Justin Rogers

Microsoft® JScript™ .NET Programming

SAMS

Microsoft® JScript .NET Progamming

Justin Rogers

SAMS

800 East 96th Street, Indianapolis, Indiana, 46240 USA

Microsoft® JScript .NET Programming

International Standard Book Number: 0-672-32148-3

Library of Congress Catalog Card Number: 2001087594

Printed in the United States of America

First Printing: December 2001

04 03 02 01 4 3 2 1

Associate Publisher
Jeff Koch

Acquisitions Editor
Neil Rowe

Development Editors
Jason Burita
Steve Rowe

Managing Editor
Matt Purcell

Project Editor
Christina Smith

Copy Editors
Margaret Berson
Cheri Clark
Kitty Jarrett

Indexer
Tom Dinse

Proofreader
Plan-It Publishing

Technical Editor
Peter Torr

Team Coordinator
Denni Bannister

Media Developer
Dan Scherf

Interior Designer
Anne Jones

Cover Designer
Aren Howell

Page Layout
Ayanna Lacey

Contents at a Glance

Contents

About the Author

Justin Rogers is currently a vendor for Microsoft Projects through CompuWare Corporation and previously worked for the Microsoft Frameworks team as the owner for the .NET QuickStart Tutorials available on the .NET Frameworks SDK. Justin enjoys working with DirectX and has worked on many open source game projects. After owning the QuickStarts, Justin moved on to an Internet-enabled Peer-to-Peer screensaver game called the Terrarium. This game was introduced at the PDC and was considered one of the leading technology demos for high performance .NET applications.

Dedication

To Amy Ashburn, who stayed up through all of the late nights of writing and coding and who bore the brunt of the weekend writing attitudes after 60+ hour work weeks. And to Neil Rowe, whose constant involvement throughout the book writing process made my first title possible.

Acknowledgments

I'd like to thank Turbine, the creators of Asheron's Call, for delivering such a poor gaming experience for several months in a row. With this break in my obsession with online role-playing games, I was able to complete this book and impart my love of JScript and the JScript .NET language to the masses.

More importantly, I'd like to thank my fiancée, Amy Ashburn, for allowing me to work on this book, for allowing the loss of so many weekends' worth of quality time and adventures in the Washington State mountainsides, and for allowing the computers to maintain such close proximity to the bedroom in case I got the urge to type a few extra samples or paragraphs out of the book. I'd also like to point out that even though I've managed to get a book published, my more talented fiancée hasn't yet gotten a chance to publish any of her children's books. I am extremely grateful for this not becoming an issue in our relationship and, Amy, when you read this, I hope you manage to become a very successful children's author.

However, books aren't just about personal allowances. There are other critical people involved in every title. I'd like to thank Peter Torr, PM for Microsoft Scripting (of Microsoft) for all of his help in the initial stages of the book and for answering my many questions about the language as it was still being developed and changed. I'd like to thank Herman Ventnor, JScript .NET compiler Dev (of Microsoft) for fixing all of the bugs I found throughout my testing of the JScript compiler, for providing a quick turnaround on all of my feature requests, and for humoring me when I thought I found a bug and really didn't. And finally Neil Rowe of Sams Publishing, who constantly got involved in the writing of my book, because every chapter I submitted was at least several months behind schedule. Still he persevered and this book is the final result.

Tell Us What You Think!

E-mail: `feedback@samspublishing.com`

Mail: Mark Taber
Associate Publisher
Sams Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

Introduction

Generally in the programming industry, the rule of thumb is that you have to learn the latest and greatest technologies to get on the high-end pay scales and to ensure job security. At times this means long hours of work learning brand-new concepts, brand-new programming languages, or leading-edge techniques. However, while spending all this time learning brand-new technologies, no one can ever find the time to become an expert in a specific field.

As Microsoft continues with the JScript language and supports ECMAScript, and now with JScript .NET and the speed and power of a compiled full-featured language, the reality is that you can bring your old skills to bear in this new generation of technologies. I am targeting this book to those who used JScript in their ASP programming instead of VBScript, to those who used JScript to manage their networks with WSH, and to those who feel the shift from Java to .NET will be best done using the JScript .NET language.

This book primarily focuses on the JScript .NET language and syntax. If you've had some experience with the JScript language, you'll have a much easier time reading the book, but it isn't necessary. I've tried to organize the book such that any features previously found in JScript are in the beginning chapters, and any new features come in the later chapters. This way you can skip through the content as you see fit and not miss anything really important.

One of the most powerful features of JScript .NET is the CLR (common language runtime) and .NET frameworks, along with all of the associated classes. Most of the programming problems that couldn't be solved in JScript without the use of many additional COM libraries can now be handled directly from the classes already available to the language through the CLR. This includes advanced file IO, access to high-speed XML classes, and direct access to the internet using Web access classes, just to name a few. Because each of these classes is important to making JScript .NET a very effective language, I've chosen to dedicate a large amount of the text to these classes alone.

In addition to a large number of support classes, Microsoft has released several frameworks for easing the more common programming tasks. ASP.NET was developed to replace ASP and greatly enhances the speed and ease of use for server-side programming. JScript .NET provides a crucial role in this transition from ASP to ASP.NET because most JScript code can be quickly and easily ported for use in ASP.NET with little or no changes. For the client side, Microsoft has developed Windows Forms for working with windowed applications. Windows Forms is extremely easy to learn and is the first API capable of adding GUI elements to the JScript .NET language.

This book contains extremely precise descriptions of the various functions and language syntax available in JScript .NET. The code listings help to enforce this description, as do the large number of samples available on disk. However, no matter how many times a feature is described and demonstrated, nothing beats the hands-on experience. You'll gain much more thorough knowledge of the JScript .NET language if you try out different features and try to break and fix the code samples provided on disk. Sometimes it also helps to type the content even though there is a copy of it on disk that you can compile and run. This reinforcement through use has saved me many times when I didn't have the documentation or a code sample readily handy to demonstrate a new feature. I hope you enjoy learning an old language over again, along with the new features. I can certainly say that I enjoyed writing the book and watching the language grow so much in such a short time.

Introduction to JScript .NET

CHAPTER

1

IN THIS CHAPTER

This chapter will serve as a platform for you to decide which chapters of the book are suitable for you. Some of you are probably beginning programmers who are somewhat proficient in JavaScript or JScript, and others are advanced programmers proficient in another .NET language or another object-oriented language. This chapter provides a sample migration path from JavaScript to JScript .NET and explains some of the history behind both. It also covers coding conventions and explains each of the appendixes, which will be extremely important to getting the various samples compiled and for experimentation with the samples beyond what is provided in the book. Let's begin by covering the history and development behind JScript .NET.

Development of JScript .NET

JavaScript began as a language that worked within a Web browser through scripted code in an HTML page. Because of this, JavaScript had a familiar syntax that Java users could quickly pick up and offered many of the object-oriented techniques available to the Java language. In addition to providing the basic Java-style types and language constructs, JavaScript also provided a Document Object Model (DOM) to access various portions of the Web page. At this point a developer could write code that interacted with the user on a Web page, and also updated or verified various portions of the content. The effect was mostly limited to changing images on the fly, in the case of mouseovers, or verifying form content so that preliminary data checking could happen on the client rather than bogging down the Web server and the user's Internet connection.

Of the many features that came out of the first versions of JavaScript, the missing portion remained the ability to update the page dynamically. Dynamic page updating is the ability to add new content and affect the flow and rendering of the current document while the user is viewing the page. To fill this void came a new version of the DOM. The new version of the DOM was not really an extension to JavaScript itself, but more a collection of objects with which JavaScript could interact to dynamically modify content on a page. With this new DOM version came a new type of HTML known as DHTML, or Dynamic HTML, which enabled JavaScript to fully modify the objects on the page. This included changing the stylesheet information, visibility of objects, the flow of text around elements, and even moving objects around the page. This new freedom to modify the document when the user interacted or viewed the page was a very well-received concept, leading to a whole new level of page-based JavaScripts. This included dynamic menus, addition or revealing of content, and the ability to create forms that could create or remove fields as needed on the fly.

While developing the DOM and DHTML implementations, Microsoft began to develop several extensions to their version of the JavaScript language. They added several objects, such as the Dictionary, that was similar to Java's Hashtable, and the `FileSystemObject` (a COM object used by scripting languages to access the files and directories) that could access files. At this point the language was usable in both client and server environments and was released under the name JScript rather than the traditional JavaScript. They also made their scripting engines available to other consumers on the machine, rather than just the Web browser.

The most notable of these consumers was the Windows Scripting Host, which expanded JScript from the IE platform to the PC administration and task platform. Now users could write small JScript programs to make modifications to handle rudimentary tasks that batch files once performed. JScript became a batch processing language and again gained access to a much more powerful object model. Windows Scripting Host introduced several new objects for accessing the user's registry, environment, and for launching applications on the machine. This was also a much better place to make use of the `FileSystemObject`. Here it could be used for maintaining files, writing logs, and backing up entire directories on the machine when used as an administration tool. In its most recent edition, WSH has gained several added capabilities from the WMI (Windows Management Instrumentation) interfaces. Scripts are able to access Active Directory information, machine information that was previously available only through API calls, and even make modifications to information on other machines in the same or another trusted network. Future versions will probably add even more benefits while making additional scripting languages available to the programmer.

To answer many of the concerns behind the limitations of the JScript language, Microsoft has made continual advancements in their implementation. Most recently JScript 5.1 and 5.5 were released. Recent enhancements include the introduction of regular expressions for textual parsing and verification, and additions to the `Array` type that make it act like a normal array, a stack, or a numerically indexed hash (sparse array). Conditional compilation was added some time ago, but doesn't come in extremely handy unless you're doing scripts that constantly access a variety of platforms. The enumeration form (`for...in`) of the `for` loop comes in extremely handy when enumerating over properties or items in an array when you don't care about modifying or maintaining an index number. And finally, starting with version 3.0, Microsoft made JScript ECMAScript-compliant. *ECMAScript* is the Web-based scripting standard supported by the ECMA-262 specification. Any version later than 3.0 of the JScript implementation remains ECMAScript-compliant while at the same time adding additional levels of functionality. To end this discussion of the JScript language's history, note that both JScript and JavaScript were developed to be fully object-oriented. Both languages allow for the creation and use of objects, but in an abstract way. The object-oriented functionality made JScript a sure candidate to become a Common Language Runtime (CLR)-compliant language.

Enter JScript .NET

The next step in JScript's development is a fully compiled JScript based on Microsoft's new Common Language Runtime (CLR). Known as JScript .NET, it is both compatible with the specifications required to be a CLR-compliant language (which means it has notions of classes and inheritance), as well as fully backward-compatible with the JScript language as it stands.

This means that all your old JScript syntax should work with little or no modification, and that all the old objects and methods are still there if the programmer would rather use those over the newer objects and methods provided by the CLR and the .NET Framework.

Porting of the JScript interpreter to the JScript .NET compiler began early in 2000 with the first usable version of the JScript .NET compiler becoming available after the Professional Developer's Conference in July 2000. This was the first usable version of the compiler, and many bugs remained to be worked out. With the Beta 1 Release of the .NET platform in early November, a nearly fully functional version of the compiler was shipped. With the version 1 release of the .NET platform, all the features available in the other .NET compilers should be available in JScript .NET. The next logical question is: What does JScript .NET have to offer compared to the other .NET languages, and what it will continue to offer even after its release and during future development?

What Does JScript .NET Have to Offer?

The most important item JScript .NET has to offer is full backward compatibility with the JScript language. All the old objects are still there. All the old syntax is still available. Even items that don't port well to the new platform have been preserved so that transitioning between the two languages is as painless as possible. Of course, many of the older features aren't the best choice for performance. The newer features of the language will ensure that JScript .NET increases in speed and performance at the same time it increases in flexibility and usability.

Backward Compatibility with JScript

To better show you the backward compatibility features of JScript .NET, let's look at some examples of the old syntax and the newer syntax. Here you will explore the top five areas that cover the majority of programming in JScript and how those will equate to the new JScript .NET:

- Strongly typed variables
- Creation and usage of COM components
- Expando properties of objects
- Arrays
- Regular expressions

Strongly Typed Variables

In previous versions of JScript, variables did not have to be declared. If a variable wasn't declared to a given scope, it was considered to be a global variable and available to the entire program. JScript .NET preserves this behavior but highly recommends that variables are declared and typed. Typing of variables is new to JScript .NET and allows more performance because the type of a variable is known at compile time. Any variables that aren't typed and are declared at a local scope are candidates to be strongly typed at compile time. If the variables can't be strongly typed, they are given a type of `Object`. Variables of type `Object` have to be explicitly cast to be used in a function call. JScript .NET will handle this for you, but at a small loss in performance, which will be described in Chapter 3 when we cover data types.

Accessing COM Components

Previous versions of JScript had a container object that could be used to instantiate COM components. This object is known as the `ActiveXObject`. The `ActiveXObject` is still available but causes a very bad performance hit because all calls have to be dynamically marshaled at runtime from the .NET platform to the COM platform. The new method for instantiating a COM component is to create a wrapper during compile time that gets included with the application and is used instead of the `ActiveXObject`. This won't always work if the object isn't known at design time.

Using Expando Properties

In prior versions of JScript, every variable was treated as an object, whether it acted as an array, a string, or a number. You could also assign a value to any property of an object, even those properties that didn't exist. This was known as an expando property because it isn't a normal property, but is dynamically generated during the assignment. JScript .NET takes this several steps further. JScript .NET uses namespaces, which group code (as discussed in Chapter 2), and classes and interfaces exist within a namespace; classes can, in turn, extend or inherit from other classes and implement interfaces. These additional language features give quite a bit more syntax to remember, but provide the programmer with the full power of an object-oriented language. Multiple inheritance is not supported, and interfaces don't allow for base implementations and are completely virtual. Through all the new syntax and strong typing, the ability to maintain expando properties has remained in the language through the use of a special expando attribute. Again, the use of any legacy JScript features, such as expando properties, increases the load and execution time of a program. These features are strongly dismissed in favor of new features of the language.

JScript Arrays

The JScript array is extremely powerful. The `Array` object is a sparse array (an array that takes up only as much space as needed to store its assigned elements) and only assigns several discrete indexes actual values. The additional ability of pushing and popping items off the array, as if it were a stack, comes in extremely handy. All in all, the JScript array is a powerful language item, but at the same time extremely slow compared to traditional arrays, which have bounds, always maintain a specific memory size, and always contain the same amount of elements (which means no dimensioning of arrays as the program advances). This traditional array is available to JScript .NET through the `System.Array` class of the Common Language Runtime. Its features don't include any of those available in the JScript-style array, but the performance increase makes up for the lack of flexibility. The JScript .NET compiler will make a decision at compile time as to whether or not a declared array should be of a certain type, or the programmer can specify which to use to ensure that the correct behavior is exhibited.

Support for Regular Expressions

My favorite aspect of JScript has always been its support for regular expressions. However limited the implementation might be, it has always come in handy for parsing jobs that normally take a 20-line or larger function to walk the characters of a string. As for verifying data integrity (whether some data obeys a pattern or not), regular expressions just can't be beat. You can specify inline regular expressions, or you can utilize the new objects available through the .NET Framework. Of the new features available to JScript .NET regular expressions, full support for backtracking and the addition of precompiled objects are the most important. Precompiling a regular expression means that it isn't stored in textual format and is translated directly to a parsing routine at compile time. This feature makes great speed enhancements when the parsing strings being used are static. The ability to backtrack means that you can take advantage of both look-ahead and look-behind assertions when trying to verify that a pattern occurs before or after your current match. All of these new features make for a great enhancement to the story of JScript .NET regular expressions.

Fully Compiled and Optimized for .NET

Being a CLR-compliant language means several things. First, it means that JScript .NET gets to take advantage of being compiled into Microsoft Intermediate Language (MSIL), which is then portable across systems and is later compiled into native code. All CLR-compliant languages produce MSIL and all go through the same JIT process that produces native code for the platform upon which the code is being run. Furthermore, any enhancements to the JIT process directly relate to an improvement in the speed of the JScript .NET code.

JScript .NET provides the ability to use any services provided by the CLR or the Microsoft .NET Framework in addition to all services that are exported by the language. You were introduced to some of these in the previous section, which covered the new regular expressions objects available in JScript .NET. Other features that are now available to JScript .NET include encryption, file I/O, a global configuration format, data access, and even XML. As other classes and features are added, JScript .NET will be able to support them by simply referencing the new objects on the command line.

Another feature of the .NET platform inherited by JScript .NET is the ability to interact with objects programmed in other languages. After objects are compiled out to IL in the form of assemblies (discussed in Chapter 2), they can be imported and used in JScript .NET. This is something that was available in COM, but was never a reality for scripting-style languages. The possibilities of this feature are endless, but it generally means that the right language can be used for each portion of a job. Inheritance is available cross-language as well, meaning that JScript .NET objects can extend other low-level base objects written in C# or Managed C++.

With all these new features available in JScript .NET and through the .NET Framework, do you ever have to really use JScript again? The answer is that some of the code you have now might be reusable. You could easily write new code to perform the same action and only use new language features and objects available through the frameworks. For this reason, I've organized the book in such a way that it doesn't have to be read chapter by chapter. I'm going to describe this outline now in the hopes that advanced users will jump ahead to the sections they need instead of treading through the wordy explanations in the first few chapters!

Organization of This Book

Chapter 1 is the introduction. So far this chapter has described the history and development of JScript .NET, enumerated some of the new features, and talked about the tight integration with the .NET Framework. This section of this chapter gives you a basic book outline. The final section of this chapter explains my coding conventions.

Chapter 2 will be an explanation of the new system imposed by the CLR and the .NET Framework. Key items of discussion will include namespaces, assemblies, and modules, as well as their JScript counterparts. In many places, you'll notice that JScript has a different naming convention than that used by the CLR or other .NET languages. In this chapter I'll be drawing those comparisons. I recommend that all readers glance over this chapter in order to learn the .NET terminology that will be used throughout the book. I also go over some of the basics of OOP (Object-Oriented Programming) in this chapter. Any JavaScript/JScript programmers who haven't used the OO features of the language should probably examine this section thoroughly.

Chapters 3 through 8 are called the Language Reference section. This section discusses all the syntax of the JScript .NET language. This section proceeds from data types, operators, control flow statements, and finally moves into classes and interfaces. Data types include both native types and Framework types. This will become important as you see all the various objects and types you can use when developing your program.

Chapter 4 covers comparison and assignment operators, arithmetic operators, bitwise and Boolean conditionals, and commenting structures. JScript .NET won't have a documentation comment feature at release and may never have one. However, several comment structures are very promising for third-party tools that generate documentation comments. Control flow covers all branching, loop, and enumeration statements. The topics of classes and interfaces are broken down into two chapters, 6 and 7.

Chapter 6 discusses, in depth, the creation of a class and any items that apply mainly to classes. Chapter 7 continues to discuss interfaces and then discusses those elements that are part of both classes and interfaces, such as member declarations and properties. Chapter 8 is devoted solely to exception handling, because controlling errors in a program is extremely important. Controlling errors is made even more important by the .NET Framework, where each individual set of objects throws different exceptions that can be caught and used to control program flow.

Chapter 9 covers the JScript compiler and its various features. This includes syntax for compiling each type of assembly available from executable files to libraries. Debugging and tracing are also discussed in this chapter, because several command-line options will enable some of these features in the compiled program. I'll be discussing conditional compilation directives. These might or might not be supported at release, but you might find the information in the chapter useful anyway. If, and when, the compiler supports conditional compilation, it will have a specified syntax. This chapter finishes up with some of the limitations of JScript .NET.

Chapters 10 through 13 discuss various portions of the Frameworks. These sections will be only console-based JScript .NET applications that exhibit the features of each individual .NET namespace. I begin in Chapter 10 by examining `System.IO` and file manipulation. I recommend this chapter for all developers because these classes will be the same no matter which language you are using. They are the basis for file manipulation across the entire .NET Frameworks. Chapter 11 discusses the use of the `System.Xml` namespaces. This includes discussions of the `XmlReader` and `XmlWriter` classes for easy manipulation of XML documents. The `XmlDocument` is also discussed in depth and is used to edit existing XML documents. Chapter 12 moves into regular expressions and provides a very good example of the transition from JScript-style to JScript .NET-style usage. This section covers regular expressions in great detail because I find they are very useful for string manipulations that happen in almost every program. If you aren't going to be doing a lot of string manipulation in your programs, Chapter 13 is certainly one you can skip. Chapter 13 finishes by examining the `System.Net` namespace and how to connect applications to the Web. The .NET Framework makes it extremely easy to write low-level applications that work on a socket level, but at the same time provides classes that easily connect to and retrieve information from Web servers. If you're not planning on Web-enabling your applications, you can probably skip these chapters as well.

Chapter 14 is a rundown of the features of ASP.NET. JScript works well with ASP.NET, and nearly all ASP.NET programming tasks can be accomplished using the JScript .NET language. Throughout Chapter 14 only the basics are discussed, enough to get a heads-up on why ASP.NET is so much better than ASP.

Chapter 15 discusses a very basic usage of the `System.Windows.Forms` namespace and how to create GUI applications. The discussion moves from the creation of basic forms and the standard controls, through the use of menus and common dialog controls, and finally ends with a Notepad-style application.

The appendix contains some items you might find helpful for your own development. Appendix A will be a JScript compiler reference. This will include all command-line switches and options available to the shipping version of the JScript .NET compiler.

Coding Conventions

I'll keep this short and to the point. I have a certain coding convention that I use whenever I code. So as not to confuse anyone and to keep me straight on my own convention, I'll explain it here for both ASP.NET and JScript .NET.

When programming in ASP.NET, I use the following conventions:

- All HTML tags except the opening and closing tags are indented using one space.
- Multiline controls such as the `DataGrid` are split and use exactly two spaces to indent each subproperty.
- The closing tag is on its own line and is flush with the opening tag.
- The ASP is always lowercase and the name of the control is always the same case as its class name, even though ASP.NET is not case sensitive.

All internal templates obey the same rules as HTML and are indented only one space, as is any cascaded text within them. This can sometimes look ugly, but tabs turn into nightmares after a while, and using too many spaces can often blur the cascading nature of an HTML document.

Any inline JScript obeys the JScript .NET style guidelines that follow.

```
<HTML>
 <HEAD>
 </HEAD>
 <BODY>
  <asp:DataGrid
    runat="server"
  />
  <asp:DataGrid
    runat="server"
  >
   <ItemTemplate>
    <SubControls />
   </ItemTemplate>
  </asp:DataGrid>
 </BODY>
</HTML>
```

When programming in JScript .NET, I use the following conventions:

- All code is indented at four spaces per block. The first piece of code starts out flush with the left side of the page and has no indentation. This is generally a package declaration.
- All braces begin on the same line as their beginning statement with one space between the end of the statement and the brace. The package, class, method, and branching statements in the following example all exhibit this arrangement.
- All attributes are on the line immediately preceding their target or in the case of multiple attributes, one on each line immediately preceding their target.
- All variable declarations consist of items that are aligned using spaces so that the type is flush with all other types, and the initializer is flush with all other initializers. All conditional statements will have one space before their conditional.
- If a conditional operator has both a block and single statement form, I always use the block statement form.
- Generally, when programming, you need to add debug messages in conditional blocks. I put in the braces all the time so that I don't have to add them when I update the block with debug statements. The `if` statement in the following example demonstrates this perfectly.
- Extensions to conditional blocks such as `else` statements immediately follow the closing brace of the previous block. This more closely ties related statements together and shrinks code.

```
import System;
import System.IO;

package Sams.Press.Samples {
    var Variable1    :Type1      = 0;
    var LongVariable2:LongType2  = 0;

    WebMethodAttribute
    ObsoleteMethodAttribute
    public class CodeConventions {
        public static void Main(Args:String[]) {
            if ( x == null ) {
                return;
            } else {
                // Notice that else doesn't have its own line
}
              return;
          }
    }
}
```

CHAPTER

2

Organization of JScript .NET

IN THIS CHAPTER

Traditionally when using either JavaScript or JScript, there was little or no organization of the code. All code was either global code, part of a function, or was used to define a new class. Now in JScript .NET, there are many extra levels of organization. Classes and code can be contained within packages, and functions can be either part of global code or contained within classes. Classes now have real properties, methods, and fields contained within their definition. In the following sections you'll discover some of these new concepts of organization.

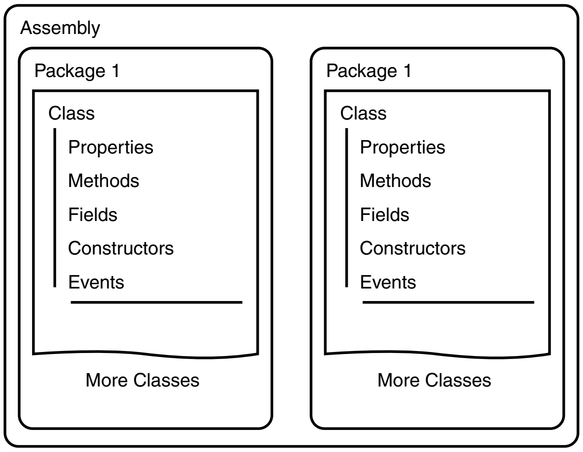


FIGURE 2.1
Organization of JScript .NET code.

Packages and Namespaces

Classes in JScript .NET are organized into packages so that objects in one package can reuse the names of objects used in other packages. The common example is that Microsoft provides a generic `String` object in `System.String`. However, JScript .NET has a different interpretation of what a `String` should be. This `String` is housed in `Microsoft.JScript.String`. As an application developer you can use either one, or you can even specify your own `String` object within your own package. The following code gives an example of creating two separate packages that both contain objects of the same name. Because the two objects are in different packages, their names don't conflict:

```
package AlphaGroup.Team1 {
    public class String {
    }
}
```

```
package AlphaGroup.Team2 {
    public class String {
    }
}
```

When accessing these classes, JScript .NET offers two separate methods. The first method is to specify the full type name of the object. This full name includes the namespace and object name. The second method is to import a namespace so that you can refer to classes by their short name. The following code demonstrates both techniques:

```
// Create a String object using full type name
var sTeamLeader1:AlphaGroup.Team1.String;

// Import a namespace and use the short name
import AlphaGroup.Team2;
var sTeamLeader2:String;
```

NOTE

The use of imported namespaces is simply an easy way of accessing commonly used types. One common mistake is to import two namespaces that both export an object of the same name. When this happens, the compiler will throw an error. At this point, you'll have to fully qualify your type names to make the compiler happy, or you'll have to remove one of your namespace imports.

Namespaces in .NET

A *namespace* in .NET is simply a logical division for the names of objects. It provides an additional level of scope so that not only can two objects contain members with the same names, but they can also be objects of the same name, given that they are in separate namespaces. A namespace can be used to break up code into modules based on what area each module affects. It can also break code down by the team that is working on each module, as is the case with the `System.IO` and `System.Data` namespaces in the .NET Framework. Any naming scheme that will help better organize classes is the sole purpose of a namespace within .NET.

The Difference Between Packages and Namespaces

Namespaces and packages are basically the same. The only difference is that any code not in a package in JScript is considered to be global code. This global code is in turn placed inside its own namespace by the JScript .NET compiler, and any executable code is made into an entry point if the code is compiled as an executable. The following code demonstrates some JScript .NET code with packages and global code:

```
import AlphaGroup.Team1;
package AlphaGroup.Team1 {
    public class String {
        ...
    }
}
var sTeamName:AlphaGroup.Team1.String;
sTeamName = "Primary String Development Team";
```

In the preceding code, the `String` object would be placed in the `AlphaGroup.Team1` Namespace. The remainder of the code would be placed in the special global namespace. And, subsequently, if compiled as an executable, the code would be made to run on program startup. This code would create a new variable `sTeamName` of type `AlphaGroup.Team1.String`. Then the value “Primary String Development Team” would be assigned to the new variable.

Classes and Interfaces

Actual code in JScript .NET is arranged in two categories. The primary category is *classes,* with *global code* being the second. Most of the time when programming in JScript .NET, you will decide to organize your code into a group of objects that each in turn work on some data. This might be a `String` object that contains methods for modifying an array of characters or a complex business object that has raw data at its base, but also contains methods for graphs, computations, and ways to update or transform the data within. Having a group of objects that each support a common set of methods, which could be used by another program to retrieve graphs, might be a good enhancement to your business objects. One business object might be for car sales and another for employee information, but a manager might only want to see trends in the data. These methods would be described in an interface that would be supported by each of your objects.

Interfaces are contracts for classes that describe a set of methods or properties that must be supported. Other classes can in turn work with a class through its interfaces instead of through the class directly.

Writing Code as Objects

When you are making the big step to object-oriented programming, there is an initial set of criteria for writing code as a set of classes or objects rather than as a set of functions. A set of functions can easily be rewritten in an object-oriented fashion by having a single object that implements each function as a static member. This is the easiest way to port function-oriented code to object-oriented code. The problem with this design is that the programmer has to maintain any data that the functions work on. This can sometimes be painful, especially when several hundred pieces of data have to be managed. The following example shows some code in which a class with a static method operates on some data that is stored and managed in the application.

```
public class DataManager {
    public static function DoStuff(Int32 a) : Boolean {
        ...
    }
}

var Var1:Int32 = 25;
var Var2:Int32 = 30;

DataManager.DoStuff(Var1);
DataManager.DoStuff(Var2);
```

The next logical step is to store the data in an instance of your class. Each time you create one of these classes, you hand off the data and it holds it for you. Any of the functions you create can work on the data local to the object, and thus you don't have to specify the data on each call. In this way, the programmer is concentrating more on program control and design rather than on managing data—because the burden of managing data is offloaded to the object.

The following code example expands on the previous `DataManager` class so that it stores data locally. This allows the programmer to initialize the data once and then not have to manage it for the lifetime of the object.

```
public class DataManager {
    private var _Stuff:Int32;

    public function DoStuff(Int32 a) : Boolean {
        ...
    }
}

var Var1:DataManager = new DataManager(25);
var Var2:DataManager = new DataManager(30);

Var1.DoStuff();
Var2.DoStuff();
```

Deciding on Base Classes Versus Interfaces

A common decision you have to make when writing for a single-inheritance programming language is when to use base classes to inherit from or when to use interfaces. The answer is actually pretty simple. If you have two distinct groups of objects that need the same set of functions, you use an interface rather than a base class. If you need to implement more than one set of methods and properties, you also need two user interfaces because JScript .NET and the CLR support only single inheritance.

Base classes become important when the member definitions and member functionality need to be defined. Interfaces can only specify that a method with a certain prototype must be exported by an object, whereas the base class will specify not only the prototype of the method, but also an implementation in code for that method. This implementation can be overridden by the child class, and can be called from within the child's implementation as well. Using base classes allows for a common set of functionality and not just a common set of member prototypes.

Class Members

Class members come in the form of properties, fields, constructors, events, and methods. All class members can have scope modifiers applied to limit visibility and usage. As each class or object is supposed to contain its own data, the members of a class must provide the programmer with ways to manipulate and retrieve that data.

Instance Members

When you need an object reference to hold and manipulate data, you can use instance members. *Constructors* provide a means of initializing the internal data of a class during creation. *Fields* can be directly accessed and assigned by the user. *Properties* can provide controlled access to internal data, and a property is generally used in place of a field when access to the raw data has to be controlled. *Methods* are used to perform the more complex actions of a class.

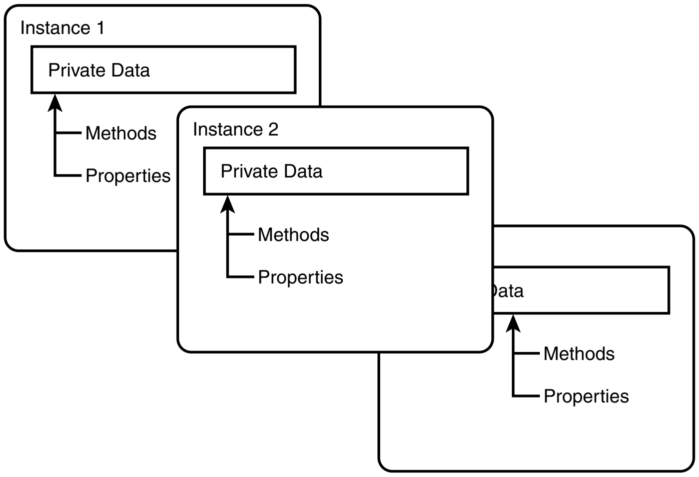


FIGURE 2.2
Instance members.

Static Members

Static members are different from instance members in that every class instance shares the same static members. This can be better illustrated by creating two objects of the same type. If you change the value of one of the instance members, then only the class you changed the value of will see the change. The other class instance will maintain the value it originally had. However, if you change a static member value, it changes on all instances of that type. Take a look at Figure 2.3 and you'll see how this mapping takes place.

In addition to having static member variables, each class can have a single static constructor. This constructor can be used to initialize any static members the first time any of the static members are accessed.

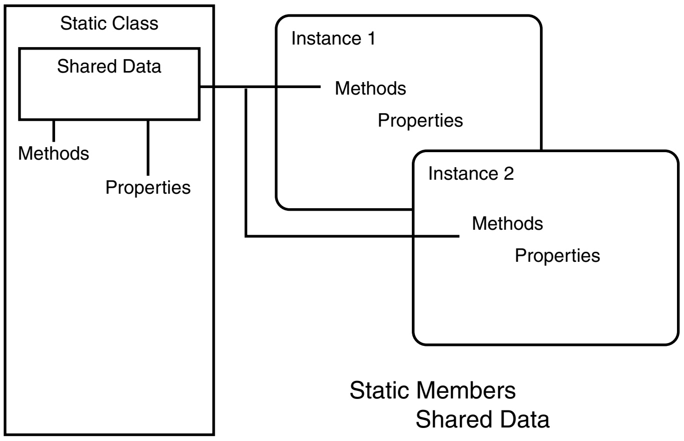


FIGURE 2.3
Static members.

Making Methods Extensible

All members marked as public or protected are available to other classes that inherit from the base class. This allows for a child class to provide custom functionality or add additional data that the base class was unaware of or not designed to handle. A good example of this might be a class that outputs its data in a well-formatted string. If a child class adds additional data, the base class method won't be able to format this new information. At this time, the child class will override the method and either provide its own formatting for all data, or allow the parent to format data in the base class, whereas the child class formats only additional data. Designing methods to be extensible can help greatly in program design.

Assemblies, Executables, and Libraries

All code, whether an executable or a module, is stored in *assemblies*. These assemblies contain versioning information, entry points, special metadata, and, of course, all the classes and code you've created. An assembly may span more than one logical disk file. Of these files, one must contain an assembly manifest that, in turn, names each of the other modules. The assembly can, of course, be in the form of an executable or a dynamic link library (DLL).

NOTE

The metadata stored in an assembly won't be discussed in great detail anywhere in this book. Any member, class, or assembly attributes and modifiers will be stored as metadata. Debug information and symbols are stored as well.

Creating Assemblies for Code Access

All code is compiled into an assembly. Both executables and libraries can contain identical items with the addition of an entry point being defined for an executable. All related code should be contained within the same assembly; a namespace defines code logically, whereas an assembly defines its physical location. When using code from other assemblies, you have to reference those assemblies at compile time. Earlier in this chapter, you saw the `import` statement, which directs the compiler to import a given namespace so that the shortened type name can be used. In addition to this, the actual assembly that contains the classes used must be imported on the command line.

The following example shows the syntax for importing an assembly. What you'll notice is that it is referencing the `System.dll` assembly. If you use any of the types available in `System.dll`, and you don't include it on the reference line, the compiler will throw an error because it can't find the classes you are trying to use.

```
jsc.exe /t:exe /r:System.dll somecode.js
```

Structure of an Executable Assembly

Several steps are required in writing the code necessary for the JScript .NET compiler to create an executable assembly. You might not require each of these three items for every executable, but these are the items that are generally required.

