Introducing ASP.NET Web Pages 2 - Programming Basics

This tutorial gives you an overview of how to program in ASP.NET Web Pages with Razor syntax.

Level: Beginner to ASP.NET Web Pages
Skills assumed: HTML, CSS
Prerequisites:
ASP.NET Web Pages 2
WebMatrix 2 (download described in Getting Started With Web Pages)
Downloads: Completed website for the ASP.NET Web Pages introductory tutorial

What you'll learn:

- The basic "Razor" syntax that you use for programming in ASP.NET Web Pages.
- Some basic C#, which is the programming language you'll use.
- Some fundamental programming concepts for Web Pages.
- How to install packages (components that contain prebuilt code) to use with your site.
- How to use *helpers* to perform common programming tasks.

Features/technologies discussed:

- NuGet and the package manager.
- The `Twitter` helper.

This tutorial is primarily an exercise in introducing you to the programming syntax that you'll use for ASP.NET Web Pages. You'll learn about *Razor syntax* and code that's written in the C# programming language. You got a glimpse of this syntax in the previous tutorial; in this tutorial we'll explain the syntax more.

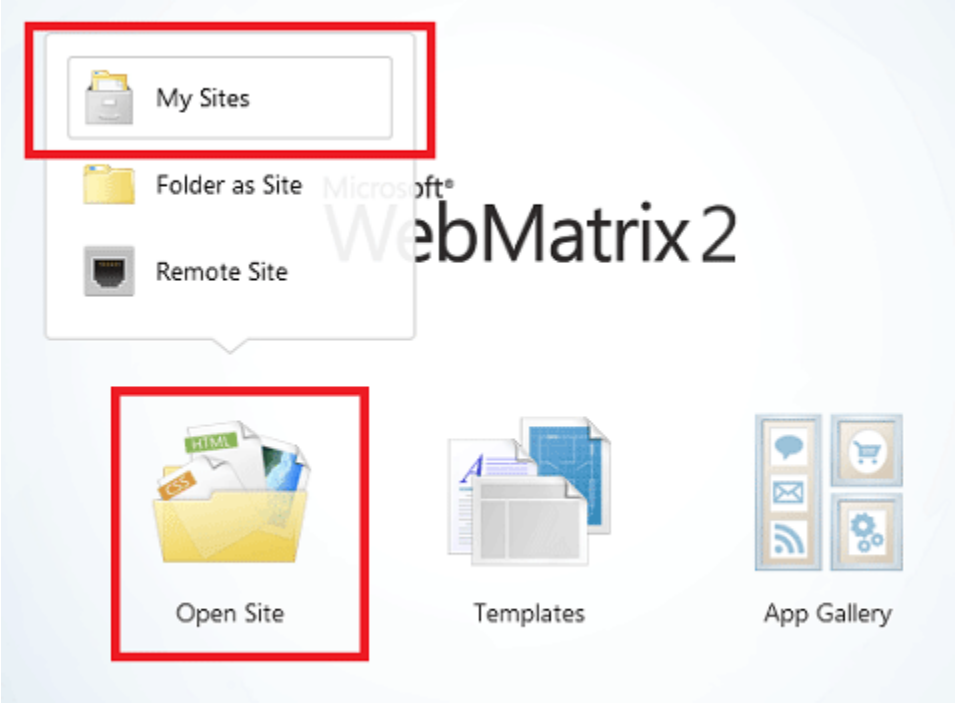
We promise that this tutorial involves the most programming that you'll see in a single tutorial, and that it's the only tutorial that is *only* about programming. In the remaining tutorials in this set, you'll actually create pages that do interesting things.

You'll also learn about *helpers*. A helper is a component — a packaged-up piece of code — that you can add to a page. The helper performs work for you that otherwise might be tedious or complex to do by hand.

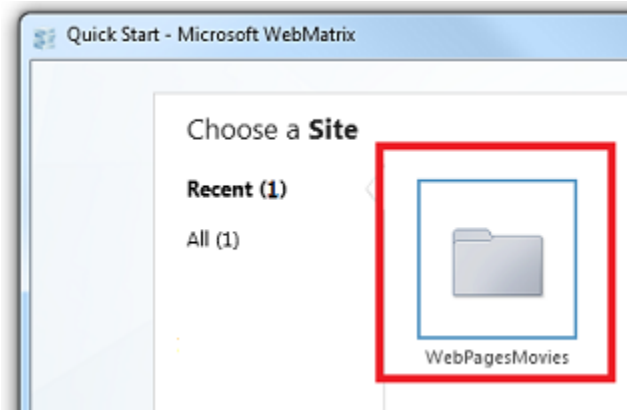
Creating a Page to Play with Razor

In this section you'll play a bit with Razor so you can get a sense of the basic syntax.

Start WebMatrix if it's not already running. You'll use the website you created in the previous tutorial (Getting Started With Web Pages). To reopen it, click **Open Site** and choose **My Sites**:



Choose the **WebPagesMovies** site, and then click **OK**.



Select the **Files** workspace.

In the ribbon, click **New** to create a page. Select **CSHTML** and name the new page *TestRazor.cshtml*.

Click **OK**.

Copy the following into the file, completely replacing what's there already.

Note When you copy code or markup from the examples into a page, the indentation and alignment might not be the same as in the tutorial. Indentation and alignment don't affect how the code runs, though.

```
@{
   // Working with numbers
   var a = 4;
   var b = 5;
   var theSum = a + b;

   // Working with characters (strings)
   var technology = "ASP.NET";
   var product ="Web Pages";

   // Working with objects
   var rightNow = DateTime.Now;
}

<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Testing Razor Syntax</title>
    <meta charset="utf-8" />
    <style>
    body {font-family:Verdana; margin-left:50px; margin-top:50px;}
    div {border: 1px solid black; width:50%; margin:1.2em;padding:1em;}
    span.bright {color:red;}
    </style>
  </head>
<body>
  <h1>Testing Razor Syntax</h1>
  <form method="post">

    <div>
      <p>The value of <em>a</em> is @a.  The value of <em>b</em> is @b.
      <p>The sum of <em>a</em> and <em>b</em> is <strong>@theSum</strong>.</p>
```

```
      <p>The product of <em>a</em> and <em>b</em> is <strong>@(a*b)</strong>.</p>
    </div>

    <div>
      <p>The technology is @technology, and the product is @product.</p>
      <p>Together they are <span class="bright">@(technology + " " +
product)</span></p>
   </div>

   <div>
     <p>The current date and time is: @rightNow</p>
     <p>The URL of the current page is<br/><br/><code>@Request.Url</code></p>
   </div>

  </form>
</body>
</html>
```

Examining the Example Page

Most of what you see is ordinary HTML. However, at the top there's this code block:

```
@{
   // Working with numbers.
   var a = 4;
   var b = 5;
   var theSum = a + b;

   // Working with characters (strings).
   var technology = "ASP.NET";
   var product ="Web Pages";

   // Working with objects.
   var rightNow = DateTime.Now;
}
```

Notice the following things about this code block:

- The `@` character tells ASP.NET that what follows is Razor code, not HTML. ASP.NET will treat everything after the `@` character as code until it runs into some HTML again. (In this case, that's the `<!DOCTYPE>` element.
- The braces (`{` and `}`) enclose a block of Razor code if the code has more than one line. The braces tell ASP.NET where the code for that block starts and ends.
- The `//` characters mark a comment — that is, a part of the code that won't execute.
- Each statement has to end with a semicolon (`;`). (Not comments, though.)
- You can store values in *variables*, which you create (*declare*) with the keyword `var`. When you create a variable, you give it a name, which can include letters, numbers, and underscore (_). Variable names can't start with a number and can't use the name of a programming keyword (like `var`).
- You enclose character strings (like "ASP.NET" and "Web Pages") in quotation marks. (They must be double quotation marks.) Numbers are not in quotation marks.
- Whitespace outside of quotation marks doesn't matter. Line breaks mostly don't matter; the exception is that you can't split a string in quotation marks across lines. Indentation and alignment don't matter.

Something that's not obvious from this example is that all code is case sensitive. This means that the variable `TheSum` is a different variable than variables that might be named `theSum` or `thesum`. Similarly, `var` is a keyword, but `Var` is not.

Objects and properties and methods

Then there's the expression `DateTime.Now`. In simple terms, `DateTime` is an *object*. An object is a thing that you can program with—a page, a text box, a file, an image, a web request, an email message, a customer record, etc. Objects have one or more *properties* that describe their characteristics. A text box object has a `Text` property (among others), a request object has a `Url` property (and others), an email message has a `From` property and a `To` property, and so on. Objects also have *methods* that are the "verbs" they can perform. You'll be working with objects a lot.

As you can see from the example, `DateTime` is an object that lets you program dates and times. It has a property named `Now` that returns the current date and time.

Using code to render markup in the page

In the body of the page, notice the following:

```
<div>
  <p>The value of <em>a</em> is @a.  The value of <em>b</em> is @b.
  <p>The sum of <em>a</em> and <em>b</em> is <strong>@theSum</strong>.</p>
  <p>The product of <em>a</em> and <em>b</em> is <strong>@(a*b)</strong>.</p>
</div>

<div>
  <p>The technology is @technology, and the product is @product.</p>
  <p>Together they are <span class="bright">@(technology + " " + product)</span></p>
</div>

<div>
  <p>The current date and time is: @rightNow</p>
     <p>The URL of the current page is<br/><br/><code>@Request.Url</code></p>
</div>
```

Again, the `@` character tells ASP.NET that what follows is code, not HTML. In the markup you can add `@` followed by a code expression, and ASP.NET will render the value of that expression right at that point. In the example, `@a` will render whatever the value is of the variable named `a`, `@product` renders whatever is in the variable named `product`, and so on.

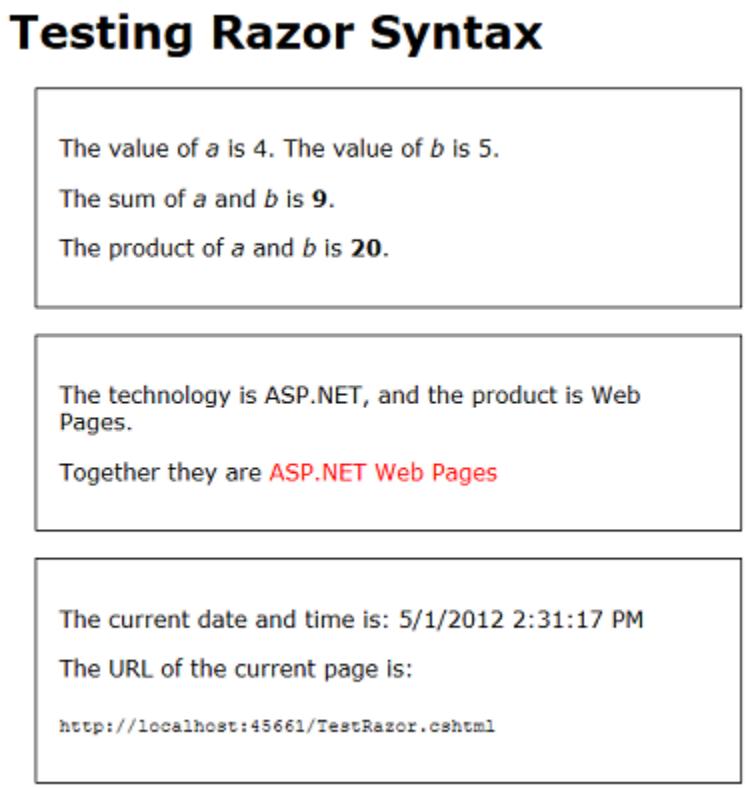
You're not limited to variables, though. In a few instances here, the `@` character precedes an expression:

- `@(a*b)` renders the product of whatever is in the variables `a` and `b`. (The `*` operator means multiplication.)
- `@(technology + " " + product)` renders the values in the variables `technology` and `product` after concatenating them and adding a space in between. The operator (`+`) for concatenating strings is the same as the operator for adding numbers. ASP.NET can usually tell whether you're working with numbers or with strings and does the right thing with the `+` operator.
- `@Request.Url` renders the `Url` property of the `Request` object. The `Request` object contains information about the current request from the browser, and of course the `Url` property contains the URL of that current request.

The example is also designed to show you that you can do work in different ways. You can do calculations in the code block at the top, put the results into a variable, and then render the variable in markup. Or you can do calculations in an expression right in the markup. The approach you use depends on what you're doing and, to some extent, on your own preference.

Seeing the code in action

Right-click the name of the file and then choose **Launch in browser**. You see the page in the browser with all the values and expressions resolved in the page.



