.

Menu Styles

The Menu control provides an overwhelming number of styles. Like the TreeView, the Menu adds a custom style class, which is named MenuItemStyle. This style adds spacing properties such as ItemSpacing, HorizontalPadding, and VerticalPadding. However, you can't set menu item images through the style, because it doesn't have an ImageUrl property.

Much like the TreeView, the Menu supports defining different menu styles for different menu levels. However, the key distinction that the Menu control encourages you to adopt is between *static* items (the root-level items that are displayed in the page when it's first generated) and *dynamic* items (the items in fly-out menus that are added when the user moves the mouse over a portion of the menu). Most websites have a definite difference in the styling of these two elements. To support this, the Menu class defines two parallel sets of styles, one that applies to static items and one that applies to dynamic items, as shown in Table 13-8.

Table 13-8. *Menu Styles*

Static Style	Dynamic Style	Description
StaticMenuStyle	DynamicMenuStyle	Sets the appearance of the overall "box" in which all the menu items appear. In the case of StaticMenuStyle, this box appears on the page, and with DynamicMenuStyle, it appears as a pop-up.
StaticMenuItemStyle	DynamicMenuItemStyle	Sets the appearance of individual menu items.
StaticSelectedStyle	DynamicSelectedStyle	Sets the appearance of the selected item. Note that the selected item isn't the item that's currently being hovered over; it's the item that was previously clicked (and that triggered the last postback).
StaticHoverStyle	DynamicHoverStyle	Sets the appearance of the item that the user is hovering over with the mouse.

Along with these styles, you can set level-specific styles so that each level of menu and submenu is different. You do this by using the LevelMenuItemStyles collection, which works like the TreeView.LevelStyles collection discussed earlier. The position of your style in the collection determines whether the menu uses it for the first level of items, the second level, the third, and so on. You can also use the LevelSelectedStyles collection to set level-specific formatting for selected items.

It might seem as if you have to do a fair bit of unnecessary work when separating dynamic and static styles. The reason for this model becomes obvious when you consider another remarkable feature of the Menu control—it allows you to choose the number of static levels. By default, only one static level exists, and everything else is displayed as a fly-out menu when the user hovers over the corresponding parent. But you can set the Menu.StaticDisplayLevels property to change all that. If you set it to 2, for example, the first two levels of the menu will be rendered in the page by using the static styles. (You can control the indentation of each level by using the StaticSubMenuIndent property.)

Figure 13-15 shows the menu with StaticDisplayLevels set to 2 (and some styles applied through the Auto Format link). Each menu item will still be highlighted when you hover over it, as in a nonstatic menu, and selection will also work the same way as it does in the nonstatic menu.

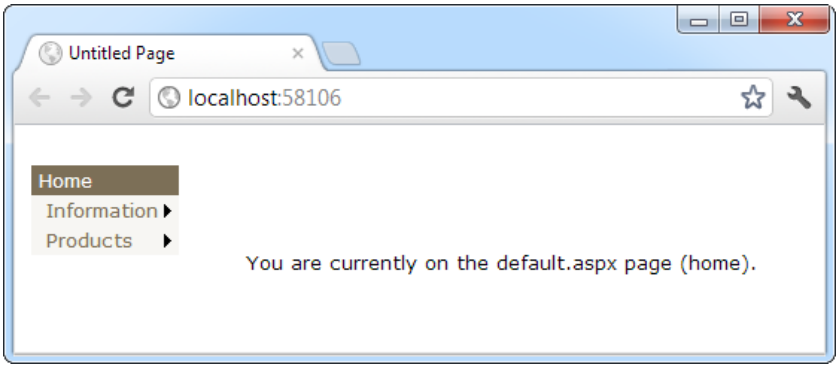

Figure 13-15. *A menu with two static levels*

If you use a menu for navigation, every time you click your way to a new page, the menu returns to its original appearance, showing all static levels but hiding all dynamic levels.

■ **Tip** The Menu control exposes many more top-level properties for tweaking specific rendering aspects. For example, you can set the delay before a pop-up menu disappears (DisappearAfter), the default images used for expansion icons and separators, the scrolling behavior (which kicks into gear when the browser window is too small to fit a pop-up menu), and much more. For more information, you can read the Menu control reference online at http://msdn.microsoft.com/library/system.web.ui.webcontrols.menu.aspx.

Menu Templates

The Menu control also supports templates through the StaticItemTemplate and DynamicItemTemplate properties. These templates determine the HTML that's rendered for each menu item, giving you complete control.

You've already seen how to create templates for the SiteMapPath, but the process of creating templates for the Menu is a bit different. Whereas each node in the SiteMapPath is bound directly to a SiteMapNode object, the Menu is bound to something else: a dedicated MenuItem object.

This subtle quirk can complicate life. For one thing, you can't rely on properties such as Title, Description, and Url, which are provided by the SiteMapNode object. Instead, you need to use the MenuItem.Text property to get the information you need to display, as shown here:

```
<asp:Menu ID="Menu1" runat="server">
  <StaticItemTemplate>
    <%# Eval("Text") %>
  </StaticItemTemplate>
</asp:Menu>
```

One reason you might want to use the template features of the Menu is to show multiple pieces of information in a menu item. For example, you might want to show both the title *and* the description from the SiteMapNode for this item (rather than just the title). Unfortunately, that's not as easy as it is with the SiteMapPath. Once again, the problem is that the Menu binds directly to MenuItem objects, not the SiteMapNode objects, and MenuItem objects just don't provide the information you need.