- To define any classes or interfaces that are necessary to program operation
- To import any additional namespaces that will be used
- To provide some global code that will get the application started

This can generally be accomplished by creating an instance of a class and calling a method to get things kicked off. Here we will start by creating a basic JScript executable that uses only global code:

```
// GlobalExecutable.js
import System;

var sMessage:String;
sMessage = "Global Code Solution!";
Console.WriteLine(sMessage);
```

```
jsc.exe /t:exe /r:System.dll GlobalExecutable.js
```

TIP

Make use of the global code feature when porting JScript code to JScript .NET. Often a quick-and-dirty port can be made with little or no effort. Functions, objects, and code can all be created in the global namespace.

In the preceding code, the `System` namespace is imported. This is done so that the `Console` object doesn't have to be fully qualified. This line can be left off if `System` is appended to the type name of `Console`. A `String` variable is then declared, assigned, and finally printed to the console. This code defines no new objects and lives within the global namespace provided by the JScript .NET compiler. Another approach would be to encompass all this code in its own class rather than leaving it all global. This following example demonstrates a more object-oriented executable:

```
// ObjectExecutable.js
import System;
import JScript.NET.Samples;

package JScript.NET.Samples {
    public class ObjectExecutable {
        public function Start() : void {
            Console.WriteLine("Object Code Solution!");
        }
    }
}

var oRunner:ObjectExecutable = new ObjectExecutable();
oRunner.Start();
```

```
jsc.exe /t:exe /r:System.dll ObjectExecutable.js
```

This code behaves identically to the first except that all executable code is placed in the `Start` method of the class. This syntax is closer to the other .NET languages in form and would make it much easier to port the code to another language later. Either way works just fine, and the first sample is actually much smaller and easier to code. This is one of the advantages of being fully backward compatible with the JScript language.

Structure of a Library Assembly

Libraries are nearly identical to executables. The only difference is that they don't contain an entry point or a block of code that executes when they are first run. This comes in handy when several programs all need to use a group of common objects. All of the .NET Framework is implemented as a set of libraries for use within your programs. The following code is a very simple object library. You'll notice that there is no global code (you can still have global code though, it's still allowed), and that the command line to the JScript compiler has changed a bit.

```
// TaxLibrary.js
import System;
package JScript.NET.Samples {
    private static var Percentage:Int32 = 100;

    public class TaxMan {
        public function Collect() : void {
        }
    }
    public class IRS {
        public function Audit() : void {
        }
    }
}
```

```
jsc.exe /t:library /r:System.dll TaxLibrary.js
```

What we have here appears to be a set of classes for use when computing the amount of taxes owed to the IRS. This library might be used by an application such as Quicken or Microsoft Money as a tax add-on. You'll notice that the `/t:` switch is now set to `library`. JScript .NET only supports console executables and libraries. A future release might add additional types, such as Win executables that don't spawn a command window and modules that are used in advanced incremental compilation.

Basic JScript .NET Program

This final section explains how to get some basic programs up and running. You learn about all the items required to get up and running successfully.

Creation of a Basic Console Program

The basic operations of a console program include getting some information from the user, generally via command-line arguments, and then displaying some data back out to the screen for the user to examine. I won't be discussing anything interactive here because I think interactive console applications are generally a waste of time. Normally, console applications will accept any required parameters on the command line so that they can be used for batch automation. We'll get more into interactive examples in Chapters 14, "Using JScript .NET in ASP .NET," and 15, "Windows Forms Programming in JScript .NET," where we'll be discussing UI concepts for ASP.NET Web applications and Windows Forms applications.

The first important item is examining how the command line is obtained. In the `System` namespace, Microsoft has provided an `Environment` class. This `Environment` class is loaded with methods for getting information about the command line, environment variables, OS version, and so on. Now let's write some code to make this useful. The following example demonstrates how to create a `String` array, retrieve the command-line parameters, and process them.

```
var Args:String[] = Environment.GetCommandLineArgs();

for(var i = 0; i < Args.Length; i++) {
    ...
}
```

CAUTION

When working with strongly typed arrays, remember that they are bounded and will throw an exception if the index provided is out of range. This is different from JScript-style arrays that will simply return an undefined element when accessing an array element that hasn't been assigned.

The first argument in the command line will be the name of the executing program. So generally the loop will start at 1 rather than 0. The preceding code just demonstrates how to get the command line. The next step is to give information back to the user. This happens through the `Console` class, which is also located in the `System` namespace. The `Console` class provides access to all three standard console streams, `Input`, `Output`, and `Error`. You can interact with these streams directly using Stream IO methods, or you can utilize the functions available on the `Console` class itself. For this example we'll be using the methods on the `Console`:

```
Console.Write (Args[i]);
Console.WriteLine(Args[i]);
```

Console can do both a `Write` and a `WriteLine`. The only difference is that `WriteLine` will output the line feed character for the system and wrap output to the next line. This is handy in languages like Visual Basic (VB) that don't support metacharacters within strings. With this set of code in mind, we can write a console application that does some work. What we have here is an application that does some simple arithmetic:

```
// CommandLine.js
import System;
import JScript.NET.Samples;

package JScript.NET.Samples {
    public class CommandLineParser {
        public function Add() : void {
            var Args:String[] = Environment.GetCommandLineArgs();

            if ( Args.Length > 2 ) {
                Console.WriteLine(Int32.Parse(Args[1]) + Int32.Parse(Args[2]));
            }
        }
    }
}

var oParser:CommandLineParser = new CommandLineParser();
oParser.Add();
```

```
jsc.exe /t:exe /r:System.dll CommandLine.js
CommandLine 2 3
5
```

If you want to do more than just add two command-line arguments together, you could simply use another command-line argument to specify the operation and then write a `switch` statement or an `if` statement to decide which action to take. I won't go into such detail here and will leave this as an exercise for the knowledgeable reader.

That is pretty much everything to getting basic console applications up and running under JScript .NET.

Summary

You should be taking away quite a bit of information from this chapter, even though you might not understand all the details yet. First and foremost, you should understand the multiple levels of code organization made available through both the Common Language Runtime (CLR) and the JScript .NET language. These levels include the concepts of an assembly to hold compiled code, and packages and namespaces that act as naming containers and allow types of the same name to coexist through classes and their members.

You've also seen what is required to compile code in both an executable assembly and a library assembly. We'll be using these basic compiler flags and principles from now until Chapter 9, where you'll discover a much more comprehensive listing of compiler options.

Next we'll move on to data types, arrays, and strings.

CHAPTER

3

Datatypes, Arrays, and Strings

IN THIS CHAPTER

This chapter discusses the declaration of variables in JScript .NET, the correlation of variables with common language runtime (CLR) types, and how the compiler makes intelligent decisions between legacy JScript semantics and new performance-oriented JScript .NET semantics.

The discussion in this chapter moves from the new data typing syntax used in JScript .NET straight to the base language datatypes. A thorough explanation of strongly typing variables is followed an exploration of how to create arrays. Both the legacy JScript style array and the newer CLR-style array are discussed. The only other remaining datatype of importance is the string. Strings are the basis for the majority of programs.

Strongly Typing in JScript .NET

Strongly typed variables allow the JScript .NET compiler to use appropriately sized internal data structures when performing operations. Typed data doesn't have to be converted to and from different datatypes when basic operations and assignments occur. It always occupies the appropriate amount of memory space for its size, thus allowing the developer to optimize for size and use smaller datatypes when appropriate. Furthermore, it allows the compiler to remove various checks that were once required when the same variable could at one point be a string and at another point an integer.

How to Strongly Type Variables

So how do you start taking advantage of these new improvements in JScript .NET? You declare variables to give them scope and you type variables so that the compiler uses the optimized operations and removes the type checks during runtime. You declare all variables by using the `var` keyword. Any additional access modifiers needed to complete the declaration can precede the `var` keyword. An additional modifier, a *custom attribute*, can also be applied (both access modifiers and attributes are discussed in Chapter 6, "Creating Classes"). The following example demonstrates how to declare a nontyped variable named `UnknownVar`:

```
// Untyped variable declaration in Global Code
var UnknownVar;
```

NOTE

For a complete explanation of the access modifiers and custom attributes, you can jump to Chapter 6, which discusses class members. The following are some of the important modifiers you can learn more about in Chapter 6:

- The `public` keyword makes variables available outside the current scope but does not make the variable global.

- The `static` keyword makes variables persistent after they go out of scope, effectively making the variable a global variable yet limited to being global inside the current scope, without additional modifiers.
- The `private` keyword declares variables as locally scoped variables that are available only to code within the current scope.

The scope-related modifiers are available only from within a class definition. So, for the most part, you won't be using them in the early chapters of this book.

Simply declaring a variable doesn't do anything more than give that variable a scope. If you performed a declaration within a class, the variable would be scoped to that class. If you performed the same declaration within a method, the variable would be scoped to the function level. Because the declaration in the preceding example occurs in global code, it creates a global variable named `UnknownVar` of type `Object`. This doesn't provide any performance benefits, and the variable isn't strongly typed. To strongly type a variable, you use the syntax *`name`*`:`*`type`* (where *`name`* is the name of the variable and *`type`* is the variable type). The colon after *`name`* tells the compiler that you want to strongly type the variable. The following is an example of strongly typing several variables (notice that whitespace doesn't matter to the compiler):

NOTE

Notice that the introduction of whitespace (that is, space characters, tabs, and carriage returns) in the code doesn't matter to the compiler and doesn't affect the way the compiler parses the source code.

```
var StringVar:String
var IntVar   :Int32
var BoolVar:    Boolean
var ArrayVar   :   String[]
```

All these statements create valid variables that are strongly typed. The whitespace between the variable name, colon, and type name doesn't matter because the compiler ignores whitespace. You can make the type name a short name, or you can make it a fully qualified name by appending the namespace to the type name. After you type this code, the compiler will throw warnings if you try to assign invalid values. Assigning to `StringVar` an object of some type that can't be converted to a string would result in a compiler error, as would assigning a string to `IntVar`.

Providing Initial Values for Variables

You can initialize variables with default values. You might want to do this to ensure that you have some working values before the code starts operating on them. You can also see some of the compiler errors by creating some initialization expressions where the type of the initializing value is of a different type than the variable to which the value is being assigned. For example, the following example assigns the string constant `"Hello"` to `StringVar` and the value `10` to `IntVar`:

```
import System;
import System.Collections;

// Valid Initializers
var StringVar:String = "Hello";
var IntVar:Int32 = 10;
var HashVar:Hashtable = new Hashtable();

// Tricky Initializers
var String2Var:String = 23;
var Int2Var:Int32 = "27";

// Invalid initializers
var IntVar:Int32 = "Hello";
```

The `new` keyword creates a new instance of a hashtable. The next assignment is rather tricky—it assigns a numeric value to `String2Var`. You would think that this would result in a compiler error, but JScript converts the numeric constant `23` to the string constant `"23"`. Furthermore, the JScript compiler turns the string constant `"27"` into the numeric constant `27` so that `Int2Var` contains a numeric value. The next couple initializers throw some errors because you can't assign a string constant that can't be parsed as a number into a numeric variable. Notice that the example does not provide invalid syntax for assigning initializer values to a `String` object because any type or value can be converted to a string to be assigned to the `String` object. We'll discuss this further in the section "Using the String Object," later in this chapter.

TIP

Every object in the .NET framework supports the `ToString()` method, which returns the string representation of an object (which can be a name, a serialized version of the object, or any other textual representation). Therefore, assigning any value or object to a `String` object does not throw an error in JScript .NET because JScript converts any objects or initializer values into string values before continuing. Note that this can cause strange values to be assigned to string variables and can sometimes cause code to go down a code path you wouldn't normally expect.

Performance and Strongly Typed Variables

The performance benefits of strongly typing variables are extremely obvious when you look at the process of performing operations on each type of code. You can directly operate on strongly typed variables without the extra level of indirection that is required for traditional JScript variables. In JScript, the underlying object currently referenced by a variable is checked for type and then converted, if necessary, to a type that is compatible with the current operation. In JScript .NET code, the operation occurs automatically. If the type is incompatible, an exception is thrown, and this forces you, as the programmer, to preemptively check variable operations or to wrap them in `try...catch` blocks (discussed in Chapter 8, "Exception Handling") and handle the exceptions. It also means that in well-programmed code, several operations are saved for each variable operation.

A second performance advantage has to do with the JScript compiler and is a direct result of *not* typing variables. The JScript compiler team assumed that there would be quite a bit of legacy code to deal with, and it wanted to see performance benefits without having to recode all the existing samples and programs. The team decided that if the context of a variable and all assignments to that variable could be determined within a local scope, it could guess the type of a variable. Basically, this means that performance-oriented variables, such as loop variables and counters, are strongly typed if they are declared in a local scope. To declare a variable in local scope, you use the `var` keyword on the variable, but it doesn't have to be strongly typed because the compiler infers the type. This option is not available for global variables or variables that aren't declared within a particular scope (because the compiler sets them as global variables).

The Flexibility of Native JScript Variables

All the existing flexibility of native JScript variables is still available in JScript .NET. All `String` variables, for instance, are stored as JScript strings and are converted to CLR strings as needed at runtime. This conversion is completed by some helper functions in the JScript namespace that are called whenever a complex conversion needs to be made. Any variable declared as an `Array` object is considered to be a JScript array, and any variable declared as a `System.Array` object is considered to be a CLR array. The automatic conversions are in place, and interaction with the .NET platform is not an issue because the compiler provides services for conversion of all native JScript types to their equivalent CLR types. So, if flexibility and backward compatibility are issues when you're working with variables, or there is some behavior about classic JScript that you loved to use, then you should feel free use it. Just be aware of the performance issues involved and of the new syntaxes that you can take advantage of for creating faster and smaller code.

Basic Datatypes

Both JScript .NET and the CLR provide many basic datatypes for nearly any operation. This section examines the primitive types available in JScript .NET, as well as their CLR counterparts. This section also discusses some of the complex datatypes, `String` and `Array`, and a few of the CLR types that aren't available in the JScript language but are available through the platform.

Primitive Types

Historically, the primitive datatypes of a language have been defined as datatypes that are directly supported by the language through built-in keywords. In addition to having direct language support, primitive types are generally very compact or well defined. (Complex objects and datatypes that take substantial amounts of initialization code don't fit into the category of primitive datatypes.)

The primitive datatypes available in JScript .NET include the numeric datatypes that support integers and floating-pointer numbers, the Boolean datatype that represents a true/false condition, and several special types, null and undefined, that can be used to test whether variables have been initialized.

Integer Numeric Types

JScript has several built-in keywords for describing various integer number types. Each of these types in turn relates to a type in the CLR. All the methods of the JScript types come from the functionality provided by the CLR. JScript supports the `byte`, `short`, `int`, and `long` keywords. Of these types, the `byte` type is unsigned, and the `short`, `int`, and `long` types are all signed. JScript doesn't have any additional keywords to specify the sign of a variable. Table 3.1 lists each of the JScript types, along with the corresponding CLR type. You'll notice some CLR types without JScript mappings that might come in handy when writing programs.

TABLE 3.1 JScript-to-CLR Type Mapping

JScript .NET Type	*CLR Type*	*Signed*
`byte`	`System.Byte`	No
	`System.SByte`	Yes
`short`	`System.Int16`	Yes
	`System.UInt16`	No
`int`	`System.Int32`	Yes
	`System.UInt32`	No
`long`	`System.Int64`	Yes
	`System.UInt64`	No

You can use the JScript .NET type names and the CLR-type names interchangeably. The CLR has many additional types that aren't available in JScript. Both the `UInt32` and `UInt64` can come in extremely handy for large number calculations or long-running loops.

Listing 3.1 is a small program that prints out the lower and upper bounds of each of the CLR types. These types have static member variables for their bounds, so a simple call to `MinValue` and `MaxValue` can get you exactly what you need.

LISTING 3.1 CLR Integer Type Bounds

```
import System;

// We have to use the actual type names because we are trying to
// make use of the static field members.
Console.WriteLine("{0,-15}{1,-25}{2,-25}", "Type Name",
    "MinValue", "MaxValue");
Console.WriteLine("{0,-15}{1,-25}{2,-25}", "Byte",
    Byte.MinValue, Byte.MaxValue);
Console.WriteLine("{0,-15}{1,-25}{2,-25}", "SByte",
    SByte.MinValue, SByte.MaxValue);
Console.WriteLine("{0,-15}{1,-25}{2,-25}", "Int16",
    Int16.MinValue, Int16.MaxValue);
Console.WriteLine("{0,-15}{1,-25}{2,-25}", "UInt16",
    UInt16.MinValue, UInt16.MaxValue);
Console.WriteLine("{0,-15}{1,-25}{2,-25}", "Int32",
    Int32.MinValue, Int32.MaxValue);
Console.WriteLine("{0,-15}{1,-25}{2,-25}", "UInt32",
    UInt32.MinValue, UInt32.MaxValue);
Console.WriteLine("{0,-15}{1,-25}{2,-25}", "Int64",
    Int64.MinValue, Int64.MaxValue);
Console.WriteLine("{0,-15}{1,-25}{2,-25}", "UInt64",
    UInt64.MinValue, UInt64.MaxValue);
```

NOTE

Listing 3.1 takes advantage of some pretty neat features of the `Console` object. So far, the only code you have seen that uses `Console` object output has been extremely simple. Listing 3.1 uses a format string that is similar to that used with the C function `printf`. Each of the items is then displayed as a parameter. Although we pass three parameters in Listing 3.1, the function can normally handle up to four, and beyond four requires that you pass in a special array of parameters rather than pass each parameter individually.

The format string consists of a parameter format specifier in between each pair of braces. Each format specifier has the offset to the parameter it should use, and then some additional formatting rules are applied (for example, each item take up a certain amount of spaces [either 15 or 25] and each item should be left aligned [by using the minus symbol]). Other formatting rules are available, and you'll see them in notes throughout the book. For more information, examine the Software Development Kit (SDK) documentation because these formatting rules tend to be different in the various types and objects.

Float Numeric Types

For floating-point numbers, JScript supports both the `float` and the `double` keywords. These types map to the CLR types `Single` and `Double`, respectively. The `Single` type represents a 32-bit floating-point number, and the `Double` type represents a 64-bit floating-point number. You can assign values to floating-point numbers by using the decimal representation, or you can use scientific notation. The syntax for scientific notation in JScript .NET is *`number`*`E`*`mantissa`*, where *`mantissa`* is the power of 10 to multiply by the number *`number`*. Listing 3.2 shows the upper and lower bounds of both the `float` and `double` datatypes, using the static fields.

LISTING 3.2 CLR Float Type Bounds

```
import System;

// JScript assignments
var _float:float = 7.333E4;
var _double:double = 1.2E10;

// We have to use the actual type names because we are trying to
// make use of the static field members.
Console.WriteLine("{0,-15}{1,-15}{2,-15}", "Type Name",
    "MinValue", "MaxValue");
Console.WriteLine("{0,-15}{1,-15}{2,-15}", "Single",
    Single.MinValue, Single.MaxValue);
Console.WriteLine("{0,-15}{1,-15}{2,-15}", "Double",
    Double.MinValue, Double.MaxValue);
```

The Boolean Type

JScript represents a Boolean typed variable with the keyword `Boolean`. This object directly converts to the CLR Type `System.Boolean` and takes up 2 bytes worth of space (the same as an `Int16`). Along with supporting a variable datatype, JScript .NET can also coerce nearly any expression to return a Boolean result. This allows for very compact code when you're trying to compare types against what their basic values should be.

For numeric datatypes, both floats and integers, any 0 value returns false. Any nonzero value returns true. This means that when checking against 0 in a conditional statement, you can leave out the comparison operator altogether, as in the following example:

```
var _int:int = 0;
function DoLoops(_int:int) {
    // Check to make sure we are doing at least one loop.
    // If not.  Set the value to one.
    if ( !_int ) { _int = 1; }
    // do something
}
```

You can also check objects to see whether they have been initialized, by using the Boolean type. Any variable that is still `null` and has not been initialized returns false. Any variable that contains an object reference returns true. The following is the best way to check any objects before using them:

```
var _obj:Object;

if ( _obj ) {
    // Do something with the object
} else {
    // We have a bad object. Decide what to tell the user.
}
```

Caution

Be sure you test the preceding example with each of the types that you expect to be used. Depending on the object type, JScript has specific ways of deciding whether to return true or false in the conditional statement. The generic method is to test the object reference against `null`, but some of the primitive datatypes aren't objects and behave differently, depending on the default value for the datatype. So, keep in mind that testing for null is valid only on objects, and not on primitive datatypes.

Special Types: `null` and `undefined`

`null` and `undefined` are actually not special types as much as they are special values of types. `null` and `undefined` behave identically to one another because the JScript compiler handles the appropriate conversion between `null` and `undefined` as necessary. For the most part, you use both `null` and `undefined` to determine whether a variable contains a null pointer or a pointer to an object. For this reason, you can't use this method to compare objects that extend `System.ValueType`, such as the integer types and the Boolean type. Listing 3.3 demonstrates the use of `null` and `undefined`.

LISTING 3.3 Using `null` and `undefined`

```
import System;

var MyInt:Int32;
var MyString:String;
var MyBoolean:Boolean;
var MyObject = undefined;

if ( MyInt == null ) {
    Console.WriteLine("MyInt == null");
}
if ( MyInt == undefined ) {
    Console.WriteLine("MyInt == undefined");
}

if ( MyString == null ) {
    Console.WriteLine("MyString == null");
}
if ( MyString == undefined ) {
    Console.WriteLine("MyString == undefined");
}

if ( MyBoolean == null ) {
    Console.WriteLine("MyBoolean == null");
}
if ( MyBoolean == undefined ) {
    Console.WriteLine("MyBoolean == undefined");
}

if ( MyObject == null ) {
    Console.WriteLine("MyObject == null");
}
if ( MyObject == undefined ) {
    Console.WriteLine("MyObject == undefined");
}
```

```
MyString == null
MyString == undefined
MyObject == null
MyObject == undefined
```

Notice in Listing 3.3 that even though the `Int32` and `Boolean` variables are not given values, they naturally assume default values. `Int32` variables are always 0 and `Boolean` variables are always false. Because the use of `null` and `undefined` is so interchangeable, we will be using `null` throughout the rest of the book when performing checks on whether variables contain real pointers.

Declaring and Typing Arrays

The use of arrays is integral to almost any programming task. JScript has always had a method for creating arrays that can perform any task, but JScript arrays were never very efficient. The new arrays available through JScript .NET, which can exist side-by-side with JScript arrays, offer both performance and flexibility, but never both at the same time.

Declaring JScript and JScript .NET Arrays

The first step in working with arrays is to declare the array. Let's start by declaring a traditional JScript-style array:

```
var _JScriptArray:Array = new Array();

_JScriptArray[0] = "123";
_JScriptArray[2000] = 123;
_JScriptArray[500] = new Object();
```

The type of this array is `Array`, and the example creates a new instance of the `Array` class. Note that all the assignments in this example are valid, and that JScript-style arrays are not strongly typed. Everything that goes in the array is of type `Object`, and everything that comes out is of type `Object`. Therefore, you can assign a `String` object to array index `0`, `int` to array index `2000`, and `new Object` to array index `500`.

JScript arrays are not bounded. New elements can be assigned on-the-fly. As you make assignments, a `length` function is updated so that the upper bounds of the array are always known. This comes in handy when you retrieve a JScript array and need to iterate over the contents. If some of the intermediate values were `null`, then you wouldn't be able to tell the end of the array without the `length` function. JScript arrays are extremely powerful, but there are some performance tradeoffs involved with them.

```
The following example demonstrates a variety of methods for finding the true
end of a JScript array. The first method searches for null:var
_JScriptArray:Array = new Array();
var _int:int = 0;

// Use JScript array like a stack
_JScriptArray[_int++] = "First Element";
_JScriptArray[_int++] = "Second Element";
```

```
// Now let's pass this off to a function that doesn't know its length
DumpArray(_JScriptArray);

function DumpArray(_jsArray:Array) {
    var _i:int = 0;
    while(_jsArray[_i]) {
        Console.WriteLine(_jsArray[_i]);
        _i++;
    }
}
```

This isn't very great code. You need to have a handle on what the top of the array is at all times so that you can add elements on the end. Then you have to search for `null`. Furthermore, if you put an element at index `10`, then the print function would miss it because this function stops at the first `null`. You could also do better with the `length` function, and you could take advantage of a couple other functions in the process. For example, the `push` function allows you to put values on the end of an array and increment the `length` pointer accordingly, and the `pop` function returns the very last element of the array, removes it from the end, and decrements the `length` pointer accordingly. You could rewrite the preceding example as follows:

```
var _JScriptArray:Array = new Array();

// Use JScript array like a stack
JScriptArray.push("First Element", "Second Element");
JScriptArray[9] = "Tenth Element";

// Now let's pass this off to a function that doesn't know its length
DumpArray(_JScriptArray);

function DumpArray(_jsArray:Array) {
    for ( var _i = 0; i < _jsArray.length(); _i++ ) {
        Console.WriteLine(_jsArray[i]);
    }
}
```

This code performs the same operations as the preceding example, but it isn't as error prone. After popping the 2 items on the stack, the 3rd through 9th elements are left unassigned, and the 10th value is assigned. The array dynamically sizes to 10 elements as soon as the 10th element is set. The `DumpArray` function won't stop after the second value and will proceed right through the `null` elements to element 10 and print the contents.

An important feature of JScript-style arrays is that they are *sparse* arrays, which means the elements don't have to be contiguous. In the first example you set elements 0, 500, and 2000. Memory isn't allocated for any other elements besides these three, but this introduces a few problems. For instance, the `DumpArray` function would try to iterate through 2,000 elements, just to print 3 of them. This isn't extremely efficient. CLR-style arrays provide better performance than this, and the next section covers them in detail.

Using CLR-Style Arrays

A CLR-style array has bounds and a definite length, so it can't be expanded and shrunk on-the-fly. This type of array is also strongly typed, meaning you get both compiler and runtime errors for performing incorrect assignments. Every element that comes out of a CLR-style array is of a specific type, so you don't have to check for types before casting. The following example creates a CLR-style array of `String` objects in JScript .NET. The type is a `null` array type, and the `new` keyword is used with a bounded array type to fully initialize the variable:

```
var _CLRArray:String[] = new String[10];

// We now have a CLR array of Strings with 10 elements
_CLRArray[0] = "Hello";
_CLRArray[9] = "World";
_CLRArray[10] = "Produces an error";  // IndexOutOfRangeException
```

This array is now strongly typed to `String` objects. If you attempt to assign any other type of variable (for example, an integer), either the object in question is cast to a `String` object or you get an `InvalidCast` error. All objects support the `ToString()` function, so you will almost never see an `InvalidCast` error when assigning objects or values to a `String` array. This can cause very unexpected results. To get an error by assigning an incorrect type, you need to change to another type of array. For example, you could use an array of `Int32` objects, such as the following:

```
var _CLRArray:Int32[] = new Int32[10];

_CLRArray[0] = 1;
// This works because the string can be parsed as a number_CLRArray[1] = "123";
_CLRArray[2] = "Hello"; // This fails
```

With this code, you get a compiler error on the third assignment. It's a basic type mismatch because the compiler knows `"Hello"` cannot be converted to or parsed as an `Int32` object. A number of other array types will also fail when values can't be converted to or from the appropriate type.

You need to be aware that CLR arrays don't have any of the functions you have seen previously in the JScript arrays. You can't push and pop values onto and off arrays, they aren't sparse, and you can't expand them at will. They do have some other properties, though. By using the `Length` property, you can determine the number of elements in an array. You can use the `CopyTo` function to transfer array contents from one array to another. You can use the `Rank` property to determine the number of dimensions an array contains. Take a look at Listing 3.4, which demonstrates the use of the `Length` property and the `CopyTo` function.

LISTING 3.4 Dynamically Sizing Fixed Arrays

```
import System;

var _WorkArray:Int32[] = new Int32[10];
var _WorkOffset:Int32 = 0;
var _CopyArray:Int32[];

while(_WorkOffset < 50) {
    while(_WorkOffset < _WorkArray.Length) {
        _WorkArray[_WorkOffset] = _WorkOffset;
        _WorkOffset++;
    }

    if ( _WorkArray.Length < 50 ) {
        _CopyArray = new Int32[_WorkArray.Length + 10];
        _WorkArray.CopyTo(_CopyArray, 0);
        _WorkArray = _CopyArray;
        Console.WriteLine("Expanded Array to " + _WorkArray.Length);
    }
}

_WorkOffset = 0;
while(_WorkOffset < _WorkArray.Length) {
    Console.WriteLine(_WorkArray[_WorkOffset]);
    _WorkOffset++;
}
```

Listing 3.4 might seem to be quite a bit of code this early in the book. After all, you have not been introduced to `while` loops, conditional statements, and a number of other features that are used in this listing. So let's walk through the most important parts of Listing 3.4. First, it creates two arrays of type `Int32`. The `work` array is given bounds of `10`. The `copy` array is not bounded and remains uninitialized until you fill up the `work` array. At this point, the `copy` array is declared to have bounds 10 greater than the `work` array, the contents of the `work` array is copied to the `copy` array, and the `copy` array is assigned to the `work` array so that work can continue.

You'll learn more about CLR-style arrays in the sections "Typing Arrays for Performance" and "Multidimensional Array Support" later in this chapter.

Typing Arrays for Performance

JScript-style arrays are slow because they aren't strongly typed. When you get an element in an array that isn't typed, you have no idea what kind of object you're getting. It could be a string, a hashtable, another array, or even a number. You just don't know, and neither does the computer. Therefore, you need code to do type checks and appropriate conversions before the value can be used.

CLR arrays are always typed. In order to create an instance of an array, the underlying type must be given first. Typed arrays can be created in two manners. Earlier in this chapter we talked about the language-specific manner of creating a typed array, which involves using a `null` array type to which you can later assign bounds by creating a new instance of a typed array. Another method is to create a variable of type `System.Array`. Later, you can assign it a typed array by using the `System.Array.CreateInstance` function. This method is extremely powerful because the type of array can be decided at runtime; however, this method also increases the chance of runtime exceptions because array assignments can't be typed and checked during compilation. The following example shows how to use both of these methods of creating typed arrays:

```
import System;

var _IntegerArray:Int32[] = new Int32[10];
var _StringArray:String[] = new String[10];
var _GenericArray:System.Array;

_GenericArray = _IntegerArray; //Success!
_GenericArray = _StringArray; //Success!

_GenericArray = System.Array.CreateInstance(
    Type.GetType("System.String,mscorlib"),
    10);
```

TIP

A generic CLR array has to be declared by using type `System.Array`. JScript interprets `Array` to mean a JScript array, even if the import `System` directive is present. This is for legacy purposes and prevents users from typing `Microsoft.JScript.Array`.

Note several neat things in the preceding example. First, you have two typed arrays that give maximum performance to the application. But you can also assign these arrays to the `_GenericArray` variable because it is of type `System.Array`, and all typed arrays inherit from this base class. (You'll learn more about inheritance in Chapters 6 and 7, "Interfaces and Class Members.") `_GenericArray` can be of any array type, but regardless of type, it has the performance of a typed array, without the compile-time warnings that would exist if you made a poor assignment to an array element. Notice that this example uses a special function on the `System.Array` class to create a `String` array. This looks pretty ugly, because it involves several levels of function calls, but it shows that arrays can be created with indeterminate types at runtime. In the example, you have to know the type of the `Int32` and the `String` arrays, but you could have created any type of array by using the `CreateInstance` function. You can also create arrays that have a variable number of dimensions, as discussed in the following section.

Multidimensional Array Support

JScript .NET supports two types of multidimensional arrays. The first type is a standard equal-bounds array, in which all rows in the array are of the same length. The second type of array is a staggered array, which can have rows of different lengths. Staggered arrays can be effective when you're creating arrays in which some rows might have thousands of elements whereas other rows may contain very few or no elements.

Let's look first at the most common type of array, the equal-bounds array. In the type specifier for this array, you simply add one comma between the brackets for each additional dimension you need to introduce. In the `new` expression, you provide the bounds for each of the dimensions. This syntax makes creating a multidimensional array very easy. The following code demonstrates the creation of a 10-by-10 array, whose elements you initialize to create a multiplication table:

```
import System;

var myArray:Int32[,] = new Int32[10,10];
var i:int;
var j:int;

for(i = 0; i < 10; i++){
    for(j = 0; j < 10; j++){
        myArray[i,j] = (i+1)*(j+1);
    }
}
```

TIP

An overloaded method of `Array.CreateInstance` allows for the creation of multidimensional arrays as well, and it is often useful when you're creating an array of a type that is not yet known or when the user can specify the number of dimensions (which might be the case with some math applications).

To use this array, you index the array by using an offset for each dimension of the array. You can continue adding offsets for any number of additional dimensions.

NOTE

Although JScript supports a staggered array, we don't examine the syntax here. It's rather difficult to work with staggered arrays, and there are very few cases in which they are useful and necessary.

Using the `String` Object

The `String` object is probably the most widely used and also the most complex object available in JScript .NET. The semantics surrounding this object are fairly complex because there is only one object that is converted on-the-fly between a JScript `String` object and a CLR `String` object. All the functions and properties of both objects are available to programmers, and so are all legacy JScript semantics. In other .NET languages, it is very easy to cause exceptions by using a `String` object. In JScript .NET, you very rarely see this because the compiler does conversions, casting, and initializations for you. This section examines the important aspects of `String` objects.

Declaring Strings

Two types in JScript (`String` and `System.String`) evaluate to the same `String` object. This is because a JScript `String` object and a CLR `String` object both map to a CLR `String` object internally. Then, special handler functions are called whenever a JScript `String` object function is called. Therefore, the programmer sees great performance from CLR `String` objects as well as great flexibility because existing code that works on the `String` object continues to work. Here's an example of calling both JScript `String` methods and CLR `String` methods on the same `String` object:

```
var MyString:String = "Hello";

MyString.toString();
MyString.ToString();
```

This example calls both `toString()`, which is a JScript function and `ToString()`, which is a CLR function. For the `toString()` function, a special call is made into a JScript helper library, and the string is printed according the rules of JScript. The `ToString()` function, on the other hand, is called directly from the `String` object and returns according to CLR rules. The two functions behave identically, but they take different code paths. To keep things simple, let's assume there is only one `String` object and that it simply has two sets of functionality that happen to overlap in certain areas. Where functionality overlaps, we'll be using the CLR versions because they map directly to the underlying object and should be a bit quicker than the JScript versions.

String Manipulation

The `String` object hosts quite a few methods for manipulating existing strings, creating new strings, and performing quite a few search-and-replace functions for string parsing. At this point, note that any modification to a `String` object results in a new `String` object being created because all `String` objects are immutable and thread safe. You will never run into work on a `String` object and have it change on you.

All the string manipulation functions can be grouped into several categories:

- **The JScript set of functions**—These are functions that retain their meanings from JScript and still exist as part of the JScript language.
- **The instance methods of the `String` object**—These are methods than you can call when you have an existing `String` object and want to perform some operations on it.
- **The static `String` object methods**—These methods accept parameters and return a new string as a result.

Each of the following sections provides short descriptions of these functions and a brief example. Some of the examples include output, but in most you should run the code yourself to see the results.

JScript Functions and Properties

Each of the functions and properties discussed in this section exists for compatibility with legacy JScript code. The only nonlegacy property is the `length` property, which gives the length of a string, in characters. All the old JScript functions that operate on the `String` object exist in JScript .NET. This can be somewhat strange because in many cases these functions were used when interacting with Hypertext Markup Language (HTML) and return an HTML string result. The following functions are some of the commonly used HTML methods of the `String` object:

- **`anchor`**—The `anchor` method surrounds the current string with an `<anchor>` tag and returns the result as a new string. The string parameter passed to the `anchor` method becomes the value of the `name` attribute.
- **`big`**—The `big` method surrounds the current string with a `<big>` tag, which increases the size of the string in HTML by one font size and returns the result as a new string.

- **`blink`**—The `blink` method surrounds the current string with a `<blink>` tag, which causes text to blink in most browsers and returns the result as a new string.
- **`bold`**—The `bold` method surrounds the current string with a `<b>` tag and returns the results as a new string.

These are just a few of the HTML-related methods, and they have all been part of the JScript standard for quite some time and are well documented both in other books on JScript and on the Web. At this point we will skip the rest of the HTML functions and move on to functions that have CLR equivalents or that can be used for more generic programming than just HTML.