Look at the source in the browser.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Testing Razor Syntax</title>
    <meta charset="utf-8" />
    <style>
    body {font-family:Verdana; margin-left:50px; margin-top:50px;}
    div {border: 1px solid black; width:50%; margin:1.2em;padding:1em;}
    span.bright {color:red;}
    </style>
  </head>
<body>
  <h1>Testing Razor Syntax</h1>
  <form method="post">

    <div>
      <p>The value of <em>a</em> is 4.  The value of <em>b</em> is 5.
      <p>The sum of <em>a</em> and <em>b</em> is <strong>9</strong></p>
      <p>The product of <em>a</em> and <em>b</em> is <strong>20</strong></p>
    </div>

    <div>
      <p>The technology is ASP.NET, and the product is Web Pages.</p>
      <p>Together they are <span class="bright">ASP.NET Web Pages</span></p>
   </div>

   <div>
     <p>The current date and time is: 4/29/2012 11:44:02 PM</p>
     <p>The URL of the current page is <code>http://localhost:45661/TestRazor.cshtml</code></p>
   </div>

  </form>
</body>
</html>
```

As you expect from your experience in the previous tutorial, none of the Razor code is in the page. All you see are the actual display values. When you run a page, you're actually making a request to the web server that's built into WebMatrix. When the request is received, ASP.NET resolves all the values and expressions and renders their values into the page. It then sends the page to the browser.

Razor and C#

Up to now we've said that you're working with Razor syntax. That's true, but it's not the complete story. The actual programming language you're using is called *C#*. C# was created by Microsoft over a decade ago and has become one of the primary programming languages for creating Windows apps. All the rules you've seen about how to name a variable and how to create statements and so on are actually all rules of the C# language.

Razor refers more specifically to the small set of conventions for how you embed this code into a page. For example, the convention of using `@` to mark code in the page and using `@{ }` to embed a code block is the Razor aspect of a page. Helpers are also considered to be part of Razor. Razor syntax is used in more places than just in ASP.NET Web Pages. (For example, it's used in ASP.NET MVC views as well.)

We mention this because if you look for information about programming ASP.NET Web Pages, you'll find lots of references to Razor. However, a lot of those references don't apply to what you're doing and might therefore be confusing. And in fact, many of your programming questions are really going to be about either working

with C# or working with ASP.NET. So if you look specifically for information about Razor, you might not find the answers you need.

Adding Some Conditional Logic

One of the great features about using code in a page is that you can change what happens based on various conditions. In this part of the tutorial, you'll play around with some ways to change what's displayed in the page.

The example will be simple and somewhat contrived so that we can concentrate on the conditional logic. The page you'll create will do this:

- Show different text on the page depending on whether it's the first time the page is displayed or whether you've clicked a button to submit the page. That will be the first conditional test.
- Display the message only if a certain value is passed in the query string of the URL (`http://...?show=true`). That will be the second conditional test.

In WebMatrix, create a page and name it *TestRazorPart2.cshtml*. (In the ribbon, click **New**, choose **CSHTML**, name the file, and then click **OK**.)

Replace the contents of that page with the following:

```
@{
   var message = "This is the first time you've requested the page.";
}
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Testing Razor Syntax - Part 2</title>
    <meta charset="utf-8" />
    <style>
      body {font-family:Verdana; margin-left:50px; margin-top:50px;}
      div {border: 1px solid black; width:50%; margin:1.2em;padding:1em;}
    </style>
  </head>
  <body>
  <h1>Testing Razor Syntax - Part 2</h1>
  <form method="post">
    <div>
```

```
      <p>@message</p>
      <p><input type="submit" value="Submit" /></p>
  </div>
  </form>
</body>
</html>
```

The code block at the top initializes a variable named `message` with some text. In the body of the page, the contents of the `message` variable are displayed inside a `<p>` element. The markup also contains an `<input>` element to create a **Submit** button.

Run the page to see how it works now. For now, it's basically a static page, even if you click the **Submit** button.

Go back to WebMatrix. Inside the code block, add the following code *after* the line that initializes `message`:

```
if(IsPost){
   message = "Now you've submitted the page.";
}
```

The if{ } block

What you just added was an `if` condition. In code, the `if` condition has a structure like this:

```
if(some condition){
    One or more statements here that run if the condition is true;
}
```

The condition to test is in parentheses. It has to be a value or an expression that returns true or false. If the condition is true, ASP.NET runs the statement or statements that are inside the braces. (Those are the *then* part of the *if-then* logic.) If the condition is false, the block of code is skipped.

Here are a few examples of conditions you can test in an `if` statement:

```
if( currentValue > 12 ){ ... }

if( dueDate <= DateTime.Today ) { ... }

if( IsDone == true ) { ... }
```

```
if( IsPost ) { ... }

if( !IsPost ) { ... }

if(a != 0) { ... }

if( fileProcessingIsDone != true && displayMessage == false ) { ... }
```

You can test variables against values or against expressions by using a *logical operator* or *comparison operator*: equal to (`==`), greater than (`>`), less than (`<`), greater than or equal to (`>=`), and less than or equal to (`<=`). The `!=` operator means not equal to — for example, `if(a != 0)` means *if `a` is not equal to 0*.

Note Make sure you notice that the comparison operator for equals to (`==`) is not the same as `=`. The `=` operator is used only to assign values (`var a=2`). If you mix these operators up, you'll either get an error or you'll get some strange results.

To test whether something is true, the complete syntax is `if(IsDone == true)`. But you can also use the shortcut `if(IsDone)`. If there's no comparison operator, ASP.NET assumes that you're testing for true.

The `!` operator by itself means a logical NOT. For example, the condition `if(!IsPost)` means *if `IsPost` is not true*.

You can combine conditions by using a logical AND (`&&` operator) or logical OR (`||` operator). For example, the last of the `if` conditions in the preceding examples means *if `fileProcessingIsDone` is set to true AND `displayMessage` is set to false*.

The else block

One final thing about `if` blocks: an `if` block can be followed by an `else` block. An `else` block is useful is you have to execute different code when the condition is false. Here's a simple example:

```
var message = "";
if(errorOccurred == true)
{
    message = "Sorry, an error occurred.";
}
else
```

```
{
     message = "The process finished without errors!";
}
```

You'll see some examples in later tutorials in this series where using an `else` block is useful.

Testing whether the request is a submit (post)

There's more, but let's get back to the example, which has the condition `if(IsPost){ ... }`. `IsPost` is actually a property of the current page. The first time the page is requested, `IsPost` returns false. However, if you click a button or otherwise submit the page — that is, you post it — `IsPost` returns true. So `IsPost` lets you determine whether you're dealing with a form submission. (In terms of HTTP verbs, if the request is a GET operation, `IsPost` returns false. If the request is a POST operation, `IsPost` returns true.) In a later tutorial you'll work with input forms, where this test becomes particularly useful.

Run the page. Because this is the first time you're requested the page, you see "This is the first time you've requested the page". That string is the value that you initialized the `message` variable to. There's an `if(IsPost)` test, but that returns false at the moment, so the code inside the `if` block doesn't run.

Click the **Submit** button. The page is requested again. As before, the `message` variable is set to "This is the first time ...". But this time, the test `if(IsPost)` returns true, so the code inside the `if` block runs. The code changes the value of the `message` variable to a different value, which is what's rendered in the markup.

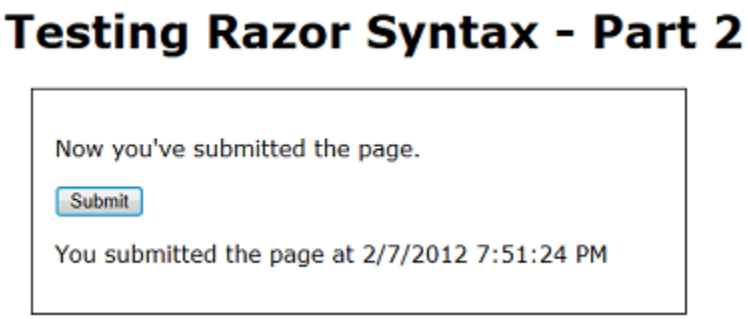
Now add an `if` condition in the markup. Below the `<p>` element that contains the **Submit** button, add the following markup:

```
@if(IsPost){
  <p>You submitted the page at @DateTime.Now</p>
}
```

You're adding code inside the markup, so you have to start with `@`. Then there's an `if` test similar to the one you added earlier up in the code block. Inside the braces, though, you're adding ordinary HTML — at least, it's ordinary until it gets to `@DateTime.Now`. This is another little bit of Razor code, so again you have to add `@` in front of it.

The point here is that you can add `if` conditions in both the code block at the top and in the markup. If you use an `if` condition in the body of the page, the lines inside the block can be markup or code. In that case, and as is true anytime you mix markup and code, you have to use `@` to make it clear to ASP.NET where the code is.

Run the page and click **Submit**. This time you not only see a different message when you submit ("Now you've submitted ..."), but you see a new message that lists the date and time.



Testing the value of a query string

One more test. This time, you'll add an `if` block that tests a value named `show` that might be passed in the query string. (Like this: `http://localhost:43097/TestRazorPart2.cshtml?show=true`) You'll change the page so that the message you've been displaying ("This is the first time ...", etc.) is only displayed if the value of `show` is true.

At the bottom (but inside) the code block at the top of the page, add the following:

```
var showMessage = false;
if(Request.QueryString["show"].AsBool() == true){
    showMessage = true;
}
```

The complete code block now look like the following example. (Remember that when you copy the code into your page, the indentation might look different. But that doesn't affect how the code runs.)

```
@{
   var message = "This is the first time you've requested the page.";

   if(IsPost){
      message = "Now you've submitted the page.";
   }

   var showMessage = false;
   if(Request.QueryString["show"].AsBool() == true){
```

```
      showMessage = true;
    }
}
```

The new code in the block initializes a variable named `showMessage` to false. It then does an `if` test to look for a value in the query string. When you first request the page, it has a URL like this one:

`http://localhost:43097/TestRazorPart2.cshtml`

The code determines whether the URL contains a variable named `show` in the query string, like this version of the URL:

`http://localhost:43097/TestRazorPart2.cshtml?show=true`

The test itself looks at the `QueryString` property of the `Request` object. If the query string contains an item named `show`, and if that item is set to true, the `if` block runs and sets the `showMessage` variable to true.

There's a trick here, as you can see. Like the name says, the query string is a string. However, you can only test for true and false if the value you're testing is a Boolean (true/false) value. Before you can test the value of the `show` variable in the query string, you have to convert it to a Boolean value. That's what the `AsBool` method does — it takes a string as input and converts it to a Boolean value. Clearly, if the string is "true", the `AsBool` method converts that value to true. If the value of the string is anything else, `AsBool` returns `false`.

Data Types and As( ) Methods

We've only said so far that when you create a variable, you use the keyword `var`. That's not the entire story, though. In order to manipulate values — to add numbers, or concatenate strings, or compare dates, or test for true/false — C# has to work with an appropriate internal representation of the value. C# can *usually* figure out what that representation should be (that is, what *type* the data is) based on what you're doing with the values. Now and then, though, it can't do that. If not, you have to help out by explicitly indicating how C# should represent the data. The `AsBool` method does that — it tells C# that a string value of `"true"` or `"false"` should be treated as a Boolean value. Similar methods exist to represent strings as other types as well, like `AsInt` (treat as an integer), `AsDateTime` (treat as a date/time), `AsFloat` (treat as a floating-point number), and so on. When you use these `As( )` methods, if C# can't represent the string value as requested, you'll see an error.

In the markup of the page, remove or comment out this element (here it's shown commented out):

```
<!-- <p>@message</p> -->
```

Right where you removed or commented out that text, add the following:

```
@if(showMessage){

  <p>@message</p>

}
```

The `if` test says that if the `showMessage` variable is true, render a `<p>` element with the value of the `message` variable.

Summary of your conditional logic

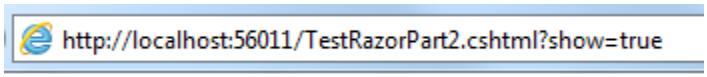
In case you're not entirely sure of what you've just done, here's a summary.

- The `message` variable is initialized to a default string ("This is the first time ...").
- If the page request is the result of a submit (post), the value of `message` is changed to "Now you've submitted ..."
- The `showMessage` variable is initialized to false.
- If the query string contains `?show=true` , the `showMessage` variable is set to true.
- In the markup, if `showMessage` is true, a `<p>` element is rendered that shows the value of `message`. (If `showMessage` is false, nothing is rendered at that point in the markup.)
- In the markup, if the request is a post, a `<p>` element is rendered that displays the date and time.

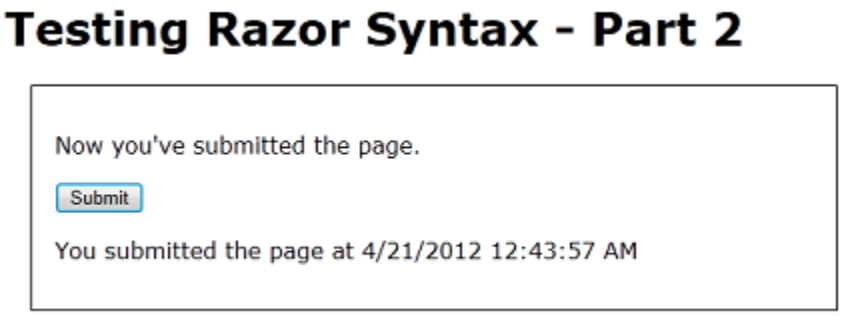
Run the page. There's no message, because `showMessage` is false, so in the markup the `if(showMessage)` test returns false.

Click **Submit**. You see the date and time, but still no message.

In your browser, go to the URL box and add the following to the end of the URL: `?show=true` and then press Enter.



The page is displayed again. (Because you changed the URL, this is a new request, not a submit.) Click **Submit** again. The message is displayed again, as is the date and time.



In the URL, change `?show=true` to `?show=false` and press Enter. Submit the page again. The page is back to how you started — no message.

As noted earlier, the logic of this example is a little contrived. However, `if` is going to come up in many of your pages, and it will take one or more of the forms you've seen here.

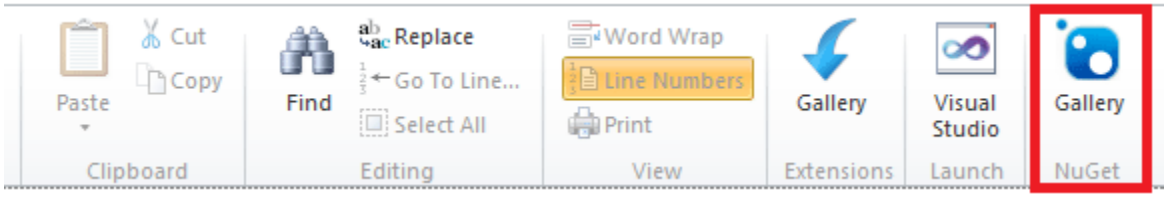
Installing a Helper (Displaying a Twitter Feed)

Some tasks that people often want to do on web pages require a lot of code or require extra knowledge. Examples: displaying a chart for data; putting a Facebook "Like" button on a page; sending email from your website; cropping or resizing images; using PayPal for your site. To make it easy to do these kinds of things, ASP.NET Web Pages lets you use *helpers*. Helpers are components that you install for a site and that let you perform typical tasks by using just a few lines of Razor code.

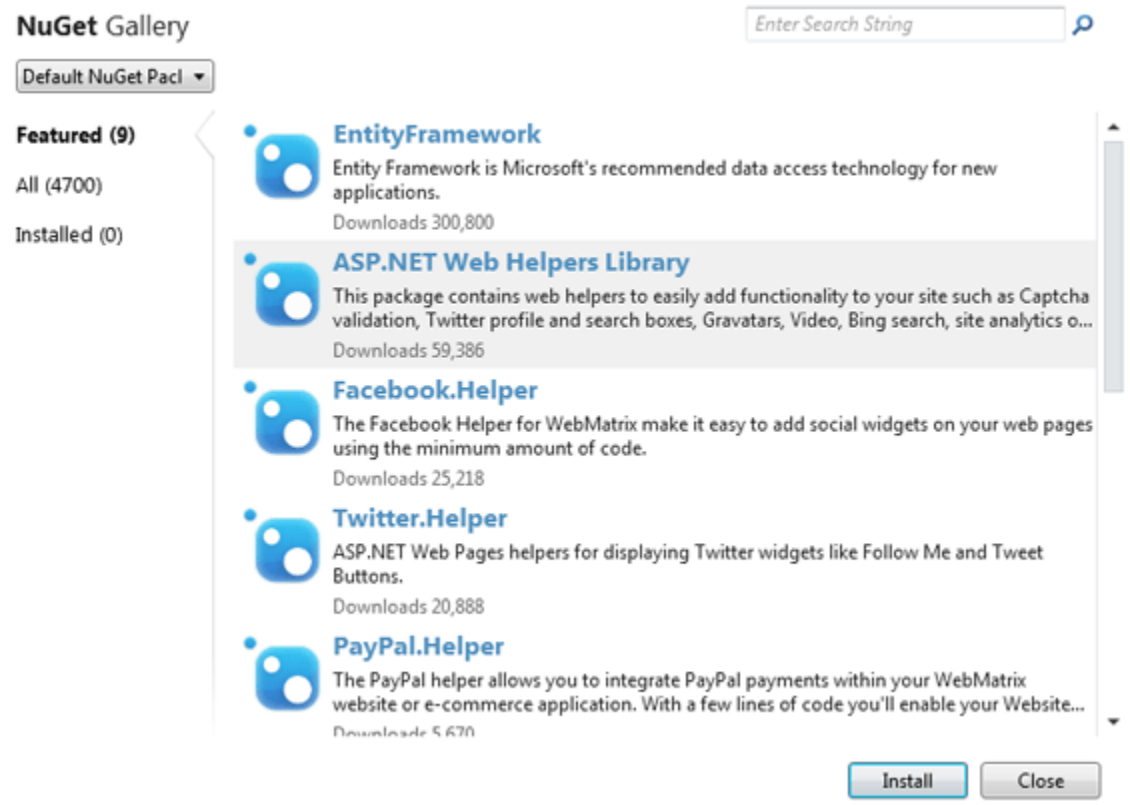
ASP.NET Web Pages has a few helpers built in. However, many helpers are available in packages (add-ins) that are provided using the NuGet package manager. NuGet lets you select a package to install and then it takes care of all the details of the installation.

In this part of the tutorial, you'll install a helper that lets you manage a Twitter feed. You'll learn two things. One is how to find and install a helper. You'll also learn how a helper makes it easy to do something you'd otherwise need to do by using a lot of code you'd have to write yourself.

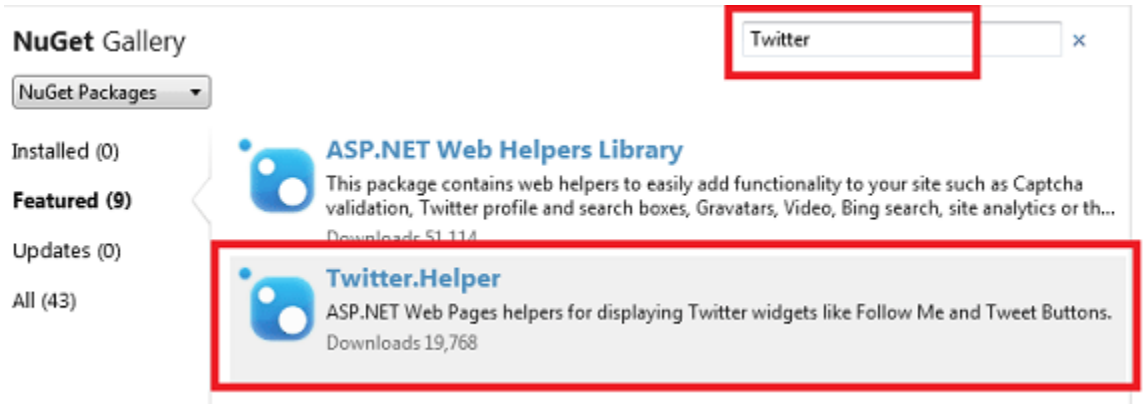
In WebMatrix, click the **Gallery** button.



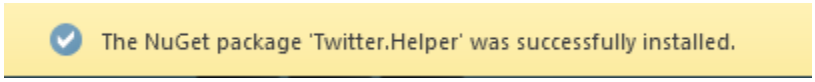
This launches the NuGet package manager and displays available packages. (Not all of the packages are helpers; some add functionality to WebMatrix itself, some are additional templates, and so on.)



In the search box, enter "Twitter". NuGet shows the packages that have Twitter functionality. (The link underneath the package icon links to details about that package.)



Select the **Twitter.Helper** package and then click **Install** to launch the installer. When it's done, you see a message in the notification area at the bottom of the screen.



That's it. NuGet downloads and installs everything, including any additional components that might be required (*dependencies*). Since this is the first time you've installed a helper, NuGet also creates folders in your website for the code that makes up the helper.

If for some reason you have to uninstall a helper, the process is very similar. Click the **Gallery** button, click the **Installed** tab, and pick the package you want to uninstall.

Using a Helper in a Page

Now you'll use the Twitter helper that you just installed. The process for adding a helper to a page is similar for most helpers.

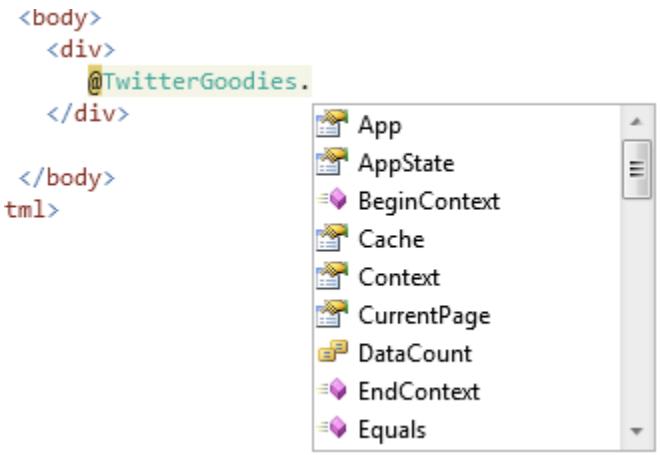
In WebMatrix, create a page and name it *TwitterTest.cshml*. (You're creating a special page to test the helper, but you can use helpers in any page in your site.)

Inside the `<body>` element, add a `<div>` element. Inside the `<div>` element, type this:

```
@TwitterGoodies.
```

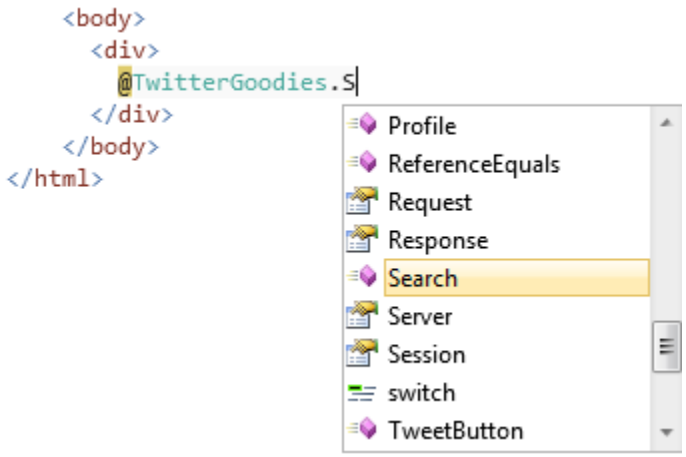
The `@` character is the same character you've been using to mark Razor code. `TwitterGoodies` is the helper object that you're working with.

As soon as you type the period (.), WebMatrix displays a list of *methods* (functions) that the `TwitterGoodies` helper makes available:



This feature is known as *IntelliSense*. It helps you code by providing context-appropriate choices. IntelliSense works with HTML, CSS, ASP.NET code, JavaScript, and other languages that are supported in WebMatrix. It's another feature that makes it easier to develop web pages in WebMatrix.

Press S on the keyboard, and you see that IntelliSense finds the Search method:

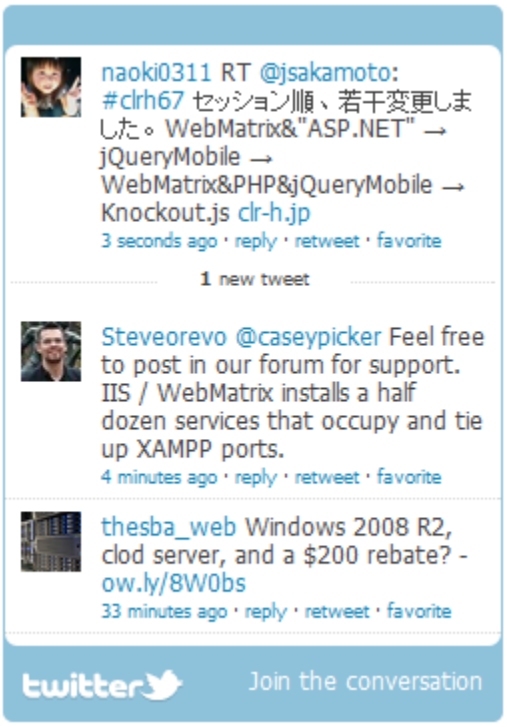


Press Tab. IntelliSense inserts the selected method (`Search`) for you. Type an open parenthesis (`(`), then the string `"webmatrix"` in quotation marks, then a closing parenthesis (`)`). When you're done, the line looks like this:

```
@TwitterGoodies.Search("webmatrix")
```

The `Search` method finds tweets that contain the string that you specify — in this case, it will look for tweets that mention "webmatrix". (Either in text or in hashtags.)

Run the page. You see a Twitter feed. (It might take a few moments for the feed to start populating.)



To get an idea of what the helper is doing for you, view the source of the page in the browser. Along with the HTML that you had in your page, you see a block of JavaScript code that looks roughly like the following block. (It might be all on one line or otherwise compressed.)

```
<script> new TWTR.Widget({ version: 2, type: 'search', search: 'webmatrix', interval: 6000, title: '', subject: '', width: 250, height: 300, theme: { shell: { background: '#8ec1da', color: '#ffffff' }, tweets: { background: '#ffffff', color: '#444444', links: '#1985b5' } }, features: { scrollbar: false, loop: true, live: true, hashtags: true, timestamp: true, avatars: true, toptweets: true, behavior: 'all' } }).render().start();< /script>
```

This is code that the helper rendered into the page at the place where you had `@TwitterGoodies.Search`. (There's also some markup that's not shown here.) The helper took the information you provided and generated the code that talks directly to Twitter in order to get back the Twitter feed that you see. If you know the Twitter programming interface (API), you can create this code yourself. But because the helper can do it for you, you don't have to know the details of how to communicate with Twitter. And even if you are familiar with the Twitter API, it's a lot easier to include the `TwitterGoodies` helper on the page and let it do the work.

Return to the page. At the bottom, inside the `<body>` element, add the following code. Substitute your own Twitter account name if you have one.

```
<div>
    @TwitterGoodies.FollowButton("microsoft")
</div>
```

This code calls the `FollowButton` method of the `TwitterGoodies` helper. As you can guess, the method adds a **Follow Me on Twitter** button. You pass a Twitter name to this method to indicate who to follow.

Run the page and you see the **Follow Me** button:

Click it, and you go to the Twitter page for the user you specified.

As before, you can look at the source of the page in the browser to see what the `Twitter` helper generated for you. This time the code looks something like the following example:

```
<a href="http://www.twitter.com/microsoft"><img src="http://twitter-
badges.s3.amazonaws.com/follow_me-a.png" alt="Follow microsoft on Twitter"/></a>
```

Again, you could have written this code yourself, but the helper makes it much easier.

Server-Side (Razor) and Client-Side (JavaScript) Programming

How does Razor code in an ASP.NET Web Pages relate to JavaScript code that runs in the browser? If you've got experience with JavaScript, you might realize as you work with these tutorials that many of the tasks could also be done in JavaScript. That's true, especially with the simple examples you've seen so far.

A *.cshtml* page can contain both Razor code and JavaScript code. The traditional division of labor has been that server code handled tasks that it made sense to run on the server. This included accessing resources like a shared database and performing various types of business logic. In contrast, client code has typically been used to create a rich user experience. Pop-up calendars, sliders, animations, and many other UI effects are created by client code, and can all be done easily using JavaScript libraries (especially jQuery).

These days the distinction has blurred a little because client code libraries now let you communicate with the server in ways that formerly could only be done in server code. In general, though, it's still useful to think of server code (Razor and C#) as being for back-end work and client code (JavaScript) as useful for UI. In subsequent tutorial sets you'll learn how to integrate JavaScript into a *.cshtml* page for just this purpose, namely to create a lively user experience.

Coming Up Next

To keep this tutorial short, we had to focus on only a few basics. Naturally, there's a *lot* more to Razor and C#. You'll learn more as you go through these tutorials. If you're interested in learning more about the programming aspects of Razor and C# right now, you can read a more thorough introduction here: Introduction to ASP.NET Web Programming Using the Razor Syntax.

The next tutorial introduces you to working with a database. In that tutorial, you'll begin creating the sample application that lets you list your favorite movies.

Complete Listing for TestRazor Page

```
@{
    // Working with numbers
    var a = 4;
    var b = 5;
    var theSum = a + b;

    // Working with characters (strings)
    var technology = "ASP.NET";
    var product ="Web Pages";