If you're really desperate, there is a workaround using an advanced data-binding technique. Rather than binding to a property of the MenuItem object, you can bind to a custom method that you create in your page class. This custom method can then include the code that's needed to get the correct SiteMapNode object (based on the current URL) and provide the extra information you need. In a perfect world, this extra work wouldn't be necessary, but unfortunately, it's the simplest workaround in this situation.

For example, consider the more descriptive menu items that are shown in Figure 13-16.

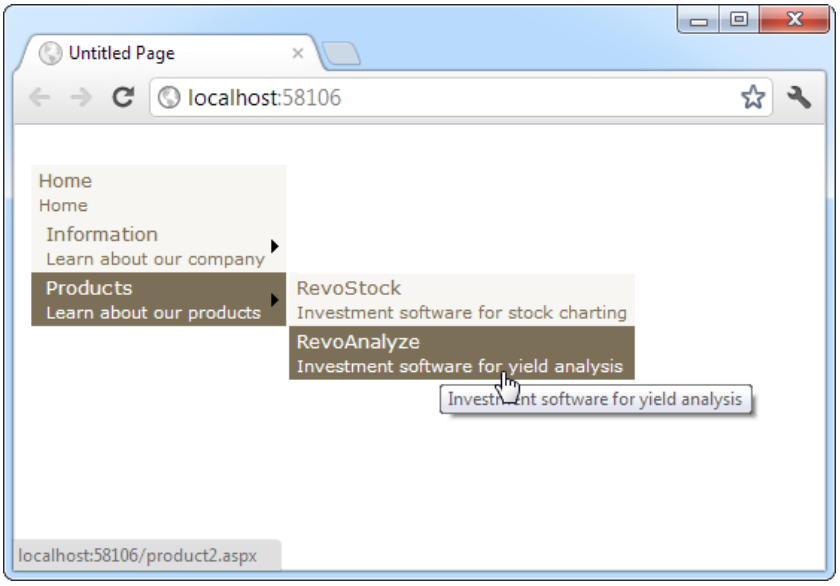

Figure 13-16. *Showing node descriptions in a menu*

To create this example, you need to build a template that uses two types of data-binding expressions. The first type simply gets the MenuItem text (which is the page title). You already know how to write this sort of data-binding expression:

```
<%# Eval("Text") %>
```

The second type of data-binding expression is more sophisticated. It uses a custom method named GetDescriptionFromTitle(), which you need to create. This method takes the page title information and returns something more interesting—in this case, the full description for that item:

```
<%# GetDescriptionFromTitle(((MenuItem)Container.DataItem).Text) %>
```

Sadly, the Eval() method can't help you out with this sort of data-binding expression. Instead, you need to explicitly grab the data object (using Container.DataItem), cast it to the appropriate type (MenuItem), and then retrieve the right property (Text). This gets the same page title as in the previous data-binding expression, but it allows you to pass it to the GetDescriptionFromTitle() method.

Here's the full template that uses both types of data-binding expressions to show the top level of static menu items and the second level of pop-up (dynamic) items:

```
<asp:Menu ID="Menu1" runat="server" DataSourceID="SiteMapDataSource1">
  <StaticItemTemplate>
    <%# Eval("Text") %><br />
```

```
    <small>
    <%# GetDescriptionFromTitle(((MenuItem)Container.DataItem).Text) %>
    </small>
  </StaticItemTemplate>
  <DynamicItemTemplate>
    <%# Eval("Text") %><br />
    <small>
    <%# GetDescriptionFromTitle(((MenuItem)Container.DataItem).Text) %>
    </small>
  </DynamicItemTemplate>
</asp:Menu>
```

The next step is to create the GetDescriptionFromTitle() method in the code for your page class. This method belongs in the page that has the Menu control, which, in this example, is the master page. The GetDescriptionFromTitle() method must also have protected (or public) accessibility, so that ASP.NET can call it during the data-binding process:

```
protected string GetDescriptionFromTitle(string title)
{... }
```

The tricky part is filling in the code you need. In this example, two custom methods are involved. In order to find the node it needs, GetDescriptionFromTitle() calls another method, named SearchNodes(). The SearchNodes() method calls itself several times to perform a recursive search through the whole hierarchy of nodes. It ends its search only when it finds a matching node, which it returns to GetDescriptionFromTitle(). Finally, GetDescriptionFromTitle() extracts the description information (and anything else you're interested in).

Here's the complete code that makes this example work:

```
protected string GetDescriptionFromTitle(string title)
{
    // This assumes there's only one node with this title.
    SiteMapNode startingNode = SiteMap.RootNode;
    SiteMapNode matchNode = SearchNodes(startingNode, title);
    if (matchNode == null)
    {
        return null;
    }
    else
    {
        return matchNode.Description;
    }
}

private SiteMapNode SearchNodes(SiteMapNode node, string title)
{
    if (node.Title == title)
    {
        return node;
    }
    else
    {
        // Perform recursive search.
        foreach (SiteMapNode child in node.ChildNodes)
```

```
        {
            SiteMapNode matchNode = SearchNodes(child, title);
            // Was a match found?
            // If so, return it.
            if (matchNode != null) return matchNode;
        }
        // All the nodes were examined, but no match was found.
        return null;
    }
}
```

Once you've finished this heavy lifting, you can use the GetDescriptionFromTitle() method in a template to get the additional information you need.

The Last Word

In this chapter, you explored the new navigation model and learned how to define site maps and bind the navigation data. You then considered three controls that are specifically designed for navigation data: the SiteMapPath, TreeView, and Menu. Using these controls, you can add remarkably rich site maps to your websites with very little coding. But before you begin, make sure you've finalized the structure of your website. Only then will you be able to create the perfect site map and choose the best ways to present the site map information in the navigation controls.