Covers MSDE/SQL and Access!

sitepoint

Build Your Own ASP.NET Website Using C# & VB.NET

A Practical Step-by-Step Guide

By Zak Ruvalcaba

Build Your Own ASP.NET Website Using C# & VB.NET (Chapters 1, 2, 3 and 4)

Thank you for downloading the sample chapters of Zak Ruvalcaba's book, *Build Your Own ASP.NET Website Using C# & VB.NET*, published by SitePoint.

This excerpt includes the Summary of Contents, Information about the Author, Editors and SitePoint, Table of Contents, Preface, 4 chapters of the book and the index.

We hope you find this information useful in evaluating this book.

Summary of Contents of this Excerpt

Summary of Additional Book Contents

Build Your Own ASP.NET Website Using C# & VB.NET

by Zak Ruvalcaba

Build Your Own ASP.NET Website Using C# & VB.NET

by Zak Ruvalcaba

Editor: Georgina Laidlaw
Managing Editor: Simon Mackie
Cover Design: Julian Carroll
Printing History:
First Edition: April 2004

Expert Reviewer: Kevin Yank
Technical Editor: Rich Deeson
Index Editor: Bill Johncocks

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

424 Smith Street Collingwood
VIC Australia 3066.

Web: www.sitepoint.com
Email: business@sitepoint.com

ISBN 0–9579218–6–1
Printed and bound in the United States of America

About The Author

Zak Ruvalcaba has been designing, developing and researching for the Web since 1995. He holds a Bachelor's Degree from San Diego State University and a Master of Science in Instructional Technology from National University in San Diego.

In the course of his career, Zak has developed Web applications for such companies as Gateway, HP, Toshiba, and IBM. More recently, he's worked as a wireless software engineer developing .NET solutions for Goldman Sachs, TV Guide, The Gartner Group, Microsoft and Qualcomm. Currently, Zak holds a programming position with ADCS Inc. in San Diego supporting internal .NET applications.

Previous books by Zak Ruvalcaba include *The 10 Minute Guide to Dreamweaver 4* (Que Publishing) and *Dreamweaver MX Unleashed* (Sams Publishing). He also lectures on various technologies and tools including Dreamweaver and ASP.NET for the San Diego Community College District.

About The Expert Reviewer

As Technical Director for SitePoint, Kevin Yank oversees all of its technical publications—books, articles, newsletters and blogs. He has written over 50 articles for SitePoint on technologies including PHP, XML, ASP.NET, Java, JavaScript and CSS, but is perhaps best known for his book, *Build Your Own Database Driven Website Using PHP & MySQL*, also from SitePoint.

Having graduated from McGill University in Montreal with a Bachelor of Computer Engineering, Kevin now lives in Melbourne, Australia. In his spare time he enjoys flying light aircraft and learning the fine art of improvised acting. Go you big red fire engine!

About The Technical Editor

Rich Deeson wrote his first programs at the age of 10 on his father's work machine, a 380Z with 256k RAM. Since then, his career has taken him around Europe, and has taught him the ins and outs of many languages, from C++ to Java, from QuickBasic (the precursor to Visual Basic) to VB.NET, from Perl and CGI to JSP and ASP.NET. Currently, he is lead JSP developer at ICTI in the UK, and most of his free time is taken up at University, having returned to study last year.

About SitePoint

SitePoint specializes in publishing fun, practical and easy-to-understand content for Web Professionals. Visit http://www.sitepoint.com/ to access our books, newsletters, articles and community forums.

For my wife Jessica.

Table of Contents

Preface

Here I am, seven years after the inception of ASP, still using a technology that I initially only glanced over as I searched for a server-side alternative to ColdFusion.

It was 1997, a big year for me. I graduated college, landed a job as a creative director, and decided it was time to build on my experience with HTML and JavaScript. I didn't consider myself a programmer—my true passions lay in design—but within months of starting my new job, I was developing the firm's Website, Intranet, and company portal. The dynamic portions of these projects were developed using CGI written in Perl. As you might expect, I was lost! After looking around, I decided ColdFusion was my best bet—the language seemed to parallel closely the constructs of HTML, and I found it easy to pick up. However, I soon discovered that ColdFusion's limitations in terms of accessing a server's file system, and error handling, posed problems.

ASP and VBScript seemed like the best alternative. I'd taken basic programming classes in college, and I guess they helped, because these two technologies came easily to me. Shortly thereafter, I went back to school and got into Visual Basic, COM, DCOM, and more. A whole new world was opening up to me through simplicity offered by ASP.

Seven years, and countless Windows, Web, and wireless applications later, I still swear by the next generation of a technology that I've always considered superior to the major alternatives. ASP.NET represents a new and efficient way of creating Web applications using the programming language with which you feel most comfortable. Though it can take some time to learn, ASP.NET is simple to use. Whether you want to create Web Forms complete with Web and validation controls, or you aim to build a feature-rich shopping cart using `DataTable`s, all the tools you'll need to get up and running are immediately available, easy to install, and require very little initial configuration.

My guess is that if you're reading this book, you're in the same boat I was: a longtime designer dabbling with HTML. Or maybe you're an advanced HTML and JavaScript developer looking to take the next step. Perhaps you're a seasoned PHP, JSP, or ColdFusion veteran who wants to know what all the fuss is about. Whatever the case, I'm sure you'll find this book helpful in showing you how simple and feature-rich ASP.NET really is.

Who Should Read This Book?

This book is aimed at beginner, intermediate, and advanced Web designers looking to make the leap into server-side programming with ASP.NET. You'll be expected to feel comfortable with HTML, as very little explanation is provided here.

By the end of this book, you should have a firm grasp on what it takes to download and install ASP.NET and the .NET Framework, configure and start your Web server, create and work with basic ASP.NET pages, install and run either Access or MSDE, create database tables, work with advanced, dynamic ASP.NET pages that query, insert, update, and delete information within a database.

All examples provided in the book are written in both Visual Basic .NET and C#, the two most popular languages for writing ASP.NET Websites. They start at beginners' level and work up. As such, no prior knowledge of the two languages is required in order to read, learn from, and apply the knowledge provided in this book. Experience with other programming or scripting languages (such as JavaScript) will certainly grease the wheels, however, and will enable you to grasp the fundamental programming concepts more quickly.

What's Covered In This Book?

This book is comprised of the following seventeen chapters. Read them from beginning to end to gain a complete understanding of the subject, or skip around if you feel you need a refresher on a particular topic.

Chapter 1: ***Introduction to .NET and ASP.NET***

Before you can start building your database-driven Web presence, you must ensure you have the right tools for the job. In this first chapter, I'll tell you how to find, download, and configure the .NET Framework. I'll explain where the Web server is located and how to install and configure it. Next, we'll walk through the installation of two Microsoft database solutions: Access and MSDE. Finally, we'll create a simple ASP.NET page to make sure that everything's running and properly configured.

Chapter 2: ***ASP.NET Basics***

In this chapter, you'll create your first useful ASP.NET page. We'll cover all of the parts that make up a typical ASP.NET page, including directives, controls, and code. We'll then walk through the process of deployment, fo-

cusing specifically on allowing the user to view the processing of a simple ASP.NET page through the Web browser.

Chapter 3: ***VB.NET and C# Programming Basics***

In this chapter, we'll look at two of the programming languages used to create ASP.NET pages: VB.NET and C#. You'll learn about the syntax of the two languages as we explore the concepts of variables, data types, conditionals, loops, arrays, functions, and more. Finally, we'll see how the two languages accommodate Object Oriented Programming principles by allowing you to work with classes, methods, properties, inheritance, and more.

Chapter 4: ***Web Forms and Web Controls***

ASP.NET is bundled with hundreds of controls that you can use within your applications, including HTML controls, Web controls, and more. This chapter will introduce you to the wonderful world of Web controls and how Microsoft basically reinvented HTML forms.

Chapter 5: ***Validation Controls***

This chapter introduces validation controls. With validation controls, Microsoft basically eliminated the heartache of fumbling through and configuring tired, reused client-side validation scripts.

Chapter 6: ***Database Design and Development***

Undoubtedly one of the most important chapters in the book, Chapter 6 will help you prepare to work with databases in ASP.NET. We'll cover the essentials you'll need in order to create a database using either Access or MSDE. In this chapter, we'll begin to build the database for our project.

Chapter 7: ***Structured Query Language***

This chapter introduces the language we'll use to facilitate communications between the database and the Web application: Structured Query Language, or SQL. After a gentle introduction to the basic concepts of SQL, we'll move on to more advanced topics such as expressions, conditions, and joins.

Chapter 8: ***ADO.NET***

The next logical step in database driven Web applications involves ADO.NET. This chapter explores the essentials of the technology, and will have you reading data in a database directly from your Web applications in just a few short steps. We'll then help you begin the transition from working with static applications to database-driven ones.

Chapter 9: *The `DataGrid` and `DataList` Controls*

Taking ADO.NET further, this chapter shows you how to utilize the `DataGrid` and `DataList` controls provided within the .NET Framework. `DataGrid` and `DataList` play a crucial role in the simplicity of presenting information with ASP.NET. In learning how to present database data within your applications in a cleaner and more legible format, you'll gain an understanding of the concept of data binding at a much higher level.

Chapter 10: *`DataSets`*

One of the most challenging concepts to grasp when transitioning from ASP to ASP.NET is that of disconnected data. In this chapter, you'll learn how to use `DataSets` to create virtual database tables within your Web applications. You'll also learn how to work with `DataTables`, and how to filter and sort information within `DataSets` and `DataTables` using `DataViews`.

Chapter 11: *Web Applications*

Chapter 11 explores the features of a Web application. We'll discuss the many parts of the `Web.config` file in depth, and understand how to work with the `Global.asax` file, application state, and session state. Finally, we'll look at the ways in which caching can improve the performance of your Web applications.

Chapter 12: *Building an ASP.NET Shopping Cart*

In this chapter, we'll create an ASP.NET shopping cart. Using the topics we've explored in previous chapters, including `DataTables` and session state, we'll walk through the process of building a purely memory-resident shopping cart for our project.

Chapter 13: *Error Handling*

Learning to handle gracefully unforeseen errors within your Web applications is the topic of this chapter. Initially, we'll discuss basic page and code techniques you can use to handle errors. We'll then talk about the debugger that's included with the .NET Framework SDK and understand how to leverage it by setting breakpoints, reading the autos and locals window, and setting watches. Finally, we'll discuss how you can take advantage of the Event Viewer to write errors as they occur within your applications.

Chapter 14: *Security and User Authentication*

This chapter will introduce you to securing your Web applications with ASP.NET. Here, we'll discuss the various security models available, including IIS, Forms, Windows, and Passport, and discusses the roles the `Web.config` and XML files can play.

Chapter 15: ***Working with Files and Email***

In this chapter, we'll look at accessing your server's file system, including drives, files, and the network. The chapter will then show you how to work with file streams to create text files, write to text files, and read from text files on your Web server. Finally, you'll learn how to send emails using ASP.NET.

Chapter 16: ***Rich Controls and User Controls***

Chapter 16 explores ASP.NET's rich controls. You'll learn how to create an interactive meeting scheduler using the `Calendar` control, sessions, and serialization. You'll also learn how to format XML with XSLT utilizing the `Xml` control. Lastly, we'll look at randomizing banner advertisements on your site using the `AdRotator` control.

Chapter 17: ***XML Web Services***

The newest buzzword in the development community is "Web Services," and this chapter hopes to shed some light on the topic. We first define Web Services before moving on to explain how they're used, where they can be found, and what WSDL and UDDI are. In this chapter, you'll create a couple of different Web Services from scratch, including one that queries your database to present information within a Web application. You'll also learn how to build a search application using the Google Search Web Service.

The Book's Website

Located at http://www.sitepoint.com/books/aspnet1/, the Website that supports this book will give you access to the following facilities:

The Code Archive

As you progress through this book, you'll note a number of references to the code archive. This is a downloadable ZIP archive that contains complete code for all the examples presented in the book.

The archive contains one folder for each chapter of the book. Each of these folders in turn contains `CS` and `VB` subfolders, which contain the C# and VB.NET versions of all the examples for that chapter, respectively. In later chapters, these files are further divided into two more subfolders: `Lessons` for standalone examples presented for a single chapter, and `Project` for files associated with the Dorknozzle Intranet Application, a larger-scale project that we'll work on throughout the book, which I'll introduce in Chapter 4.

Updates and Errata

No book is perfect, and we expect that watchful readers will be able to spot at least one or two mistakes before the end of this one. The Errata page on the book's Website will always have the latest information about known typographical and code errors, and necessary updates for new releases of ASP.NET and the various Web standards that apply.

The SitePoint Forums

If you'd like to communicate with me or anyone else on the SitePoint publishing team about this book, you should join SitePoint's online community[2]. The .NET forum[3] in particular can offer an abundance of information above and beyond the solutions in this book.

In fact, you should join that community even if you *don't* want to talk to us, because there are a lot of fun and experienced Web designers and developers hanging out there. It's a good way to learn new stuff, get questions answered in a hurry, and just have fun.

The SitePoint Newsletters

In addition to books like this one, SitePoint publishes free email newsletters including *The SitePoint Tribune* and *The SitePoint Tech Times*. In them, you'll read about the latest news, product releases, trends, tips, and techniques for all aspects of Web development. If nothing else, you'll get useful ASP.NET articles and tips, but if you're interested in learning other technologies, you'll find them especially valuable. Sign up to one or more SitePoint newsletters at http://www.sitepoint.com/newsletter/.

Your Feedback

If you can't find your answer through the forums, or if you wish to contact us for any other reason, the best place to write is `<books@sitepoint.com>`. We have a well-manned email support system set up to track your inquiries, and if our support staff members are unable to answer your question, they will send it

[2] http://www.sitepoint.com/forums/
[3] http://www.sitepoint.com/forums/forumdisplay.php?f=141

straight to me. Suggestions for improvements as well as notices of any mistakes you may find are especially welcome.

Acknowledgements

First and foremost, I'd like to thank the SitePoint team for doing such a great job in making this book possible, for being understanding as deadlines inevitably slipped past, and for the team's personal touch, which made it a pleasure to work on this project.

Particular thanks go to Simon Mackie, whose valuable insight and close cooperation throughout the process has tied up many loose ends and helped make this book both readable and accessible. Thanks again Simon for allowing me to write this book—I appreciate the patience and dedication that you've shown.

Finally, returning home, I'd like to thank my wife Jessica, whose patience, love and understanding throughout continue to amaze me.

1 Introduction to .NET and ASP.NET

It's being touted as the "next big thing." Microsoft has invested millions in marketing, advertising, and development to produce what it feels is the foundation of the future Internet. It's a corporate initiative, the strategy of which was deemed so important, that Bill Gates himself, Microsoft Chairman and CEO, decided to oversee personally its development. It is a technology that Microsoft claims will reinvent the way companies carry out business globally for years to come. In his opening speech at the Professional Developers' Conference (PDC) held in Orlando Florida in July of 2000, Gates stated that a transition of this magnitude only comes around once every five to six years. What is this show-stopping technology? It's .NET.

What is .NET?

.NET is the result of a complete make-over of Microsoft's software development products, and forms part of the company's new strategy for delivering software as a service. The key features that .NET offers include:

- **.NET Platform:** The .NET platform includes the .NET Framework and tools to build and operate services, clients, and so on. ASP.NET, the focus of this book, is a part of the .NET Framework.

- **.NET Products:** .NET products currently include MSN.NET, Office.NET, Visual Studio.NET, and Windows Server 2003, originally known as Windows .NET Server. This suite of extensively revised systems provides developers with a friendly, usable environment in which they may create applications with a range of programming languages including C++. NET, Visual Basic.NET, ASP.NET, and C#. Because all these products are built on top of .NET, they all share key components, and underneath their basic syntaxes you'll find they have much in common.

- **.NET My Services:** An initiative formerly known as "Hailstorm", .NET My Services is a set of XML Web Services[1] currently being provided by a host of partners, developers, and organizations that are hoping to build corporate services and applications for devices and applications, as well as the Internet. The collection of My Services currently extends to passport, messenger, contacts, email, calendars, profiles, lists, wallets, location, document stores, application settings, favorite Websites, devices owned, and preferences for receiving alerts.

The book focuses on one of the core components within the .NET Framework: **ASP.NET**.

What is ASP.NET?

For years now, Active Server Pages (ASP) has been arguably the leading choice for Web developers building dynamic Websites on Windows Web servers. ASP has gained popularity by offering the simplicity of flexible scripting via several languages. That, combined with the fact that it's built into every Microsoft Windows-based Web server, has made ASP a difficult act to follow.

Early in 2002, Microsoft released its new technology for Internet development. Originally called ASP+, it was finally released as ASP.NET, and represents a leap forward from ASP both in sophistication and productivity for the developer. It continues to offer flexibility in terms of the languages it supports, but instead of a range of simple scripting languages, developers can now choose between several fully-fledged programming languages. Development in ASP.NET requires not only an understanding of HTML and Web design, but also a firm grasp of the concepts of object-oriented programming and development.

In the next few sections, I'll introduce you to the basics of ASP.NET. I'll walk you through installing it on your Web server, and take you through a simple

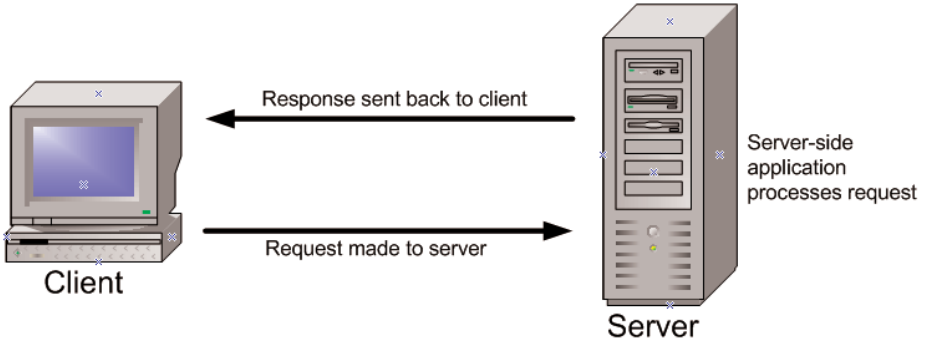
[1] Don't worry if you don't yet know what a Web Service is. I'll explain all about them in Chapter 17.

dynamic example that demonstrates how ASP.NET pages are constructed. First, let's define what ASP.NET actually is.

ASP.NET is a server-side technology for developing Web applications based on the Microsoft .NET Framework. Let's break that jargon-filled sentence down.

ASP.NET is server-side; that is, it runs on the Web server. Most Web designers start by learning client-side technologies like HTML, JavaScript, and Cascading Style Sheets (CSS). When a Web browser requests a Web page created with client-side technologies, the Web server simply grabs the files that the browser (the **client**) requests and sends them down the line. The client is entirely responsible for reading the code in the files and interpreting it to display the page on the screen. Server-side technologies, like ASP.NET, are different. Instead of being interpreted by the client, server-side code (for example, the code in an ASP.NET page) is interpreted by the Web server. In the case of ASP.NET, the code in the page is read by the server and used dynamically to generate standard HTML/JavaScript/CSS that is then sent to the browser. As all processing of ASP.NET code occurs on the server, it's called a server-side technology. As Figure 1.1 shows, the user (client) only sees the HTML, JavaScript, and CSS within the browser. The server (and server-side technology) is entirely responsible for processing the dynamic portions of the page.

Figure 1.1. The Web server is responsible for processing the server-side code and presenting the output to the user (client).



ASP.NET is a technology for developing Web applications. A Web application is just a fancy name for a dynamic Website. Web applications usually (but not always) store information in a database on the Web server, and allow visitors to

the site to access and change that information. Many different programming technologies and supported languages have been developed to create Web applications; PHP, JSP (using Java), CGI (using Perl), and ColdFusion (using CFML) are just a few of the more popular ones. Rather than tying you to a specific technology and language, however, ASP.NET lets you write Web applications using a variety of familiar programming languages.

Finally, **ASP.NET is based on the Microsoft .NET Framework.** The .NET Framework collects all the technologies needed for building Windows applications, Web applications, and Web Services into a single package with a set of more than twenty programming languages. To develop Websites with ASP.NET, you'll need to download the .NET Framework Software Development Kit, which I'll guide you through in the next few sections.

Even with all the jargon demystified, you're probably still wondering: what makes ASP.NET so good? Compared with other options for building Web applications, ASP.NET has the following advantages:

- ASP.NET lets you use your favorite programming language, or at least one that's really close to it. The .NET Framework currently supports over twenty languages, four of which may be used to build ASP.NET Websites.

- ASP.NET pages are **compiled**, not interpreted. Instead of reading and interpreting your code every time a dynamic page is requested, ASP.NET compiles dynamic pages into efficient binary files that the server can execute very quickly. This represents a big jump in performance when compared with the technology's interpreted predecessor, ASP.

- ASP.NET has full access to the functionality of the .NET Framework. Support for XML, Web Services, database interaction, email, regular expressions, and many other technologies are built right into .NET, which saves you from having to reinvent the wheel.

- ASP.NET allows you to separate the server-side code in your pages from the HTML layout. When you're working with a team composed of programmers and design specialists, this separation is a great help, as it lets programmers modify the server-side code without stepping on the designers' carefully crafted HTML—and vice versa.

With all these advantages, ASP.NET has relatively few downsides. In fact, only two come to mind:

- ASP.NET is a Microsoft technology. While this isn't a problem in itself, it does mean that, at least for now, you need to use a Windows server to run an ASP.NET Website. If your organization uses Linux or some other operating system for its Web servers, you're out of luck.
- Serious ASP.NET development requires an understanding of object-oriented programming, which we'll cover over the next few chapters.

Still with me? Great! It's time to gather the tools and start building!

What Do I Need?

For the moment, if you're going to learn ASP.NET, you'll need a Windows-based Web server. Open source initiatives are underway to produce versions of ASP.NET that will run on other operating systems, such as Linux; however, these are not expected to be available in stable form for a while.

While developers had the option of getting their feet wet with ASP on Windows 95, 98, or ME, using a scaled-down version of IIS called a Personal Web Server (PWS), ASP.NET requires the real deal. As a bare minimum, you'll need a computer equipped with Windows 2000 Professional before you can get started. Windows XP Professional will work fine too, as will any of the Windows 2000 Server packages and Windows 2003 Server.

Other than that, all you need is enough disk space to install the Web server **Internet Information Services** (18 MB), the .NET Framework SDK (which includes ASP.NET; 108 MB), and a text editor. Notepad or Web Matrix[1] will be fine for getting started, and are certainly all you'll need for this book. However, if you get serious about ASP.NET, you'll probably want to invest in a development environment like Visual Studio .NET[2].

Installing the Required Software

This section tackles the necessary installation and configuration of software that you'll need for this book, including:

- **Internet Information Services (IIS):** IIS is the Web server we will use. You'll need your copy of the Windows CD for the installation and configuration.

[1] http://www.asp.net/webmatrix/
[2] http://msdn.microsoft.com/vstudio/

- **A Modern Web Browser:** You can use any modern, standards-compliant browser to test your work. Throughout this book, we'll be using Internet Explorer 6.

- **The .NET Framework Redistributable:** As you've already learned in this chapter, the .NET Framework is what drives ASP.NET. Installing the .NET Framework installs the necessary files to run ASP.NET.

- **The .NET Framework SDK:** The .NET Framework Software Development Kit (SDK) contains necessary Web application development tools, a debugger for error correcting, a development database engine in MSDE, and a suite of samples and documentation.

We're also going to need a database. In this book, we'll use:

- **Microsoft Access:** Access is a cheap and easy-to-use alternative to its more robust big brother, SQL Server, and can be purchased separately, or installed from a Microsoft Office CD.

Or alternatively, you might use:

- **Microsoft SQL Server Desktop Engine (MSDE):** SQL Server is the enterprise alternative to smaller databases such as Access. If you're working within a corporation where your company's data is its lifeblood, then SQL Server is the perfect choice. MSDE is a free, cut down version of SQL Server that you can use for development purposes.

- **Web Data Administrator:** If you're going to use MSDE, then you'll need a tool for modifying the data within the database. Web Data Administrator is Microsoft's free Web-based database management tool.

Installing Internet Information Services (IIS)

Do you need to install IIS locally even if the final site will not be hosted locally? The answer is: yes. Even if you're uploading your Web applications via FTP to your Web host, installing IIS allows you to view, debug, and configure your applications locally before deployment.

IIS comes with most versions of server-capable Windows operating systems, including Windows 2000 Professional, Server, and Advanced Server, Windows XP Professional, and Windows Server 2003, but it's not installed automatically in all versions, which is why it may not be present on your computer. To see

whether you have IIS installed and running, simply navigate to your Administrative Tools menu and check to see if Internet Information Services is an option. Users of Windows 2000 Professional will find the Administrative Tools in their Control Panels, while XP and Server family users also have shortcuts in their start menus.

If the shortcut is not visible, then you don't have it installed. To install IIS, simply follow these steps:

1. In the Control Panel, select Add or Remove Programs.
2. Choose Add/Remove Windows Components. The list of components will become visible within a few seconds.
3. In the list of components, check Internet Information Services (IIS).
4. Click Next. Windows prompts you to insert the Windows CD and installs IIS.

Once IIS is installed, close the Add or Remove Programs dialog. You can check that IIS has installed correctly by seeing if you can find it within the Administrative Tools menu. If you can, it's installed.

You are now ready to begin hosting Web applications. Although we won't cover the configuration of IIS for external use, I will show you how to configure IIS to support local development of ASP.NET applications in order that they may be uploaded to your external Web hosting provider later.

Installing Internet Explorer

As a Windows user, you have Internet Explorer installed by default, but I recommend you run at least version 5.5. You can check your version by selecting About Internet Explorer from the Help menu.

