CHAPTER 5

Working with Razor

A *view engine* procezses ASP.NET content and looks for instructions, typically to insert dynamic content into the output sent to a browser and *Razor* is the name of the MVC Framework view engine. There are no changes to Razor in MVC 5 and if you are already familiar with the syntax from earlier versions you can skip ahead.

In this chapter, I give you a quick tour of the Razor syntax so you can recognize Razor expressions when you see them. I am not going to supply an exhaustive Razor reference in this chapter; think of this more as a crash course in the syntax. I explore Razor in depth as I continue through the book, within the context of other MVC Framework features. Table 5-1 provides the summary for this chapter.

Table 5-1. *Chapter Summary*

Problem	Solution	Listing
Define and access the model type	Use the `@model` and `@Model` expressions	1–4, 15
Reduce duplication in views	Use a layout	5–7, 10–12
Specify a default layout	Use the view start view	8, 9
Pass data values to the view from the controller	Pass a view model object or the view bag	13, 14
Generate different content based on data values	Use Razor conditional statements	16, 17
Enumerate an array or a collection	Use a `@foreach` expression	18, 19
Add a namespace to a view	Use a `@using` expression	20

Preparing the Example Project

To demonstrate Razor, I created a new Visual Studio project called Razor using the `ASP.NET MVC Web Application` template. I selected the `Empty` option and checked the box to get the initial configuration for an MVC project.

Defining the Model

I am going to start with a simple domain model called `Product`, defined in a class file called `Product.cs`, which I added to the `Models` folder. You can see the contents of the new file in Listing 5-1.

Listing 5-1. The Contents of the Product.cs File

```
namespace Razor.Models {

    public class Product {

        public int ProductID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Category { set; get; }
    }
}
```

Defining the Controller

I am going to follow the MVC Framework convention and define a controller called `Home` as the initial starting point for my project. Right-click the `Controllers` folder in the Solution Explorer, select `Add` ➤ `Controller`, select `MVC 5 Controller-Empty`, click `Add` and set the name to `HomeController`. When you click the second `Add` button, Visual Studio will create the `Controllers/HomeController.cs` file. Edit the contents to match those shown in Listing 5-2.

Listing 5-2. The Content of the HomeController.cs File

```
using System.Web.Mvc;
using Razor.Models;

namespace Razor.Controllers {

    public class HomeController : Controller {
        Product myProduct = new Product {
            ProductID = 1,
            Name = "Kayak",
            Description = "A boat for one person",
            Category = "Watersports",
            Price = 275M
        };

        public ActionResult Index() {
            return View(myProduct);
        }
    }
}
```

The controller defines an action method called `Index`, in which I create and populate the properties of a `Product` object. I pass the `Product` to the `View` method so that it is used as the model when the view is rendered. I do not specify the name of a view file when I call the `View` method, so the default view for the action method will be used.

Creating the View

Right-click on the `Index` method in the `HomeController` class and select `Add View` from the pop-up menu. Ensure that the name of the view is `Index`, change the `Template` to `Empty` and select `Product` for the model class. (If you don't see `Product` as an option for the model, compile the project and start over). Uncheck the `View Option` boxes and click `Add` to create the `Index.cshtml` file in the `Views/Home` folder. The initial contents of the new view are shown in Listing 5-3.

Listing 5-3. The Contents of the Index.cshtml File

```
@model Razor.Models.Product

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>

    </div>
</body>
</html>
```

In the sections that follow, I go through the different parts of a Razor view and demonstrate some of the different things you can do with one. When learning about Razor, it is helpful to bear in mind that views exist to express one or more parts of the model to the user—and that means generating HTML that displays data that is retrieved from one or more objects. If you remember that I am always trying to build an HTML page that can be sent to the client, then everything that Razor does begins to make sense.

■ **Note** I repeat some information in the following sections that I already touched on in Chapter 2. I want to provide you with a single place in the reference that you can turn to when you need to look up a Razor feature and I thought that a small amount of duplication made this worthwhile.

Working with the Model Object

Let us start with the first line in the view:

```
...
@model Razor.Models.Product
...
```

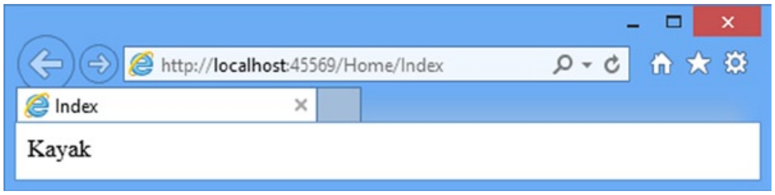
Razor statements start with the `@` character. In this case, the `@model` statement declares the type of the model object that I will pass to the view from the action method. This allows me to refer to the methods, fields, and properties of the view model object through `@Model`, as shown in Listing 5-4, which shows a simple addition to the `Index` view.

Listing 5-4. Referring to a View Model Object Property in the Index.cshtml File

```
@model Razor.Models.Product

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        @Model.Name
    </div>
</body>
</html>
```

■ **Note** Notice that I declare the view model object type using `@model` (lower case `m`) and access the `Name` property using `@Model` (upper case `M`). This is slightly confusing when you start working with Razor, but it becomes second nature pretty quickly.

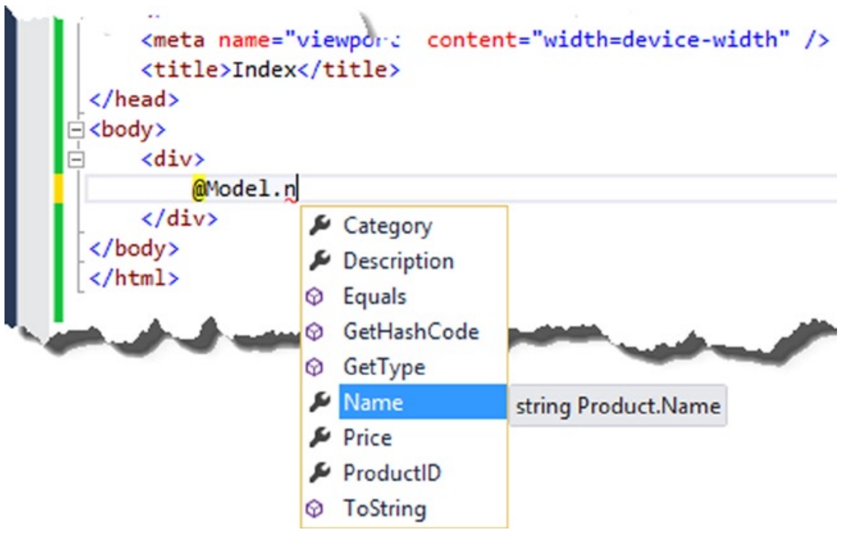
If you start the project, you'll see the output shown in Figure 5-1.