    // Working with objects
    var rightNow = DateTime.Now;
}

<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Testing Razor Syntax</title>
    <meta charset="utf-8" />
    <style>
    body {font-family:Verdana; margin-left:50px; margin-top:50px;}
    div {border: 1px solid black; width:50%; margin:1.2em;padding:1em;}
    span.bright {color:red;}
    </style>
  </head>
<body>
```

```
  <h1>Testing Razor Syntax</h1>
  <form method="post">

    <div>
      <p>The value of <em>a</em> is @a.  The value of <em>b</em> is @b.
      <p>The sum of <em>a</em> and <em>b</em> is <strong>@theSum</strong>.</p>
      <p>The product of <em>a</em> and <em>b</em> is <strong>@(a*b)</strong>.</p>
    </div>

    <div>
      <p>The technology is @technology, and the product is @product.</p>
      <p>Together they are <span class="bright">@(technology + " " +
product)</span></p>
   </div>

   <div>
     <p>The current date and time is: @rightNow</p>
     <p>The URL of the current page is<br/><br/><code>@Request.Url</code></p>
   </div>

  </form>
</body>
</html>
```

Complete Listing for TestRazorPart2 Page

```
@{
   var message = "This is the first time you've requested the page.";

   if(IsPost){
      message = "Now you've submitted the page.";
   }

   var showMessage = false;
   if(Request.QueryString["show"].AsBool() == true){
     showMessage = true;
   }
}
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Testing Razor Syntax - Part 2</title>
    <meta charset="utf-8" />
    <style>
      body {font-family:Verdana; margin-left:50px; margin-top:50px;}
      div {border: 1px solid black; width:50%; margin:1.2em;padding:1em;}
    </style>
  </head>
  <body>
  <h1>Testing Razor Syntax - Part 2</h1>
  <form method="post">
    <div>
      <!--<p>@message</p>-->
      @if(showMessage){
        <p>@message</p>
      }
      <p><input type="submit" value="Submit" /></p>
      @if(IsPost){
        <p>You submitted the page at @DateTime.Now</p>
      }
    </div>
  </form>
</body>
</html>
```

Complete Listing for TwitterTest Page

```
@{
}
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title></title>
    </head>
```

```
    <body>
        <div>
        @TwitterGoodies.Search("webmatrix")
        </div>

        <div>
        @TwitterGoodies.FollowButton("microsoft")
        </div>
    </body>
</html>
```

Introduction to ASP.NET Web Programming Using the Razor Syntax (C#)

This article gives you an overview of programming with ASP.NET Web Pages using the Razor syntax. ASP.NET is Microsoft's technology for running dynamic web pages on web servers. This articles focuses on using the C# programming language.

What you'll learn:

- The top 8 programming tips for getting started with programming ASP.NET Web Pages using Razor syntax.
- Basic programming concepts you'll need.
- What ASP.NET server code and the Razor syntax is all about.

Note The information in this article applies to ASP.NET Web Pages 1.0 and Web Pages 2. Where there are differences between versions, the text describes the differences.

The Top 8 Programming Tips

This section lists a few tips that you absolutely need to know as you start writing ASP.NET server code using the Razor syntax.

Note The Razor syntax is based on the C# programming language, and that's the language that's used most often with ASP.NET Web Pages. However, the Razor syntax also supports the Visual Basic language, and

everything you see you can also do in Visual Basic. For details, see the appendix Visual Basic Language and Syntax.

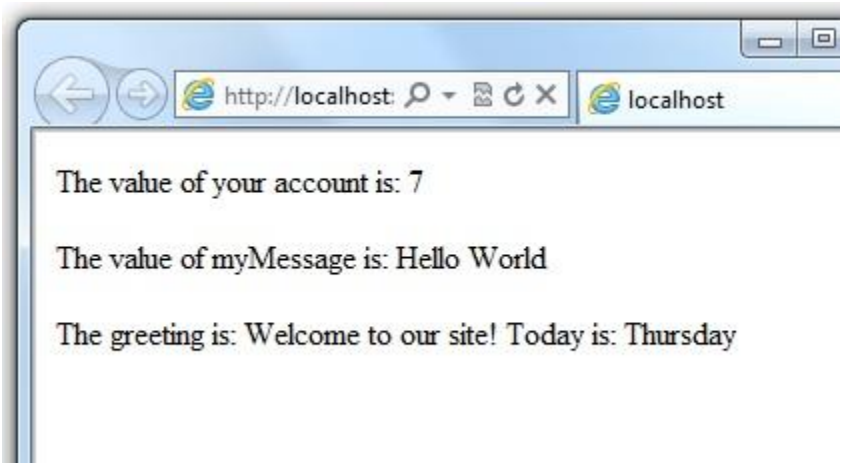
You can find more details about most of these programming techniques later in the article.

1. You add code to a page using the @ character

The @ character starts inline expressions, single statement blocks, and multi-statement blocks:

```
<!-- Single statement blocks  -->
@{ var total = 7; }
@{ var myMessage = "Hello World"; }

<!-- Inline expressions -->
<p>The value of your account is: @total </p>
<p>The value of myMessage is: @myMessage</p>

<!-- Multi-statement block -->
@{
    var greeting = "Welcome to our site!";
    var weekDay = DateTime.Now.DayOfWeek;
    var greetingMessage = greeting + " Today is: " + weekDay;
}
<p>The greeting is: @greetingMessage</p>
```

This is what these statements look like when the page runs in a browser:



HTML Encoding

When you display content in a page using the `@` character, as in the preceding examples, ASP.NET HTML-encodes the output. This replaces reserved HTML characters (such as `<` and `>` and `&`) with codes that enable the characters to be displayed as characters in a web page instead of being interpreted as HTML tags or entities. Without HTML encoding, the output from your server code might not display correctly, and could expose a page to security risks.

If your goal is to output HTML markup that renders tags as markup (for example `<p></p>` for a paragraph or `<em></em>` to emphasize text), see the section Combining Text, Markup, and Code in Code Blocks later in this article.

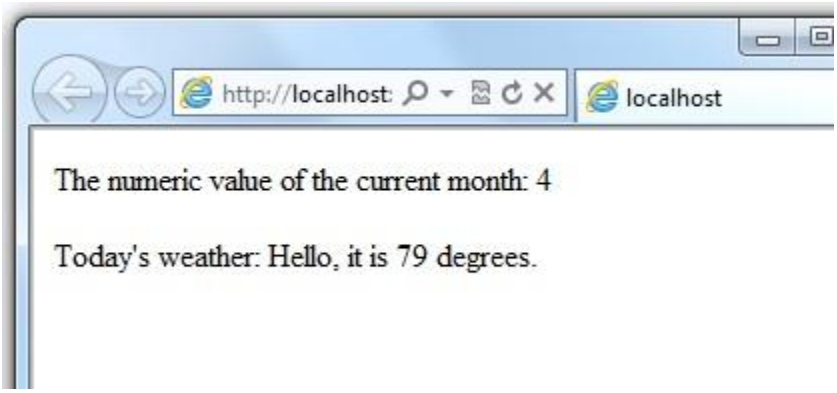
You can read more about HTML encoding in Working with Forms.

2. You enclose code blocks in braces

A *code block* includes one or more code statements and is enclosed in braces.

```
<!-- Single statement block.  -->

@{ var theMonth = DateTime.Now.Month; }

<p>The numeric value of the current month: @theMonth</p>



<!-- Multi-statement block. -->

@{

    var outsideTemp = 79;

    var weatherMessage = "Hello, it is " + outsideTemp + " degrees.";

}

<p>Today's weather: @weatherMessage</p>
```

The result displayed in a browser:



3. Inside a block, you end each code statement with a semicolon

Inside a code block, each complete code statement must end with a semicolon. Inline expressions don't end with a semicolon.

```
<!-- Single-statement block -->

@{ var theMonth = DateTime.Now.Month; }



<!-- Multi-statement block -->

@{

    var outsideTemp = 79;

    var weatherMessage = "Hello, it is " + outsideTemp + " degrees.";

}



<!-- Inline expression, so no semicolon -->
```

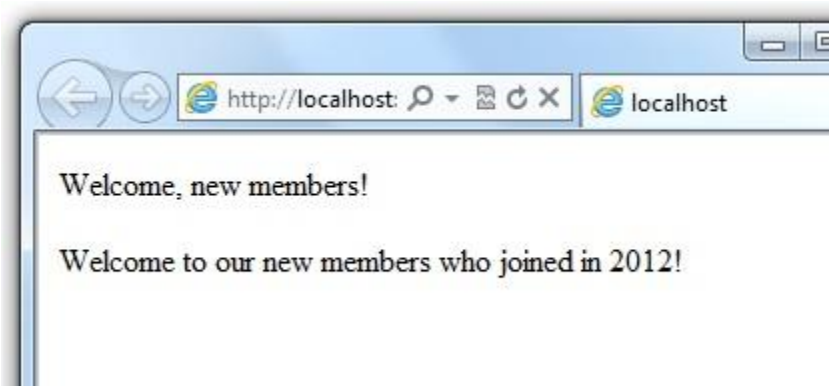
```
<p>Today's weather: @weatherMessage</p>
```

4. You use variables to store values

You can store values in a *variable*, including strings, numbers, and dates, etc. You create a new variable using the `var` keyword. You can insert variable values directly in a page using `@`.

```
<!-- Storing a string -->

@{ var welcomeMessage = "Welcome, new members!"; }

<p>@welcomeMessage</p>



<!-- Storing a date -->

@{ var year = DateTime.Now.Year; }



<!-- Displaying a variable -->

<p>Welcome to our new members who joined in @year!</p>
```

The result displayed in a browser:



5. You enclose literal string values in double quotation marks

A *string* is a sequence of characters that are treated as text. To specify a string, you enclose it in double quotation marks:

```
@{ var myString = "This is a string literal"; }
```

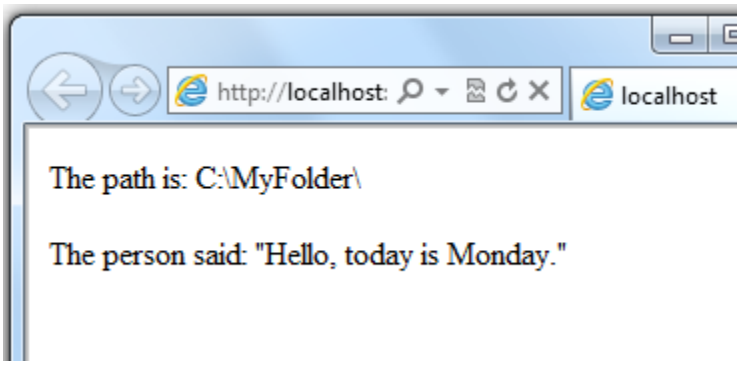
If the string that you want to display contains a backslash character (\) or double quotation marks ("), use a *verbatim string literal* that's prefixed with the @ operator. (In C#, the \ character has special meaning unless you use a verbatim string literal.)

```
<!-- Embedding a backslash in a string -->

@{ var myFilePath = @"C:\MyFolder\"; }

<p>The path is: @myFilePath</p>
```

To embed double quotation marks, use a verbatim string literal and repeat the quotation marks:

```
<!-- Embedding double quotation marks in a string -->

@{ var myQuote = @"The person said: ""Hello, today is Monday."""; }

<p>@myQuote</p>
```

Here's the result of using both of these examples in a page:



Note Notice that the `@` character is used both to mark verbatim string literals in C# and to mark code in ASP.NET pages.

6. Code is case sensitive

In C#, keywords (like `var`, `true`, and `if`) and variable names are case sensitive. The following lines of code create two different variables, `lastName` and `LastName.`

```
@{

    var lastName = "Smith";

    var LastName = "Jones";

}
```

If you declare a variable as `var lastName = "Smith";` and if you try to reference that variable in your page as `@LastName`, an error results because `LastName` won't be recognized.

Note In Visual Basic, keywords and variables are *not* case sensitive.

7. Much of your coding involves objects

An *object* represents a thing that you can program with — a page, a text box, a file, an image, a web request, an email message, a customer record (database row), etc. Objects have properties that describe their characteristics and that you can read or change — a text box object has a `Text` property (among others), a request object has a `Url` property, an email message has a `From` property, and a customer object has a

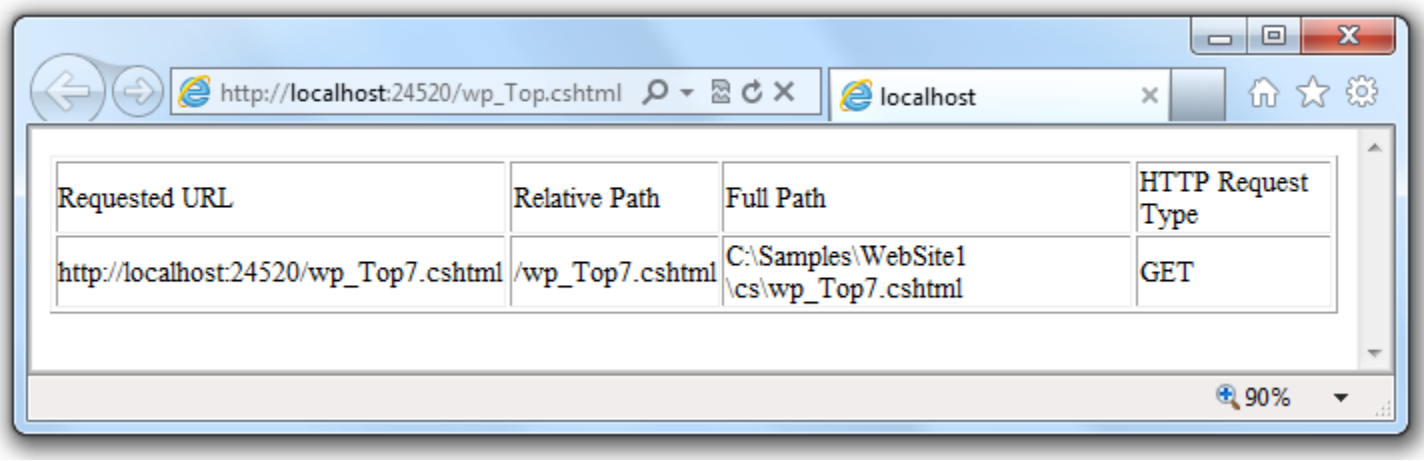
`FirstName` property. Objects also have methods that are the "verbs" they can perform. Examples include a file object's `Save` method, an image object's `Rotate` method, and an email object's `Send` method.

You'll often work with the `Request` object, which gives you information like the values of text boxes (form fields) on the page, what type of browser made the request, the URL of the page, the user identity, etc. The following example shows how to access properties of the `Request` object and how to call the `MapPath` method of the `Request` object, which gives you the absolute path of the page on the server:

```
<table border="1">

<tr>

    <td>Requested URL</td>

    <td>Relative Path</td>

    <td>Full Path</td>

    <td>HTTP Request Type</td>

</tr>

<tr>

    <td>@Request.Url</td>

    <td>@Request.FilePath</td>

    <td>@Request.MapPath(Request.FilePath)</td>

    <td>@Request.RequestType</td>

</tr>
```

```
</table>
```

The result displayed in a browser:



8. You can write code that makes decisions

A key feature of dynamic web pages is that you can determine what to do based on conditions. The most common way to do this is with the `if` statement (and optional `else` statement).

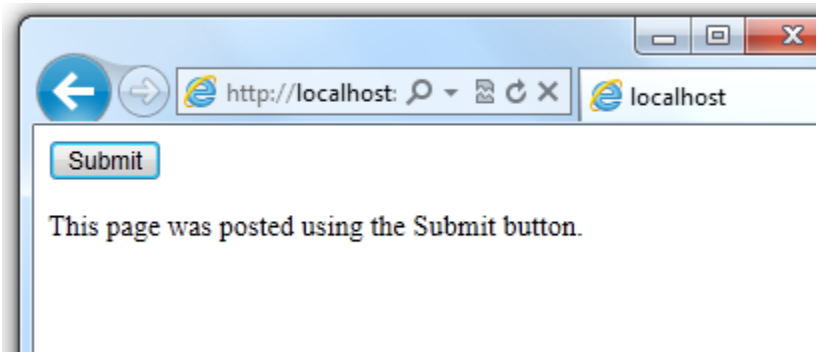
```
@{

   var result = "";

   if(IsPost)

   {

      result = "This page was posted using the Submit button.";

   }

   else

   {
```

```
      result = "This was the first request for this page.";

   }

}



<!DOCTYPE html>

<html>

    <head>

        <title></title>

    </head>

<body>

<form method="POST" action="" >

  <input type="Submit" name="Submit" value="Submit"/>

  <p>@result</p>

</form>

</body>

</html>
```

```
    </body>


</html>
```

The statement `if(IsPost)` is a shorthand way of writing `if(IsPost == true)`. Along with `if` statements, there are a variety of ways to test conditions, repeat blocks of code, and so on, which are described later in this article.

The result displayed in a browser (after clicking **Submit**):



HTTP GET and POST Methods and the IsPost Property

The protocol used for web pages (HTTP) supports a very limited number of methods (verbs) that are used to make requests to the server. The two most common ones are GET, which is used to read a page, and POST, which is used to submit a page. In general, the first time a user requests a page, the page is requested using GET. If the user fills in a form and then clicks a submit button, the browser makes a POST request to the server.

In web programming, it's often useful to know whether a page is being requested as a GET or as a POST so that you know how to process the page. In ASP.NET Web Pages, you can use the `IsPost` property to see whether a request is a GET or a POST. If the request is a POST, the `IsPost` property will return true, and you can do things like read the values of text boxes on a form. Many examples you'll see show you how to process the page differently depending on the value of `IsPost`.

A Simple Code Example

This procedure shows you how to create a page that illustrates basic programming techniques. In the example, you create a page that lets users enter two numbers, then it adds them and displays the result.

1. In your editor, create a new file and name it *AddNumbers.cshtml*.
2. Copy the following code and markup into the page, replacing anything already in the page.

```
3. @{
4.     var total = 0;
5.     var totalMessage = "";
6.     if(IsPost) {
7.
8.         // Retrieve the numbers that the user entered.
9.         var num1 = Request["text1"];
10.        var num2 = Request["text2"];
11.
12.        // Convert the entered strings into integers numbers and add.
13.        total = num1.AsInt() + num2.AsInt();
14.        totalMessage = "Total = " + total;
15.    }
16.}
17.
18.<!DOCTYPE html>
```

```
<html lang="en">
  <head>
    <title>Add Numbers</title>
    <meta charset="utf-8" />
    <style type="text/css">
      body {background-color: beige; font-family: Verdana, Arial;
            margin: 50px; }
      form {padding: 10px; border-style: solid; width: 250px;}
    </style>
  </head>
<body>
  <p>Enter two whole numbers and then click <strong>Add</strong>.</p>
  <form action="" method="post">
    <p><label for="text1">First Number:</label>
      <input type="text" name="text1" />
    </p>
```

```
    <p><label for="text2">Second Number:</label>
      <input type="text" name="text2" />
    </p>
    <p><input type="submit" value="Add" /></p>
  </form>

  <p>@totalMessage</p>

</body>
</html>
```

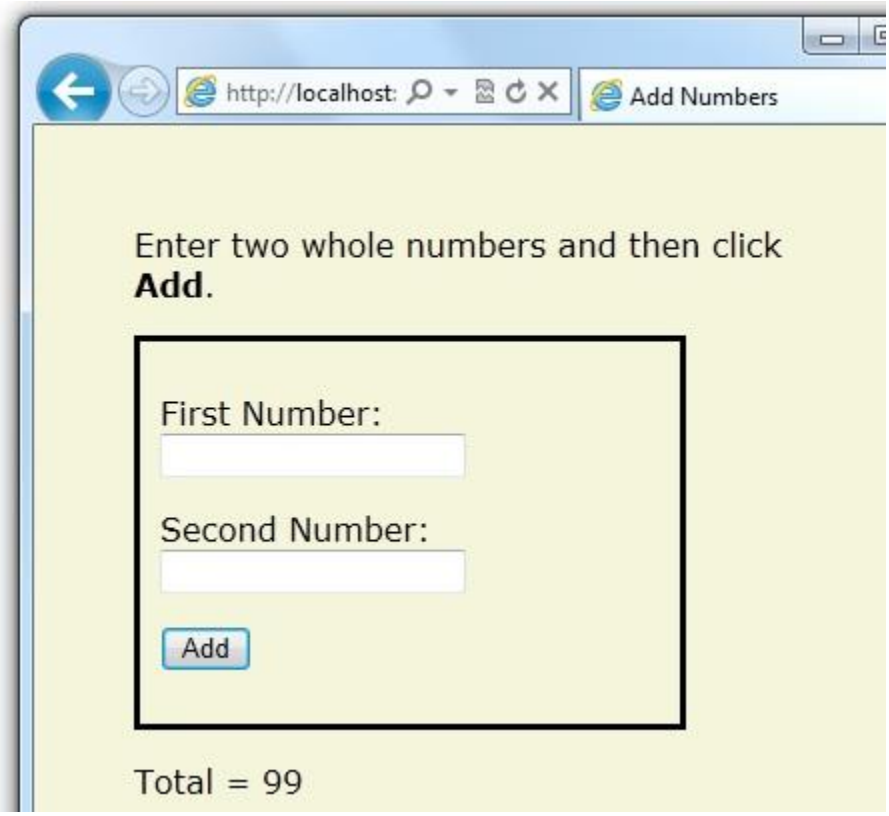
Here are some things for you to note:

- The `@` character starts the first block of code in the page, and it precedes the `totalMessage` variable that's embedded near the bottom of the page.
- The block at the top of the page is enclosed in braces.
- In the block at the top, all lines end with a semicolon.
- The variables `total`, `num1`, `num2`, and `totalMessage` store several numbers and a string.
- The literal string value assigned to the `totalMessage` variable is in double quotation marks.
- Because the code is case-sensitive, when the `totalMessage` variable is used near the bottom of the page, its name must match the variable at the top exactly.

- The expression `num1.AsInt() + num2.AsInt()` shows how to work with objects and methods. The `AsInt` method on each variable converts the string entered by a user to a number (an integer) so that you can perform arithmetic on it.

- The `<form>` tag includes a `method="post"` attribute. This specifies that when the user clicks **Add**, the page will be sent to the server using the HTTP POST method. When the page is submitted, the `if(IsPost)` test evaluates to true and the conditional code runs, displaying the result of adding the numbers.

44. Save the page and run it in a browser. (Make sure the page is selected in the **Files** workspace before you run it.) Enter two whole numbers and then click the **Add** button.



Basic Programming Concepts

This article provides you with an overview of ASP.NET web programming. It isn't an exhaustive examination, just a quick tour through the programming concepts you'll use most often. Even so, it covers almost everything you'll need to get started with ASP.NET Web Pages.

But first, a little technical background.

The Razor Syntax, Server Code, and ASP.NET

Razor syntax is a simple programming syntax for embedding server-based code in a web page. In a web page that uses the Razor syntax, there are two kinds of content: client content and server code. Client content is the stuff you're used to in web pages: HTML markup (elements), style information such as CSS, maybe some client script such as JavaScript, and plain text.

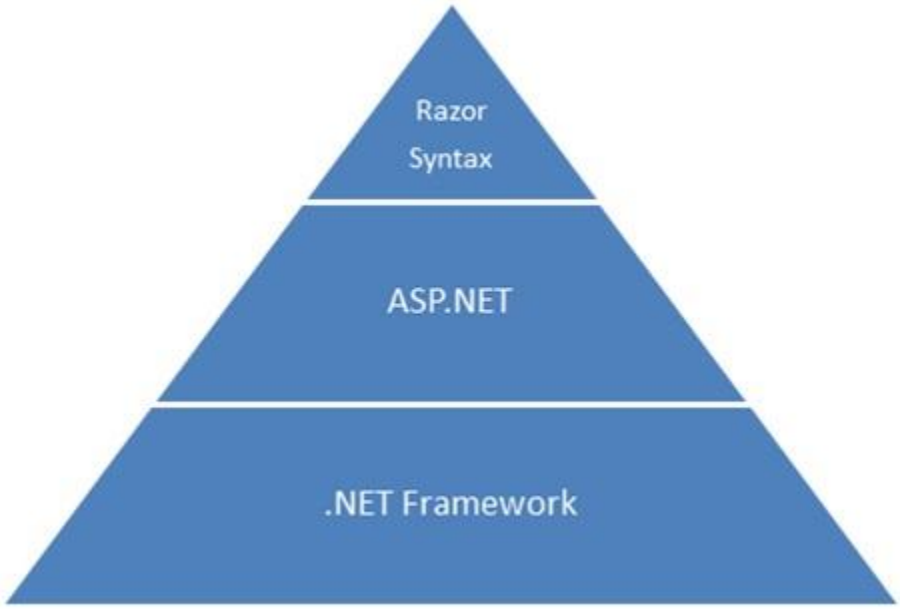
Razor syntax lets you add server code to this client content. If there's server code in the page, the server runs that code first, before it sends the page to the browser. By running on the server, the code can perform tasks that can be a lot more complex to do using client content alone, like accessing server-based databases. Most importantly, server code can dynamically create client content — it can generate HTML markup or other content on the fly and then send it to the browser along with any static HTML that the page might contain. From the browser's perspective, client content that's generated by your server code is no different than any other client content. As you've already seen, the server code that's required is quite simple.

ASP.NET web pages that include the Razor syntax have a special file extension (*.cshtml* or *.vbhtml*). The server recognizes these extensions, runs the code that's marked with Razor syntax, and then sends the page to the browser.

Where does ASP.NET fit in?

Razor syntax is based on a technology from Microsoft called ASP.NET, which in turn is based on the Microsoft .NET Framework. The.NET Framework is a big, comprehensive programming framework from Microsoft for developing virtually any type of computer application. ASP.NET is the part of the .NET Framework that's specifically designed for creating web applications. Developers have used ASP.NET to create many of the largest and highest-traffic websites in the world. (Any time you see the file-name extension *.aspx* as part of the URL in a site, you'll know that the site was written using ASP.NET.)

The Razor syntax gives you all the power of ASP.NET, but using a simplified syntax that's easier to learn if you're a beginner and that makes you more productive if you're an expert. Even though this syntax is simple to use, its family relationship to ASP.NET and the .NET Framework means that as your websites become more sophisticated, you have the power of the larger frameworks available to you.



Classes and Instances

ASP.NET server code uses objects, which are in turn built on the idea of classes. The class is the definition or template for an object. For example, an application might contain a `Customer` class that defines the properties and methods that any customer object needs.

When the application needs to work with actual customer information, it creates an instance of (or *instantiates*) a customer object. Each individual customer is a separate instance of the `Customer` class. Every instance supports the same properties and methods, but the property values for each instance are typically different, because each customer object is unique. In one customer object, the `LastName` property might be "Smith"; in another customer object, the `LastName` property might be "Jones."

Similarly, any individual web page in your site is a `Page` object that's an instance of the `Page` class. A button on the page is a `Button` object that is an instance of the `Button` class, and so on. Each instance has its own characteristics, but they all are based on what's specified in the object's class definition.

Basic Syntax

Earlier you saw a basic example of how to create an ASP.NET Web Pages page, and how you can add server code to HTML markup. Here you'll learn the basics of writing ASP.NET server code using the Razor syntax — that is, the programming language rules.

If you're experienced with programming (especially if you've used C, C++, C#, Visual Basic, or JavaScript), much of what you read here will be familiar. You'll probably need to familiarize yourself only with how server code is added to markup in *.cshtml* files.

Combining Text, Markup, and Code in Code Blocks

In server code blocks, you often want to output text or markup (or both) to the page. If a server code block contains text that's not code and that instead should be rendered as is, ASP.NET needs to be able to distinguish that text from code. There are several ways to do this.

- Enclose the text in an HTML element like `<p></p>` or `<em></em>`:

```
@if(IsPost) {
    // This line has all content between matched <p> tags.
    <p>Hello, the time is @DateTime.Now and this page is a postback!</p>
} else {
    // All content between matched tags, followed by server code.
    <p>Hello <em>stranger</em>, today is: <br /> </p>  @DateTime.Now
}
```

  The HTML element can include text, additional HTML elements, and server-code expressions. When ASP.NET sees the opening HTML tag (for example, `<p>`), it renders everything including the element and its content as is to the browser, resolving server-code expressions as it goes.

- Use the `@:` operator or the `<text>` element. The `@:` outputs a single line of content containing plain text or unmatched HTML tags; the `<text>` element encloses multiple lines to output. These options are useful when you don't want to render an HTML element as part of the output.

