CHAPTER 1

Introducing ASP.NET

When Microsoft created .NET, it wasn't just dreaming about the future—it was also worrying about the headaches and limitations of the current generation of web development technologies. Before you get started with ASP.NET 3.5, it helps to take a step back and consider these problems. You'll then understand the solution that .NET offers.

In this chapter you'll consider the history of web development leading up to ASP.NET, take a whirlwind tour of the most significant features of .NET, and preview the core changes in ASP.NET 3.5. If you're new to ASP.NET, this chapter will quickly get you up to speed. On the other hand, if you're a seasoned .NET developer, you have two choices. Your first option is to read this chapter for a brisk review of where we are today. Alternatively, you can skip to the section "ASP.NET 3.5: The Story Continues" to preview what ASP.NET 3.5 has in store.

The Evolution of Web Development

More than 15 years ago, Tim Berners-Lee performed the first transmission across HTTP (Hypertext Transfer Protocol). Since then, HTTP has become exponentially more popular, expanding beyond a small group of computer-science visionaries to the personal and business sectors. Today, it's almost a household word.

When HTTP was first established, developers faced the challenge of designing applications that could discover and interact with each other. To help meet these challenges, standards such as HTML (Hypertext Markup Language) and XML (Extensible Markup Language) were created. HTML established a simple language that can describe how to display rich documents on virtually any computer platform. XML created a set of rules for building platform-neutral data formats that different applications can use to exchange information. These standards guaranteed that the Web could be used by anyone, located anywhere, using any type of computing system.

At the same time, software vendors faced their own challenges. Not only did they need to develop languages and programming tools that could integrate with the Web, but they also needed to build entire frameworks that would allow developers to architect, develop, and deploy these applications easily. Major software vendors including IBM, Sun Microsystems, and Microsoft rushed to meet this need with a host of products.

ASP.NET 3.5 is the latest chapter in this ongoing arms race. With .NET, Microsoft has created an integrated suite of components that combines the building blocks of the Web—markup languages and HTTP—with proven object-oriented methodology.

The Early Web Development World

The first generation of web applications were difficult to program and difficult to manage, and they faced significant performance and scalability challenges. Overall, early web development technologies fall into two basic categories:

- **Separate, tiny applications that are executed by server-side calls**: Early implementations of CGI (Command Gateway Interface) are a good example. The key problem with this development model is that it consumes large amounts of server resources, because each request requires a separate application instance. As a result, these applications don't scale to large numbers.
- **Scripts that are interpreted by a server-side resource**: Classic ASP (Active Server Pages) and early implementations of ColdFusion fall into this category. To use these platforms, you create files that contain HTML and embedded script code. The script file is examined by a parser at runtime, which alternates between rendering ordinary HTML and executing your embedded code. This process is much less efficient than executing compiled code.

ASP.NET is far more than a simple evolution of either type of application. ASP.NET is *not* a set of clumsy hooks that let you trigger applications or run components on the server. Instead, ASP.NET web applications are full .NET applications that run compiled code and are managed by the .NET runtime. ASP.NET also uses the full capabilities of the .NET Framework—a comprehensive toolkit of classes—just as easily as an ordinary Windows application. In essence, ASP.NET blurs the line between *application* development and *web* development by extending the tools and technologies of desktop developers into the web development world.

What's Wrong with Classic ASP?

If you've programmed only with classic ASP before, you might wonder why Microsoft changed everything with ASP.NET. Learning a whole new framework isn't trivial, and .NET introduces a slew of concepts and a significant learning curve (but it's well worth it).

Overall, classic ASP is a solid tool for developing web applications using Microsoft technologies. However, as with most development models, ASP solves some problems but also raises a few of its own. The following sections outline these problems.

Spaghetti Code

If you've created applications with ASP, you've probably seen lengthy pages that contain server-side script code intermingled with HTML. Consider the following example, which fills an HTML drop-down list with the results of a database query:

```
<%
  Set dbConn = Server.CreateObject("ADODB.Connection")
  Set rs = Server.CreateObject("ADODB.Recordset")

  dbConn.Provider = "sqloledb"
  dbConn.Open "Server=SERVER_NAME; Database=Pubs; Trusted_Connection=yes"
%>
```

```
<select name="cboAuthors">
  <%
    rs.Open "SELECT * FROM Authors", dbConn, 3, 3
    Do While Not rs.EOF
  %>
  <option value="<%=rs("au_id")%>">
    <%=rs("au_lname") & ", " & rs("au_fname")%>
  </option>
  <%
    rs.MoveNext
    Loop
  %>
</select>
```

This example needs an unimpressive 19 lines of code to generate the output for simple HTML list control. But what's worse is the way this style of coding diminishes application performance because it mingles HTML and script. When this page is processed by the ASP ISAPI (Internet Server Application Programming Interface) extension that runs on the web server, the scripting engine needs to switch on and off multiple times just to handle this single request. This increases the amount of time needed to process the whole page and send it to the client.

Furthermore, web pages written in this style can easily grow to unmanageable lengths. If you add your own custom COM components to the puzzle (which are needed to supply functionality ASP can't provide), the management nightmare grows. The bottom line is that no matter what approach you take, ASP code tends to become beastly, long, and incredibly difficult to debug—if you can even get ASP debugging working in your environment at all.

In ASP.NET, these problems don't exist. Web pages are written with traditional object-oriented concepts in mind. Your web pages contain controls that you can program against in a similar way to desktop applications. This means you don't need to combine a jumble of HTML markup and inline code. If you opt to use the code-behind approach when creating ASP.NET pages, the code and presentation are actually placed in two different files, which simplifies code maintenance and allows you to separate the task of web-page design from the heavy-duty work of web coding.

Script Languages

At the time of its creation, ASP seemed like a perfect solution for desktop developers who were moving to the world of the Web. Rather than requiring programmers to learn a completely new language or methodology, ASP allowed developers to use familiar languages such as VBScript on a server-based programming platform. By leveraging the already-popular COM (Component Object Model) programming model as a backbone, these scripting languages also acted as a convenient vehicle for accessing server components and resources. But even though ASP was easy to understand for developers who were already skilled with scripting languages such as VBScript, this familiarity came with a price. Because ASP was based on old technologies that were originally designed for client use, it couldn't perform as well in the new environment of web development.