Kayak

Figure 5-1. *The Effect of Reading a Property Value in the View*

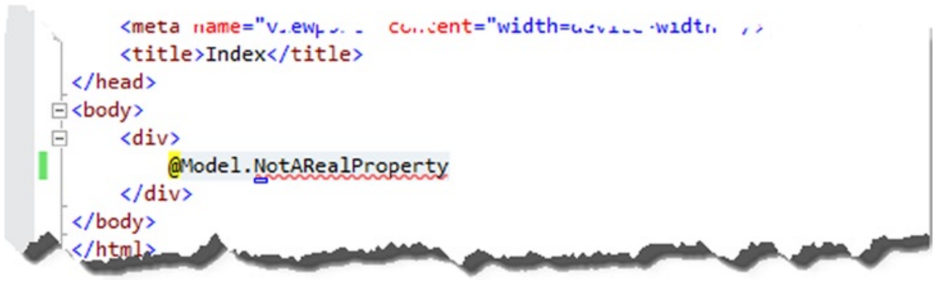
By using the `@model` expression, I tell MVC what kind of object I will be working with and Visual Studio takes advantage of this in a couple of ways. First, as you are writing your view, Visual Studio will pop up suggestions of member names when you type `@Model` followed by a period, as shown in Figure 5-2. This is similar to the way that autocomplete for lambda expressions passed to HTML helper methods works, which I described in Chapter 4.



```
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        @Model.n
    </div>
</body>
</html>
```

Figure 5-2. *Visual Studio offering suggestions for member names based on the @Model expression*

Equally useful is that Visual Studio will flag errors when there are problems with the view model object members you are referring to. You can see an example of this in Figure 5-3, where I have tried to reference `@Model.NotARealProperty`. Visual Studio has realized that the `Product` class I specified at the model type does not have such a property and has highlighted an error in the editor.



```
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        @Model.NotARealProperty
    </div>
</body>
</html>
```

Figure 5-3. *Visual Studio reporting a problem with an @Model expression*

Working with Layouts

The other Razor expression in the `Index.cshtml` view file is this one:

```
...
@{
    Layout = null;
}
...
```

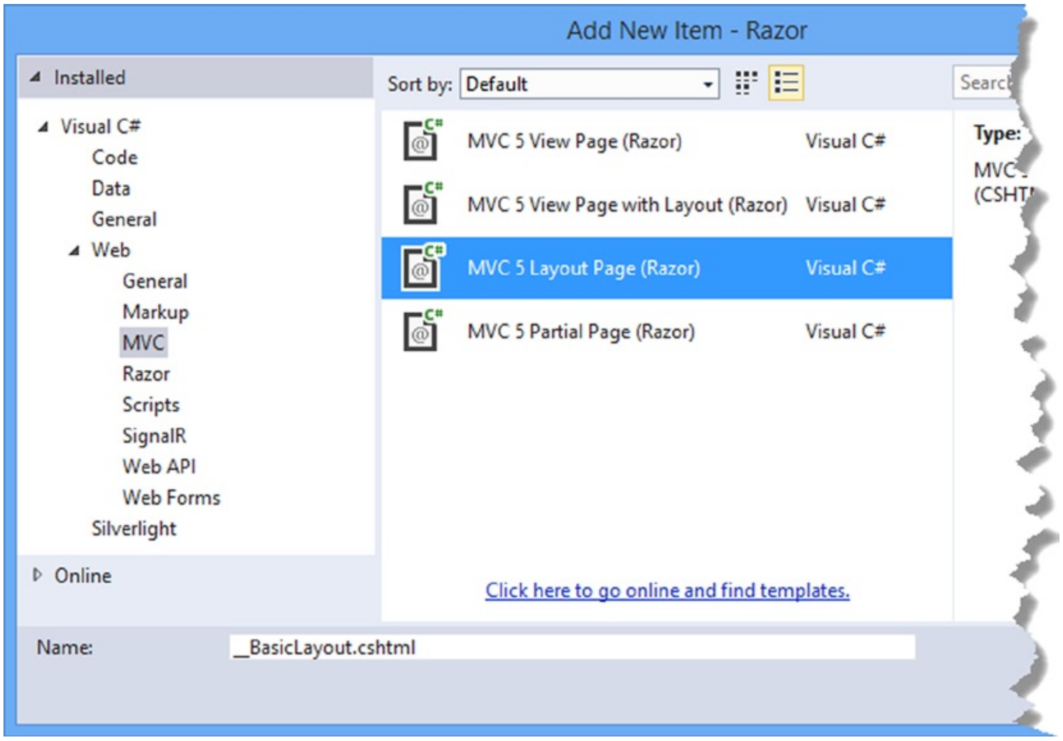
This is an example of a Razor *code block*, which allows me to include C# statements in a view. The code block is opened with `@{` and closed with `}` and the statements it contains are evaluated when the view is rendered.

This code block sets the value of the `Layout` property to `null`. As I explain in detail in Chapter 20, Razor views are compiled into C# classes in an MVC application and the base class that is used defines the `Layout` property. I'll show you how this all works in Chapter 20, but the effect of setting the `Layout` property to `null` is to tell the MVC framework that the view is self-contained and will render all of the content required for the client.

Self-contained views are fine for simple example apps, but a real project can have dozens of views. Layouts are effectively templates that contain markup that you use to create consistency across your app—this could be to ensure that the right JavaScript libraries are included in the result or that a common look and feel is used throughout.

Creating the Layout

To create a layout, right-click on the `Views` folder in the Solution Explorer, click `Add` ➤ `New Item` from the `Add` menu and select the `MVC 5 Layout Page (Razor)` template, as shown in Figure 5-4.


