Microsoft has a time-honored reputation for creating innovative technologies and wrapping them in buzzwords that confuse everyone. The .NET Framework is the latest example—it's been described as a feeble Java clone, a meaningless marketing term, and an attempt to take over the Internet with proprietary technology. But none of these descriptions is truly accurate. .NET is actually a cluster of technologies—some revolutionary, some not—that are designed to help developers build a variety of different types of applications. Developers can use the .NET Framework to build rich Windows applications, long-running services, and even command-line tools. Of course, if you're reading this book you're most interested in using .NET to craft web applications. You'll use a specific subset of the .NET Framework called ASP.NET, and you'll work with one of .NET's core languages: C#.
In this you'll examine the technologies that underlie .NET. First, you'll take a quick look at the history of web development and learn why the .NET Framework was created. Next, you'll get a high-level overview of the different parts of .NET and see how ASP.NET 3.5 fits into the picture.

The Evolution of Web Development

The Internet began in the late 1960s as an experiment. Its goal was to create a truly resilient information network—one that could withstand the loss of several computers without preventing the others from communicating. Driven by potential disaster scenarios (such as nuclear attack), the U.S. Department of Defense provided the initial funding. The early Internet was mostly limited to educational institutions and defense contractors. It flourished as a tool for academic collaboration, allowing researchers across the globe to share information. In the early 1990s, modems were created that could work over existing phone lines, and the Internet began to open up to commercial users. In 1993, the first HTML browser was created, and the Internet revolution began.

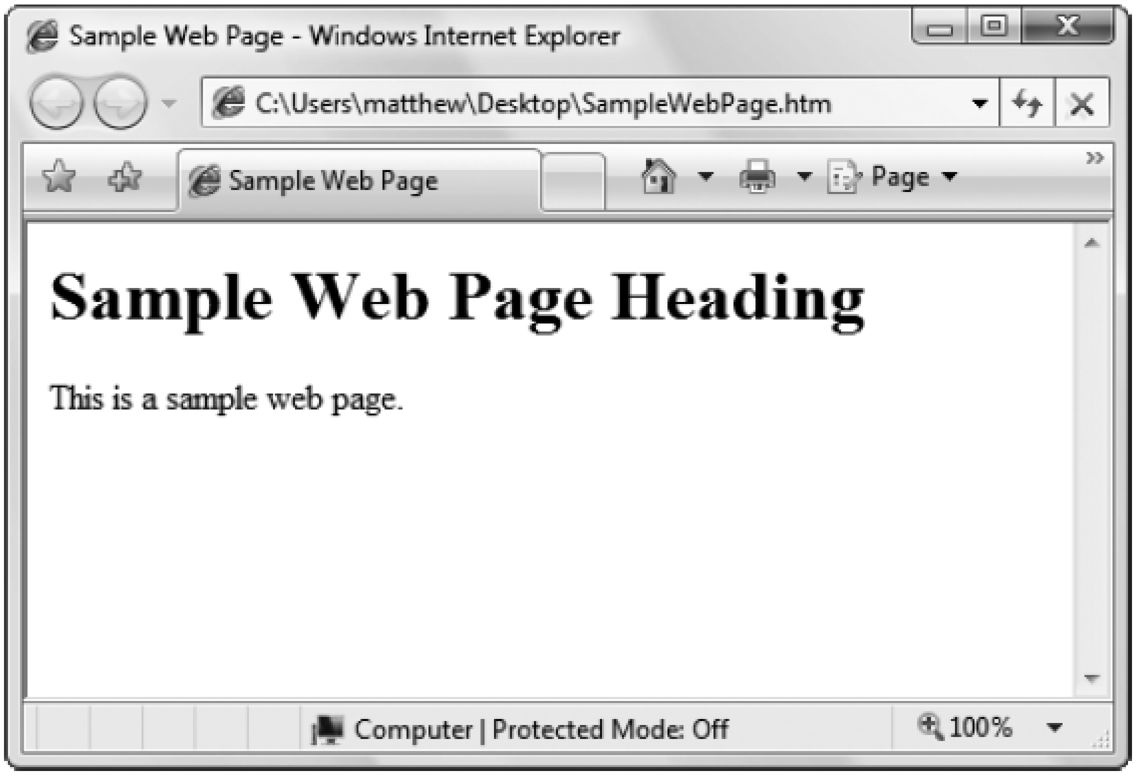
HTML and HTML Forms

It would be difficult to describe early websites as web *applications*. Instead, the first generation of websites often looked more like brochures, consisting mostly of fixed HTML pages that needed to be updated by hand.