If your version of Internet Explorer is earlier than 5.5, you can download the latest version (version 6 SP1 as of this writing) for free from the Internet Explorer Website[3]. Remember, although ASP.NET will work with older versions of IE, certain ASP.NET functionality works best with the latest version.

The Internet Explorer Website does not allow you to install a version of your choice; it permits you to download only the most recent version that's available.

[3] http://www.microsoft.com/windows/ie/

Because the newest versions of Internet Explorer will include the latest patches, it's a good idea to stick with what they give you.

Installing the .NET Framework and SDK

To begin creating ASP.NET applications, you'll need to install the .NET Framework and SDK. The .NET Framework includes the necessary files to run and view ASP.NET pages, while the .NET Framework SDK includes samples, documentation, and a variety of free tools.

The .NET Framework SDK also provides you with the ability to install MSDE, the free database server that you can use with this book. Once the .NET Framework and SDK are installed, little else needs to be done for you to begin working with ASP.NET. The .NET Framework is installed as part of the operating system if you're lucky enough to be running Windows .NET Server 2003, in which case you can skip directly to installing the SDK. If not, you will need to download the .NET redistributable package, which is approximately 21 MB, and includes the files necessary for running ASP.NET applications.

To develop .NET applications, you also need to install the software development kit, which includes necessary tools along with samples and documentation. Be aware that the .NET Framework SDK is 108 MB in size—be prepared to wait!

Installing the .NET Framework before you install IIS will prevent your applications from working correctly.

Download and Install the Redistributable

The best method of acquiring the .NET Framework is to download and install it directly from the Web. To accomplish this, simply follow the steps outlined below:

1. Go to the ASP.NET support site at http://www.asp.net/ and click the Download link.

2. Click the Download .NET Framework Redist Now link. Remember, we will install the redistributable first, then we will install the SDK. The link will advance you to a download page.

3. Choose the language version of the install you want, and click Download.

4. When prompted, save the file to a local directory by choosing Save.

5. After the download is complete, double-click the executable to begin the installation.

6. Follow the steps presented by the .NET Setup Wizard until installation completes.

Download and Install the SDK

Now that you've installed the redistributable, you need to install the software development kit (SDK):

1. Go to the ASP.NET support site at http://www.asp.net/ and click the Download link.

2. Click the Download .NET Framework SDK Now link. The link will advance you to a download page.

3. Choose the language version of the install you want to use and click Download, as you did to download the redistributable.

4. When prompted to do so, save the file to a local directory by choosing Save.

5. After the download is complete, double-click the executable to begin the installation. Before you do, I strongly recommend closing all other programs to ensure the install proceeds smoothly.

6. Follow the steps outlined by the .NET Setup Wizard until installation completes.

The SDK will take slightly longer to install than the redistributable. Once it's finished, check to see if it exists in your programs menu; navigate to Start > Programs > Microsoft .NET Framework SDK.

Configuring IIS

Although little configuration needs to be done before you begin working with IIS, I'll use this section to introduce some basic features and functionality within IIS:

- ❑ Determining whether ASP.NET installed correctly
- ❑ Determining where files are located on the Web server

- Using localhost
- How to start and stop the Web server
- How to create a new virtual directory and modify its properties

Determining whether ASP.NET Installed Correctly

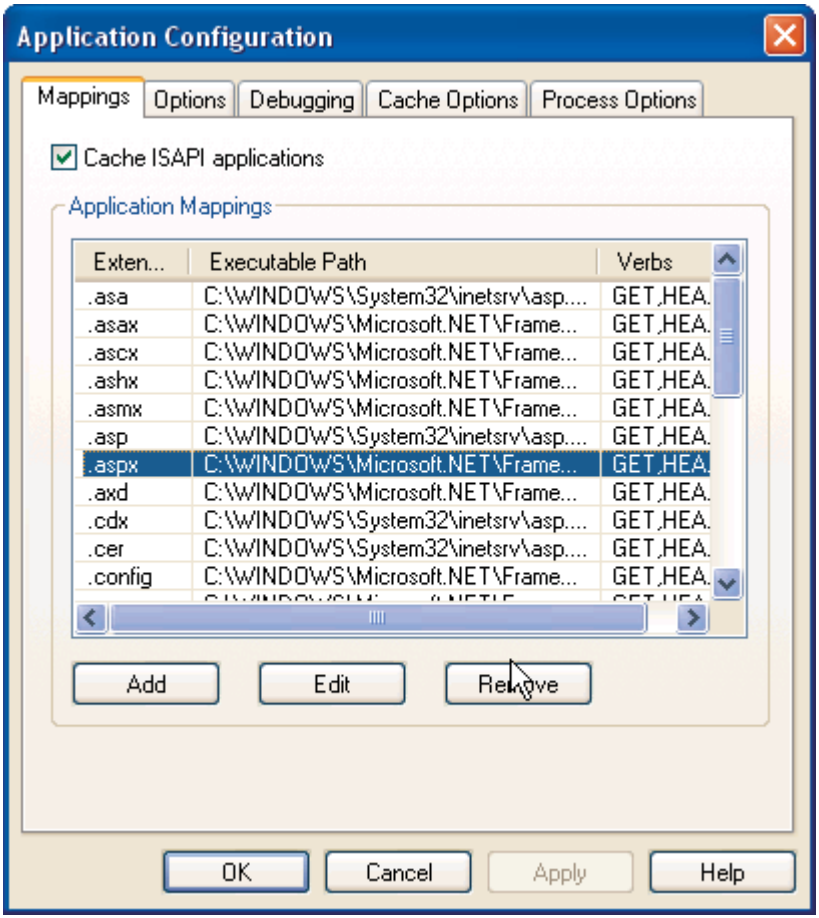
Once IIS is installed on your computer, you can open it by selecting Internet Information Services from the Administrative Tools menu. The first task is to make sure that ASP.NET was integrated into IIS when you installed the .NET Framework. Although, logically, ASP.NET should install automatically because it's a component of the .NET Framework, sometimes it doesn't. Don't let this alarm you—it's a common occurrence and is addressed in the Microsoft Knowledge Base. You can determine whether IIS was installed correctly by following these steps:

1. Open IIS, if you haven't already done so, and click on the + symbol next to your computer's name.
2. Right-click Default Web Site and select Properties.
3. Navigate to the Documents tab. If `default.aspx` appears within the list, ASP.NET was installed correctly.

Another way to check whether ASP.NET installed correctly is by following these steps:

1. Navigate to the Application Mappings menu by right-clicking the root Website node (your computer's name) and choosing Properties.
2. Select the Home Directory tab, and choose Configuration.
3. The Application Mappings menu displays all of the extensions and their associated ISAPI Extension DLLs, as we see in Figure 1.2.

Figure 1.2. If the `.aspx` ISAPI Extension DLL appears within the Application Mappings menu, then ASP.NET was installed correctly.



Since I can imagine you're dying to know what an ISAPI Extension DLL is, let me explain. You may know that a DLL is a **Dynamically Linked Library**, which is essentially a self-contained code module that any number of applications can draw on. When a Web server hosts a dynamic Website, page requests must be processed by program code running on the server before the resultant HTML can be sent back to the requesting browser (the **client**). Now, as was the case with traditional ASP, ASP.NET performs this processing with the help of its **Internet Server Application Programming Interface (ISAPI)** extension DLL. ISAPI allows Web requests to be processed through the Web server by a DLL,

rather than an EXE, as is the case with **Common Gateway Interface (CGI)** pages. This approach is advantageous because DLLs are much more efficient, and require far less resources and memory than executables. IIS uses the file extension of a requested page to determine which DLL should process the request according to the mappings shown in the screenshot above. So, we can see that pages ending in `.aspx`, `.asmx`, or `.ascx`, among others, will now be passed by IIS to the ASP.NET DLL (`aspnet_isapi.dll`) for processing. OK, enough of the tech-talk. Let's get back to it!

If you've come to the conclusion that ASP.NET was not installed on your computer, you'll have to install it manually from the command prompt:

1. Open the command prompt by selecting Start > Run, type CMD, and select OK.

2. Type the following command (all on one line) to install ASP.NET on Windows 2000 Professional, Server, or Advanced Server:

   ```
   C:\WINNT\Microsoft.NET\Framework\ver\aspnet_regiis.exe -i
   ```

   Or on Windows XP Professional:

   ```
   C:\WINDOWS\Microsoft.NET\Framework\ver\aspnet_regiis.exe -i
   ```

   In these commands, *ver* is the directory corresponding to the version of the .NET Framework you have installed.

3. Once ASP.NET is installed, close the command prompt and check again to confirm whether ASP.NET installed correctly.

If it still hasn't installed, try visiting the Microsoft Knowledge Base[6] for help.

Where Do I Put My Files?

Now that you have ASP.NET up and running, let's take a look at where the files for your Web applications are kept on the computer. You can readily set IIS to look for Web applications within any folder of your choice, including the `My Documents` folder or even a network share. By default, IIS maps the `wwwroot` subfolder of `C:\Inetpub` on the server to your Website's root directory, and it is generally considered a good repository for storing and managing your Web applications.

[6] http://support.microsoft.com/

If you open this `wwwroot` folder in Windows Explorer, and compare it with the folder tree that appears on the left of the IIS console, you'll notice that the folders in Explorer also appear under your Default Web Site node. Note that, while several of these folders have the regular Explorer folder icon in the IIS view, others have a special Web application icon, indicating that these folders contain the pages and other items for a particular Web application. These special folders are what IIS calls **Virtual Directories**, and, in fact, they do not have to share the name of the physical folder to which they map. We'll see more on this shortly.

Using Localhost

By putting your files within `C:\Inetpub\wwwroot`, you've given your Web server access to them. If you've been developing Web pages for a long time, habit may drive you to open files directly in your browser by double-clicking on the HTML files. Because ASP.NET is a server-side language, your Web server needs to have a crack at the file before it's sent to your browser for display. If the server doesn't get this opportunity, the ASP.NET code is not converted into HTML that your browser can understand. For this reason, ASP.NET files can't be opened directly from Windows Explorer.

Instead, you need to open them in your browser using the special Web address that indicates the current computer, http://localhost/. If you try this now, IIS will open up some HTML help documentation, because we've not yet set up a default Website. This localhost name is, in fact, equivalent to the so-called **loopback IP address**, 127.0.0.1, IP which you can check out by entering http://127.0.0.1/ in your browser; you should see the same page you saw using localhost. If you know them, you can also use the name of your server or the real IP address of your machine to the same effect.

Note that if you do try any of these equivalents, a dialog will appear before the page is opened, asking for your network credentials, because you're no longer using your local authentication implicit with localhost.

Stopping and Starting IIS

Now that we have IIS up and running, and ASP.NET installed, let's look at how you can start, stop, and restart IIS if the need arises. For the most part, you'll always want to have IIS running, except when you're using certain programs locally that open ports and allow intruders to compromise the security of your computer. Some programs, like Kazaa, automatically stop IIS upon launch, because of potential security vulnerabilities. If you want to stop IIS when it's not being used, simply follow the steps outlined below:

1. With IIS open, select Default Web Site. The Play, Stop, and Pause icons will become visible.

2. Select Stop, as shown in Figure 1.3.

Figure 1.3. Select the Stop icon to stop IIS.

3. To start IIS again, all you need to do is click the Play icon.

Virtual Directories

I've already briefly introduced the concept of **virtual directories**, which are a key mechanism in IIS; now I'd like to define a virtual directory a little more clearly.

A virtual directory is simply a name (or **alias**) that points to a local folder or network share on the server. This alias is then used to access the Web application held in that physical location. For instance, imagine your company has a Web server that serves documents from `C:\Inetpub\wwwroot\mySiteA`. Your users can access these documents through this URL:

http://www.mycompany.com/mySiteA/

You could also set up another physical location as a different virtual directory in IIS. If, for instance, you were developing another Web application, you could store the files for it in `C:\dev\newSiteB`. You could then create in IIS a new virtual directory called, say, `CoolPages`, which maps to this location. This new site would then be accessible through this URL:

http://www.mycompany.com/CoolPages/

As this application is in development, you would probably want to set IIS to hide this virtual directory from the public until the project is complete. Your existing Website would still be visible.

Let's create a virtual directory on your server now:

1. Right-click on Default Web Site and select Virtual Directory from the New submenu. The Virtual Directory Creation Wizard will appear. Click Next.

2. Type in an alias for your virtual directory. I'll type in **`WebDocs`**. Click Next.

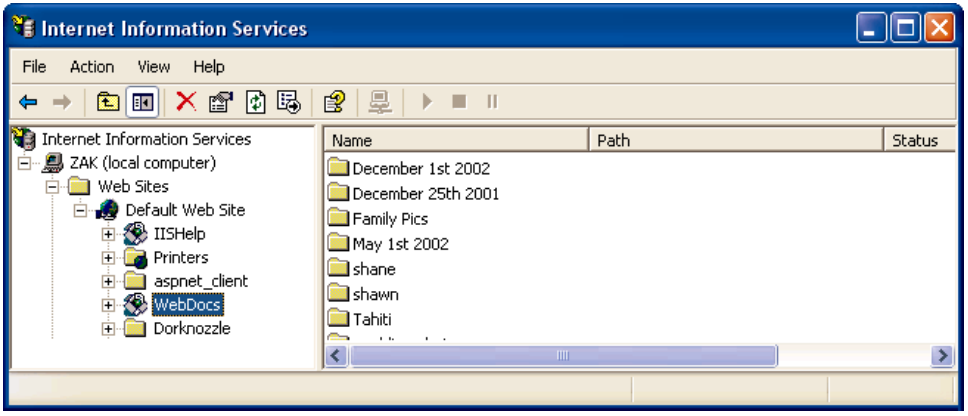
3. Browse for the directory in which your application is located. For this example, I'm going to choose the `My Pictures` folder located within the `My Documents` directory. Click Next.

4. Set Access Permissions for your directory. Typically, you'll want to check Read, Run scripts, and Browse. You will not need to select Write until we get into accessing the file system, discussed in Chapter 15. Click Next.

5. Click Finish.

Once your new virtual directory has been created, it will appear within the Website list as shown in Figure 1.4.

Figure 1.4. Once the virtual directory has been created, it will appear within the list of sites.



Now, if you type http://localhost/WebDocs/ in your browser, IIS will recognize that you're looking for a Website held in the `My Pictures` directory. By default, when we request a virtual directory in this way, IIS looks for an index HTML page such as `index.html` or `default.htm`. If there is no index page—in this case there isn't—IIS assumes we want to see the contents of the requested location.

However, viewing the entire content of a location like this is not usually something we want our users to do; they could then freely see and access all the files and directories that make up our Web page. Not only is this a little messy and unprofessional, but it also can provide information to hackers that could let them attack our site. So, by default, IIS won't allow this—we'll receive a message reading, "Directory Listing Denied" in our browser.

Bearing that in mind, there are, however, circumstances in which we *do* want to allow directory listings, so let's see how we can enable this in IIS. First, we have to right click the virtual directory in the IIS console, and choose Properties. Then, we select the Virtual Directory tab, and check the Directory browsing box. When we click OK and open (or refresh) the same URL in our browser, we'll see a list of all the files within the `My Pictures` folder.

The Properties dialog that we've just used lets us configure various other useful properties, including:

Virtual Directory	Allows you to configure directory-level properties including path information, virtual directory name, access permissions, etc. Everything that was set up through the wizard is modifiable through this tab.
Document	Allows you to configure a default page that displays when the user types in a full URL. For instance, because `default.aspx` is listed as a default page, the user needs only to type in http://www.mysite.com/ into the browser's address bar, rather than http://www.mysite.com/default.aspx. You can easily change and remove these by selecting the appropriate button to the right of the menu.
Directory Security	Provides you with security configuration settings for the virtual directory.
HTTP Headers	Gives you the ability to forcefully control page caching on the server, add custom HTTP Headers, Edit Ratings (helps identify the content your site provides to users), and create MIME types. Don't worry about this for now.
Custom Errors	Allows you to define your own custom error pages. Rather than the standard error messages that appear within Internet Explorer, you can customize error

messages with your company's logo and an error message of your choice.

One thing to note at this point is that we can set properties for the Default Web Site node, and choose to have them 'propagate' down to all the virtual directories we've created. So, let's now go ahead and enable directory browsing as the default for our Web applications. Please do remember what I've said about the dangers of allowing directory browsing on a production Web application, and keep in mind that you should never normally allow it in a publicly accessible environment (even on an intranet). However, during development, this facility can be very handy, as it allows us to navigate and run all our virtual directories by clicking on the listing in our browser, rather than having to type in long URLs each time.

To enable directory browsing:

1. Right-click Default Web Site and select Properties. The Default Web Site Properties dialog will appear.

2. First, we need to remove the default setting which opens up the IIS help documentation for our root directory, so choose the Documents tab.

3. Select `iisstart.asp`, and click Remove.

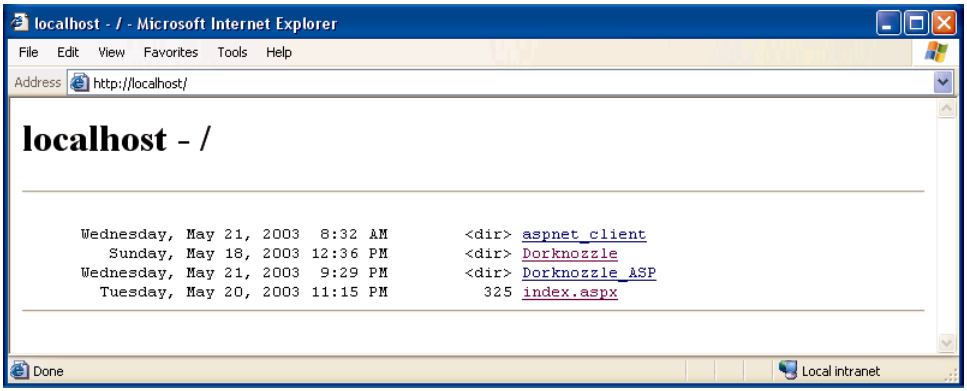
4. Now choose the Home Directory tab.

5. Check the Directory Browsing check box and select OK.

6. When the Inheritance Overrides dialog appears, click Select All and then OK.

To try it out, open your browser and type http://localhost/ in the address bar. The directory listing will appear within the browser as shown in Figure 1.5.

Figure 1.5. Enabling directory browsing for the Web server provides you with the ability to view directories in a way that's similar to the view you'd see within Windows Explorer.



As you create Web applications, you'll only need to select the directory that the Web application resides in to launch your work, but do remember to disable directory browsing should you later make your IIS Web server publicly visible.

Installing Microsoft Access

Access is Microsoft's database solution for both developers and small companies who need to house data within a small yet reliable store. Because Microsoft Access is widely available, it's usually the perfect choice for discussion and use within books such as this. Although we won't be covering data access until Chapter 5, you may want to start thinking about the scope of your or your company's needs and choose a database accordingly. If you're a small company looking for something cheap, reliable, and easy to use, then Access is for you. This book will cover examples using both Access and MSDE. Even if you plan on using MSDE, you may still want to read this section, as Access provides some good data modeling tools that aren't available to you through Web Data Administrator.

You can find more information on Access from the Access Website[15]. Here, you can find the latest updates, news, and purchase information for Microsoft Access.

[15] http://www.microsoft.com/office/access/

Access is bundled with Professional editions of the Microsoft Office suite, so you may already have it installed. If you've already installed Microsoft Office on your computer, but didn't install Access at the same time, you'll need to add it to your installation. The following assumes that you have either Microsoft Office 2000 or XP Professional handy, and that you'll be installing from that CD:

1. Navigate to the Add or Remove Programs menu located within the Control Panel.

2. Select your Microsoft Office installation from the Programs menu and select Change.

3. When the Microsoft Office Setup dialog appears, select Add/Remove Features and click Next.

4. Select Run from My Computer from the Access program menu.

5. Click Update. You will be prompted to insert your Microsoft Office CD, so make sure you have it handy. Access will now install.

If you plan to purchase Access, you might like to consider purchasing the Microsoft Office bundle, as you receive Access, Word, Outlook, PowerPoint, and Excel for much less than the total cost of each of the components. Installing Access from either the Microsoft Access or Microsoft Office CDs is easy—just insert the CD, follow the onscreen prompts, and accept the default installation.

That's all there is to it. You are now ready to begin working with database-driven Web applications.

Installing SQL Server Desktop Engine (MSDE)

SQL Server 2000 is Microsoft's database solution for medium to large companies and enterprises. It is quite a bit more expensive than Access, generally requires its own dedicated "database server", and, at times, requires the hiring of a certified database administrator (DBA) to maintain; yet it offers a robust and scalable solution for larger Web applications.

I'll assume that if you're reading this book, you probably don't want to invest in something as massive as SQL Server, and that your needs are better suited to something free that's nearly as powerful for testing and development purposes. If this is the case, then Microsoft's SQL Server Desktop Engine, or MSDE, is perfect for you. MSDE is Microsoft's free database alternative to SQL Server. It

functions and stores data exactly as SQL Server does, but is licensed for development purposes only.

Once the .NET Framework SDK is installed, installing MSDE is a snap and can be completed as follows:

1. Select Start > Programs > Microsoft Framework SDK, and choose Samples and QuickStart Tutorials.

2. Choose the Download and Install the Microsoft SQL Server 2000 Desktop Engine link. You will be redirected to a download page on Microsoft's Website.

3. Select Step 1: Download the Microsoft SQL Server 2000 Desktop Engine (68.4 MB).

4. Save the file onto your hard drive. At nearly 70 MB, this may take some time, so you may want to move onto the section called "Your First ASP.NET Page" later in this chapter while the download continues, as our first example doesn't use a database. Once the download is done, come back and continue the installation process.

5. Double-click the downloaded file and follow the instructions to unpack the MSDE setup files.

6. Open the Command Prompt by selecting Start > Run; type **`cmd`**, and select OK.

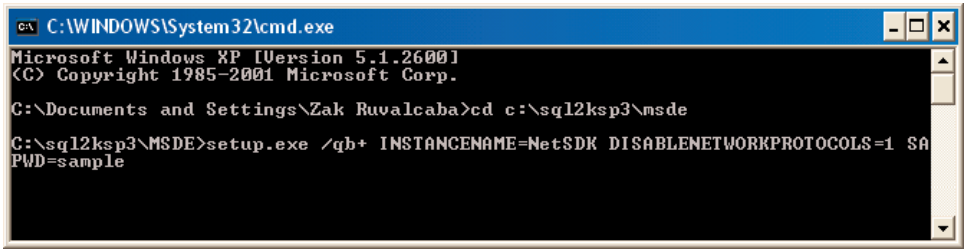
7. Change to the directory to which you extracted the files using cd on the command line. MSDE extracts to `C:\sql2ksp3\MSDE` by default.

8. Type the following command (all on one line) in the MSDE directory to set up MSDE:

```
Setup.exe /qb+ INSTANCENAME=NetSDK DISABLENETWORKPROTOCOLS=1
SAPWD=PASSWORD
```

   The complete set of commands is shown in Figure 1.6.

Figure 1.6. Install MSDE by running the command line executable and setting necessary parameters.



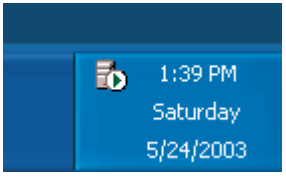
It's a good idea to set a suitable system administrator password using the SAPWD parameter as shown above, although you can apply the traditional blank password by using the **`BLANKSAPWD=1`** parameter instead.

9. MSDE will now install.

10. Restart your computer for changes to take effect.

If all goes well, when the computer restarts, you'll notice in the task bar tray a small icon that looks like a cylinder with a play icon on top, as shown in Figure 1.7.

Figure 1.7. MSDE runs out of sight within the task bar tray.



That icon represents the database Service Manager. It lets you start and stop the database engine; all you have to do is double-click that icon within the task bar tray. Double-click the icon now to open the Service Manager Dialog, where you can select the Play icon to start the service, or the Stop icon to stop the service.

In some cases, you may not see either a green triangle or a red square; instead, you see an empty white circle. When you open Service Manager, you'll see the message "Not Connected" appear in the status bar at the bottom. You'll need to type ***`YourComputer`*`\netsdk`** in the Server drop-down (where *`YourComputer`* is

the name of your computer), and click Refresh services. MSDE should then connect, and the green triangle should appear.

Installing and Configuring Web Data Administrator

In order to use MSDE effectively, you'll need some sort of administration tool to work with your databases. Web Data Administrator is another free tool provided by Microsoft that allows you to manage your instance of MSDE locally and remotely using a Web-based interface. You can download this program from Microsoft's developer site[16]. Scroll to the bottom of that page and you'll find two search boxes. Leave the top one at All, and type **`Web Data Administrator`** in the bottom one, then click search. The search results should include the correct page.

Once you've downloaded it, simply double-click the `.msi` file to install. Once installed, Web Data Administrator can be accessed through your browser at the URL http://localhost/SqlWebAdmin, but before it can be used, you'll need to enable what is known as **SQL Mixed Mode authentication**.