Add New Item - Razor
Installed
Visual C#
Code
Data
General
Web
General
Markup
MVC
Razor
Scripts
SignalR
Web API
Web Forms
Silverlight
Online
Sort by: Default
Searc
MVC 5 View Page (Razor) Visual C#
MVC 5 View Page with Layout (Razor) Visual C#
MVC 5 Layout Page (Razor) Visual C#
MVC 5 Partial Page (Razor) Visual C#
Type:
MVC
(CSHT
Click here to go online and find templates.

Name: _BasicLayout.cshtml

Figure 5-4. *Creating a new layout*

Set the name of the file to `_BasicLayout.cshtml` (notice the first character is an underscore) and click the `Add` button to create the file. Listing 5-5 shows the contents of the file as it is created by Visual Studio.

■ **Note** Files in the `Views` folder whose names begin with an underscore (_) are not returned to the user, which allows the file name to differentiate between views that you want to render and the files that support them. Layouts, which are support files, are prefixed with an underscore.

Listing 5-5. The Initial Contents of the _BasicLayout.cshtml File

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

Layouts are a specialized form of view and I have highlighted the `@` expressions in the listing. The call to the `@RenderBody` method inserts the contents of the view specified by the action method into the layout markup. The other Razor expression in the layout looks for a property called `Title` in the `ViewBag` in order to set the contents of the `title` element.

The elements in the layout will be applied to any view that uses the layout and this is why layouts are essentially templates. In Listing 5-6, I have added some simple markup to the layout to demonstrate how this works.

Listing 5-6. Adding Elements to the _BasicLayout.cshtml File

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <h1>Product Information</h1>
    <div style="padding: 20px; border: solid medium black; font-size: 20pt">
        @RenderBody()
    </div>
    <h2>Visit <a href="http://apress.com">Apress</a></h2>
</body>
</html>
```

I have added a couple of header elements and applied some CSS styles to the `div` element which contains the `@RenderBody` expression, just to make it clear what content comes from the layout and what comes from the view.

Applying a Layout

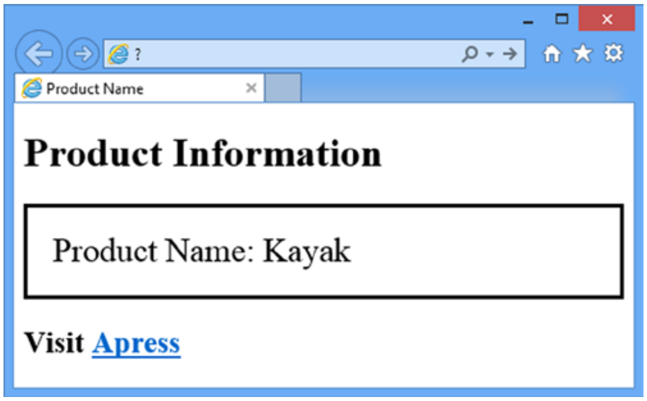
To apply the layout to the view, I just need to set the value of the Layout property. The layout contains the HTML elements that define the structure of the response to the browser, so I can also remove those elements from the view. You can see how I have applied the layout in Listing 5-7, which shows a drastically simplified `Index.cshtml` file.

■ **Tip** I also set a value for the `ViewBag.Title` property, which will be used as the contents of the title element in the HTML document sent back to the user—this is optional, but good practice. If there is no value for the property, the MVC framework will return an empty `title` element.

Listing 5-7. Using the Layout Property in the Index.cshtml File

```
@model Razor.Models.Product

@{
    ViewBag.Title = "Product Name";
    Layout = "~/Views/_BasicLayout.cshtml";
}

Product Name: @Model.Name
```

The transformation is dramatic, even for such a simple view. What I am left with is focused on presenting data from the view model object to the user, which is ideal–not only does it let me work with simpler markup, but it means that I don't have to duplicate common elements in every view that I create. To see the effect of the layout, run the example app. The results are shown in Figure 5-5.



Product Information

Product Name: Kayak

Visit Apress

Figure 5-5. *The Effect of Applying a Simple Layout to a View*

Using a View Start File

I still have a tiny wrinkle to sort out, which is that I have to specify the layout file I want in every view. That means that if I need to rename the layout file, I am going to have to find every view that refers to it and make a change, which will be an error-prone process and counter to the general theme of easy maintenance that runs through the MVC framework.

I can resolve this by using a *view start file*. When it renders a view, the MVC framework will look for a file called _ViewStart.cshtml. The contents of this file will be treated as though they were contained in the view file itself and I can use this feature to automatically set a value for the Layout property.

To create a view start file, add a new layout file to the Views folder and set the name of the file to _ViewStart.cshtml (once again, notice the leading underscore). Edit the contents of the new file so that they match those shown in Listing 5-8.

Listing 5-8. The contents of the _ViewStart.cshtml File

```
@{
    Layout = "~/Views/_BasicLayout.cshtml";
}
```

My view start file contains a value for the Layout property, which means that I can remove the corresponding expression from the Index.cshtml file, as shown in Listing 5-9.

Listing 5-9. Updating the Index.cshtml File to Reflect the Use of a View Start File

```
@model Razor.Models.Product

@{
    ViewBag.Title = "Product Name";
}

Product Name: @Model.Name
```

I do not have to specify that I want to use the view start file. The MVC framework will locate the file and use its contents automatically. The values defined in the view file take precedence, which makes it easy to override the view start file.

■ **Caution** It is important to understand the difference between omitting the Layout property from the view file and setting it to null. If your view is self-contained and you do not want to use a layout, then set the Layout property to null. If you omit the Layout property, then the MVC framework will assume that you *do* want a layout and that it should use the value it finds in the view start file.

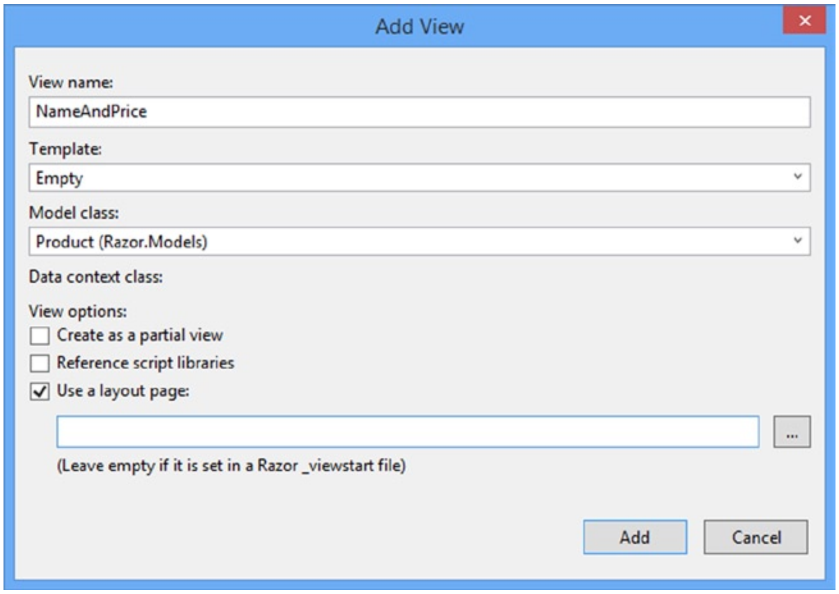
Demonstrating Shared Layouts

As a quick and simple demonstration of how layouts are shared, I have added a new action method to the Home controller called NameAndPrice. You can see the definition of this method in Listing 5-10, which shows the changes I made to the HomeController.cs file.

Listing 5-10. Adding a New Action Method to the HomeController.cs File