A basic HTML page is a little like a word-processing document—it contains formatted content that can be displayed on your computer, but it doesn't actually *do* anything. The following example shows HTML at its simplest, with a document that contains a heading and single line of text:

```
<html>
<head>
<title>Sample Web Page</title>
</head>
<body>
<h1>Sample Web Page Heading</h1>
<p>This is a sample web page.</p>
</body>
</html>
```

An HTML document has two types of content: the text and the elements (or tags) that tell the browser how to format it. The elements are easily recognizable, because they are designated with angled brackets (< >). HTML defines elements for different levels of headings, paragraphs, hyperlinks, italic and bold formatting, horizontal lines, and so on. For example, <h1>Some

Text</h1> uses the <h1> element. This element tells the browser to display *Some Text* in the Heading 1 style, which uses a large, bold font. Similarly, <p>This is a sample web page.</p> creates a paragraph with one line of text. The <head> element groups the header information together, including the title that appears in the browser window, while the <body> element groups together the actual document content that's displayed in the browser window. Figure 1-1 shows this simple HTML page in a browser. Right now, this is just a fixed file (named sample_web_page_heading.htm) that contains HTML content. It has no interactivity, doesn't require a web server, and certainly can't be considered a web application.


Sample Web Page - Windows Internet Explorer
C:\Users\matthew\Desktop\SampleWebPage.htm
Sample Web Page
Page
Computer | Protected Mode: Off
100%

Sample Web Page Heading

This is a sample web page.

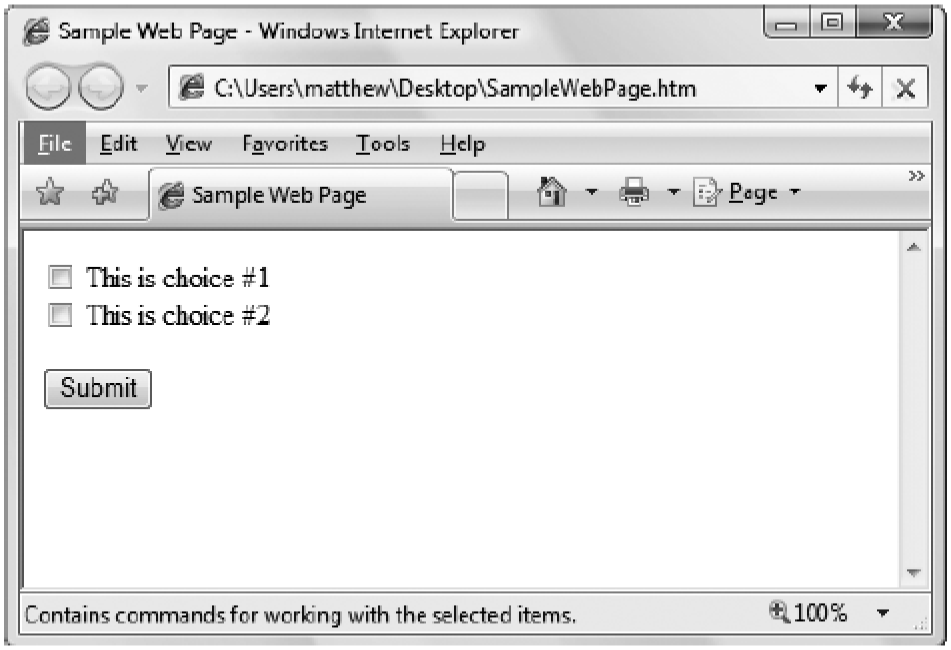
Figure 1-1. *Ordinary HTML: the "brochure" site*

HTML 2.0 introduced the first seed of web programming with a technology called *HTML forms*. HTML forms expand HTML so that it includes not only formatting tags but also tags for graphical widgets, or *controls*. These controls include common ingredients such as drop-down lists, text boxes, and buttons. Here's a sample web page created with HTML form controls:

```
<html>
<head>
<title>Sample Web Page</title>
</head>
<body>
<form>
<input type="checkbox" />
This is choice #1<br />
<input type="checkbox" />
```

```
This is choice #2<br /><br />
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

In an HTML form, all controls are placed between the <form> and </form> tags. The preceding example includes two check boxes (represented by the <input type="checkbox" /> element) and a button (represented by the <input type="submit" /> element). The <br /> element adds a line break in between lines. In a browser, this page looks like Figure 1-2.