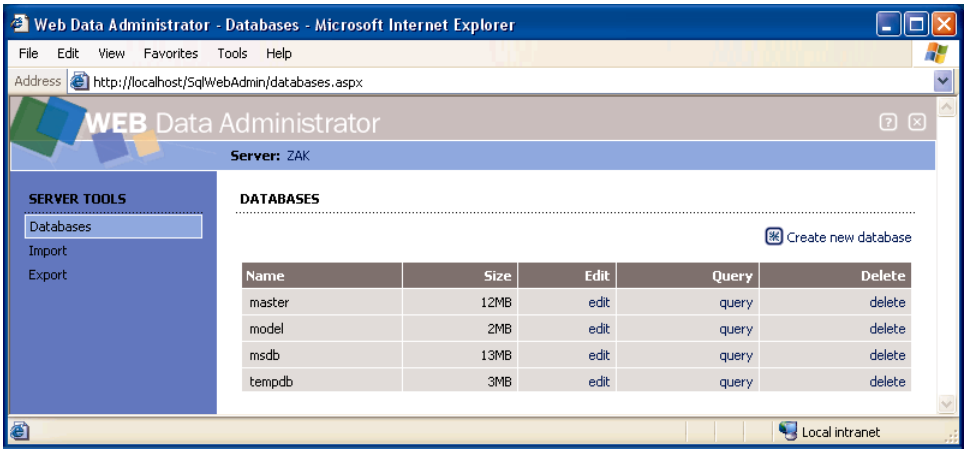
This involves making a small change to the registry, but don't be put off. If you follow these instructions exactly, you won't do any harm. Let's do it! Click Start, then Run.... In the dialog, type **`regedit`** and press Enter to open the registry editor. Now expand the HKEY_LOCAL_MACHINE node in the left hand pane, then expand the SOFTWARE node. Next, find and open the Microsoft node, and, inside that, open one labeled Microsoft SQL Server. In there, you should find a node called NETSDK, which contains another, called MSSQLServer. Select that node, and find the key (in the right hand pane) called LoginMode. Double-click that, and change its Value data from 1 to 2, then click OK. Now, close regedit, and restart your computer. Phew! That was a bit of a trek, but I hope you found it easier in practice than it appears on paper!

Now, open the Web Data Administrator URL given above. You'll be asked for the login, password, and server name for your instance of MSDE. Type **`sa`** in the user name box, and the password that you supplied during the installation of MSDE. If you're unsure what the name of your server is, double-click the database engine icon within the task bar tray. The name of your server is located within the server drop-down menu.

[16] http://msdn.microsoft/downloads/

Once you've done this and clicked Login, you will see a list of the databases that are currently available from MSDE, as shown in Figure 1.8.

Figure 1.8. Web Data Administrator allows you to work with your databases within MSDE.

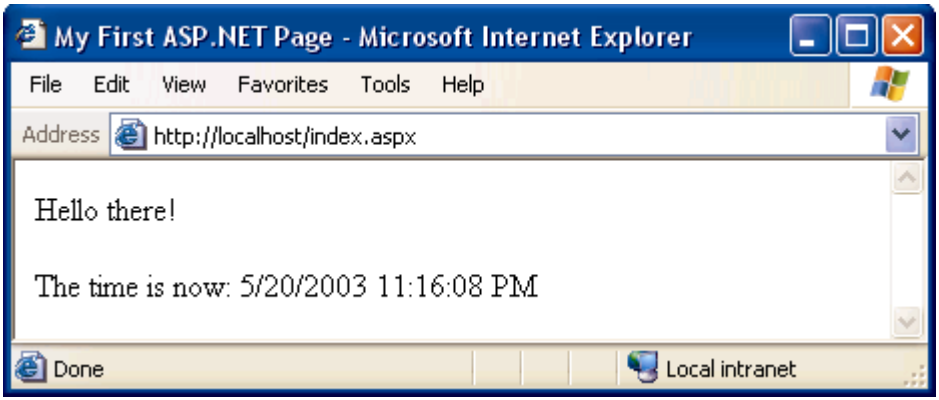


More information on Web Data Administrator, MSDE, and databases will be covered in Chapter 6.

Your First ASP.NET Page

For your first run at ASP.NET, we'll create the simple example shown in Figure 1.9.

Figure 1.9. We'll create a simple ASP.NET page that says "Hello there" and displays the time.



Let's get started! Open your text editor (Notepad[2] is fine). If you have software that creates ASP.NET pages automatically, such as Visual Studio .NET, please do not use it yet. These programs provide lots of powerful tools for building complex ASP.NET pages in a hurry, but for simple examples like this one, they tend to get in the way, rather than provide assistance.

Open your text editor, and start by entering the plain HTML for our page:

```
<html>
<head>
<title>My First ASP.NET Page</title>
</head>
<body>
<p>Hello there!</p>
<p>The time is now: </p>
</body>
</html>
```

So far, so good, right? Now, we'll add some ASP.NET code that will create the dynamic elements of the page, starting with the time.

```
<html>
<head>
```

[2]If you do use Notepad, be aware that to need to put quotes around any filename that doesn't end with `.txt` in the Save As dialog. Most ASP.NET file names end with `.aspx`; if you forget to put quotes around them when saving, you'll end up with files called *`filename`*`.aspx.txt`!

```
<title>My First ASP.NET Page</title>
</head>
<body>
<p>Hello there!</p>
<p>The time is now: <asp:Label runat="server" id="lblTime" /></p>
</body>
</html>
```

We've added an `<asp:Label>` tag to the document. This is a special tag that lets us insert dynamic content into the page. The `asp:` part of the tag name identifies it as a built-in ASP.NET tag. ASP.NET comes with numerous built-in tags; `<asp:Label>` is arguably the simplest.

The `runat="server"` attribute identifies the tag as something that needs to be handled on the server. In other words, the Web browser will never see the `<asp:Label>` tag; ASP.NET sees it and converts it to regular HTML tags before the page is sent to the browser. It's up to us to write the code that will tell ASP.NET to replace this particular tag with the current time.

To do this, we must add some script to our page. Like ASP before it, ASP.NET gives you the choice of a number of different languages to use in your scripts. The two most common languages are Visual Basic.NET (VB.NET) and C# (pronounced "C sharp"). Let's take a look at examples using both. Here's a version of the page in VB.NET:

VB.NET File: **FirstPage.aspx**

```
<html>
<head>
<title>My First ASP.NET Page</title>
<script runat="server" language="VB">
Sub Page_Load(s As Object, e As EventArgs)
  lblTime.Text = DateTime.Now.ToString()
End Sub
</script>
</head>

<body>
<p>Hello there!</p>
<p>The time is now: <asp:Label runat="server" id="lblTime" /></p>
</body>
</html>
```

Here's the same page written in C#:

C# File: **FirstPage.aspx**

```
<html>
<head>
<title>My First ASP.NET Page</title>
<script runat="server" language="C#">
protected void Page_Load(Object s, EventArgs e)
{
  lblTime.Text = DateTime.Now.ToString();
}
</script>
</head>

<body>
<p>Hello there!</p>
<p>The time is now: <asp:Label runat="server" id="lblTime" /></p>
</body>
</html>
```

Both versions of the page achieve exactly the same thing. If you've never done any server-side programming before, this may be starting to look a little scary. Let's break down the new elements of this page:

File: **FirstPage.aspx (excerpt)**

```
<script runat="server">
```

This tag, otherwise known as a **code declaration block**, marks the start of server-side code. Like the `<asp:Label>` tag, this `<script>` tag uses the `runat="server"` attribute to let ASP.NET know that the tag should be processed before sending the page to the browser.

VB.NET File: **FirstPage.aspx (excerpt)**

```
Sub Page_Load(s As Object, e As EventArgs)
```

C# File: **FirstPage.aspx (excerpt)**

```
protected void Page_Load(Object s, EventArgs s) {
```

I won't go into too much detail here. For now, all you need to know is that you can write script fragments that are run in response to different events, such as a button being clicked or an item being selected from a drop-down list. What the first line basically says is "execute the following script whenever the page is loaded." Note that C# groups code into blocks with curly braces, while Visual Basic tends to use statements such as `End Sub` to mark the end of a particular sequence. So, the curly brace in the C# code above (`{`) marks the start of the script that will be executed when the page loads for the first time. For the technically minded,

the code we've just seen is a method definition for a page load event handler, which is essentially the code that the server runs when the page is requested for the first time.

Finally, here's the line that actually displays the time on the page:

VB.NET File: **FirstPage.aspx (excerpt)**

```
  lblTime.Text = DateTime.Now.ToString()
```

C# File: **FirstPage.aspx (excerpt)**

```
  lblTime.Text = DateTime.Now.ToString();
```

You can see that these two .NET languages have much in common, because they are both built on the .NET Framework. In fact, the only difference with the above line is that C# ends code lines with a semicolon (;). In plain English, here's what this line says:

> Set the `Text` property of `lblTime` to the current date/time, expressed as a string of text.

Note that `lblTime` is the value we gave for the `id` attribute of the `<asp:Label>` tag where we want to show the time. So, `lblTime.Text`, the `Text` property of `lblTime`, refers to the text that will be displayed by the tag. `DateTime` is a **class** that's built into the .NET Framework, and which lets you perform all sorts of useful functions with dates and times. There are thousands of these **classes** that do all sorts of useful things within the .NET Framework. These classes are also known as the **.NET Framework Class Library**.

The `DateTime` class has a **property** called `Now` that always contains the current date and time. This `Now` property has a **method** called `ToString()` that expresses that date and/or time as text (a segment of text is commonly called a **string** in programming circles). Classes, properties, and methods: these are all important words in the vocabulary of any programmer, and we'll discuss them later on in the book. For now, all you need to take away from this discussion is that `DateTime.Now.ToString()` will give you the current date and time as a text string, which you can then tell your `<asp:Label>` tag to display. The rest of the script block simply ties up loose ends:

VB.NET File: **FirstPage.aspx (excerpt)**

```
End Sub
</script>
```

C# File: **FirstPage.aspx (excerpt)**

```
}
</script>
```

The closing (`End Sub`) and (`}`) mark the end of the script to be run when the page is loaded, and the `</script>` tag marks the end of the script block.

Create a new subdirectory of `C:\Inetpub\wwwroot` on your Web server, and save your file there under the name `FirstPage.aspx`. Now, open your browser and point type this URL in the address bar:

`http://localhost/test/FirstPage.aspx`

Replace *`test`* with the name that you gave to the directory in which you saved the file. You should see a page similar to the one we saw in Figure 1.9.

If the time isn't displayed, chances are that you opened the file directly in your browser instead of loading it through your Web server. Because ASP.NET is a server-side language, your Web server needs to access the file before it's sent to your browser for display. If it doesn't get access to the file, the ASP.NET code is never converted into HTML that your browser can understand, so make sure you load the page by typing an actual URL (e.g. http://localhost/test/index.aspx), not just a path and filename.

With the page displayed in your browser, use the View Source feature (View, Source in Internet Explorer) to view the HTML code for the page. Here's what you'll see:

```
<html>
<head>
<title>My First ASP.NET Page</title>
</head>
<body>
<p>Hello there!</p>
<p>The time is now: <span id="lblTime">10/13/2003 1:55:09
PM</span></p>
</body>
</html>
```

Notice that all the ASP.NET code has gone! Even the script block has been completely removed, and the `<asp:Label>` tag has been replaced by a `<span>` tag (with the same `id` attribute as the `<asp:Label>` tag that we used) containing the date and time string.

That's how ASP.NET works. From the Web browser's point of view, there is nothing special about an ASP.NET page; it's just plain HTML like any other. All the ASP.NET code is run by your Web server and converted to plain HTML that's sent to the browser. So far, so good: the example above was fairly simple. The next chapter will get a bit more challenging as we begin to introduce you to some valuable programming concepts.

The ASP.NET Support Site

The official Microsoft ASP.NET support Website can be found at http://www.asp.net/. As you develop ASP.NET Web applications, you will undoubtedly have questions and problems that need to be answered. The ASP.NET support Website was developed by Microsoft as a portal for the ASP.NET community to answer the questions and solve the problems that developers have while using ASP.NET. The support Website provides useful information, such as news, downloads, articles, and discussion forums. You can also ask questions of the experienced community members in the SitePoint Forums[20].

Summary

In this chapter, you learned about .NET. You also learned of the benefits of ASP.NET and that it's a part of the .NET Framework. First, you learned about the constructs of ASP.NET and how to locate and install the .NET Framework. Then, we explored the software that's required not only for this book, but also in order for you or your company to progress with ASP.NET.

You've gained a solid foundation in the world of ASP.NET! The next chapter will build on this knowledge and begin to introduce you to ASP.NET in more detail, including page structure, languages to use, programming concepts, and form processing.

[20] http://www.sitepoint.com/forums/

2 ASP.NET Basics

So far, you've learned what ASP.NET is, and what it can do—you even know how to create a simple ASP.NET page. Don't worry if it seems a little bewildering right now, because, as this book progresses, you'll learn how to use ASP.NET at more advanced levels. So far, you've installed the necessary software to get going and have been introduced to some very simple form processing techniques.

As the next few chapters unfold, we'll introduce more advanced topics, including controls, programming techniques, and more. Before we can begin developing applications with ASP.NET, however, you'll need to understand the inner workings of a typical ASP.NET page. This will help you identify the various parts of the ASP.NET page referenced by the many examples within the book. In this chapter, we'll talk about some key mechanisms of an ASP.NET page, specifically:

- Page structure
- View state
- Namespaces
- Directives

We'll also cover two of the "built-in" languages supported by the .NET Framework: VB.NET and C#. As this section begins to unfold, we'll explore the differences,

similarities, and power that the two languages provide in terms of creating ASP.NET applications.

So, what exactly makes up an ASP.NET page? The next few sections will give you an in-depth understanding of the constructs of a typical ASP.NET page.

ASP.NET Page Structure

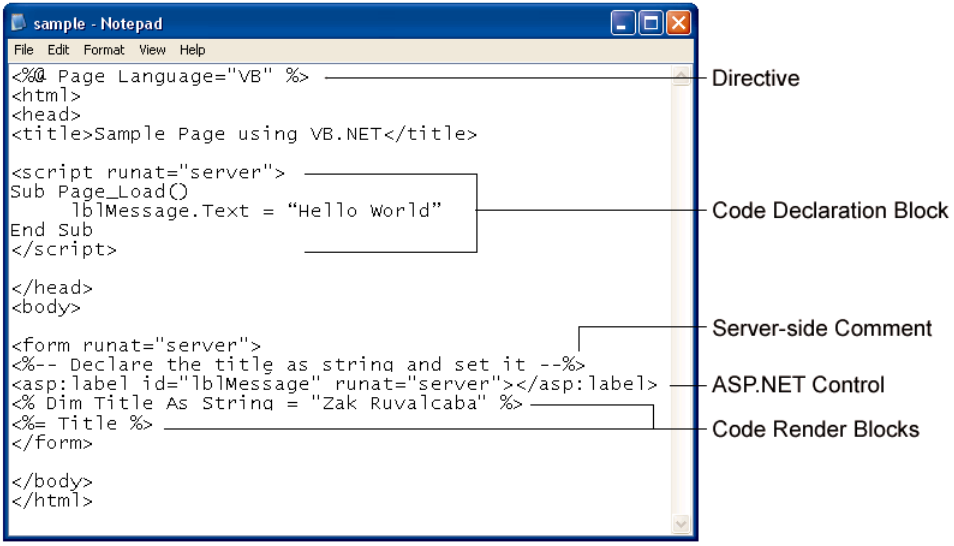
ASP.NET pages are simply text files with the `.aspx` file name extension that can be placed on an IIS server equipped with ASP.NET. When a browser requests an ASP.NET page, the ASP.NET runtime (as a component of the .NET Framework's Common Language Runtime, or CLR) parses and compiles the target file into a .NET Framework class. The application logic now contained within the new class is used in conjunction with the presentational HTML elements of the ASP.NET page to display dynamic content to the user. Sounds simple, right?

An ASP.NET page consists of the following elements:

- Directives
- Code declaration blocks
- Code render blocks
- ASP.NET server controls
- Server-side comments
- Server-side include directives
- Literal text and HTML tags

It's important to remember that ASP.NET pages are just text files with an `.aspx` extension that are processed by the runtime to create standard HTML, based on their contents. Presentational elements within the page are contained within the `<body>` tag, while application logic or code can be placed inside `<script>` tags. Remember this pattern from the sample at the end of the previous chapter? Figure 2.1 illustrates the various parts of that page.

Figure 2.1. All the elements of an ASP.NET page are highlighted. Everything else is literal text and HTML tags.



As you can see, this ASP.NET page contains examples of all the above components (except server-side includes) that make up an ASP.NET page. You won't often use every single element in a given page, but you should become familiar with these elements, the purpose that each serves, and how and when it's appropriate to use them.

Directives

The directives section is one of the most important parts of an ASP.NET page. **Directives** control how a page is compiled, specify settings when navigating between pages, aid in debugging (error-fixing), and allow you to import classes to use within your page's code. Directives start with the sequence `<%@`, followed by the directive name, plus any attributes and their corresponding values, then end with `%>`. Although there are many directives that you can use within your pages, the two most important are the `Import` and `Page` directives. We will discuss directives in greater detail later, but, for now, know that the `Import` and `Page` directives are the most useful for ASP.NET development. Looking at the sample ASP.NET page in Figure 2.1, you can see that a `Page` directive was used at the top of the page as shown:

VB.NET

```
<%@ Page Language="VB" %>
```

C#

```
<%@ Page Language="C#" %>
```

The `Page` directive, in this case, specifies the language that's to be used for the application logic by setting the `Language` attribute appropriately. The value provided for this attribute, in quotes, specifies that we're using either VB.NET or C#. There's a whole range of different directives; we'll see a few more later in this chapter.

Unlike ASP, in ASP.NET, directives can appear anywhere on a page, but are most commonly written as the very first lines.

Code Declaration Blocks

In Chapter 3 we'll talk about code-behind pages and how they let us separate our application logic from an ASP.NET page's HTML presentation code. If you're not working with code-behind pages, however, **code declaration blocks** must be used to contain all the application logic of your ASP.NET page. This application logic defines variables, subroutines, functions, and more. In our page, we place the code inside `<script>` tags, like so:

VB.NET

```
<script runat="server">
Sub mySub()
  ' Code here
End Sub
</script>
```

Here, the tags enclose some VB.NET code, but it could just as easily be C# if our page language were set thus:

C#

```
<script runat="server">
void mySub() {
  // Code here
}
</script>
```

Comments in VB.NET and C# Code

Both of these code snippets contain **comments**—explanatory text that will be ignored by ASP.NET, but which serves to describe how the code works.

In VB.NET code, a single quote or apostrophe (`'`) indicates that the remainder of the line is to be ignored as a comment.

In C# code, two slashes (`//`) does the same. C# code also lets you span a comment over multiple lines by beginning it with `/*` and ending it with `*/`.

Before .NET emerged, ASP also supported such script tags using a `runat="server"` attribute, although they could only ever contain VBScript, and, for a variety of reasons, they failed to find favor among developers. Code declaration blocks are generally placed inside the `<head>` tag of your ASP.NET page. The sample ASP.NET page shown in Figure 2.1, for instance, contained the following code declaration block:

VB.NET

```
<script runat="server">
Sub Page_Load()
  lblMessage.Text = "Hello World"
End Sub
</script>
```

Perhaps you can work out what the equivalent C# code would be:

C#

```
<script runat="server">
void Page_Load() {
  lblMessage.Text = "Hello World";
}
</script>
```

The `<script runat="server">` tag accepts two other attributes, as well. You can set the language used in the block with the `language` attribute:

VB.NET

```
<script runat="server" language="VB">
```

C#

```
<script runat="server" language="C#">
```

If you don't specify a language within the code declaration block, the ASP.NET page will use the language provided by the `language` attribute of the `Page` direct-

ive. Each page may only contain code in a single language; for instance, it is not possible to mix VB.NET and C# in the same page.

The second attribute available is `src`, which lets you specify an external code file to use within your ASP.NET page:

VB.NET
```
<script runat="server" language="VB" src="mycodefile.vb">
```

C#
```
<script runat="server" language="C#" src="mycodefile.cs">
```

Code Render Blocks

You can use **code render blocks** to define inline code or inline expressions that execute when a page is rendered, and you may recognize these blocks from traditional ASP. Code within a code render block is executed immediately as it is encountered, usually when the page is loaded or rendered for the first time, and every time the page is loaded subsequently. Code within a code declaration block, on the other hand, occurring within script tags, is only executed when it is called or triggered by user or page interactions. There are two types of code render blocks: **inline code** and **inline expressions**, both of which are typically written within the body of the ASP.NET page.

Inline code render blocks execute one or more statements and are placed directly inside a page's HTML within `<%` and `%>` characters.

Inline expression render blocks can be compared to `Response.Write()` in classic ASP. They start with `<%=` and end with `%>`, and are used to display values of the variables and methods on a page.

Looking back at Figure 2.1, you can see both types of code render blocks:

VB.NET
```
<% Dim Title As String = "Zak Ruvalcaba" %>
<%= Title %>
```

This equates to the following C#:

C#
```
<% String Title = "Zak Ruvalcaba"; %>
<%= Title %>
```

The first line represents an inline code render block and must contain complete statements in the appropriate language. Here, we're setting the value of the `Title` variable to the string `Zak Ruvalcaba`. The last line is an example of an inline expression render block used to write out the value of the `Title` variable, `Zak Ruvalcaba`, onto the page.

ASP.NET Server Controls

At the heart of ASP.NET pages lies the **server controls**, which represent dynamic elements that your users can interact with. There are four basic types of server control: ASP.NET controls, HTML controls, validation controls, and user controls.

All ASP.NET controls must reside within a `<form runat="server">` tag in order to function correctly. The only two exceptions to this rule are the `HtmlGenericControl` and the `Label` Web control.

Server controls offer the following advantages to ASP.NET developers:

- We can access HTML elements easily from within our code: we can change their characteristics, check their values, or even dynamically update them straight from our server-side programming language of choice.
- ASP.NET controls retain their properties even after the page has been processed. This process is known as **view state**. We'll be covering view state later in this chapter. For now, just know that view state prevents the user from losing data that has already been entered into a form once it's been sent to the server for processing. When the response comes back to the client's browser, text box values, drop-down list selections, etc., are all retained through view state.
- With ASP.NET controls, developers are able to separate the presentational elements (everything the user sees) and application logic (dynamic portions of the ASP.NET page) of a page so that each can be considered separately.

Because ASP.NET is all about controls, we'll be discussing them in greater detail as we move through this book. For instance, in the next few chapters, we'll discuss HTML controls and Web controls (Chapter 4), Validation controls (Chapter 5), Data controls (Chapter 9), and so on.

Server-Side Comments

Server-side comments allow you to include, within the page, comments or notes that will not be processed by ASP.NET. Traditional HTML uses the `<!--` and `-->` character sequences to delimit comments; anything found within these will not be displayed to the user by the browser. ASP.NET comments look very similar, but use the sequences `<%--` and `--%>`.

Our ASP.NET example contains the following server-side comment block:

VB.NET

```
<%-- Declare the title as string and set it --%>
```

The difference between ASP.NET comments and HTML comments is that ASP.NET comments are not sent to the client at all. Don't use HTML comments to try and comment out ASP.NET code. Consider the following example:

VB.NET

```
<!--
<button runat="server" id="myButton" onServerClick="Click">Click
Me</button>
<% Title = "New Title" %>
-->
```

Here, it looks as if a developer has attempted to use an HTML comment to hide not only an HTML button control, but a code render block as well. Unfortunately, HTML comments will only hide things from the browser, not the ASP.NET runtime. So, in this case, while we won't see anything in the browser that represents these two lines, they will, in fact, have been processed by ASP.NET, and the value of the variable `Title` will be changed to `New Title`. The code could be modified to use server-side comments very simply:

VB.NET

```
<%--
<button runat="server" id="myButton" onServerClick="Click">Click
Me</button>
<% Title = "New Title" %>
--%>
```

Now, the ASP.NET runtime will ignore the contents of this comment, and the value of the `Title` variable will not be changed.

Server-Side Include Directives

Server-side include directives enable developers to insert the contents of an external file anywhere within an ASP.NET page. In the past, developers used server-side includes when inserting connection strings, constants, and other code that was generally repeated throughout the entire site.

There are two ways your server-side includes can indicate the external file to include: using either the `file` or the `virtual` attribute. If we use `file`, we specify its filename as the physical path on the server, either as an absolute path starting from a drive letter, or as a path relative to the current file. Below, we see a `file` server-side include with a relative path:

```
<!-- #INCLUDE file="myinclude.aspx" -->
```

`virtual` server-side includes, on the other hand, specify the file's location on the Website, either with an absolute path from the root of the site, or with a path relative to the current page. The example below uses an absolute virtual path:

```
<!-- #INCLUDE virtual="/directory1/myinclude.aspx" -->
```

Note that although server-side includes are still supported by ASP.NET, they have been replaced by a more robust and flexible model known as **user controls**. Discussed in Chapter 16, user controls allow for developers to create a separate page or module that can be inserted into any page within an ASP.NET application.

Literal Text and HTML Tags

The final element of an ASP.NET page is plain old text and HTML . Generally, you cannot do without these elements, and HTML is the means for displaying the information from your ASP.NET controls and code in a way that's suitable for the user. Returning to the example in Figure 2.1 one more time, let's focus on the literal text and HTML tags:

VB.NET

```
<%@ Page Language="VB" %>
<html>
<head>
<title>Sample Page</title>

<script runat="server">
Sub ShowMessage(s As Object, e As EventArgs)
  lblMessage.Text = "Hello World"
```

```
End Sub
</script>

</head>
<body>

<form runat="server">
<%-- Declare the title as string and set it --%>
<asp:Label id="lblMessage" runat="server" />
<% Dim Title As String = "Zak Ruvalcaba's Book List" %>
<%= Title %>
</form>

</body>
</html>
```

As you can see in the bold code, literal text and HTML tags provide the structure for presenting our dynamic data. Without them, there would be no format to the page, and the browser would be unable to understand it.

Now you should understand what the structure of an ASP.NET page looks like. As you work through the examples in this book, you'll begin to realize that in many cases you won't need to use all these elements. For the most part, all of your development will be modularized within code declaration blocks. All of the dynamic portions of your pages will be contained within code render blocks or controls located inside a `<form runat="server">` tag.