Performance wasn't the only problem. Every object or variable used in a classic ASP script is created as a *variant* data type. As most Visual Basic programmers know, variant data types are weakly typed. They require larger amounts of memory, their data type is only known (and checked) at runtime, and they result in slower performance than strongly typed variables. Additionally, the Visual Basic compiler and development tools can't identify them at design time. This made it all but

impossible to create a truly integrated IDE (integrated development environment) that could provide ASP programmers with anything like the powerful debugging, IntelliSense, and error checking found in Visual Basic and Visual C++. And without debugging tools, ASP programmers were hard-pressed to troubleshoot the problems in their scripts.

ASP.NET circumvents all these problems. For starters, ASP.NET web pages are executed within the CLR (common language runtime), so they can be authored in any language that has a CLR-compliant compiler. No longer are you limited to using VBScript or JavaScript—instead, you can use modern object-oriented languages such as C# or Visual Basic.

It's also important to note that ASP.NET pages are not interpreted but are instead compiled into *assemblies* (the .NET term for any unit of compiled code). This is one of the most significant enhancements to Microsoft's web development model. Even if you create your C# or Visual Basic code in Notepad and copy it directly to a virtual directory on a web server, the application is dynamically compiled as soon as a client accesses it, and it is cached for future requests. If any of the files are modified after this compilation process, the application is recompiled automatically the next time a client requests it.

ASP.NET

Microsoft developers have described ASP.NET as their chance to "hit the reset button" and start from scratch with an entirely new, more modern development model. The traditional concepts involved in creating web applications still hold true in the .NET world. Each web application consists of web pages. You can render rich HTML and even use JavaScript, create components that encapsulate programming logic, and tweak and tune your applications using configuration settings. However, behind the scenes ASP.NET works quite differently than traditional scripting technologies such as classic ASP or PHP.

Some of the differences between ASP.NET and earlier web development platforms include the following:

- ASP.NET features a completely object-oriented programming model, which includes an event-driven, control-based architecture that encourages code encapsulation and code reuse.
- ASP.NET gives you the ability to code in any supported .NET language (including Visual Basic, C#, J#, and many other languages that have third-party compilers).
- ASP.NET is dedicated to high performance. ASP.NET pages and components are compiled on demand instead of being interpreted every time they're used. ASP.NET also includes a fine-tuned data access model and flexible data caching to further boost performance.

These are only a few of the features, which include enhanced state management, practical data binding, dynamic graphics, and a robust security model. You'll look at these improvements in detail in this book and see what ASP.NET 3.5 adds to the picture.

Seven Important Facts About ASP.NET

If you're new to ASP.NET (or you just want to review a few fundamentals), you'll be interested in the following sections. They introduce seven touchstones of .NET development.

Fact 1: ASP.NET Is Integrated with the .NET Framework

The .NET Framework is divided into an almost painstaking collection of functional parts, with a staggering total of more than 10,000 *types* (the .NET term for classes, structures, interfaces, and other core programming ingredients). Before you can program any type of .NET application, you need a basic understanding of those parts—and an understanding of why things are organized the way they are.

The massive collection of functionality that the .NET Framework provides is organized in a way that traditional Windows programmers will see as a happy improvement. Each one of the thousands of classes in the .NET Framework is grouped into a logical, hierarchical container called a *namespace*. Different namespaces provide different features. Taken together, the .NET namespaces offer functionality for nearly every aspect of distributed development from message queuing to security. This massive toolkit is called the *class library*.

Interestingly, the way you use the .NET Framework classes in ASP.NET is the same as the way you use them in any other type of .NET application (including a stand-alone Windows application, a Windows service, a command-line utility, and so on). In other words, .NET gives the same tools to web developers that it gives to rich client developers.

■ **Tip** One of the best resources for learning about new corners of the .NET Framework is the .NET Framework class library reference, which is part of the MSDN Help library reference. If you have Visual Studio 2008 installed, you can view the MSDN Help library by selecting Start ➤ Programs ➤ Microsoft Visual Studio 2008 ➤ Microsoft Visual Studio 2008 Documentation (the exact shortcut depends on your version of Visual Studio). Once you've loaded the help, you can find class reference information grouped by namespace under the .NET Development ➤ .NET Framework SDK ➤ .NET Framework ➤ .NET Framework Class Library node.

Fact 2: ASP.NET Is Compiled, Not Interpreted

One of the major reasons for performance degradation in classic ASP pages is its use of interpreted script code. Every time an ASP page is executed, a scripting host on the web server needs to interpret the script code and translate it to lower-level machine code, line by line. This process is notoriously slow.

■ **Note** In fact, in this case the reputation is a little worse than the reality. Interpreted code is certainly slower than compiled code, but the performance hit isn't so significant that you can't build a professional website using old-style ASP.

ASP.NET applications are always compiled—in fact, it's impossible to execute C# or Visual Basic code without it being compiled first.

ASP.NET applications actually go through two stages of compilation. In the first stage, the C# code you write is compiled into an intermediate language called Microsoft Intermediate Language (MSIL), or just IL. This first step is the fundamental reason that .NET can be language-interdependent. Essentially, all .NET languages (including C#, Visual Basic, and many more) are compiled into virtually identical IL code. This first compilation step may happen automatically when the page is first requested, or you can perform it in advance (a process known as *precompiling*). The compiled file with IL code is an *assembly*.

The second level of compilation happens just before the page is actually executed. At this point, the IL code is compiled into low-level native machine code. This stage is known as *just-in-time* (JIT) compilation, and it takes place in the same way for all .NET applications (including Windows applications, for example). Figure 1-1 shows this two-step compilation process.

.NET compilation is decoupled into two steps in order to offer developers the most convenience and the best portability. Before a compiler can create low-level machine code, it needs to know what type of operating system and hardware platform the application will run on (for example, 32-bit or 64-bit Windows). By having two compile stages, you can create a compiled assembly with .NET code and still distribute this to more than one platform.