Sample Web Page - Windows Internet Explorer
C:\Users\matthew\Desktop\SampleWebPage.htm
File Edit View Favorites Tools Help
Sample Web Page
Page
This is choice #1
This is choice #2
Submit
100%

Contains commands for working with the selected items.

Figure 1-2. *An HTML form*

HTML forms allow web application developers to design standard input pages. When the user clicks the Submit button on the page shown in Figure 1-2, all the data in the input controls (in this case, the two check boxes) is patched together into one long string of text and sent to the web server. On the server side, a custom application receives and processes the data.

Amazingly enough, the controls that were created for HTML forms more than ten years ago are still the basic foundation that you'll use to build dynamic ASP.NET pages! The difference is the type of application that runs on the server side. In the past, when the user clicked a button on a form page, the information might have been e-mailed to a set account or sent to an application on the server that used the challenging Common Gateway Interface (CGI) standard. Today, you'll work with the much more capable and elegant ASP.NET platform.

Server-Side Programming

To understand why ASP.NET was created, it helps to understand the problems of early web development technologies. With the original CGI standard, for example, the web server must launch a completely separate instance of the application for each web request. If the website is popular, the web server struggles under the weight of hundreds of separate copies of the application, eventually becoming a victim of its own success. Furthermore, technologies such as CGI provide a bare-bones programming environment. If you want higher-level features, like the ability to authenticate users, store personalized information, or display records you've retrieved from a database, you need to write pages of code from scratch. Building a web application this way is tedious and error-prone.

To counter these problems, Microsoft created higher-level development platforms, such as ASP and ASP.NET. Both of these technologies allow developers to program dynamic web pages without worrying about the low-level implementation details. For that reason, both platforms have been incredibly successful.

The original ASP platform garnered a huge audience of nearly one million developers, becoming far more popular than even Microsoft anticipated. It wasn't long before it was being wedged into all sorts of unusual places, including mission-critical business applications and highly trafficked e-commerce sites. Because ASP wasn't designed with these uses in mind, performance, security, and configuration problems soon appeared.

That's where ASP.NET comes into the picture. ASP.NET was developed as an industrial strength web application framework that could address the limitations of ASP. Compared to classic ASP, ASP.NET offers better performance, better design tools, and a rich set of readymade features. ASP.NET was wildly popular from the moment it was released—in fact, it was put to work in dozens of large-scale commercial websites while still in beta form.

Client-Side Programming

At the same time that server-side web development was moving through an alphabet soup of technologies, a new type of programming was gaining popularity. Developers began to experiment with the different ways they could enhance web pages by embedding miniature applets built with JavaScript, ActiveX, Java, and Flash into web pages. These client-side technologies don't involve any server processing. Instead, the complete application is downloaded to the client browser, which executes it locally.

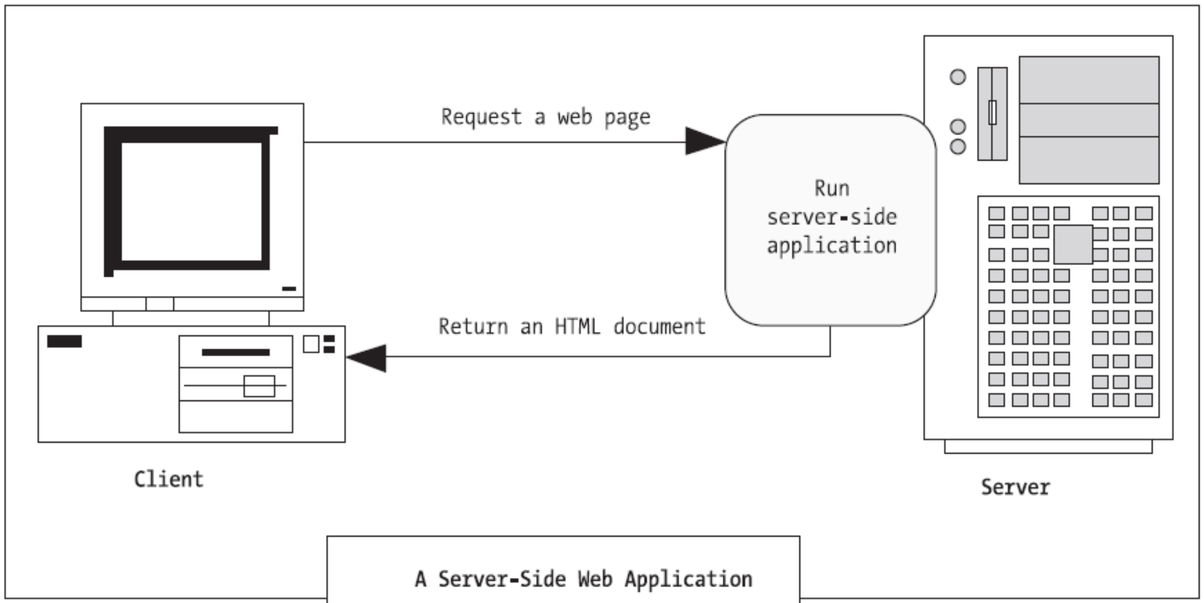
The greatest problem with client-side technologies is that they aren't supported equally by all browsers and operating systems. One of the reasons that web development is so popular in the first place is because web applications don't require setup CDs, downloads, and other tedious (and error-prone) deployment steps. Instead, a web application can be used on any computer that has Internet access. But when developers use client-side technologies, they encounter a few familiar headaches. Suddenly, cross-browser compatibility becomes a problem. Developers are forced to test their websites with different operating systems and browsers, and they might even need to distribute browser updates to their clients. In other words, the client-side model sacrifices some of the most important benefits of web development.