In the following sections, we'll outline the various languages used within ASP.NET, talk a little about view state, and look at working with directives in more detail.

View State

As I mentioned briefly in the previous section, ASP.NET controls automatically retain their data when a page is sent to the server by a user clicking a submit button. Microsoft calls this persistence of data **view state**. In the past, developers would have to hack a way to remember the item selected in a drop-down menu or keep the contents of a text box, typically using a hidden form field. This is no longer the case; ASP.NET pages, once submitted to the server for processing, automatically retain all information contained within text boxes, items selected within drop-down menus, radio buttons, and check boxes. Even better, they keep dynamically generated tags, controls, and text. Consider the following ASP page, called `sample.asp`:

```
<html>
<head>
  <title>Sample Page using VBScript</title>
</head>
<body>
<form method="post" action="sample.asp">
  <input type="text" name="txtName"/>
  <input type="Submit" name="btnSubmit" text="Click Me"/>
<%
If Request.Form("txtName") <> "" Then
  Response.Write(Request.Form("txtName"))
End If
%>

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

If you save this example in the `WebDocs` subdirectory of `wwwroot` that you created in Chapter 1, you can open it in your browser by typing http://localhost/WebDocs/sample.asp, to see that view state is not automatically preserved. When the user submits the form, the information that was previously typed into the text box is cleared, although it is still available in `Request.Form("txtName")`. The equivalent page in ASP.NET, `ViewState.aspx`, demonstrates data persistence using view state:

VB.NET File: **ViewState.aspx**

```
<html>
<head>
<title>Sample Page using VB.NET</title>
<script runat="server" language="VB">
Sub Click(s As Object, e As EventArgs)
  lblMessage.Text = txtName.Text
End Sub
</script>
</head>

<body>
<form runat="server">
  <asp:TextBox id="txtName" runat="server" />
  <asp:Button id="btnSubmit" Text="Click Me" OnClick="Click"
      runat="server" />
  <asp:Label id="lblMessage" runat="server" />
</form>
```

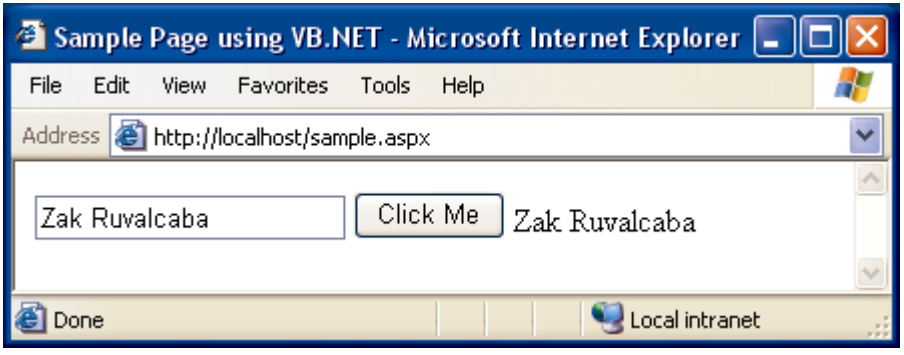
```
</body>
</html>
```

C# File: **ViewState.aspx**

```
<html>
<head>
<title>Sample Page using C#</title>
<script runat="server" language="C#">
void Click(Object s, EventArgs e) {
  lblMessage.Text = txtName.Text;
}
</script>
</head>

<body>
<form runat="server">
  <asp:TextBox id="txtName" runat="server" />
  <asp:Button id="btnSubmit" Text="Click Me" OnClick="Click"
      runat="server" />
  <asp:Label id="lblMessage" runat="server" />
</form>
</body>
</html>
```

In this case, the code uses ASP.NET controls with the `runat="server"` attribute. As you can see in Figure 2.2, the text from the box appears on the page when the button is clicked, but also notice that the data remains in the text box! The data in this example is preserved because of view state:

Figure 2.2. ASP.NET supports view state. When a page is submitted, the information within the controls is preserved.



You can see the benefits of view state already. But where is all that information stored? ASP.NET pages maintain view state by encrypting the data within a hidden form field. View the source of the page after you've submitted the form, and look for the following code:

```
<input type="hidden" name="__VIEWSTATE" value="dDwtMTcyOTAyO
DAwNztOPDtsPGk8Mj47PjtsPHQ8O2w8aTwzPjs+O2w8dDxwPGw8aW5uZXJodG
1sOz47bDxIZWxsbyBXb3JsZDs+Pjs7Pjs+Pjs+d2wl7GlhgweO9LlUihS
FaGxk6t4=" />
```

This is a standard HTML hidden form field with the value set to the encrypted data from the form element. As soon as you submit the form for processing, all information relevant to the view state of the page is stored within this hidden form field.

View state is enabled for every page by default. If you do not intend to use view state, you can turn it off, which will result in a slight performance gain in your pages. To do this, set the `EnableViewState` property of the `Page` directive to `false`:

```
<%@ Page EnableViewState="False" %>
```

Speaking of directives, it's time we took a closer look at these curious beasts!

Working With Directives

For the most part, ASP.NET pages resemble traditional HTML pages, with a few additions. In essence, just using an extension like `.aspx` on an HTML file will make the .NET Framework process the page. However, before you can work with certain, more advanced features, you will need to know how to use directives.

We've already talked a little about directives and what they can do earlier in this chapter. You learned that directives control how a page is created, specify settings when navigating between pages, aid in finding errors, and allow you to import advanced functionality to use within your code. Three of the most commonly used directives are:

`Page` Defines page-specific attributes for the ASP.NET page, such as the language used.

`Import` Makes functionality defined elsewhere available in a page through the use of namespaces. You will become very familiar with this directive as you progress through this book.

`Register` Asyou will see in Chapter 16, you would use this directive to link a user control to the ASP.NET page.

You will become very familiar with these three directives, as they're the ones that we'll be using the most in this book. You've already seen the `Page` directive in use. The `Import` directive imports extra functionality for use within your application logic. The following example, for instance, imports the `Mail` class, which you could use to send email from a page:

```
<%@ Import Namespace="System.Web.Mail" %>
```

The `Register` directive allows you to register a **user control** for use on your page. We'll cover these in Chapter 16, but the directive looks something like this:

```
<%@ Register TagPrefix="uc" TagName="footer" Src="footer.ascx" %>
```

ASP.NET Languages

As we saw in the previous chapter, .NET currently supports many different languages and there is no limit to the number of languages that could be made available. If you're used to writing ASP, you may think the choice of VBScript would be obvious. With ASP.NET however, Microsoft has done away with VBScript and replaced it with a more robust and feature-rich alternative: VB.NET. ASP.NET's support for C# is likely to find favor with developers from other backgrounds. This section will introduce you to both these new languages, which are used throughout the remainder of the book. By the end of this section, you will, I hope, agree that the similarities between the two are astonishing—any differences are minor and, in most cases, easy to figure out.

Traditional server technologies are much more constrained in the choice of development language they offer. For instance, old-style CGI scripts were typically written with Perl or C/C++, JSP uses Java, Coldfusion uses CFML, and PHP is a language in and of itself. .NET's support for many different languages lets developers choose based on what they're familiar with, and start from there. To keep things simple, in this book we'll consider the two most popular, VB.NET and C#, giving you a chance to choose which feels more comfortable to you, or stick with your current favorite if you have one.

VB.NET

Visual Basic.NET or VB.NET is the result of a dramatic overhaul of Microsoft's hugely popular Visual Basic language. With the inception of **Rapid Application**

Development (RAD) in the nineties, Visual Basic became extremely popular, allowing inhouse teams and software development shops to bang out applications two-to-the-dozen. VB.NET has many new features over older versions of VB, most notably that it has now become a fully object-oriented language. At last, it can call itself a true programming language on a par with the likes of Java and C++. Despite the changes, VB.NET generally stays close to the structured, legible syntax that has always made it so easy to read, use, and maintain.

C#

The official line is that Microsoft created C# in an attempt to produce a programming language that coupled the simplicity of Visual Basic with the power and flexibility of C++. However, there's little doubt that its development was at least hurried along. Following legal disputes with Sun about Microsoft's treatment (some would say abuse) of Java, Microsoft was forced to stop developing its own version of Java, and instead developed C# and another language, which it calls J#. We're not going to worry about J# here, as C# is preferable. It's easy to read, use, and maintain, because it does away with much of the confusing syntax for which C++ became infamous.

Summary

In this chapter, we started out by introducing key aspects of an ASP.NET page including directives, code declaration blocks, code render blocks, includes, comments, and controls. As the chapter progressed, you were introduced to the two most popular languages that ASP.NET supports, which we'll use throughout the book.

In the next chapter, we'll create more ASP.NET pages to demonstrate some form processing techniques and programming basics, before we finally dive in and look at object oriented programming for the Web.

3 VB.NET and C# Programming Basics

As you learned at the end of the last chapter, one of the great things about ASP.NET is that we can pick and choose which of the various .NET languages we like. In this chapter, we'll look at some key programming principles using our two chosen languages, VB.NET and C#. We'll start off with a run-down of some basic programming concepts as they relate to ASP.NET using both languages. We'll introduce programming fundamentals such as control and page events, variables, arrays, functions, operators, conditionals, and loops. Next, we'll dive into namespaces and address the topic of classes—how they're exposed through namespaces, and which you'll use most often.

The final sections of the chapter cover some of the ideas underlying modern, effective ASP.NET design, starting with that of code-behind and the value it provides by helping us separate code from presentation. We finish with an examination of how object-oriented programming techniques impact the ASP.NET developer.

Programming Basics

One of the building blocks of an ASP.NET page is the application logic: the actual programming code that allows the page to function. To get anywhere with this, you need to grasp the concept of **events**. All ASP.NET pages will contain controls, such as text boxes, check boxes, lists, and more, each of these controls

allowing the user to interact with it in some way. Check boxes can be checked, lists can be scrolled, items on them selected, and so on. Now, whenever one of these actions is performed, the control will raise an event. It is by handling these events with code that we get our ASP.NET pages to do what we want.

For instance, say a user clicks a button on an ASP.NET page. That button (or, strictly, the ASP.NET `Button` control) raises an event (in this case it will be the `Click` event). When the ASP.NET runtime registers this event, it calls any code we have written to handle it. We would use this code to perform whatever action that button was supposed to perform, for instance, to save form data to a file, or retrieve requested information from a database. Events really are key to ASP.NET programming, which is why we'll start by taking a closer look at them. Then, there's the messy business of writing the actual handler code, which means we need to check out some common programming techniques in the next sections. Specifically, we're going to cover the following areas:

- Control events and handlers
- Page events
- Variables and variable declaration
- Arrays
- Functions
- Operators
- Conditionals
- Loops

It wouldn't be practical, or even necessary, to cover all aspects of VB.NET and C# in this book, so we're going to cover enough to get you started, completing the projects and samples using both languages. Moreover, I'd say that the programming concepts you'll learn here will be more than adequate to complete the great majority of day-to-day Web development tasks using ASP.NET.

Control Events and Subroutines

As I just mentioned, an event (sometimes more than one) is raised, and handler code is called, in response to a specific action on a particular control. For instance,

the code below creates a server-side button and label. Note the use of the `OnClick` attribute on the Button control:

File: **ClickEvent.aspx (excerpt)**

```
<form runat="server">
  <asp:Button id="btn1" runat="server" OnClick="btn1_Click"
      Text="Click Me" />
  <asp:Label id="lblMessage" runat="server" />
</form>
```

When the button is clicked, it raises the `Click` event, and ASP.NET checks the `OnClick` attribute to find the name of the handler subroutine for that event. Here, we tell ASP.NET to call the `btn1_Click()` routine. So now we have to write this **subroutine**, which we would normally place within a code declaration block inside the `<head>` tag, like this:

VB.NET File: **ClickEvent.aspx (excerpt)**

```
<head>
<script runat="server" language="VB">
  Public Sub btn1_Click(s As Object, e As EventArgs)
    lblMessage.Text = "Hello World"
  End Sub
</script>
</head>
```

C# File: **ClickEvent.aspx (excerpt)**

```
<head>
<script runat="server" language="C#">
  public void btn1_Click(Object s, EventArgs e) {
    lblMessage.Text = "Hello World";
  }
</script>
</head>
```

This code simply sets a message to display on the label that we also declared with the button. So, when this page is run and users click the button, they'll see the message "Hello World" appear next to it.

I hope you can now start to come to grips with the idea of control events and how they're used to call particular subroutines. In fact, there are many events that your controls can use, some of which are only found on certain controls—not others. Here's the complete set of attributes the `Button` control supports for handling events:

OnClick	As we've seen, the subroutine indicated by this attribute is called for the `Click` event, which occurs when the user clicks the button.
OnCommand	As with `OnClick`, the subroutine indicated by this attribute is called when the button is clicked.
OnLoad	The subroutine indicated by this attribute is called when the button is loaded for the first time—generally when the page first loads.
OnInit	When the button is initialized, any subroutine given in this attribute will be called.
OnPreRender	We can run code just before the button is rendered, using this attribute.
OnUnload	This subroutine will run when the control is unloaded from memory—basically, when the user goes to a different page or closes the browser entirely.
OnDisposed	This occurs when the button is released from memory.
OnDataBinding	This fires when the button is bound to a data source.

Don't worry too much about the intricacies of all these events and when they happen; I just want you to understand that a single control can produce a number of different events. In the case of the `Button` control, you'll almost always be interested in the `Click` event, as the others are only useful in rather obscure circumstances.

When a control raises an event, the specified subroutine (if there is one) is executed. Let's now take a look at the structure of a typical subroutine that interacts with a Web control:

VB.NET

```
Public Sub mySubName(s As Object, e As EventArgs)
  ' Write your code here
End Sub
```

C#

```
public void mySubName(Object s, EventArgs e) {
  // Write your code here
}
```

Let's break down all the components that make up a typical subroutine:

`Public` **`public`**	Defines the **scope** of the subroutine. There are a few different options to choose from, the most frequently used being `Public` (for a global subroutine that can be used anywhere within the entire page) and `Private` (for subroutines that are available for the specific class only). If you don't yet understand the difference, your best bet is to stick with `Public` for now.
`Sub` **`void`**	Defines the chunk of code as a subroutine. A subroutine is a named block of code that doesn't return a result; thus, in C#, we use the `void` keyword, which means exactly that. We don't need this in VB.NET, because the `Sub` keyword already implies that no value is returned.
***`mySubName`*`(…)`**	This part gives the name we've chosen for the subroutine.
`s As Object` **`Object s`**	When we write a subroutine that will function as an event handler, it must accept two **parameters**. The first is the control that generated the event, which is an `Object`. Here, we are putting that `Object` in a **variable** named `s` (more on variables later in this chapter). We can then access features and settings of the specific control from our subroutine using the variable.
`e As EventArgs` **`EventArgs e`**	The second parameter contains certain information specific to the event that was raised. Note that, in many cases, you won't need to use either of these two parameters, so you don't need to worry about them too much at this stage.

As this chapter progresses, you'll see how subroutines associated with particular events by the appropriate attributes on controls can revolutionize the way your user interacts with your application.

Page Events

Until now, we've considered only events that are raised by controls. However, there is another type of event—the **page event**. The idea is the same as for control events[1], except that here, it is the page as a whole that generates the events. You've already used one of these events: the `Page_Load` event. This event is fired when the page loads for the first time. Note that we don't need to associate handlers for page events the way we did for control events; instead, we just place our handler code inside a subroutine with a preset name. The following list outlines the page event subroutines that are available:

`Page_Init`	Called when the page is about to be initialized with its basic settings
`Page_Load`	Called once the browser request has been processed, and all of the controls in the page have their updated values.
`Page_PreRender`	Called once all objects have reacted to the browser request and any resulting events, but before any response has been sent to the browser.
`Page_UnLoad`	Called when the page is no longer needed by the server, and is ready to be discarded.

The order in which the events are listed above is also the order in which they're executed. In other words, the `Page_Init` event is the first event raised by the page, followed by `Page_Load`, `Page_PreRender`, and finally `Page_UnLoad`.

The best way to illustrate the `Page_Load` event is through an example:

VB.NET File: **`PageEvents.aspx (excerpt)`**

```
<html>
<head>
<script runat="server" language="VB">
Sub Page_Load(s As Object, e As EventArgs)
  lblMessage.Text = "Hello World"
End Sub
```

[1] Strictly speaking, a page is simply another type of control, and so page events *are* actually control events. When you're first coming to grips with ASP.NET, however, it can help to think of them differently, especially since you don't usually use `On`*`EventName`* attributes to assign subroutines to handle them.

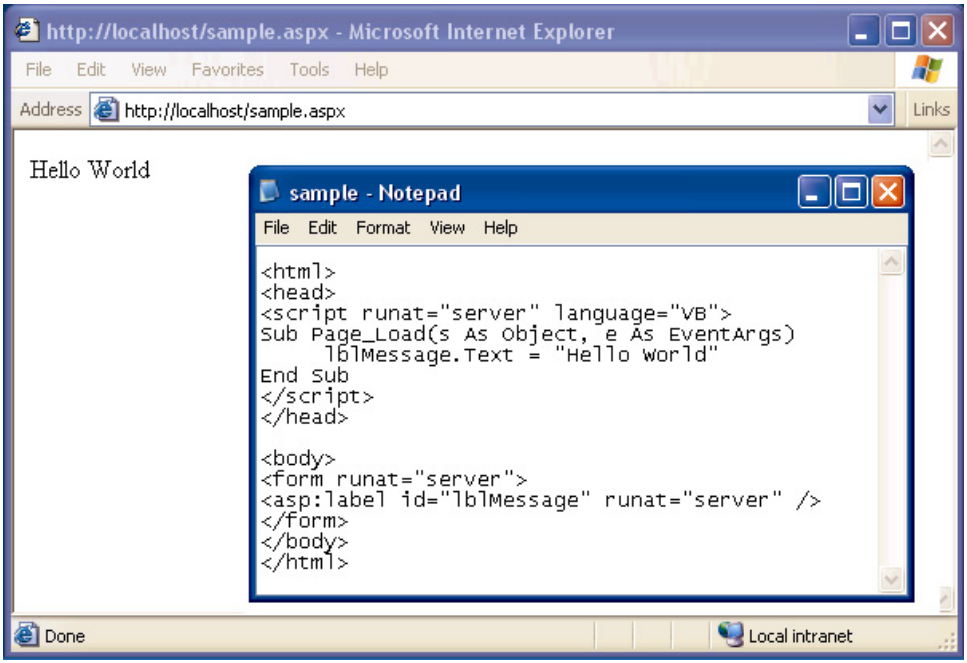
```
</script>
</head>

<body>
<form runat="server">
<asp:Label id="lblMessage" runat="server" />
</form>
</body>
</html>
```

C# File: **PageEvents.aspx (excerpt)**

```
<html>
<head>
<script runat="server" language="C#">
void Page_Load(Object s, EventArgs e) {
  lblMessage.Text = "Hello World";
}
</script>
</head>

<body>
<form runat="server">
<asp:Label id="lblMessage" runat="server" />
</form>
</body>
</html>
```

You can see that the control on the page does not specify any event handlers. There's no need, because we're using the special `Page_Load` subroutine, which will be called when the page loads. As the page loads, it will call the `Page_Load` routine, to display "Hello World" in the `Label` control, as shown in Figure 3.1.

Figure 3.1. The Page_Load event is raised, the subroutine is called, and the code within the subroutine is executed.



Variables and Variable Declaration

Variables are fundamental to programming, and you've almost certainly come across the term before. Basically, they let you give a name, or **identifier**, to a piece of data; we can then use that identifier to store, modify, and retrieve the data.

However, there are, of course, many different kinds of data, such as strings, integers (whole numbers), and floating point numbers (fractions or decimals). Before you can use a variable in VB.NET or C#, you must specify what type of data it can contain, using keywords such as `String`, `Integer`, `Decimal`, and so on, like this:

VB.NET

```
Dim strName As String
Dim intAge As Integer
```

C#
```
string strName;
int intAge;
```

These lines declare what type of data we want our variables to store, and are therefore known as **variable declarations**. In VB.NET, we use the keyword `Dim`, which stands for "dimension", while in C#, we simply precede the variable name with the appropriate data type.

Sometimes, we want to set an initial value for variables that we declare; we can do this using a process known as **initialization**:

VB.NET
```
Dim strCarType As String = "BMW"
```

C#
```
string strCarType = "BMW";
```

We can also declare and/or initialize a group of variables of the same type all at once:

VB.NET
```
Dim strCarType As String, strCarColor = "blue", strCarModel
```

C#
```
string strCarType, strCarColor = "blue", strCarModel;
```

Table 3.1 below lists the most useful data types available in VB.NET and C#.

Table 3.1. A List of the Commonly Used Data Types

VB.NET	C#	Description
`Integer`	`int`	Whole numbers in the range -2,147,483,648 to 2,147,483,647.
`Decimal`	`decimal`	Up to 28 decimal places. You'll use decimal most often when dealing with costs of items.
`String`	`string`	Any text value.
`Char`	`char`	A single character (letter, number, or symbol).
`Boolean`	`bool`	True or false.
`Object`	`Object`	In .NET, all types are ultimately a type of object, and so variables of this type can hold just about any kind of data.

There are many more data types that you may encounter as you progress, but this list provides an idea of the ones you'll use most often.

So, to sum up, once you've declared a variable as a given type, it can only hold data of that type. You can't put a string into an integer variable, for instance. However, there are frequently times when you'll need to convert one data type to another. Have a look at this code:

VB.NET

```
Dim intX As Integer
Dim strY As String = "35"
intX = strY + 6
```

C#

```
int intX;
String strY = "35";
intX = strY + 6;
```

Now, while you or I might think that this could make sense—after all, the string `strY` *does* contain a number, so we may well wish to add it to another number—the computer will not be happy, and we'll get an error. What we have to do is explicitly convert, or **cast**, the string into an integer first:

VB.NET

```
Dim intX As Integer
Dim strY As String = "35"
intX = Int32.Parse(strY) + 6
```

C#

```
int intX;
String strY = "35";
intX = Convert.ToInt32(strY) + 6;
```

Now, the computer will be happy, as we've told it that we want to turn the string into an integer before it's used as one. This same principle holds true when mixing other types in a single expression.

Arrays

Arrays are a special variety of variable tailored for storing related items of the same data type. Any one item in an array can be accessed using the array's name, followed by that item's position in the array (its **offset**). Let's create a sample page to show what I mean:

VB.NET File: **Arrays.aspx**

```
<html>
<head>
<script runat="server" language="VB">
Sub Page_Load()

  ' Declare an array
  Dim drinkList(4) As String

  ' Place some items in it
  drinkList(0) = "Water"
  drinkList(1) = "Juice"
  drinkList(2) = "Soda"
  drinkList(3) = "Milk"

  ' The line below accesses an item in the array by its position
  lblArrayList.Text = drinkList(1)
End Sub
</script>
</head>

<body>
<form runat="server">
<asp:Label id="lblArrayList" runat="server"/>
</form>
</body>
</html>
```

C# File: **Arrays.aspx**

```
<html>
<head>
<script runat="server" language="C#">
void Page_Load() {

  // Declare an array
  String[] drinkList = new String[4];

  // Place some items in it
  drinkList[0] = "Water";
  drinkList[1] = "Juice";
  drinkList[2] = "Soda";
  drinkList[3] = "Milk";

  // The line below accesses an item in the array by its position
  lblArrayList.Text = drinkList[1];
}
</script>
</head>

<body>
<form runat="server">
<asp:Label id="lblArrayList" runat="server"/>
</form>
</body>
</html>
```

There are some important points to pick up from this code. First, notice how we declare an array. In VB.NET, it looks like a regular declaration for a string, except that the number of items we want the array to contain is given in brackets after the name:

VB.NET File: **Arrays.aspx (excerpt)**

```
  Dim drinkList(4) As String
```

In C#, it's a little different. First, we declare that `drinkList` is an array by following the datatype with two empty square brackets. We must then specify that this is an array of four items, using the `new` keyword:

C# File: **Arrays.aspx (excerpt)**

```
  String[] drinkList = new String[4];
```

A crucial point to realize here is that the arrays in both C# and VB.NET are what are known as **zero-based** arrays. This means that the first item actually has

position 0, the second has position 1, and so on, through to the last item, which will have a position that's one less than the size of the array (3, in this case). So, we specify each item in our array like this:

VB.NET File: **Arrays.aspx (excerpt)**

```
drinkList(0) = "Water"
drinkList(1) = "Juice"
drinkList(2) = "Soda"
drinkList(3) = "Milk"
```

C# File: **Arrays.aspx (excerpt)**

```
drinkList[0] = "Water";
drinkList[1] = "Juice";
drinkList[2] = "Soda";
drinkList[3] = "Milk";
```

Notice that C# uses square brackets for arrays, while VB.NET uses standard parentheses. We have to remember that arrays are zero-based when we set the label text to the second item, as shown here:

VB.NET File: **Arrays.aspx (excerpt)**

```
lblArrayList.Text = drinkList(1)
```

C# File: **Arrays.aspx (excerpt)**

```
lblArrayList.Text = drinkList[1];
```

To help this sink in, you might like to try changing this code to show the third item in the list instead of the second. Can you work out what change you'd need to make?

That's right—you only need to change the number given in the brackets to match the position of the new item (don't forget to start at zero). In fact, it's this ability to select one item from a list using only its numerical location that makes arrays so useful in programming, as we'll see as we get further into the book.

Functions

Functions are exactly the same as subroutines, but for one key difference: they return a value. In VB.NET, we declare a function using the `Function` keyword in place of `Sub`, while, in C#, we simply have to specify the return type in place of using `void`. The following code shows a simple example:

VB.NET File: **Functions.aspx**