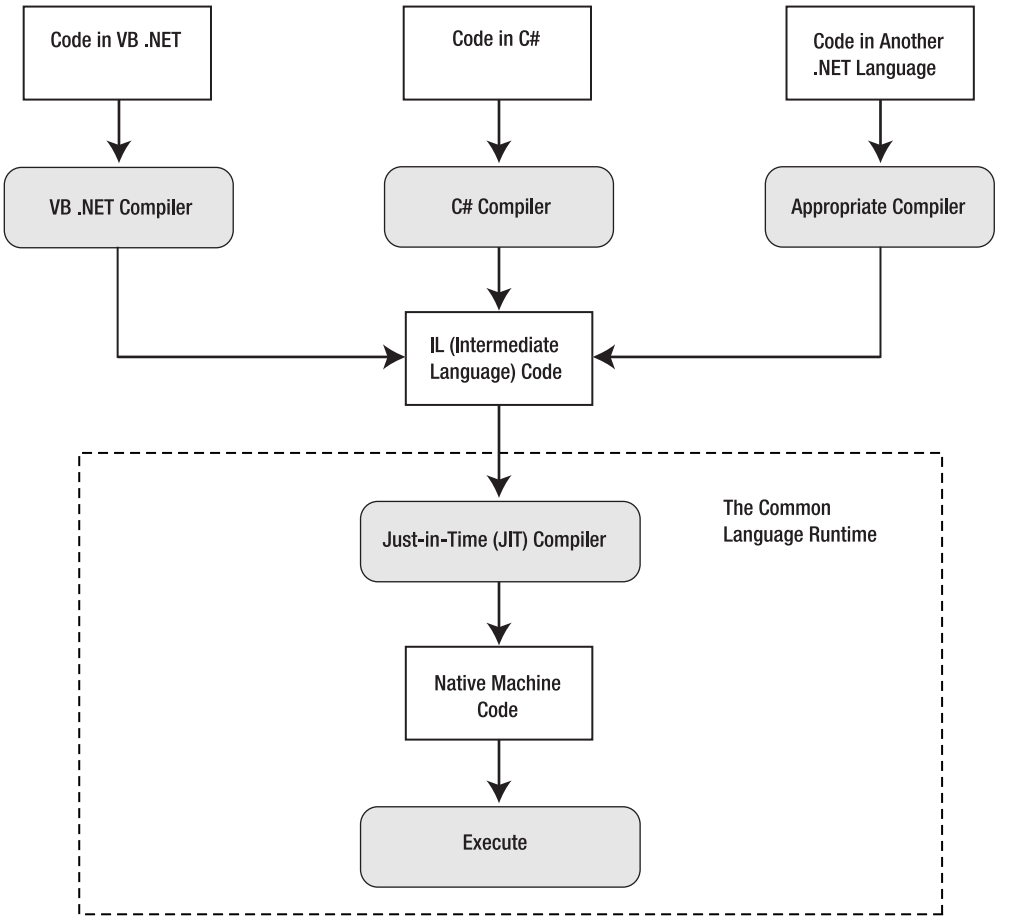

Figure 1-1. *Compilation in an ASP.NET web page*

Of course, JIT compilation probably wouldn't be that useful if it needed to be performed every time a user requested a web page from your site. Fortunately, ASP.NET applications don't need to be compiled every time a web page is requested. Instead, the IL code is created once and regenerated only when the source is modified. Similarly, the native machine code files are cached in a system

directory that has a path like c:\Windows\Microsoft.NET\Framework\v2.0.50727\Temporary ASP.NET Files.

Note You might wonder why the temporary ASP.NET files are found in a directory with a 2.0 version number rather than a 3.5 version number. Technically, ASP.NET 3.5 uses the ASP.NET 2.0 engine (with a few new features piled on). You'll learn more about this design later in the chapter, in the "ASP.NET 3.5: The Story Continues" section.

As you'll learn in Chapter 2, the actual point where your code is compiled to IL depends on how you're creating and deploying your web application. If you're building a web project in Visual Studio, the code is compiled to IL when you compile your project. But if you're building a lighter-weight projectless website, the code for each page is compiled the first time you request that page. Either way, the code goes through its second compilation step (from IL to machine code) the first time it's executed.

ASP.NET also includes precompilation tools that you can use to compile your application right down to machine code once you've deployed it to the production web server. This allows you to avoid the overhead of first-time compilation when you deploy a finished application (and prevent other people from tampering with your code). Precompilation is described in Chapter 18.

Note Although benchmarks are often controversial, you can find a few interesting comparisons of Java and ASP.NET at `http://msdn2.microsoft.com/en-us/vstudio/aa700836.aspx`. Keep in mind that the real issues limiting performance are usually related to specific bottlenecks, such as disk access, CPU use, network bandwidth, and so on. In many benchmarks, ASP.NET outperforms other solutions because of its support for performance-enhancing platform features such as caching, not because of the speed boost that results from compiled code.

Fact 3: ASP.NET Is Multilanguage

Though you'll probably opt to use one language over another when you develop an application, that choice won't determine what you can accomplish with your web applications. That's because no matter what language you use, the code is compiled into IL.

IL is a stepping stone for every managed application. (A *managed application* is any application that's written for .NET and executes inside the managed environment of the CLR.) In a sense, IL is *the* language of .NET, and it's the only language that the CLR recognizes.

To understand IL, it helps to consider a simple example. Take a look at this code written in C#:

```
using System;

namespace HelloWorld
{
    public class TestClass
    {
        private static void Main(string[] args)
        {
            Console.WriteLine("Hello World");
        }
    }
}
```

This code shows the most basic application that's possible in .NET—a simple command-line utility that displays a single, predictable message on the console window.

Now look at it from a different perspective. Here's the IL code for the Main()method:

```
.method private hidebysig static void  Main(string[] args) cil managed
{
    .entrypoint
    // Code size       13 (0xd)
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ldstr    "Hello World"
    IL_0006:  call     void [mscorlib]System.Console::WriteLine(string)
    IL_000b:  nop
    IL_000c:  ret
} // end of method TestClass::Main
```

It's easy enough to look at the IL for any compiled .NET application. You simply need to run the IL Disassembler (named ildasm.exe), which is installed with Visual Studio and the .NET SDK (software development kit). The easiest way to run the IL Disassembler is to use the Visual Studio Command Prompt (open the Start menu and choose Programs ➤ Microsoft Visual Studio 2008 ➤ Visual Studio Tools ➤ Visual Studio 2008 Command Prompt. At the command prompt, type ildasm.exe. Once you've loaded ildasm.exe, use the File ➤ Open command, and select any DLL or EXE that was created with .NET.

■ **Tip** For even more disassembling power, check out the remarkable (and free) Reflector tool at `http://www.aisto.com/roeder/dotnet`. With the help of community-created add-ins, you can use Reflector to diagram, analyze, and decompile the IL code in any assembly.

If you're patient and a little logical, you can deconstruct the IL code fairly easily and figure out what's happening. The fact that IL is so easy to disassemble can raise privacy and code control issues, but these issues usually aren't of any concern to ASP.NET developers. That's because all ASP.NET code is stored and executed on the server. Because the client never receives the compiled code file, the client has no opportunity to decompile it. If it *is* a concern, consider using an obfuscator that scrambles code to try to make it more difficult to understand. (For example, an obfuscator might rename all variables to have generic, meaningless names such as f__a__234.) Visual Studio includes a scaled-down version of one popular obfuscator, called Dotfuscator.