For that reason, ASP.NET is designed as a server-side technology. All ASP.NET code executes on the server. When the code is finished executing, the user receives an ordinary HTML page, which can be viewed in any browser. Figure 1-3 shows the difference between the server-side and the client-side model.

These are some other reasons for avoiding client-side programming:

***Isolation*:** Client-side code can't access server-side resources. For example, a client-side application has no easy way to read a file or interact with a database on the server (at least not without running into problems with security and browser compatibility).

***Security*:** End users can view client-side code. And once malicious users understand how an application works, they can often tamper with it.
***Thin clients*:** As the Internet continues to evolve, web-enabled devices such as mobile phones, palmtop computers, and PDAs (personal digital assistants) are appearing. These devices can communicate with web servers, but they don't support all the features of a traditional browser. Thin clients can use server-based web applications, but they won't support client-side features such as JavaScript.
However, client-side programming isn't truly dead. In many cases, ASP.NET allows you to combine the best of client-side programming with server-side programming. For example, the best ASP.NET controls can intelligently detect the features of the client browser. If the browser Supports JavaScript, these controls will return a web page that incorporates JavaScript for a richer, more responsive user interface.
However, no matter what the capabilities of the browser, *your* code is always executed on the Server. The client-side frills are just the icing on the cake.



A Server-Side Web Application

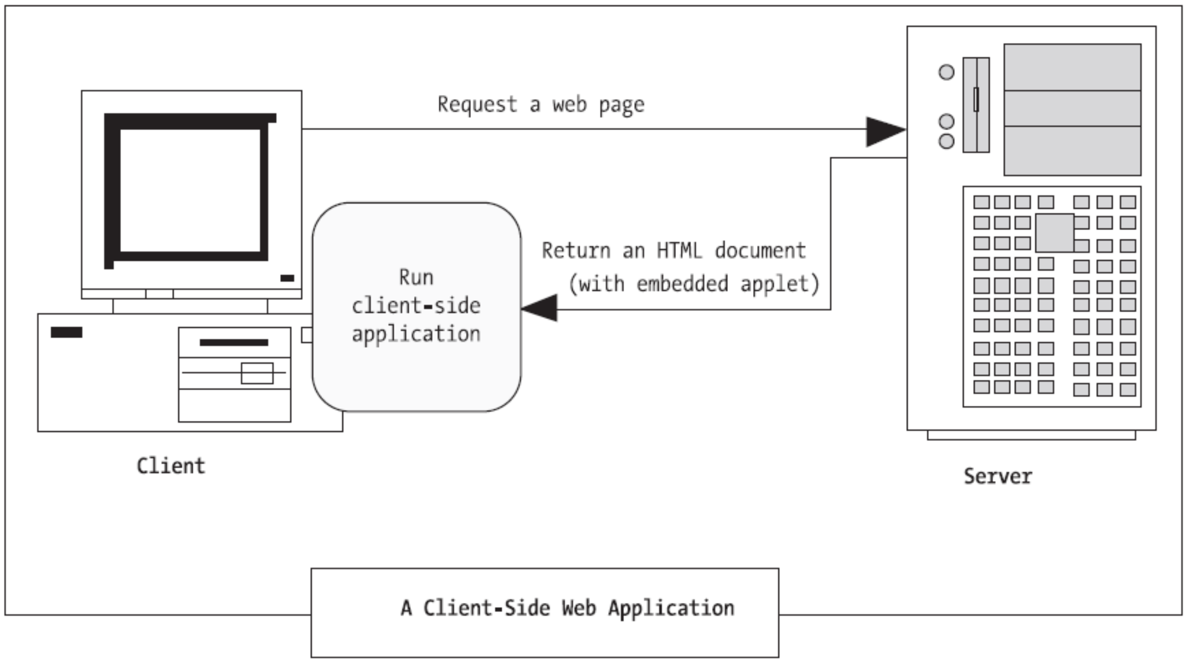

Request a web page
Run client-side application
Return an HTML document (with embedded applet)
Client
Server
A Client-Side Web Application

Figure 1-3. *Server-side and client-side web applications*

The .NET Framework

As you've already learned, the .NET Framework is really a cluster of several technologies. These include the following:

The .NET languages**:** These include Visual Basic, C#, JScript .NET (a server-side version of JavaScript), J# (a Java clone), and C++.