```
<html>
<head>
<script runat="server" language="VB">

' Here's our function
Function getName() as String
  Return "Zak Ruvalcaba"
End Function

' And now we'll use it in the Page_Load handler
Sub Page_Load(s As Object, e As EventArgs)
  lblMessage.Text = getName()
End Sub
</script>
</head>

<body>
<form runat="server">
<asp:Label id="lblMessage" runat="server" />
</form>
</body>
</html>
```

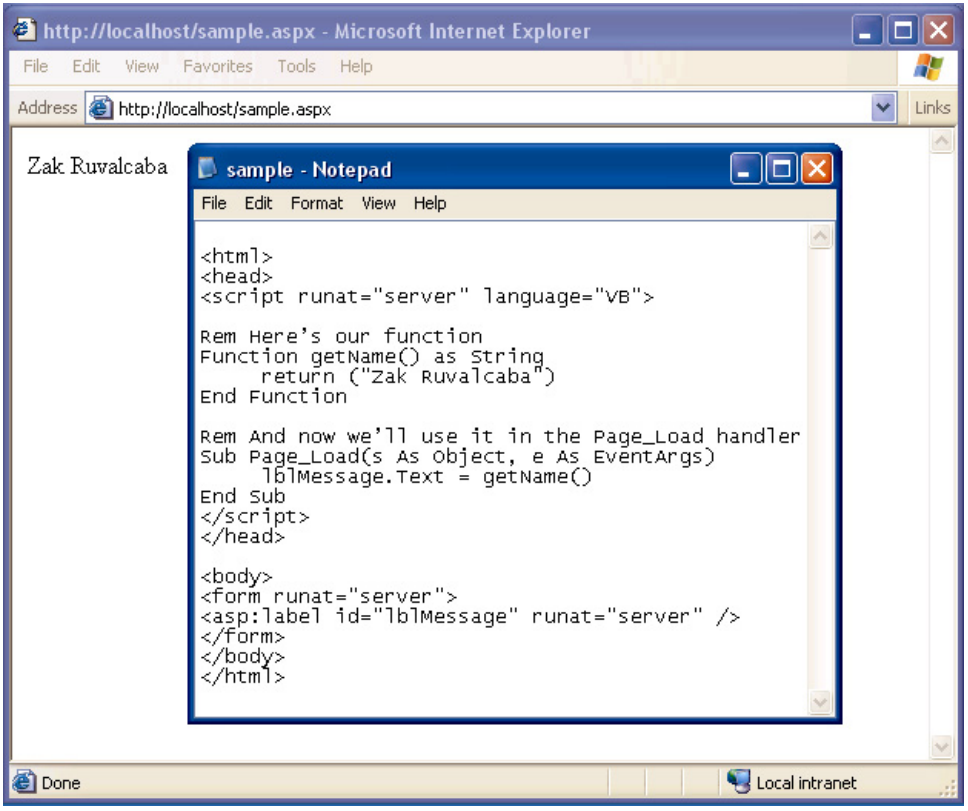
C# File: **Functions.aspx**

```
<html>
<head>
<script runat="server" language="C#">

// Here's our function
string getName() {
  return "Zak Ruvalcaba";
}

// And now we'll use it in the Page_Load handler
void Page_Load() {
  lblMessage.Text = getName();
}
</script>
</head>

<body>
<form runat="server">
<asp:Label id="lblMessage" runat="server" />
</form>
```

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

Figure 3.2 shows the result in the browser.

Figure 3.2. The `Page_Load` event is raised, the function is called, and the code within the function is executed.



Here's what's happening: the line in our `Page_Load` subroutine calls our function, which returns a simple string that we can then assign to our label. I hope this illustrates what functions are about and how you can use them. In this simple example, we're merely returning a fixed string (my name), but the function could just as well retrieve the name from a database—or somewhere else. The point is that, regardless of how the function gets its data, we use it (that is, call it) in just the same way.

When we're declaring our function, we must remember to specify the correct **return type**. Take a look at the following code:

VB.NET

```
' Here's our function
Function addUp(x As Integer, y As Integer) As Integer
  Return x + y
End Function

' And now we use it in Page_Load
Sub Page_Load(s As Object, e As EventArgs)
  lblMessage.Text = addUp(5, 5).ToString()
End Sub
```

C#

```
// Here's our function
int addUp(int x, int y) {
  return x + y;
}

// And now we use it in Page_Load
void Page_Load() {
  lblMessage.Text = Convert.ToString(addUp(5, 5));
}
```

You can easily adapt the previous example to use this new code and see the results in your browser.

Have a look at this code, and see if you can spot what's different and why. The first thing you might notice is that our function now accepts **parameters**. Any function or subroutine can take any number of parameters, each of any type (there's no need for parameter types to match the return type—that's just coincidental in this example).

We can then readily use the parameters inside the function or subroutine just by using the names we gave them in the function declaration (here, we've chosen `x` and `y`, but we could have chosen different names).

The other difference between this and the function declaration we had before is that we now declare our function with a return type of `Integer` or `int`, rather than `String`, because we want it to return a whole number.

When we now call the new function, we simply have to specify the required number of parameters, and remember that the function will return a value with

the type we specified. In this case, that means we have to convert the integer value it returns to a string, so we can assign it to the label.

In VB.NET, we tack `.ToString()` onto the end of the function call, while in C# we use the `Convert.ToString(…)`. Note the differences in how these two methods are used—converting numbers to strings is a very common task in ASP.NET, so it's good to get a handle on it early. Don't be too concerned if you're a little confused by how these conversions work, though—the syntax will become clear once we discuss the object oriented concepts involved later in this chapter.

Again, a complete discussion of functions could take up an entire chapter, but I hope the brief examples here are enough to prepare you for what we're going to cover in future chapters. Don't worry too much if you're still a bit unsure what functions and subroutines are all about right now—they'll become second nature very quickly.

Operators

Throwing around values with variables and functions isn't much use unless you can use them in some meaningful way, and to do this we use **operators**. An operator is a symbol that has a certain meaning when applied to values. Don't worry—they're nowhere near as scary as they sound! In fact, in the last example, when our function added two numbers, we were using an operator—the **addition operator**, `+`. Most of the other available operators are just as well known, although there are one or two that will probably be new to you. Table 3.2 outlines the operators that you'll use most often.

Table 3.2. ASP.NET Operators

VB.NET	C#	Description
>	>	greater than
>=	>=	greater than or equal to
<	<	less than
<=	<=	less than or equal to
<>	!=	not equal to
=	==	equals
=	=	assigns a value to a variable
Or	||	or
And	&&	and
&	+	concatenate strings
New	New	create object or array
*	*	multiply
/	/	divide
+	+	add
-	-	subtract

The following code uses some of these operators:

VB.NET

```
If (user = "Zak" And itemsBought <> 0) Then
  lblMessage.Text = "Hello Zak! Do you want to proceed to " & _
    "checkout?"
End If
```

C#

```
if (user == "Zak" && itemsBought != 0) {
  lblMessage.Text = "Hello Zak! Do you want to proceed to " +
    "checkout?";
}
```

Here, we use the equality, inequality (not equals to) and logical 'and' operators in an `If` statement to print a message only for a given user, and only when he or she has bought something. Of particular note is C#'s equality operator, ==, which is used when comparing two values to see if they are equal. Don't use a single

equals sign in C# unless you are assigning a value to a variable, or your code will have a very different meaning than you expect!

Breaking Long Lines of Code

Since the message string in the above example was too long to fit on one line in this book, I also used the string concatenation operator to combine two shorter strings on separate lines to form the complete message. In VB.NET, I also had to break one line of code into two using the line continuation symbol (_, an underscore at the end of the line to be continued). Since C# marks the end of each command with a semicolon (`;`), I can split a single command over two lines without having to do anything special.

I'll use these techniques throughout this book to show long lines of code within a limited page width. Feel free to recombine the lines in your own code if you like—there is no actual length limit on lines of code in VB.NET and C#.

Conditional Logic

As you develop ASP.NET applications, there will be many instances in which you'll need to perform an action only if a certain condition is met, for instance, if the user has checked a certain checkbox, selected a certain item from a `DropDownList` control, or typed a certain string into a `TextBox` control.

We check for such things using **conditionals**, the simplest of which is probably the `If` statement. This statement is often used in conjunction with an `Else` statement, which specifies what should happen if the condition is not met. So, for instance, we may wish to check *if* the name entered in a text box is "Zak," redirecting to one page if it is, or *else* redirecting to an error page:

VB.NET

```
If (txtUsername.Text = "Zak") Then
  Response.Redirect("ZaksPage.aspx")
Else
  Response.Redirect("errorPage.aspx")
End If
```

C#

```
if (txtUsername.Text == "Zak") {
  Response.Redirect("ZaksPage.aspx");
} else {
  Response.Redirect("errorPage.aspx");
}
```

Often, we want to check for one of many possibilities, and perform a particular action in each case. In that event, we can use the `Switch Case` (VB.NET) or `switch` (C#) construct:

VB.NET

```
Dim strName As String = txtUsername.Text
Select Case strName
  Case "Zak"
    Response.Redirect("ZaksPage.aspx")
  Case "Mark"
    Response.Redirect("MarksPage.aspx")
  Case "Fred"
    Response.Redirect("FredsPage.aspx")
  Case Else
    Response.Redirect("errorPage.aspx")
End Select
```

C#

```
string strName = txtUsername.Text;
switch (strName) {
  case "Zak":
    Response.Redirect("ZaksPage.aspx");
    break;
  case "Mark":
    Response.Redirect("MarksPage.aspx");
    break;
  case "Fred":
    Response.Redirect("FredsPage.aspx");
    break;
  default:
    Response.Redirect("errorPage.aspx");
    break;
}
```

Loops

As you've just seen, an `If` statement causes a code block to execute once if the value of its test expression is true. Loops, on the other hand, cause a code block to execute repeatedly for as long as the test expression remains true. There are two basic kinds of loop:

- `While` loops, also called `Do` loops, which sounds like something Betty Boop might say!

- `For` loops, including `For Next` and `For Each`

A `While` loop is the simplest form of loop; it makes a block of code repeat for as long as a particular condition is true. Here's an example:

VB.NET

```
Dim Counter As Integer = 0

Do While Counter <= 10

  ' Convert out Integer to a String
  lblMessage.Text = Counter.ToString()

  ' Below we use the += operator to increase our variable by 1
  Counter += 1
Loop
```

C#

```
int counter = 0;

while (counter <= 10) {

  // Below we use a sneaky way to convert our int to a string
  lblMessage.Text = counter + "";

  // C# has the operator ++ to increase a variable by 1
  counter++;
}
```

You can try out this code—enter it inside a `Page_Load` subroutine of one of the pages you've already created. The page illustrating `Page_Load` at the start of this chapter would be ideal. Make sure you remove any other code in the subroutine, and that there is an ASP.NET `Label` control in the HTML of the page with the ID `lblMessage`. When you open the page, the label will be set to show the number 0, then 1, then 2, all the way to 10. Of course, since all this happens in `Page_Load` (i.e. before any output is sent to the browser), you'll only see the last value assigned, 10.

This demonstrates that the loop repeats until the condition is no longer met. Try changing the code so that the counter variable is initialized to 20 instead of 10. When you open the page now, you won't see *anything* on screen, because the loop condition was never met.

There is another form of the `While` loop, called a `Do While` loop, which checks if the condition has been met at the *end* of the code block, rather than at the beginning:

VB.NET

```
Dim Counter As Integer = 0

Do

  ' Convert our Integer to a String
  lblMessage.Text = Counter.toString()

  ' Below we use the += operator to increase our variable by 1
  Counter += 1
Loop While Counter <= 10
```

C#

```
int counter = 0;

do {

  // Below we use a sneaky way to convert our int to a string
  lblMessage.Text = counter + "";

  // C# has the operator ++ to increase a variable by 1
  counter++;
} while (counter <= 10);
```

If you run this code, you'll see it provides the exact same output we saw when we tested the condition before the code block. However, we can see the crucial difference if we again change it so the counter variable is initialized to 20. In this case, we will, in fact, see 20 on screen, because the loop code is executed once before the condition is even checked! There are some instances when this is just what we want, so being able to place the condition at the end of the loop can be very handy.

A `For` loop is similar to a `While` loop, but is typically used when the number of times we need it to execute is known beforehand. The following example displays the count of items within a `DropDownList` control called `ddlProducts`:

VB.NET

```
Dim i As Integer
For i = 1 To ddlProducts.Items.Count
  lblMessage.Text = i.toString()
Next
```

C#

```
int i;
for (i = 1; i <= ddlProducts.Items.Count; i++) {
  lblMessage.Text = Convert.ToString(i);
}
```

In VB.NET, the loop syntax specifies the starting and ending values for our counter variable in the `For` statement itself. In C#, we assign a starting value (`i = 1`), a condition to be tested each time through the loop, just like a `While` loop (`i <= ddlProducts.Items.Count`), and how the counter variable should be incremented after each loop (`i++`). While this allows for some powerful variations on the theme in C#, it can be confusing at first. In VB.NET, the syntax is considerably simpler, but can be a bit limiting in exceptional cases.

The other type of `For` loop is `For Each`, which loops through every item within a collection. The following example loops through an array called `arrayName`:

VB.NET

```
For Each item In arrayName
  lblMessage.Text = item
Next
```

C#

```
foreach (string item in arrayName) {
  lblMessage.Text = item;
}
```

You may also come across instances in which you need to exit a loop prematurely. In this case, you would use `Exit` (VB.NET) or `break` (C#) to terminate the loop:

VB.NET

```
Dim i As Integer
For i = 0 To 10
  If (i = 5) Then
    Response.Write("Oh no! Not the number 5!!")
    Exit For
  End If
Next
```

C#

```
int i;
for (i = 0; i <= 10; i++) {
  if (i == 5) {
    Response.Write("Oh no! Not the number 5!!");
    break;
```

```
  }
}
```

In this case, as soon as our `For` loop hits 5, it displays a warning message, using the `Response.Write()` method that will be familiar to those with past ASP experience, and exits the loop so that no further passes through the loop will be made.

Although we have only scratched the surface, VB.NET and C# provide a great deal of power and flexibility to the Web developer, and time spent learning the basics now will more than pay off in the future.

Understanding Namespaces

Because ASP.NET is part of the .NET Framework, we have access to all the goodies that are built into it in the form of the **.NET Framework Class Library**. This library represents a huge resource of tools and features in the form of **classes**, all organized in a hierarchy of **namespaces**. When we want to use certain features that .NET provides, we have only to find the namespace that contains that functionality, and **import** that namespace into our ASP.NET page. Once we've done that, we can make use of the .NET classes in that namespace to achieve our own ends.

For instance, if we wanted to access a database from a page, we would import the namespace that contains classes for this purpose, which happens to be the `System.Data.OleDb` namespace. The dots (`.`) here indicate different levels of the hierarchy I mentioned—in other words, the `System.Data.OleDb` namespace is grouped within the `System.Data` namespace, which in turn is contained in the `System` namespace.

To import a particular namespace into an ASP.NET page, we use the `Import` directive. Consider the following excerpt from an ASP.NET page; it imports the `System.Data.OleDb` namespace, which contains classes called `OleDbConnection`, `OleDbCommand`, and `OleDbDataReader`. Importing the namespace lets us use these classes in a subroutine to display records from an Access database:

VB.NET

```
<%@ Import Namespace="System.Data.OleDb" %>
<html>
<head>
<script runat="server" language="VB">
Sub ReadDatabase(s As Object, e As EventArgs)
  Dim objConn As New OleDbConnection( _
```

```
    "Provider=Microsoft.Jet.OLEDB.4.0;" & _
    "Data Source=C:\Database\books.mdb")
  Dim objCmd As New OleDbCommand("SELECT * FROM BookList", _
    objConn)
  Dim drBooks As OleDbDataReader

  objConn.Open()
  drBooks = objCmd.ExecuteReader()
  While drBooks.Read()
    Response.Write("<li>")
    Response.Write(drBooks("Title"))
  End While
  objConn.Close()
End Sub
</script>
</head>
```

C#

```
<%@ Import Namespace="System.Data.OleDb" %>
<html>
<head>
<script runat="server" language="C#">
void ReadDatabase(Object s, EventArgs e) {
  OleDbConnection objConn = new OleDbConnection(
    "Provider=Microsoft.Jet.OLEDB.4.0;" +
    "Data Source=C:\\Database\\books.mdb");
  OleDbCommand objCmd = new OleDbCommand("SELECT * FROM BookList",
    objConn);
  OleDbDataReader drBooks;
  objConn.Open();
  drBooks = objCmd.ExecuteReader();
  while (drBooks.Read()) {
    Response.Write("<li>");
    Response.Write(drBooks["Title"]);
  }
  objConn.Close();
}
</script>
</head>
```

Don't worry too much about the code right now (we cover this in detail in Chapter 6). Suffice it to say that, as we've imported that namespace, we have access to all the classes that it contains, and we can use them to get information from an Access database for display on our page.

Specifically, the classes from `System.Data.OleDb` that are used in the above code are:

`OleDbConnection`	Used for connecting to the database
`OleDbCommand`	Used for creating a statement of contents to read from the database.
`OleDbConnection`	Used for connecting to the database
`OleDbCommand`	Used for creating a statement of contents to read from the database
`OleDbDataReader`	Used for actually reading contents from database

Object Oriented Programming Concepts

VB.NET and C# are great programming languages because they offer a structured way of programming. By structured, I mean that code is separated into modules, where each module defines classes that can be imported and used in other modules. Both languages are relatively simple to get started with, yet offer features sophisticated enough for complex, large-scale enterprise applications.

The languages' ability to support more complex applications—their scalability—stems from the fact that both are **object oriented programming (OOP)** languages. But ask a seasoned developer what OOP really *is*, and they'll start throwing out buzzwords and catch phrases that are sure to confuse you—terms like **polymorphism**, **inheritance**, and **encapsulation**. In this section, I aim to explain the fundamentals of OOP and how good OOP style can help you develop better, more versatile Web applications down the road. This section will provide a basic OOP foundation angled towards the Web developer. In particular, we'll cover the following concepts:

- Objects
- Properties
- Methods
- Classes
- Scope

- Events
- Inheritance

Objects

In OOP, one thinks of programming problems in terms of objects, properties, and methods. The best way to get a handle on these terms is to consider a real world object and show how it might be represented in an OOP program. Many books use the example of a car to introduce OOP. I'll try to avoid that analogy and use something friendlier: my dog, an Australian Shepherd named Rayne.

Rayne is your average great, big, friendly, loving, playful mutt. You might describe him in terms of his physical properties: he's gray, white, brown, and black, stands roughly one and a half feet high, and is about three feet long. You might also describe some methods to make him do things: he sits when he hears the command "Sit", lies down when he hears the command "Lie down", and comes when his name is called.

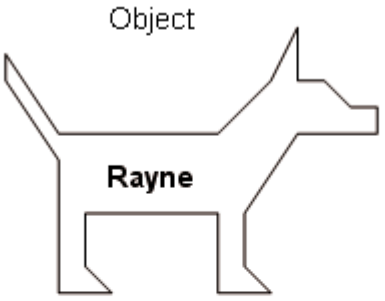
So, if we were to represent Rayne in an OOP program, we'd probably start by creating a class called `Dog`. A **class** describes how certain types of objects look from a programming point of view. When we define a class, we must define the following two things:

Properties — Properties hold specific information relevant to that class of object. You can think of properties as characteristics of the objects that they represent. Our `Dog` class might have properties such as `Color`, `Height`, and `Length`.

Methods — Methods are actions that objects of the class can be told to perform. Methods are subroutines (if they don't return a value) or functions (if they do) that are specific to a given class. So the `Dog` class could have methods such as `sit()`, and `lie_down()`.

Once we've defined a class, we can write code that creates **objects** of that class, using the class a little like a template. This means that objects of a particular class expose (or make available) the methods and properties defined by that class. So, we might create an **instance** of our `Dog` class called Rayne, set its properties accordingly, and use the methods defined by the class to interact with Rayne, as shown in Figure 3.3.

Figure 3.3. The methods defined by the class interact with the object.



Property values	Methods
Color: Gray, White, and Black	Sit
Eye Color: Blue and Brown	Lay Down
Height: 18 Inches	Shake
Length: 36 Inches	Come
Weight: 30 Pounds	

This is just a simple example to help you visualize what OOP is all about. In the next few sections, we'll cover properties and methods in greater detail, talk about classes and class instances, scope, events, and even inheritance.

Properties

As we've seen, properties are characteristics shared by all objects of a particular class. In the case of our example, the following properties might be used to describe any given dog:

- `Color`
- `Height`
- `Length`

In the same way, the more useful ASP.NET `Button` class exposes properties including:

- `Width`
- `Height`
- `ID`
- `Text`
- `ForeColor`
- `BackColor`

Unfortunately for me, if I get sick of Rayne's color, I can't change it. ASP.NET objects, on the other hand, let us change their properties very easily in the same way that we set variables. For instance, we've already used properties when setting text for the `Label` control, which is actually an object of class `Label` in the namespace `System.Web.UI.WebControls`:

VB.NET
```
lblMyText.Text = "Hello World"
```

C#
```
lblMyText.Text = "Hello World";
```

In this example, we're using a `Label` control called `lblMyText`. Remember, ASP.NET is all about controls, and, as it's built on OOP, all control types are represented as classes. In fact, as you'll learn in Chapter 4, all interaction with ASP.NET pages is handled via controls. When we place a control on a page, we give it a name through its `id` attribute, and this ID then serves as the name of the control. Rayne is an object. His name, or ID, is Rayne. Rayne has a height of eighteen inches. The same holds true for the `Label` control. The `Label` control's name or ID in the previous example is `lblMyText`. Next, we use the **dot operator** (`.`) to access the property `Text` that the object exposes and set it to the string "Hello World."

Methods

With our dog example, we can make a particular dog do things by calling commands. If I want Rayne to sit, I tell him to sit. If I want Rayne to lie down, I tell him to lie down. In object oriented terms, I tell him what I want him to do by calling a predefined command or **method**, and a resulting action is performed. In VB.NET or C#, we would write this as `rayne.Sit()`, or `rayne.LieDown()`.

As Web developers, we frequently call methods when a given event occurs. For instance, the example earlier in this chapter that took information from an Access database created an object called `objConn` to represent the connection to the database. We then opened the connection by calling the `Open()` method on that object as follows:

VB.NET

```
Dim objConn As As New OleDbConnection(
  "Provider=Microsoft.Jet.OLEDB.4.0;" & _
  "Data Source=C:\Database\books.mdb")
…
objConn.Open()
```

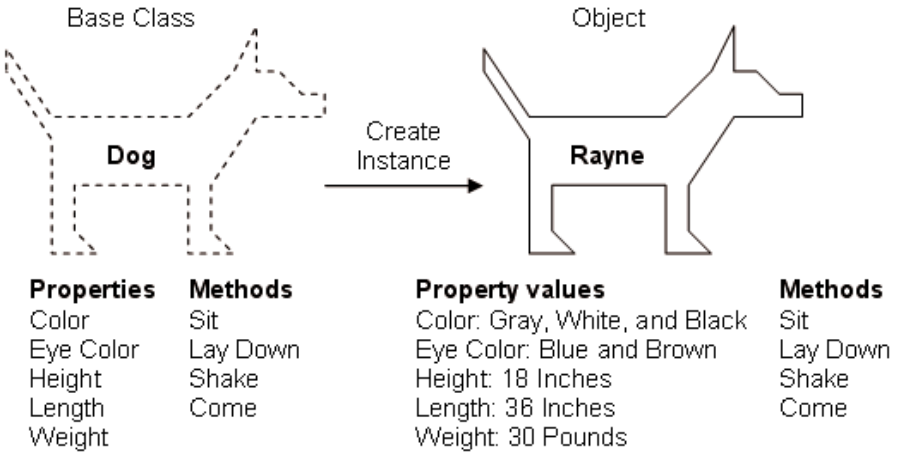
We say that the `Open()` method is **exposed** by the connection object, and that we're calling the `Open()` method on the `OleDbConnection` object stored in `objConn`. We don't need to know what dark secrets the method uses to do its magic; all we need to know is its name and what we use it for.

Classes

You can think of a class as a template for building as many objects as you like of a particular type. When you create an instance of a class, you are creating an object of that class, and the new object has all the characteristics and behaviors (properties and methods) defined by the class.

In our dog example, Rayne was an instance of the `Dog` class as shown in Figure 3.4.

Figure 3.4. A class serves as the blueprint for an object.



We see that Rayne is an object of class `Dog`. In our code, we could create a new instance of the `Dog` class, call it `rayne`, and use all the properties and methods exposed by the object.

In OOP, when we create new instances of a class, we say we're **instantiating** that class. For instance (no pun intended!), if we need to programmatically create a new instance of the `Button` control class, we could write the following code:

VB.NET

```
Dim myButton As New Button()
```

C#

```
Button myButton = new Button();
```

As you can see, we've essentially created a new object called `myButton` from the `Button` class. We can then access the properties and methods that the `Button` exposes through our new instance:

VB.NET