```
using System.Web.Mvc;
using Razor.Models;

namespace Razor.Controllers {

    public class HomeController : Controller {
        Product myProduct = new Product {
            ProductID = 1,
            Name = "Kayak",
            Description = "A boat for one person",
            Category = "Watersports",
            Price = 275M
        };

        public ActionResult Index() {
            return View(myProduct);
        }

        public ActionResult NameAndPrice() {
            return View(myProduct);
        }
    }
}
```

The action method passes the `myProduct` object to the view method, just like the `Index` action method does—this is not something that you would do in a real project, but I am demonstrating Razor functionality and so a simple example suits my needs. Right-click on the `NameAndPrice` action method and select `Add View` from the pop-up menu. Fill in the `Add View` dialog to match Figure 5-6: set `View Name` to `NameAndPrice`; set `Template` to `Empty` and set the `Model Class` to `Product`.



View name: NameAndPrice
Template: Empty
Model class: Product (Razor.Models)
Data context class:
View options:
[ ] Create as a partial view
[ ] Reference script libraries
[✓] Use a layout page:
(Leave empty if it is set in a Razor _viewstart file)

Figure 5-6. *Adding a view that relies on a layout*

Ensure that the `Use a Layout Page` box is checked–and notice the text beneath the text field. It says that you should leave the textbox empty if you have specified the view you want to use in a view start file. If you were to click the `Add` button at this point, the view would be created without a value for the `Layout` property.

I want to explicitly specify the view, so click on the button with an ellipsis label (`...`) that is to the right of the text field. Visual Studio will present you with a dialog that allows you to select a layout file, as shown in Figure 5-7.

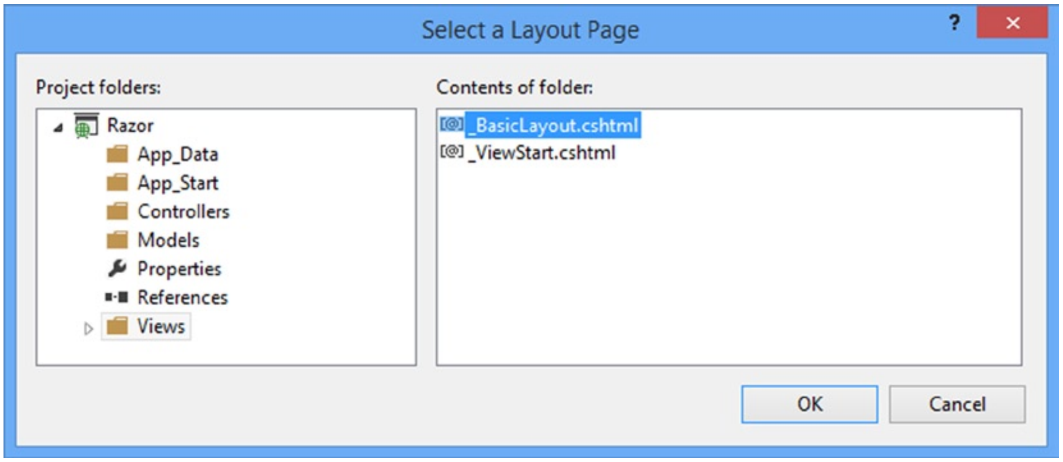
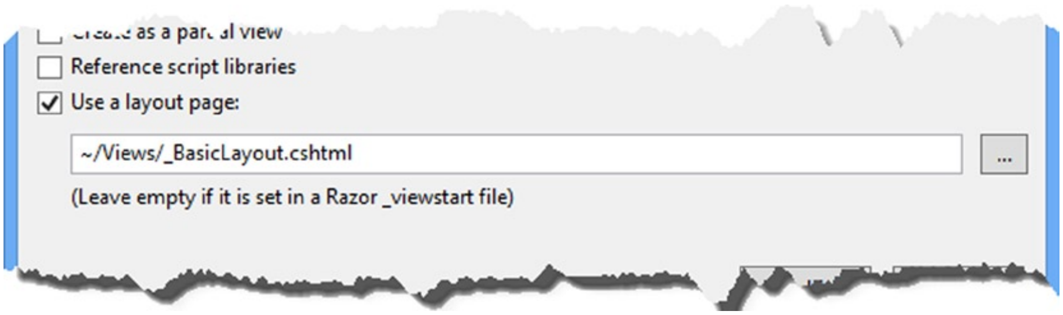


Figure 5-7. *Selecting the layout file*

The convention for an MVC project is to place layout files in the `Views` folder, but is only a convention, which is why the left-hand panel of the dialog lets you navigate around the project, just in case you have decided to put them somewhere else.

I have only defined one layout file, so select `_BasicLayout.cshtml` and click the `OK` button to return to the `Add View` dialog. You will see that the name of the layout file has been placed in the textbox, as shown in Figure 5-8.


Create as a partial view
Reference script libraries
Use a layout page:
...
(Leave empty if it is set in a Razor _viewstart file)

~/Views/_BasicLayout.cshtml

Figure 5-8. *Specifying a layout file when creating a view*

Click the `Add` button to create the `/Views/Home/NameAndPrice.cshtml` file. You can see the contents of this file in Listing 5-11.

Listing 5-11. The Contents of the NameAndPrice.cshtml File

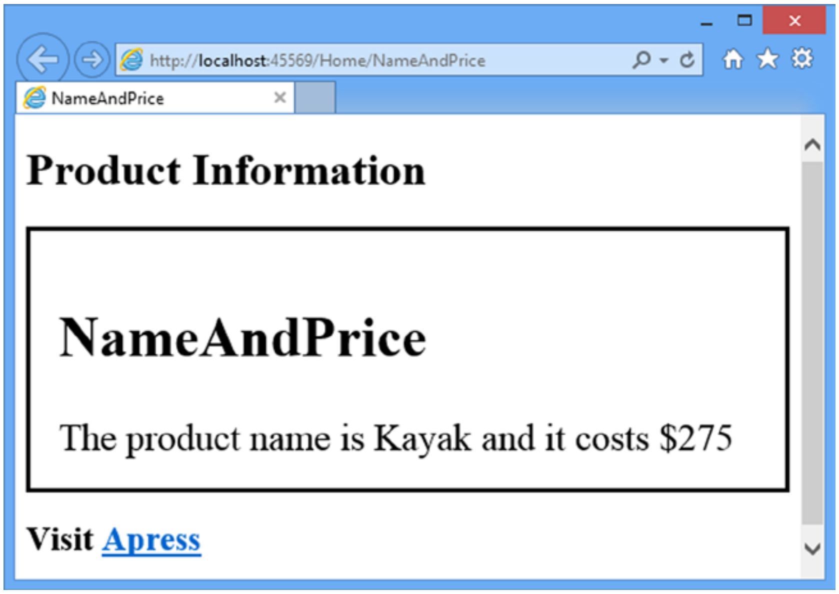
```
@model Razor.Models.Product