```
@if(IsPost) {
    // Plain text followed by an unmatched HTML tag and server code.
```

```
    @: The time is: <br /> @DateTime.Now

    <br/>

    // Server code and then plain text, matched tags, and more text.

    @DateTime.Now @:is the <em>current</em> time.
```

```
}
```

If you want to output multiple lines of text or unmatched HTML tags, you can precede each line with `@:`, or you can enclose the line in a `<text>` element. Like the `@:` operator, `<text>` tags are used by ASP.NET to identify text content and are never rendered in the page output.

```
@if(IsPost) {

    // Repeat the previous example, but use <text> tags.

    <text>

    The time is: <br /> @DateTime.Now

    <br/>

    @DateTime.Now is the <em>current</em> time.

    </text>

}
```

```
@{

    var minTemp = 75;

    <text>It is the month of @DateTime.Now.ToString("MMMM"), and

    it's a <em>great</em> day! <br /><p>You can go swimming if it's at

    least @minTemp degrees. </p></text>

}
```

The first example repeats the previous example but uses a single pair of `<text>` tags to enclose the text to render. In the second example, the `<text>` and `</text>` tags enclose three lines, all of which have some uncontained text and unmatched HTML tags (`<br />`), along with server code and matched HTML tags. Again, you could also precede each line individually with the `@:` operator; either way works.

Note When you output text as shown in this section — using an HTML element, the `@:` operator, or the `<text>` element — ASP.NET doesn't HTML-encode the output. (As noted earlier, ASP.NET does encode the output of server code expressions and server code blocks that are preceded by `@`, except in the special cases noted in this section.)

Whitespace

Extra spaces in a statement (and outside of a string literal) don't affect the statement:

```
@{ var lastName =    "Smith"; }
```

A line break in a statement has no effect on the statement, and you can wrap statements for readability. The following statements are the same:

```
@{ var theName =

"Smith"; }
```

```
@{

    var

    personName

    =

    "Smith"

    ;

}
```

However, you can't wrap a line in the middle of a string literal. The following example doesn't work:

```
@{ var test = "This is a long

    string"; }  // Does not work!
```

To combine a long string that wraps to multiple lines like the above code, there are two options. You can use the concatenation operator (+), which you'll see later in this article. You can also use the @ character to create a verbatim string literal, as you saw earlier in this article. You can break verbatim string literals across lines:

```
@{ var longString = @"This is a

    long

    string";
```

```
}
```

Code (and Markup) Comments

Comments let you leave notes for yourself or others. They also allow you to disable (*comment out*) a section of code or markup that you don't want to run but want to keep in your page for the time being.

There's different commenting syntax for Razor code and for HTML markup. As with all Razor code, Razor comments are processed (and then removed) on the server before the page is sent to the browser. Therefore, the Razor commenting syntax lets you put comments into the code (or even into the markup) that you can see when you edit the file, but that users don't see, even in the page source.

For ASP.NET Razor comments, you start the comment with `@*` and end it with `*@`. The comment can be on one line or multiple lines:

```
@*  A one-line code comment. *@



@*

    This is a multiline code comment.

    It can continue for any number of lines.

*@
```

Here is a comment within a code block:

```
@{

    @* This is a comment. *@

    var theVar = 17;
```

```
}
```

Here is the same block of code, with the line of code commented out so that it won't run:

```
@{

    @* This is a comment. *@

    @* var theVar = 17;  *@

}
```

Inside a code block, as an alternative to using Razor comment syntax, you can use the commenting syntax of the programming language you're using, such as C#:

```
@{

    // This is a comment.

    var myVar = 17;

    /* This is a multi-line comment

    that uses C# commenting syntax. */

}
```

In C#, single-line comments are preceded by the `//` characters, and multi-line comments begin with `/*` and end with `*/`. (As with Razor comments, C# comments are not rendered to the browser.)

For markup, as you probably know, you can create an HTML comment:

```
<!-- This is a comment.  -->
```

HTML comments start with `<!--` characters and end with `-->`. You can use HTML comments to surround not only text, but also any HTML markup that you may want to keep in the page but don't want to render. This HTML comment will hide the entire content of the tags and the text they contain:

```
<!-- <p>This is my paragraph.</p>  -->
```

Unlike Razor comments, HTML comments *are* rendered to the page and the user can see them by viewing the page source.

Variables

A variable is a named object that you use to store data. You can name variables anything, but the name must begin with an alphabetic character and it cannot contain whitespace or reserved characters.

Variables and Data Types

A variable can have a specific data type, which indicates what kind of data is stored in the variable. You can have string variables that store string values (like "Hello world"), integer variables that store whole-number values (like 3 or 79), and date variables that store date values in a variety of formats (like 4/12/2012 or March 2009). And there are many other data types you can use.

However, you generally don't have to specify a type for a variable. Most of the time, ASP.NET can figure out the type based on how the data in the variable is being used. (Occasionally you must specify a type; you'll see examples where this is true.)

You declare a variable using the `var` keyword (if you don't want to specify a type) or by using the name of the type:

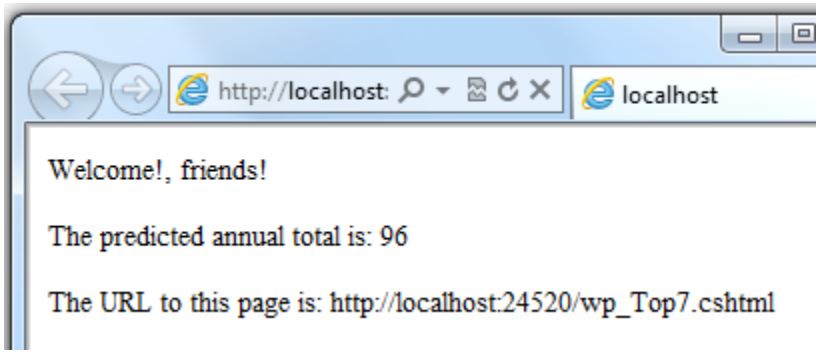
```
@{

    // Assigning a string to a variable.

    var greeting = "Welcome!";



    // Assigning a number to a variable.
```

```
    var theCount = 3;



    // Assigning an expression to a variable.

    var monthlyTotal = theCount + 5;



    // Assigning a date value to a variable.

    var today = DateTime.Today;



    // Assigning the current page's URL to a variable.

    var myPath = this.Request.Url;



    // Declaring variables using explicit data types.

    string name = "Joe";

    int count = 5;

    DateTime tomorrow = DateTime.Now.AddDays(1);

}
```

The following example shows some typical uses of variables in a web page:

```
@{

    // Embedding the value of a variable into HTML markup.

    <p>@greeting, friends!</p>



    // Using variables as part of an inline expression.

    <p>The predicted annual total is: @( monthlyTotal * 12)</p>



    // Displaying the page URL with a variable.

    <p>The URL to this page is: @myPath</p>

}
```

If you combine the previous examples in a page, you see this displayed in a browser:



Converting and Testing Data Types

Although ASP.NET can usually determine a data type automatically, sometimes it can't. Therefore, you might need to help ASP.NET out by performing an explicit conversion. Even if you don't have to convert types, sometimes it's helpful to test to see what type of data you might be working with.

The most common case is that you have to convert a string to another type, such as to an integer or date. The following example shows a typical case where you must convert a string to a number.

```
@{

    var total = 0;



    if(IsPost) {

        // Retrieve the numbers that the user entered.

        var num1 = Request["text1"];

        var num2 = Request["text2"];

        // Convert the entered strings into integers numbers and add.

        total = num1.AsInt() + num2.AsInt();

    }

}
```

As a rule, user input comes to you as strings. Even if you've prompted users to enter a number, and even if they've entered a digit, when user input is submitted and you read it in code, the data is in string format.

Therefore, you must convert the string to a number. In the example, if you try to perform arithmetic on the values without converting them, the following error results, because ASP.NET cannot add two strings:

Cannot implicitly convert type 'string' to 'int'.

To convert the values to integers, you call the `AsInt` method. If the conversion is successful, you can then add the numbers.

The following table lists some common conversion and test methods for variables.

Method	Description	Example

`AsInt()`, `IsInt()`	Converts a string that represents a whole number (like "593") to an integer.	`var myIntNumber = 0;`
		`var myStringNum = "539";`
		`if(myStringNum.IsInt()==true){`
		`    myIntNumber = myStringNum.AsInt();`
		`}`

`AsBool()`, `IsBool()`	Converts a string like "true" or "false" to a Boolean type.	`var myStringBool = "True";` `var myVar = myStringBool.AsBool();`
`AsFloat()`, `IsFloat()`	Converts a string that has a decimal value like "1.3" or "7.439" to a floating-point	`var myStringFloat = "41.432895";` `var myFloatNum = myStringFloat.AsFloat();`

	number.	

`AsDecimal()`, `IsDecimal()`	Converts a string that has a decimal value like "1.3" or "7.439" to a decimal number. (In ASP.NET, a decimal number is more precise than a floating-point number.)	`var myStringDec = "10317.425";`
		
		`var myDecNum = myStringDec.AsDecimal();`

`AsDateTime()`, `IsDateTime()`	Converts a string that represents a date and time value to the ASP.NET `DateTime` type.	`var myDateString = "12/27/2012";`
		
		`var newDate = myDateString.AsDateTime();`

`ToString()`	Converts any other data type to a string.	`int num1 = 17;`
		
		`int num2 = 76;`
		
		`// myString is set to 1776`
		
		`string myString = num1.ToString() +`

		num2.ToString();

Operators

An operator is a keyword or character that tells ASP.NET what kind of command to perform in an expression. The C# language (and the Razor syntax that's based on it) supports many operators, but you only need to recognize a few to get started. The following table summarizes the most common operators.

Operator	Description	Examples

`+`
`-`
`*`
`/`
	Math operators used in numerical expressions.	`@(5 + 13)`
		
		`@{ var netWorth = 150000; }`
		
		`@{ var newTotal = netWorth * 2; }`
		
		`@(newTotal / 2)`

`=`	Assignment. Assigns the value on the right side of a statement to the object on the left side.	`var age = 17;`

`==`	Equality. Returns `true` if the values are equal. (Notice the distinction between the `=` operator and the `==` operator.)	`var myNum = 15;`
		
		`if (myNum == 15) {`
		
		`    // Do something.`
		
		`}`

`!=`	Inequality. Returns `true` if the values are not equal.	`var theNum = 13;`
		
		`if (theNum != 15) {`
		
		`    // Do something.`
		
		`}`

`<`
`>`
`<=`
`>=`
	Less-than,
	greater-than,
	less-than-or-equal, and
	greater-than-or-equal.
		`if (2 < 3) {`
		
		`    // Do something.`
		
		`}`
		
		`var currentCount = 12;`
		
		`if(currentCount >= 12) {`
		
		`    // Do something.`
		
		`}`

`+`	Concatenation, which is used to join strings. ASP.NET knows the difference between this operator and the addition operator based on the data type of the expression.	`// The displayed result is "abcdef".`
		
		`@("abc" + "def")`

`+=`
 	
`-=`
	The increment and decrement operators, which add and subtract 1 (respectively) from a variable.	`int theCount = 0;`
		
		`theCount += 1; // Adds 1 to count`

`.`	Dot. Used to distinguish objects and their properties and methods.	`var myUrl = Request.Url;`
		
		`var count = Request["Count"].AsInt();`

`()`	Parentheses. Used to group expressions and to pass parameters to methods.	`@(3 + 7)`
		
		`@Request.MapPath(Request.FilePath);`

`[]`	Brackets. Used for accessing values in arrays or collections.	`var income = Request["AnnualIncome"];`

`!`	Not. Reverses a `true` value to `false` and vice versa. Typically used as a shorthand way to test for `false` (that is, for not `true`).	`bool taskCompleted = false;`
		
		`// Processing.`
		
		`if(!taskCompleted) {`
		
		`    // Continue processing`

		`}`

`&&`
 	
`||`
	Logical AND and OR, which are used to link conditions together.	`bool myTaskCompleted = false;`
		
		`int totalCount = 0;`
		
		`// Processing.`
		
		`if(!myTaskCompleted && totalCount < 12) {`
		
		`    // Continue processing.`
		
		`}`

Working with File and Folder Paths in Code

You'll often work with file and folder paths in your code. Here is an example of physical folder structure for a website as it might appear on your development computer:

```
C:\WebSites\MyWebSite
    default.cshtml
    datafile.txt
    \images
        Logo.jpg
    \styles
        Styles.css
```

Here are some essential details about URLs and paths:

- A URL begins with either a domain name (*http://www.example.com*) or a server name (*http://localhost*, *http://mycomputer*).

- A URL corresponds to a physical path on a host computer. For example, *http://myserver* might correspond to the folder *C:\websites\mywebsite* on the server.
- A virtual path is shorthand to represent paths in code without having to specify the full path. It includes the portion of a URL that follows the domain or server name. When you use virtual paths, you can move your code to a different domain or server without having to update the paths.

Here's an example to help you understand the differences:

Complete URL	*http://mycompanyserver/humanresources/CompanyPolicy.htm*
Server name	*mycompanyserver*
Virtual path	*/humanresources/CompanyPolicy.htm*
Physical path	*C:\mywebsites\humanresources\CompanyPolicy.htm*

The virtual root is /, just like the root of your C: drive is \. (Virtual folder paths always use forward slashes.) The virtual path of a folder doesn't have to have the same name as the physical folder; it can be an alias. (On production servers, the virtual path rarely matches an exact physical path.)

When you work with files and folders in code, sometimes you need to reference the physical path and sometimes a virtual path, depending on what objects you're working with. ASP.NET gives you these tools for working with file and folder paths in code: the `Server.MapPath` method, and the `~` operator and `Href` method.

Converting virtual to physical paths: the Server.MapPath method

The `Server.MapPath` method converts a virtual path (like */default.cshtml*) to an absolute physical path (like *C:\WebSites\MyWebSiteFolder\default.cshtml*). You use this method any time you need a complete physical path. A typical example is when you're reading or writing a text file or image file on the web server.

You typically don't know the absolute physical path of your site on a hosting site's server, so this method can convert the path you do know — the virtual path — to the corresponding path on the server for you. You pass the virtual path to a file or folder to the method, and it returns the physical path:

```
@{
```

```
    var dataFilePath = "~/dataFile.txt";

}

<!-- Displays a physical path C:\Websites\MyWebSite\datafile.txt  -->

<p>@Server.MapPath(dataFilePath)</p>
```

Referencing the virtual root: the ~ operator and Href method

In a *.cshtml* or *.vbhtml* file, you can reference the virtual root path using the ~ operator. This is very handy because you can move pages around in a site, and any links they contain to other pages won't be broken. It's also handy in case you ever move your website to a different location. Here are some examples:

```
@{

    var myImagesFolder = "~/images";

    var myStyleSheet = "~/styles/StyleSheet.css";

}
```

If the website is *http://myserver/myapp*, here's how ASP.NET will treat these paths when the page runs:

- `myImagesFolder`: *http://myserver/myapp/images*
- `myStyleSheet` : *http://myserver/myapp/styles/Stylesheet.css*

(You won't actually see these paths as the values of the variable, but ASP.NET will treat the paths as if that's what they were.)

In ASP.NET Web Pages 2, you can use the ~ operator both in server code (as above) and in markup, like this:

```
<!-- Examples of using the ~ operator in markup in ASP.NET Web Pages 2 -->
```

```
<!-- (Using the ~ operator like this in markup is not supported in ASP.NET

     Web Pages 1.0) -->



<a href="~/Default">Home</a>

<img src="~/images/MyImages.png" />
```

In markup, you use the `~` operator to create paths to resources like image files, other web pages, and CSS files. When the page runs, ASP.NET looks through the page (both code and markup) and resolves all the `~` references to the appropriate path.

In ASP.NET Web Pages 1, you can use the `~` operator in server code blocks, like the first example above. But in order to use it in markup, you have to put the `~` operator inside a call to the `Href` method. (ASP.NET does not parse through the markup looking for the `~` operator.)

For example, you can use the `Href` method in HTML markup for attributes of `<img>` elements, `<link>` elements, and `<a>` elements. Notice that the `Href` method is preceded by `@` to mark it as server code. Also notice that the `Href` method is inside the double quotation marks that enclose attribute values.

```
<!-- Examples of using the Href method in ASP.NET Web Pages 1.0 to include

     the ~ operator in markup. -->



<a href="@Href("~/Default")">Home</a>



<!-- This code creates the path "../images/Logo.jpg" in the src attribute. -->
```

```
<img src="@Href("~/images")/Logo.jpg" />



<!-- This creates a link to the CSS file using ther server variable. -->

<link rel="stylesheet" type="text/css" href="@Href(myStyleSheet)" />
```

Conditional Logic and Loops

ASP.NET server code lets you perform tasks based on conditions and write code that repeats statements a specific number of times (that is, code that runs a loop).

Testing Conditions

To test a simple condition you use the `if` statement, which returns true or false based on a test you specify:

```
@{

  var showToday = true;

  if(showToday)

  {

    @DateTime.Today;

  }

}
```

The `if` keyword starts a block. The actual test (condition) is in parentheses and returns true or false. The statements that run if the test is true are enclosed in braces. An `if` statement can include an `else` block that specifies statements to run if the condition is false:

```
@{

  var showToday = false;

  if(showToday)

  {

    @DateTime.Today;

  }

  else

  {

    <text>Sorry!</text>

  }

}
```

You can add multiple conditions using an `else if` block:

```
@{

    var theBalance = 4.99;

    if(theBalance == 0)
```

```
    {

        <p>You have a zero balance.</p>

    }

    else if (theBalance  > 0 && theBalance <= 5)

    {

        <p>Your balance of $@theBalance is very low.</p>

    }

    else

    {

        <p>Your balance is: $@theBalance</p>

    }

}
```

In this example, if the first condition in the if block is not true, the `else if` condition is checked. If that condition is met, the statements in the `else if` block are executed. If none of the conditions are met, the statements in the `else` block are executed. You can add any number of else if blocks, and then close with an `else` block as the "everything else" condition.

To test a large number of conditions, use a `switch` block:

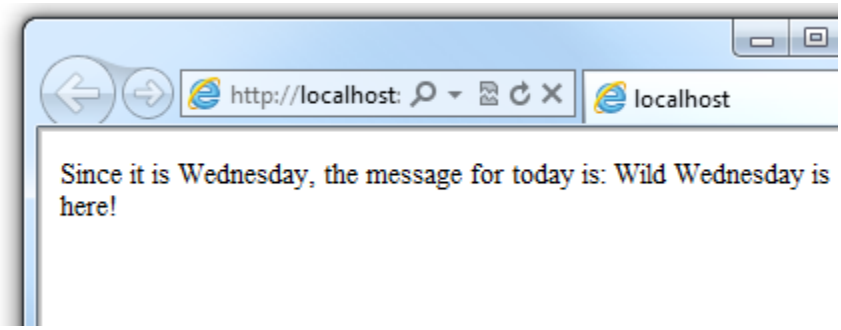
```
@{
```

```
var weekday = "Wednesday";

var greeting = "";



switch(weekday)

{

    case "Monday":

        greeting = "Ok, it's a marvelous Monday";

        break;

    case "Tuesday":

        greeting = "It's a tremendous Tuesday";

        break;

    case "Wednesday":

        greeting = "Wild Wednesday is here!";

        break;

    default:

        greeting = "It's some other day, oh well.";
```

```
            break;

    }



    <p>Since it is @weekday, the message for today is: @greeting</p>

}
```

The value to test is in parentheses (in the example, the `weekday` variable). Each individual test uses a `case` statement that ends with a colon (:). If the value of a `case` statement matches the test value, the code in that case block is executed. You close each case statement with a `break` statement. (If you forget to include break in each `case` block, the code from the next `case` statement will run also.) A `switch` block often has a `default` statement as the last case for an "everything else" option that runs if none of the other cases are true.

The result of the last two conditional blocks displayed in a browser:



Looping Code

You often need to run the same statements repeatedly. You do this by looping. For example, you often run the same statements for each item in a collection of data. If you know exactly how many times you want to loop, you can use a `for` loop. This kind of loop is especially useful for counting up or counting down:

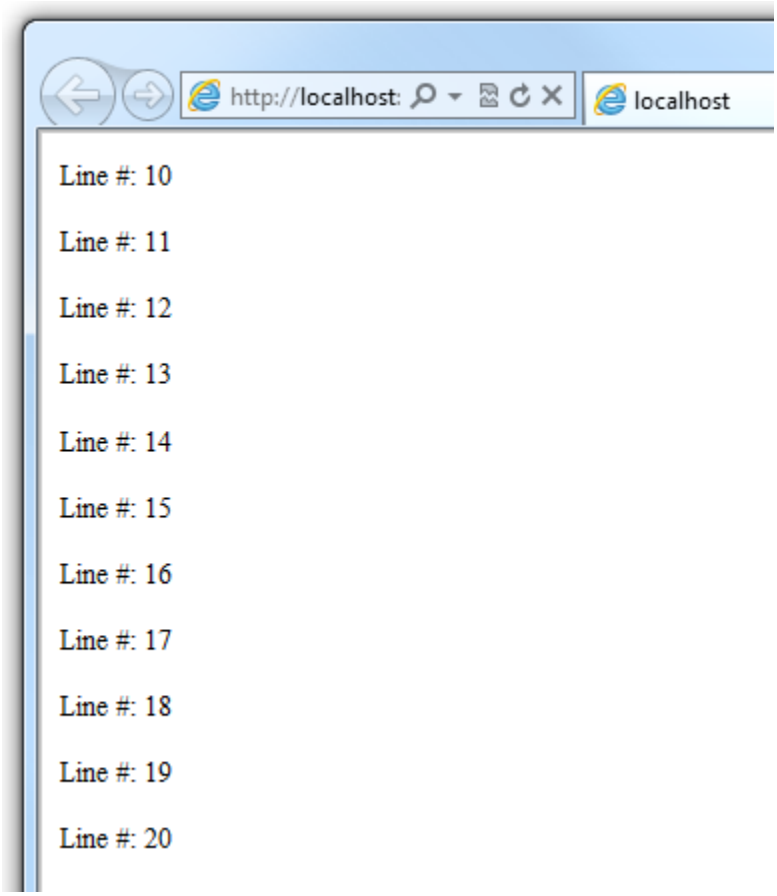
```
@for(var i = 10; i < 21; i++)


{
```

```
    <p>Line #: @i</p>


}
```

The loop begins with the `for` keyword, followed by three statements in parentheses, each terminated with a semicolon.

- Inside the parentheses, the first statement (`var i=10;`) creates a counter and initializes it to 10. You don't have to name the counter `i` — you can use any variable. When the `for` loop runs, the counter is automatically incremented.
- The second statement (`i < 21;`) sets the condition for how far you want to count. In this case, you want it to go to a maximum of 20 (that is, keep going while the counter is less than 21).
- The third statement (`i++`) uses an increment operator, which simply specifies that the counter should have 1 added to it each time the loop runs.

Inside the braces is the code that will run for each iteration of the loop. The markup creates a new paragraph (`<p>` element) each time and adds a line to the output, displaying the value of `i` (the counter). When you run this page, the example creates 11 lines displaying the output, with the text in each line indicating the item number.



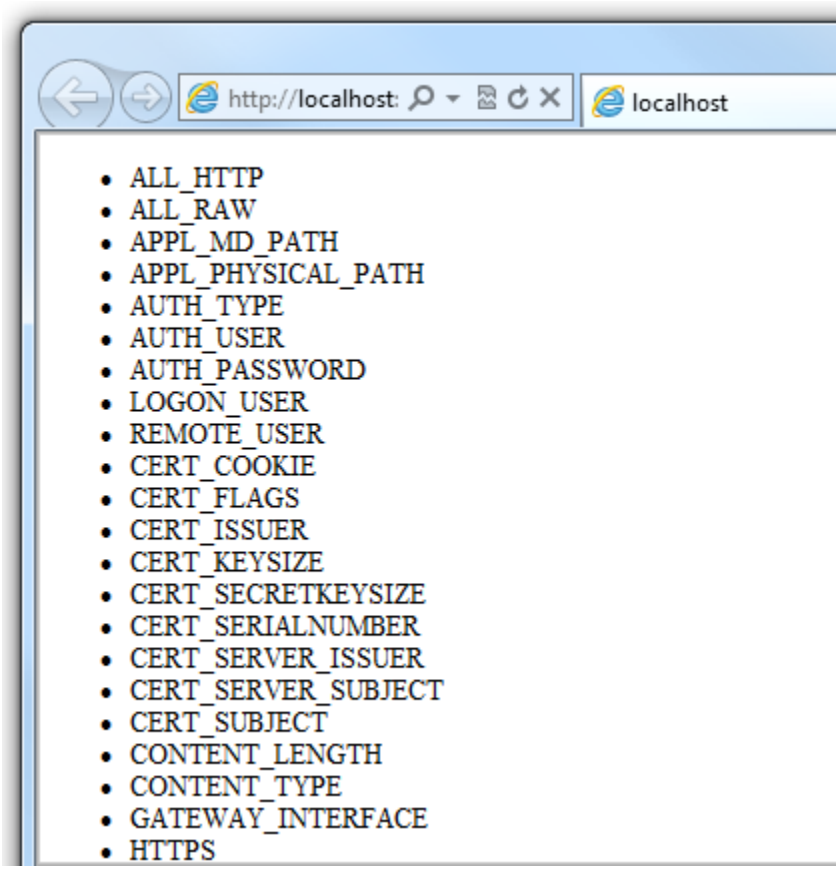
If you're working with a collection or array, you often use a `foreach` loop. A collection is a group of similar objects, and the `foreach` loop lets you carry out a task on each item in the collection. This type of loop is convenient for collections, because unlike a `for` loop, you don't have to increment the counter or set a limit. Instead, the `foreach` loop code simply proceeds through the collection until it's finished.

For example, the following code returns the items in the `Request.ServerVariables` collection, which is an object that contains information about your web server. It uses a `foreach` loop to display the name of each item by creating a new `<li>` element in an HTML bulleted list.

```
<ul>

@foreach (var myItem in Request.ServerVariables)

{
```

```
    <li>@myItem</li>

}

</ul>
```

The `foreach` keyword is followed by parentheses where you declare a variable that represents a single item in the collection (in the example, `var item`), followed by the `in` keyword, followed by the collection you want to loop through. In the body of the `foreach` loop, you can access the current item using the variable that you declared earlier.



To create a more general-purpose loop, use the `while` statement:

```
@{

    var countNum = 0;
```

```
    while (countNum < 50)

    {

        countNum += 1;

        <p>Line #@countNum: </p>

    }

}
```

A `while` loop begins with the `while` keyword, followed by parentheses where you specify how long the loop continues (here, for as long as `countNum` is less than 50), then the block to repeat. Loops typically increment (add to) or decrement (subtract from) a variable or object used for counting. In the example, the `+=` operator adds 1 to `countNum` each time the loop runs. (To decrement a variable in a loop that counts down, you would use the decrement operator `-=`).

Objects and Collections

Nearly everything in an ASP.NET website is an object, including the web page itself. This section discusses some important objects you'll work with frequently in your code.

Page Objects

The most basic object in ASP.NET is the page. You can access properties of the page object directly without any qualifying object. The following code gets the page's file path, using the `Request` object of the page:

```
@{

    var path = Request.FilePath;

}
```

To make it clear that you're referencing properties and methods on the current page object, you can optionally use the keyword `this` to represent the page object in your code. Here is the previous code example, with `this` added to represent the page:

```
@{

    var path = this.Request.FilePath;

}
```

You can use properties of the `Page` object to get a lot of information, such as:

- `Request`. As you've already seen, this is a collection of information about the current request, including what type of browser made the request, the URL of the page, the user identity, etc.
- `Response`. This is a collection of information about the response (page) that will be sent to the browser when the server code has finished running. For example, you can use this property to write information into the response.