The Common Language Runtime (CLR)**:** This is the engine that executes all .NET programs and provides automatic services for these applications, such as security checking, memory management, and optimization.

The .NET Framework class library**:** The class library collects thousands of pieces of prebuilt functionality that you can "snap in" to your applications. These features are sometimes organized into technology sets, such as ADO.NET (the technology for creating database applications) and Windows Forms (the technology for creating desktop user interfaces).

ASP.NET**:** This is the engine that hosts the web applications you create with .NET, and supports almost any feature from the .NET class library. ASP.NET also includes a set of web-specific services, like secure authentication and data storage.

Visual Studio**:** This optional development tool contains a rich set of productivity and debugging features. The Visual Studio setup DVD includes the complete .NET Framework, so you won't need to download it separately.

Sometimes the division between these components isn't clear. For example, the term *ASP.NET* is sometimes used in a narrow sense to refer to the portion of the .NET class library used to design web pages. On the other hand, ASP.NET also refers to the whole topic of .NET web applications, which includes .NET languages and many fundamental pieces of the class library that aren't web-specific. (That's generally the way we use the term in this book. Our exhaustive examination of ASP.NET includes .NET basics, the C# language, and topics that any .NET developer could use, such as component-based programming and database access.)

Figure 1-4 shows the .NET class library and CLR—the two fundamental parts of .NET.

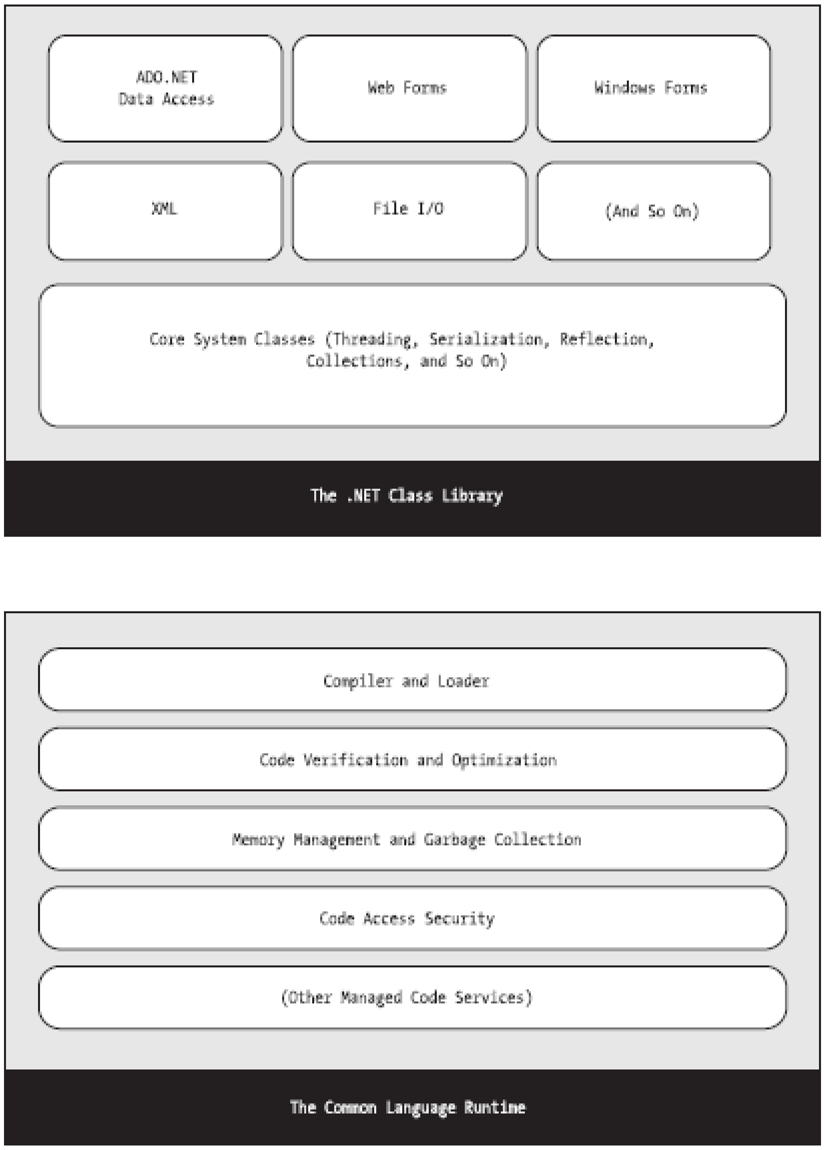

ADO.NET Data Access
Web Forms
Windows Forms
XML
File I/O
(And So On)
Core System Classes (Threading, Serialization, Reflection, Collections, and So On)
The .NET Class Library
Compiler and Loader
Code Verification and Optimization
Memory Management and Garbage Collection
Code Access Security
(Other Managed Code Services)
The Common Language Runtime

Figure 1-4. *The .NET Framework*

C#,VB, and the .NET Languages