@{
    ViewBag.Title = "NameAndPrice";
    Layout = "~/Views/_BasicLayout.cshtml";
}

<h2>NameAndPrice</h2>
```

Visual Studio uses slightly different default content for view files when you specify a layout, but you can see that the result contains the same Razor expressions I used when I applied the layout to a view directly. To complete this example, Listing 5-12 shows a simple addition to the `NameAndPrice.cshtml` file that displays data values from the view model object.

Listing 5-12. Adding to the NameAndPrice.cshtml File

```
@model Razor.Models.Product

@{
    ViewBag.Title = "NameAndPrice";
    Layout = "~/Views/_BasicLayout.cshtml";
}

<h2>NameAndPrice</h2>
The product name is @Model.Name and it costs $@Model.Price
```

If you start the app and navigate to `/Home/NameAndPrice`, you will see the results illustrated by Figure 5-9. As you might have expected, the common elements and styles defined in the layout have been applied to the view, demonstrating how a layout can be used as a template to create a common look and feel (albeit a simple and unattractive one in this example).



Product Information

NameAndPrice

The product name is Kayak and it costs $275

Visit Apress

Figure 5-9. *The content in the layout file applied to the NameAndPrice view*

■ **Note** I would have gotten the same result if I had left the text field in the `Add View` dialog empty and relied on the view start file. I specified the file explicitly only because I wanted to show you the Visual Studio feature which helps you make a selection.

Using Razor Expressions

Now that I have shown you the basics of views and layouts, I am going to turn to the different kinds of expressions that Razor supports and how you can use them to create view content. In a good MVC Framework application, there is a clear separation between the roles that the action method and view perform. The rules are simple and I have summarized them in Table 5-2.

Table 5-2. *The Roles Played by the Action Method and the View*

Component	Does Do	Doesn't Do
Action Method	Passa view model object to the view	Pass formatted data to the view
View	Use the view model object to present content to the user	Change any aspect of the view model object

I am going to come back to this theme throughout this book. To get the best from the MVC Framework, you need to respect and enforce the separation between the different parts of the app. As you will see, you can do quite a lot with Razor, including using C# statements—but you should not use Razor to perform business logic or manipulate your domain model objects in any way.

Equally, you should not format the data that your action method passed to the view. Instead, let the view figure out data it needs to display. You can see a simple example of this in the previous section of this chapter. I defined an action method called `NameAndPrice`, which displays the value of the `Name` and `Price` properties of a `Product` object. Even though I knew which properties I needed to display, I passed the complete `Product` object to the view model, like this:

```
...
public ActionResult NameAndPrice() {
    return View(myProduct);
}
...
```

I then used the Razor `@Model` expression in the view to get the value of the properties I was interested in, like this:

```
...
The product name is @Model.Name and it costs $@Model.Price
...
```

I could have created the string I wanted to display in the action method and passed it as the view model object to the view. It would have worked, but taking this approach undermines the benefit of the MVC pattern and reduces my ability to respond to changes in the future. As I said, I will return to this theme again, but you should remember that the MVC Framework does not enforce proper use of the MVC pattern and that you must remain aware of the effect of the design and coding decisions you make.

PROCESSING VERSUS FORMATTING DATA

It is important to differentiate between *processing* data and *formatting* it. Views *format* data, which is why I passed the `Product` object in the previous section to the view, rather than formatting the object's properties into a display string. Processing data–including selecting the data objects to display–is the responsibility of the controller, which will call on the model to get and modify the data it requires. It can sometimes be hard to figure out where the line between processing and formatting is, but as a rule of thumb, I recommend erring on the side of caution and pushing anything but the most simple of expressions out of the view and into the controller.

Inserting Data Values

The simplest thing you can do with a Razor expression is to insert a data value into the markup. You can do this using the `@Model` expression to refer to properties and methods defined by the view model object or use the `@ViewBag` expression to refer to properties you have defined dynamically using the view bag feature (which I introduced in Chapter 2).

You have already seen examples of both these expressions, but for completeness, I have added a new action method to the `Home` controller called `DemoExpressions` that passes data to the view using a model object and the view bag. You can see the definition of the new action method in Listing 5-13.

Listing 5-13. The DemoExpression Action Method in the HomeController.cs File

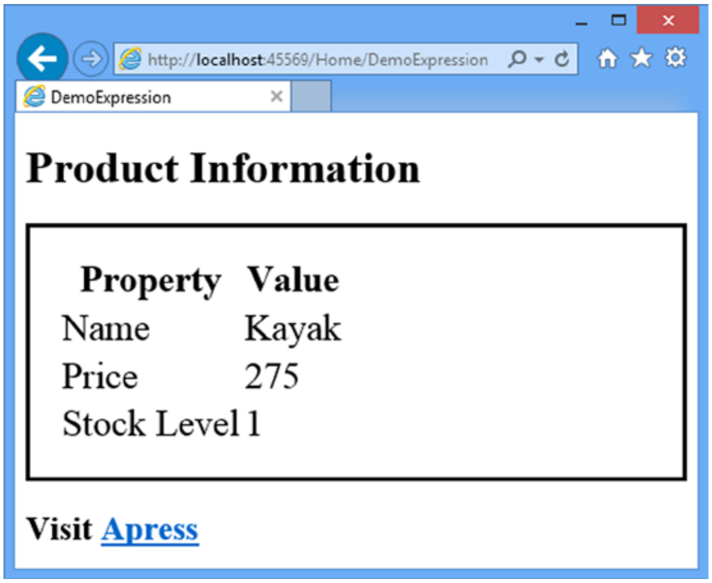
```
using System.Web.Mvc;
using Razor.Models;