The following code shows the same console application in Visual Basic code:

```
Imports System

Namespace HelloWorld
    Public Class TestClass
        Private Shared Sub Main(args() As String)
            Console.WriteLine("Hello World")
        End Sub
    End Class
End Namespace
```

THE COMMON LANGUAGE SPECIFICATION

The CLR expects all objects to adhere to a specific set of rules so that they can interact. The CLS is this set of rules.

The CLS defines many laws that all languages must follow, such as primitive types, method overloading, and so on. Any compiler that generates IL code to be executed in the CLR must adhere to all rules governed within the CLS. The CLS gives developers, vendors, and software manufacturers the opportunity to work within a common set of specifications for languages, compilers, and data types. You can find a list of a large number of CLS-compliant languages at `http://dotnetpowered.com/languages.aspx`.

Given these criteria, the creation of a language compiler that generates true CLR-compliant code can be complex. Nevertheless, compilers can exist for virtually any language, and chances are that there may eventually be one for just about every language you'd ever want to use. Imagine—mainframe programmers who loved COBOL in its heyday can now use their knowledge base to create web applications!

If you compile this application and look at the IL code, you'll find that it's nearly identical to the IL code generated from the C# version. Although different compilers can sometimes introduce their own optimizations, as a general rule of thumb no .NET language outperforms any other .NET language, because they all share the same common infrastructure. This infrastructure is formalized in the CLS (Common Language Specification), which is described in the previous sidebar, entitled "The Common Language Specification."

It's worth noting that IL has been adopted as an Ecma and ISO standard. This adoption could quite possibly spur the adoption of other common language frameworks on other platforms. The Mono project at `http://www.mono-project.com` is an example of one such project.

Fact 4: ASP.NET Is Hosted by the Common Language Runtime

Perhaps the most important aspect of the ASP.NET engine is that it runs inside the runtime environment of the CLR. The whole of the .NET Framework—that is, all namespaces, applications, and classes—is referred to as *managed* code. Though a full-blown investigation of the CLR is beyond the scope of this chapter, some of the benefits are as follows:

Automatic memory management and garbage collection: Every time your application instantiates a reference-type object, the CLR allocates space on the *managed heap* for that object. However, you never need to clear this memory manually. As soon as your reference to an object goes out of scope (or your application ends), the object becomes available for garbage collection. The garbage collector runs periodically inside the CLR, automatically reclaiming unused memory for inaccessible objects. This model saves you from the low-level complexities of C++ memory handling and from the quirkiness of COM reference counting.

Type safety: When you compile an application, .NET adds information to your assembly that indicates details such as the available classes, their members, their data types, and so on. As a result, other applications can use them without requiring additional support files, and the compiler can verify that every call is valid at runtime. This extra layer of safety completely obliterates whole categories of low-level errors.

Extensible metadata: The information about classes and members is only one of the types of metadata that .NET stores in a compiled assembly. *Metadata* describes your code and allows you to provide additional information to the runtime or other services. For example, this metadata might tell a debugger how to trace your code, or it might tell Visual Studio how to display a custom control at design time. You could also use metadata to enable other runtime services, such as transactions or object pooling.

Structured error handling: .NET languages offer structured exception handling, which allows you to organize your error-handling code logically and concisely. You can create separate blocks to deal with different types of errors. You can also nest exception handlers multiple layers deep.

Multithreading: The CLR provides a pool of threads that various classes can use. For example, you can call methods, read files, or communicate with web services asynchronously, without needing to explicitly create new threads.

Figure 1-2 shows a high-level look at the CLR and the .NET Framework.

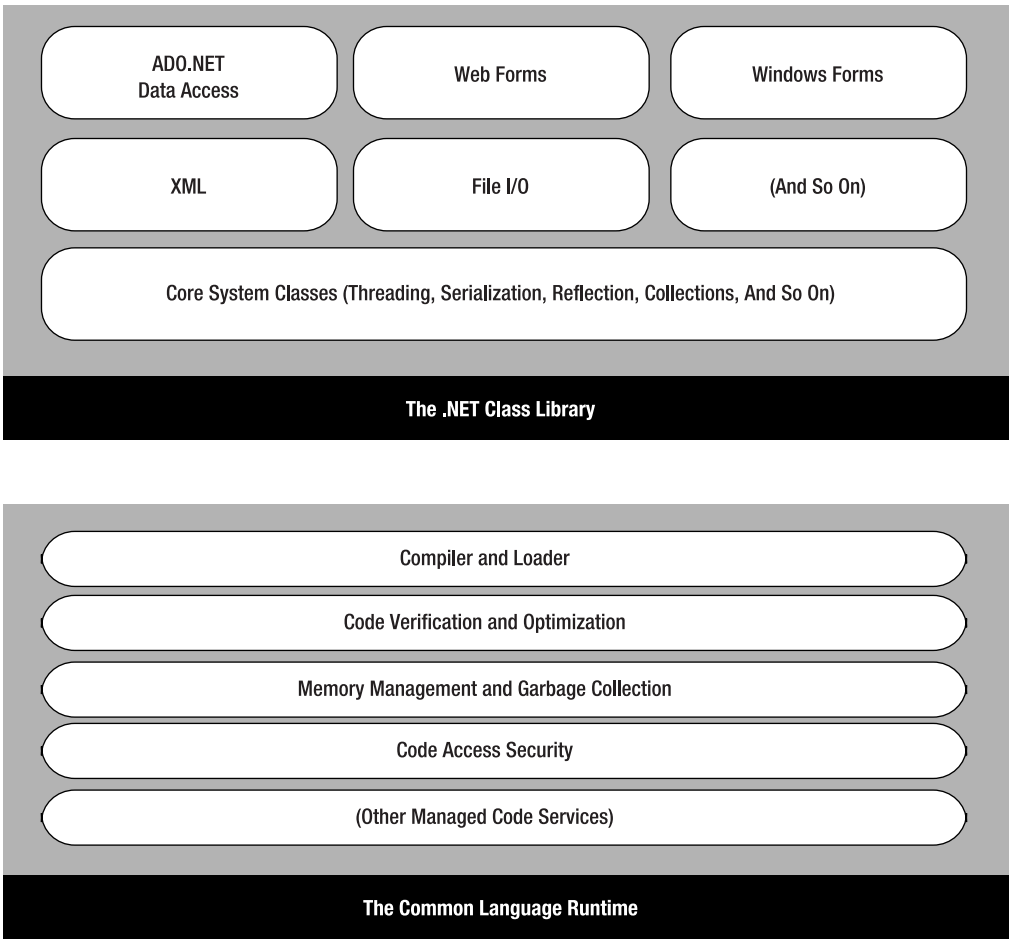

Figure 1-2. *The CLR and the .NET Framework*

Fact 5: ASP.NET Is Object-Oriented

ASP provides a relatively feeble object model. It provides a small set of objects; these objects are really just a thin layer over the raw details of HTTP and HTML. On the other hand, ASP.NET is truly object-oriented. Not only does your code have full access to all objects in the .NET Framework, but you can also exploit all the conventions of an OOP (object-oriented programming) environment. For example, you can create reusable classes, standardize code with interfaces, extend existing classes with inheritance, and bundle useful functionality in a distributable, compiled component.