This book uses C#, Microsoft's .NET language of preference. C# is a new language that was designed for .NET 1.0. It resembles Java and C++ in syntax, but no direct migration path exists from Java or C++.

Interestingly, VB and C# are actually quite similar. Though the syntax is different, both VB and C# use the .NET class library and are supported by the CLR. In fact, almost any block of C# code can be translated, line by line, into an equivalent block of VB code (and vice versa). An occasional language difference pops up (for example, VB supports a language feature called *optional parameters*, while C# doesn't), but for the most part, a developer who has learned one .NET language can move quickly and efficiently to another.

In short, both VB and C# are elegant, modern languages that are ideal for creating the next Generation of web applications.

Intermediate Language

All the .NET languages are compiled into another lower-level language before the code is executed.

This lower-level language is the Common Intermediate Language (CIL, or just IL). The CLR, the engine of .NET, uses only IL code. Because all .NET languages are designed based on IL, they all have profound similarities. This is the reason that the VB and C# languages provide essentially the same features and performance. In fact, the languages are so compatible that a
web page written with C# can use a VB component in the same way it uses a C# component,
and vice versa.

The .NET Framework formalizes this compatibility with something called the Common Language Specification (CLS). Essentially, the CLS is a contract that, if respected, guarantees
that a component written in one .NET language can be used in all the others. One part of the CLS is the common type system (CTS), which defines the rules for data types such as strings, numbers, and arrays that are shared in all .NET languages. The CLS also defines object-oriented
ingredients such as classes, methods, events, and quite a bit more. For the most part, .NET developers don’t need to think about how the CLS works, even though they rely on it every day.

Figure 1-5 shows how the .NET languages are compiled to IL. Every EXE or DLL file that
you build with a .NET language contains IL code. This is the file you deploy to other computers.
In the case of a web application, you deploy your compiled code to a live web server.