namespace Razor.Controllers {
    public class HomeController : Controller {
        Product myProduct = new Product {
            ProductID = 1,
            Name = "Kayak",
            Description = "A boat for one person",
            Category = "Watersports",
            Price = 275M
        };

        public ActionResult Index() {
            return View(myProduct);
        }

        public ActionResult NameAndPrice() {
            return View(myProduct);
        }

        public ActionResult DemoExpression() {

            ViewBag.ProductCount = 1;
            ViewBag.ExpressShip = true;
            ViewBag.ApplyDiscount = false;
            ViewBag.Supplier = null;

            return View(myProduct);
        }
    }
}
```

I have created a strongly typed view called `DemoExpression.cshtml` in the `Views/Home` folder to show these basic expressions and you can see the contents of the view file in Listing 5-14.

Listing 5-14. The Contents of the DemoExpression.cshtml File

```
@model Razor.Models.Product

@{
    ViewBag.Title = "DemoExpression";
}

<table>
    <thead>
        <tr><th>Property</th><th>Value</th></tr>
    </thead>
    <tbody>
        <tr><td>Name</td><td>@Model.Name</td></tr>
        <tr><td>Price</td><td>@Model.Price</td></tr>
        <tr><td>Stock Level</td><td>@ViewBag.ProductCount</td></tr>
    </tbody>
</table>
```

I created a simple HTML table and used the properties from the model object and the view bag to populate some of the cell values. You can see the result of starting the app and navigating to the `/Home/DemoExpression` URL, as shown in Figure 5-10. This is just a reconfirmation of the basic Razor expressions that I have been using in the examples so far.



Product Information

Property	Value
Name	Kayak
Price	275
Stock Level	1

Visit Apress

Figure 5-10. *Using basic Razor expressions to insert data values into the HTML markup*

The result is not pretty because I have not applied any CSS styles to the HTML elements that the view and the layout generate, but the example serves to reinforce the way in which the basic Razor expressions can be used to display the data passed from the action method to the view.

Setting Attribute Values

All of my examples so far have set the content of elements, but you can also use Razor expressions to set the value of element *attributes*. Listing 5-15 shows how I have changed the `DemoExpression.cshtml` view to use view bag properties to set attribute values.

Listing 5-15. Using a Razor Expression to Set an Attribute Value in the DemoExpression.cshtml File

```
@model Razor.Models.Product

@{
    ViewBag.Title = "DemoExpression";
    Layout = "~/Views/_BasicLayout.cshtml";
}

<table>
    <thead>
        <tr><th>Property</th><th>Value</th></tr>
    </thead>
    <tbody>
        <tr><td>Name</td><td>@Model.Name</td></tr>
        <tr><td>Price</td><td>@Model.Price</td></tr>
        <tr><td>Stock Level</td><td>@ViewBag.ProductCount</td></tr>
    </tbody>
</table>
```

```
<div data-discount="@ViewBag.ApplyDiscount" data-express="@ViewBag.ExpressShip"
    data-supplier="@ViewBag.Supplier">
    The containing element has data attributes
</div>

Discount:<input type="checkbox" checked="@ViewBag.ApplyDiscount" />
Express:<input type="checkbox" checked="@ViewBag.ExpressShip" />
Supplier:<input type="checkbox" checked="@ViewBag.Supplier" />
```

I used basic Razor expressions to set the value for some `data` attributes on a `div` element.

■ **Tip** Data attributes, which are attributes whose names are prefixed by `data-`, have been an informal way of creating custom attributes for many years and have been made part of the formal standard as part of HTML5. I have used the values of the `ApplyDiscount`, `ExpressShip` and `Supplier` view bag properties to set the value of these attributes.

Start the example app, target the action method, and look at the HTML source that has been used to render the page. You will see that Razor has set the value of the attributes like this:

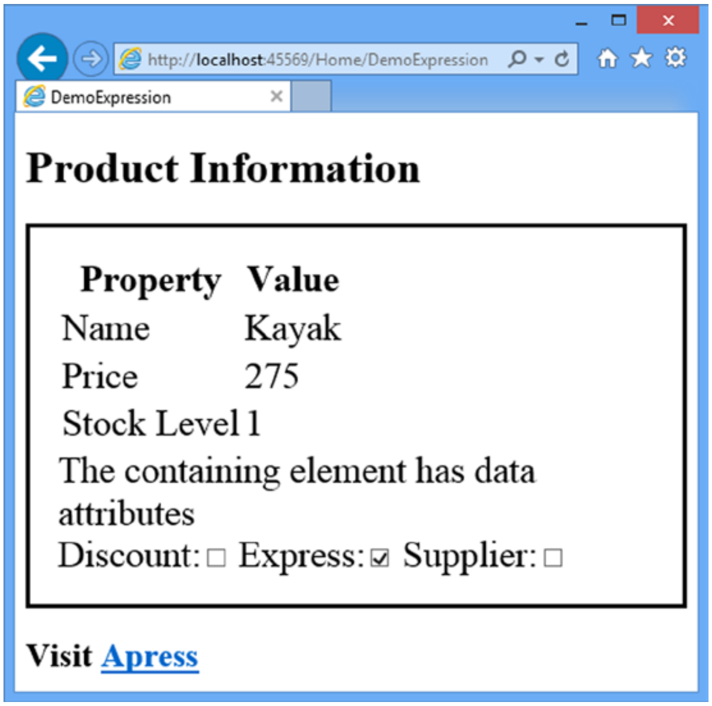
```
...
<div data-discount="False" data-express="True" data-supplier="">
    The containing element has data attributes
</div>
...
```

The `False` and `True` values correspond to the Boolean view bag values, but Razor has done something sensible for the property whose value is `null`, which is to render an empty string.

But things get interesting in the second set of additions to the view, which are a series of checkboxes whose `checked` attribute is set to the same view bag properties that I used for the data attributes. The HTML that Razor has rendered is as follows:

```
...
Discount: <input type="checkbox" />
Express:  <input type="checkbox" checked="checked" />
Supplier: <input type="checkbox" />
...
```

Razor is aware of the way that attributes such as `checked` are used, where the presence of the attribute rather than its value changes the configuration of the element (known as a *Boolean attribute* in the HTML specification). If Razor had inserted `False` or `null` or the empty string as the value of the `checked` attribute, then the checkbox that the browser displays would be checked. Instead, Razor has deleted the attribute from the element entirely when the value is `false` or `null`, creating an effect that is consistent with the view data, as shown in Figure 5-11.



Product Information

Property	Value
Name	Kayak
Price	275
Stock Level	1

The containing element has data attributes

Discount: ☐ Express: ☑ Supplier: ☐

Visit Apress

Figure 5-11. *The effect of deleting attributes whose presence configures their element*

Using Conditional Statements

Razor is able to process conditional statements, which means that I can tailor the output from a view based on values in the view data. This kind of technique is at the heart of Razor, and allows you to create complex and fluid layouts that are still reasonably simple to read and maintain. In Listing 5-16, I have updated the `DemoExpression.cshtml` view file to include a conditional statement.

Listing 5-16. Using a Conditional Razor Statement in the DemoExpression.cshtml File

```
@model Razor.Models.Product

@{
    ViewBag.Title = "DemoExpression";
    Layout = "~/Views/_BasicLayout.cshtml";
}

<table>
    <thead>
        <tr><th>Property</th><th>Value</th></tr>
    </thead>
    <tbody>
        <tr><td>Name</td><td>@Model.Name</td></tr>
        <tr><td>Price</td><td>@Model.Price</td></tr>
        <tr>
            <td>Stock Level</td>
            <td>
            @switch ((int)ViewBag.ProductCount) {
                case 0:
                    @: Out of Stock
                    break;
                case 1:
                    <b>Low Stock (@ViewBag.ProductCount)</b>
                    break;
                default:
                    @ViewBag.ProductCount
                    break;
            }
            </td>
        </tr>
    </tbody>
</table>
```

To start a conditional statement, you place an `@` character in front of the C# conditional keyword, which is `switch` in this example. You terminate the code block with a close brace character (`}`) just as you would with a regular C# code block.

■ **Tip** Notice that I had to cast the value of the `ViewBag.ProductCount` property to an `int` in order to use it with a `switch` statement. This is required because the Razor `switch` expression cannot evaluate a dynamic property–you must cast to a specific type so that it knows how to perform comparisons.

Inside of the Razor code block, you can include HTML elements and data values into the view output just by defining the HTML and Razor expressions, like this:

```
...
<b>Low Stock (@ViewBag.ProductCount)</b>
...
```

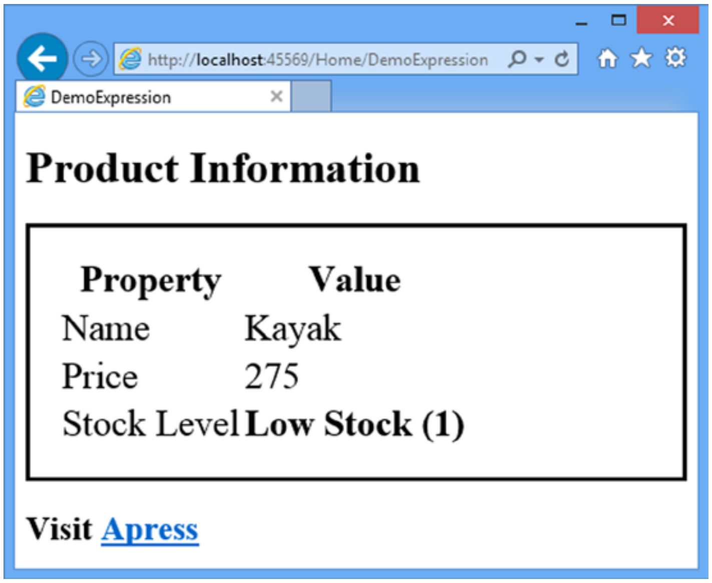
Or like this:

```
...
@ViewBag.ProductCount
...
```

I do not have to put the elements or expressions in quotes or denote them in any special way—the Razor engine will interpret these as output to be processed. However, if you want to insert literal text into the view when it is not contained in an HTML element, then you need to give Razor a helping hand and prefix the line like this:

```
...
@: Out of Stock
...
```

The `@:` characters prevent Razor from interpreting this as a C# statement, which is the default behavior when it encounters text. You can see the result of the conditional statement in Figure 5-12.



Product Information

Property	Value
Name	Kayak
Price	275
Stock Level	Low Stock (1)

Visit Apress

Figure 5-12. *Using a switch statement in a Razor view*

Conditional statements are important in Razor views because they allow content to be varied based on the data values that the view receives from the action method. As an additional demonstration, Listing 5-17 shows the addition of an `if` statement to the `DemoExpression.cshtml` view. As you might imagine, this is a commonly used conditional statement.

Listing 5-17. Using an if Statement in a Razor View in the DemoExpression.cshtml File

```
@model Razor.Models.Product

@{
    ViewBag.Title = "DemoExpression";
    Layout = "~/Views/_BasicLayout.cshtml";
}