One of the best examples of object-oriented thinking in ASP.NET is found in *server-based controls*. Server-based controls are the epitome of encapsulation. Developers can manipulate control objects programmatically using code to customize their appearance, provide data to display, and even react to events. The low-level HTML markup that these controls render is hidden away behind the scenes. Instead of forcing the developer to write raw HTML manually, the control objects render themselves to HTML when the page is finished rendering. In this way, ASP.NET offers server controls as a way to abstract the low-level details of HTML and HTTP programming.

HTML CONTROLS VS. WEB CONTROLS

When ASP.NET was first created, two schools of thought existed. Some ASP.NET developers were most interested in server-side controls that matched the existing set of HTML controls exactly. This approach allows you to create ASP.NET web-page interfaces in dedicated HTML editors, and it provides a quick migration path for existing ASP pages. However, another set of ASP.NET developers saw the promise of something more—rich server-side controls that didn't just emulate individual HTML tags. These controls might render their interface from dozens of distinct HTML elements while still providing a simple object-based interface to the programmer. Using this model, developers could work with programmable menus, calendars, data lists, validators, and so on.

After some deliberation, Microsoft decided to provide both models. You've already seen an example of HTML server controls, which map directly to the basic set of HTML tags. Along with these are ASP.NET *web controls*, which provide a higher level of abstraction and more functionality. In most cases, you'll use HTML server-side controls for backward compatibility and quick migration, and use web controls for new projects.

ASP.NET web control tags always start with the prefix *asp:* followed by the class name. For example, the following snippet creates a text box and a check box:

```
<asp:TextBox id="myASPText" Text="Hello ASP.NET TextBox" runat="server" />
<asp:CheckBox id="myASPCheck" Text="My CheckBox" runat="server" />
```

Again, you can interact with these controls in your code, as follows:

```
myASPText.Text = "New text";
myASPCheck.Text = "Check me!";
```

Notice that the Value property you saw with the HTML control has been replaced with a Text property. The HtmlInputText.Value property was named to match the underlying value attribute in the HTML <input> tag. However, web controls don't place the same emphasis on correlating with HTML syntax, so the more descriptive property name Text is used instead.

The ASP.NET family of web controls includes complex rendered controls (such as the Calendar and TreeView), along with more streamlined controls (such as TextBox, Label, and Button), which map closely to existing HTML tags. In the latter case, the HTML server-side control and the ASP.NET web control variants provide similar functionality, although the web controls tend to expose a more standardized, streamlined interface. This makes the web controls easy to learn, and it also means they're a natural fit for Windows developers moving to the world of the Web, because many of the property names are similar to the corresponding Windows controls.

Here's a quick example with a standard HTML text box:

```
<input type="text" id="myText" runat="server" />
```

With the addition of the runat="server" attribute, this static piece of HTML becomes a fully functional server-side control that you can manipulate in your code. You can now work with events that it generates, set attributes, and bind it to a data source.

For example, you can set the text of this box when the page first loads using the following C# code:

```
void Page_Load(object sender, EventArgs e)
{
    myText.Value = "Hello World!";
}
```

Technically, this code sets the Value property of an HtmlInputText object. The end result is that a string of text appears in a text box on the HTML page that's rendered and sent to the client.

Fact 6: ASP.NET Is Multidevice and Multibrowser

One of the greatest challenges web developers face is the wide variety of browsers they need to support. Different browsers, versions, and configurations differ in their support of HTML. Web developers need to choose whether they should render their content according to HTML 3.2, HTML 4.0, or something else entirely—such as XHTML 1.0 or even WML (Wireless Markup Language) for mobile devices. This problem, fueled by the various browser companies, has plagued developers since the World Wide Web Consortium (W3C) proposed the first version of HTML. Life gets even more complicated if you want to use an HTML extension such as JavaScript to create a more dynamic page or provide validation.

ASP.NET addresses this problem in a remarkably intelligent way. Although you can retrieve information about the client browser and its capabilities in an ASP.NET page, ASP.NET actually encourages developers to ignore these considerations and use a rich suite of web server controls. These server controls render their HTML adaptively by taking the client's capabilities into account. One example is ASP.NET's validation controls, which use JavaScript and DHTML (Dynamic HTML) to enhance their behavior if the client supports it. This allows the validation controls to show dynamic error messages without the user needing to send the page back to the server for more processing. These features are optional, but they demonstrate how intelligent controls can make the most of cutting-edge browsers without shutting out other clients. Best of all, you don't need any extra coding work to support both types of client.

■ **Note** Unfortunately, ASP.NET 3.5 still hasn't managed to integrate mobile controls into the picture. As a result, if you want to create web pages for *smart devices* such as mobile phones, PDAs (personal digital assistants), and so on, you need to use a similar but separate toolkit. The architects of ASP.NET originally planned to unify these two models so that the standard set of server controls could render markup using a scaled-down standard such as WML or HDML (Handheld Device Markup Language) instead of HTML. However, this feature was cut late in the beta cycle.

Fact 7: ASP.NET Is Easy to Deploy and Configure

One of the biggest headaches a web developer faces during a development cycle is deploying a completed application to a production server. Not only do the web-page files, databases, and components need to be transferred, but components need to be registered and a slew of configuration settings need to be re-created. ASP.NET simplifies this process considerably.

Every installation of the .NET Framework provides the same core classes. As a result, deploying an ASP.NET application is relatively simple. For no-frills deployment, you simply need to copy all the files to a virtual directory on a production server (using an FTP program or even a command-line command like XCOPY). As long as the host machine has the .NET Framework, there are no time-consuming registration steps. Chapter 18 covers deployment in detail.