This next set of functions is very helpful in retrieving pieces of the current string:

- **`charAt`**—Sometimes you might want only a character, the numeric value of a character, or only a portion of a string. The `charAt` method takes the 0-based index of the character to return within the string. If the offset in the string doesn't exist, the method simply returns an empty string.
- **`charCodeAt`**—The `charCodeAt` method behaves identically to the `charAt` method, except that it returns the numeric representation of the character rather than a string. If the offset does not exist, `charCodeAt` returns the value `NaN`.
- **`fromCharCode`**—The `fromCharCode` method is related to `charCodeAt`, and turns a list of character codes into a string. This is generally helpful when you're trying to shift the values of characters up or down a few places to perform a primitive form of cryptography.

Listing 3.5 demonstrates each of these functions in use. Note that the Caesar shift functionality (which is a form of simple encryption) of the last part of the listing doesn't work very well, but it handles the encryption (if you can call it that) of Unicode characters outside of the English character set.

LISTING 3.5 Character Manipulation in Strings

```
import System;

var MyString:String = "Getting Character values from Strings";

// Here we are going to get some characters out of the string
for(var i = 0; i < 7; i++) {
    System.Console.Write(MyString.charAt(i));
}
System.Console.WriteLine();

// Now lets print their Unicode values
for(var i = 0; i < 7; i++) {
    System.Console.Write(MyString.charCodeAt(i) + ",");
```

Listing 3.5 continued

```
}
System.Console.WriteLine();

// Here we are going to do a modified Caesar shift
// Modified because it will use computer characters
// and won't wrap the alphabet (a won't wrap to z)
var MyShift:Int32 = 5; // Lets shift by 5 characters
for(var i = 0; i < MyString.length; i++) {
    System.Console.Write(
        String.fromCharCode(MyString.charCodeAt(i) + MyShift)
    );
}
System.Console.WriteLine();
```

The character code methods in Listing 3.5 are great for working with single characters when character offsets and values are known. To help find character offsets and search strings, you use the `indexOf` and `lastIndexOf` methods. These methods find the first or last occurrences of a substring, starting at a given index. The `indexOf` method takes the substring to search for within the string and an optional beginning index. If it is left `null`, the beginning index is `0` or the first character in the string. The result is the beginning of the substring within the current string. If the substring is not found, the result is `-1`. The `lastIndexOf` method is identical to `indexOf` method. However, this method searches the string, starting from the end, which means that the second optional parameter, rather than 0, is by default the last character. The return values of `indexOf` and `lastIndexOf` are the same.

To retrieve substrings, you use another set of functions. The `substr` and `substring` methods provide different ways of getting part of the current string:

- **`substr`**—The `substr` method takes the beginning index of the string to retrieve and an optional length. If the index is past the end of the string, an empty string is returned. If the length is `0` or negative, an empty string is returned. If the length proceeds past the end of the string or the length is not specified, the function retrieves a substring from the offset to the end of the string.
- **`substring`**—The `substring` method behaves identically to `substr`, except that in place of the length, it takes the ending index. If you make a mistake and pass in a higher starting index than ending index, JScript automatically swaps the parameters for you. The string returned includes the starting index up to, but not including, the ending index.

Finally, a less well-known method, `slice`, has the same parameters as `substring` as well as the same return results for normal operation. The one difference is that the `slice` method has a different set of rules for treating starting and ending values that are out of bounds for the current string. These rules allow you to index from the end of the string by specifying negative values for either the starting or ending indexes.

Some of these methods might seem strange. Look at Listing 3.6 for an example of their use.

LISTING 3.6 Capturing Substrings of a String

```
var MyString:String = "Parsing strings and obtaining substrings";
var MyStartIndex:Int32;
var MyEndIndex:Int32;

// Success and Failure of indexOf
System.Console.WriteLine("Index of (substring) and (notinstring)");
System.Console.WriteLine(MyString.indexOf("substring") + "," +
    MyString.indexOf("notinstring"));
System.Console.WriteLine();

// indexOf versus lastIndexOf
System.Console.WriteLine("Index of (string) and Last Index of (string)");
System.Console.WriteLine(MyString.indexOf("string") + "," +
    MyString.lastIndexOf("string"));
System.Console.WriteLine();

// substr Semantics
System.Console.WriteLine("The method substr");
System.Console.WriteLine(MyString.substr(8));
System.Console.WriteLine(MyString.substr(8,5));
System.Console.WriteLine(MyString.substr(8,100));
System.Console.WriteLine(MyString.substr(8,-1));
System.Console.WriteLine();

// substring versus slice
System.Console.WriteLine("The method substring versus slice");
System.Console.WriteLine(MyString.substring(8,14) + "," +
    MyString.slice(8,14));
System.Console.WriteLine(MyString.substring(8) + "," +
    MyString.slice(8));
System.Console.WriteLine(MyString.substring(8,-1) + "," +
    MyString.slice(8,-1));
System.Console.WriteLine(MyString.substring(-5,8) + "," +
    MyString.slice(-5,8));
System.Console.WriteLine();
```

There are only a few remaining functions—the replacement functions and a few miscellaneous, useful functions. The `replace` function in JScript runs against the current `String` object. It replaces a regular expression pattern with a replacement string. The regular expression pattern can be inline, or it can be a regular expression object defined by JScript.

NOTE

Chapter 12, "Regular Expressions," discusses replacement strings and regular expressions.

The replacement string can contain replacement text or escape characters that can be used to include portions of the matched text in the replacement text. (You'll learn about the advanced portions of this parameter, including several ways of manipulating the replacement string, in Chapter 11. At this point, we will use very basic strings as replacements.)

The two remaining functions we'll explore here are `toLowerCase` and `toUpperCase`. These functions perform the roles their names suggest. The `toLowerCase` function returns a copy of the current string with all characters converted to their lowercase equivalents. The `toUpperCase` function returns a copy of the current string with all characters converted to their uppercase equivalents. Listing 3.7 uses the `replace` function with a simple case-sensitive set of regular expressions to match and perform replacements against strings that are modified with the `toUpperCase` and `toLowerCase` functions.

LISTING 3.7 Basic JScript Regular Expressions

```
import System;

var reUpperCase = /MATCH/g;
var reLowerCase = /match/g;
var reNoCase = /MaTcH/ig;

var strCamelCase:String = "Where Is The Match In This String?";

// Let's try to Replace Match with Sasquatch in the string
Console.WriteLine(strCamelCase.replace(reUpperCase, "Sasquatch"));
Console.WriteLine(strCamelCase.replace(reLowerCase, "Sasquatch"));
Console.WriteLine(strCamelCase.replace(reNoCase, "Sasquatch"));
Console.WriteLine();

// Now let's make a lowercase version of the string and do it again
var strLowerCase:String = strCamelCase.toLowerCase();
Console.WriteLine(strLowerCase.replace(reUpperCase, "Sasquatch"));
```

LISTING 3.7 continued

```
Console.WriteLine(strLowerCase.replace(reLowerCase, "Sasquatch"));
Console.WriteLine(strLowerCase.replace(reNoCase, "Sasquatch"));
Console.WriteLine();

// And one final time, with the uppercase string
var strUpperCase:String = strCamelCase.toUpperCase();
Console.WriteLine(strUpperCase.replace(reUpperCase, "Sasquatch"));
Console.WriteLine(strUpperCase.replace(reLowerCase, "Sasquatch"));
Console.WriteLine(strUpperCase.replace(reNoCase, "Sasquatch"));
Console.WriteLine();
```

CLR String Instance Methods and Properties

This section covers all the instance methods and properties that you can use with a `String` object. It also discusses the set of methods and properties that you can use without an instance of a string.

Let's start by examining all the different ways you can get an instance of a `String` object. JScript allows you to create a string from a string literal in code. But the CLR can create strings from arrays of characters, from string literals, or from a single character repeated many times. These methods of creating `String` objects are best demonstrated via code. The following example uses several `String` object methods to create character arrays:

```
import System;

// Creation of a literal string
var LiteralString:String = "Hello World";
Console.WriteLine(LiteralString);

// Creation of a Character string created from the LiteralString
var CharString1:String = new System.String(LiteralString.ToCharArray());
Console.WriteLine(CharString1);

// Creation of a Character string using only parts of the LiteralString
// 0 is the offset to begin in the array, and 5 is the length.
var CharString2:String = new System.String(LiteralString.ToCharArray(), 0, 5);
Console.WriteLine(CharString2);

// Creation of a Character string by repeating a single character
// 20 means to repeat the character 20 times
var CharString3:String = new System.String('0', 20);
Console.WriteLine(CharString3);
```

This example converts a string literal to a character array and uses the character array to then create a `String` object. The method used in this example is simpler than creating a character array from scratch.

This example creates two `"Hello World"` strings, one `"Hello"` string, and a string of 20 zeros. Now that you have some `String` objects to play with, you can begin calling some of the methods and properties. To begin, you can get the length of any string in characters (not in bytes), by calling the `Length` property. No matter how the string is stored, you can always get the number of characters by using this property. You can then get individual characters from the `String` object by using the `Chars` property, which is an indexed property (that is, you can treat it just like an array). Simply passing in an array index retrieves a character. Make sure you use numbers in the range of 0 to `Length -1`, or you will get a runtime error, specifying that an invalid parameter was passed into the property. This is one of the many features of the strongly typed CLR datatypes.

All the string manipulation functions discussed earlier in the chapter are also made available here, in their CLR variants. The string constructors that use a character array to build a string and the `Chars` property are directly equivalent to the JScript character code functions.

The string search functions consist of the familiar `IndexOf` and `LastIndexOf` functions. Both of these functions can take a substring to search for, take a substring and a beginning offset, or take a substring with the beginning and ending offset. The major difference between the two functions lies in the ability to search for a single character and in being able to search for any given number of characters (for instance, `a` or `b` or `c` as opposed to the string `"abc"`). All versions of these functions—whether taking a substring, character, or array of characters—optionally accept a starting offset and an ending offset. If a string or character match is found, the offset within the string is returned; otherwise, `-1` is returned. (This is the same behavior as in the JScript functions.)

The CLR offers the functions `StartsWith` and `EndsWith`, which are shortcuts for comparing substrings against the beginning or end of a given string. The return value is a Boolean true or false, depending on whether the string matched. However, these functions are often useful only after the string has been fully *normalized* (that is, any whitespace at the beginning or end of the string has been removed so that the string match can happen). To aid in removing extra characters, the CLR provides the trim functions. The `Trim` function trims whitespace from the beginning and ending of a string, or it trims an array of characters from the string instead of trimming just whitespace. The `TrimStart` and `TrimEnd` functions take an array of characters or a `null` value to indicate whitespace. With this set of functions, you can trim any given number of characters or amount of whitespace from the beginning and end of a string.

The CLR also has padding functions. You can use the `PadLeft` and `PadRight` functions to add padding to each side of a string. To use either function, you simply pass in the length you want the string to be, and the function will pad the string with the needed spaces. You can pass an optional parameter in the form of a single character to specify padding with something other than a space. For example, later in this section, we'll be using an uppercase A instead of a space character to pad strings.

The CLR also provides the `Substring` function. Remember that JScript has three separate functions for getting substrings, each of which treats parameters in a different way. The CLR needs only one function for obtaining substrings, so if you want JScript-like functionality with multiple functions that do nearly the same thing, use the JScript functions. But be warned that they simply map back into the single CLR `Substring` function in the end. The `Substring` function takes just a starting index and returns up to the end of the string. It optionally takes a length parameter and returns the number of characters specified in the length parameter rather than returning to the end of the string. Note that if the starting index plus the length goes beyond the end of the string, the program throws a runtime exception. So in many cases, this function is not as nice as the JScript wrappers.

Listing 3.8 demonstrates all the functionality of the methods talked about so far.

Listing 3.8 CLR String Methods and Character Arrays

```
import System;

var Searchable:String = "This is our searchable string!";
var CharString:String = "aeiou";

// Demonstrates of the basic IndexOf and LastIndexOf functions
Console.WriteLine(Searchable.IndexOf("is"));
Console.WriteLine(Searchable.LastIndexOf("is"));
Console.WriteLine();

// Using Character Arrays.  Will return vowels
Console.WriteLine(Searchable.IndexOf(CharString.ToCharArray()));
Console.WriteLine(Searchable.LastIndexOf(CharString.ToCharArray()));
Console.WriteLine();

// Advanced indexes for IndexOf and LastIndexOf functions
// These will return the second occurence of the string is from the
// beginning and from the end using the first search for the beginning
// offset of the second search.
Console.WriteLine(Searchable.IndexOf("is", Searchable.IndexOf("is") + 1));
Console.WriteLine(
    Searchable.LastIndexOf("is", Searchable.LastIndexOf("is") - 1)
);
Console.WriteLine();
```

LISTING 3.8 continued

```
// Set up variables for use with EndsWith and StartsWith
var NonPaddedString:String = "Match me in the beginning and the end.";
var PaddedString:String = "AAAAMatch me in the beginning and the end.AAAA";

Console.WriteLine(PaddedString.StartsWith("Match"));
Console.WriteLine(PaddedString.EndsWith("end."));
Console.WriteLine();

Console.WriteLine(NonPaddedString.StartsWith("Match"));
Console.WriteLine(NonPaddedString.EndsWith("end."));
Console.WriteLine();

// Trim Some variables
var TrimString:String = PaddedString.Trim("A".ToCharArray());
var TrimEndString:String = PaddedString.TrimEnd("A".ToCharArray());
var TrimStartString:String = PaddedString.TrimStart("A".ToCharArray());

Console.WriteLine(TrimString);
Console.WriteLine(TrimEndString);
Console.WriteLine(TrimStartString);
Console.WriteLine();

// Pad some variables
var PadLeftString:String =
    NonPaddedString.PadLeft(NonPaddedString.Length + 4, 'A');
var PadRightString:String =
    NonPaddedString.PadRight(NonPaddedString.Length + 4, 'A');

Console.WriteLine(PadLeftString);
Console.WriteLine(PadRightString);
Console.WriteLine();

// And finally some Substrings
var Substring1:String = PaddedString.Substring(4);
var Substring2:String = PaddedString.Substring(4, PaddedString.Length - 8);

Console.WriteLine(Substring1);
Console.WriteLine(Substring2);
Console.WriteLine();
```

Another set of instance functionality that you should understand is the conversion functions, which behave identically to the JScript `to*` functions. The CLR maintains a `ToUpper` method and a `ToLower` method similar to the JScript equivalents. You've already seen the `ToCharArray` function, which allows you to convert a string into an array of its constituent characters. This comes in handy when you need to create character arrays from strings on-the-fly, as we've done several times in this chapter.

You also need a set of functions that work on the current string to add, remove, and insert substrings. The CLR has instance methods for removing and inserting strings, but it doesn't have one for appending or concatenating strings together. However, you can use the `Insert` method to make up for this shortcoming. The `Insert` method works on the current string and takes a starting index along with the substring to be inserted. You can use this function for appending a substring by setting the starting index to the very end of the current string. An associated `Remove` method takes the starting index and a length to specify a substring to be removed. Together, these two functions are capable of performing most string maintenance tasks.

You can compare any `String` object to any other `String` object by using the `CompareTo` method. The `CompareTo` method can also be used to compare any CLR objects.

The following is a fairly simple code example that shows how to do basic string manipulation by using the `Insert` and `Remove` methods

```
import System;

var TargetString:String = "Here is the final string!";
var StartString:String = "is the wrong stuff!";

// Lets go ahead and start the string off correctly
StartString = StartString.Insert(0, "Here ");
Console.WriteLine(StartString);
Console.WriteLine(TargetString.CompareTo(StartString));

// We need to get rid of that ending
StartString = StartString.Remove(12, 12);
Console.WriteLine(StartString);
Console.WriteLine(TargetString.CompareTo(StartString));

// Now lets finish this bad boy off
StartString = StartString.Insert(StartString.Length, "final string!");
Console.WriteLine(StartString);
Console.WriteLine(TargetString.CompareTo(StartString));
```

In this example, notice the results of the `CompareTo` method, which keeps working until the strings are equal. The value `-1` means that the current string is less than the string being compared to, the value `1` means that the current string is greater than the string being compared to, and the result `0` means the two strings are equal.

CLR String Static Methods and Properties/Fields

There are only a couple helpful CLR string static methods and properties worth mentioning, so this section is short and to the point. There is only one field of note: the `Empty` field. The `Empty` field returns an empty string, and it is mainly used for comparing the current string against an empty string or for initializing string variables with nothing in them so that you can add to them later, using the string modification methods.

It can be useful to be able to print out a list of strings delimited with spaces, tabs, or commas. The `Join` method makes this pretty easy. With a separator string and an array of strings to join, the `Join` method returns a single string with all the strings in the array separated by the string.

What if you want to have a string returned in a special format? In this case, you need to use the `Format` method of the `String` object. The `Format` method enables you to pass in all sorts of formatting parameters, along with a number of objects. This method works on strings, numbers, and many other types of objects. The actual formatting string can be quite complex, but the general syntax is in the form of a string containing special formatting characters. These formatting characters begin and end with braces, in the form `{`*`numberofobjects`*`,` *`formatoptions`*`}`, where *`numberofobjects`* is the number of the object to format starting from 0, and *`formatoptions`* is a special set of formatting options that is passed to the object, allowing it to format itself (or not format itself, as the case may be).

Finally, the `Compare` method behaves identically to the `CompareTo` instance method. However, `Compare` lets you pass in the two strings to be compared rather than work against a string instance. This is handy when you need to compare two parameters to a function or compare command-line options. Listing 3.9 shows examples of formatting, joining, and comparing some numbers and converting them to strings. Numbers format very well and support quite a few format specifiers, so they work best when demonstrating the `Format` function.

LISTING 3.9 String Comparisons and Formatting

```
import System;

// We need an array of Integers
var i:Int32 = 0;
var IntArray:Int32[] = new Int32[6];
var StringArray:String[] = new String[6];
var ObjectArray:Object[] = new Object[6];

IntArray[0] = 1;
IntArray[1] = -1;
IntArray[2] = 2;
IntArray[3] = -2;
```

LISTING 3.9 continued

```
IntArray[4] = 100000;
IntArray[5] = -100000;

// Let's join some Strings together
for(i = 0; i < 6; i++){
    // Lets cast to a string
    StringArray[i] = String(IntArray[i]);

    // And an Object
    ObjectArray[i] = Object(IntArray[i]);
}
Console.WriteLine(String.Join("|sep|", StringArray));
Console.WriteLine();

// Let's compare some strings together
for(i = 0; i < 6; i++,i++){
    // Print out the results of string comparisons
    Console.WriteLine(String.Compare(StringArray[i], StringArray[i+1]));
    Console.WriteLine(String.Compare(StringArray[i], StringArray[i]));
}
Console.WriteLine();

// Let's finish up with the formatting
var FormatString:String =
  "Hex the number\n{0:X}\n" +
  "Left align 20 characters\n{4,-20}\n" +
  "Right align 20 characters\n{5,20}\n";

Console.WriteLine(String.Format(FormatString, ObjectArray));
Console.WriteLine();
```

Using `StringBuilder` Instead of `String`

Now that we have explored the greatness of the `String` object, let's talk about what isn't so great about it. The problem with a `String` object is that every time it is modified, a brand new string is created. This happens so that any piece of a program that is working on or examining the older version of the string doesn't have it magically change. It's a nice feature of the CLR to make strings immutable and thread-safe.

It is fairly obvious that any process that concatenates many values to a single string, or works on very large strings with even a few items being appended to the end, needs more performance than does a process that copies the old `String` object with any new changes every time a modification is made. The CLR `StringBuilder` class is exactly what you need in this case. The `StringBuilder` class resides in the `System.Text` namespace and has only a few methods and properties that are of interest to us.

To add to the current string, you use the `StringBuilder` class's `Append` method. This method is overloaded and can take a piece of a string, an integer value, or even a Boolean variable. The `AppendFormat` method allows you to specify a formatting string along with the variables to be formatted. This is similar to the format string that is used elsewhere in the book with the `Console` methods. For more information on formatting, take a look at the .NET framework SDK documentation.

You can insert and remove values from a string. The `Insert` method takes a character offset in the string and begins placing either a string you've passed in or another value. The `Insert` method doesn't have a formatted version, so you have to do any formatting of the string beforehand. The `Remove` method takes both a starting index and a length for the string to be removed.

`Append` also includes some utility methods. The replace methods replace a given string, integer, or character with another. These are highly specialized and extremely quick replace functions. A more powerful alternative would be to use regular expressions, but for simple character replacement, the replace methods are fastest. Another important utility function is `ToString()`. Though this has to be exported by every object in the CLR, its use here is extremely important because it is the only way of turning a `StringBuilder` into a string that you can use elsewhere in an application.

Some of the properties of the `StringBuilder` object are also pretty neat. Each `StringBuilder` object has a `Capacity` property and a `MaxCapacity` property, which control how much space is allocated for the current string and how much space can be allocated as the string grows, respectively. If you know a string is going to grow to a certain size, it can be extremely helpful to set the capacity in the beginning. This prevents the `StringBuilder` object from having to grow as you expand the string and enables you to preallocate the space needed. You can pass an indexed property, `Chars`, the index of a character to return. It behaves like the `charAt` function of the JScript string. Finally, you can use the `Length` property to find the number of characters currently loaded into the `StringBuilder` object.

Listing 3.10 shows how to use the `StringBuilder` object.

LISTING 3.10 Using `StringBuilder` to Create Strings

```
import System;
import System.Text;
```

LISTING 3.10 continued

```
BuildString();

function BuildString() {
    var MySB:StringBuilder = new StringBuilder();

    // Appending normal text
    MySB.Append("Lets start with this\nLittle bit of text\n\n");

    // Appending Formatted text
    MySB.Append("Here we are formatting some numbers\n");
    MySB.AppendFormat("{0:X}, {1,10}, {2}\n\n", 500, 300, 10);

    // Using the Insert method
    MySB.Insert(0, "This text is getting inserted\n\n");

    // Removing some text
    MySB.Insert(0, "This is getting inserted to be removed\n");
    MySB.Remove(0, "This is getting inserted to be removed\n".Length);

    // Let's make numbers a bit bigger!
    MySB = MySB.Replace("numbers", "NUMBERS");

    // And Finally lets examine the Properties
    MySB.AppendFormat("Capacity:     {0}\n", MySB.Capacity);

    // Let's set a higher capacity
    MySB.Capacity = MySB.MaxCapacity / 100;

    MySB.AppendFormat("New Capacity: {0}\n", MySB.Capacity);
    MySB.AppendFormat("Max Capacity: {0}\n", MySB.MaxCapacity);
    MySB.AppendFormat("Length:       {0}\n", MySB.Length);
    MySB.AppendFormat("Some Chars:   {0} {1} {2}\n",
        MySB.Chars(0),
        MySB.Chars(1),
        MySB.Chars(2));

    // Notice the use of ToString()!
    Console.WriteLine(MySB.ToString());
}
```

Summary

In this chapter you have learned about all the basic datatypes available in JScript .NET, along with quite a few of the object types that are available through the CLR. Services are provided in two locations in JScript .NET: through the JScript .NET language and through the CLR, which hosts the JScript .NET language while it is running. You'll make use of both of these services throughout the book because they are both important to effective .NET programming.

New performance-oriented options are available in JScript .NET. For example, if you strongly type code, compile-time warnings appear if a variable assignment is inappropriate. You can also use bounded arrays through the CLR, as opposed to the JScript .NET dynamic arrays. Bounded arrays offer more performance than dynamic arrays, but dynamic arrays offer more functionality and are more convenient.

In this chapter you learned how to use strings and how the new `StringBuilder` class can provide great memory usage and speed improvements when you're building long strings from many smaller parts.

In Chapter 4, "Operators and Commenting," you'll learn about the various operators available in JScript .NET, including the arithmetic, assignment, comparison, Boolean, and bitwise operators. You'll also learn about some of the object-oriented operators that allow you to create new instances of objects and find out what type of object you're working with.

CHAPTER

4

Operators and Commenting

IN THIS CHAPTER

In Chapter 3, "Datatypes, Arrays, and Strings," you created variables of all different types, and now you need to do some programming to make them work. Operators allow you to operate on the variables and data structures that you have defined. Sample code in both Chapter 2, "Organization of JScript .NET," and Chapter 3 makes use of various operators; there is simply no way around addressing these frequently used elements, no matter what kind of simple code you create. You've seen various equality operators (as well as equality functions), the assignment operator, a few of the arithmetic operators, and even some operators you don't commonly think of as operators (such as the `new` operator). This chapter examines each of the operators available within JScript .NET, their purposes, and any artifacts that might creep in from legacy support of the old JScript language.

We'll start by discussing the comparison and assignment operators, which you use every time you need to change or check the value of a variable. You'll learn about the `new` operator, which you primarily use when assigning a new instance of an object to a variable.

The arithmetic operators allow for the mathematical modification of numerical variables. Several of the arithmetic operators can also be used in a special manner, so that they act like assignment operators. The concatenation operator, which is an arithmetic operator for strings and characters, is very useful for building strings through a section of code to produce literal output.

Bitwise operators are used for exactly what their name suggests: working on the bits of a number rather than on the number itself. This comes in handy for dealing with structured return values where instead of needing a single return value, you need a series of flags to be returned. Boolean operators are used to combine conditional statements together. Almost entirely reserved for use in the conditional statements of `while` loops and `if` statements, these operators combine the true/false return values of a conditional statement to give a single collaborative value.

At the end of the chapter we'll discuss commenting code, including the primary code constructs for commenting, where you can and can't use comments, and some guidelines for commenting that might come in handy when you're programming with JScript .NET. You'll also learn the infamous C# documentation comments, which probably won't make it into the JScript language, but whose style is important. Using documentation comments similar in style to the C# documentation comments adds a degree of cross-language support and provides a familiar interface for programmers who program in languages other than JScript .NET.

Comparison and Assignment Operators

A *conditional statement* is a code construct that returns a true/false value based on the result of the conditions, and the comparison operator is the core of all conditional statements. The most simple of all comparisons is whether a variable is equal to another variable. In JScript .NET, you use the equality operator, ==, to make this comparison.

There are three main types of variable comparison:

- The first type encompasses all the various numerical types and a set of rules for converting the two values to a common type before the comparison is made.
- The second type involves the comparison of strings. String comparison is rather trivial in the common language runtime (CLR), in that an assembly (as described in Chapter 2) holds a string table. Whenever a new string is created, it is given a new descriptor in the table. However, if a string with the same contents as an existing string is created, the descriptor for the existing string is supplied. In this manner, two strings with the same contents contain the same descriptor in the assembly string table.
- The final type of comparison involves the comparison of various `Object` types. Strings can essentially fall into this scenario under the CLR because they are of type `Object`. All `Object` types implement `Equals` methods that are called to compare one `Object` type to another. This method can examine the type of object it is comparing against and apply special rules, it can use a string representation of the object for comparison purposes, or it can use a number of other functions or algorithms. The important thing to note is that when you use the equality operator, JScript handles all the conversions, and it calls the `Equals` function for you.

The previous operator just handles the equality portion of the equation. To determine whether two variables are not equal to one another, you use the inequality operator, `!=`. This syntax resembles the negation operator (see the section “Boolean Operators” later in this chapter) along with the equality operator. This operator reverses the Boolean result of the equality function. The same effect can be accomplished by negating the result of an equality function as shown in the following code:

```
var a:int = 1;
var b:int = 2;
var result:Boolean = false;

// These two comparisons are equivalent
result = (a != b);
result = !(a == b);
```

You can review the values in the result by placing some Console code after each of the assignments. The result in both cases is true. In the first case, `a != b`, `a` is certainly not equal to `b`, so the comparison evaluates to true. In the second case, `a == b` (that is, `a` equals `b`) is false. You then take the negation of that false value, which returns true. Either way, you get the same results, but using the inequality operator is the preferred method because it makes use of a single operator rather than two and executes more quickly than the two-operator version.

The next set of operators deals with relative equality. Relative equality is the value of one object compared to the value of another object—not for purposes of equality, but for purposes of determining whether a value is less than or greater than.

Let's start with the less than operator, `<`. If the value on the left, or `lvalue`, is less than the value on the right, or `rvalue`, then the statement evaluates to true. The less than or equal operator, `<=`, adds the condition that if the values are equal, then the statement also evaluates to true. The greater than operator, `>`, evaluates to true if `lvalue` is greater than `rvalue`. The greater than or equal operator, `>=`, takes the `>` operator one step further and evaluates true if `rvalue` and `lvalue` are equal.

For numerical variables, the concept of relative equality is very well defined, down to the point of being implemented within the microprocessor architecture. The same rules for conversion of types apply when performing any sort of comparison, so you can refer to the discussion on conversions in Chapter 3 if you need to confirm the process for a specific comparison.

The concept of relative equality becomes very muddled when working with strings. Various character encodings, punctuation, and accented characters can confuse most character-by-character comparisons into returning results that are less than desirable. You don't need to know how algorithms and punctuation are handled or the various locales that exist on a given machine to perform efficient string comparisons. It is important to keep in mind that JScript .NET does a very good job of handling comparisons in strings without any additional work by the programmer to handle punctuation or locales. In reality, the advanced string comparison behavior is provided by the CLR. If you need low-level handling when performing comparisons, you should use the string comparison classes and functions provided by the CLR instead of the normal comparison operators we've discussed here. For more information on handling string comparison by using the CLR, you should investigate the .NET framework documentation in the `System.Text` namespace.

Arithmetic Operators and Concatenation

JScript has many mathematical operators. For simplicity, this section examines them in several different groups. These mathematical operators are extremely important to programming in JScript .NET because they enable counting for looping constructs and the creation of numerical algorithms for solving problems. All programs need to use basic arithmetic, so JScript .NET provides operators for this in addition to many complex mathematical operators:

- **Operators that appear within statements**—This is how the arithmetic operators are most commonly used.
- **In-place operators**—These come in handy when only simple operations need to be performed on an existing variable.

- **The `Math` operator**—JScript doesn't contain operators for many mathematical operations, and you often need to use either the JScript `Math` object or the CLR `System.Math` object.
- **The concatenation operator**—This is probably the most commonly used operator. You use it any time you need to merge the results of several operations, variables, or statements into a presentable string format.

Arithmetic Operators in Statements

Operators most commonly appear in statements. A statement can be as simple as two operands and a single operator or as complex as hundreds of grouping parentheses and multiple operands and operators. For the most part, only simple statements are needed to understand the purpose of each operator. These operators include the basic arithmetic operators (`+`, `-`, `*`, and `/`), the increment and decrement operators (`++` and `--`), and the modulus operator (`%`).The basic operators include the addition operator (`+`), the subtraction operator (`-`), the multiplication operator (`*`), and the division operator (`/`). Whenever you use these operators, the final result is made available in the form of the smallest datatype needed to house the result. For instance, if you add two numbers, one of which is an `int`, and the other of which is a `float`, then the result is a `float` because it's the simplest datatype that can hold the result. Assigning the result to a datatype with less precision than is needed results in a loss of any data that can't be converted. So casting the `float` result to an `int` would result in the loss of any fractional portion of the number. You can do this, for example, to do integer-only arithmetic, where you're using a quotient and remainder instead of using the fractional result. You can see some of these operators in action in Listing 4.1.

LISTING 4.1 Using Basic Arithmetic Operators

```
var IntegerResult:int = 0;
var FloatResult:float = 0;
var DoubleResult:double = 0;

IntegerResult = 5 + 5; // Result is 10
IntegerResult = int(5 + 5.5); // Result is still 10, notice the cast to int
IntegerResult = 5 - 5; // Result is 0
IntegerResult = 5 * 5; // Result is 25
IntegerResult = 5 / 5; // Result is 1
IntegerResult = int(5 / 1.5);
// Result is 3 or the integer result of the division

// Both float result and double result will hold float identical
// values. However, in some cases the DoubleResult will be to a
```

LISTING 4.1 continued

```
// higher precision.
FloatResult = 5 / 1.5; // Result is 3.333333
DoubleResult = 5 / 1.5;  // Result is 3.33333333333333
```

In addition to the basic arithmetic operators described so far, there are increment (++) and decrement (--) operators. These operators are handy for basic looping operations and modifying a variable before or after using it in a statement.

You can use the increment and decrement operators in two different manners. You can apply them to the beginning of a variable, in which case they are the preincrement (`++var`)/predecrement (`--var`) operators, or you can apply them after the variable, in which case they are the postincrement (`var++`)/postdecrement (`var--`) operators. When such an operator is used in front of a variable, the variable is first incremented or decremented and then used in whatever statement in which it has been placed. When such an operator is used after a variable, the variable is used in the statement in which it is located and then incremented or decremented after use. Listing 4.2 demonstrates this; the code in this listing prints the values of the variables before and after each of these operators is used.

LISTING 4.2 Increment and Decrement Operators

```
import System;

var IncrementInt:int = 5;  // Let's initialize these to 5
var DecrementInt:int = 5;

Console.WriteLine("Incrementing using operators");
Console.WriteLine(IncrementInt);
Console.WriteLine(++IncrementInt);
Console.WriteLine(IncrementInt);
Console.WriteLine(IncrementInt++);
Console.WriteLine(IncrementInt);

Console.WriteLine();
Console.WriteLine("Decrementing using operators");
Console.WriteLine(DecrementInt);
Console.WriteLine(--DecrementInt);
Console.WriteLine(DecrementInt);
Console.WriteLine(DecrementInt--);
Console.WriteLine(DecrementInt);
```

```
Incrementing using operators
5
6
6
6
7

Decrementing using operators
5
4
4
4
3
```

NOTE

The increment and decrement operators are technically in-line operators because they update the value of a variable as a result of using the operator. They have a unique behavior in that they modify the variable in question and use it in the current statement if applicable. Also, the fact that they have two separate locations in which they can be used is unique. These two operators will be very important in later sections of the book.

You use the modulus (`%`) operator in very specific scenarios, and it has a number of important applications. The modulus operator always returns the remainder of an integer division. For example, in the statement `11 / 3`, the integer result is `3`, and there is a fraction left over. The remainder of the operation is `2` and using the modulus operator will return `2` as the result. The simple calculator program shown in Listing 4.3 demonstrates all the arithmetic operators we've discussed so far. This program takes two numeric operands, assigned to the variables `operand1` and `operand2`. An operator in the form of a string is assigned to `operator1`. If you want to change the input, you have to recompile the program. You should feel free to write some parsing code to accept input from the command line.

LISTING 4.3 A Simple Calculator Program

```
import System;

// This is a calculator program.
// You can change the operands by assigning new values
// to operand1 and operand2.  Change the operator by
// assigning the appropriate string to operator1.
```

LISTING 4.3 continued

```
// Feel free to write some parsing logic to change
// this functionality.

// You can change the types here.  Using doubles because
// they are the largest datatype supported by JScript
var operand1:double = 350;
var operand2:double = 27;
var result1:double = 0;

var operator1:String = "-";

switch(operator1) {
    case "+":
        result1 = operand1 + operand2;
        break;
    case "-":
        result1 = operand1 - operand2;
        break;
    case "*":
        result1 = operand1 * operand2;
        break;
    case "/":
        result1 = operand1 / operand2;
        break;
    case "%":
        result1 = operand1 % operand2;
        break;
}

Console.WriteLine("The result of " + operand1 + " " +
    operator1 + " " + operand2 + " is " + result1);
```

There are two ways for you to improve the calculator in Listing 4.3. First, you could write some fairly complex string parsing code to evaluate the string representation of the numeric formula straight from the command line and turn it into a series of operations that can be performed by the calculator logic in Listing 4.3. Second, you could use the JScript `eval` method to run arbitrary JScript code. You would need to pass the command line straight to the `eval` method, and the command line would need to be a valid numeric expression.

Inline Arithmetic Operators

An inline arithmetic operator operates directly on the value within a variable rather than creating an intermediate result and storing that result as another variable. Many times, you'll want to write code in which you work on a variable mathematically and immediately assign the result back to that same variable. With long enough variable names, this can become extremely tedious, so some intuitive operators were added to JScript to ease the amount of code required for these operations. All the arithmetic operators discussed previously in this chapter—except for the increment and decrement operators—have equivalent inline operators. (The increment and decrement operators already work inline on a variable.)