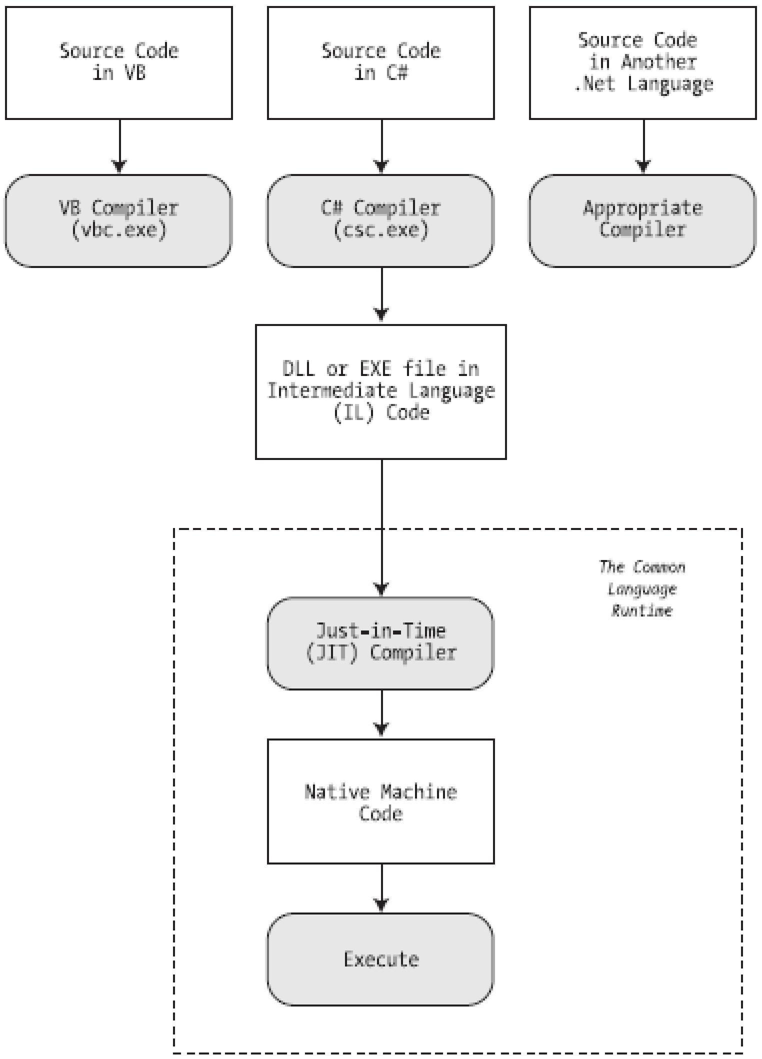

Source Code in VB
Source Code in C#
Source Code in Another .Net Language
VB Compiler (vbc.exe)
C# Compiler (csc.exe)
Appropriate Compiler
DLL or EXE file in Intermediate Language (IL) Code
The Common Language Runtime
Just-in-Time (JIT) Compiler
Native Machine Code
Execute

Figure 1-5. *Language compilation in .NET*

The CLR runs only IL code, which means it has no idea which .NET language you originally used. Notice, however, that the CLR actually performs another compilation step—it takes the IL code and transforms it to native machine language code that's appropriate for the current platform. This step occurs when the application is launched, just before the code is actually executed. In an ASP.NET application, these machine-specific files are cached while the web application is running so they can be reused, ensuring optimum performance. Other .NET Languages
VB and C# aren't the only choices for ASP.NET development. Developers can also use J# (a language with Java-like syntax). You can even use a .NET language provided by a third-party developer, such as a .NET version of Eiffel or even COBOL. This increasing range of language choices is possible thanks to the CLS and CTS, which define basic requirements and standards that allow other companies to write languages that can be compiled to IL. Although you can use any .NET language to create an ASP.NET web application, some of them do not provide the same level of design support in Visual Studio, and virtually all ASP.NET developers use VB and C#.

The Common Language Runtime

The CLR is the engine that supports all the .NET languages. Many modern languages use runtimes. In VB 6, the runtime logic is contained in a DLL file named msvbvm60.dll. In C++, many applications link to a file named mscrt40.dll to gain common functionality. These runtimes may provide libraries used by the language, or they may have the additional responsibility of executing the code (as with Java).

Runtimes are nothing new, but the CLR is Microsoft's most ambitious runtime to date. Not only does the CLR execute code, it also provides a whole set of related services such as code verification, optimization, and object management.

All .NET code runs inside the CLR. This is true whether you're running a Windows application or a web service. For example, when a client requests an ASP.NET web page, the ASP.NET service runs inside the CLR environment, executes your code, and creates a final HTML page to send to the client.

The implications of the CLR are wide-ranging:

***Deep language integration*:** VB and C#, like all .NET languages, compile to IL. In other words, the CLR makes no distinction between different languages—in fact, it has no way of knowing what language was used to create an executable. This is far more than mere language compatibility; it's language *integration*.

***Side-by-side execution*:** The CLR also has the ability to load more than one version of a component at a time. In other words, you can update a component many times, and the correct version will be loaded and used for each application. As a side effect, multiple versions of the .NET Framework can be installed, meaning that you're able to upgrade to new versions of ASP.NET without replacing the current version or needing to rewrite your applications.