Distributing the components your application uses is just as easy. All you need to do is copy the component assemblies along with your website files when you deploy your web application. Because all the information about your component is stored directly in the assembly file metadata, there's no need to launch a registration program or modify the Windows registry. As long as you place these components in the correct place (the Bin subdirectory of the web application directory), the ASP.NET engine automatically detects them and makes them available to your web-page code. Try that with a traditional COM component!

Configuration is another challenge with application deployment, particularly if you need to transfer security information such as user accounts and user privileges. ASP.NET makes this deployment process easier by minimizing the dependence on settings in IIS (Internet Information Services). Instead, most ASP.NET settings are stored in a dedicated web.config file. The web.config file is placed in the same directory as your web pages. It contains a hierarchical grouping of application settings stored in an easily readable XML format that you can edit using nothing more than a text editor such as Notepad. When you modify an application setting, ASP.NET notices that change and smoothly restarts the application in a new application domain (keeping the existing application domain alive long enough to finish processing any outstanding requests). The web.config file is never locked, so it can be updated at any time.

ASP.NET 3.5: The Story Continues

When Microsoft released ASP.NET 1.0, even it didn't anticipate how enthusiastically the technology would be adopted. ASP.NET quickly became the standard for developing web applications with Microsoft technologies and a heavy-hitting competitor against all other web development platforms. Since that time, ASP.NET has had one minor release (ASP.NET 1.1) and two more significant releases (ASP.NET 2.0 and ASP.NET 3.5).

■ **Note** Adoption statistics are always contentious, but the highly regarded Internet analysis company Netcraft (`http://www.netcraft.com`) suggests that ASP.NET usage continues to surge and that it now runs on more web servers than JSP. This survey doesn't weigh the relative size of these websites, but ASP.NET powers the websites for a significant number of Fortune 1000 companies and heavily trafficked Web destinations like MySpace.

For the most part, this book won't distinguish between the features that are new in ASP.NET 3.5 and those that have existed since ASP.NET 2.0 and ASP.NET 1.0. However, in the next few sections you'll take a quick look at how ASP.NET has evolved.

ASP.NET 2.0

It's a testament to the good design of ASP.NET 1.0 and 1.1 that few of the changes introduced in ASP.NET 2.0 were fixes for existing features. Instead, ASP.NET 2.0 kept the same underlying plumbing and concentrated on adding new, higher-level features. Some of the highlights include the following:

- **More rich controls**: ASP.NET 2.0 introduced more than 40 new controls, from long-awaited basics like a collapsible TreeView to a JavaScript-powered Menu.
- **Master pages**: Master pages are reusable page templates. For example, you can use a master page to ensure that every web page in your application has the same header, footer, and navigation controls.
- **Themes**: Themes allow you to define a standardized set of appearance characteristics for web controls. Once defined, you can apply these formatting presets across your website for a consistent look.
- **Security and membership**: ASP.NET 2.0 added a slew of security-related features, including automatic support for storing user credentials, a role-based authorization feature, and prebuilt security controls for common tasks like logging in, registering, and retrieving a forgotten password.
- **Data source controls**: The data source control model allows you to define how your page interacts with a data source *declaratively* in your markup, rather than having to write the equivalent data access code by hand. Best of all, this feature doesn't force you to abandon good component-based design—you can bind to a custom data component just as easily as you bind directly to the database.
- **Web parts**: One common type of web application is the *portal*, which centralizes different information using separate panes on a single web page. Web parts provide a prebuilt portal framework complete with a flow-based layout, configurable views, and even drag-and-drop support.
- **Profiles**: This feature allows you to store user-specific information in a database without writing any database code. Instead, ASP.NET takes care of the tedious work of retrieving the profile data when it's needed and saving the profile data when it changes.

THE PROVIDER MODEL

Many of the features introduced in ASP.NET 2.0 work through an abstraction called the *provider model*. The beauty of the provider model is that you can use the simple providers to build your page code. If your requirements change, you don't need to change a single page—instead, you simply need to create a custom provider.

For example, most serious developers will quickly realize that the default implementation of profiles is a one-size-fits-all solution that probably won't suit their needs. It doesn't work if you need to use existing database tables, store encrypted information, or customize how large amounts of data are cached to improve performance. However, you can customize the profile feature to suit your needs by building your own profile provider. This allows you to use the convenient profile features but still control the low-level details. Of course, the drawback is that you're still responsible for some of the heavy lifting, but you gain the flexibility and consistency of the profile model.

You'll learn how to use provider-based features and create your own providers throughout this book.

ASP.NET 3.5

Developers who are facing ASP.NET 3.5 for the first time are likely to wonder what happened to ASP.NET 3.0. Oddly enough, it doesn't exist. Microsoft used the name .NET Framework 3.0 to release new technologies—most notably, WPF (Windows Presentation Foundation), a slick new user interface technology for building rich clients, WCF (Windows Communication Foundation), a technology for building message-oriented services, and WF (Windows Workflow Foundation), a technology that allows you to model a complex business process as a series of actions (optionally using a visual flow-chart-like designer). However, the .NET Framework 3.0 doesn't include a new version of the CLR or ASP.NET. Instead, the next release of ASP.NET was rolled into the .NET Framework 3.5.

Compared to ASP.NET 2.0, ASP.NET 3.5 is a more gradual evolution. Its new features are concentrated in two areas: LINQ and Ajax, as described in the following sections.

LINQ

LINQ (Language Integrated Query) is a set of extensions for the C# and Visual Basic languages. It allows you to write C# or Visual Basic code that manipulates in-memory data in much the same way you query a database.

Technically, LINQ defines about 40 query operators, such as select, from, in, where, and orderby (in C#). These operators allow you to code your query. However, there are various types of data on which this query can be performed, and each type of data requires a separate flavor of LINQ.

The most fundamental LINQ flavor is *LINQ to Objects*, which allows you to take a collection of objects and perform a query that extracts some of the details from some of the objects. LINQ to Objects isn't ASP.NET-specific. In other words, you can use it in a web page in exactly the same way that you use it in any other type of .NET application.

Along with LINQ to Objects is *LINQ to DataSet*, which provides similar behavior for querying an in-memory DataSet object, and *LINQ to XML*, which works on XML data. Third-party developers and tool providers are certain to create more LINQ providers. However, the flavor of LINQ that's attracted the most attention is *LINQ to SQL*, which allows you to use the LINQ syntax to execute a query against a SQL Server database. Essentially, LINQ to SQL creates a properly parameterized SQL query based on your code, and executes the query when you attempt to access the query results. You don't need to write any data access code or use the traditional ADO.NET objects.