All the inline operators use the left-side variable as both the leftmost operand in the statement and as the variable for result assignment. The addition inline operator (`+=`) can be used as follows to add 10 to a given variable:

```
var MyVariable:int = 25;
MyVariable += 10;  // MyVariable now equals 35
MyVariable = MyVariable + 10;  // This is equivalent to the previous code.
```

Notice that the addition inline assignment operator saves quite a bit of typing. It is also much cleaner for simple operations, which can come in handy when you're writing large programs.

What happens when you need to write an entire expression on the right side of the addition inline assignment operator? JScript resolves the entire expression on the right side of the operator, and then it uses the result against the variable. So to add the result of a complex expression to `MyVariable`, you would use code like the following:

```
var MyVariable:int = 25;
MyVariable += 10 * 3 + 6;  // MyVariable is equal to 49
MyVariable = MyVariable + (10 * 3 + 6);

// This is equivalent to the previous code.
```

The inline subtraction operator (`-=`), inline multiplication operator (`*=`), inline division operator (`/=`), and inline modulus operator (`%=`) all perform their operations by using the right-side expression as the right-side operand and the variable as the left-side operand, assigning the result back to the variable. They all accept the same syntax as the addition operator.

The JScript `Math` Object

The JScript `Math` object provides several commonly used mathematical functions that simply can't be expressed easily as operators. Most of these functions are trigonometry functions and their inverse functions, as well as functions such as `log`, `pow`, and `sqrt`. You can access all these functions directly from the `Math` object.

Each trigonometric function (`sin`, `cos`, `tan`) takes a single numeric parameter, in radian form. (If you want to work with degrees, you have to multiply each angle by pi/180.) The `Math` object has a convenient property, `PI`, that returns a long version of pi. Each inverse trigonometry function (`asin`, `acos`, `atan`) takes a single numeric parameter that represents a valid sine, cosine, or tangent, and returns the angle, in radians. These functions all have common uses in graphics, gaming, and fractals.

Many languages have an operator for computing numeric powers. JScript doesn't have such an operator, and thus you have to use the `Math.pow` method. This method takes two parameters: a base number and an exponent. To reverse the `pow` operation, you use the `sqrt` method. The `sqrt` method takes a number and returns its square root. The `log` function returns the natural log of a given number. JScript doesn't have a method for returning the log of a number in a base other than the natural number `e`, so you have to make one yourself based on the laws of logarithms.

Let's add these new methods to the calculator started in Listing 4.3. Listing 4.4 is code for a calculator that is capable of processing trigonometric functions as well as the basic arithmetic operators used in Listing 4.3. Notice that some command-line processing has been added. This should come in handy for playing with this slightly more enhanced version of the simple calculator.

LISTING 4.4 An Advanced Calculator

```
import System;
var CommandLine:String[] = Environment.GetCommandLineArgs();
if ( CommandLine.Length < 3 ) {
    Console.WriteLine("Usage: 4-4.exe {operator1} {operand1} [{operand2}]");
}

var operator1:String = CommandLine[1];
var operand1:double = 0;
var operand2:double = 0;

// Only 1 argument is needed for some functions
operand1 = int(CommandLine[2]);
if ( operator1 == "+" || operator1 == "-" || operator1 == "*" ||
     operator1 == "/" || operator1 == "%" ){
    operand2 = int(CommandLine[3]);
}

var result1:double = 0;


switch(operator1) {
    case "+":
        result1 = operand1 + operand2;
        break;
    case "-":
```

LISTING 4.4 continued

```
        result1 = operand1 - operand2;
        break;
    case "*":
        result1 = operand1 * operand2;
        break;
    case "/":
        result1 = operand1 / operand2;
        break;
    case "%":
        result1 = operand1 % operand2;
        break;
    case "sqrt":
        result1 = Math.sqrt(operand1);
        break;
    case "log":
        result1 = Math.log(operand1);
        break;
    case "sin":
        result1 = Math.sin((Math.PI/180) * operand1); // Accept degrees
        break;
    case "cos":
        result1 = Math.cos((Math.PI/180) * operand1); // Accept degrees
        break;
    case "tan":
        result1 = Math.tan((Math.PI/180) * operand1); // Accept degrees
        break;
    case "asin":
        result1 = (180/Math.PI) * Math.asin(operand1); // Output degrees
        break;
    case "acos":
        result1 = (180/Math.PI) * Math.acos(operand1); // Output degrees
        break;
    case "atan":
        result1 = (180/Math.PI) * Math.atan(operand1); // Output degrees
        break;
}

// We have to print in two different formats
if ( operator1 == "+" || operator1 == "-" || operator1 == "*" ||
     operator1 == "/" || operator1 == "%" ){
    Console.WriteLine("The result of " + operand1 + " " +
        operator1 + " " + operand2 + " is " + result1);
```

LISTING 4.4 continued

```
} else {
    Console.WriteLine("The result of " + operator1 + " " +
        operand1 + " is " + result1);
}
```

NOTE

The advanced calculator program is quite big. If you already came up with some better parsing code for the simple calculator, you can probably apply that here. Maybe you even looked up the `eval` method and found it to be a good tool.

To get some practice, you can try to add the `pow` method to the calculator. Many languages support the `**` operator as the exponential operator. See if you can rewrite the parsing code to accept this operator and other forms of some of the other operators that you might have seen.

The Concatenation Operator

JScript uses the same operator (+) for the concatenation operator and the addition operator, so you have to be careful when using it, to make sure you're doing string concatenation rather than the addition of two nonstring variables. You need to know the rules that govern when concatenation occurs and when numerical addition occurs. Whenever the left or right side operand is a string variable, JScript treats the + operator as a concatenation operator. However, when you string these operators together, you have to be careful because trying to concatenate several numbers at the beginning of the string might return unexpected results. Because concatenation and addition have the same precedence, the numbers at the beginning of the expression are added together rather than being concatenated together as a string. Take the following code sample as an example of how trying to concatenate numbers can return unexpected results:

```
import System;

Console.WriteLine(10 + 10 + 10 + "Hey");
Console.WriteLine("Hey" + 10 + 10 + 10 + "Hey");
```

```
30Hey
Hey101010Hey
```

In the first set of concatenations, the `+` operator is treated as an addition operator; it adds the three `tens` to make 30 before appending the string `"Hey"`. In the second example, each of the three tens is appended to the string in sequence. To change the first string,

```
Console.WriteLine(10 + 10 + 10 + "Hey");
```

so that it appends three tens to the string, you could prepend a blank string to the beginning:

```
import System;

Console.WriteLine("" + 10 + 10 + 10 + "Hey");

101010Hey
```

In this example:

```
Console.WriteLine("Hey" + 10 + 10 + 10 + "Hey");
```

if you really wanted to perform addition instead of concatenation, you would group the three tens together to give them a higher priority (in other words, you want the two additions between the tens to happen before the concatenation between the first `"Hey"` string). The following code would do this:

```
import System;

Console.WriteLine("Hey" + (10 + 10 + 10) + "Hey");

Hey30Hey
```

TIP

When you're unsure about whether a + operator will be treated as a concatenation operator or an addition operator, be sure to raise the priority of the operators by grouping like operators with parenthesis. It's better to be explicit and get the wanted behavior than to introduce strange concatenation bugs by being less explicit than necessary.

The string concatenation operator can convert any object or numeric constant into a string. Sometimes the string doesn't accurately represent the object, but you can rest assured that you won't get any type conversion errors from the JScript compiler. The concatenation operator can even appropriately convert objects that are null to displayable strings.

Bitwise and Boolean Operators

The bitwise and Boolean operators are generally considered to be more advanced than the operators discussed to this point because they work on pieces or parts of a variable (bits) rather than on the entire variable. Like the arithmetic operators, bitwise and Boolean operators come in two flavors: statements and inline assignment operators.

Bitwise Operators

There are bitwise operators for every possible bitwise operation except for rotation. A bitwise operator works on the individual bits of a value rather than the entire value. Bitwise operators are capable of combining the bits of values in different ways (`&` [and], `|` [or], and `^` [xor]) and shifting or transposing bits (>> [shift right] and << [shift left]) in a value.

The shift operator shifts the bits of a number either left or right by a given number of positions. The left shift operator (<<) shifts bits to the left a given number of spaces, increasing the value of the numeric expression. To shift bits to the right, you use a right shift operator (>> or >>>). The >> operator preserves that the sign, either negative or positive, of the value, and the >>> operator is sign independent, meaning that it works directly on the bits and doesn't take into account that the number might be negative or positive. (Having two versions of this operator is unique to the JScript language.) There are inline assignment versions of each of these operators (that is, <<=, >>=, >>>=).

Oftentimes, shifting operations can do the same work as more expensive operations. It used to be extremely costly to multiply and divide on most consumer machines, so mechanisms called *shift multiplication* and *shift division* were developed. If you shift a given value left one space, it has the same effect as multiplying by 2, and shifting right by 1 has the effect of dividing by 2. Shift multiplication and shift division are especially handy when computing pixel offset in games.

Listing 4.5 shows the shift operators in action. At the end of the bitwise operators, let's see if we can come up with an encryption/decryption algorithm and a hash algorithm. As you can see in Listing 4.5, you can shift 1 left by 10 spaces to get a kilobyte, by 20 for a megabyte, and by 30 for a gigabyte.

LISTING 4.5 Using Shift Operators

```
import System;

Console.WriteLine(1 << 10); // 1 Kilobyte
Console.WriteLine(1 << 20); // 1 Megabyte
Console.WriteLine(1 << 30); // 1 Gigabyte
```

LISTING 4.5 continued

```
Console.WriteLine(1024 >> 10);
Console.WriteLine(-1024 >> 10);
Console.WriteLine(-1024 >>> 10);
```

```
1024
1048576
1073741824
1
-1
4194303
```

The remaining bitwise operators correspond with the `AND`, `OR`, `XOR`, and `NOT` operations. When you use `AND` on two variables or numeric values, each bit that is set (that is, a 1 bit) in both values is set (that is, a 1 bit) in the result, whereas any other combination is unset (that is, a 0 bit).

The most common use of the JScript `AND` operator (`&`) and its inline assignment equivalent (`&=`) is to test the existence of a single bit or set of specific bits in a number. For instance, because `1110 (14) & 1100 (12) = 1100 (12)`, you can use the value `1100 (12)` to mask out bits 3 and 4 of a number so that you can test those bits and only those bits. As shown in Listing 4.6, you can use the `Convert` object to print out the binary values.

LISTING 4.6 Binary Bit Masking

```
import System;

var FullValue:int = 255;
var MaskHigh:int = 240;
var MaskLow:int = 15;
var MaskStripe:int = 170;

var Result:int = 0;

// Lets mask some values out to Result
Result = FullValue & MaskHigh;
Console.WriteLine(Convert.ToString(Result, 2));

Result = FullValue & MaskLow;
Console.WriteLine(Convert.ToString(Result, 2));

Result = FullValue & MaskStripe;
Console.WriteLine(Convert.ToString(Result, 2));
```

Listing 4.6 continued

```
// Lets use the inline operators just for fun.
// We are going to mask out the high values and
// then do a stripe mask.
FullValue &= MaskHigh;
FullValue &= MaskStripe;
Console.WriteLine(Convert.ToString(FullValue, 2));
```

To pull bits out of a numerical value, you use the `AND` (`&`) operator. To put bits together or create flag values, you use `OR` (`|`) or its inline assignment equivalent (`|=`) on the values. If the bit is set in either operand, then the bit in the result is also set; the only time a bit is not set is when the bit is `0` in both the operands. This makes it really easy to merge values together and create flags. You can generally generate a flag value by adding bit values together, but then if you add the same flag twice, you get incorrect results. You can prevent this by using `OR` (`|`) on the flags.

Almost identical to the `OR` operator (`|`) is the `XOR` operator (`^`) and its inline assignment equivalent, `^=`. The only difference between `|` and `^` or `^=` is that with `^` or `^=`, when a bit is set in both operands, its value is unset in the result. The exclusivity comes from the fact that one of the operands has the bit set; if both operands had the bit set, the result would be false or an unset bit. If you use the `XOR` operator on one value against the same value twice, you get the original value back. We'll use this operator to generate an encryption algorithm later in this section.

Unlike the rest of the binary operators we've discussed so far, the `NOT` operator (`~`) doesn't work between two operands. It simply returns the value, with all its bits swapped. Using `AND` (`&`) on a value with its negated value always produces a bitmask with all `1`s. Using `OR` (`|`) or `XOR` (`^`) on a value with its negated value always returns all `0`s. The `NOT` operator isn't always this useful, and it doesn't have an inline assignment format, but Listing 4.7 shows an example of putting it to good use in encryption/decryption code.

Listing 4.7 Encryption with Bitwise Operators

```
import System;

var OriginalValue:int = 222; // A relatively mixed-up set of bits
var NewValue:int = 0;
var XorSalt:int = 11; // We only want 4 bits of Salt, so use values <15
var MaskHigh:int = 240;
var MaskLow:int = 15;
```

LISTING 4.7 continued

```
var LowValue:int = 0;
var HighValue:int = 0;

// The algorithm is simple
//   1.  Work on the top and bottom halves separately
//   2.  Use an XOR NOT mask on the top half
//   3.  Use a NOT XOR mask on the bottom half

Console.WriteLine("\nShowing values after we mask them out");
LowValue = OriginalValue & MaskLow;   // Get bottom 4 bits
Console.WriteLine(Convert.ToString(LowValue, 2));

HighValue = OriginalValue & MaskHigh; // Get top 4 bits
HighValue >>= 4; // Lets move the top 4 bits down to the bottom 4
Console.WriteLine(Convert.ToString(HighValue, 2));

Console.WriteLine("\nShowing values after perform the *encryption*");
LowValue ^= XorSalt;
LowValue = ~LowValue; // After we NOT we have to mask off our low 4 bits
LowValue &= MaskLow;
Console.WriteLine(Convert.ToString(LowValue, 2));

HighValue = ~HighValue;
HighValue &= MaskLow; // After we NOT we have to mask off our low 4 bits
HighValue ^= XorSalt;
Console.WriteLine(Convert.ToString(HighValue, 2));

Console.WriteLine("\nShowing original value and the new value");
NewValue = LowValue + (HighValue << 4); // Put the value back together
Console.WriteLine(Convert.ToString(OriginalValue, 2));
Console.WriteLine(Convert.ToString(NewValue, 2));
```

```
Showing values after we mask them out
1110
1101

Showing values after perform the *encryption*
1010
1001

Showing original value and the new value
11011110
10011010
```

TIP

Try to write the decryption code for the example shown in Listing 4.7. The trick is to perform all the steps in reverse and make sure after each step that you have clean values. Notice that after you perform the `NOT` operation, you have to remask the bits because the size of the `int` datatype is 16 to 32 bits, and you were working with only 4 of them. Therefore, negating a value means that the top 12 to 28 bits are being set (that is, to the bit value `1`). That throws the whole equation off, so it is important to mask off the low-order bits that you want.

Boolean Operators

The Boolean operators are simple yet extremely important when used with the comparison operators discussed earlier in this chapter. Whenever you need to use the results of more than one comparison in a complex statement, you need to combine all the comparison statements with one or more of the Boolean operators.

For example, you can use Boolean operators to check user input. As the user is inputting commands at the console, you might have to check those commands and map them to functions within an application. The `q`, `quit`, and a number of other commands might all map to the shutdown procedures for the application. To check all those values together, you have to use one of the Boolean operators. The Boolean `OR` operator (`||`) returns true if any of the statements are true, and the Boolean `AND` operator (`&&`) returns true if all the statements are true. The Boolean negation operator (`!`) negates the result of a Boolean expression. All this can seem rather confusing, so Listing 4.8 shows an example of how to use Boolean operators.

TIP

You should remember some of these operators from the calculator application shown earlier in the chapter, in Listing 4.3 and Listing 4.4. The calculator checks the command-line input for the operator so that you can determine which actions to take and how many additional numeric arguments to parse on the command line.

LISTING 4.8 Combining Boolean Expressions

```
import System;

// Let's do the basic comparisons first
Console.WriteLine("True values");
```

LISTING 4.8 continued

```
Console.WriteLine(true && true);
Console.WriteLine(true || true);

Console.WriteLine("True/False values");
Console.WriteLine(true && false);
Console.WriteLine(true || false);

Console.WriteLine("False values");
Console.WriteLine(false && false);
Console.WriteLine(false || false);

Console.WriteLine("Negation Operator");
Console.WriteLine(!true);
Console.WriteLine(!false);

Console.WriteLine("Complex Combinations");
Console.WriteLine((true && true) || (false || false));
Console.WriteLine((!true) && (false && false));
```

```
True values
True
True
True/False values
False
True
False values
False
False
Negation Operator
False
True
Complex Combinations
True
False
```

The code shown in Listing 4.8 isn't extremely complex, nor does it represent a special set of Boolean comparisons. All the comparisons are rather arbitrary and are just meant to show you the scope of different expressions that can be handled. In later chapters, you'll be using actual comparison expressions in place of the static Boolean true and false values.

Object Operators

You need to understand the object operators to gain a thorough understanding of how to make full use of the JScript .NET language. Using these operators, you can easily create new instances of objects and query existing objects for their types or compare them against known types.

NOTE

To fully understand a couple of the object operators mentioned in this section, you need to refer to Chapters 6, "Creating Classes," and 7, "Interfaces and Class Members," for more information.

The `new` operator creates an instance of any object or type that you can create in the current scope. In object-oriented programming, a common programming practice is to encapsulate data items inside special objects by using special methods or properties that give your program indirect access to the values within. For every new piece of data, you would have to create a `new` object. The following example defines a structure that holds a single integer value, but every time you want a new piece of data, you have to create a new instance of this structure:

```
import System;

public class DataPoint {
    public var Point:int;
}

var MyDataPoint:DataPoint = new DataPoint();
MyDataPoint.Point = 5;
Console.WriteLine(MyDataPoint.Point);
MyDataPoint = new DataPoint();
Console.WriteLine(MyDataPoint.Point);
```

```
5
0
```

You create the first point of data and assign it the value `5`, and you use `Console.WriteLine` to verify that the value is indeed `5`. Then, you create another new data point. This time, you do not set a value for the data point, so the default value—which for an `int` is always `0`—appears. Note that you never reassign a new value to `MyDataPoint.Point`. However, you do assign a different object to the variable, thus changing the value.

NOTE

The `new` operator is extremely important in object-oriented programming. Programs tend to use objects to represent individual pieces of data in a program, and most programs contain hundreds, if not thousands, of these small objects. Each object must be created by using the `new` operator and then assigned values to make it distinct from other objects of the same type. The dependency of code on the `new` operator is covered in more detail in Chapter 6 and Chapter 7.

Often in object-oriented programming, code has to operate on derived objects (that is, objects that are based on other classes with which the program knows how to work). A new derived class often implements new functionality or overrides some of the methods to work with different sets of internal data. Although this is extremely common, you shouldn't have to rewrite your original code every time there is a new derived object; it should just work. You can test whether an object implements a certain interface, is derived from a certain class, or is a certain class by using the `instanceof` operator. The `instanceof` operator can be better examined with the basic knowledge that all objects in the CLR are derived from (that is, related to) the `System.Object` type. All basic datatypes are derived from (that is, related to) the CLR `System.ValueType` type. Listing 4.9 shows the use of the `instanceof` operator to determine if certain classes are related.

LISTING 4.9 Using the `instanceof` Operator

```
import System;

public class DataClass {
}
public class AntiDataClass {
}

var DataObj:DataClass = new DataClass();
var AntiObj:AntiDataClass = new AntiDataClass();

Console.WriteLine(DataObj instanceof System.Object);
Console.WriteLine(DataObj instanceof DataClass);
Console.WriteLine(DataObj instanceof AntiDataClass);

Console.WriteLine();
Console.WriteLine(AntiObj instanceof System.Object);
Console.WriteLine(AntiObj instanceof DataClass);
Console.WriteLine(AntiObj instanceof AntiDataClass);
```

```
True
True
False

True
False
True
```

NOTE

Play with this code a bit and examine why the statements evaluate to true or false the way they do. When you get to Chapters 6 and 7, you can try rewriting `DataClass` and `AntiDataClass` to further derive types, and then you can implement some interfaces and examine how the `instanceof` operator acts in those instances. If `AntiDataClass` were derived from `DataClass`, how would the program be different?

Using Comments

You're probably wondering why comments can be considered so important to code—and even more important than the code itself in many cases. The answer is simple: After months of coding, you might not remember the code you wrote at the beginning of the project. This often happens when a programmer spends several weeks optimizing a piece of code and then later goes back to work on that same code and can't remember why the code was written a certain way. If the programmer uses comments in the code and explains why he or she did certain things, that programmer—or another programmer—wouldn't have any trouble working on the code again, even after months or years.

There are many other reasons to comment code. Comments are a first-line defense against coworkers asking you questions about your code. And if a programmer leaves a company, comments minimize the disruption because a new programmer can get up to speed simply by reading the code comments rather than asking people questions. Furthermore, at the end of the product cycle, you can use comments as a base for documentation.

Comments can go anywhere in JScript .NET. Comments, like whitespace, are ignored by the compiler, leaving you free to place them on class definitions, on constructors, before functions, inside functions, and even inline with the parameters of a function call. You should put comments in all these places because they'll all be helpful down the road.

JScript supports a couple different commenting syntaxes: the inline/block comment syntax and the single-line/end-of-line syntax.

Inline/Block Comments

You can place inline/block comments anywhere in code, and you can use them to quickly comment out large blocks of code or documentation. The inline/block comment is the most commonly used type of comment. The comment starts with a begin comment tag (`/*`) and ends with an end of comment tag (`*/`). There are some places where these tags are not recognized. For example, you can't embed them in string literals and expect them to act as comments. Listing 4.10 shows an example of the use of inline/block comments.

LISTING 4.10 Using Inline/Block Comments

```
/*
    This is a file header comment.
    Generally author information, last modified date, and a description
    of what the file does is placed here.
*/

/* This comment describes a piece of code to follow */
var ImSomeFollowingCode:int = 0; /* I'm a piece of code at the end-of-line */

/*
var ImCommentedOut:int = 0;
*/

function SomeFunction(param1:int /*I'm describing the parameter*/) {
}
```

All the comments shown in Listing 4.10 are valid. One is a block at the beginning of the file, some are single-line comments above code, some are single-line comments after code, and some are placed on the same line as code, right after a parameter to a function, to describe the usage of the parameter. Single-line comments, which are discussed in the next section, can be used in many of the same ways as the block comments shown in Listing 4.10. However, they should not be used to comment large blocks of code or documentation, and they cannot be used to place comments between code elements.

Single-line and End-of-line Comments

Single-line and end-of-line comments can be used to comment out code with a single begin-of-comment marker (`//`). The end-of-comment marker is always the newline (that is, a hard return), so executing code cannot be placed anywhere between the comment marker and the end of the line, or it will be commented out. End-of-line comments are generally used to comment single lines of code so that they aren't compiled, to apply single-line descriptions to code, or to attach short descriptions to the end of a line of code. Listing 4.11 shows how single-line and end-of-line comments are used.

LISTING 4.11 Using Single-line/End-of-line Comments

```
//
//    You can still use them for file headers
//    And many people do
//

// The best place is above a line of code
var ImSomeFollowingCode:int = 0; // Or after a line of code

// var YouCanStillCommentCodeOutToo:int = 0;
```

TIP

You can use single-line and end-of-line comments to describe function parameters if you're crafty and make use of multiline function declarations. See if you can write a function declaration with two parameters, using single-line comments to describe those parameters. If you do this correctly, it should take at least three lines, and most likely it will take four lines if you follow common conventions.

Summary

So far in this book you have learned about all the datatypes, how to compile code, and how to use the numerous operators available in JScript .NET. The operators described in this chapter—comparison, assignment, arithmetic, bitwise, Boolean, and object operators—help you solve different programming problems. You can't write a program without using most of them, but you'll find that you use some more than others. In this chapter you have also learned the importance of using comments in code.

In the next chapter we'll be discussing the programming structures used to control program flow. These structures include conditional branching statements, looping statements, and collection enumeration.

CHAPTER 5

Controlling Program Flow

IN THIS CHAPTER

Program flow governs which portions of your code are executed and in what order. It is essential that there be some way to control which code gets to run based on user input, file input, random variables, timings, and system conditions. It might also be important to iterate a single piece of code a certain number of times, iterate over collections of objects, or even break out of the loop so the program can continue doing something else.

We'll begin this chapter by examining the conditional statements, which allow you to check certain conditions, variables, or other information and branch to a certain piece of code. JScript has several forms of conditional statements—`if...else`, `if...else if...else`, `switch...case`, and the ternary operator—each of which is made for a specific purpose.

Conditional statements are the basis for the looping constructs. Each looping construct uses at least one conditional statement to determine when a loop should terminate. You'll learn about basic incremental loops that use the `for` statement, and then we'll discuss the `while` and `do...while` statements, which are used for conditional looping.

Programmers often need to cycle through a collection of objects and do some work with the properties and methods. Using an `Enumerator` object to iterate over code can simplify the look and feel of a program and reduce the number of lines of code normally required to iterate over a collection. In addition, JScript .NET provides the `for...in` statement, which allows for a typed enumeration of objects in a collection within a single statement. In addition, the Microsoft .NET framework provides a large number of enumeration services and interfaces, which we'll examine briefly in this chapter.

We'll end this chapter with a brief section on the `break` and `continue` statements and where they can be used to short circuit the switch statement and the looping statements.. You'll also see the use of labeled statements, which can sometimes simplify a programming task but should almost never be used in object-oriented code.

NOTE

Various features of program flow are not discussed in this chapter because they are introduced later in the book. Any type of function or property accessor is a program flow element, as is the `return` statement, which is used for functions and property accessors. You'll find these constructs later in Chapters 6, "Creating Classes" and 7, "Interfaces and Class Members."

Conditional Statements

Conditional statements are crucial to program flow because they are the programmatic method of deciding which pieces of code should be run. This section discusses all the conditional statements available in JScript .NET:

- `if`
- `if...else`
- `if...else if...else`
- `switch`
- The ternary operator

Often, you can use any of the available statements, and this section describes when and why it's best to use each statement.

Branching with `if`, `if...else`, and `if...else if...else`

The `if` statement has a lot of forms, and each form can be extremely useful in different situations. You use the standard `if` statement whenever a block of code needs to be executed based on some condition. For instance, you might want to ensure that a variable is not equal to `null` before you start calling its properties and methods:

```
if ( myVar != null ) {
    myVar.Property1 = "I wasn't null";
}
```

Because we only performed a single operation within the code block, a shortcut for the `if` statement is to remove the braces altogether and simply follow the statement with the code to be executed. The following piece of code demonstrates the shortcut method as well as some of the common errors:

```
// These are legal
if ( myVar != null ) myVar.Property1 = "I wasn't null";
if ( myVar != null )
    myVar.Property1 = "I wasn't null";

// These are illegal or produce unexpected results
if ( myVar != null )
    Debug.WriteLine("MyVar wasn't null");
    myVar.Property1 = "I wasn't null";

if ( myVar != null )
    if ( myVar.Property1 != null )
        Debug.WriteLine("MyVar wasn't null");
        myVar.Property1 = "I wasn't null";
```

TIP

I don't recommend using this style of the `if` statement because often times you might go back later and add additional statements or some debug code and you can forget to place the braces in. It can also become confusing when you nest `if` statements.

Let's take a quick look at some of the illegal code in this example. In the first illegal snippet:

```
if ( myVar != null )
    Debug.WriteLine("MyVar wasn't null");
    myVar.Property1 = "I wasn't null";
```

notice that the compile doesn't throw any errors. This is because it uses only the simplest form of the `if` statement and does not have an `else` statement with it. This snippet gives some unexpected results. The debug statement executes whenever `myVar != null`, but the property access statement always executes because it is outside the `if` statements block. To fix this, you have to use braces so that the compiler knows that both statements should be in the `if` statements block:

```
if ( myVar != null ) {
    Debug.WriteLine("MyVar wasn't null");
    myVar.Property1 = "I wasn't null";
}
```

The error from the second snippet:

```
if ( myVar != null )
    if ( myVar.Property1 != null )
        Debug.WriteLine("MyVar wasn't null");
        myVar.Property1 = "I wasn't null";
```

is even more confusing than the first, but it has nearly identical results to those of the first error. The first `if` statements block executes the second `if` statement. The second `if` statement block contains the debug statement, and the property access statement appears outside both blocks and will always be executed, regardless of the two `if` statements.

The following code clarifies what each piece of code is doing by drawing out the blocks the compiler sees. Note that the code uses braces so that it produces the expected behavior:

```
// These are what the compiler thought we meant
if ( myVar != null ) {
    Debug.WriteLine("MyVar wasn't null");
}
myVar.Property1 = "I wasn't null"; // Throws runtime error if myVar is null
```

```
if ( myVar != null ) {
    if ( myVar.Property1 != null ) {
        Debug.WriteLine("MyVar wasn't null");
    }
}
myVar.Property1 = "I wasn't null"; // Throws a runtime error if myVar is null

// Here is what we meant to write
if ( myVar != null ) {
    Debug.WriteLine("MyVar wasn't null");
    myVar.Property1 = "I wasn't null";
    // Throws runtime error if myVar is null
}

if ( myVar != null ) {
    if ( myVar.Property1 != null ) {
        Debug.WriteLine("MyVar wasn't null");
        myVar.Property1 = "I wasn't null";
        // Throws a runtime error if myVar is null
    }
}
```

As you can see from this example, you should try and be as explicit as possible with block statements. Remember that the less guesswork for the compiler, the better. The few extra characters you need to include allows the compiler to behave as you expect it to.

What if you had another piece of code that you wanted to execute whenever `myVar` is null? What would be the easiest way to set this up? You could use multiple `if` statements, with the first one checking whether `myVar` is null and doing some action and the second checking whether it is not null and then doing some other action. But what if the statement is so complex that it is hard to create an inverse statement to use? In this case, you use the `else` statement in conjunction with the `if` statement, as in the following example:

```
if ( myVar != null ) {
    myVar.Property1 = "Wasn't null";
} else {
    Debug.WriteLine("Need to fix the program, myVar is null");
}
```

All you have to do is add the new statement, with its own block. Whenever `myVar` is null, you can print some debug code which tells you that the program is broken. Note that again, you can take the braces out on the statement, and everything will compile correctly and run because if you're only working with one statement, you don't need to specify a block. However, some other statements would then cause some problems. Let's take a look at some of them:

```
// Legal
if ( myVar != null )
    myVar.Property1 = "Wasn't null";
else
    Debug.WriteLine("Need to fix the program, myVar is null");

// Compiler error
if ( myVar != null )
    Debug.WriteLine("Yay!");
    myVar.Property1 = "Wasn't null";
else
    Debug.WriteLine("Need to fix the program, myVar is null");

// Fix the compiler error
if ( myVar != null ) {
    Debug.WriteLine("Yay!");
    myVar.Property1 = "Wasn't null";
} else
    Debug.WriteLine("Need to fix the program, myVar is null");
```

```
You can see in this example that you can remove the braces, and everything
works. But if you add lines, you need to add braces back in. This example
results in a compiler error because the compiler wasn't expecting an else
statement:// Compiler error
if ( myVar != null )
    Debug.WriteLine("Yay!");
    myVar.Property1 = "Wasn't null";
else
    Debug.WriteLine("Need to fix the program, myVar is null");
```

The compiler sees the debug statement after the `if` statement, and it notices that the code does not define a block, so it closes the `if` statement. The compiler then sees the `else` statement and throws an error because there is no corresponding `if` statement. Simply using braces fixes the problem, as shown in the final example:

```
// Fix the compiler error
if ( myVar != null ) {
    Debug.WriteLine("Yay!");
    myVar.Property1 = "Wasn't null";
} else
    Debug.WriteLine("Need to fix the program, myVar is null");
```

Notice also that you can leave off the braces on the `else` statement. You can mix and match blocks this way if you like.

You can also use the `if` statement to specify multiple conditional blocks. This is often useful when you're working with numeric datatypes or strings. You can specify as many `else if` statements as you like, but they must follow each other and obey the block order we've talked about before (use braces!). Optionally, you can specify an `else` clause at the very end. Here's an example:

```
var myInput:String = "Option1";
if ( myInput == "Option3" ) {
    // Do Option 3 code
} else if ( myInput == "Option2" ) {
    // Do Option 2 code
} else if ( myInput == "Option1" ) {
    // Do Option 1 code
}

myInput:String = "UnknownOption1";
if ( myInput == "Option3" ) {
    // Do Option 3 code
} else if ( myInput == "Option2" ) {
    // Do Option 2 code
} else if ( myInput == "Option1" ) {
    // Do Option 1 code
} else {
    Debug.WriteLine("Unknown Option");
}
```

Again, the additions to the `if` statement are easy to add to your programming toolbox. The preceding example simply declares a string variable and then tests it multiple times to see if it matches any of the conditions. If it does, then you perform some code. In the second `if` construct, notice that the option doesn't exist, so you specify an `else` statement to catch these unknown values and perform some debug code so that you know why nothing happened.

The code in the previous example is extremely efficient if you have complex comparisons to perform in each of the `if` and `else` if statements. But the previous code example does not exhibit any complex comparisons and only compares a single variable against a single known value. You might be asking yourself, "Isn't there an easier way?" There is the `switch` statement, which we discuss next.

Variable Comparison with `switch`

The `switch` operator makes simple comparisons of a single variable to multiple known values extremely pleasing to the eye. The `if...else if...else` statements can get fairly unseemly, and there isn't really a way to do fall-through by using `if` statements either. What is fall-through? Let's examine the `switch` statement and find out.

In its simplest form, the `switch` statement has only a couple `case` statements. A `switch` statement begins by evaluating an expression and getting back some arbitrary value, normally some numeric result or a string. Then the value is compared against each of the case statements to see if a match is made. If a match is made, the code inside the case statement is executed until a break statement is found or the end of the switch statement is executed. The following example shows an if statement from the previous section rewritten as a switch clause:

```
var myInput:String = "Option1";
switch(myInput) {
    case "Option1":
        // Do Option 1 code
}
```

Although it is not done yet, this is one of the simplest switch statements you can create. If the value in myInput matches the case label of "Option1", code up to the end of the switch block is executed. Let's add two more options blocks:

```
var myInput:String = "Option1";
switch(myInput) {
    case "Option1":
        // Do Option 1 code
    case "Option2":
        // Do Option 2 code
    case "Option3":
        // Do Option 3 code
}
```

Although this example looks right, it's completely wrong. Remember that after a label is evaluated, the code from that case label up to the next break statement, or until the end of the switch block, is executed. Therefore, Option1 executes all the code from Option1, Option2, and Option3, and Option2 executes code from Option2 and Option3. Option3 is the only piece of code that behaves appropriately. This is called fall-through, and it allows you to have multiple case labels that share the same code. Before we finish the example using the Option code, take a look at this example of fall-through code:

```
var myVar:Object = null;
switch(myVar==null) {
    case true:
        myVar = "Hello";
    case false:
        myVar.ToString();
}
```

Never actually use this code! This is one of the worst possible uses of the switch statement, but it's a great example of how fall-through works. The switch statement tests whether myVar == null. Because this is a Boolean expression that returns either true or false, you use true and false case statements. If myVar is null, you can create a new String object and assign it to myVar. You assign the string "Hello" to a variable of type Object, but JScript is handling the cast from String to Object for you so that the String object really gets stuffed in myVar. Then you call ToString on the object, just to make sure myVar isn't still null. If myVar had already been set, the true label would have been skipped, and everything would have proceeded as planned with the false label.

Let's look again at the Option example and add in the break statements. Let's also add a default statement, or a default label, so that whenever you don't make a match, you can run some extra code that lets us know a match was not made:

```
var myInput:String = "Option1";
switch(myInput) {
    case "Option1":
        // Do Option 1 code
        break;
    case "Option2":
        // Do Option 2 code
        break;
    case "Option3":
        // Do Option 3 code
        break;
    default:
        Debug.WriteLine("Unknown Option");
        break;
}
```

This example includes break statements so that the code doesn't fall through the labels and execute code it isn't supposed to. A default label is executed whenever one of the other labels isn't matched.