```
myButton.Text = "Click Me!"
```

C#

```
myButton.Text = "Click Me!";
```

Scope

You should now have a concept of programming objects as entities that exist in a program and are manipulated through the methods and properties they expose. However, in some cases, we want to create methods to use inside our class, which are not available when that class is used in code. Let's return to the `Dog` class to illustrate.

Imagine we're writing the `Sit()` method inside this class, and we realize that before the dog can sit, it has to shuffle its back paws forward a little (Just bear with me on this one…)! We could create a method called `ShufflePaws()`, then call that method from inside the `Sit()` method. However, we don't want code in an ASP.NET page or in some other class to call this method—it'd just be silly. We can prevent this by controlling the **scope** of that method.

The two types of scope available in VB.NET and C# that you should know about are:

`Public` Defining a property or method of a class as public allows that property or method to be called from outside the class itself. In other words, if an instance of this class is created inside another object (remember, too, that ASP.NET pages themselves are objects), public methods and properties are freely available to the code that created it. This is the default scope in VB.NET and C# classes.

`Private` If a property or method of a class is private, it cannot be used from outside the class itself. If an instance of this class is created inside an object of a different class, the creating object has no access to private methods or properties of the created object.

Events

We've covered **events** fairly well already. To sum up, events occur when a control object sends a message in response to some change that has happened to it. Generally, these changes occur as the result of user interaction with the control in the browser. For instance, when a button is clicked, a `Click` event is raised, and we can handle that event to perform some action. The object that **triggers** the event is referred to as the **event sender**, while the object that receives the event is referred to as the **event receiver**. You'll learn more about these terms in Chapter 4.

Understanding Inheritance

The term **inheritance** refers to the ability for one class to use properties and methods exposed by another class.

In our dog example, we first created a class called `Dog`, then created instances of that class to represent individual dogs such as Rayne. However, dogs are types of animals, and many characteristics of dogs are shared by all (or most) animals. For instance, Rayne has four legs, two ears, one nose, two eyes, etc. It might be better, then, for us to create a **base class** called `Animal`. When we then define the `Dog` class, it would **inherit** from the `Animal` class, and all public properties and methods of `Animal` would be available to instances of the `Dog` class.

Similarly, we could create a new class based on the `Dog` class. In programming circles, this is called **deriving a subclass** from `Dog`. For instance, we might create a class for Australian Shepherd and one for my other dog Amigo, called Chihuahua, both of which would **inherit** the properties and methods of the `Dog` base class, and define new ones specific to each breed.

Don't worry too much if this is still a little unclear. The best way to appreciate inheritance is to see it used in a real program. The most obvious use of inheritance in ASP.NET comes with the technique of code-behind.

Separating Code From Content With Code-Behind

Most companies that employ development teams usually split projects into two groups, visual design and functional development, because software engineers are usually poor designers, and designers are often poor engineers. Until now, our ASP.NET pages have contained code render blocks that place VB.NET or C# code directly within the ASP.NET page. The problem with this approach is that there is no separation between the presentational elements of the page and the application logic. Traditional ASP was infamous for creating "spaghetti" code, which was scattered and intertwined throughout the presentation elements. This made it very tricky to manage the code between development teams, as you'll know if you've ever tried to pick apart someone else's ASP code. In response to these problems, ASP.NET introduces a new way of developing pages that allows code developers to work separately from the presentational designers who lay out individual pages.

This new method, called **code-behind**, keeps all of your presentational elements (controls) inside the `.aspx` file, but moves all your code to a separate class in a `.vb` or `.cs` code-behind file. Consider the following ASP.NET page, which displays a simple button and label:

VB.NET

```
<html>
<head>
  <title>Sample Page using VB.NET</title>
  <script runat="server" language="VB">
    Sub Click(s As Object, e As EventArgs)
      lblMessage.Text = "Hello World"
    End Sub
  </script>
</head>

<body>
  <form runat="server">
    <asp:Button id="btnSubmit" Text="Click Me" runat="server"
      OnClick="Click" />
    <br /><br /><asp:Label id="lblMessage" runat="server" />
  </form>
</body>
</html>
```

C#

```
<html>
<head>
  <title>Sample Page using C#</title>
  <script runat="server" language="C#">
    void Click(Object s, EventArgs e) {
      lblMessage.Text = "Hello World";
    }
  </script>
</head>

<body>
  <form runat="server">
    <asp:Button id="btnSubmit" Text="Click Me" runat="server"
      OnClick="Click" />
    <br /><br /><asp:Label id="lblMessage" runat="server" />
  </form>
</body>
</html>
```

Let's see how this example could be separated into the following two distinct files:

sample.aspx	layout, presentation, and static content
sample.vb **sample.cs**	code-behind files containing a custom page class

First, we take all the code and place it in the code-behind file (`sample.vb` or `sample.cs`). This file is a pure code file, and contains no HTML or other markup tags. What it *does* contain is a class definition. Nevertheless, we can still access presentation elements from this file, using their IDs, such as `lblMessage`:

VB.NET — File: **sample.vb**

```
' First off we import some useful namespaces
Imports System
Imports System.Web.UI
Imports System.Web.UI.WebControls

' All code-behind classes generally inherit from Page
Public Class Sample
  Inherits Page

  ' Declare the presentation elements on the ASPX page
  Protected WithEvents lblMessage As Label
  Protected WithEvents btnSubmit As Button

  ' Here's the Click handler just as it appeared before
  Sub Click(s As Object, e As EventArgs)
    lblMessage.Text = "Hello World"
  End Sub
End Class
```

C# — File: **sample.cs**

```
// First off we import some useful namespaces
using System;
using System.Web.UI;
using System.Web.UI.WebControls;

// All code-behind classes generally inherit from Page
public class Sample : Page
{
  // Declare the presentation elements on the ASPX page
  protected Label lblMessage;
  protected Button btnSubmit;
```

```
  // Here's the Click handler just as it appeared before
  public void Click(Object s, EventArgs e) {
    lblMessage.Text = "Hello World";
  }
}
```

Without code, the main ASP.NET page becomes much simpler:

VB.NET File: **sample.aspx**

```
<%@ Page Inherits="Sample" Src="Sample.vb" %>
<html>
<head>
  <title>Sample Page using VB.NET</title>
</head>

<body>
  <form runat="server">
    <asp:Button id="btnSubmit" Text="Click Me" runat="server"
      OnClick="Click" />
    <br /><br /><asp:Label id="lblMessage" runat="server" />
  </form>
</body>
</html>
```

C# File: **sample.aspx**

```
<%@ Page Inherits="Sample" Src="Sample.cs" %>
<html>
<head>
  <title>Sample Page using C#</title>
</head>

<body>
  <form runat="server">
  <asp:Button id="btnSubmit" Text="Click Me" runat="server"
    OnClick="Click" />
  <br /><br /><asp:Label id="lblMessage" runat="server" />
  </form>
</body>
</html>
```

As you can see, the only line that's different between the `.aspx` pages is the `Page` directive:

VB.NET File: **sample.aspx (excerpt)**

```
<%@ Page Inherits="Sample" Src="Sample.vb" %>
```

C# File: **sample.aspx (excerpt)**

```
<%@ Page Inherits="Sample" Src="Sample.cs" %>
```

The only real change between the VB.NET and C# versions of the page is the source filename extension. In both cases, the page **inherits** from the class `Sample`.

The code-behind file is written differently from what you're used to seeing so far. While we no longer need `<script>` tags, we find a class definition in its place. Looking at the VB.NET example, we start with three lines that import namespaces to be used in the code:

VB.NET File: **sample.aspx (excerpt)**

```
Imports System
Imports System.Web.UI
Imports System.Web.UI.WebControls
```

The next lines create a new class, named `Sample`. Because our code-behind page contains code for an ASP.NET page, our class inherits from the `Page` class:

VB.NET File: **sample.aspx (excerpt)**

```
Public Class Sample
  Inherits Page
```

This is the practical application of inheritance that I mentioned above. Instead of using the built-in `Page` class, the code-behind method has you derive a subclass of `Page` for each page in your site. Next, we have to declare the controls that we want to use from the `.aspx` page—if we forget this step, we won't be able to access them from our code:

VB.NET File: **sample.aspx (excerpt)**

```
  Protected WithEvents lblMessage As Label
  Protected WithEvents btnSubmit As Button
```

Finally, we create the `Click` subroutine just as before, and terminate the class:

VB.NET File: **sample.aspx (excerpt)**

```
  Sub Click(s As Object, e As EventArgs)
    lblMessage.Text = "Hello World"
  End Sub
End Class
```

As I hope you can see, code-behind files are reasonably easy to work with, and they can make managing and using our pages much more straightforward. On a typical project, I tend to use code-behind files quite frequently, but for simplicity's sake, we'll stick with code declaration blocks for at least the next few chapters.

Summary

Phew! That's quite a few concepts to understand over the course of a single chapter. Don't worry—with a little practice, these concepts will become second nature. I hope you leave this chapter with a basic understanding of programming concepts as they relate to the Web developer. The next chapter will begin to put all the concepts that we've covered so far into practice, beginning by covering HTML Controls, Web Forms, and Web Controls, before launching into our first hands-on project.

4

Web Forms and Web Controls

At the heart of ASP.NET is its ability to create dynamic form content. Whether you're creating a complex shopping cart application, or a simple page to collect user information and send the results out via email, Web Forms have a solution. They allow you to use HTML controls and Web controls to create dynamic pages with which users can interact. In this chapter, you will learn how Web Forms, HTML controls, and Web controls, in conjunction with VB.NET and C# code, should change the way you look at, and develop for, the Web. In this chapter I'll introduce you to the following concepts:

- HTML controls
- Web Forms
- Web controls
- Handling page navigation
- Formatting controls with CSS

Toward the end of the chapter, you'll put all of these concepts to work into a real world application! I'll introduce the Dorknozzle Intranet Application that you'll be building throughout this book, and see how what you learned in this chapter can be applied to some of the pages for the project.

Working with HTML Controls

HTML controls are outwardly identical to plain old HTML 4.0 tags, but employ the `runat="server"` attribute. For each of HTML's most common tags, a corresponding server-side HTML control exists, although Microsoft has added a few tags and some extra properties for each. Creating HTML controls is easy—we simply stick a `runat="server"` attribute on the end of a normal HTML tag to create the HTML control version of that tag. The complete list of current HTML control classes and their associated tags is given in Table 4.1.

These HTML control classes are all contained within the `System.Web.UI.HtmlControls` namespace.

Because HTML controls are processed on the server side by the ASP.NET runtime, we can easily access their properties through code elsewhere in the page. If you're familiar with JavaScript, HTML, and CSS, then you'll know that manipulating text within HTML tags, or even manipulating inline styles within an HTML tag, can be cumbersome and error-prone. HTML controls aim to solve this by allowing you to manipulate the page easily with your choice of .NET language, for instance, using VB.NET or C#. We'll start by looking at the HTML controls library, then we'll explore in more detail the properties exposed by the controls when we process a simple form containing HTML controls and code.

Table 4.1. HTML Control Classes

Class	Associated Tags
HtmlAnchor	<a href="…" runat="server">
HtmlButton	<button runat="server">
HtmlForm	<form runat="server">
HtmlImage	<img runat="server">
HtmlInputButton	<input type="submit" runat="server"> <input type="reset" runat="server"> <input type="button" runat="server">
HtmlInputCheckBox	<input type="checkbox" runat="server">
HtmlInputFile	<input type="file" runat="server">
HtmlInputHidden	<input type="hidden" runat="server">
HtmlInputImage	<input type="image" runat="server">
HtmlInputRadioButton	<input type="radio" runat="server">
HtmlInputText	<input type="text" runat="server">
HtmlSelect	<select runat="server">
HtmlTable	<table runat="server">
HtmlTableRow	<tr runat="server">
HtmlTableCell	<td runat="server"> <th runat="server">
HtmlTextArea	<textarea runat="server">
HtmlGenericControl	All other HTML tags, including <body runat="server"> <font runat="server">

HtmlAnchor

The HtmlAnchor control creates a server-side HTML <a href="…"> tag.

```
<a href="somepage.aspx" runat="server">Click Here</a>
```

This line would create a new hyperlink with the text "Click Here." Once the link is clicked, the user would be redirected to `somepage.aspx` as given by the `href` attribute.

HtmlButton

The `HtmlButton` control creates a server-side HTML `<button>` tag.

```
<button id="myButton" OnServerClick="Click" runat="server">Click
Here</button>
```

Notice that we're using events here. On HTML controls, we need to use `OnServerClick` to specify the ASP.NET handler for clicks on the button, because `onclick` is reserved for handling clicks with JavaScript on the client side. In this example, the handler subroutine is called `Click`, and would be declared in a script block with the same form as the `Click` handlers we looked at for `<asp:Button>` tags previously:

VB.NET

```
<script runat="server" language="VB">
Sub Click(s As Object, e As EventArgs)
  Response.Write(myButton.ID)
End Sub
</script>
```

C#

```
<script runat="server" language="C#">
void Click(Object s, EventArgs e) {
  Response.Write(myButton.ID);
}
</script>
```

In this case, when the user clicks the button, the `ServerClick` event is raised, the `Click()` subroutine is called to handle it, and the ID of the `HtmlButton` control is written onto the screen with `Response.Write()` (the `Write()` method of the `Response` object).

HtmlForm

The `HtmlForm` control creates a server-side `<form>` tag. Most HTML controls, Web controls, etc., must be placed inside an `HtmlForm` control.

```
<form runat="server">
<!-- ASP.NET controls in here -->
</form>
```

HtmlImage

The `HtmlImage` control creates a server-side `<img>` tag. The following code shows how we might place an `HtmlImage` control on a page, along with an `HtmlButton`:

```
<img id="myimage" src="arrow.gif" runat="server" />
<button id="myButton" runat="server" OnServerClick="Click">Click
Here</button>
```

The user could change this image dynamically by pressing the button if we add code as follows:

VB.NET

```
<script runat="server" language="VB">
Sub Click(s As Object, e As EventArgs)
  myimage.Src = "welcome.gif"
End Sub
</script>
```

C#

```
<script runat="server" language="C#">
void Click(Object s, EventArgs e) {
  myimage.Src = "welcome.gif";
}
</script>
```

What will happen if these controls are placed on a page along with the script block? First of all, the image `arrow.gif` will appear. When the `HtmlButton` control is clicked, it changes to `welcome.gif`. Behind the scenes, the `ServerClick` event is raised when the button is clicked, thus the `Click()` subroutine is called, and the `Src` property of the `HtmlImage` control is changed from `arrow.gif` to `welcome.gif`.

HtmlGenericControl

The `HtmlGenericControl` creates a server-side control for HTML tags that do not have an HTML control associated with them. Perfect examples of this are the `<span>` and `<div>` tags. The following example illustrates how you can

modify text within a `<span>` tag to change the content from `I like ASP.NET` to `Why would anyone need PHP?` dynamically.

```
<span id="myGenericControl" runat="server">I like ASP.NET</span>
<br />
<button id="myButton" runat="server" OnServerClick="Click">Click
Here</button>
```

We simply add the following code to respond to the `ServerClick` event and change the text:

VB.NET

```
<script runat="server" language="VB">
Sub Click(s As Object, e As EventArgs)
  myGenericControl.InnerText = "Why would anyone need PHP?"
End Sub
</script>
```

C#

```
<script runat="server" language="C#">
void Click(Object s, EventArgs e) {
  myGenericControl.InnerText = "Why would anyone need PHP?";
}
</script>
```

HtmlInputButton

The `HtmlInputButton` control creates a server-side `<input type="submit">`, `<input type="reset">`, or `<input type="button">` HTML tag.

```
<input type="submit" value="Click Here" runat="server" />
```

```
<input type="reset" value="Click Here" runat="server" />
```

```
<input type="button" value="Click Here" runat="server" />
```

As with `HtmlButton`, you can assign a server-side event handler to controls of this type with the `OnServerClick` attribute.

HtmlInputCheckBox

The `HtmlInputCheckBox` control creates a server-side `<input type="checkbox">` HTML tag.

```
<input type="checkbox" id="cb1" value="ASP.NET" runat="server"
  />ASP.NET<br />
<input type="checkbox" id="cb2" value="PHP" runat="server"
  />PHP<br />
<input type="checkbox" id="cb3" value="JSP" runat="server"
  />JSP<br />
<input type="checkbox" id="cb4" value="CGI" runat="server"
  />CGI<br />
<input type="checkbox" id="cb5" value="Coldfusion" runat="server"
  />Coldfusion<br>
```

The `HtmlInputCheckBox` control is the perfect choice when you want to allow your users to select multiple items from a list.

HtmlInputFile

The `HtmlInputFile` control creates a server-side `<input type="file">` tag in the HTML. This displays a text box and Browse button to allow users to upload files from ASP.NET pages. There is no Web control equivalent for this tag, so it's typically required when working with file uploads—even with Web Forms (which we'll discuss shortly).

```
<input type="file" id="fileUpload" runat="server" />
```

HtmlInputHidden

The `HtmlInputHidden` control creates a server-side `<input type="hidden">` tag.

```
<input type="hidden" id="hiddenField" runat="server" />
```

Try viewing the source of any one of your ASP.NET pages from your browser, and you're likely to find this tag being used to store view state information.

HtmlInputImage

The `HtmlInputImage` control creates a server-side `<input type="image">` tag.

```
<input type="image" id="imgMap" runat="server"
  src="ButtonImage.jpg" />
```

This tag provides an alternative to the `HtmlInputButton` control. They both function in the same way; the difference is that the `HtmlInputImage` control uses a custom image rather than the beveled gray Windows-style button. The mouse

coordinates are also sent along with the form submission when the user clicks a control of this type.

HtmlInputRadioButton

The `HtmlInputRadioButton` control creates a server-side radio button. The following code, for instance, offers a choice of Male or Female:

```
Gender?<br />
<input type="radio" id="radio1" runat="server" />Male<br />
<input type="radio" id="radio2" runat="server" />Female
```

Similar to the `HtmlInputCheckBox` control, the `HtmlInputRadioButton` control creates a list of items for users to choose from. The difference, however, is that the user is only able to select one item at a time.

HtmlInputText

The `HtmlInputText` control creates a server-side `<input type="text">` or `<input type="password">` tag.

```
Please Login<br />
Username:<br />
<input type="text" id="username" runat="server" /><br />
Password:<br />
<input type="password" id="password" runat="server" />
```

The preceding code creates a typical login screen layout.

HtmlSelect

The `HtmlSelect` control creates a server-side version of the `<select>` tag for creating drop-down lists or list boxes. The following code creates a drop-down menu:

```
Select your favorite movie:<br />
<select id="selectMovie" runat="server">
<option>Star Wars</option>
<option>Spider Man</option>
<option>The Godfather</option>
<option>Lord of the Rings</option>
</select>
```

The following code creates a multiple-selection list box:

```
Which of these movies do you like?<br />
<select id="selectMovie" runat="server" multiple="true" size="4">
<option>Star Wars</option>
<option>Spider Man</option>
<option>The Godfather</option>
<option>Lord of the Rings</option>
</select>
```

You'll notice the `<option>` tag within the main `<select>` tag; this is used to denote each item to appear in the list box or drop-down menu.

HtmlTable, HtmlTableRow and HtmlTableCell

The `HtmlTable`, `HtmlTableRow`, and `HtmlTableCell` controls create server-side versions of the `<table>`, `<tr>`, `<td>`, and `<th>` tags. The following code creates a server-side table:

```
<table id="myTable" border="1" cellspacing="0" cellpadding="0"
runat="server">
<tr runat="server" id="row1">
<td runat="server" id="cell1">Table Data 1</td>
<td runat="server" id="cell2">Table Data 2</td>
</tr>
<tr runat="server" id="row2">
<td runat="server" id="cell3">Table Data 3</td>
<td runat="server" id="cell4">Table Data 4</td>
</tr>
</table>
<button id="myButton" OnServerClick="Click" runat="server">Click
Here</button>
```

You could add the following code to respond to the `Click` event raised by the `HtmlButton` control and change the content of the first cell to read "Hello World."

VB.NET

```
<script runat="server" language="VB">
Sub Click(s As Object, e As EventArgs)
  cell1.InnerText = "Hello World"
End Sub
</script>
```

C#

```
<script runat="server" language="C#">
void Click(Object s, EventArgs e) {
  cell1.InnerText = "Hello World";
}
</script>
```

HtmlTextArea

The `HtmlTextArea` control creates a server-side version of the `<textarea>` tag.

```
<textarea cols="60" rows="10" runat="server"></textarea>
```

We've glanced only briefly over the HTML controls, as they should all be fairly familiar from your experience with HTML. But if you'd like more information on the HTML controls including the properties, methods, and events for each, see Appendix A.

Processing a Simple Form

Now that you have a basic understanding of ASP.NET page structure, the languages VB.NET and C#, and HTML controls, let's put everything together and create a simple ASP.NET application. The application that we will create, in VB.NET and C#, will be a simple survey form that uses the following HTML controls:

- `HtmlForm`
- `HtmlButton`
- `HtmlInputText`
- `HtmlSelect`

Let's begin by creating a new file within your favorite code editor. The following code creates the visual interface for the survey:

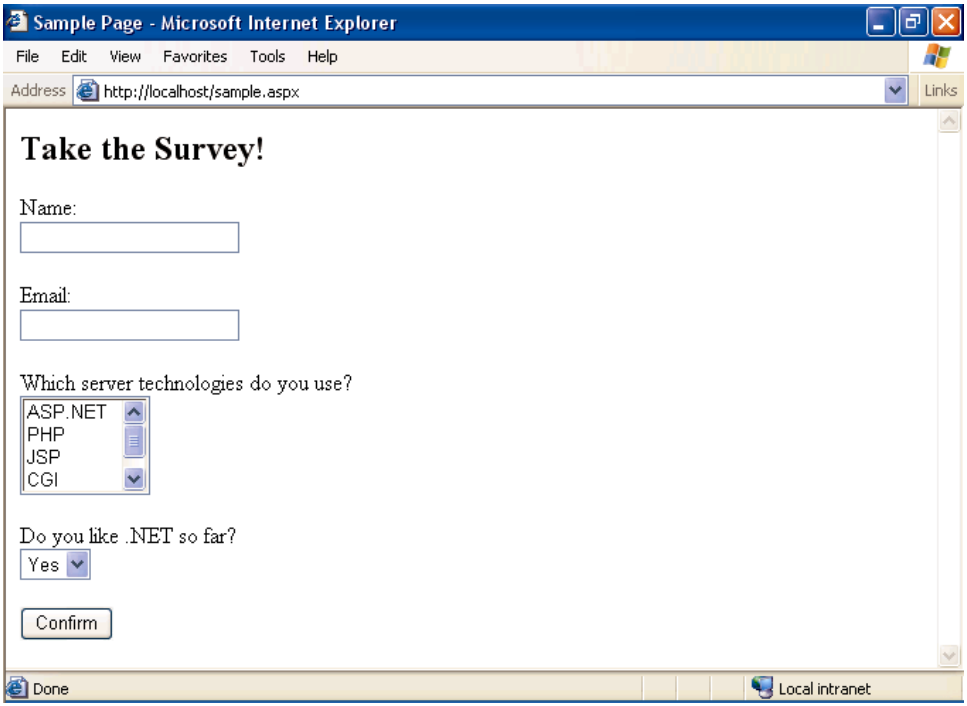
File: **`SimpleForm.aspx` (excerpt)**

```
<html>
<head>
…
</head>
```

```
<body>
<form runat="server">
  <h2>Take the Survey!</h2>
  <p>Name:<br />
  <input type="text" id="txtName" runat="server" /></p>
  <p>Email:<br />
  <input type="text" id="txtEmail" runat="server" /></p>
  <p>Which server technologies do you use?<br />
  <select id="servermodel" runat="server" multiple="true">
    <option>ASP.NET</option>
    <option>PHP</option>
    <option>JSP</option>
    <option>CGI</option>
    <option>Coldfusion</option>
  </select></p>
  <p>Do you like .NET so far?<br />
  <select id="likedotnet" runat="server">
    <option selected>Yes</option>
    <option>No</option>
  </select></p>
  <p><button id="myButton" OnServerClick="Click" runat="server">
  Confirm</button></p>
</form>
</body>
</html>
```

From what we've already covered on HTML controls, you should have a good idea of what this page will look like. All we've done is place some `HtmlInputText` controls, an `HtmlButton` control, and an `HtmlSelect` control inside the obligatory `HtmlForm` control. Remember, HTML controls are essentially just HTML tags with the `runat="server"` attribute. When it's complete, the interface will resemble Figure 4.1.

Figure 4.1. Create the interface of the ASP.NET page using HTML controls.



When users click the button, we'll simply display their responses in their browsers. In a real application, we'd probably be more likely to save this to a database and perhaps show the results as a chart. Whatever the case, we'd access the properties of the HTML controls as shown in the following code:

VB.NET File: **SimpleForm.aspx (excerpt)**

```
<script runat="server" language="VB">
Sub Click(s As Object, e As EventArgs)
  Response.Write("Your name is: " & txtName.value & "<br />")
  Response.Write("Your email is: " & txtEmail.value & "<br />")
  Response.Write("You like to work with: " & servermodel.value & _
    "<br />")
  Response.Write("You like .NET: " & likedotnet.value)
End Sub
</script>
```

C# File: **SimpleForm.aspx (excerpt)**