LINQ to Objects, LINQ to DataSet, and LINQ to XML are features that complement ASP.NET, and aren't bound to it in any specific way. However, ASP.NET includes enhanced support for LINQ to SQL, including a data source control that lets you perform a query through LINQ to SQL and bind the results to a web control, with no extra code required. You'll take a look at LINQ to Objects, LINQ to DataSet, and LINQ to SQL in Chapter 13. You'll consider LINQ to XML in Chapter 14.

ASP.NET AJAX

Recently, web developers have refocused to consider some of the weaknesses of web applications. One of the most obvious is the lack of *responsiveness* in server-side programming platforms such as ASP.NET. Because ASP.NET does all its work on the web server, every time an action occurs in the page the browser needs to post some data to the server, get a new copy of the page, and refresh the display. This process, though fast, introduces a noticeable flicker. It also takes enough time that it isn't practical to respond to events that fire frequently, such as mouse movements or key presses.

Web developers work around these sorts of limitations using JavaScript, the only broadly supported client-side scripting language. In ASP.NET, many of the most powerful controls use a healthy bit of JavaScript. For example, the Menu control responds immediately as the user moves the mouse over different subheadings. When you use the Menu control, your page doesn't post back to the server until the user clicks an item.

In traditional ASP.NET pages, developers use server controls such as Menu and gain the benefit of the client-side script these controls emit. However, even with advanced controls, some postbacks are unavoidable. For example, if you need to update the information on a portion of the page, the only way to accomplish this in ordinary ASP.NET is to post the page back to the server and get an entirely new HTML document. The solution works, but it isn't seamless.

Restless web developers have responded to challenges like these by using more client-side code and applying it in more advanced ways. One of the most talked about examples today is *Ajax* (Asynchronous JavaScript and XML). Ajax is programming shorthand for a client-side technique that allows your page to call the server and update its content without triggering a complete postback. Typically, an Ajax page uses client-side script code to fire an asynchronous request behind the scenes. The server receives this request, runs some code, and then returns the data your page needs (often as a block of XML markup). Finally, the client-side code receives the new data and uses it to perform another action, such as refreshing part of the page. Although Ajax is conceptually quite simple, it allows you to create pages that work more like seamless, continuously running applications. Figure 1-3 illustrates the differences.

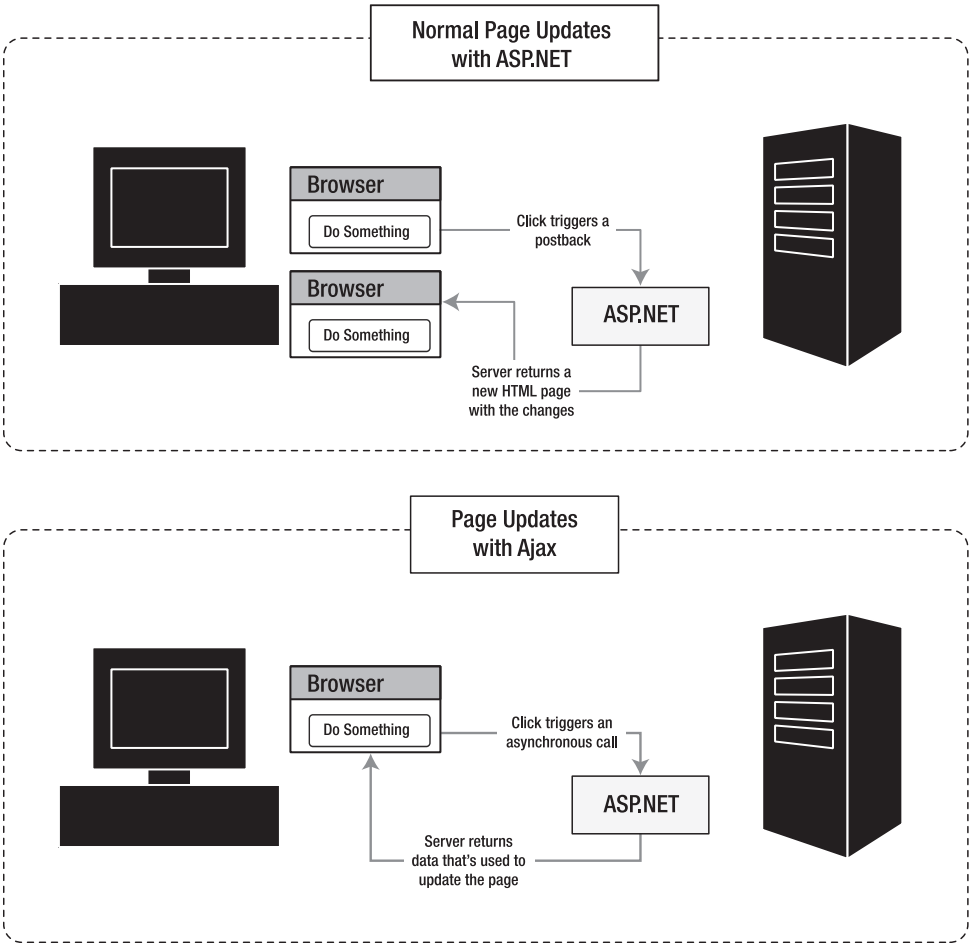

Figure 1-3. *Ordinary server-side pages vs. Ajax*

Ajax and similar client-side scripting techniques are nothing new, but in recent years they've begun to play an increasingly important role in web development. One of the reasons is that the XMLHttpRequest object—the plumbing that's required to support asynchronous client requests—is now present in the majority of modern browsers, including the following:

- Internet Explorer 5 and newer
- Netscape 7 and newer
- Opera 7.6 and newer
- Safari 1.2 and newer
- Firefox 1.0 and newer

However, writing the client-side script in such a way that it's compatible with all browsers and implementing all the required pieces (including the server-side code that handles the asynchronous requests) can be a bit tricky. As you'll see in Chapter 31, ASP.NET provides a client callback feature that handles some of the work. However, ASP.NET also includes a much more powerful abstraction layer named ASP.NET AJAX, which extends ASP.NET with impressive Ajax-style features.

■ **Note** It's generally accepted that Ajax isn't written in all capitals, because the word isn't an acronym. However, Microsoft chose to capitalize it when naming ASP.NET AJAX. For that reason, you'll see two capitalizations of Ajax in this book—*Ajax* when talking in general terms about the technology and philosophy of Ajax, and *AJAX* when talking about ASP.NET AJAX, which is Microsoft's specific implementation of these concepts.