In some instances, fall-through is positive. Because a case label can be used to represent only one possible value, it often makes sense to stack case statements that fall through so that multiple values execute the same code. The following example is capable of executing a code block for a number of values, but it executes the default statement otherwise:

```
var myInput:int = 10;
switch(myInput) {
    case 1:
    case 2:
    case 3:
    case 5:
    case 10:
        Console.WriteLine("myInput is either 1, 2, 3, 5, or 10");
        break;
    default:
        Debug.WriteLine("Unexpected value");
}
```

As you can see, in some instances (especially when you're comparing a single variable to multiple variables), the `switch` statement is simpler to use than a corresponding `if` structure. The `if` structure is the most complex of the conditional statements and can perform all the operations of the `switch` statement and the ternary operator (described in the next section).

Inline Return Conditions and the Ternary Operator

The *ternary operator* is composed of three parts: the conditional statement, the true return value, and the false return value. The ternary operator allows you to perform a conditional loop and assign one of two values to a given variable. This can shorten code considerably if used correctly, and it can introduce difficult-to-find bugs if it's misused. The ternary operator evaluates to a value, so it's best to use it on the right side of an assignment or as the value for a return statement from a function or property accessor. It consists of a conditional expression, followed by a question mark, followed by the two possible values, separated by a colon. A true value appears on the left of the colon and a false value appears on the right side. The following is an example of the use of a ternary operator:

NOTE

A *property accessor* is a special function that returns the value of a property. By making the method for accessing a property a function, more work can be done to return the value than simply returning the contents of a variable. For more information on properties, see Chapter 7.

```
var myVar:Object = null;
var myString:String = (myVar == null) ? "Null" : "Not Null";
```

If `myVar` is null, which it is, then the variable `myString` contains the string `"Null"`. However, if you did something in the program to change `myVar` to so that it wasn't null, then you might get different values. The ternary operator can get fairly complex. Listing 5.1 shows code that first uses an `if` statement and then uses a ternary operator to do the same thing as the `if` statement. (I don't recommend using the ternary operator in ways that make the program less readable than it should be; Listing 5.1 just shows what you can do with the ternary operator.)

LISTING 5.1 The Ternary Operator Versus the `if` Statement

```
import System;

var Args:String[] = Environment.GetCommandLineArgs();
```

LISTING 5.1 continued

```
if ( Args.Length >= 2 ) {
    var Percentage:int = int(Args[1]);

    if ( Percentage >= 90 ) {
        Console.WriteLine("Grade A");
    } else if ( Percentage >= 80 ) {
        Console.WriteLine("Grade B");
    } else if ( Percentage >= 70 ) {
        Console.WriteLine("Grade C");
    } else if ( Percentage >= 60 ) {
        Console.WriteLine("Grade D");
    } else {
        Console.WriteLine("Grade F");
    }

    var Grade:String = (Percentage >= 90) ?
        "Grade A" : (Percentage >= 80) ?
        "Grade B" : (Percentage >= 70) ?
        "Grade C" : (Percentage >= 60) ?
        "Grade D" : "Grade F";

    Console.WriteLine(Grade);
}
```

Even to a seasoned programmer, the ternary operator in Listing 5.1 might seem a bit confusing. With this program, you convert `Percentage` into a `String` that is displayed on the console. This is done by using `if` and `else if` statements as well as several blocks that display the grade to the console. (The program could assign a variable, but it's easier to simply display the text to the console.) Using the ternary operator, the program checks the initial value and returns `"Grade A"` if true; otherwise, the program proceeds to another ternary operator, and so forth, until the comparison returns either `"Grade D"` or `"Grade F"`. You can use the ternary operator because you are doing an assignment; however, if you were performing blocks of operations, then you would have to use the `if` construct.

TIP

If you haven't been testing some of this code and playing around with it, then go back and do so. You're never going to learn by just reading the book; you have to experiment with the constructs and find new ways and places to use them, quite possibly ways that haven't been discussed in these chapters.

Loop Statements

You use loop statements in code when an action has to be performed multiple times or when an action needs to be performed until a special condition is met (for example, printing out 30 random characters, assigning initial values to every element in an array, looping while the user is inputting data). As with any program flow element, there are always mechanisms to break out of the loop so that the program can continue performing its duties.

This section discusses three types of looping structures:

- The `for` loop is the most powerful of the loop structures and can be used in place of the others.
- The `while` loop is a bit more specialized than the `for` loop. It is used to test a condition at the beginning of each loop.
- The `do...while` construct can be used to test a condition at the end of the loop.

Incremental Looping with `for`

The most common use of the `for` statement is to iterate a piece of code a given number of times. This can be useful for plotting pixels on the screen for a game, operating on each element of an array, or computing a recursive math function such as a factorial. The `for` statement can be broken down into four parts.

The first part of the `for` statement is the initialization statement, which initializes any variables that will be used in the looping process. The parts of this statement are separated by a comma, and the entire statement ends with a semicolon. The initialization statement can't get very complex, but you can use it to declare multiple variables of different data types. The following is an example of an initialization statement:

```
for(var i:int = 0; ...; ...) {
}
for(var i:int = 0, j:String = "Even"; ... ; ...) {
}
```

The second part of the `for` statement is the comparison. While this statement is true, the loop continues to run. This statement can be as complex or as simple as needed. Generally, you iterate while the value is less than some predefined limit, but in the case of enumeration or unbounded array access, you might be looking for other values, such as null or another empty type. The following is an example of the comparison part of a `for` statement:

```
for(var i:int = 0; i < 11; ...) {
}
for(var i:int = 0, j:String = "Even"; i < 11; ...) {
}
```

The third part of the `for` statement is an iteration statement, which is executed after each iteration of the loop. This value drives the `for` loop and is the reason that the value eventually meets whatever condition is required to exit the loop. The following example increments the variable `i` so that it eventually equals 11 and the loop exits:

```
for(var i:int = 0; i < 11; i++) {
}
for(var i:int = 0, j:String = "Even"; i < 11; i++) {
}
```

The fourth part of the `for` statement is the code block, which is executed after each iteration of the loop and is where all the real work is done (even though all the looping work is contained in the `for` statement itself). You can do something simple like print out values, you can modify loop variables, and you can even exit the `for` loop from within the loop by using the `break` statement.

CAUTION

I don't recommend modifying any variables that are used in conditional statements. If you make a mistake in code that modifies the conditional variable to be within bounds, then your loop will never exit. Because the code for this is buried in the code block and not within the `for` statement, you or another programmer might not be able to realize this without wasting valuable time picking through the code block.

Listing 5.2 demonstrates the complex statements that can be used in a `for` statement. Notice that this code sets the variable `j` to either `Even` or `Odd`, based on the value of `i`. There is one problem with this code, though: The very first time through the loop, the iteration statement isn't called, and you end up printing whatever the initialized value of `j` is, which happens to be an empty string. Figure 5.1 demonstrates the code paths that this `for` statement takes.

LISTING 5.2 A Complex `For` Statement

```
import System;

for(var i:int = 0, j:String = ""; i < 10;
    i++, j=(i%2 == 0) ? "Even" : "Odd") {
    Console.WriteLine(i + " " + j);
}
```

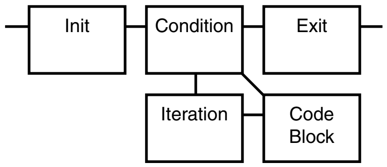


FIGURE 5.1
The Path of the `for` *Statement in Listing 5.2.*

The `for` statement can be used to create any type of loop necessary to control the flow of a program. But sometimes it is more than you need, when a simple conditional loop could do the job.

Conditional Looping with `while`

Simpler alternatives to the `for` loop are the `while` and `do...while` statements. These statements have only two statements, rather than the four statements used by the `for` loop: A `while` loop begins with a conditional statement and ends with a code block that is executed when the conditional statement is true. If you need to initialize loop variables, you need to do so outside the loop. If you need to increment iteration variables, you need to do so inside the loop, somewhere near the end. The following example demonstrates a very simple `while` loop that prints the values 0 through 9:

```
var i:int = 0;
while(i<10) {
    Console.WriteLine(i);
    i++;
}
```

Generally speaking, you use `while` for more abstract loops than for simple incremental loops. For instance, a program might keep running unless a special key is pressed. The program might perform this action by setting a global Boolean variable to true while the program should run, and when it detects this key, it flips the variable to false. When it is time for the conditional statement in the loop to run, the program breaks out of the loop and ends. The following example demonstrates a `while` loop that grabs user input until the user presses the Q key, at which point the `globalBoolean` variable is set to false and the `while` loop terminates:

```
var globalBoolean:Boolean = true;
while(globalBoolean) {
    // Grab some user input
    if ( UserInput == "Q" ) {
        globalBoolean = false;
    }
}
```

Another form of the `while` statement iterates endlessly, unless a break statement allows the program to break free. In this form, the conditional statement always evaluates to true and some complex logic inside the loop decides when the loop should end. The following example demonstrates the use of the `break` statement to short-circuit a `while` loop and exit, even though the conditional statement isn't used to exit the loop:

```
while(true) {
    // Grab some user input
    if ( UserInput == "Q" ) {
        break;
    }
}
```

CAUTION

Using the `break` statement to jump out of a loop construct is extremely bad coding practice because it puts all the logic into the code block. This makes it extremely difficult to find the location from which the loop can be exited in case the logic has to be updated.

The `do...while` statement is used less frequently than the `while` statement. There is only one difference between a `while` and a `do...while` statement: A `while` statement executes the conditional statement before the code block is ever executed, which means a `while` loop can execute zero or more times. A `do...while` statement executes the conditional statement after the code block has been executed, which guarantees that the block will be run at least once before the loop is exited. Figure 5.2 shows the difference between the `while` and `do...while` statements.

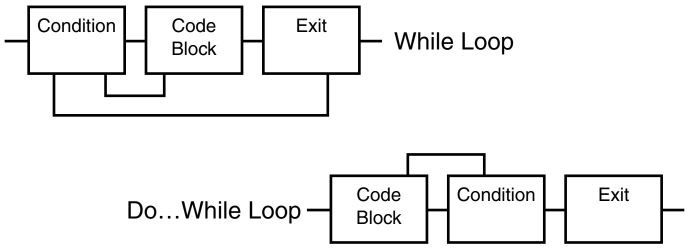


FIGURE 5.2
Differences Between `while` *and* `do...while`.

Enumerations and the `for...in` Loop

In an enumeration loop, the program enumerates through all the elements in a collection or an array. Enumerations are a strange topic in JScript .NET because there are many ways of dealing with them. For instance, you can create a special object to enumerate over all the objects in a collection, you can use the `for...in` statement to do the same thing with a slightly different structure, and you can revert to using the CLR methods and interfaces instead. We'll start by examining the JScript .NET `Enumerator` object.

NOTE

You can generally accomplish enumeration by using existing looping structures, if you know the interfaces.

Creating an `Enumerator` Object

Using the JScript .NET `Enumerator` object is usually the best way to enumerate over a collection of objects. (This is because it's been in the JScript language for quite some time, and many developers are accustomed to using it.) You create an `Enumerator` object by passing the collection to be enumerated in the constructor. Then you can use either a `while` loop or a `for` loop to iterate through the enumeration until it reaches the end. Let's use the `for` statement first to get a good idea of what is going on. Listing 5.3 uses the results of `Environment.GetCommandLineArgs` as input to the `Enumerator` object.

LISTING 5.3 Using an Enumerator on a String Array

```
import System;

for(var e = new Enumerator(Environment.GetCommandLineArgs());
    !e.atEnd(); e.moveNext()) {
    Console.WriteLine(e.item());
}
```

As mentioned earlier, Listing 5.3 creates a new `Enumerator` object by passing in the results of the `GetCommandLineArgs` method, which is a string array. This is the initialization statement of the `for` loop. Next, the listing makes sure the `e.atEnd` method is false because you want to leave the loop as soon as the program reaches the end of the `Enumerator` object. The iteration statement is set up to move the `Enumerator` object to the next item each time through the loop. Without this, the program would never reach the end, and it would keep enumerating the same object over and over. Finally, in the code block you use the `e.item` method to return the object in question. In this case, it is a string, but in other cases it could be another type of object, or the `Enumerator` object could contain many different objects. Listing 5.4 shows how you could work with different objects types in an enumerated collection that returns a generic base type rather than a strong type.

LISTING 5.4 Strongly Typing Objects Out of an Enumerator

```
import System;
import System.Collections;

var arrayList:ArrayList = new ArrayList(); // Make a new collection
arrayList.Add("String");
arrayList.Add(123);
arrayList.Add(DateTime.UtcNow);

for(var e = new Enumerator(arrayList); !e.atEnd(); e.moveNext()) {
    var o:Object = e.item();

    Console.WriteLine();
    Console.WriteLine(o.GetType().ToString());

    if ( o instanceof System.String) {
        Console.WriteLine("String");
    }
    if ( o instanceof System.DateTime) {
        Console.WriteLine("DateTime");
    }
    if ( o instanceof System.Int32) {
        Console.WriteLine("Int");
    }
}
```

```
System.String
String

System.Int32
Int

System.DateTime
DateTime
```

Listing 5.4 shows how you can place several different types of objects in the collection. Then you test the objects on their way out of the `e.item` method so you can do something based on the test results. Inside the `if` constructs, you would probably cast the `o` object to some strongly typed object and begin using its properties. If you know that all the objects within a collection are going to be of the same type, though, it might be easier to use the `for...in` statement than to create an `Enumerator` object.

Using `for...in` for Typed Enumeration

The `Enumerator` object is an extremely powerful object that allows you to enumerate over any collection, regardless of the underlying type. Most of the time, you know what the type of the underlying object is, and you put similar objects in a collection. Instead of creating an enumerator, testing the type of the underlying object, and casting it, you can perform the iteration in a single statement.

There are some complications, however. The JScript .NET `for...in` statement works differently, based on the type of collection you are using. An array always returns the index of the current item as an integer. This means you have to use the index to access the actual value at that array location. If you are using a collection such as `ArrayList`, you use code to strongly type the enumerated value because it is an actual object reference and not an index value. You also have to make sure every item in the collection can be correctly coerced to the type of the enumeration variable, or the function will fail at runtime. This means you are back to the code from the previous section, with multiple types of objects coming out of the collection. Listing 5.5 shows how to work with the `for...in` statement when enumerating arrays, enumerating strongly typed arraylists, and working with collections that return objects of different types.

LISTING 5.5 Working with the `for...in` Statement

```
import System;
import System.Collections;

// Initialize
var i:int;
var o:Object;
var s:String;
var arrayList:ArrayList;
var Args:String[];

// Enumerate an Array
Console.WriteLine();
Args = Environment.GetCommandLineArgs()
for(i in Args) {
    Console.WriteLine(Args[i]);
}

// Enumerate a string arraylist
Console.WriteLine();
arrayList = new ArrayList(); // Make a new collection
```

LISTING 5.5 continued

```
arrayList.Add("String1");
arrayList.Add("String2");
arrayList.Add("String3");
for(s in arrayList) {
    Console.WriteLine(s);
}

// Enumerate an object arraylist
arrayList = new ArrayList(); // Make a new collection
arrayList.Add("String1");
arrayList.Add(123);
arrayList.Add(DateTime.UtcNow);
for(o in arrayList) {
    Console.WriteLine();
    Console.WriteLine(o.GetType().ToString());

    if ( o instanceof System.String) {
        Console.WriteLine("String");
    }
    if ( o instanceof System.DateTime) {
        Console.WriteLine("DateTime");
    }
    if ( o instanceof System.Int32) {
        Console.WriteLine("Int");
    }
}
```

```
>5-5 Hello World
5-5
Hello
World

String1
String2
String3

System.String
String

System.Int32
Int

System.DateTime
DateTime
```

Although Listing 5.5 is a big program that shows a lot, the `for...in` statement really can be used under one condition: when every item in a collection will be of the same type. You can also use it to iterate over arrays, but oftentimes a simple `for` loop is a much clearer way to do this.

The CLR Enumeration Interfaces

The CLR provides a special interface that either an object or a special collection class can use to specify that it supports the methods needed for a program to enumerate through its members. This would be a great time to actually create a class that implements `IEnumerable` and all the required functionality, but you haven't read Chapters 6 and 7 yet, so you don't know how to create classes. However, if you are interested in how this might be done, take a look at the following sidebar.

Exposing Class Properties via `IEnumerable`

If you haven't looked at Chapters 6 and 7 and you don't already know JScript .NET and object-oriented programming, you might want to turn ahead to those chapters. I urge you to try to understand the code of this sidebar, though, because it is very interesting and will probably help you further understand the purpose of the CLR enumeration interfaces.

First, you need to generate a special class that will support your interfaces. Call this the `PropBrowser` class and create three fields to enumerate through. The `PropBrowser` class implements the `IEnumerable` interface, which establishes that you must override the `GetEnumerator` method and return your own custom `IEnumerator` method.

This is rather easy, but you don't have an `IEnumerator` method, so you need to create a `PropBrowserEnumerator` class that implements `IEnumerator`. `IEnumerator` in turn has its own contract that you must support. This includes the `MoveNext` and `Reset` methods and the `Current` property. If you look at Listing 5.6, you'll notice that it initializes `PropBrowserEnumerator` by passing in the values of the three fields and storing them locally. You could just as well pass in a reference to the `PropBrowser` class so that the enumerator could get at the class directly, but in many cases, passing in the static data eases the creation of the enumerator because you don't care if the underlying object is changed while you're doing the enumeration.

LISTING 5.6 Creating a Class for Enumeration

```
import System;
import System.Collections;

public class PropBrowser implements IEnumerable {
    public var Field1:int;
```

LISTING 5.6 continued

```
    public var Field2:int;
    public var Field3:int;

    public function PropBrowser() {
        this.Field1 = 1;
        this.Field2 = 10;
        this.Field3 = 100;
    }

    public override function GetEnumerator() : IEnumerator {
        return new
PropBrowserEnumerator(this.Field1,this.Field2,this.Field3);
    }

    public class PropBrowserEnumerator implements IEnumerator {
        var _current:int;
        var Field1:int;
        var Field2:int;
        var Field3:int;

        public function PropBrowserEnumerator(field1:int,
                                     field2:int,field3:int) {
            this.Field1 = field1;
            this.Field2 = field2;
            this.Field3 = field3;
            _current = -1;
        }

        public function get Current() : Object {
            if ( _current == 3 || _current == -1 ) {
                throw new InvalidOperationException();
            } else {
                switch(_current) {
                    case 0:
                        return this.Field1;
                    case 1:
                        return this.Field2;
                    case 2:
                        return this.Field3;
                }
            }
        }
```

LISTING 5.6 continued

```
        public override function MoveNext() : Boolean {
            _current++;

            if ( _current > 2 ) {
                return false;
            } else {
                return true;
            }
        }

        public override function Reset() : void {
            _current = -1;
        }
    }
}

var p:PropBrowser = new PropBrowser();

for(var e:IEnumerator = p.GetEnumerator(); e.MoveNext();) {
    Console.WriteLine(e.Current);
}
```

`MoveNext` simply updates the `_current` index, which in turn maps to each of the three properties. The `Reset` function sets the current variable back to `-1`, which means you need to have `MoveNext` called before `Current` becomes valid. `Current` maps each of the properties, using the `_current` index, and returns the appropriate value. The code to call the enumerator and iterate over the properties is the same code used for Listing 5.7, so for a better explanation of that code, see the text following this sidebar.

You need an object that supports `IEnumerable`, so use the `ArrayList` class you've been using throughout this chapter. `IEnumerable` specifies that an object must support the `GetEnumerator` method. So to create an `IEnumerator` object we simply call `GetEnumerator` on the `ArrayList` class:

```
var arrayList:ArrayList = new ArrayList();
var e:IEnumerator = arrayList.GetEnumerator();
```

Listing 5.7 demonstrates several CLR enumeration concepts. When you have an `IEnumerator` object, you can get the `Current` object by using the `Current` property, move to the next item by using the `MoveNext` method, and move the enumerator back to the beginning of the collection by using the `Reset` method. Note that `MoveNext()` must be called at least once after the `IEnumerator` object is created or after a call to `Reset` has been made in order for the `Current` property to have a valid object. The `MoveNext` method returns true if the move is successful and another value is available, and it returns false if another object is not available. The CLR enumeration methods are really starting to look a lot like the JScript .NET `Enumerator` object. Listing 5.7 demonstrates the use of the methods made available by the `IEnumerable` and `IEnumerator` interfaces to enumerate over a collection.

LISTING 5.7 Using the `IEnumerable` and `IEnumerator` Interfaces

```
import System;
import System.Collections;

var arrayList:ArrayList = new ArrayList(); // Make a new collection
arrayList.Add("String1");
arrayList.Add("String2");
arrayList.Add("String3");

for(var e:IEnumerator = arrayList.GetEnumerator(); e.MoveNext();) {
    Console.WriteLine(e.Current);
}
```

Listing 5.7 is almost the same as the JScript `Enumerator` object! You should use the JScript `Enumerator` object if you plan on maintaining your code by using JScript .NET. If you want to make your code more easily portable, you should use the CLR interfaces shown in Listing 5.7 instead.

Using Labels with `continue` and `break`

In addition to the exit statements for loops, JScript provides the `continue` and `break` statements, which allow for a short-circuit method for breaking out of a loop structure from within the looping structure. You can optionally label control structures so that you can apply more control regarding where the `continue` and `break` statements move the instruction pointer.

Using the `continue` Statement

You use the `continue` statement when you want to start the next iteration of a loop without performing the rest of the code in the loop. Most of the time, you do this to skip over null objects in collections or arrays or when you are moving some of the processing out of the loop. Listing 5.8 shows how to do this, using graphics code.

LISTING 5.8 Using `continue` to Optimize a Loop

```
import System;
import System.Drawing;

// We are going to use some special code from System.Drawing
var p:Point[] = new Point[10];
var s:Rectangle = new Rectangle(0,0,400,400);

// Special Random class for making random numbers
var rand:Random = new Random();
var i:int;

// Initialize
for(i = 0; i < p.Length; i++) {
    // Generate points from 1 to 1000 for both x and y
    p[i] = new Point(rand.Next(1,1001), rand.Next(1,1001));
}

// Test
for(i = 0; i < p.Length; i++) {
    if ( !s.Contains(p[i]) ) {
        continue;
    }

    Console.WriteLine("Drawing point " + i +
        " to the screen at " + p[i].ToString());
}
```

```
>5-8
Drawing point 1 to the screen at {X=162,Y=353}
Drawing point 3 to the screen at {X=334,Y=392}
Drawing point 6 to the screen at {X=347,Y=19}
```

For this code, the screen is a fixed size of 400 pixels in width and 400 pixels in height. This means that any sprites (that is, small graphics images that portray game objects such as the user's character, monsters, or obstacles) or game objects outside the viewing area don't have to have their graphics-related information updated each time, but you still need to continue processing the rest of the sprites.

The optimization in Listing 5.8 means you only have to write 3 of the 10 sprites to the screen. This is a major performance gain if there is a lot of logic involved in writing the sprites. An additional performance gain could be added if you knew that the points were sorted from least to greatest: You could use the `break` statement, described in the next section, after the code failed the first time and exit the loop entirely.

Where and When to Use `break`

The `switch` statement is where you are most likely to use the `break` statement, but if you're in a complex loop and you've determined that you don't need to do any more work, you can always break out of the loop and continue with the rest of the program. An example of this would be drawing a line on the screen for a graphics program. Again, the screen is 400×400 pixels, and you're drawing a horizontal line, so only the x value is important. Therefore, you set the start pixel and begin working toward the end pixel. However, you need to test when you've reached the end of the screen so you don't draw a pixel right off the edge. The loop could continue going after the first pixel is determined to be lost, but it would be smarter to exit the loop, by using the `break` statement, as shown in Listing 5.9.

LISTING 5.9 Using `break` to Optimize a Loop

```
import System;
import System.Drawing;

// We are going to use some special code from System.Drawing
var p:Point = new Point(200, 0);
var l:int = 400; // Length of our line
var s:Rectangle = new Rectangle(0,0,400,400);

var i:int;

// Test
for(i = 0; i < l; i++, p.X++) {
    if ( !s.Contains(p) ) {
        Console.WriteLine("Stop Drawing at point " + i +
            " to the screen at " + p.ToString());
        break;
    }
}
```

```
>5-9
Stop Drawing at point 200 to the screen at {X=400,Y=0}
```

Using Labels for More Flow Control

The `break` and `continue` statements give you great control over the loop structures and the ability to greatly increase your programs' performance. But what happens when `continue` and `break` simply can't do the actions you want them to? In many instances nested control structures greatly increase the complexity of a program and make it hard to use the `continue` and `break` statements. This is where labeling the flow statements comes in handy.

In JScript, any statement can be labeled, but you really can't do much with the labels. Most languages have a `goto` or `jump` statement that you can use to bounce from label to label, but JScript doesn't have one. JScript does have the `continue` and `break` statements, both of which obey properly labeled statements.

Normally, both the `break` and the `continue` statements exit from the loop structure that was last entered. Thus, if you have two nested loops, `break` exit from the first loop only; the second loop continues, and it is likely to call right back to the loop in which it is nested. To defeat this, you label the outside statement so that you can break out of the entire set of loops, rather than just the inner loop. Normally, the following code would print 12 values:

```
import System;

Outer:
for(var i = 0; i < 3; i++) {
    Console.WriteLine(i);
    Inner:
    for(var j = 0; j < 3; j++) {
        Console.WriteLine(j);
        break Outer;
    }
}
```

```
0
0
```

Breaking out of the inner loop would still mean printing 6 values. However, because you are exiting the entire loop structure by explicitly breaking out of the outer `for` loop, you print only 2 of the 12 values.

CAUTION

I can't emphasize this point enough: Using labels with the `continue` and `break` statements is a really poor programming practice. If you can use extra variables to control the behavior of the loops rather than use these statements, do it.

Summary

In this chapter you have learned about the conditional statements—including the `if`, `if...else`, `if...else if...else`, and `switch...case` statements, as well as the ternary operator—which allow branching in a program.

You have learned how looping statements (that is, `for`, `while`, and `do...while` loops) allow a block of code to be executed more than once. In addition, you have learned that JScript .NET's enumeration abilities allows you to programmatically loop over the objects in a collection by using the `Enumerator` object, the `for...in` statement, and the `IEnumerable` interfaces of the CLR. You have seen how you can use the `continue` and `break` statements to modify the flow of loops and other flow statements so that they behave differently.

In Chapter 6 you will learn about creating and manipulating classes in JScript .NET, and in Chapter 7 you will learn about interfaces and class members. These two chapters are the meat of the book, and they describe a large number of the changes between the previous versions of JScript and the new JScript .NET feature set. You should examine those chapters carefully because they are crucial to understanding the rest of the book.

CHAPTER

6

Creating Classes

IN THIS CHAPTER

In this chapter you will learn all about class definitions, constructors, modifiers, and attributes. You'll learn about the body of the class in Chapter 7, "Interfaces and Class Members," so if you're looking for information on properties, methods, and fields, skip ahead to Chapter 7.

This chapter begins with an overview of classes. You'll learn what makes up a class, and then you'll learn about class attributes, including the visibility attributes and the new .NET metadata attributes.

This chapter discusses inheritance and subclassing, using both base classes and interfaces. It also covers encapsulation, which you can often use in place of true subclassing to achieve nearly the same results. The chapter ends with a discussion on defining and declaring class constructors, overloading, and calling base constructors.

An Overview of Classes

A class is a logical storage container in JScript .NET and object-oriented programming. Every class maintains a series of private data and a unique set of methods for manipulating that data. Every class needs to be defined in code, using special class declaration syntax. Class declarations can appear only in special places in code:

- You can place a class declaration at the global level, outside any package constructs, in which case it is an assembly-scoped class without a namespace.
- You can declare a class declaration within a package block, which simply gives the class a more fully qualified name. If you declare a class at this level, you need to either fully qualify the namespace and class name when instantiating the class, or you need to import the namespace at the top of the file.
- You can place classes within other classes. Classes at this level have special properties, which we'll talk about later in this section.

A Simple Class Declaration

Class declarations always look more complex than they really are. At the simplest level, you can declare a class by prefixing the `class` keyword to the class name and then beginning the class member block.

```
[Attributes] class ClassName [[extends base class],
➥[implements interfaces, ...]] {
...Code Block...
}
```

Inside the class member block, any members of the class are defined, functions are declared, and constructors are written. As discussed earlier in the chapter, other classes might appear inside this block. In the following code, JScript automatically assumes the `public` visibility attribute, as it does with all classes, interfaces, and their member declarations:

```
class SimpleClass {
}
```

The rest of the attributes are discussed later in this chapter, in the section "Class and Member Attributes." The keywords `extends` and `implements` are used to specify a base class and any interfaces for the class. This is discussed later in this chapter as well, in the section "Inheritance and Subclassing."

What a Class Can Contain

Classes can contain quite a few different constructs, including the following:

- *Fields and properties* are used to access data.
- *Functions* can be called to have the class perform some work.
- *Event handlers* (a fancy name for functions) can be defined and attached to events on other objects to receive event notifications.
- *Constructors* can provide extra information to the class at creation time.
- *Classes* sometimes contain other classes, and this can be confusing.

Classes and constructors are discussed in this chapter, and the rest of the components of a class are discussed in Chapter 7.

Class and Member Attributes

Class attributes are used to affect the behavior, visibility, and metadata of a class. Some of these attributes, which we'll be discussing in this chapter, are defined by the JScript .NET language. Other attributes are defined by the common language runtime (CLR) and persist as metadata instead of actually modifying the definition of the class. The following sections discuss these attributes:

- **Visibility attributes**—Visibility attributes are capable of controlling the types of code that can access the method or class in question. Visibility can make classes or members entirely inaccessible, accessible only to a select number of other classes, or completely visible to all other code.
- **Behavior attributes**—Behavior attributes are capable of affecting how classes or members can be used in code. In JScript .NET, they are also capable of enabling special language features.

- **Existential attributes**—Existential attributes are used to control the class inheritance model.
- **Versioning attributes**—Versioning attributes are used to override the normal inheritance model. These attributes are used to break normal versioning semantics.
- **Custom attributes**—Custom attributes can be defined and used by the programmer or by the CLR. Custom attributes are special in that they don't directly affect the assembly output, but they become part of the assembly metadata.

Visibility Attributes

Visibility attributes—that is, the `public`, `private`, `protected`, and `internal` attributes—control whether a member or class can be seen from another member or class. Visibility can be a limiting factor when methods are marked to be accessible only in the current class or inheriting classes. Visibility can also be used to make classes or members globally accessible or public to all code outside the class.

You apply the `public` attribute to members that are globally visible inside and outside a package, and thus any public class can be constructed by any other class or piece of global code. This is the default for all classes, interfaces, and class and interface members.

You cannot apply the `private` attribute to a class, but only to the class's member definitions. Any class member defined as `private` is visible only within its class. The members cannot be accessed outside that class, by any subclasses, or by any nested classes. This is the strictest of the visibility attributes, and you use it only when you need to hide members and data for use by only one class.

For the most part, you cannot apply the `protected` attribute to classes. There is one exception: The nested class can have the `protected` attribute applied. The `protected` attribute is almost identical to the `private` attribute. However, any subclasses of a class marked as `protected` can access the member as well as the class in which it is defined. After this attribute is used on a nested class, the class can be used only by the class it is defined in and by any classes derived from that class.

You use the `internal` attribute to package scope classes or members. When a class is defined outside a package and the `internal` attribute is specified, the class can be used by any global code in the assembly and also from within any classes that are in other packages in the same assembly. When the `internal` attribute is used on a class within a package, that class is available only within that package. Even other packages within the same assembly, global code, or global classes are unable to access the class when this attribute is applied. When this attribute is applied to members of a class or nested classes, the same rules as the class rules for the internal attribute apply.

Behavior Attributes

Behavior attributes affect how a class or member works. There are only three behavior attributes (`expando`, `compatible`, and `static`), and only two of them (`expando` and `compatible`) apply to classes. These classes are also capable of enabling special language features that can't otherwise be used without the attribute. In many cases, they are used to enable legacy JScript features or force JScript .NET classes to behave like JScript objects.

You can apply the `expando` attribute to a class so that it is given a default indexed property that can store and retrieve dynamic properties. This feature doesn't exactly fit into the .NET scheme, but it has been ported from JScript for compatibility purposes. With the special default indexed property, you can access any `expando` properties by using square brackets or parenthesis. You cannot access the class's non-`expando` properties in this fashion, so you should be careful with this feature.

CAUTION

If you're programming JScript .NET and want to use the new features and syntax rather than the legacy syntax, I recommend that you not use the `compatible` attribute because it causes a number of new features you'll want to make use of inoperable. If you use this attribute, you also miss quite a few compiler warnings that normally occur when you mistype property names because JScript .NET automatically creates a new property and uses the mistyped name to name it.

You use the `compatible` attribute to make classes in JScript .NET act the same as objects in previous versions of JScript. When you use this attribute, you lose the functionality of visibility modifiers and versioning features. However, you gain the ability to access `expando` properties by using the dot operator instead of having to use brackets or parenthesis.

You can't apply the `static` attribute to classes, but you can apply it to class members. Static members can be made available to users without an instance of the class having to be available. Static properties and fields maintain the same value across multiple instances of the same object being created and manipulated. We'll look more at static members in the section "Declaring Constructors" later in this chapter and again in Chapter 7, when we actually get into member methods.

The `static` attribute has a special behavior when applied to nested classes. Normally, nested class cannot be instantiated unless you first instantiate an instance of the class within which it is contained. You then use the instantiated class to create a new instance of the nested class. If you apply the `static` attribute, JScript simply treats the containing class name as if it were an addition to the namespace or package name, and you don't need an instance of the containing class to get an instance of the nested class.

Existential Attributes

Existential attributes—`abstract` and `final`—are extremely important in object-oriented programming and thus have gained great importance in JScript .NET. These attributes enable code to issue requirements on all derived classes. In the case of the `abstract` attribute, this means requiring a derived class to implement a method. The `final` attribute prevents the derived class from implementing a method.

When the `abstract` attribute is applied to classes, it makes them unavailable for use by the programmer. However, `abstract` classes can be inherited and used as base classes. When you have an `abstract` class, you need to first create a subclass and use the subclass rather than the `abstract` base class directly. You can also declare members to be `abstract`; this simply means that the member doesn't have an implementation and will expect any base classes to provide one.

By default, all `public` and `protected` members of a base class can be overridden and all classes can be used as base classes. To prevent this, you have to use the `final` attribute on the class or member. This effectively marks it as off limits so that it cannot be overridden or inherited by a derived class. The existential attributes work hand in hand with the versioning attributes, which are discussed in the following section.

Versioning Attributes

The versioning attributes, `hide` and `override`, are used to specify the type of behavior exhibited when implementing members of the base class in a derived subclass. The `hide` attribute specifies a new method that does not override the base class method, and it hides the base class implementation. The `override` attribute specifies that an implementation is provided for the base class method. Providing an overridden function is the default in JScript. The versioning attributes are a fairly complex set of attributes, and we'll discuss them further in the section "Inheritance and Subclassing."

NOTE

The most common use of the `hide` attribute is to change the return type of a function rather than simply overriding the function by using its normal return type. You can also use `hide` to specify implementations for members marked `final` in the base class.

Custom Attributes

The final group of attributes is the custom, or .NET, attributes. You can use these attributes on any classes or members, and these classes or members then become available as metadata in the final assembly. Consumers of the class or member act on custom attributes at runtime. For example, the `STAThreadAttribute` attribute forces the method or member to be executed under an STA (single-threaded apartment) thread instead of an MTA (multithreaded apartment) thread.

You can create your own attributes to hold special information about a class or function by creating classes that override `System.Attribute`. For example, you might want to create a documentation program that allows you to place descriptions on classes and members, using custom attributes. Later, you can use the reflection application programming interfaces to walk through the assembly and pull out the documentation along with type information for the classes and methods. The next section discusses the use of custom attributes for documentation purposes.