```
<script runat="server" language="C#">
void Click(Object s, EventArgs e) {
  Response.Write("Your name is: " + txtName.Value + "<br />");
  Response.Write("Your email is: " + txtEmail.Value + "<br />");
  Response.Write("You like to work with: " + servermodel.Value +
    "<br />");
  Response.Write("You like .NET: " + likedotnet.Value);
}
</script>
```

Just as you've seen with examples from previous chapters, we place our VB.NET and C# code inside a server-side script block within the `<head>` part of the page. Next, we create a new `Click` event handler which takes the two usual parameters. Finally, we use the `Response` object's `Write()` method to print out the user's responses within the page.

Once you've written the code, you can save your work and test the results from your browser. Enter some information and click the button. What you type in should appear at the top of the page when the button is clicked.

Introduction to Web Forms

With the inception of new technologies, there's always new terminology to master. ASP.NET is no different. With ASP.NET, even the simplest terms that were previously used to describe a Web page have changed to reflect the processes that occur within them. Before we begin to describe the process followed by Web Forms, let's discuss the foundation concept of Web pages.

On the most basic level, a **Web page** is a text file that contains markup. Web pages are meant to be viewed from a browser window, which parses the file containing markup to present the information to the user in the layout envisaged by the developer. Web pages can include text, video, sound, animations, graphics, and even chunks of "code" from a variety of technologies.

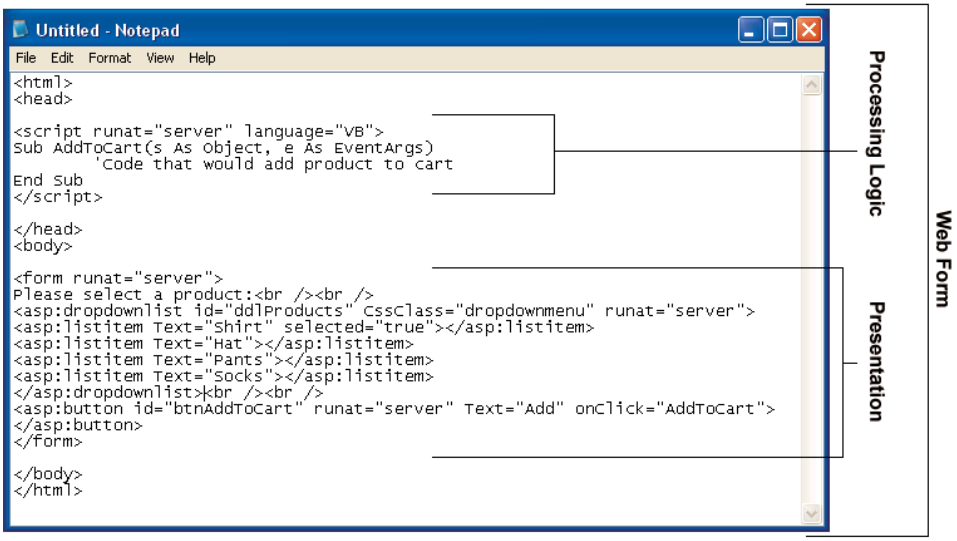
An **HTML form**, as you learned in the previous sections, is a page that contains one or more form elements grouped together within an HTML `<form>` tag. Users interact with the various form elements to make certain choices, or provide certain information; this information is then sent to the server for processing upon the click of a submit button. This is useful to us as ASP.NET developers because regular HTML forms have a built-in mechanism that allows forms to be submitted to the server. Once the form has been submitted, some kind of extra techno-

logy—in this case, ASP.NET—needs to be present on the server to perform the actual form processing.

In ASP.NET, we call Web pages **Web Forms**; they contain presentational elements (ASP.NET Web controls) in an HTML form, as well as any code (the processing logic) we've added for the page's dynamic features.

A typical Web Form is shown in Figure 4.2:

Figure 4.2. A Web Form contains code for processing logic and Web controls for presentational purposes.



The next section looks at the various Web controls and how they may be used within your Web Forms. They're very similar in appearance to HTML, so you shouldn't have any trouble coming to grips with them.

Introduction to Web Controls

As we've just seen, Web Forms allow users to interact with our site using **Web controls**. With Web controls, Microsoft basically reinvented HTML from scratch. For example, it created two different Web controls that correspond to the two different versions of the HTML `<select>` tag: a `DropDownList` control and a `ListBox` control. This means there isn't a direct one-to-one correspondence

between the Web controls and standard HTML tags, as there is with HTML controls. Web controls follow the same basic pattern as HTML tags, but the tag name is preceded by `asp:` and the name is capitalized using "CamelCasing." Consider the HTML `<input>` tag, which creates an input text box on screen:

```
<input type="text" name="username" size="30" />
```

The equivalent Web control is the `TextBox` control, and it would look like this:

```
<asp:TextBox id="username" Columns="30" runat="server">
</asp:TextBox>
```

Note that, unlike many HTML tags, Web controls always require a closing tag (the `</asp:TextBox>` part above). We can also use the shorthand `/>` syntax if our Web control tag doesn't contain anything between its opening and closing tags. So, we could also write this `TextBox` like so:

```
<asp:TextBox id="username" Columns="30" runat="server" />
```

To sum up, the key points to remember when working with Web controls are:

- All Web controls must be placed within a `<form runat="server">` tag to function properly.
- All Web controls require `id` and `runat="server"` properties to function properly.
- All Web controls follow the same pattern, but different properties (attributes) are available to different controls.
- They all start with the `asp` prefix, followed by a colon.

There are more Web controls than HTML controls, and some offer advanced features that simply aren't available in HTML alone. Controls that we'll discuss in this and future chapters are as follows:

- basic Web controls (Chapter 4)
- validation Web controls (Chapter 5)
- data controls (Chapter 9)
- user controls (Chapter 16)

- rich controls (Chapter 16)

Basic Web Controls

The basic Web controls perform the on-screen layout of a Web page, and mirror in many ways the HTML controls that are based on regular HTML. However, they offer some new refinements and enhancements, and should be used in place of HTML whenever possible. In this section, we'll look at the controls in this group, namely:

- `Label`
- `TextBox`
- `Button`
- `Image`
- `ImageButton`
- `LinkButton`
- `HyperLink`
- `RadioButton`
- `RadioButtonList`
- `CheckBox`
- `CheckBoxList`
- `DropDownList`
- `ListBox`
- `Panel`
- `PlaceHolder`

Label

The easiest way to display static text on your page is simply to add the text to the body of the page without enclosing it in any tag. However, if you want to modify the text displayed on a page from ASP.NET code, you can display your text within a `Label` control. Here's a typical example:

```
<asp:Label id="lblMessage" Text="" runat="server" />
```

The following code sets the `Text` property of the `Label` control to display the text "Hello World":

VB.NET

```
Public Sub Page_Load()
  lblMessage.Text = "Hello World"
End Sub
```

C#

```
public void Page_Load() {
  lblMessage.Text = "Hello World";
}
```

Reading this `Page_Load()` handler code, we can see that when the page first loads, the `Text` property of the `Label` control with the ID of `lblMessage` will be set to "Hello World."

TextBox

The `TextBox` control is used to create on screen a box in which the user can type or read standard text. This Web control can be set to display a standard HTML text input field, an HTML password field, or an HTML text area, using the `TextMode` property. The following code shows how we might use it in a simple login page:

```
<p>Username:
<asp:TextBox id="txtUser" TextMode="SingleLine" Columns="30"
    runat="server" /></p>

<p>Password:
<asp:TextBox id="txtPassword" TextMode="Password" Columns="30"
    runat="server" /></p>

<p>Comments:
```

```
<asp:TextBox id="txtComments" TextMode="MultiLine" Columns="30"
    Rows="10" runat="server" /></p>
```

In each of the three instances above, the attribute `TextMode` dictates the kind of text box to render.

Button

By default, the `Button` control renders the same form submit button that's rendered by the HTML `<input type="Submit">` tag. When a button is clicked, the form containing the button is submitted to the server for processing, and both click and command events are raised. The following code displays a `Button` control and a `Label`:

```
<asp:Button id="btnSubmit" Text="Submit" runat="server"
    OnClick="WriteText" />
<asp:Label id="lblMessage" runat="server" />
```

Notice the `OnClick` attribute on the control. Unlike the `HtmlButton` HTML control, `OnClick` assigns a *server-side* event handler—there is no need to remember to use `OnServerClick`. When the button is clicked, the `Click` event is raised and the `WriteText()` subroutine is called. The `WriteText()` subroutine will contain the code that performs the intended function for this button, such as displaying a message for the user:

VB.NET

```
Public Sub WriteText(s As Object, e As EventArgs)
  lblMessage.Text = "Hello World"
End Sub
```

C#

```
public void WriteText(Object s, EventArgs e) {
  lblMessage.Text = "Hello World";
}
```

It's important to realize that most Web controls have events associated with them, and the basic idea and techniques are the same as for the `Click` event of the `Button` control.

Image

An `Image` control places on the page an image that can be accessed dynamically from code; it equates to the `<img>` tag in HTML. Here's an example:

```
<asp:Image id="myImage" ImageUrl="mygif.gif" runat="server"
    AlternateText="description" />
```

ImageButton

An `ImageButton` control is similar to a `Button` control, but it uses an image you supply in place of the typical gray Windows-style button. For example:

```
<asp:ImageButton id="myImgButton" ImageUrl="myButton.gif"
    runat="server" />
```

LinkButton

A `LinkButton` control renders a hyperlink on your page. From the point of view of ASP.NET code, `LinkButton`s can be treated in much the same way as buttons, hence the name.

```
<asp:LinkButton id="myLinkButon" Text="Click Here" runat="server"
    />
```

HyperLink

The `HyperLink` control, which is similar to the `LinkButton` control, creates a hyperlink on your page. It's simpler and faster to process than `LinkButton`, but, unlike the `LinkButton` control, which offers features such as `Click` events and validation, `HyperLink` can be used only to click and navigate from one page to the next.

```
<asp:HyperLink id="myLink" NavigateUrl="http://www.example.com/"
    ImageUrl="myButton.gif" runat="server">My Link</asp:HyperLink>
```

The `ImageUrl` attribute, if specified, causes the control to display a linked image instead of the text provided.

RadioButton

You can add individual radio buttons to your page one by one, using the `RadioButton` control. Radio buttons are grouped together using the `GroupName` property. Only one `RadioButton` control from each group can be selected at a time.

```
<asp:RadioButton id="radSanDiego" GroupName="City"
    Text="San Diego" runat="server" />
```

```
<asp:RadioButton id="radBoston" GroupName="City" Text="Boston"
    runat="server" />
<asp:RadioButton id="radPhoenix" GroupName="City" Text="Phoenix"
    runat="server" />
<asp:RadioButton id="radSeattle" GroupName="City" Text="Seattle"
    runat="Server" />
```

The main event associated with `RadioButtons` is the `CheckChanged` event; which can be handled with the `OnCheckChanged` attribute.

RadioButtonList

Like the `RadioButton` control, the `RadioButtonList` control represents radio buttons. However, the `RadioButtonList` control represents a list of radio buttons and uses more compact syntax. Here's an example:

```
<asp:RadioButtonList id="radlFavColor" runat="server">
  <asp:ListItem Text="Red" Value="red" />
  <asp:ListItem Text="Blue" Value="blue" />
  <asp:ListItem Text="Green" Value="green" />
</asp:RadioButtonList>
```

One of the great features of the `RadioButtonList` is its ability to **bind** to a data source. For instance, imagine you have a list of employees in a database. You could create a page that binds a selection from that database to the `RadioButtonList` control, to list dynamically certain employees within the control. The user would then be able to select one (and only one) employee from that list, and our code could determine the choice.

The most useful event produced by `RadioButtonList` is the `SelectedIndexChanged` event, to which you can assign a handler with the `OnSelectedIndexChanged` attribute

CheckBox

You can use a `CheckBox` control to represent a choice that can be only a yes (checked) or no (unchecked) value.

```
<asp:CheckBox id="chkQuestion" Text="I like .NET!" runat="server"
  />
```

As with the `RadioButton` control, he main event associated with a `CheckBox` is the `CheckChanged` event; which can be handled with the `OnCheckChanged` attribute.

CheckBoxList

As you may have guessed, the `CheckBoxList` control represents a group of check boxes; it's equivalent to using several `CheckBox` controls in row:

```
<asp:CheckBoxList id="chklFavDrinks" runat="server">
  <asp:ListItem Text="Pizza" Value="pizza" />
  <asp:ListItem Text="Tacos" Value="tacos" />
  <asp:ListItem Text="Pasta" Value="pasta" />
</asp:CheckBoxList>
```

Like the `RadioButtonList` control, the `CheckBoxList` control has the capability to bind to a data source, and produces a `SelectedIndexChanged` event that you can handle with `OnSelectedIndexChanged`.

DropDownList

A `DropDownList` control is similar to the HTML `<select>` tag. The `DropDownList` control allows you to select one item from a list using a drop-down menu.

```
<asp:DropDownList id="ddlFavColor" runat="server">
  <asp:ListItem Text="Red" value="red" />
  <asp:ListItem Text="Blue" value="blue" />
  <asp:ListItem Text="Green" value="green" />
</asp:DropDownList>
```

As is the case with other collection-based controls, such as the `CheckBoxList` and `RadioButtonList` controls, the `DropDownList` control can be bound to a database, thus allowing you to extract dynamic content into a drop-down menu. The main event produced by this control, as you might expect, is `SelectedIndexChanged`, handled with `OnSelectedIndexChanged`.

ListBox

A `ListBox` control equates to the HTML `<select>` tag with the `size` attribute set to 2 or more. The `ListBox` control allows you to select items from a multiline menu. If you set the `SelectionMode` attribute to `Multiple`, the user will be able to select more than one item from the list, as in this example:

```
<asp:ListBox id="listTechnologies" runat="server"
    SelectionMode="Multiple">
  <asp:ListItem Text="ASP.NET" Value="aspnet" />
  <asp:ListItem Text="JSP" Value="jsp" />
  <asp:ListItem Text="PHP" Value="php" />
```

```
  <asp:ListItem Text="CGI" Value="cgi" />
  <asp:ListItem Text="Coldfusion" Value="cf" />
</asp:ListBox>
```

Again, because the `ListBox` control is a collection-based control, it can be dynamically bound to a data source. The most useful event that this control provides is—you guessed it—`SelectedIndexChanged`, with the corresponding `OnSelectedIndexChanged` attribute.

Panel

The `Panel` control functions similarly to the `<div>` tag in HTML, in that the set of items that resides within the tag can be manipulated as a group. For instance, the `Panel` could be made visible or hidden by a `Button`'s `Click` event:

```
<asp:Panel id="pnlMyPanel" runat="server">
  <p>Username:
    <asp:TextBox id="txtUsername" Columns="30" runat="server" />
  </p>
  <p>Password:
    <asp:TextBox id="txtPassword" TextMode="Password"
        Columns="30" runat="server" /></p>
</asp:Panel>

<asp:Button id="btnHide" Text="Hide Panel" OnClick="HidePanel"
    runat="server" />
```

The code above creates two `TextBox` controls within a `Panel` control. The `Button` control is outside of the panel. The `HidePanel()` subroutine would then control the `Panel`'s visibility by setting its `Visible` property to `False`:

VB.NET

```
Public Sub HidePanel(s As Object, e As EventArgs)
  pnlMyPanel.Visible = False
End Sub
```

C#

```
public void HidePanel(Object s, EventArgs e) {
  pnlMyPanel.Visible = false;
}
```

In this case, when the user clicks the button, the `Click` event is raised and the `HidePanel()` subroutine is called, which sets the `Visible` property of the `Panel` control to `False`.

PlaceHolder

The `PlaceHolder` control lets us add elements at a particular place on a page at any time, dynamically, through code.

```
<asp:PlaceHolder id="phMyPlaceHolder" runat="server" />
```

The following code dynamically adds a new `HtmlButton` control within the place holder.

VB.NET

```
Public Sub Page_Load()
  Dim btnButton As HtmlButton = New HtmlButton()
  btnButton.InnerText = "My New Button"
  phMyPlaceHolder.Controls.Add(btnButton)
End Sub
```

C#

```
public void Page_Load() {
  HtmlButton btnButton = new HtmlButton();
  btnButton.InnerText = "My New Button";
  phMyPlaceHolder.Controls.Add(btnButton);
}
```

That's it for our quick tour of the basic Web controls. For more information on Web controls, including the properties, methods, and events for each, have a look at Appendix B.

Handling Page Navigation

Links from page to page are what drives the Web. Without linking, the Web would be little more than a simple page-based information source. Links enable us to move effortlessly from page to page with a single click; they bridge the gaps between related ideas, regardless of the boundaries imposed by geography and politics. This section focuses on page navigability using:

- the `HyperLink` control
- navigation objects and their methods

Suppose for a minute that you have created a Website that allows your users to choose from a selection of items on one page. You could call this page `viewcatalog.aspx`. Imagine that you have a second page, called `viewcart.aspx`. Once

users select an item from `viewcatalog.aspx`, you'd probably want to link them directly to `viewcart.aspx` so that they can keep track of their orders. To achieve this, we clearly must pass the information from the `viewcatalog.aspx` page over to the `viewcart.aspx` page.

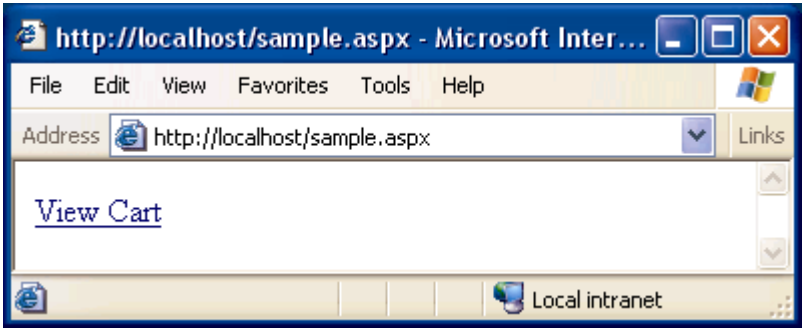
Using The `HyperLink` Control

The `HyperLink` control creates a simple HTML hyperlink on a page. Once it's clicked, the user is redirected to the page specified by the `NavigateUrl` property. For instance:

```
<asp:HyperLink id="hlAddToCart" NavigateUrl="viewcart.aspx"
    runat="server" Text="View Cart" />
```

Here, the `NavigateUrl` property specifies that this link leads to the page called `viewcart.aspx`. Figure 4.3 shows how the `HyperLink` control is rendered in the browser.

Figure 4.3. The `HyperLink` control renders similar to the anchor tag in the browser.



However, once we've arrived at the new page, it has no way of accessing the information from the first page. If we need to provide the user some continuity of information, we need something else.

Navigation Objects And Their Methods

The previous example rendered a simple control similar to the HTML anchor tag. Once the link is followed, however, we have no record of the previous page or any data it contained (the Web is a **stateless** technology).

If we wish to pass information from one page to the next, we can use one of the three methods listed below to create the link between the pages:

`Response.Redirect()`	Navigates to a second page from code. This is equivalent to using the `HyperLink` control, but allows us to set parameters on the query string dynamically.
`Server.Transfer()`	Ends the current Web Form and begins executing a new Web Form. This method only works when the user is navigating to a new Web Form page (`.aspx`).
`Server.Execute()`	Begins executing a new Web Form while displaying the current Web Form. The contents of both forms are combined in the response sent to the browser. Again, this method only works when the user is navigating to a Web Forms page (`.aspx`).

The easiest and quickest way to redirect your users from the `viewcatalog.aspx` page to the `viewcart.aspx` page would be using `Reponse.Redirect()`:

VB.NET

```
Sub linkClk(s As Object, e As EventArgs)
  Response.Redirect("viewcart.aspx")
End Sub
```

C#

```
void linkClk(Object s, EventArgs e) {
  Response.Redirect("viewcart.aspx");
}
```

You could then use the `LinkButton` control to call this subroutine as follows:

```
<asp:LinkButton id="lbAddToCart" Text="Add To Cart"
    OnClick="linkClk" runat="server"/>
```

This time, when you click the `LinkButton` control, the `Click` event is raised, the subroutine is called, and `Response.Redirect()` is called with the name of the page we want to link to as a parameter. In this way, we're redirecting to the new page directly from the code, rather than by using a particular tag. This enables us to pass information to the new page in the **query string**.

The query string is a list of variables and their respective values that we can append to a page's URL, allowing us to retrieve those variables and values from that page's code.

As an illustration, imagine you have a drop-down list that contains the following product information:

```
<p><asp:DropDownList id="ddlProducts" runat="server">
  <asp:ListItem Text="Pants" />
  <asp:ListItem Text="Shirt" />
  <asp:ListItem Text="Hat" />
  <asp:ListItem Text="Socks" />
</asp:DropDownList></p>

<p><asp:LinkButton id="lbAddToCart" Text="Add To Cart"
    OnClick="linkClk" runat="server" /></p>
```

The code you use to handle link clicks will need to find the item selected in the drop-down list and append it to the query string of the URL to which the user is to be redirected, as follows:

VB.NET

```
Sub linkClk(s As Object, e As EventArgs)
  Dim strQueryStr As String = "?Product=" & _
    Server.UrlEncode(ddlProducts.SelectedItem.Text)
  Response.Redirect("viewcart.aspx" & strQueryStr)
End Sub
```

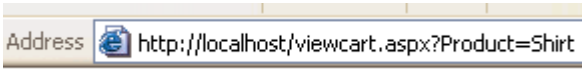
C#

```
void linkClk(Object s, EventArgs e) {
  string strQueryStr = "?Product=" +
    Server.UrlEncode(ddlProducts.SelectedItem.Text);
  Response.Redirect("viewcart.aspx" + strQueryStr);
}
```

Note the use of the `Server.UrlEncode()` method, which converts characters not allowed in query string values (e.g. `&`) to URL-safe character codes (e.g. `%26`) that the browser will understand. You should always use this method when adding arbitrary values to query strings.

When a user selects an item from the drop-down list and clicks the `LinkButton` control, the `viewcart.aspx` page is opened with the selected product appended as a parameter of the query string. This is illustrated in Figure 4.4.

Figure 4.4. Append the selected item to the query string.



Now that you've passed the product to the `viewcart.aspx` page, you have to grab it from the query string in the new page. We get hold of variables from the query string by accessing the `Request.QueryString` collection, like so:

VB.NET

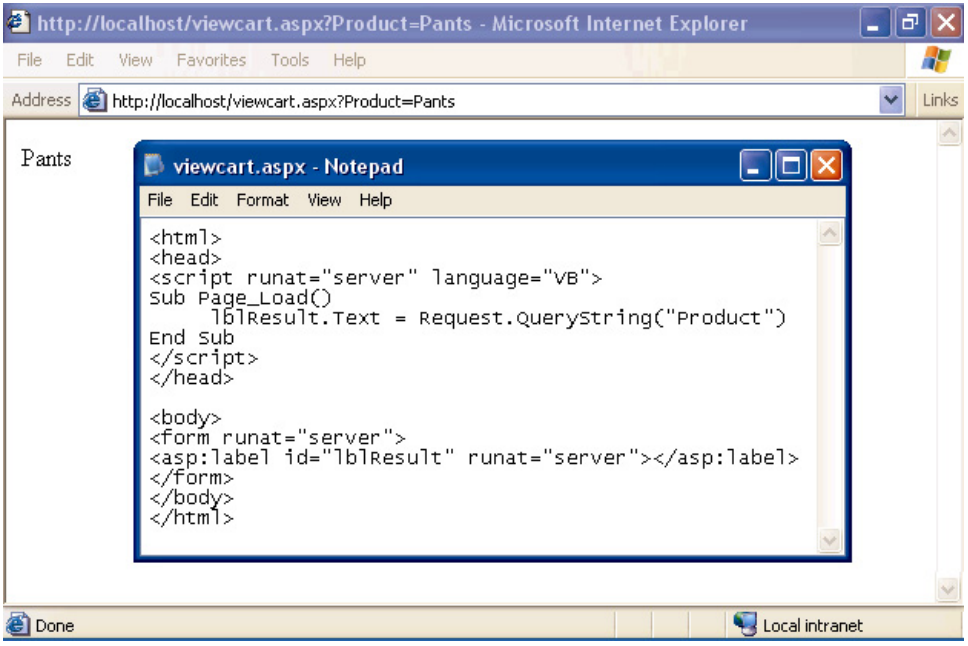
```
Sub Page_Load()
  lblResult.Text = Request.QueryString("Product")
End Sub
```

C#

```
void Page_Load() {
  lblResult.Text = Request.QueryString["Product"];
}
```

Here, we simply display the value of the `Product` query string parameter, as we see in Figure 4.5.

Figure 4.5. Set the text property of the label control within a `Page_Load` event handler to accept the new parameter value.



Now, when you select a product and add it to the cart, the result is displayed in the redirected page on a label with an `id` of `lblResult`. Now sure, a real product

catalog and shopping cart has a lot more to it, but in this section we've uncovered an important building block.

Postback

Postback can be confusing to newcomers because, while most ASP.NET developers know what it is, they can't seem to explain it clearly. The topics we've covered so far, like subroutines, functions, and events, are not new to most Web developers. HTML, in combination with client-side JavaScript, has been doing all that for years. ASP.NET is different to this model, though, because it is a server-side, not client-side, technology—events that occur on a page are handled by code running on the server. For this to work, ASP.NET uses the mechanism of postback.

When an event is triggered, for instance, a button is clicked, or an item in a grid is selected, the page is submitted back to the server for processing, along with information about the event and any preexisting data on the page (via view state). We say the page "posts back" to the server. This is a powerful concept to grasp because it is postback that lets us run code on the server rather than on the client's browser, and it is postback that lets our server code know which items within a drop-down list were selected, or what information a user typed into a text box.

But what would happen if you had multiple `DropDownList` controls that were populated with database data? Users could interact with those `DropDownList` controls and, in turn, we could set certain options within the page based on what they selected from the drop-down menus. Although this seems like a common task, with traditional ASP it incurred considerable overhead. The problem is that while the data that's bound to the drop-down menu from the database never changes, every time the user selects an item from the drop-down menu and a postback has to be done, the database must be accessed again to rebuild the contents of each drop-down list on the page. However, this is not a problem in ASP.NET.

In ASP.NET we can check for postback with the `IsPostBack` property, and thus avoid performing any time consuming tasks unnecessarily. `IsPostBack` is a page-level property—meaning that it's a property of the page itself—and we'd most commonly use it in the `Page_Load()` event handler to execute code only when the page is first loaded. Consider the following example:

VB.NET File: **PostBack.aspx**