```
@{
    // Access the page's Request object to retrieve the Url.
    var pageUrl = this.Request.Url;
}
```

```
<a href="@pageUrl">My page</a>
```

Collection Objects (Arrays and Dictionaries)

A *collection* is a group of objects of the same type, such as a collection of `Customer` objects from a database. ASP.NET contains many built-in collections, like the `Request.Files` collection.

You'll often work with data in collections. Two common collection types are the *array* and the *dictionary*. An array is useful when you want to store a collection of similar items but don't want to create a separate variable to hold each item:

```
@* Array block 1: Declaring a new array using braces. *@

@{

    <h3>Team Members</h3>

    string[] teamMembers = {"Matt", "Joanne", "Robert", "Nancy"};

    foreach (var person in teamMembers)

    {

        <p>@person</p>

    }

}
```

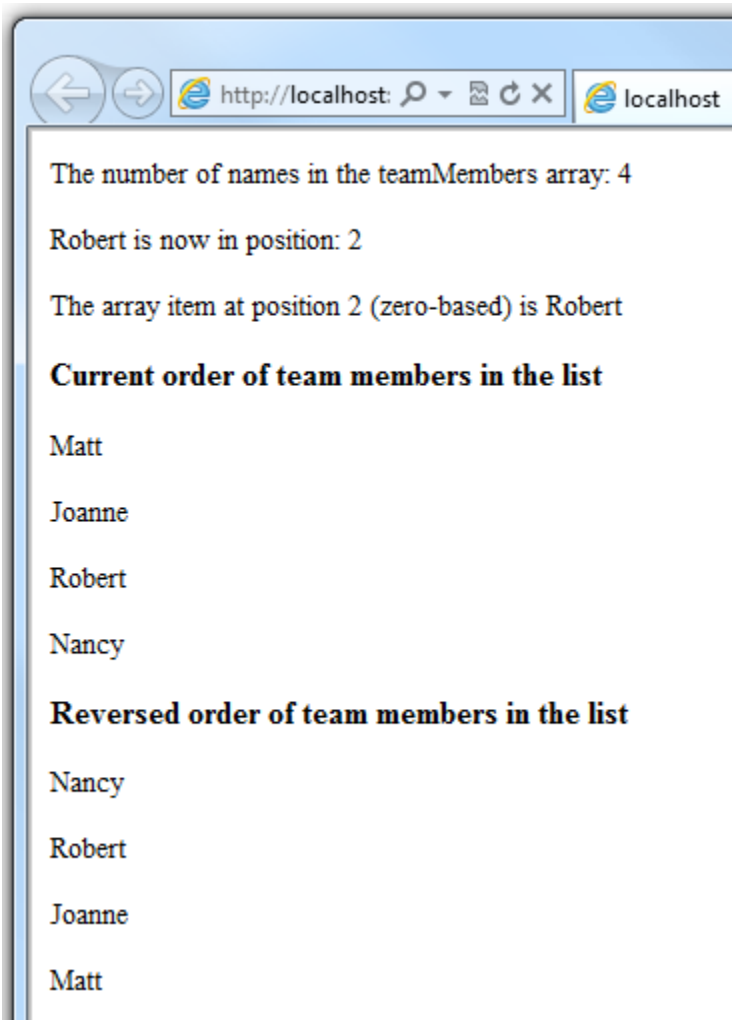
With arrays, you declare a specific data type, such as `string`, `int`, or `DateTime`. To indicate that the variable can contain an array, you add brackets to the declaration (such as `string[]` or `int[]`). You can access items in an array using their position (index) or by using the `foreach` statement. Array indexes are zero-based — that is, the first item is at position 0, the second item is at position 1, and so on.

```
@{

    string[] teamMembers = {"Matt", "Joanne", "Robert", "Nancy"};

    <p>The number of names in the teamMembers array: @teamMembers.Length </p>

    <p>Robert is now in position: @Array.IndexOf(teamMembers, "Robert")</p>
```

```
    <p>The array item at position 2 (zero-based) is @teamMembers[2]</p>

    <h3>Current order of team members in the list</h3>

    foreach (var name in teamMembers)

    {

        <p>@name</p>

    }

    <h3>Reversed order of team members in the list</h3>

    Array.Reverse(teamMembers);

    foreach (var reversedItem in teamMembers)

    {

        <p>@reversedItem</p>

    }

}
```

You can determine the number of items in an array by getting its `Length` property. To get the position of a specific item in the array (to search the array), use the `Array.IndexOf` method. You can also do things like reverse the contents of an array (the `Array.Reverse` method) or sort the contents (the `Array.Sort` method).

The output of the string array code displayed in a browser:



A dictionary is a collection of key/value pairs, where you provide the key (or name) to set or retrieve the corresponding value:

```
@{

    var myScores = new Dictionary<string, int>();

    myScores.Add("test1", 71);

    myScores.Add("test2", 82);

    myScores.Add("test3", 100);
```

```
    myScores.Add("test4", 59);

}

<p>My score on test 3 is: @myScores["test3"]%</p>

@(myScores["test4"] = 79)

<p>My corrected score on test 4 is: @myScores["test4"]%</p>
```

To create a dictionary, you use the `new` keyword to indicate that you're creating a new dictionary object. You can assign a dictionary to a variable using the `var` keyword. You indicate the data types of the items in the dictionary using angle brackets (`< >`). At the end of the declaration, you must add a pair of parentheses, because this is actually a method that creates a new dictionary.

To add items to the dictionary, you can call the `Add` method of the dictionary variable (`myScores` in this case), and then specify a key and a value. Alternatively, you can use square brackets to indicate the key and do a simple assignment, as in the following example:

```
myScores["test4"] = 79;
```

To get a value from the dictionary, you specify the key in brackets:

```
var testScoreThree = myScores["test3"];
```

Calling Methods with Parameters

As you read earlier in this article, the objects that you program with can have methods. For example, a `Database` object might have a `Database.Connect` method. Many methods also have one or more parameters. A *parameter* is a value that you pass to a method to enable the method to complete its task. For example, look at a declaration for the `Request.MapPath` method, which takes three parameters:

```
public string MapPath(string virtualPath, string baseVirtualDir,
```

```
    bool allowCrossAppMapping);
```

(The line has been wrapped to make it more readable. Remember that you can put line breaks almost any place except inside strings that are enclosed in quotation marks.)

This method returns the physical path on the server that corresponds to a specified virtual path. The three parameters for the method are `virtualPath`, `baseVirtualDir`, and `allowCrossAppMapping`. (Notice that in the declaration, the parameters are listed with the data types of the data that they'll accept.) When you call this method, you must supply values for all three parameters.

The Razor syntax gives you two options for passing parameters to a method: *positional parameters* and *named parameters*. To call a method using positional parameters, you pass the parameters in a strict order that's specified in the method declaration. (You would typically know this order by reading documentation for the method.) You must follow the order, and you can't skip any of the parameters — if necessary, you pass an empty string (`""`) or `null` for a positional parameter that you don't have a value for.

The following example assumes you have a folder named *scripts* on your website. The code calls the `Request.MapPath` method and passes values for the three parameters in the correct order. It then displays the resulting mapped path.

```
@{

    // Pass parameters to a method using positional parameters.

    var myPathPositional = Request.MapPath("/scripts", "/", true);

}

<p>@myPathPositional</p>
```

When a method has many parameters, you can keep your code more readable by using named parameters. To call a method using named parameters, you specify the parameter name followed by a colon (:), and then the value. The advantage of named parameters is that you can pass them in any order you want. (A disadvantage is that the method call is not as compact.)

The following example calls the same method as above, but uses named parameters to supply the values:

```
@{

    // Pass parameters to a method using named parameters.

    var myPathNamed = Request.MapPath(baseVirtualDir: "/",

        allowCrossAppMapping: true, virtualPath: "/scripts");

}

<p>@myPathNamed</p>
```

As you can see, the parameters are passed in a different order. However, if you run the previous example and this example, they'll return the same value.

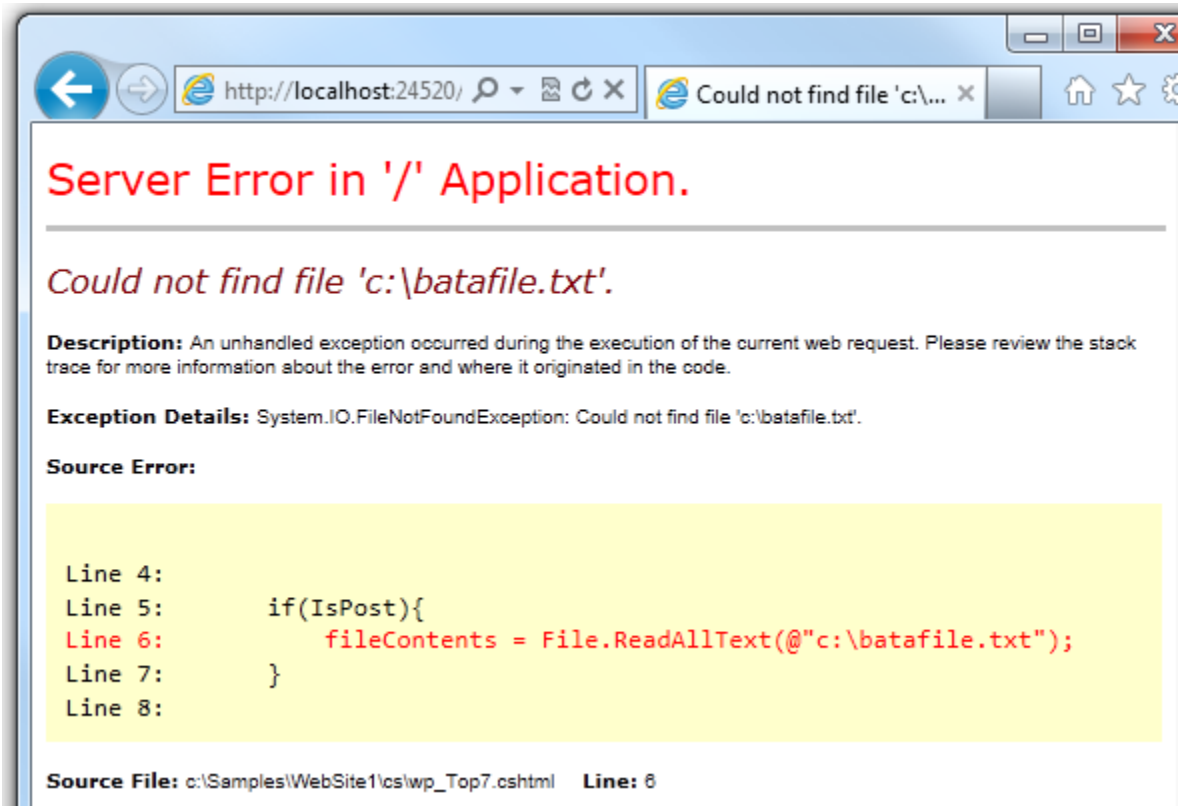
Handling Errors

Try-Catch Statements

You'll often have statements in your code that might fail for reasons outside your control. For example:

- If your code tries to create or access a file, all sorts of errors might occur. The file you want might not exist, it might be locked, the code might not have permissions, and so on.
- Similarly, if your code tries to update records in a database, there can be permissions issues, the connection to the database might be dropped, the data to save might be invalid, and so on.

In programming terms, these situations are called *exceptions*. If your code encounters an exception, it generates (throws) an error message that's, at best, annoying to users:



In situations where your code might encounter exceptions, and in order to avoid error messages of this type, you can use `try/catch` statements. In the `try` statement, you run the code that you're checking. In one or more `catch` statements, you can look for specific errors (specific types of exceptions) that might have occurred. You can include as many `catch` statements as you need to look for errors that you are anticipating.

Note We recommend that you avoid using the `Response.Redirect` method in `try/catch` statements, because it can cause an exception in your page.

The following example shows a page that creates a text file on the first request and then displays a button that lets the user open the file. The example deliberately uses a bad file name so that it will cause an exception. The code includes `catch` statements for two possible exceptions: `FileNotFoundException`, which occurs if the file name is bad, and `DirectoryNotFoundException`, which occurs if ASP.NET can't even find the folder. (You can uncomment a statement in the example in order to see how it runs when everything works properly.)

If your code didn't handle the exception, you would see an error page like the previous screen shot. However, the `try/catch` section helps prevent the user from seeing these types of errors.

```
@{
    var dataFilePath = "~/dataFile.txt";
    var fileContents = "";
```

```
    var physicalPath = Server.MapPath(dataFilePath);
    var userMessage = "Hello world, the time is " + DateTime.Now;
    var userErrMsg = "";
    var errMsg = "";

    if(IsPost)
    {
        // When the user clicks the "Open File" button and posts
        // the page, try to open the created file for reading.
        try {
            // This code fails because of faulty path to the file.
            fileContents = File.ReadAllText(@"c:\batafile.txt");

            // This code works. To eliminate error on page,
            // comment the above line of code and uncomment this one.
            //fileContents = File.ReadAllText(physicalPath);
        }
        catch (FileNotFoundException ex) {
            // You can use the exception object for debugging, logging, etc.
            errMsg = ex.Message;
            // Create a friendly error message for users.
            userErrMsg = "A file could not be opened, please contact "
                + "your system administrator.";
        }
        catch (DirectoryNotFoundException ex) {
            // Similar to previous exception.
            errMsg = ex.Message;
            userErrMsg = "A directory was not found, please contact "
                + "your system administrator.";
        }
    }
    else
    {
        // The first time the page is requested, create the text file.
        File.WriteAllText(physicalPath, userMessage);
    }
}
```

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Try-Catch Statements</title>
    </head>
    <body>
    <form method="POST" action="" >
      <input type="Submit" name="Submit" value="Open File"/>
    </form>

    <p>@fileContents</p>
    <p>@userErrMsg</p>

    </body>
</html>
```