Writing a Custom Documentation Attribute

Some .NET languages have special documentation constructs. For instance, C# enables inline documentation comments that are parsed by the compiler and placed in an eXtensible Markup Language (XML) document. The compiler even validates that all the inline comments you specify are syntactically correct for the member to which they are applied.

JScript .NET does not offer such a feature, so you have to use some other methods to provide code documentation. For example, you can supply a special commenting structure that can later be parsed (similarly to C#). To do so, you can use a naming convention in which `.doc` files have the same names as source files, but different extensions. Or you can use the method you're about to create by applying special attributes with documentation.

To begin creating a custom attribute, you simply extend a class from `System.Attribute`. This isn't required, but it's highly recommended. You might also want to end the class name with `Attribute` because it helps in identifying attribute classes from normal classes. The following example creates an extremely basic attribute declaration:

```
class DocumentationAttribute extends System.Attribute {
}
```

Next, you have to apply some .NET attributes that describe the behavior of the documentation attribute. `AttributeUsageAttribute` accepts a single ordinal parameter and optional named parameters. The ordinal parameter accepts the type `AttributeTargets`, which is an enumeration for the types of members on which the attribute is valid. You could limit the `DocumentationAttribute` to only classes or only functions, but if you use the `All` option, it can be applied to any construct.

Next, specify the two optional parameters `AllowMultiple`, with a value of `false` because you want only one documentation item per construct, and the `Inherited` property, with a value of `false` because you don't want documentation applied at the class level to be inherited by class members and properties. The following example uses a special attribute to specify how `DocumentationAttribute` can be used in code:

```
public System.AttributeUsageAttribute(
    AttributeTargets.All,
    Inherited=false,
    AllowMultiple=false)
class DocumentationAttribute extends System.Attribute {
}
```

Listing 6.1 demonstrates how to expand on the documentation attribute to specify that there is one required property, `ShortName`, for the attribute. Sometimes class names aren't as descriptive as they should be and the `ShortName` property of the new documentation attribute can be multiple words, a small title, or as long as it needs to be (we can't limit the length really). You can also add some other properties, such as `LastModified`, `Author`, and `Description`. You can add more properties later if you like. Compile this documentation attribute library as `Documentation.dll`, and maybe later you can use the attributes on some of your source code.

Listing 6.1 A Documentation Attribute Library (`Documentation.js`)

```
import System;

public System.AttributeUsageAttribute(
    AttributeTargets.All,
    Inherited=false,
    AllowMultiple=false)
class DocumentationAttribute extends System.Attribute {
    private var shortName:String;
    private var lastModified:String;
    private var author:String;
    private var description:String;

    // One require property
    function DocumentationAttribute(shortName:String) {
        this.shortName = shortName;
    }

    // Since it is required we only need a get;
    public function get ShortName() : String {
        return shortName;
    }
```

LISTING 6.1 continued

```
    // These are optional and name so we need a get; set;
    public function get LastModified() : String {
        return lastModified;
    }
    public function set LastModified(value:String) {
        lastModified = value;
    }

    public function get Author() : String {
        return author;
    }
    public function set Author(value:String) {
        author = value;
    }

    public function get Description() : String {
        return description;
    }
    public function set Description(value:String) {
        description = value;
    }
}
```

Inheritance and Subclassing

A language gains a lot of power through code reuse. Inheritance and subclassing both enable code reuse by allowing new classes to reuse old classes by inheriting and overriding the old methods while at the same time providing new methods on the new subclass. This can be extremely powerful because older programs can continue to work with the new object, using only the older methods, and newer programs can use the updated object in its entire splendor by accessing both new and old methods.

In this section we'll first talk about extending other classes. The `extends` keyword allows a class to extend a base class, as long as it isn't marked `final`, in which case it can't be overridden. Also notice the `implements` keyword. This keyword enables you to implement interfaces which is a contract made by your object that you'll support all the methods defined in the interface and thus an object can act as if it were an instance of the interface in question. (You can find more information and a more complete description of interfaces in Chapter 7.)

This section also discusses a third form of inheritance, called encapsulation. In encapsulation, you never extend a base class or implement any interface; instead, you instantiate an instance of the object in question and provide your own methods as wrappers around the object's methods. This enables you to perform some custom logic.

Extending Classes

In JScript .NET you use the `extends` keyword to append a single class name to the class declaration for a class. Extending a class allows you to provide new functionality for existing classes, to inherit basic functionality into multiple derived objects, or even to specify some strong contract between derived classes so that code that works with the base class also works with the derived classes. Extending a base class allows the new derived class to perform in three different situations:

- It can be used to provide some implementation for a defined but not implemented method.
- It can provide new functionality for an existing method.
- It can implement new methods with different names.

Earlier in this chapter, we discussed the `abstract` attribute that forces you to subclass another class before it can be used. Whenever the `abstract` attribute is specified at the class level, that class cannot be instantiated. Therefore, you are forced to derive a class and implement all the methods and properties of the class. In this way, the base class acts as only a contract (that is, it acts like an interface that defines the definitions for methods and properties yet doesn't provide the implementation) between the derived class and any code that uses the base class, and it expects the derived class to operate accordingly.

You can also override some common functionality of a base class and provide some new functionality. This is a fairly common practice, and new methods are often created for some process performed by the base class. In turn, you derive a new class by inheriting all the other methods of the base class, and then you override the one method where a new process has been found and implement that process. This is versioning at its finest because the base class is left unchanged and the new class can act in place of the base class with its new functionality.

If you want a class to be used by older programs, you can inherit the base class, but you allow newer programs to use some of the additional methods you supply. In this manner, you extend the existing contract by adding new methods and properties. Any classes that are in turn derived from your class will then inherit this enhanced contract. This is another form of versioning because the new class can still be used in place of the base class in programs that only know about the base class for the class. The following example demonstrates the use of abstract classes and methods, as well as how to override base class methods and do method inheritance:

```
import System;

// A can't be instantiated.  But you can still program against its methods
abstract class A {
    function AFunction() : void {
    }

    function AProcess() : void {
    }

    // A defines a function that must be overridden by a base class since
    // it provides no implementation
    abstract function OverrideProcess() : void;
}

class B extends A {
    // B inherits AFunction()
    // B overrides AProcess since a new one is found
    override function AProcess() : void {
    }

    // B implements the OverrideProcess function since it has to by contract
    override function OverrideProcess() : void {
    }

    // B defines new methods in the contract
    function BFunction() : void {
    }
}

class C extends B {
    // C gets all of the functions from A not overridden by B
    // C gets all of the functions from B including the new function
    // At this point C can continue the chain of evolution
}
```

Using Interfaces Over a Subclass

Interfaces are extremely powerful. Whereas a class can implement only one base class, it can implement more than one interface. So when should a base class be used and when should an interface be used? There is a simple answer to this question, and it hinges on something you've seen already, in the code for extending classes. Base classes can provide function and property implementations. This is not true of interfaces. Interfaces are contracts, and they provide no implementation; they require the implementer to provide any implementation. Basically, an interface is like an abstract class, with all abstract members and properties. You therefore use a base class when you want base implementations, and you use interfaces when you find yourself deriving from a class that has no implementation and is fully abstract.

NOTE

If you extend a base class that implements an interface, that interface is in turn automatically inherited by the derived class. Oftentimes it might feel right to implement the interface in the derived class as well, but that is unnecessary. Although it's not a compiler error or a runtime error, it introduces additional code baggage.

To implement an interface or multiple interfaces, you use the `implements` keyword, either immediately following the class name or immediately following the `extends` declaration. The interfaces that are being implemented should be separated by a comma. The following example demonstrates the use of the `implement` and `extends` keywords to extend base classes and implement interfaces:

```
class Class {
    // Normal class
}

class Class1 implements Interface1 {
    // Class definition with one interface
}

class Class2 extends Class implements Interface1 {
    // Class definition with base class and one interface
}

class Class3 implements Interface1, Interface2, Interface3 {
    // Class definition with many interfaces
}
```

Using Encapsulation Over a Subclass

Often, extending a single subclass in a single inheritance language puts you at a programming disadvantage because as soon as you decide you need to create one type of object, you limit it from ever being another type of object. You might also face situations in which you need to extend the abilities of a class that is sealed with the `final` keyword. In both of these instances, encapsulation comes into play because you're not limited to the number of types you can encapsulate, and you can always encapsulate those sealed classes by housing instances of them in a class.

Say you want to override more than one base type. For example, you need a special collection class that enables you to treat a collection as a normal array that simply maintains the order of the elements inserted, but you also need a hashtable because lookup times on an array with a large number of elements would be extremely poor. One option is to subclass the `System.Collections.CollectionBase` class, which gives you access to an `ArrayList` property. But this means that `CollectionBase` is already encapsulating another object because it is exposing `System.Collections.ArrayList` as a property. Therefore, you should encapsulate both an `ArrayList` and a `Hashtable` property within the object because it allows you to customize access to both the arraylist and hashtable through your own custom methods and properties.

In addition to encapsulating the `ArrayList` and the `Hashtable` properties, you can limit additions to the collections to a single type (the `Employee` class) that you create. The `Employee` class can be searched by name (the reason for the `Hashtable` property) or another lookup can be done via index (the reason for the `ArrayList` property). Providing the `ArrayList` property also gives you the option to sort the contents for display purposes.

Listing 6.2 shows how you can add a `Clear` method to clear both of the encapsulated lists. This is a good example of special logic because instead of only clearing the base collection type, which you would have done had you inherited, you are clearing all the contained collections.

LISTING 6.2 Encapsulating Collection Classes

```
import System;
import System.Collections;

class OrderedHash {
    private var _sortList = new ArrayList();
    private var _arrayList = new ArrayList();
    private var _hashTable = new Hashtable();

    public function Clear() : void {
        _arrayList.Clear();
        _hashTable.Clear();
    }

    public function Add(emp:Employee ) : int {
        if ( _hashTable.ContainsKey(emp.Name) ) {
            return -1;
        }

        _hashTable[emp.Name] = emp;
        return _arrayList.Add(emp);
    }
```

LISTING 6.2 continued

```
    public function Sort(dir:Boolean) : ArrayList {
        _sortList = _arrayList.Clone();

        if ( dir )
            _sortList.Sort(new EmployeeSortAsc());
        else
            _sortList.Sort(new EmployeeSortDesc());

        return _sortList;
    }

    public function Lookup(key:String) : Employee {
        return _hashTable[key];
    }

    public function get OrderedList() : ArrayList {
        return _arrayList;
    }
}

class EmployeeSortAsc implements IComparer {
    function Compare(x:Object, y:Object) : int {
        var emp1name:String = Employee(x).Name;
        var emp2name:String = Employee(y).Name;

        return emp1name.CompareTo(emp2name); // Compare using String comparer
    }
}

class EmployeeSortDesc implements IComparer {
    function Compare(x:Object, y:Object) : int {
        var emp1name:String = Employee(x).Name;
        var emp2name:String = Employee(y).Name;

        return emp2name.CompareTo(emp1name); // Reverse Comparison
    }
}

class Employee {
    private var name:String;

    function Employee(name:String) {
        this.name = name;
    }
```

LISTING 6.2 continued

```
    public function get Name() : String {
        return this.name;
    }
}

var rand:Random = new Random();
var o:OrderedHash = new OrderedHash();
var j:int = 0;
var empNumber:int = 0;
for(j = 0; j < 10; j++) {
    empNumber = rand.Next(1000,10000);

    o.Add(new Employee("Employee" + empNumber));
}

Console.WriteLine();
Console.WriteLine("Unsorted");
for(j = 0; j < 10; j++) {
    Console.WriteLine(o.OrderedList[j].Name);
}


Console.WriteLine();
Console.WriteLine("Sorted ASC");
for(j = 0; j < 10; j++) {
    Console.WriteLine(o.Sort(true)[j].Name);
}

Console.WriteLine();
Console.WriteLine("Sorted DESC");
for(j = 0; j < 10; j++) {
    Console.WriteLine(o.Sort(false)[j].Name);
}

Console.WriteLine();
Console.WriteLine("Lookups");
Console.WriteLine(o.Lookup("Employee" + empNumber).Name);
```

In Listing 6.2, the `Add` function takes only an `Employee` class. The program checks to make sure the employee doesn't already exist (functionality gained by having a `Hashtable` property) and then adds the employee by name to the hashtable and then by ordinal to the arraylist. The function returns the ordinal, in case you want to use it later, but because you'll usually do employee name lookups, the value isn't of much importance.

The `Sort` function is supplied so that the employee collection can be sorted by name in either ascending order or descending order. This function uses two special objects, which support an `IComparable` interface so you can specify sort direction. Otherwise, you could sort only by ascending order.

Finally, the `Lookup` function returns the employee object from the hashtable by employee name and the `OrderedList` function returns the arraylist without doing any sorting or modifications to the collection.

Although encapsulation can be used to support the functionality of multiple objects, it is generally used when a class is marked `final` and is sealed so that it can't be overridden, you have to encapsulate that object when you want to increase its functionality. A prime candidate for this would be the `System.String` class. Although the string class probably already contains most of the methods you could think of, you might want some of its default functionality changed or you might want to add a method.

A prime example for changing the default functionality would be encapsulating the `String` class in a new object and providing a new implementation for the `Trim` function, as shown in Listing 6.3. By default, `Trim` specifies about 30 different ASCII and Unicode characters as being valid characters to trim from a string. However, you might want only the default space (that is, `32`) to be trimmed. Therefore, you could specify a custom `Trim` method that in turn calls the overloaded `Trim` method on the `String` class, with the new whitespace characters.

LISTING 6.3 Encapsulating the `String` Class

```
import System;

class EnhancedString {
    private var _string:String;

    function EnhancedString(str:String) {
        _string = str;
    }

    // You'll have to implement all of the
    // String methods you wish to expose
    // This is sometimes time consuming and tedious

    // We are just going to do the Trim function
    // though.
    function Trim() : String {
        return _string.Trim([' ']);
    }
}
```

LISTING 6.3 continued

```
var enhstr:EnhancedString = new EnhancedString("   Hello World   ");
Console.WriteLine(enhstr.Trim());
```

Listing 6.3 includes a special trim function, and because you have encapsulated the `String` datatype, you now also need to at least supply stub functions for the rest of the methods or you need to expose the internal `String` object as a property and thus defeat the purpose of making the user use the new functions. If it isn't important that the user use the new functions, and it is only a nice-to-have feature, then exposing the `String` datatype as a property wouldn't hurt. It is a design decision, and more importantly it's your design decision.

Declaring Constructors

Constructors are the lifeblood of a class definition. Whenever a user creates an instance of a class, he needs an easy way to specify defaults for parameters or pass in any information needed for the class to perform its basic operation. Listing 6.3 encapsulates the `String` class, so it includes a constructor that takes a `String` object. The `EnhancedString` class is worthless without this initialization data because it can't perform operations on a null object.

Many classes simply can't function without some up-front data. Constructors fill this gap. In the following subsections we'll talk about the default constructors, overloaded constructors, and base constructors.

Default Constructors

The default constructors (`empty` and `copy`) are constructors that the JScript compiler automatically creates for you whenever you specify a class definition. The empty constructor is important because it enables the creation of the class. Without the empty constructor, you would have to create at least one custom constructor for each class, or the classes could not be used. The `copy` constructor is abstract and is used to create an exact copy of a class.

By default, JScript creates the `empty` constructor, which is simply a constructor for an object that takes no parameters. If you need to do some custom initialization in the `empty` constructor, you can specify a constructor yourself, in which case the JScript compiler doesn't generate an `empty` constructor for you.

What happens if you don't want any constructors at all for a class, such as when you want to generate a class with only static members? It doesn't make sense for the class to be created as an instance type, so you can specify an `empty` constructor with the `private` attribute. In this case, the class cannot be created outside the class itself, but any static member within the class can create an instance and return that to the program. The following example includes class declarations that use default constructors, custom constructors, and private constructors:

```
// This creates a class with a default constructor
class DefaultConstructor {
}

// This creates a class where the user specifies a default constructor
class SpecifyDefaultConstructor {
    function SpecifyDefaultConstructor() {
    }
}

// This creates a class which can't be instantiated because it has a private
➥constructor
class PrivateConstructor {
    private function PrivateConstructor {
    }

    // However, static methods can return instances still
    static function CreateInstance() : PrivateConstructor {
        return new PrivateInstance();
    }
}
```

Read this code well because it throws curve ball: It adds the `CreateInstance` method, which can still return an instance of the `PrivateConstructor` class. Why would you ever use initialization methods instead of constructors? It's a matter of programming style. For example, I try to fit everything I can into constructor overloads, but often I still end up having a `private` method I call with all the details and let the function, rather than one of the overloads, do the work of initializing the instance. One bonus of this is that if the call to `CreateInstance` specifies all the parameters needed to fully initialize a class, you can use it to make a clone or copy of the class.

JScript .NET automatically generates some code for a `copy` constructor, but you'll never see this code unless you open the assembly by using `ILDasm` and actually look at the generated intermediate language (IL) code and metadata.

TIP

If you need to control the way an object is copied, you should support the `ICloneable` interface. This interface specifies that you support the `Clone` function and is very easy to implement as long as all the values that need to be copied from private member variables also support the `ICloneable` interface. Some things to watch out for are passing objects from one instance to a cloned instance. In some cases, you might find that operating on data in one object automatically updates data in another object because you passed the data by reference and not by value.

It’s nice of JScript to create the `empty` and `copy` constructors for us to use when we are in a hurry. But classes are complex and often need some initialization data. For that, you need to use overloaded constructors, which are discussed in the next section.

Overloaded Constructors

An overloaded constructor contains one or more parameters that are used by the class to initialize internal data during class creation. You can add as many or as few parameters as you like. Overloaded constructors allow very powerful customization of the initial state of a class, as shown in the following example that is capable of taking a single parameter of type `string` or several parameters of type `String`, `Object`, and `int`:

```
class ConstructorWithOneParm {
    function ConstructorWithOneParm(myParm:String) {
    }
}

class ConstructorWithSeveralParms {
    function ConstructorWithSeveralParms(parm1:String, parm2:Object, parm3:int) {
    }
}
```

As soon as you specify an overloaded constructor, JScript omits its default `empty` constructor. So if you want one of those as well, you need to specify it yourself or you need to use the new constructor you’ve just created for each instance.

So far in this chapter, we have specified only a single constructor for each of the classes. You can specify as many as you like, as long as you don’t specify two constructors of exactly the same type. They can even have the same number of arguments, as long as the arguments are of different types. You need to be careful when calling constructors with the same number of parameters but different types because JScript can often coerce one type to become multiple types and might throw an error if it can’t decide which constructor it should call.

You can also call another constructor from the constructor you are currently within by using the `this` keyword. The only rule for calling into another constructor is that you must call it on the first line of the current constructor. There are two methods for calling additional constructors to initialize an object. The first method is to cascade down the constructor tree until you reach the empty constructor. This works when you don’t require data to create an instance of the class, but you want the user to be able to specify as much on the constructor as desired and then specify the rest later, using methods for parameters. In Listing 6.4, `CascadeDownConstructor` implements this type of constructor system. If the user only calls the default constructor, the `rand` variable is the only initialized variable. If the user calls the constructor that allows users to pass in name, then the name constructor calls down into the default constructor such that both `name` and `rand` are initialized. Listing 6.4 is an example of a class that uses `CascadeDownConstructor` and `CascadeUpConstructor`.

LISTING 6.4 Cascading Constructor Methods

```
import System;

class CascadeDownConstructor {
    private var rand:Random;
    private var name:String;
    private var id:int;

    function CascadeDownConstructor() {
        this.rand = new Random();
    }

    function CascadeDownConstructor(name:String) {
        this();
        this.name = name;
    }

    function CascadeDownConstructor(name:String, id:int) {
        this(name);
        this.id = id;
    }
}

class CascadeUpConstructor {
    function CascadeUpConstructor() {
        this("");
    }

    function CascadeUpConstructor(name:String) {
        this(name, 0);
    }

    function CascadeUpConstructor(name:String, id:int) {
        this.name = name;
        this.id = id;
        this.rand = new Random();
    }
}
```

The second and more widely used method for calling additional constructors to initialize an object is cascading up the tree from the least explicit constructor to the most explicit constructor. When you're using this method, you supply default values for each call up the constructor tree. This means that at the end of the constructor calls, you will eventually have a fully initialized class instance, even though many of the values are default values. Listing 6.4 identifies a class by cascading up the constructors. Notice that in case a user specifies the default constructor, you specify an empty string to the second constructor, which takes the name as a string. This constructor in turn calls up to the most explicit constructor with the value of `name` and `0` for the default value of `id`. When the final constructor is reached, you perform all the initialization for the class and set up all the variables. Depending on which constructor the user calls, there will be more or fewer default values.

Default constructors let you specify data for a class instance, but what about when you're overloading other classes? How is the data normally handled by the base class instance called and handled? These questions are answered in the following section, on base constructors.

Base Constructors

Base constructors enable a derived class to notify a base class of the construction phase. This notification of the construction allows the base class to initialize any private member data to which the derived class might not have access. When you derive a class from another class, the derived class automatically gains all the `public` or `protected` methods and properties of the base class. However, it doesn't inherit any of the constructors. Therefore, every derived class must implement its own constructors, and if it needs to call the base constructors, it must do so explicitly. You use the `super` keyword to access the immediate base class of a class. For example, you specify that the `Every` class has a base class, even if you don't specify one. The `System.Object` class is the default base class for any classes that are not explicitly derived from another class.

You're probably wondering why you need to call a constructor for the base class of a derived class. Unless you override all the methods that are available from the base class, some data will be needed so that if the user calls a derived method, the program doesn't throw an error. For instance, let's use the `EnhancedString` from Listing 6.3 as an example that we can override as `SuperAdvancedString` in Listing 6.5.

LISTING 6.5 Using Base Class Constructors

```
import System;

class EnhancedString {
    private var _string:String;

    function EnhancedString(str:String) {
        _string = str;
    }

    // You'll have to implement all of the
```

LISTING 6.5 continued

```
    // String methods you wish to expose
    // This is sometimes time consuming and tedious

    // We are just going to do the Trim function
    // though.
    function Trim() : String {
        return _string.Trim([' ']);
    }
}

class SuperEnhancedString extends EnhancedString {
    private var _string:String;

    function SuperEnhancedString(str:String) {
        super(str);

        _string = str;
    }
}

var enhstr:SuperEnhancedString = new SuperEnhancedString("   Hello World   ");
Console.WriteLine(enhstr.Trim());
```

Because the `_string` variable of the `EnhancedString` class is `private` and not `protected`, you can't access it directly from the derived class. The ideal method for fixing this problem is to recompile `EnhancedString` and change the visibility of the `_string` variable so that you can access it from within the constructor of `SuperAdvancedString`. Because this design flaw wasn't seen by the makers of the original `EnhancedString` class, you have to work around it by using the base class constructor and maintaining your own private version of the same string.

NOTE

It's important to plan classes for the future rather than resort to after-the-fact fixes such as maintaining two instances of the same variable. If you continued along this path and kept declaring a variable `private` every time, you would end up having many instances of the same variable. Also, what happens if you have a property that lets you change the value in `_string`? `SuperEnhancedString` has this property, but you now have no way of setting the new value on the base class because it never implemented this property. In this way, the two strings can get out of sync. The lesson here is to plan classes intelligently and always enable them for being subclassed in the future or mark them as final so they will never be subclassed.

After you have compiled and run Listing 6.5, try removing the call to the base class constructor. You get a null reference exception because when you call the `Trim` method, you are actually calling the `Trim` method of the `EnhancedString` base class. Because that class hasn't been properly initialized, its `_string` variable is null, so you get a null reference exception. You could work around this another way, by simply overriding the `Trim` method, but that defeats the purpose of subclassing base types and using their functionality within the new type.

Summary

You should now know everything there is to know about the various class declarations available to you in the JScript .NET language. In this chapter you learned about all the various attributes and visibility modifiers. You also learned about inheriting from a base class and implementing interfaces. In addition, you learned about the various constructor options that are available, including default constructors, overloaded constructors, and base constructors.

Chapter 7 covers the intricacies of creating your own interfaces. It also discusses implementing methods, properties, and fields, which are be the real meat of the class declaration. Chapter 7 also includes a discussion of event support in JScript .NET.

Interfaces and Class Members

CHAPTER

7

IN THIS CHAPTER

In this chapter we'll discuss some more complex areas of object-oriented programming with JScript .NET. We'll briefly discuss the use and construction of interfaces, including the basic interface structure, the attributes that apply to interfaces only, and the types of members an interface can contain. We'll also discuss how class members must be declared for the compiler to hook them up to any interfaces you're implementing, and how to resolve conflicts when interfaces have definitions for the same member.

We talked briefly about methods in Chapter 6, "Creating Classes," and in this chapter we'll get much more in-depth about them, covering all the intricacies of return types, the attributes that can be applied, where those attributes can be applied, and parameter overloading.

In this chapter we'll also discuss fields and properties, which are programmatically and syntactically very similar in nature. We'll discuss all the ins and outs of using fields over properties and vice versa, as well as some of the more complex options available for properties that can't be performed using field variables.

This chapter ends with a description of event support in JScript .NET. JScript .NET currently doesn't support many of the event features of the common language runtime (CLR). In the future, support for events will become more feature rich, but for now you need to consume events and derive events from classes that already have events for your JScript .NET class to inherit.

An Overview of Interfaces

This section is devoted to interfaces in JScript .NET: their structure and definition, as well as the various attributes and members that are applicable only to interface definitions. An interface definition can be considered an extremely limited subset of what is available when you're defining classes. It is also important to remember that an interface is never created directly, but has to be implemented by a class before it can be used. For this reason, it is good practice to define an interface with as few methods as possible so that it is easy to implement in a class.

Interface Structure

You declare an interface by specifying attributes, a name for the interface, and a block in which the interface members are defined. Like the formal declaration for a class shown in Figure 6.1 in Chapter 6, the formal interface syntax seen below shows more information than is required to work with basic interfaces. You create an interface, in the simplest form, by simply using the `interface` keyword along with the name of the interface. An interface can have an empty code block without any method or property definitions. The following code demonstrates a simple interface:

```
interface A {
}
```

Here is a more formal interface syntax:

```
[Attributes] interface InterFaceName [implements interfaces, ...]{
...Code Block...
}
```

You can cause interfaces to inherit from a base interface by using the `implements` keyword. For classes, the `implements` keyword causes a class to implement all the methods and properties defined by the interface. With interfaces, this definition changes a bit because an interface by definition can't provide any implementation—it can only define the prototypes for the methods and properties. This means that an interface can use the `implements` keyword to subclass other interfaces, as in the following example:

```
interface A {
}
interface B {
}
interface C implements A,B {
}
```

In this example, any methods and properties defined by `interface A` or `interface B` will also be inherited by `interface C`. This can be extremely handy as a grouping construct, in which a series of interfaces can be grouped into one larger interface to ease the programming burden. The following example groups a series of graphics interfaces together so that someone wanting to implement all the interfaces would need to implement only the grouped interface and not all the smaller interfaces:

```
interface IViewport {
}
interface IRender {
}
interface ILoadGraphics {
}
interface ISetPixel {
}
interface IGraphics implements IViewport, IRender, ILoadGraphics, ISetPixel {
}
```

In addition to reducing the amount of code needed to implement the interfaces, this also reduces the amount of code needed to check whether an object supports the necessary interfaces. If you have a piece of code that works with a class that you know supports some of these interfaces and you need to use two of them, then you need to check whether it supports both. However, if the class supports the `IGraphics` group of interfaces, you need to make only one check. The following example works in conjunction with the previous example and implements the interfaces defined there:

```
class NormalGraphics implements IViewport, IRender {
}
class SmartGraphics implements IGraphics {
}

var norm:NormalGraphics = new NormalGraphics();
if ( norm instanceof IViewport && norm instanceof IRender ) {
    // do some work
}

var smart:SmartGraphics = new SmartGraphics();
if ( smart instanceof IGraphics ) {
    // do some work
    // note that smart would also pass the
    // first test against IViewport and IRender
}

// Perform both code-paths to see perf gain when using IGraphics
if ( norm instanceof IGraphics ||
     (norm instanceof IViewport && norm instanceof IRender)) {
    // If the object supports IGraphics then we fall through quickly
    // else we perform the two extra comparisons to make sure
    // the right methods are supported.
}
```

When you implement the `IGraphics` interface, the class also implements the other four defined interfaces. This means `smart` would pass the test against `IGraphics` or any of the inherited interfaces. This means you really wouldn't have to have different code paths to test `IGraphics` unless you really wanted the performance benefit of performing only one test.

Interface Attributes

Interfaces are extremely simple in nature, and as a result, they don't allow very many modifications from different attributes. They do support two visibility attributes, though, `public` and `internal`. As with classes, the `public` attribute can be used to make the interface visible to any piece of code. The `public` attribute is the default attribute, and any interface not explicitly marked otherwise is `public`.

The `internal` attribute defines a packagewide interface. If it is defined outside of package, the interface becomes visible to all code that is defined in the same assembly. The best use of the `internal` attribute is to limit access of a given interface to only the code within an assembly. The following example defines several interfaces, using various visibility attributes:

```
public interface A {
    // public is default and redundant, but also explicit
}
internal interface B {
    // internal makes this available only to code within the current assembly
}
package Hide {
    internal interface C {
        // internal make this available only to code within the same package
        // it would be illegal to access this from outside of the hide package
    }
}
```

One final form of attribute can exist on the interface: the CLR custom attributes. Any attribute with a target of `All` or `Interface` can be attached to an interface definition and be used to modify the interface definitions metadata.

The remaining interface definition items exist within the interface code block. These are the member definitions for the interface itself and appear in the form of methods and properties. These, in turn, can also have some additional attributes applied, and we'll discuss that in the following section.

Interface Members

Interface members are method or property definitions. This type of definition contains no implementation and is abstract and must be implemented by the class that implements it. Only two types of definitions can appear in an interface code block. You can declare method prototypes, which is similar to declaring the `abstract` prototypes for an `abstract` class. You can also declare properties with `get` and `set` functions. You cannot declare fields or property indexers in JScript .NET as you can in some other .NET languages.

All interface members, both methods and properties, can only be given the `public` visibility attribute. Because this is the default attribute, you don't need to specify it unless you want to be explicit. Each prototype ends with a semicolon and cannot contain a code block. The following snippet shows some basic examples of interface methods and properties:

```
interface A {
    function MyFunction() : void;
    function get MyProperty() : int;
    function set MyProperty(i:int);
}
```

Implementing Interfaces

Implementing interfaces in JScript .NET is as easy as declaring a method or property with the same prototype as that found in the interface. Because an interface is a contract to other classes that says the class will support the required methods and properties, you must always declare the members you implement as `public`.

To show interface implementation, let's begin by defining the interfaces we'll be using. The following example defines three interfaces to show the different types of member implementation:

```
interface A {
    function MethodA() : void;
    function get PropertyA() : int;
    function set PropertyA(i:int);
}
interface B implements A {
    function MethodB() : void;
    function get PropertyB() : int;
    function set PropertyB(i:int);
}
interface C {
    function MethodA() : void;
    function MethodB() : void;
    function get PropertyA() : int;
    function set PropertyB(i:int);
}
```

The first interface, `interface A`, has a simple method and a simple property with both the `get` and `set` methods. The second interface, `interface B`, implements `interface A`; it also has its own set of methods. The third interface, `interface C`, is what you could call the spoiler. This interface does not implement either `interface A` or `interface B`, but it has some methods and properties whose names are identical to method and property names in the other interfaces.

Simple Interface Implementation

The first class definition, shown in Listing 7.1, attempts to implement `interface A` only. You implement each of the prototypes for `interface A` and declare them with the `public` attribute, as required by the interface contract. Notice that you don't specify that the methods in any way belong to `interface A` because the compiler does all the name resolution for you.

LISTING 7.1 Implementing a Simple Interface

```
interface A {
    function MethodA() : void;
    function get PropertyA() : int;
    function set PropertyA(i:int);
}
```

LISTING 7.1 continued

```
class SimpleInterface implements A {
    public function MethodA() : void {
    }
    public function get PropertyA() : int {
        return 1;
    }
    public function set PropertyA(i:int) {
    }
}
```

Inherited Interface Implementation

The class definition shown in Listing 7.2 consumes `interface B`. This means you need to implement both the methods of `interface A` and `interface B` before the program can compile because of method inheritance from `interface A` to `interface B`. You also need to write some code that casts an instance of the class to each of the underlying interfaces and tests the methods. This should help you understand what is happening behind the scenes and prove that the methods are overriding the interface methods as they should.

LISTING 7.2 Implementing an Inherited Interface

```
import System;

interface A {
    function MethodA() : void;
    function get PropertyA() : int;
    function set PropertyA(i:int);
}
interface B implements A {
    function MethodB() : void;
    function get PropertyB() : int;
    function set PropertyB(i:int);
}

class InheritedInterface implements B {
    public function MethodA() : void {
    }
    public function get PropertyA() : int {
        return 1;
    }
    public function set PropertyA(i:int) {
    }
```

LISTING 7.2 continued

```
    public function MethodB() : void {
    }
    public function get PropertyB() : int {
        return 1;
    }
    public function set PropertyB(i:int) {
    }
}

var o:InheritedInterface = new InheritedInterface();
var a:A = o; // get interface instances
var b:B = o; // get interface instances

Console.WriteLine(o.PropertyB);
Console.WriteLine(o.PropertyA);

Console.WriteLine(a.PropertyA); // Doesn't have a PropertyB

Console.WriteLine(b.PropertyB);
Console.WriteLine(b.PropertyA);
```

Compiler-Defined Member Implementation

What happens when multiple interfaces have the same definition? There are two options: You can let the compiler use the same function for both interfaces or you can explicitly tell the compiler that a particular implementation member belongs to a specific interface. Listing 7.3 demonstrates implementing both `interface B` and `interface C`. Note that you need to provide only one implementation for each of the function names, but the compiler uses the same function, whether you are using `interface B` or `interface C`.

LISTING 7.3 Implementing Interfaces with Identical Prototype Members

```
import System;

interface A {
    function MethodA() : void;
    function get PropertyA() : int;
    function set PropertyA(i:int);
}
interface B implements A {
    function MethodB() : void;
    function get PropertyB() : int;
```

LISTING 7.3 continued

```
    function set PropertyB(i:int);
}
interface C {
    function MethodA() : void;
    function MethodB() : void;
    function get PropertyA() : int;
    function set PropertyB(i:int);
}

class InheritedInterface implements B, C {
    public function MethodA() : void {
    }
    public function get PropertyA() : int {
        return 1;
    }
    public function set PropertyA(i:int) {
    }

    public function MethodB() : void {
    }
    public function get PropertyB() : int {
        return 1;
    }
    public function set PropertyB(i:int) {
    }
}

var o:InheritedInterface = new InheritedInterface();
var b:B = o; // get interface instances
var c:C = o;

Console.WriteLine(o.PropertyB);
Console.WriteLine(o.PropertyA);

Console.WriteLine(c.PropertyA); // Doesn't have a PropertyB

Console.WriteLine(b.PropertyB);
Console.WriteLine(b.PropertyA);
```

Explicitly Defined Member Implementation

Listing 7.4 declares different implementations for `interface B` and `interface C`. To do this, you need to explicitly tell the compiler that a particular method or property implementation belongs to a specific interface, by using the interface name followed by a period before the member name. You do this for just the `interface C` functions. The rest of the functions use the format demonstrated in the section "Compiler-Defined Member Implementation." This provides a default implementation for `interface A`, `interface B`, and the class, but it provides an explicit implementation for `interface C`. This is an advanced interface topic, but it comes in handy whenever there are two interfaces with identical member names, but the implementations for members need to be explicitly different. It is also possible that the class might have a method-naming conflict with an implemented interface method, and the explicit interface implementation might allow the class and interface implementation to be defined separately and have different functionality. Listing 7.4 demonstrates how to provide explicit member implementations.