```
<html>
<head>
```

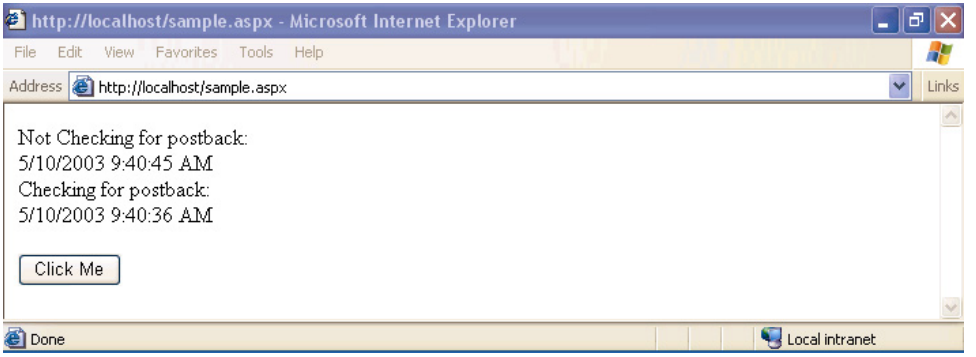
```
<script runat="server" language="VB">
Sub Page_Load(s As Object, e As EventArgs)
  lblMessage1.Text = Now()
  If Not IsPostBack Then
    lblMessage2.Text = Now()
  End If
End Sub
</script>
</head>

<body>
<form runat="server">
  <p>Not Checking for postback:<br />
    <asp:Label id="lblMessage1" runat="server" /></p>
  <p>Checking for postback:<br />
    <asp:Label id="lblMessage2" runat="server" /></p>
  <p><asp:Button id="btnClick" Text="Click Me" runat="server" />
  </p>
</form>
</body>
</html>
```

C# File: **PostBack.aspx**

```
<html>
<head>
<script runat="server" language="C#">
void Page_Load(Object s, EventArgs e) {
  lblMessage1.Text = Convert.ToString(DateTime.Now);
  if (!IsPostBack) {
    lblMessage2.Text = Convert.ToString(DateTime.Now);
  }
}
</script>
</head>

<body>
<form runat="server">
  <p>Not Checking for postback:<br />
    <asp:Label id="lblMessage1" runat="server" /></p>
  <p>Checking for postback:<br />
    <asp:Label id="lblMessage2" runat="server" /></p>
  <p><asp:Button id="btnClick" Text="Click Me" runat="server" />
  </p>
</form>
</body>
</html>
```

The result will look similar to Figure 4.6.

Figure 4.6. The `IsPostBack` property checks to make sure the user isn't resubmitting the page.



In this example, the `IsPostBack` check means that the second label doesn't refresh when the `Button` control is clicked. Similarly, we could use `IsPostBack` within the `Page_Load()` subroutine to set up database-driven drop-down menus just once within each user's session, making the online experience smoother, and making our application more scalable. Don't worry if postback seems a bit confusing now—we'll use it more in upcoming chapters, so if it doesn't yet, it should make sense after a few more practical examples.

Formatting Controls with CSS

HTML was deliberately designed to pay little attention to the specifics of how particular items on a page were rendered. It is left up to the individual browser to work out these intricacies, and tailor the output to the limitations and strengths of the user's machine. While we can change font styles, sizes, colors, and so on using HTML tags, this is a practice that can lead to verbose code and pages that are very hard to restyle at a later date.

The **Cascading Style Sheets (CSS)** language aims to provide the degree of control, flexibility, and pizzazz that modern Web designers seek. It's a standard that's widely supported by all the popular browsers, in its oldest version (CSS1) at the very least.

CSS is a powerful tool for Web developers because it gives us the power to create one set of styles in a single sheet, and apply those styles to all the pages in our

Website. All the pages then use the same fonts, colors, and sizes for the same sections, giving the site a consistent feel throughout. Regardless of whether our site contains three pages or three hundred, when we alter the styles in the style sheet, those changes are immediately applied to all pages based on that style sheet.

Types of Styles and Style Sheets

There are three different ways of associating styles to elements of a particular Web page. I've already mentioned the first, and usually the best, which is an external file:

External File — By placing your **style rules** in an external style sheet, you can link this one file to any Web pages where you want those styles to be used. This makes updating a Website's overall look a cakewalk.

Document Wide — Rather than having an external sheet, you can place style rules for a page within a `<style>` tag inside that page's head element. The problem is that we can't then use those styles in another page without typing them in again, which makes global changes to the entire site difficult to manage.

Inline — Inline styles allow us to set styles for a single tag using the `style` attribute. For instance, we might create a text box in regular HTML with a `style` attribute that draws a border around the text box like so:

```
<input type="text"
    style="border-style:groove" />
```

CSS style rules create styles that are applied to elements of a page in one of two ways[1]:

Classes — Arguably the most popular way to use styles within your pages, classes allow you to set up a custom style that will be applied to any tag or control that has a

[1]This is, to some extent, a simplified view of how CSS works. For the complete story, refer to *HTML Utopia: Designing Without Tables Using CSS* (SitePoint, ISBN 0-9579218-2-9).

class attribute that matches the name of your custom style.

Tag Redefinition Redefining a tag affects the appearance of certain standard HTML tags. For instance, the `<hr>` tag is generally given a width of 100% by default, but you could redefine the tag in CSS to have a width of 50%.

Whether you're building external, document-wide, or inline style sheets, properties for classes and tag redefinitions use the same syntax. To create a class within an external style sheet file, you'd use the following syntax:

```
.myClass {
  font-family: arial;
  font-size: 10pt;
  color: red;
}
```

This would then be saved in a file with a `.css` extension, such as `styles.css`, and linked into the Web Form with the following line in the `<head>` tag of your document:

```
<link href="styles.css" rel="stylesheet" />
```

Similarly, to define a class within a document-wide style sheet, you would use the following syntax:

```
<head>
<style type="text/css">
  .myClass {
    font-family: arial;
    font-size: 10pt;
    color: red;
  }
</style>
</head>
```

When you're using inline styles, use the following syntax:

```
<span style="font-family: arial; font-size: 10pt; color: red;">My
Stylized Text</span>
```

For inline styles, simply add all properties to the tag in question with the `style` attribute. Above, we've used the `<span>` tag, but the principle remains the same for the other tags.

Now that you have a basic understanding of some of the fundamental concepts behind CSS, let's look at the different types of styles that can be used within our ASP.NET applications.

Style Properties

There are many different types of properties that you can modify using style sheets. Below is a list of the common types:

Font — This category provides you with the ability to format text level elements, including their font face, size, decoration, weight, color, etc.

Background — This category allows you to customize backgrounds for objects and text. Modifying these values gives you control over the color, image, and whether or not you want to repeat an image.

Block — This category allows you to modify the spacing between paragraphs, lines of text, and spaces between text and words.

Box — The box category provides changes and customizations for tables. If you need to modify borders, padding, spacing, and colors on a table, row, or cell, you can modify elements within this category.

Border — This category lets you draw boxes of different colors, styles and thicknesses around page elements.

List — This category allows you to customize the way ordered and unordered lists are created.

Positioning — Modifying positioning allows you to move and position tags and controls freely.

These categories provide a list of what can generally be modified using CSS. As we progress through the book, the many types of properties will become evident.

The `CssClass` Property

Once you have defined a class in a style sheet (be it external or internal), you'll want to begin associating that class with elements in your Web Forms. You can associate classes with ASP.NET Web controls using the `CssClass` property. The following example uses classes defined within a document-wide style sheet:

```
<html>
<head>
<style type="text/css">
  .dropdownmenu {
    font-family: Arial;
    background-color: #0099FF;
  }
  .textbox {
    font-family: Arial;
    background-color: #0099FF;
    border: 1px solid;
  }
  .button {
    font-family: Arial;
    background-color: #0099FF;
    border: 1px solid;
  }
  .text {
    font-family: Arial, Helvetica, sans-serif;
    font-size: 10px;
  }
</style>
</head>

<body>
<form runat="server">
<p class="text">Please select a product:</p>
<p><asp:DropDownList id="ddlProducts" CssClass="dropdownmenu"
    runat="server">
  <asp:ListItem Text="Shirt" selected="true" />
  <asp:ListItem Text="Hat" />
  <asp:Listitem Text="Pants" />
  <asp:ListItem Text="Socks" />
</asp:DropDownList></p>
<p><asp:TextBox id="txtQuantity" CssClass="textbox" runat="server"
    /></p>
<p><asp:Button id="btnAddToCart" CssClass="button" runat="server"
    Text="Add To Cart" /></p>
```

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

A Navigation Menu and Web Form for the Intranet Application

Now that you have a solid foundation in HTML controls, Web Forms, Web controls, Page Interaction, Navigation, and Style Sheets, you're ready to begin working on the project that we'll build on throughout the remainder of this book. With the **Dorknozzle Intranet Application**, I hope to introduce you to real world development in simple stages, as we work through the following chapters together.

Introducing the Dorknozzle Intranet Application

While most books give you a series of simple, isolated examples to illustrate particular techniques, this book is a little different. Many of the examples provided in these pages will involve work on a single project—an intranet application for the fictional Dorknozzle company. We'll build on this application as we go along, illustrating the many different concepts that are important to developers of any type of Web application. The intranet application we'll develop will offer the following functionality:

Welcome — Displays company event information to the user of the Web application.

Helpdesk — Allows any Dorknozzle employees to submit a problem as a helpdesk ticket to an IT administrator regarding issues they experience with software, hardware, or their computer.

Employee Store — Employee stores boost company morale. By building an online store, we'll allow Dorknozzle employees to buy life-enriching items such as mugs, shirts, and mouse pads. All will proudly bear the Dorknozzle logo, of course!

Newsletter Archive — Another way to improve morale is to keep employees informed of company events and news. Each month,

	the Dorknozzle HR Manager will send out a company newsletter to all employees.
Employee Directory	Employees will likely want to call each other to discuss important, company-related affairs… such as last night's television viewing! The employee directory should let employees find other staff members' details.
Address Book	While the employee directory houses handy information for use by staff, the purpose of the address book is to provide more detailed information about all of the employees within the company
Admin Tools	Administrators will need a way to modify closed helpdesk tickets, delete the records of fired employees, create newly hired employees' profiles, modify information on current employees, and more. The admin tools section will provide the interface for this.

Before we can begin creating all these smaller applications, we must build the framework that will act as a template across the site. In this section, we'll accomplish the following introductory tasks for the development of our intranet application:

- ❑ Build the navigation menu.
- ❑ Create the style sheet.
- ❑ Design the template and Web Form for the helpdesk application.

Building the Navigation Menu

Once it's complete, our fictitious intranet application will have modules for an IT helpdesk, employee store, newsletter archive, employee directory, address book, and admin console. Obviously, we're going to need some kind of navigation menu to make those sub-applications simple to find. Throughout this chapter, we've studied numerous ways of navigating from page to page, and we could use any of these methods here. We've discussed controls such as the `Button` control, `HyperLink` control, and `LinkButton` control, and we've explored various objects and methods for navigating from code. Although all these would work to a certain degree, in this case, only one makes the most sense in terms of performance and practicality.

Before we begin, you'll want to obtain the necessary files from the code archive for this book. The files for this chapter include a starting template that you can use for this project, as well as the complete version in case you run into problems.

Because we're not submitting any data for processing, we can eliminate the `Button` and `LinkButton` controls; each involves extra work from the server in order to process the `Click` event it raises. As we only want to link from one page to the next, and don't care about performing any tasks programmatically, we can use the simpler `HyperLink` control instead. Remember, we add a `HyperLink` control to the page by inserting the following code inside the form:

```
<asp:HyperLink NavigateUrl="index.aspx" runat="server"
    Text="Home" />
```

This would add a link that showed the text "Home."

Open up your text editor and create a new file with the standard HTML tags required by ASP.NET pages, including an empty form with a `runat="server"` attribute. Inside this form, add the `HyperLink` controls for helpdesk, employee store, newsletter archive, employee directory, address book, and admin tools, like so:

File: **index.aspx (excerpt)**

```
<!-- HyperLink controls for navigation -->
<img src="Images/book_closed.gif" width="16" height="16" alt="+"
    />
<asp:HyperLink NavigateUrl="index.aspx" runat="server" Text="Home"
    />
<br />
<img src="Images/book_closed.gif" width="16" height="16" alt="+"
    />
<asp:HyperLink NavigateUrl="helpdesk.aspx" runat="server"
    Text="HelpDesk" />
<br />
<img src="Images/book_closed.gif" width="16" height="16" alt="+"
    />
<asp:HyperLink NavigateUrl="employeestore.aspx" runat="server"
    Text="Employee Store" />
<br />
<img src="Images/book_closed.gif" width="16" height="16" alt="+"
    />
<asp:HyperLink NavigateUrl="newsletterarchive.aspx" runat="server"
    Text="Newsletter Archive" />
<br />
<img src="Images/book_closed.gif" width="16" height="16" alt="+"
```

```
    />
<asp:HyperLink NavigateUrl="employeedirectory.aspx" runat="server"
    Text="Employee Directory" />
<br />
<img src="Images/book_closed.gif" width="16" height="16" alt="+"
    />
<asp:HyperLink NavigateUrl="addressbook.aspx" runat="server"
    Text="Address Book" />
<br /><br />
<img src="Images/book_closed.gif" width="16" height="16" alt="+"
    />
<asp:HyperLink NavigateUrl="admintools.aspx" runat="server"
    Text="Admin Tools" />
<!-- End HyperLink controls -->
```

Once the links have been added to the page and you've placed the `book_closed.gif` file in a subdirectory called `Images`, you could save your work (as `index.aspx`) and view the results in your browser. At this stage, however, it would look fairly bland. What we need is a few pretty graphics to provide visual appeal! Although modern Web design practices would have us use CSS for our page layout and visual design, we'll resort to HTML tables here in order to stay focused on the server-side aspects of our application.

Open `index.aspx` and create the following two regular (i.e. not server-side) HTML tables at the very start of the page body:

File: **`index.aspx` (excerpt)**

```
<body>
<form runat="server">

<table width="100%" border="0" cellspacing="0" cellpadding="0"
    background="Images/header_bg.gif">
  <tr>
    <td><img src="Images/header_top.gif" width="450" height="142"
        alt="the official dorknozzle company intranet"
        /></td>
  </tr>
</table>

<table width="100%" border="0" cellspacing="0" cellpadding="0">
  <tr>
    <td width="157"><img src="Images/header_bottom.gif"
        width="157" height="37" alt="" /></td>
    <td></td>
```

```
  </tr>
</table>
```

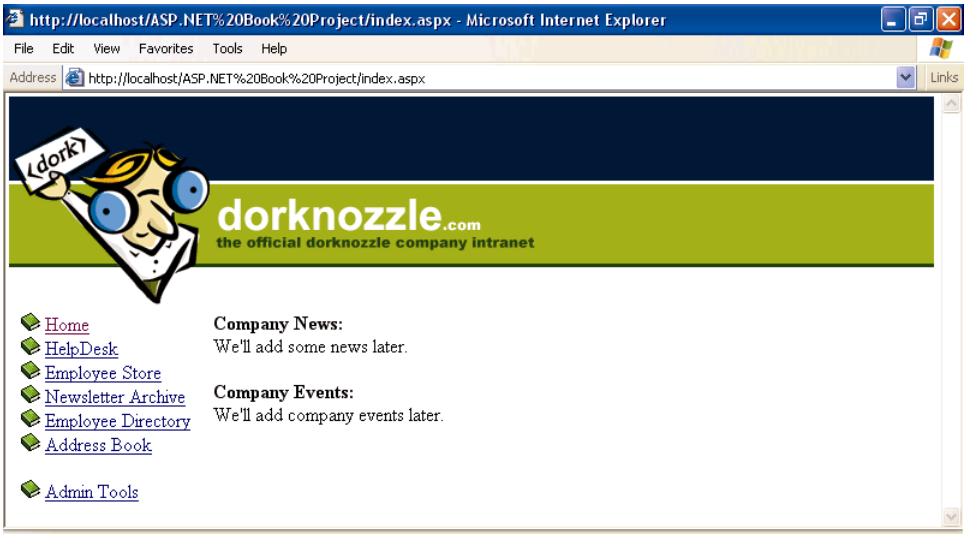
We'll want to place our links in a table too. While we're there, we'll add some news items to the main index page. Open up `index.aspx` once more, and place the following HTML table around the links we've already added:

File: **index.aspx (excerpt)**

```
<table width="100%" border="0" cellspacing="0" cellpadding="10">
  <tr>
    <td valign="top" width="160">
      <!-- HyperLink controls for navigation -->
      …
      <!-- End HyperLink controls -->
    </td>
    <td valign="top">
      <h1>Company News:</h1>
      <p>We'll add some news later.</p>
      <h1>Company Events:</h1>
      <p>We'll add company events later.</p>
    </td>
  </tr>
</table>

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

The result will look similar to Figure 4.7.

Figure 4.7. Add `HyperLink` controls for the Intranet navigation menu.



Isn't it amazing the difference some well-chosen graphics can make? Don't forget to place the pictures from the download in the `Images` subdirectory. You can, of course, find the completed source in the code archive, although I do recommend you type the code in yourself as we progress, for practice value.

Create the Corporate Style Sheet

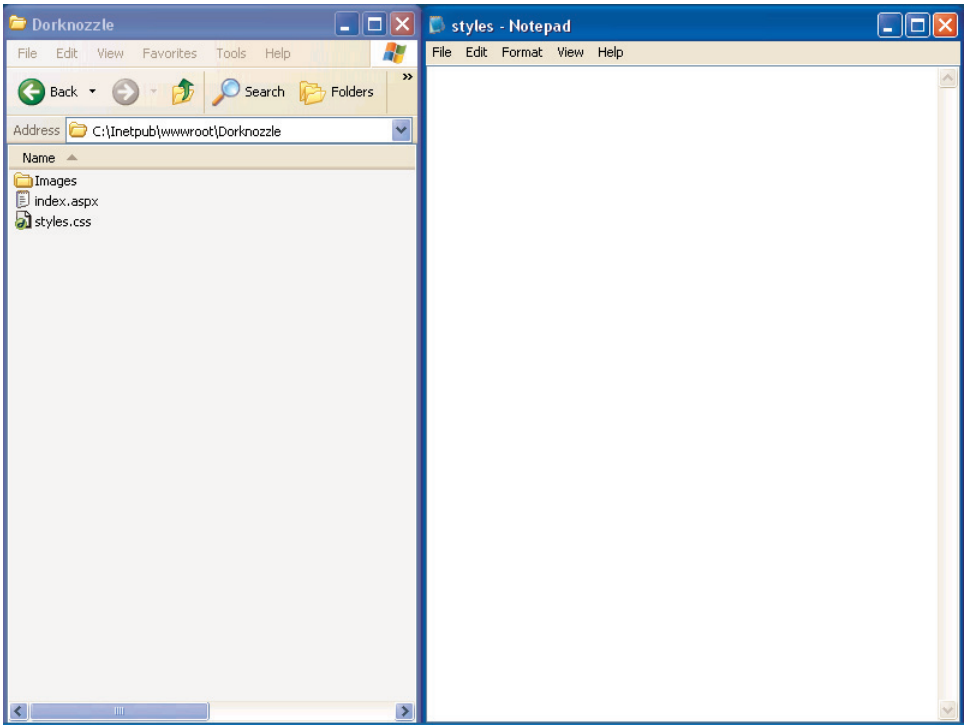
If you don't mind the ordinary look of standard Web pages, then you can skip this section. If, however, you don't like standard blue hyperlinks, black, Times New Roman text, and beveled form controls, this section is for you.

As you've already read, style sheets provide developers with flexibility and control over the "look" of Web applications. In this section, we'll explore the addition of a customizable style sheet to our fictitious intranet application. We will define styles for the following elements within our application:

- Hyperlinks
- Text (including body text and headings)
- Boxed controls (including text boxes and drop-down menus)

You can start by creating the CSS file that the styles will reside in. I've opened Notepad and immediately saved the file as `styles.css` within the root directory of the application, as shown in Figure 4.8.

Figure 4.8. Open Notepad and save the file as `styles.css` within the root directory of the application folder.



Now, let's apply some style properties to the following tags:

- `body`
- `p`
- `h1`
- `a:link`
- `a:hover`

You'll notice the `a:link` and `a:hover` items in this list, which are not strictly-speaking tags. In the world of CSS, these are known as a **pseudo-elements**. `a:link` narrows the selection to `<a>` tags that are links (as opposed to `<a name="…">` tags, which are targets). Assigning properties to `a:hover` will apply those properties only to links over which the user is hovering the mouse.

We'll also define a few classes for certain Web controls that don't map directly to a particular HTML tag:

`.textbox`	For `<asp:TextBox>` controls, which become `<input type="text">` and `<textarea>` tags when sent to the browser.
`.button`	For `<asp:Button>` controls, which become `<input type="button">`, `<input type="submit">`, and `<input type="reset">` tags.
`.dropdownmenu`	For `<asp:DropDownList>` controls, which become `<select>` tags.

Below is the code for the CSS rules that will apply the desired basic formatting to our site. Type the following just as it appears into your `styles.css` file:

```
body {
  background: #FFFFFF;
  color: #000000;
  margin: 0;
  padding: 0;
}
p {
  font-family: Arial;
  font-size: 12px;
}
h1 {
  font-family: Arial;
  font-size: 14px;
  color: #000000;
}
a:link {
  font-family: Arial;
  font-size: 12px;
  color: #000000;
}
a:hover {
  font-family: Arial;
```

```
  font-size: 12px;
  color: #FF0000;
}
.textbox {
  font-family: Arial;
  font-size: 12px;
  border: 1px solid black;
}
.button {
  font-family: Arial;
  border: 1px solid black;
  background-color: #CCCCCC;
}
.dropdownmenu {
  font-family: Arial;
  font-size: 12px;
  background-color: #CCCCCC;
}
```

Now that the style sheet file has been created, we can link the style sheet file to `index.aspx` by inserting the following line into the `<head>` tag of the document:

```
<link href="styles.css" rel="stylesheet" />
```

We'll need to assign the CSS classes we have defined (`textbox`, `button`, and `dropdownmenu`) to relevant controls as we create them, but for now our simple HTML template will automatically benefit from the tags we have redefined.

Remember, we're not limited to these styles. If, throughout the development of our application, we decide to add more styles, we'll simply need to open the `styles.css` file and add them as necessary.

You can save your work at this point, and view it in the browser.

Design the Web Form for the Helpdesk Application

The last part of the project is to add the employee Helpdesk request Web Form. This will be a Web page that allows our fictitious employees to report hardware, software, and workstation problems. The Web Form will be arranged as a series of simple steps that users can follow to report their problems:

- Pick from a predefined category of potential problem areas. (`DropDownList` control)
- Pick from predefined subjects within the categories. (`DropDownList` control)
- Type a description of the problem. (Multiline `TextBox` control)
- Submit the request. (`Button` control)

Rather than creating a new, blank page and retyping all the code, you can simply copy `index.aspx` and rename it `helpdesk.aspx` (or save a copy with the new name if it's already open in your editor). The only portion of the code that will change to accommodate the HelpDesk interface is the last table in the body—the one that contains the news items on `index.aspx`. Everything else stays the same, because we want to have a single look for all our pages[2]. Change the final column in the table to create two drop-down lists, a multiline text box, and a button, as shown:

```
      <!-- End HyperLink controls -->
    </td>
    <td valign="top">
      <h1>Employee HelpDesk Request</h1>
      <p>Problem Category:<br />
        <asp:DropDownList id="ddlCategory" CssClass="dropdownmenu"
            runat="server" /></p>
      <p>Problem Subject:<br />
        <asp:DropDownList id="ddlSubject" CssClass="dropdownmenu"
            runat="server" /></p>
      <p>Problem Description:<br />
        <asp:TextBox id="txtDescription" CssClass="textbox"
            Columns="40" Rows="4" TextMode="MultiLine"
            runat="server" /></p>
      <p><asp:Button id="btnSubmit" CssClass="button"
            Text="Submit Request" runat="server" /></p>
    </td>
```

Notice how we've applied our CSS classes to the appropriate controls here.

Don't worry that the `DropDownList` controls don't have items associated with them—the categories and subjects will be predefined within database tables. Later, we'll bind these database tables to their respective controls.

When you're finished, save your work and view it in a browser.

[2]We'll see better ways to do this in later chapters…

Summary

In this chapter, we discussed HTML controls, Web Forms, and Web controls. We also explored how to link between pages, and how to add style to controls. You even built your first project, putting together the information you've learned in this and previous chapters.

Your Web application efforts will focus predominantly on Web controls. In the next chapter, we'll learn how to check user input on those Web controls through the use of the ASP.NET validation controls.

Index

Symbols

A

B

C

D

E

F

G

H

I

N

O

P

Q

R

S

T

U

V

W

X

Z

What's Next?

If you've enjoyed these chapters from *Build Your Own ASP.NET Website Using C# & VB.NET*, why not order yourself a copy?

In the rest of the book, you'll learn how to put ASP.NET to full use in a number of practical solutions. In particular, you'll discover how to use databases in conjunction with ASP.NET to create dynamic Websites. And because instructions are provided for Microsoft Access, SQL Server, and even Microsoft's free database, MSDE, you can work through the book no matter what database software you have.

You'll also gain access to the code archive download, so you can try out all the examples without retyping. Just like in the book, all examples in the code archive are offered in C# and VB.NET varieties.

Here are just a few of the things you'll learn to do:

- validate form input automatically
- set up a relational database for your dynamic Website
- create a database-driven employee directory
- design a slick Web interface to update the database
- build an online store, complete with shopping cart and credit card processing
- control access to your site with a database of users
- consume and build your own XML Web Services
- And a whole lot more…