ASP.NET AJAX is a multilayered technology that gives you a wide range of options for integrating Ajax features into ordinary ASP.NET web pages. At the lowest level, you can use ASP.NET AJAX to write more powerful JavaScript. At higher levels, you can use server-side components to harness new features such as autocompletion, drag and drop, and animation without doing any of the client-side programming yourself. You'll explore ASP.NET AJAX in Chapter 32.

Green Bits and Red Bits

Oddly enough, ASP.NET 3.5 doesn't include a full new version of ASP.NET. Instead, ASP.NET 3.5 is designed as a set of features that add on to .NET 2.0. The parts of .NET that haven't changed in .NET 3.5 are often referred to as *red bits*, while the parts that have changed are called *green bits*.

The red bits include the CLR, the ASP.NET engine, and all the class libraries from .NET 2.0. In other words, if you build a new ASP.NET 3.5 application, it gets exactly the same runtime environment as an ASP.NET 2.0 application. Additionally, all the classes you used in .NET 2.0—including those for connecting to a database, reading and writing files, using web controls, and so on—remain the same in .NET 3.5. The red bits also include the features that were included in .NET 3.0, such as WCF.

All the assemblies in .NET 3.5 retain their original version numbers. That means .NET 3.5 includes a mix of 2.0, 3.0, and 3.5 assemblies. If you're curious when an assembly was released, you simply need to check the version number.

■ **Note** In many cases, the red bits *do* have minor changes—most commonly for bug fixes. On the whole, the level of changes is about the same as a service pack release.

The ASP.NET 3.5 green bits consist of a small set of assemblies with new types. For ASP.NET developers, the important new assemblies include the following:

- **System.Core.dll**: Includes the core LINQ functionality
- **System.Data.Linq.dll**: Includes the implementation for LINQ to SQL
- **System.Data.DataSetExtensions.dll:**. Includes the implementation for LINQ to DataSet
- **System.Xml.Linq.dll**: Includes the implementation for LINQ to XML
- **System.Web.Extensions.dll**: Includes the implementation for ASP.NET AJAX and new web controls

When building an ASP.NET 3.5 application, you'll use the C# 3.0 language compiler. It includes support for a few new features, most of which are required for LINQ. Figure 1-4 shows the collection of classes and components that comprise ASP.NET 3.5.

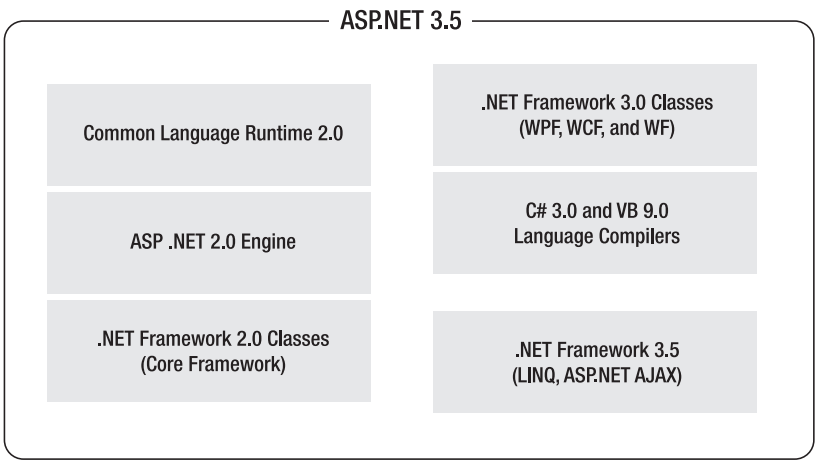

Figure 1-4. *The components of ASP.NET 3.5*

■ **Note** To match the Visual Studio 2008 product name, Microsoft has rebranded C# 3.0 as C# 2008, and VB 9.0 as VB 2008.

It's important to understand the multilayered architecture of ASP.NET 3.5, because you'll still see some of the fingerprints of past versions. For example, ASP.NET places temporary files and web server configuration files in subdirectories of the c:\Windows\Microsoft.NET\Framework\v2.0.50727 directory. This is because ASP.NET 3.5 uses the ASP.NET 2.0 engine, and the final release version of ASP.NET 2.0 was v2.0.50727.

Silverlight

Recently, there's been a lot of excitement about *Silverlight*, a new Microsoft technology that allows a variety of browsers on a variety of operating systems to run true .NET code. Silverlight works through a browser plug-in, and provides a subset of the .NET Framework class library. This subset includes a slimmed-down version of WPF, the technology that developers use to craft next-generation Windows user interfaces.

So where does Silverlight fit into the ASP.NET world? Silverlight is all about client code—quite simply, it allows you to create richer pages than you could with HTML, DHTML, and JavaScript alone. In many ways, Silverlight duplicates the features and echoes the goals of Adobe Flash. By using Silverlight in a web page, you can draw sophisticated 2D graphics, animate a scene, and play video and other media files.

Silverlight is perfect for creating a mini-applet, like a browser-hosted game. It's also a good choice for adding interactive media and animation to a website. However, Silverlight obviously *isn't* a good choice for tasks that require server-side code, such as performing a secure checkout in an e-commerce shop, verifying user input, or interacting with a server-side database. And because Silverlight is still a new, emerging technology, it's too early to make assumptions about its rate of adoption. That means it's not a good choice to replace basic ingredients in a website with Silverlight content. For example, although you can use Silverlight to create an animated button, this is a risky strategy. Users without the Silverlight plug-in won't be able to see your button or interact with it. (Of course, you could create more than one front end for your web application, using Silverlight if it's available or falling back on regular ASP.NET controls if it's not. However, this approach requires a significant amount of work.)

In many respects, Silverlight is a complementary technology to ASP.NET. ASP.NET 3.5 doesn't include any features that use Silverlight, but future versions of ASP.NET will. For example, a future version of ASP.NET might include a server control that emits Silverlight content. By adding this control to your web page, you would gain the best of both worlds—the server-side programming model of ASP.NET and the rich interactivity of client-side Silverlight.

In Chapter 33, you'll get a thorough introduction to Silverlight. You'll also look at the ASP.NET Futures add-in, which gives you a preview of features that may appear in future versions of ASP.NET, including web server controls that render Silverlight content.

Summary

So far, you've just scratched the surface of the features and frills that are provided in ASP.NET and the .NET Framework. You've taken a quick look at the high-level concepts you need to understand in order to be a competent ASP.NET programmer. You've also previewed the new features that ASP.NET 3.5 offers. As you continue through this book, you'll learn much more about the innovations and revolutions of ASP.NET 3.5 and the .NET Framework.