LISTING 7.4 Using Different Implementations for Different Interfaces

```
import System;

interface A {
    function MethodA() : void;
    function get PropertyA() : int;
    function set PropertyA(i:int);
}
interface B implements A {
    function MethodB() : void;
    function get PropertyB() : int;
    function set PropertyB(i:int);
}
interface C {
    function MethodA() : void;
    function MethodB() : void;
    function get PropertyA() : int;
    function set PropertyB(i:int);
}

class ExplicitInterface implements B, C {
    public function MethodA() : void {
    }
    public function get PropertyA() : int {
        return 1;
    }
    public function set PropertyA(i:int) {
    }

    public function MethodB() : void {
    }
```

LISTING 7.4 continued

```
    public function get PropertyB() : int {
        return 1;
    }
    public function set PropertyB(i:int) {
    }

    public function C.MethodA() : void {
    }
    public function C.MethodB() : void {
    }
    public function get C.PropertyA() : int {
        return 2;
    }
    public function set C.PropertyB(i:int) {
    }
}

var o:ExplicitInterface = new ExplicitInterface();
var a:a = o;
var b:B = o;
var c:C = o;

Console.WriteLine(o.PropertyB);
Console.WriteLine(o.PropertyA);

Console.WriteLine(a.PropertyA); // Doesn't have a PropertyB

Console.WriteLine(c.PropertyA); // Doesn't have a PropertyB

Console.WriteLine(b.PropertyB);
Console.WriteLine(b.PropertyA);
```

That is the most advanced form of interface implementation that can be performed. Very rarely will you need to override the member implementation that the compiler does for you automatically. In many cases, unless the functionality needs to be drastically different, you don't want to provide explicit implementations for each interface. Doing so creates quite a bit more programming work, but it means you can specify default implementations for all interfaces, as shown in Listing 7.4, and you override only the methods and properties that you really need to override.

Declaring Methods

Nearly every class must have at least one method to be useful. Methods allow you to modify the internal state of a class while it is running by passing in parameters. You can also retrieve information about the state of the class through method return values.

Methods aren't just for the programmer on the outside looking in. A class can implement special private methods that only the class itself can call. For example, you might not want to put an initialization function in a constructor, and you might instead call it from the constructor.

Some methods can be made visible only to classes that are derived from your given class. These methods can become extremely important when you're creating a series of classes that share some functionality and you don't want to have to copy the function around. By making the methods available only to a class and its derived classes, you can ensure that code outside these classes won't call the function. You can use this to limit access to important data.

Class Method Syntax

Every method definition in JScript .NET must include the `function` keyword so that JScript .NET can determine that it is indeed a function prototype and not some other form of inline code or variable syntax. The following syntax shows that the `function` keyword is a required component of the class method definition. Properties immediately precede the `function` keyword, and the function name immediately follows the `function` keyword.

```
function MethodName() : ReturnType{
...Code Block...
}
```

Every function must also have at least a blank parameter list, delimited by parentheses, followed by a colon and the return type of the function. The return type can generally be of type `void` to indicate that the function has no real return value. Let's put this to work now with a sample method:

```
class Sample {
    function A() : void {
    }
}
```

In this example you don't even need to specify a visibility attribute because `public` is the default. (If you've forgotten the available class and member attributes, refer to Chapter 6.)

If you need to specify any attributes, you do so before the `function` keyword. The attributes you specify could be any of the JScript .NET defined attributes or any custom CLR attributes. Placing these attributes on methods can get pretty complicated in terms of order and such, so the general recommendation is to place visibility attributes first, CLR attributes next, and then any of the other forms of attributes that are applicable to class methods. (This isn't a rule, but it can really help with code manageability.) The following syntax example implements this extended class method syntax.

```
[Attributes] function MethodName([ParmName:ParmType, ...]) : ReturnType {
...Code Block...
}
```

You can specify any number of parameters to functions, and you specify them between the parentheses that define the parameter list. You type each parameter by giving it a name, followed by a colon, and then the type. This is identical to the variable declaration syntax, except that it does not require visibility attributes or the `var` keyword. The following example demonstrates the definition of some very simple class methods:

```
class Sample {
    function A() : void {
    }
    function B(name:String) : void {
    }
    function C(firstName:String, lastName:String) : void {
    }
}
```

Three things appear in the code block: local variable declarations, programming statements and expressions, and the `return` statement.

Any variables declared at the method level become local to that method and override any variables declared at the class level that have the same name, unless you explicitly call for them by using the `this` keyword. The following is an example of a method with a member implementation:

```
import System;

class Sample {
    private var i:int = 1;

    function Me() : void {
        var i:int = 0;

        Console.WriteLine("Local Variable");
        Console.WriteLine(i);
        Console.WriteLine("Class Instance Variable");
        Console.WriteLine(this.i);
    }
}
```

```
var o:Sample = new Sample();
o.Me();
```

```
Local Variable
0
Class Instance Variable
1
```

The code block for the method declaration can contain any of the conditional statements, loops, and other syntax discussed throughout this book. In the previous example, you can see that the `Console.WriteLine` function calls are normal statements. These types of statements comprise the majority of the function block.

The `return` statement is used to exit the function and return control to the calling code. The `return` statement can return nothing, in which case you follow it with a semicolon. Or it can return some return value of the same type as the return value declared for the function. So far in this chapter, all our functions have been of type `void`, so this hasn't been an issue. The following code snippet makes use of the `return` statement for exiting the function early and returning a value:

```
import System;

class Sample {
    function ReturnBeforeDone() : void {
        Console.WriteLine("Beginning");
        return;
        Console.WriteLine("I'm never called");
    }

    function ReturnString() : String {
        return "Hello World";
    }
}
```

For any `void return` function, JScript assumes a `return` statement at the end of the code block, so you don't have to explicitly include one. This is a nice thing the compiler does to limit the amount of code you have to write.

Rules for Overloading Methods

Method overloading is likely to be your best friend when you program in JScript .NET. It allows you to use different sets of parameters for the same function and perform either the same action or different actions, based on those variables. You need to keep in mind two considerations when overloading a function: You must obey the rules of method overloading and you need to understand the rules JScript uses for overload resolution.

Method overloading has two rules:

- Two functions cannot have the same prototype definition. The functions must differ in number of parameters, type of parameters, and/or return value types.
- Two functions cannot have the same parameter list, differing only in return type.

Let's examine a couple valid overloaded functions. These functions all perform the same actions, just on different sets of data. The following example includes an `Add` function that adds a string to an existing string; the `Add` function is overloaded to take an integer, a string, and an array of strings:

```
class Sample {
    private var internalString:String;

    public function Add(i:int) : void {
        internalString += i.ToString();
    }
    public function Add(s:String) : void {
        internalString += s;
    }
    public function Add(s:String[]) : void {
        internalString += String.Join(" ", s);
    }
}
```

This code performs perfectly well if the user calls the `Add` function with any of the available parameter overloads. As long as you make sure that the parameter list is different for each function, the overloaded functions won't throw any compiler errors. You need to watch for the second rule of method overloading, though. The following code sample has a `ValidSample` class that shows how you can change the return types of a function, and a `InvalidSample` class that is an example of a compiler error because you change the return type but still have the same parameter list:

```
class ValidSample {
    var i:int;
    var s:String;

    public function Add(i:int) : int {
        return (this.i += i);
    }
    public function Add(s:String) : String {
        return (this.s += s);
    }
}
```

```
class InvalidSample {
    public function Add(s:String) : int {
        return 1;
    }
    public function Add(s:String) : String {
        return s;
    }
}
```

Specifying a Variable-Length Parameter

The parameters of a method can take a special form. If the final parameter type is a single-dimensional array of any type, you can declare the final parameter as a *variable-arguments parameter*. This means that rather than having to explicitly create an array of items, you can call the method with a list of items of the same type as the array.

To denote that the final parameter is a variable-arguments parameter, you precede the parameter declaration with an ellipsis (`...`), which tells the compiler to add a special tag, the `params` attribute, in the IL. The following function takes three initial parameters and any number of additional parameters, provided that they are of the type of the variable-arguments parameter:

```
function VarArgsFunction(first:String, middle:String, last:String,
➥ ...phone:String[]) {
}
```

You can call this special function by using several different declarations. You can specify an array of strings as the fourth parameter, you can specify a single string as the fourth parameter, or you can specify a fourth and more parameters of type `string`, and they will all be concatenated into the array. You can even call it with no fourth parameter, in which case the array is zero length. Variable-length parameters are an advanced concept that is supported by several of the .NET languages, including C# and VB .NET. Some examples of variable-argument parameters in the CLR are the `String.Format` function and the `Console.WriteLine` function. The following is an example of using methods to call a function with a variable-length parameter:

```
VarArgsFunction("Justin", "E", "Rogers", "555 555-5555");
VarArgsFunction("Justin", "E", "Rogers",
    String[](["111 111-1111", "555 555-5555"]));
VarArgsFunction("Justin", "E", "Rogers",
    "111 111-1111", "555 555-5555", "777 777-7777");
VarArgsFunction("Justin", "E", "Rogers");
```

Method overloading appears rather simple on the surface, and it really is, considering that you only have two basic rules to remember. Note that the only factors in function overloading are the parameter list and the return type; you can't affect method overloading by using visibility attributes. The one exception is instance versus static methods. An *instance method* is any method that doesn't have the `static` attribute applied. *Static methods* are those methods that are shared by all instances of a class, whereas *instance methods* have access to private member data that isn't shared between instances of the same class. You could make the `InvalidSample` class above work correctly by declaring one of the methods `static`.

Static methods and instance methods are entirely different from one another when it comes to determining method overload validity at compile time because they are accessed in two different manners. An instance method requires that an instance of the class be used, whereas you can call a static method without ever creating an instance of the object, by simply using the type name.

Using Static Methods

Programmers often need functions that don't work with the internal state of the class on which they are defined. These functions would return the same state, no matter which object they were defined on. In a static function, all the data needed to complete the function must exist solely within the parameters or within the static properties and fields of the class or other classes. In this form, a static function is identical to a global JScript function. However, the class name acts as a namespace because you must always qualify the static function with the name of the class in which it resides. The following example demonstrates the use of a static function that can be accessed without creating an instance of the class:

```
class StaticFunctions {
    public static function Print(str:String) : void {
        Console.WriteLine(str);
    }
}

StaticFunctions.Print("Hello World");
```

You can also use static functions to return predefined versions of a given class. Remember that in Chapter 6 that we talked about making a private constructor so that an object can't be instantiated. When you do so, you can still call the private constructor and return an instance of the object from a static function. Examples of this include functions that generate some class based on another similar class, such as the following:

```
class IntegerSquare {
    private IntegerSquare(x:int, y:int, height:int, width:int) {
    }
    public function FromArray(square:int[]) : IntegerSquare {
        return new IntegerSquare(square[0], square[1], square[2], square[3]);
    }
}
```

This example demonstrates that a static function can be used to generate an otherwise uninstantiable class. Normally, when invalid parameters are passed to a constructor, an exception is thrown. In the case of a static function, you can choose to return `null` rather than throw an exception. This form of programming sometimes means performance gains if you plan on creating a large number of invalid objects. Static members often come into play when you're working with properties and fields, which are discussed in the following section.

Properties and Fields

You use properties and fields to get data from a class or set new pieces of data. Generally, you should set up several properties and fields within a class that gives access to any class data in a managed way. After all, some data should be kept private, some data might have to be presented in a special manner, and still other data might have to be initialized before the user is given access.

Fields are the easier choice for allowing access to internal variables. When a class variable is defined as public, it automatically becomes a field, and when this happens, a user can get and set the value of that variable without any access control.

Properties are more complicated than fields, but you can control whether a property can be read from or written to. With a property you can also provide some data abstraction. The user isn't directly setting the variable, you can test any conditions within the property before you set the member variable to the new value, or you can make sure the property contains a valid value before passing it back to the user.

The discussion in the following subsections first focuses on fields as public and protected member variables because these are the most easily employed options for data availability in a class.

Fields for Simple Variable Access

You create fields whenever you declare a variable within a class definition. Fields can either be private fields accessed only by a class, public fields that can be accessed by anyone using the class, or protected fields that can be accessed only by the class or from its derived classes.

The most often-used type of field is the public field. You can set and retrieve public fields from any code that makes use of the class. Public fields offer no access control and are the easiest way for you to have users interact with member variables in a class. In code, they look just like any of the variable declarations you've seen before in this book, only they must appear within the class definition code block and outside any constructors, functions, or property definitions, as in the following example:

```
class ClassWithPublicFields {
    public var PublicField:int = 5;
}

var o:ClassWithPublicFields = new ClassWithPublicFields();
o.PublicField = 20;
Console.WriteLine(o.PublicField);
```

```
20
```

This example clearly demonstrates the declaration and use of a public field. It also shows that a field can be initialized to a value when the class instance is initialized. This can come in handy for providing default values for any variables you don't want the user to explicitly set.

You can also use protected fields when you want to limit access to only a class or its derived classes. In this manner, you can provide some data to any subclasses that you might not want to make available to the normal user of the class. By marking it `protected`, you can also ensure that future derived classes can never change the access type to `public`.

You should use private fields only when you need to hide some data from the user. Most often, `public` and `protected` properties on an object use private field members to hold any data that they might need. You will see examples of using private field members to hold data for properties later in this chapter, in the section "Properties for Versioning."

Another special type of field is the assemblywide, or internal, field, which you use when you want the field to be visible only within the current assembly. This is a great way to share the variable between different parts of code within a library while at the same time preventing that data from being available to programs that use the assembly. To further expand on the internal field, you can also specify the `protected` visibility modifier to make the field available within the assembly and to any derived classes. The following example demonstrates how classes can define access fields of varying levels of visibility:

```
import System;

class ClassWithManyFields {
    public var PublicField:int = 5;
    protected var ProtectedField:int = 6;
    internal var InternalField:int = 7;
    protected internal var ProtectedInternalField:int = 8;
    private var PrivateField:int = 9;
    public function PrintVars() : void {
        Console.WriteLine(this.PublicField);
        Console.WriteLine(this.ProtectedField);
        Console.WriteLine(this.InternalField);
        Console.WriteLine(this.ProtectedInternalField);
        Console.WriteLine(this.PrivateField);
    }
}

class ClassOverriding extends ClassWithManyFields {
    public function PrintVars() : void {
        Console.WriteLine(this.PublicField);
        Console.WriteLine(this.ProtectedField);
        Console.WriteLine(this.InternalField);
        Console.WriteLine(this.ProtectedInternalField);
    }
}

var o:ClassWithManyFields = new ClassWithManyFields();
var o2:ClassOverriding = new ClassOverriding();

Console.WriteLine("Fields visible from within the defining class");
o.PrintVars();

Console.WriteLine("Fields visible from a class in the same package");
Console.WriteLine(o.PublicField);
Console.WriteLine(o.InternalField);
Console.WriteLine(o.ProtectedInternalField);

Console.WriteLine("Fields visible from a derived class in the same package");
o2.PrintVars();
```

```
Fields visible from within the defining class
5
6
7
8
9
Fields visible from a class in the same package
5
7
8
Fields visible from a derived class in the same package
5
6
7
8
```

All the access modifiers in this example protect or limit access to data so that the class can operate in a well-defined manner rather than allowing random access to all of the fields. Fields are only a temporary solution, though. When you really want to control access to a variable, you should use properties rather than fields because you can write custom logic within a property function, whereas you can't write such custom logic into a field.

All the field definitions discussed so far implement fields at the instance level of a class. That is, you have to have an instance of the class to get at the data, and this means each class has its own copy of the data. If you want to share some data between all the classes so only one instance of the variable exists, you need to use the `static` attribute in the field definition. This allows only a single version of the data, no matter how instances of the class are made; regardless of how many times the data is changed, all class instances see the same value. Let's rewrite the previous code sample to demonstrate how it could be changed to use static fields instead of instance fields. This example does not provide the output because it would be the same as the output for the previous code sample:

```
import System;

class ClassWithManyFields {
    static public var PublicField:int = 5;
    static protected var ProtectedField:int = 6;
    static internal var InternalField:int = 7;
    static protected internal var ProtectedInternalField:int = 8;
    static private var PrivateField:int = 9;

    public function PrintVars() : void {
        Console.WriteLine(ClassWithManyFields.PublicField);
        Console.WriteLine(ClassWithManyFields.ProtectedField);
        Console.WriteLine(ClassWithManyFields.InternalField);
        Console.WriteLine(ClassWithManyFields.ProtectedInternalField);
        Console.WriteLine(ClassWithManyFields.PrivateField);
    }
}

class ClassOverriding extends ClassWithManyFields {
    public function PrintVars() : void {
        Console.WriteLine(ClassOverriding.PublicField);
        Console.WriteLine(ClassOverriding.ProtectedField);
        Console.WriteLine(ClassOverriding.InternalField);
        Console.WriteLine(ClassOverriding.ProtectedInternalField);
    }
}

var o:ClassWithManyFields = new ClassWithManyFields();
var o2:ClassOverriding = new ClassOverriding();
```

```
Console.WriteLine("Fields visible from within the defining class");
o.PrintVars();

Console.WriteLine("Fields visible from a class in the same package");
Console.WriteLine(ClassWithManyFields.PublicField);
Console.WriteLine(ClassWithManyFields.InternalField);
Console.WriteLine(ClassWithManyFields.ProtectedInternalField);

Console.WriteLine("Fields visible from a derived class in the same package");
o2.PrintVars();
```

Properties for Versioning

The common argument to rationalize the overhead of properties compared to the simplicity of fields is that you can version properties. *Versioning* means you can change the data the property accesses as many times as you see fit, but the property can always maintain the same prototype. For instance, you can always return a `String` object for someone's name, but you could get that string first from a custom object, and then maybe from a dataset later in the program's development, and possibly even from a Web service in the end. No matter what the backend data becomes, the friendly face of the property remains the same.

This is possible because a property is implemented as a function or set of functions with specific signatures. We'll use the `Name` property in the following example, which implements a property `get` function, which defines the commands to be run whenever someone tries to access the `Name` property (it can be as complex as returning something from a dataset or as simple as returning a static string):

```
class Simple {
    // Simple
    public function get Name() : String {
        return "Justin Rogers";
    }
}

class Complex {
    // Complex
    public function get Name() : String {
        Dataset ds = MyClass.ReturnDataSet();
        return ds.Tables["Names"].Rows[0]["Name"];
    }
}
```

No matter what goes into the function, the prototype always remains the same. The important thing to note about the prototype is that it always takes zero arguments and returns some type. The type becomes the type for the property and also defines what the type of the property `set` function is. The property `set` for the `Name` property can be simple or complex, and the following is an example of a simple one:

```
class Simple {
    // Simple
    private var _privateName:String;
    public function set Name(value:String) {
        _privateName = value;
    }
}
```

Notice that this example includes a private member variable. This is an important concept because generally you need to store values from properties somewhere, and the best place for this is in a typed private field.

Just as with methods and fields, properties can be defined as static. Static properties have several uses. First, they can be used to hold instances of a class (as can static fields, but remember that static properties can be versioned and dynamically initialized). The `System.Drawing.Color` object is a series of static properties, each of which is a different color. If you request the `Color.Beige` property, you get back a `Color` object with the `rgb` value for beige in the private instance data. If you and someone else both request `Beige`, you both get the same `Color` object, and objects for the same color are never duplicated.

Note

An interesting option for a static property is to implement a `Current` property. You can do this is in many places in the CLR, and you can do it to reference a single instance of the current class. Any classes that use this class know that it supports a `Current` property and shares the object instance. This is often useful for high-availability objects that need to be shared by many classes.

In some cases, all classes need to share the same instance, but for different reasons. For instance, they might use the object to pass messages back and forth, like a managed queue. In this case, you need to make the `Current` property read-only because you don't want rogue classes overwriting the object or setting it to `null`. This would be a good time to use a static constructor to initialize the `Current` property when the class is loaded for the first time.

You can also use static properties to initialize data on demand. Any resource that requires a long computation to retrieve something you might need to use again can easily be computed on demand and cached, using a private static field and a public static property. This is shown in the following example, which caches a value from a file:

```
class StaticProperties {
    private var _staticInfo:String = null;

    public function get Info() : String {
        if ( _staticInfo == null ) {
            // If this is the first time we need to get the info from file
            StreamReader sr = new StreamReader(path);
            _staticInfo = sr.ReadToEnd();
            sr.Close();
        }
        return _staticInfo; // If info wasn't null we'd do this immediately.
    }
}
```

This example demonstrates the power of a static property `get` function. The same principles apply to the static property `set` functions. Every visibility attribute, along with all the strange behavior that exists for a field, applies identically to property functions. (If you need a refresher on the various attributes for properties, see the section "Class and Member Attributes" in Chapter 6.

As mentioned earlier in the chapter, properties can control access to variables. One way of doing this is by limiting a property to only read or write status. If you specify a property `get` function without a corresponding property `set` function, the property is read-only. If you specify only the property `set` function without the corresponding property `get` function, the property is write-only.

Event Support

Support for events in JScript .NET is extremely limited. The general rule is that JScript can consume events by declaring event handlers, which are functions that have a specific set of parameters that are defined by the class or object that raises the event, and it might also inherit the events that are declared in a base class that it inherits. JScript .NET can't declare new event types or *delegates* (that is, functions that define an event), and it has no mechanism for including existing delegates in a class. What does this mean in terms of event support? First, JScript can consume events. This means you can hook up an event handler to the events of another object and receive notifications in the form of the event handler being called.

Second, you can derive a type from some base class that was written in another .NET language and therefore inherit its events. You can't modify these events in any way or add new events, but you can use the existing events. For example, you could derive from a button class that has a click event. The derived object then has a click event, and at the same time, you can write custom painting functionality or whatever other code you might need to customize the class.

In the following subsections, we'll examine event handlers and how you hook into events. Then, we'll look at a sample of a subclassed control where events are inherited. Finally, we'll discuss delegates—why JScript .NET doesn't support them now and how it might support them in the future.

Consuming Events

Classes export events for many different purposes. Whether the purpose is for asynchronous operation of some algorithm or to notify the programmer of some special condition within the class, there is always a degree of special work that must be performed for the event to be consumed. A class can fire an event all day long, but unless another class or function consumes that event, no action ever takes place.

Consuming an event requires two steps:

1. Write a function that will become the event handler. Whenever the event is fired, the function is called, and you can do some work based on the event in question. The function can also contain some parameters because each event exported by a class might optionally define that it passes back some extra data to the callback function.
2. Call the `add_`*`EventName`* function on the class that hosts the event that the program needs to catch.

The most basic event handler is used to consume the `EventHandler` delegate that is defined by the CLR. This event delegate is used whenever the event has no data to pass, and it accepts two parameters. The first parameter is an object variable that identifies the source of the event. Often this is `null` and the event does not contain any meaningful information. The second parameter is an `EventArgs` class. `EventArgs` doesn't contain any fields or properties, and thus it can't be used to pass any information to the event handler. Setting up an event handler is easy, as long as you know the types of the parameters for the event delegate. The following example creates an event handler that might be used to trap an event of type `EventHandler`:

```
function MyEventHandler(sender:Object, e:EventArgs) : void {
}
```

TIP

Some programmers consider it extremely important to use a special naming scheme with event handlers so that they aren't confused with normal functions. The easiest way to do this is to use the name of the class or variable to which you're going to hook the handler, followed by an underscore and then the name of the event to consume. So the name for a function handling the click of a button named `Button1` might be `Button1_Click`.

Many events are declared by using the simple event delegate EventHandler because they don't need to pass any data to the consuming class; they only need to notify the class that something has happened. When data does need to be passed, the second parameter is changed from an `EventArgs` class to a class that inherits from `EventArgs`. Additional properties or fields can be declared on this new class, which means the event handler prototype would have to change to support the new class, and possibly the code inside the event handler would have to change to examine the new data provided. The following example demonstrates the event handler needed to consume a more complex event that passes back a custom data structure:

```
function SampleButtonHandler(sender:Object, clickArgs:ClickEventArgs) : void {
    var x:int = clickArgs.X;
    var y:int = clickArgs.Y;
}
```

When the event handler is defined, you still need to hook up a handler to the event in question. In JScript .NET, you do this by calling two special functions (`add_`*`EventName`* and `remove_`*`EventName`*) that are generated whenever a class implements an event.

The first function is the `add_`*`EventName`* (where *`EventName`* is the name of the function that will be the event handler) function. To hook an event, you call this function. The following example demonstrates adding an event handler to an event:

```
function MyEventHandler(sender:Object, e:EventArgs) : void {
}

var myClass:WorkerClass = new WorkerClass();
WorkerClass.add_WorkDone(MyEventHandler);
WorkerClass.BeginWorking();
```

This example includes a special event handler for the `WorkDone` event. When the `WorkerClass`, `myClass`, is done working, it notifies the program, and you can either grab some information out of the event arguments or simply examine the data exported by the class that is exporting the event. This example assumes that `BeginWorking()` starts working asynchronously by spawning another thread and that the program can continue to run while the `WorkerClass`, `myClass`, is coming up with a response.

The second function allows you to remove event handlers from an event. This can be useful if you need to process the event only once or twice and then you wouldn't care about it in the future. For this purpose, you use the `remove_`*`EventName`* function (where *`EventName`* is the name of the function that will be removed), which removes the event handler from the event so that it cannot be called. The following example demonstrates when removing event handlers can be beneficial. In this example, an event handler processes a timer callback, and after the program receives five callbacks, it removes the event handler from the event:

```
var timesCalled:int = 0;
var timer1:Timer = new Timer();
timer1.Interval = 1000; // 1 second
timer1.Enabled = true;

function MyEventHandler(sender:Object, e:EventArgs) : void {
    timesCalled++;
    if ( timesCalled == 5 ) {
        timer1.remove_Elapsed(MyEventHandler);
    }
}
```

Event-oriented programming can be extremely powerful. Even with the limited support of only consuming events, you can take advantage of all the controls and classes that utilize events for state notifications and asynchronous execution.

Subclassing Controls for Events

One method for gaining access to events in JScript .NET is to subclass classes and controls that already contain events. This either means subclassing an existing class or learning another CLR-compliant language to build a base class with the predefined events. Either way, subclassing is the only option if you really want to support events in your JScript .NET program.

TIP

If you decide to make your own classes in another language, I recommend C# because it is very similar to the JScript .NET language in syntax and structure. Declaring events in C# is as simple as declaring the delegate type, creating an event variable of that delegate type, and implementing a function that allows you to fire the event. This is generally in the form `On`*`EventName`* (where *`EventName`* is the name of the event the class has defined) and takes the same parameters as the event delegate type.

An example of a control that exports events is the `System.Windows.Forms.Control` class. This class is the base class for any windowed components you see or use in the Windows Forms application programming interface. By subclassing the `Control` class, you gain access to all of the various events that the `Control` class defines, including many mouse events, keyboard events, events for drag-and-drop support, and many other events that correspond to property changes in the control.

Deriving a class from the `Control` class is perfect if you are trying to make a windowed component. The following example shows how to use the Control class as the base class for inheriting events:

```
class CustomWindowedControl extends System.Windows.Forms.Control {
   // That is all there is to it.  We now support events.
}

function o_Click(sender:Object, e:EventArgs) : void {
    // Do some work.
}

var o:CustomWindowedControl = new CustomWindowedControl();
o.add_Click(o_Click);
```

As you can see in the example, the events are simply inherited, so as soon as you get an instance of the new control, you can use all of the base control's events.

JScript .NET and Delegates

JScript .NET doesn't currently support declaring new delegate types or creating the special event variables required by the CLR to support events. The European Computer Manufacturers' Association will have to review and make a decision about this feature at some later point.

So, can you declare events by using existing delegate types within JScript .NET? The short answer is no because the feature is on a list for future consideration. The long answer is that it is possible to support only events that are within the constructs of the JScript .NET language, but this is incompatible with the CLR in several areas.

First, every event has to declare two special methods—`add_`*`EventName`* and `remove_`*`EventName`* (where *`EventName`* is the name of the event being implemented)—which you've seen before. A special attribute is placed on these functions in some compilers, namely C# and VB .NET, when they declare an event that tells the CLR that these are a special type of method and not just a normal method. The JScript .NET compiler does not place this special attribute on the methods, and thus C# and VB .NET do not recognize them as event methods but as normal methods.

Second, you have to be capable of adding delegates to events, and this is not possible in JScript .NET using normal methods. The problem is that JScript .NET is unable to cast a `System.Delegate` type to the delegate type for the defined event. This case is necessary for delegates to be added to an event.

Event support can therefore be considered unimplemented in JScript .NET at this time. However, the next version of the JScript .NET language and compiler will most likely have support for events that will be CLR and ECMA compliant.

Summary

The CLR can impose many restrictions on code so that it can support many disparate languages and syntaxes. You have learned that interfaces are a special contract between a class and other classes that use it. A class can support a series of methods and properties, and this makes up for the lack of multiple inheritance in the CLR, and some say it makes for a cleaner object inheritance tree than multiple inheritance could offer.

In this chapter you have learned how to define interfaces for programs and how to specify the various interface members, both at the implicit level (that is, by allowing the compiler to decide the interface mappings) and at the explicit level (that is, by telling the compiler which function implementation is mapped to each of the interfaces that define the function).

In this chapter you have learned about class methods, fields, and properties, as well as the lack of event support in JScript .NET.

In Chapter 8, “Exception Handling,” you’ll learn all about exception handling, including the `try...catch...finally` block statements and creating your own exceptions. Exceptions are generated by every program, and often exceptions are used as an alternative to a returning an error value from a function. It is important that you learn how to deal with exceptions thrown in code, how to implement the proper cleanup blocks, and also when and how to throw your own exceptions.

CHAPTER

8

Exception Handling

IN THIS CHAPTER

By now you've learned everything it takes to create an actual program using JScript .NET. Next, its time to learn exactly what you can do in your code to make sure you catch all the possible errors and various conditions that might cause a program to end unexpectedly. *Exception handling* is a series of special code instructions that allows one piece of code to raise an error condition that must be handled by another piece of code. This error condition is called an *exception* because it is outside the normal operating boundaries of the code.

One of the main reasons exception handling is important in programming with the Microsoft .NET framework is that many of the libraries, which you'll be using in future chapters, use exceptions to demonstrate error conditions. Traditionally, programs use return codes to indicate success or error conditions, but if a function is already returning something of importance, then there isn't room for a return code. Another issue with return codes is they are extremely hard to maintain. A program might work through several different libraries, and all the return code values might overlap. You might even want to pass a return code you just received while running a function to the caller of the function. But what happens if the return codes don't match correctly (for example, the return code from the library means one thing, but that same return code for the function means something completely different)? This can lead to code that is hard to understand at first glance.

Exception handling overcomes most of these hurdles. Any function or library call can throw any number of exceptions that you can later catch. Each exception must have at its root a common base exception, so if you need to trap all exceptions, you can do so by simply trapping the base exception. Because the exceptions are classes themselves, they are shared between libraries and programs. So if you receive an exception that your library isn't set to handle, you can pass that exception down to the calling program untouched. There isn't any reason to worry about differences in return code values or mapping a new unknown exception to an exception that your program normally emits.

The benefits and merits of using the exception handling and error system of the common language runtime (CLR) definitely outweigh the time it takes to add the lines of code that are needed to use the system. Throughout this chapter, we'll discuss these merits as well as pitfalls of the system, as we discuss the `try...catch...finally` statement and managing exceptions.

This chapter discusses throwing exceptions from code to exhibit error conditions in a program. Whether to rethrow a caught exception is a rather tough decision to make in a program, but this chapter will help you think about times when rethrowing an exception is beneficial. It also talks about where you should rethrow the exception from because there are a couple locations in new exception blocks that might make sense, but not all of them are the right place from which to throw or rethrow exceptions.

We'll wrap up this chapter by talking about the `finally` block in more detail because it's truly one of the most beneficial constructs to use, especially for purposes of resource management.

Wrapping Code with `try` and `catch`

Exceptions are always enabled while programs are running. Any exceptions that aren't caught by a program either result in a dialog box alerting the user about the error or the program quietly dying. You can wrap the `try...catch...finally` statement around any code that might throw an exception to make sure you handle these errors and perform some appropriate action so the user never has to detect that the program isn't working correctly. You can use the `try...catch...finally` statement for a number of other purposes as well.

The Purpose of `try...catch...finally` Blocks

`try...catch...finally` blocks are generally used so that a program can detect error conditions that are thrown from library code that is executing. Library code is generally a black box: You simply don't know what is going on inside the code, but more often than not, you know that the code you're calling throws certain exceptions based on error conditions. These error conditions, for example, could state that parameters are invalid. They could also indicate that some resource required by the library function, such as a database or network connection, was unavailable.

Another purpose of the `try...catch...finally` blocks is to allow the system to alert the program about resource issues. Many library functions provided by the CLR acquire system resources for the program (for example, database connections, file handles, and memory). Exception blocks are extremely crucial when working with system resources because they enable a program to detect the absence of some resource the program needs without crashing.

Users often create exceptions in code as they work with a program, insert bogus or inaccurate data, or use the program in a manner for which it was not programmed. Most programs allow for some user input, whether through the console, through a graphical user interface (GUI), or via input from a file or network stream. Users are notorious for providing bad data, such as an invalid number, one value when the program is looking for three, or data that for some other reason can't be parsed by the program. Exception handling is crucial here because many data types throw cast exceptions, format exceptions, or null reference exceptions if the data isn't in the expected format. This class of error is extremely difficult to handle appropriately, and it is a prime candidate for a good error reporting and logging structure, which we'll talk about later in this section.

Probably the best reason to use exception handling is that it provides benefits in code. It enables you to write small function prototypes because you don't have to take into account return codes. It allows you to raise a myriad of error conditions, which you can later catch in your code or pass down to user code (if you're programming a library).

Exceptions can be extremely detailed because they are objects with properties and methods. Some exceptions might give the name of the parameter that was invalid or provide a human-readable description, which can be invaluable in deciphering library exceptions. All exceptions show a stack trace so you can determine exactly where an exception occurred in code. This feature alone is reason enough to use exception handling rather than a system based on function return codes.

NOTE

A *stack trace* is a current list of called methods. This stack starts at the currently executing functions and goes backward through all the functions that called this function, until it reaches the bottom of the stack. If the functions have debug information enabled, the stack trace also includes filenames and line numbers that indicate from where the functions are being called.

The Structure of `try...catch...finally` Blocks

`try...catch...finally` blocks have a very simple layout. The simplest form is the `try` block with a single general `catch` block. The `try` block simply wraps code like any of the other statements (such as the `for` and `while` blocks) we've seen thus far in the book. Why might you want to use `try` blocks without any of the associated specific `catch` or `finally` blocks? Well, it comes in handy when you're first working through a program because it enables you to block anything that might throw an exception. Later, you can look for any `try` blocks you've created and add the `catch` and `finally` blocks as necessary. Error handling is an integral part of programming, but you generally don't want to code both error handling constructs and the program itself at the same time. It is often best to complete functions first and add error-handling code last. The following code block demonstrates a very simply `try...catch` block:

```
try {
    // Because we are casting from String to int, we might throw
    // a cast exception
    var UserID:int = Int32.Parse("HelloWorld");
} catch(e:Exception) {
    // Come back later and add more handling.
}
```

You can nest a `try` block in order to isolate a certain piece of code inside another `try` block; in fact, you can nest `try` blocks to any depth. For example, you can use a `try...catch` block for an entire function that logs exceptions to a file or to an event log. Then each location in code where you want to specifically handle an error, you place a nested `try` block so that the program can perform the appropriate action. The following is an example of how to do this:

```
import System;

try {
    try {
        var UserID:int = Int32.Parse("HelloWorld");
    } catch(e:Exception) {
        UserID = 0; // Supply a default value
    }

    // Use the ID somewhere down here
} catch(e:Exception) {
    // Place code to write out exception here
    // This is our fallback code, not our REAL error code
}
```

You need to be careful when writing `try` blocks to ensure that all the code paths make sense. Whenever an exception is thrown, the most explicit `catch` clause for the current `try` block is called. If a `catch` clause is not found, the exception-handling mechanism searches through any `try` blocks that contain the current `try` block until it finds a match or is caught by the system. In this manner, `try` blocks cascade all the way back to the beginning of a program and beyond if you don't handle the type of exception that is being thrown.