<table>
    <thead>
        <tr><th>Property</th><th>Value</th></tr>
    </thead>
    <tbody>
        <tr><td>Name</td><td>@Model.Name</td></tr>
        <tr><td>Price</td><td>@Model.Price</td></tr>
        <tr>
            <td>Stock Level</td>
            <td>
                @if (ViewBag.ProductCount == 0) {
                    @:Out of Stock
                } else if (ViewBag.ProductCount == 1) {
                    <b>Low Stock (@ViewBag.ProductCount)</b>
                } else {
                    @ViewBag.ProductCount
                }
            </td>
        </tr>
    </tbody>
</table>
```

This conditional statement produces the same results as the `switch` statement, but I wanted to demonstrate how you can mesh C# conditional statements with Razor views. I explain how this all works in Chapter 20, when I explore views in depth.

Enumerating Arrays and Collections

When writing an MVC application, you will often want to enumerate the contents of an array or some other kind of collection of objects and generate content that details each one. To demonstrate how this is done, I have defined a new action method in the `Home` controller called `DemoArray` which you can see in Listing 5-18.

Listing 5-18. The DemoArray Action Method in the HomeController.cs File

```
using System.Web.Mvc;
using Razor.Models;

namespace Razor.Controllers {

    public class HomeController : Controller {

        Product myProduct = new Product {
            ProductID = 1,
            Name = "Kayak",
            Description = "A boat for one person",
            Category = "Watersports",
            Price = 275M
        };

        // ...other action methods omitted for brevity...

        public ActionResult DemoArray() {

            Product[] array = {
                new Product {Name = "Kayak", Price = 275M},
                new Product {Name = "Lifejacket", Price = 48.95M},
                new Product {Name = "Soccer ball", Price = 19.50M},
                new Product {Name = "Corner flag", Price = 34.95M}
            };
            return View(array);
        }
    }
}
```

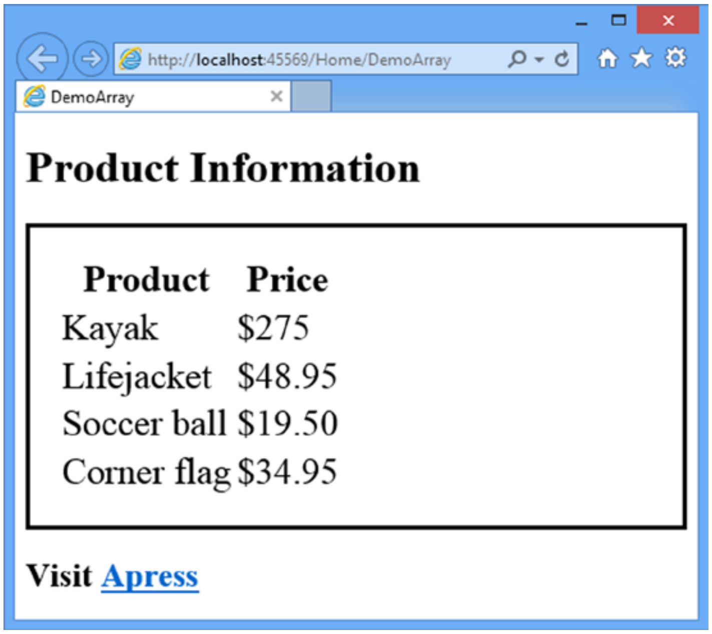
This action method creates a Product[] object that contains simple data values and passes them to the View method so that the data is rendered using the default view. The Visual Studio scaffold feature won't let you specify an array as a model type. (I don't know why, since Razor itself happily supports array.) To create a view for an action method that passes an array, the best approach is to create a view without a model and then manually add the @model expression after the file has been created. In Listing 5-19, you can see the contents of the DemoArray.cshtml file that I created this way in the Views/Home folder and then edited.

Listing 5-19. The Contents of the DemoArray.cshtml File

```
@model Razor.Models.Product[]

@{
    ViewBag.Title = "DemoArray";
    Layout = "~/Views/_BasicLayout.cshtml";
}

@if (Model.Length > 0) {
    <table>
        <thead><tr><th>Product</th><th>Price</th></tr></thead>
        <tbody>
            @foreach (Razor.Models.Product p in Model) {
                <tr>
                    <td>@p.Name</td>
                    <td>$@p.Price</td>
                </tr>
            }
        </tbody>
    </table>
} else {
    <h2>No product data</h2>
}
```

I used an @if statement to vary the content based on the length of the array that I receive as the view model and a @foreach expression to enumerate the contents of the array and generate a row in an HTML table for each of them. You can see how these expressions match their C# counterpart. You can also see how I created a local variable called p in the foreach loop and then referred to its properties using the Razor expressions @p.Name and @p.Price.

The result is that I generate an h2 element if the array is empty and produce one row per array item in an HTML table otherwise. Because my data is static in this example, you will always see the same result, which I have shown in Figure 5-13.



http://localhost:45569/Home/DemoArray

DemoArray

Product Information

Product	Price
Kayak	$275
Lifejacket	$48.95
Soccer ball	$19.50
Corner flag	$34.95

Visit Apress

Figure 5-13. Generating elements using a foreach statement

Dealing with Namespaces

You will notice that I had to refer to the `Product` class by its fully qualified name in the `foreach` loop in the last example, like this:

```
...
@foreach (Razor.Models.Product p in Model) {
...
```

This can become annoying in a complex view, where you will have many references to view model and other classes. I can tidy up my views by applying the `@using` expression to bring a namespace into context for a view, just like I would in a regular C# class. Listing 5-20 shows how I have applied the `@using` expression to the `DemoArray.cshtml` view.

Listing 5-20. Applying the @using Expression in the DemoArray.cshtml File

```
@using Razor.Models
@model Product[]

@{
    ViewBag.Title = "DemoArray";
    Layout = "~/Views/_BasicLayout.cshtml";
}
```

```
@if (Model.Length > 0) {
    <table>
        <thead><tr><th>Product</th><th>Price</th></tr></thead>
        <tbody>
            @foreach (Product p in Model) {
                <tr>
                    <td>@p.Name</td>
                    <td>$@p.Price</td>
                </tr>
            }
        </tbody>
    </table>
} else {
    <h2>No product data</h2>
}
```

A view can contain multiple `@using` expressions. I have used the `@using` expression to import the `Razor.Models` namespace, which means that I can remove the namespace from the `@model` expression and from within the `foreach` loop.

Summary

In this chapter, I have given you an overview of the Razor view engine and how it can be used to generate HTML. I showed you how to refer to data passed from the controller via the view model object and the view bag, and how Razor expressions can be used to tailor responses to the user based on data values. You will see many different examples of how Razor can be used in the rest of the book and I describe how the MVC view mechanism works in detail in Chapter 20. In the next chapter, I describe the essential development and testing tools that underpin the MVC Framework and that help you to get the best from your projects.