***Fewer errors*:** Whole categories of errors are impossible with the CLR. For example, the CLR prevents many memory mistakes that are possible with lower-level languages such as C++. Along with these truly revolutionary benefits, the CLR has some potential drawbacks. Here are three issues that are often raised by new developers but aren't always answered:

***Performance*:** A typical ASP.NET application is much faster than a comparable ASP application, because ASP.NET code is compiled to machine code before it's executed. However, processor-crunching algorithms still can't match the blinding speed of well-written C++ code, because the CLR imposes some additional overhead. Generally, this is a factor only in a few performance-critical high-workload applications (such as real-time games). With high-volume web applications, the potential bottlenecks are rarely processor-related but are usually tied to the speed of an external resource such as a database or the web server's file system. With ASP.NET caching and some well-written database code, you can ensure excellent performance for any web application.

***Code transparency*:** IL is much easier to disassemble, meaning that if you distribute a compiled application or component, other programmers may have an easier time determining how your code works. This isn't much of an issue for ASP.NET applications, which aren't distributed but are hosted on a secure web server.

***Questionable cross-platform support*:** No one is entirely sure whether .NET will ever be adopted for use on other operating systems and platforms. Ambitious projects such as Mono (a free implementation of .NET on Linux, Unix, and Windows) are currently underway (see www.mono-project.com). However, .NET will probably never have the wide reach of a language such as Java because it incorporates too many different platform-specific and operating system–specific technologies and features.

The .NET Class Library
The .NET class library is a giant repository of classes that provide prefabricated functionality for everything from reading an XML file to sending an e-mail message. If you've had any exposure to Java, you may already be familiar with the idea of a class library. However, the .NET class library is more ambitious and comprehensive than just about any other programming framework. Any .NET language can use the .NET class library's features by interacting with the right objects. This helps encourage consistency among different .NET languages and removes the need to install numerous components on your computer or web server. Some parts of the class library include features you'll never need to use in web applications (such as the classes used to create desktop applications with the Windows interface).
Other parts of the class library are targeted directly at web development. Still more classes can be used in various programming scenarios and aren't specific to web or Windows development.
These include the base set of classes that define common variable types and the classes for data access, to name just a few. You'll explore the .NET Framework throughout this book. You can think of the class library as a well-stocked programmer's toolkit. Microsoft's philosophy is that it will provide the tedious infrastructure so that application developers need only to write business-specific code. For example, the .NET Framework deals with thorny issues like database transactions and concurrency, making sure that hundreds or thousands of simultaneous users can request the same web page at once. You just add the logic needed for your specific application.
Visual Studio
The last part of .NET is the Visual Studio development tool, which provides a rich environment where you can rapidly create advanced applications. Although in theory you could create an ASP.NET application without Visual Studio (for example, by writing all the source code in a text editor and compiling it with .NET's command-line compilers), this task would be tedious, painful, and prone to error. For that reason, all professional ASP.NET developers use a design tool like Visual Studio.
Some of the features of Visual Studio include the following:
Page design**:** You can create an attractive page with drag-and-drop ease using Visual Studio's integrated web form designer. You don't need to understand HTML.
Automatic error detection**:** You could save hours of work when Visual Studio detects and reports an error before you run your application. Potential problems are underlined, just like the "spell-as-you-go" feature found in many word processors.

Debugging tools**:** Visual Studio retains its legendary debugging tools, which allow you to watch your code in action and track the contents of variables. And you can test web applications just as easily as any other application type, because Visual Studio has a built-in web server that works just for debugging.

IntelliSense**:** Visual Studio provides statement completion for recognized objects and automatically lists information such as function parameters in helpful tooltips. You don't need to use Visual Studio to create web applications. In fact, you might be tempted to use the freely downloadable .NET Framework and a simple text editor to create ASP.NET web pages and web services. However, in doing so you'll multiply your work, and you'll have a much harder time debugging, organizing, and maintaining your code. Visual Studio is available in several editions. The Standard Edition has all the features you need to build any type of application (Windows or web). The Professional Edition and the Team Edition increase the cost and pile on more tools and frills (which aren’t discussed in this book). For example, they incorporate features for managing source code that’s edited by multiple people on a development team and running automated tests.

The scaled-down Visual Web Developer Express Edition is a completely free version of Visual Studio that’s surprising capable, but it has a few significant limitations. Visual Web Developer Express Edition gives you full support for developing web applications, but it doesn’t support any other type of application. This means you can’t use it to develop separate components for use in your applications or to develop Windows applications. However, rest assured that Visual Web Developer Express Edition is still a bona fide version of Visual Studio, with a similar set of features and development interface.