If you wrap an entire function in one `try` block, and the first instruction throws an exception, the `catch` block is executed, and finally, any code at the end of the `catch` block is executed as well. If the previous example occurred in real code, the final statement after the `catch` block would probably be a return statement. However, this return statement would have nothing to return.

`Try` blocks are not re-entrant (that is, you cannot modify the conditions that caused the exception and try again). So if you need to use error handling to recover from error conditions, you have to use nested `try` blocks appropriately. This can sometimes mean using `try` blocks around several statements in a row if they are important enough that you want to recover and still return valid results if one of them should fail.

Where to use `try` blocks in code is a design decision that you have to make based on each application. Next we'll talk about how to specify more explicit `catch` statements that can help you make these decisions.

Specifying the Right `catch` Statement

You sometimes need to catch certain exceptions and let the rest fall through to your general exception handler. This might be the case, for example, when you're dealing with user input and you only care about parsing and formatting exceptions, but the actual statement might return a strange general exception or an `OutOfMemoryException` object. To make a `catch` statement more general, you simply change the type of the exception variable to something more appropriate, as in the following example:

```
try {
    var UserID:int = Int32.Parse("HelloWorld");
} catch(e:FormatException) {
    // Here we only catch format exceptions.
}
```

So what happens if something other than a format exception is thrown? Maybe for some reason the system runs out of memory or the `Int32.Parse` function performs some sort of security violation (which is highly unlikely, but let's talk about it for argument's sake). In this case, you simply specify the more generic handler around the `try...catch` block and print the exception information out to a log file. In this manner, you can write a program that processes all foreseen errors but still logs any unforeseen errors so that you can review them later and maybe institute additional error handling. The following example demonstrates using two exception handlers, the first of which is very specific and the second of which is a generic handler to catch unforeseen error conditions:

```
import System;

try {
    try {
        var UserID:int = Int32.Parse("HelloWorld");
    } catch(e:FormatException) {
        UserID = 0; // Supply a default value ONLY for FormatExceptions
    }

    // Use the ID somewhere down here
} catch(e:Exception) {
    // Place code to write out Exception here
    // This is our fallback code not our REAL error code
}
```

Any exception continues to cascade down a series of `try` blocks until it finds a `catch` clause that can handle it. If the exception reaches the system without finding a `catch` block for the exception, the program either terminates or gives the user an exception dialog box—and you certainly don't want that.

Another form of the `catch` block allows you to specify more than one `catch` block so you can intercept more than one explicit exception while still not catching any unknown or general exceptions, as we'll discuss in the next section.

Using Multiple `catch` Statements

Using multiple `catch` statements can be extremely useful in performing a variety of error-handling functions for known errors while still not catching unknown or general errors. This is best demonstrated when working with variables that refer to objects. For example, at some point in any program you are likely to get a `null` object, and code can quickly become littered with checks to see if a value is `null`. Depending on the application, using a single `if` statement every time you need to check a variable can make or break the performance of the function (believe me, sometimes you need every instruction to count). If you know that a value passed to a function should never be `null`, you can get a performance gain by removing any parameter checks. The user may still pass `null`, so you should either implement a `try...catch` block to catch a `NullReferenceException` object or simply allow any exceptions to cascade down to the user. Even if an object is not `null`, it could throw some other exceptions that you might want to catch. Say, for instance, that an object also throws a `FormatException` error, as in the following example:

```
function FastAlgorithm(obj:ParmObject) : int {
    try {
        var strResult:String = obj.Method1();
        obj.Parse(strResult);
    } catch(nullException:NullReferenceException) {
        // Throw an invalid argument exception here
    } catch(formException:FormatException) {
        obj.Value = -1; // Set a default value
    }
}
```

This code has quite a few merits. Instead of the users seeing a `NullReferenceException` error in some code they don't own, they see a handy invalid argument exception. The invalid argument exception makes more sense to users and directly points out what they did wrong so that they can fix the problem immediately. You can even supply an error message that says the value can't be `null`.

If the object in the previous code example is not `null`, then its methods are called instead of a `NullReferenceException` object being thrown. When the program gets to the `Parse` method, if `strResult` is invalid for some reason, then the program can't parse it, and the user gets a `FormatException` error. Instead of throwing a special error for the user to catch, you can set a default value. Having multiple `catch` blocks allows you to catch both of the explicit exceptions shown in the preceding code example, and if a more general exception occurs, then it is up to the user to catch it.

Ensuring That Code Is Cleaned Up with `finally`

When exiting a logical code path and jumping into a `catch` block, sometimes objects are not cleaned up the way they should be or when they should be. There are sometimes things that need to happen whether an exception is thrown or not. You can't place code to do this in a `catch` block for two reasons. First, the `catch` block might never be called if the exception is explicit. Second, even if the `catch` block is general and would always be called on an exception, you still need that code to exit when there isn't an exception. In this situation, you use a special code block that logically extends whatever code is in the `try` block and also executes immediately following `catch` blocks, if and when they are called.

The format of the `finally` block, which is appended to the end of any existing `catch` blocks for the `try` statement, is the same as that of the `try` block:

```
try {
    // Some code
} catch(e:Exception) {
} finally {
}
```

`finally` blocks are powerful, but you have to be extremely careful with what you do inside a `finally` construct. `finally` blocks clean up code by closing any objects or resources that might have been allocated inside the `try` block before the `try` block was exited without the allocation statements being executed. You should always check any variables that you use and make sure you're not generating further exceptions from within the `finally` block.

`finally` blocks have the special behavior of being called recursively back through the `try` statement until the first `try` statement with a matching `catch` statement is executed. This can make an exception very costly if you have a series of explicit `try...catch` blocks, each with its own `finally` statement; the first general exception through causes all the `finally` blocks to execute recursively up the call stack until a matching `catch` statement is called. You need to make sure your program design takes this feature into consideration, or you'll experience extremely poor performance in code paths that are executed a lot.

Throwing Exceptions

Throwing exceptions is a lot more fun than catching them. But throwing exceptions is a major design decision for an application. Each possible exception that gets thrown from a function is another code path a user of the code might have to implement in the form of a `catch` statement later. Exceptions are also costly. The general rule is that a `try...catch` block is without cost until the instance an exception is actually thrown and the system has to kick in and start the process of searching for a `catch` block and executing the `finally` blocks.

Throwing an Exception from Normal Code

To throw an exception, you use the `throw` keyword, which can appear by itself with no exception inside a `catch` block, in which case it rethrows the exception that the `catch` block just caught. You can also specify the `throw` keyword along with an expression that evaluates to an exception object. In this case, a new exception is generated and thrown. The following example shows an exception being rethrown and a new exception being thrown:

```
try {
    var UserID:int = Int32.Parse("HelloWorld");
} catch(e:Exception) {
    throw;
}

throw new Exception();
```

Exceptions are easy to generate because they are objects that you define and/or create that are derived from the `System.Exception` base object. The CLR supports many types of exceptions for many common programming errors. You should try to find a suitable exception that is defined by the CLR before you begin implementing your own exceptions to throw (as discussed later in the chapter). This ensures that there is some continuity between implementing your code libraries and implementing the CLR code libraries. It also means less explicit `catch` blocks in code are needed to catch both the CLR exceptions and your exceptions.

Some of the most common exceptions are `NullReferenceException`, `InvalidCastException`, `ArgumentException`, and `ArgumentOutOfRangeException`. All these exception classes take a special message parameter in which you can stash human-readable error messages. If you are catching one exception and throwing your own new exception, as you did in the example of multiple `catch` clauses earlier in this chapter, you can also supply the original exception as part of the new exception. The following example shows how to catch one exception and then include it with a newly generated exception:

```
// Pass a message
throw new NullReferenceException("Your Object Was Null");

import System;

function Work(obj:Object) : void {
    try {
        obj.Method();
    } catch(nullExc:NullReferenceException) {
        // Pass a message, The name of the argument, and our old exception
        throw new ArgumentException("Your Worker Object Was Null",
            "Work Object",
            nullExc);
    }
}
```

NOTE

In the CLR documentation, look up the `System.Exceptionclass` and examine all its derived types. `System.SystemExceptionclass` is the root for most of the items you'd normally throw in user code.

Where to Rethrow Unhandled Exceptions

You are likely to trap exceptions that you don't handle, but simply log in your code. Library code should pass back to the user code any exceptions that it doesn't handle adequately. But the library code might still need to log the exception because both the user and library programmer are possible sources for the exception that is being thrown.

So where and when do you rethrow an exception? The answers aren't clear-cut; they depend on the design of the program. For the most part, you need to rethrow the existing exception unless you can come up with a valid reason for changing the exception to another type and rethrowing it as a new exception.

You should always throw a new exception if you've done something to modify the original situation that caused the original exception. Also, you should make sure to throw a new exception if you know what happened and have a message you can use to explain to the user what happened. Nothing is worse than an exception that gets to the user's screen and has no information about the real cause of the message (that is, it is nothing more than a stack trace).

You should also never throw an exception from within a `finally` block. First, you don't have access to the original exception from the `finally` block, so you don't know if an exception really occurred. `finally` blocks should always be as short and direct as possible. You should limit the `finally` block to just the code needed to clean up resources that can't be cleaned up by the garbage collection engine.

How to Make Your Own Exceptions

Making your own exceptions is probably the most unimportant thing you will ever do in a program. The CLR provides so many predefined exceptions that only in rare cases will you have to implement your own exceptions.

In this section we'll discuss a sample application that uses custom exceptions to identify various errors in the code. The program will be responsible for loading bitmaps from a user's disk and performing some filtering routines on them. What are some exceptions that you might throw for this set of operations? The first exception you generate will be a base exception so that you can have a generic exception used to catch any and all exceptions generated by the program. The following example demonstrates the creation of some custom exceptions:

```
import System;

class FilterException extends Exception {
    protected var file:String;

    // Remember that we have to supply constructors
    function FilterException() {
        super();
    }

    function FilterException(message:String) {
        super(message);
    }

    function FilterException(message:String, innerException:Exception) {
        super(message, innerException);
    }

    function FilterException(message:String, file:String,
        innerException:Exception) {
        super(message, innerException);

        this.file = file;
    }

    public function get File() : String {
        return this.file;
    }
}
```

This code sets up all the basic constructors. This isn't necessary; it just provides a variety of ways to make the `FilterException` class. The only required constructor is the `final` constructor, which lets you pass in the name of the file that is having a problem. The base classes specify only the `final` constructor and any others they might need to initialize new variables.

The next set of exceptions extends the base class. For instance, say the bitmap isn't large enough for the filter to be applied. The required size might be different for different filters, so you need to add a new property that accepts the name of the filter. You can also add a filter that identifies whether the image is in the appropriate format. This exception, unlike the other one, doesn't need any special variables. The following example demonstrates the creation of the sample exceptions we just discussed:

```
class FilterSizeOutOfRangeException extends FilterException {
    protected var filter:String;

    function FilterSizeOutOfRangeException(message:String, file:String,
        innerException:Exception) {
super(message, file, innerException);
    }

    function FilterSizeOutOfRangeException(message:String, file:String,
        filter:String, innerException:Exception) {
super(message, file, innerException);

        this.filter = filter;
    }

    public function get Filter() : String {
        return this.filter;
    }
}

class FilterInvalidSourceException extends FilterException {
    function FilterInvalidSourceException(message:String, file:String,
        innerException:Exception) {
super(message, file, innerException);
    }
}
```

You can implement code to make use of these new exceptions. By now you should have an idea of when these exceptions might be useful and some of the design decisions that have to be made.

NOTE

A program should almost always make a base exception just for that application, and the rest of the exceptions should extend that one. This way, you have the ability to trap all the exceptions your application throws into a single `catch` clause by using the base exception.

Summary

If you're familiar with exception handling in other languages, you'll notice that you handle errors slightly different in the .NET framework. You should now have a good knowledge of how to use the `try...catch` statements. You should also understand the full abilities of a `finally` clause to help you clean up your code.

When you start implementing libraries and functions of your own, you'll better understand when and how to throw exceptions. Originating the proper exceptions is crucial to good library design and for giving error information to users of your library. You should try to throw predefined exceptions whenever possible and generate new types of exceptions only when absolutely necessary.

The next chapter is a short compiler reference, in which we'll discuss all the various compiler options and modes available to compiled JScript programs. We'll also go over conditional compilation for special code generation and debugging.

JScript .NET Compiler Features

CHAPTER

9

IN THIS CHAPTER

One of the greatest features of the JScript .NET language is that instead of being interpreted at runtime like a scripting language, it is now compiled into an actual assembly and gains the extra speed of running as native code instructions on the processor. This greatly enhances the speed at which JScript .NET runs, introduces many new features at the compiler level, and introduces some limitations as a result of both being compiled and maintaining full support for ECMAScript and the CLR.

We'll start by examining various command-line features. If you recall, we compiled some basic programs using the compiler in Chapter 2, "Organization of JScript .NET." We've also been compiling all the code listings up to this chapter using a special makefile. However, unless you looked at the makefile closely, you probably don't know everything that is happening, and you certainly won't be familiar with the many additional options available to you at the compiler level.

We'll be covering the basic syntaxes for libraries, executables, and Windows executables, along with how to reference other assemblies and use their code and objects. We'll learn how to affect code generation through several options that allow varying levels of debug output, code profiling, and warning generation.

This will move us directly into the conditional compilation section. Here you'll discover how to define symbols on the command line that can be used by special statements in the source file to include or omit code for the current build. We'll also go over the rest of the directives supported by JScript .NET that allow for custom debug information. Finally, we'll discuss the special compiler symbols, `DEBUG` and `TRACE`, used by the CLR that also affect debug output and the use of several very important CLR classes.

Several assembly-level features require the use of an assembly-level attribute. Up to now we've placed attributes on everything except for the assembly itself. We'll go over the syntax for assembly-level attributes, as well as some of the more common attributes you can use to specify version information, version comments, and keys for strongly naming your assemblies.

Command-Line Compiler Usage

No matter what type of program you're going to be creating using JScript .NET, you'll need to know how to use the command-line compiler. This compiler is capable of creating normal console executables, Windows executables that spawn without a console, and libraries of code able to be referenced and used by other libraries and executable content. The compiler is also responsible for resolving any external references by investigating your referenced assemblies. During this link stage, the compiler ensures that all the types and methods you've used are valid and actually exist in one of the assemblies you've imported. This link stage provides for a good deal of error checking at compile time to reduce runtime errors for missing symbols.

In addition to the more common options, there are several code-generation compiler flags that affect the output from the compiler. These code-generation compiler flags can be used to specify character encoding output, set version safety options, and enable some enhanced code profiling. A couple of warning-related flags are used to specify warning levels during compilation and change the effect of a warning on the compilation step. We'll begin by examining executable code compiler flags.

Compiling Executable Code

More than likely you'll write more executable-style code assemblies than you will libraries of code. This is a generalization that most people write small utilities and code to be executed by end users, and a much smaller percentage actually write code that will be used by other programmers in the form of libraries. For this reason you'll most likely use the `target` option to specify executable content.

The `/target` option or `/t` option is used to specify which kind of assembly will be generated. In JScript .NET you have three options: exe, winexe, or library. We'll discuss two of the targets, the exe and the winexe, as well as the differences between them. I've chosen to talk about these two target options because of their use in creating executable content. The exe target specifies that the assembly will be executable within the host environment (it can be run from the command line, from the Windows Explorer shell, or through some other hosting means yet to be created). Additionally, it specifies that the program be launched with a console window. This is the DOS-style window you see when running a command prompt.

Console-style programs are extremely useful because they generally don't have any UI and any input or output is done through a standard set of streams. So far in this book, all the executable programs we have created have been console programs and we have used the `Console` class to write to the console. You can also use the `Console` class to read user input from the console, but generally console programs take their input from the command line rather than from actual user intervention.

Traditionally, JScript has been used to write automation scripts under Windows Scripting Host. Windows Scripting Host had two hosts, the cscript.exe, which allowed the script to run as a console application, and the wscript.exe, which allowed the program to run as a Windows executable. Generally, authors of these automation scripts would choose the cscript.exe host because they could provide user feedback to the console without having to generate some sort of UI. The following code demonstrates the use of the target option exe being used to generate some executable content:

```
jsc /target:exe Source.js
jsc /t:exe Source.js
```

Although the exe target option will be the most widely used for console and utility type applications, there is still a great need to launch Windows executables in JScript .NET as well. This is because of the Windows Forms API, which makes it extremely trivial to generate a program that has a simple or even a quite advanced UI. To create a Windows executable, the winexe target option is used instead of the exe target. The only difference between the exe and winexe target options is that the winexe target option suppresses the creation of a console window. This means that standard operations that print and read from the console won't work and that the `Console` class becomes somewhat useless. Instead of using the console, the program would have to take advantage of some of the Windows Forms UI features instead. The following code demonstrates the use of the winexe target option to create a windows executable that launches without an associated console:

```
jsc /target:winexe Source.js
jsc /t:winexe Source.js
```

The choice between a console application and a Windows executable is completely up to the programmer. Windows Forms applications with a full UI can be compiled with either flag and will still function normally. Often it comes in handy to catch console output while developing a Windows Forms application and then later disable this pseudo debug output by compiling with the winexe target to get rid of the console. I'll leave you with the decision as to which you prefer.

Compiling Library Code

Code libraries are excellent for sharing code between multiple projects or for shipping some useful set of classes to another development team. Previously in JScript you could always create COM libraries and controls, thus making use of a multitude of already-available code. Making a library in JScript .NET is somewhat similar.

Libraries are precompiled classes that are ready to be used by your program. We've seen some of these previously in our code listings as we've imported the System.dll, and behind the scenes the .NET core library, mscorlib.dll, has also been making an appearance. All the basic data types, the `Console` classes we've made so much use of, and the `Environment` object we've used to get command-line options—all of these are classes that are defined in these standard code libraries. It makes a lot of sense to bundle up any classes you find yourself using across many different projects into their own code library as well.

Making a code library is as simple as using the library target. When you specify the library target, a library is generated instead of an executable assembly. What this means is that any global code present in the library isn't executed, and no entry point is defined. Running the library from the command line or Windows Explorer won't have any effect as it did with the executable targets we've seen previously. The following code demonstrates the use of the library target option to create library content:

```
jsc /target:library Source.js
jsc /t:library Source.js
```

Creating libraries of code doesn't do you much good unless you know how to reference these new libraries in your programs. We'll look at the compiler reference options next.

NOTE

Often you create brand-new classes for your executable programs. It can be difficult to consolidate all of these special classes into their own library, so many times what I do is compile all the source files for each of my programs into the same library. Usually, this works and you simply get a huge assembly of classes. Any global code is still placed in the global object, but without a defined entry point it never executes and you can simply use the classes you defined for each project. Being able to compile executable source into library source and vice versa in JScript .NET is very powerful. Often you can't do the same in other languages without special compiler options.

Referencing Libraries

There are several compiler options—`reference`, `lib`, `nostdlib`, and `autoref`—that help you reference other libraries and assemblies from your programs. The most prominent is a direct library reference using the `/reference` or `/r` option. In this form you specify which library you want to include as a reference directly on the command line. It is the most specific and guarantees compilation of the source as long as the library can be found. The following code uses the `reference` compiler option to reference some code libraries:

```
jsc /t:exe /reference:System.dll Source.js
jsc /t:exe /r:System.dll Source.js
```

You can specify multiple references by using more than one `/r` option or by separating the assemblies by commas or semicolons:

```
jsc /t:exe /r:System.dll /r:System.Drawing.dll Source.js
jsc /t:exe /r:System.dll,System.Drawing.dll Source.js
jsc /t:exe /r:System.dll;System.Drawing.dll Source.js
```

The compiler needs to know where to find the assemblies and libraries you are referencing. Generally, the search path consists of any libraries that exist in the Global Assembly Cache, or GAC, and any assemblies that reside in the same directory as the files being compiled. You can specify alternate search paths using the `/lib` option. You can specify as many alternate directory paths as you'd like by separating each with a semicolon. You'll most likely use this option if you store your libraries in a central location on your machine and don't want to take the time to install them to the GAC or copy them into the directory for your current project. The following code uses the `lib` compiler option to include alternative library search paths:

```
jsc /t:exe /r:MyPersonalLib.dll /lib:c:\LibPath Source.js
jsc /t:exe /r:Personal.dll;Common.dll /lib:c:\PersonalLibs;c:\CommonLibs Source.js
```

Now that I've shown you how to include all the libraries you'll need, I should tell you that the JScript compiler will do most, if not all, of this referencing for you automatically. It is a special feature available only to the JScript compiler known as the `autoref` option. You can turn `autoref` either on or off, and it is on by default. The basic premise is that when you specify an import statement in your code, the namespace or package you are importing has a matching assembly. In other words, if you have some code that imports the System namespace, JScript tries to automatically reference the System.dll assembly. The `autoref` feature works as long as the namespace you are importing has an assembly with a matching name.

TIP

If you turn off the `autoref` option, you can speed up the compilation time of the program. The reason is that the compiler doesn't look for a number of non-existent assemblies when compiling. This can make a difference of several seconds or more when you have a large number of import statements in your source code.

So why would you want to turn off this option? If you are importing namespaceA and have a namespaceA assembly, but you just created a namespaceB assembly with a special test version of the namespaceA package, then you don't want the compiler to reference the wrong assembly. In this case you would turn `autoref` off so that namespaceA.dll didn't get automatically referenced by the compiler, and then you'd use a reference option to include namespaceB.dll. The following code demonstrates the use of the `autoref` compiler option to automatically include libraries based on import directives in the code:

```
jsc /t:exe /autoref+ Source.js
jsc /t:exe /autoref- /r:namespaceB.dll Source.js
```

As previously mentioned, mscorlib.dll is automatically referenced by the JScript compiler as well. But you haven't imported this namespace, so simply turning off the `autoref` option won't disable this specific import. To prevent mscorlib.dll from being referenced, you'll have to use the `/nostdlib` option. You can either turn this option on, which disables the mscorlib.dll reference, or turn it off, which is the default and allows the automatic reference of mscorlib.dll. In most cases you'll never use this option.

One side effect of the `/nostdlib+` option is that it changes the default of the `autoref` option to off as well. This means by enabling this option you'll have to either include all required references or turn the `autoref` option back on. Both of these options are rarely used with the JScript .NET compiler.

Code-Generation Options

It seems that the JScript compiler offers an endless supply of compiler options. I promise that these next compiler options—`out`, `debug`, `fast`, `versionsafe`, `utf8output`, `warn`, and `warnaserror`—will be much more important in actually getting your programs to run.

We start by examining the `/out` option. The `out` option allows you to change the target assembly name for your program. Generally, the compiler takes the name of the first source file and appends either an .exe or a .dll extension as appropriate and uses that as the target assembly name. Many times this is extremely inappropriate, so you want to supply the output filename on the command line. The following code specifies the output assembly through the use of the `out` compiler option:

```
jsc /t:exe /out:SuperUtility.exe Source.js
```

Along with code output comes debug output. Programs compiled under .NET with debug enabled emit a special file named identically to the assembly and ending with a .pdb extension. This special debug file contains information that maps code instructions back to the actual lines in the source file. This means you can tell exactly what line of source code was being executed when an exception was thrown. This can come in handy when you're debugging your programs.

To enable this special debug feature, you use the `/debug` option. This option can be turned either on or off and is turned off by default, disabling any special debug information. You should always turn this option on so that you have these debug libraries available to get better stack traces. You can delete the debug libraries before you distribute your programs so that others can't gain information about the internal workings of your program. However, if something happens in the field, you can always ship a customer your debug libraries to help you narrow down the issue. The following code demonstrates the use of the `debug` option to turn debugging symbols on and off:

```
jsc /t:exe /debug- Source.js
jsc /t:exe /debug+ Source.js
```

If you really want your programs to run as fast as possible, you'll obviously disable the debug output. But you'll also want to enable the `/fast` option. The `fast` option is another flag option and can be turned either on or off. The `fast` option is turned on by default, and it disables various language features that are considered to be legacy features and that are known bottlenecks to programming fast code.

CAUTION

Fast mode is defined by the JScript compiler team. They can change their optimizations at any point in time, possibly causing breaks in your code whenever they deprecate some feature you're using.

Fast mode requires that any variables are defined before they are used or a compiler error will result. Functions become constant, and they cannot be assigned to or redefined. Any predefined properties for built-in JScript objects become read-only, and you won't be able to use expando properties on those objects. The special variable, `arguments`, is disabled for use within function calls. The following code demonstrates turning the `fast` compiler option on and off:

```
jsc /t:exe /fast+ Source.js
jsc /t:exe /fast- Source.js
```

NOTE

There are many more fundamental changes you can make as well to improve your performance. First, be sure to use CLR arrays instead of JScript arrays. Second, always use StringBuilder to dynamically build strings rather than using string concatenation. And third, never make use of the more abstract JScript language features like expando properties except when initially testing a concept.

As if all the optimization and debugging options we've seen aren't enough, you can also specify the `/versionsafe` option to force an additional level of compliance on your code. The `versionsafe` option is another of the flag options. The default value is `off`, which allows your code to not explicitly define the `override` or `hide` keywords on members that would otherwise cover some member of an inherited class. If you decide to turn on this option, the compiler gives you a warning in the code where you haven't explicitly chosen either `override` or `hide` yet your derived method is covering an inherited method. The following code demonstrates the `versionsafe` compiler option being turned on and off:

```
jsc /t:exe /versionsafe+ Source.js
jsc /t:exe /versionsafe- Source.js
```

With your compiler throwing warnings for code fragments that aren't well defined or that don't perform very well, you'll want to have a way to specify that the compiler flag the warnings as errors and prevent the build step so that you can go back and fix these code deficiencies. The `/warnaserror` option is a flag attribute that allows you to turn warning messages into errors. The default for this value is `off` because most often warnings won't actually affect the operation of your code. When you turn this option on, the most basic of warnings become errors that stop compilation, and you'll have to either fix the warning to finish compiling or turn the option back off.

CAUTION

Depending on what warning level is set when you turn on the `warnaserror` option, you might not be able to fix everything the compiler is reporting. An example of this paradox is when you are casting a base class to a subclass in your code because you know that the code actually returns an instance of the subclass. The compiler labels this with a Level 1 warning because it knows that the cast might fail at runtime, but there's nothing you can do to the code to prevent the compiler from emitting the warning.

In addition to turning warnings into errors, you can raise the level of reported warnings using the `/warn` or `/w` option. All warnings fall into levels that detail their severity. Level 0 is the most severe type of warning, whereas a Level 4 warning is almost insignificant. By default, the warning level is set to 0; however, you can change this up to Level 4 so that the compiler returns more warnings during the compilation step so that you can see what types of performance issues your code is creating. The following code demonstrates how to turn warnings into errors, and how to set the warning level that the compiler emits during compilation:

```
jsc /t:exe /warnaserror+ Source.js
jsc /t:exe /warnaserror+ Source.js

jsc /t:exe /warn:4 Source.js
jsc /t:exe /warn:3 Source.js
jsc /t:exe /warn:2 Source.js
jsc /t:exe /w:1 Source.js
jsc /t:exe /w:0 Source.js
```

The final option is the `/utf8output` option. This is an extremely important option for localization purposes because it allows for the use of the UTF-8 character set in source and in the output binary. This option is off by default so the compiler emits normal ASCII; but most of the time if you're using any special UTF-8 characters, you simply need to enable this option flag to get the output you want. The following code demonstrates the creation of UTF8 assemblies via the `utf8output` compiler option:

```
jsc /t:exe /utf8output+ Source.js
jsc /t:exe /utf8output- Source.js
```

Most of these compiler options might be boring right now, but you'll find these options much more important later in the chapter when some of the sample code makes use of these features.

Conditional Compilation

Conditional compilation is used to enable or disable pieces of code before the code is sent to the compiler to generate the actual assembly. Conditional compilation comes in two steps:

1. Define your conditional symbols. This happens on the command line by use of the `define` compiler option. A sample symbol would be the `DEBUG` or `TRACE` symbols used by the CLR.
2. Wrap blocks of code within your program using some compiler directives.

The first step of conditional compilation is to define your conditional symbols on the command line. This happens on the command line using a special compiler flag. In this manner you can simply recompile your code using different flags to achieve different behaviors. This might be used to version code or provide extra levels of debug information depending on how the files are compiled.

After you've defined the symbols you want on the command line, the next step is to wrap some blocks of code within your program using some compiler directives. The compiler effectively runs over the file one time to look for these special directives. It either removes or inserts the code between the directives depending on what symbols you specified on the command line (we'll delve deeper into command-line symbols in the next section). This way the same source file can contain multiple code paths you can decide on at runtime. Only the code that is actually selected by the directives gets compiled and included in the executable. So you can use this to try out alternative forms of the same function or analyze your methods using performance calls at the beginning and end of code blocks.

Defining Symbols on the Command Line

Defining symbols on the command line is generally the last step in using the conditional compilation feature of JScript .NET. We'll discuss it first because you'll need to at least figure out what your symbols are going to be before you start coding, and you'll probably want to set up your build system with the various symbols before you start coding as well.

In JScript .NET the `/d` option is used to pass in new symbols. Symbols are only defined and don't hold any special values, so there are a couple of ways to specify multiple symbols to the JScript compiler. The first method to specify multiple symbols on the command line is to use one `define` compiler option for each symbol:

```
jsc /d:DebugSymbol /d:PerfSymbol /d:AdminSymbol Source.js
```

CAUTION

Try not to use the `DEBUG` or `TRACE` symbols because they are already defined and used by the CLR. If you do decide to use the `DEBUG` and `TRACE` symbols for your own code, you won't be able to control your code and the CLR code separately, which might lead to limitations in your ability to debug your program. You always want to be able to control debugging at the most atomic level when examining bugs, but you also want a system in which you can easily disable all debugging in your code for a final release.

The second method is to use a single `define` compiler option and to separate each symbol with semicolons or commas. The following code listing demonstrates the use of semicolons and commas to separate multiple symbols:

```
jsc /d:DebugSymbol,PerfSymbol /d:AdminSymbol;RetailSymbol Source.js
```

The command line isn't the only place where symbols can be defined. You can also define symbols in the source file; however, this isn't as effective as using the command line. One option is to generate different source files that contain only your conditional compilation directives and include these files depending on which symbols you want to define.

Defining symbols within the source file itself has some additional benefits we didn't see in the command-line symbols. For one, symbols defined in the source file can contain values, either numeric or Boolean, but might not contain string values. In this manner you could use a single conditional compilation variable to specify a version number and perform conditional compilation against the version rather than just checking to see whether the flag is defined.

One other feature of conditional compilation variables is that they can be used anywhere in JScript code. So we can examine their values to find out what is going on behind the scenes or to figure out why our code isn't getting compiled the way we think it should. Listing 9.1 demonstrates the use of source symbols, command-line symbols, and their various values depending on how they are either set in code or not set. The one thing to note is that any symbol that isn't set will be defined as NaN, or not a number. We can use this later in source to check for the existence of various symbols.

LISTING 9.1 Conditional Compilation Variables and Their Values

```
import System;

@set @SourceSymbol = true
Console.WriteLine("Source Symbol set to true");
Console.WriteLine(@SourceSymbol);

@set @SourceSymbol = false
Console.WriteLine("Source Symbol set to false");
Console.WriteLine(@SourceSymbol);

@set @SourceSymbol = 21
Console.WriteLine("Source Symbol set to 21");
Console.WriteLine(@SourceSymbol);

Console.WriteLine("Command Line Symbol that we define");
Console.WriteLine(@CommandLineSymbol);

Console.WriteLine("Command Line Symbol that we don't define");
Console.WriteLine(@UndefinedSymbol);
```

```
>nmake macros=/d:CommandLineSymbol
jsc.exe /t:exe  /d:CommandLineSymbol /out:9-1.exe 9-1.js

>9-1
Source Symbol set to true
True
Source Symbol set to false
False
Source Symbol set to 21
21
Command Line Symbol that we define
True
Command Line Symbol that we don't define
NaN
```

Now that you can define symbols both in code and on the command line, it's time you figured out how to incorporate that into your code. Next we'll talk about the various directives you can use to mark your code for conditional compilation.

Marking Code to Use Conditional Compilation

Conditional compilation in JScript .NET has two features: defining symbols and assigning values to symbols. You've already seen how to define symbols to the JScript .NET compiler;

that was the first feature. The second feature actually comes from being able to assign values to those symbols and then use them in normal code. However, using these symbols in normal code has nothing to do with conditional compilation.

To use the real features of conditional compilation, you'll first want to enable conditional compilation at the top of your source file. This can be done with the `@cc_on` directive:

```
@cc_on

import System;
   ...
```

After it's enabled, you can use the `@if...@elif...@else...@end` directive to wrap the code you want to conditionally compile. These statements are nearly identical to the `if` statement you learned about in Chapter 5, "Controlling Program Flow." You can use any of the complex conditional statements and any of the conditional operators. What you won't be able to use are any source variables; you can use only source symbols, command-line symbols, and numeric/Boolean literals. In Listing 9.2 we use command-line symbols to define which build of the source we want: Retail, Debug, or Checked. We also make use of a source variable in some cases to spit out versioning information.

LISTING 9.2 Using Conditional Directives to Modify Compiler Output

```
@cc_on
@set @MajorVersion = 1
@set @MinorVersion = 0

import System;

@if (@DebugBuild)
  Console.WriteLine("We are running Debug bits version " +
    @MajorVersion + "." + @MinorVersion);
@elif (@CheckedBuild)
  Console.WriteLine("We are running Checked bits verson " +
    @MajorVersion + "." + @MinorVersion);
@else
  Console.WriteLine("We are running Retail bits");
@end

@if (@MajorVersion < 1 && @RetailBuild)
  Console.WriteLine("Wait, we shouldn't be shipping" +
                    " with bad version numbers");
@end
```

You're going to want to play with this code quite a bit to get the hang of conditional compilation. First you'll want to play with the command-line flags. If you're using the source code I've provided along with the makefile, you'll need to use a special technique to pass in the command-line symbols. The following code demonstrates using a special command-line option for the nmake program to specify `define` compiler options to the JScript compiler for the sample code:

```
nmake 9-2.exe macros=/d:DebugBuild
nmake 9-2.exe macros=/d:CheckedBuild
nmake 9-2.exe macros=/d:RetailBuild
```

When you're running in Retail build, there is one additional check to perform. You want to make sure that Retail doesn't ship using an invalid version, so you must make sure that `@MajorVersion` is greater than `0`. To test this option you'll need to change the source file to set `@MajorVersion` equal to `0`, and you'll also need to compile the source using the `nmake` step for the `RetailBuild` symbol.

Because conditional compilation is used only in special circumstances, we'll end our discussion here. Feel free to play with the source code a bit and examine conditional compilation in more detail. We are going to move on to some additional code directives that might become useful when you're debugging your source code.

Additional Code Directives

JScript .NET has several additional code directives, including the `@option` directive, `@debug` directive, and `@position` directive. Many of these code directives can be turned on or off from the command line. But you can also turn them on or off from source. If you're distributing source code to people who aren't familiar with your program, this can be extremely handy because they don't need any special instructions for compiling the code.

The first code directive is the `@option` directive. This can currently take only one value, `versionsafe`, but in the future it might be used to set many more compilation options. If you don't remember how the `versionsafe` option modifies compilation, you'll want to refer to the beginning of this chapter for a refresher course. The following code demonstrates setting the `versionsafe` compiler option from within the source code:

```
@set @option (versionsafe)
```

You can also turn debug information on and off at the source level. This is actually more effective than the command-line equivalent, because it allows you to turn debugging on only for certain pieces of the code and off for others. This might be because you have some private code containing debug informatin that you wouldn't want an end user to see. The following code demonstrates the use of the `@debug` source directive to turn debugging symbols on and off within the source code:

```
// No debugging information here
@set @debug(off)
var PrivateVariable;

// Turn debugging information back on
@set @debug(on)
var debugVariable;
function debugFunction() { /* ... */ };

// Turn debugging information back off again
@set @debug(off)
function PrivateFunction() { /* ... */ };
```

The next directive lets you modify the debug output for the current source file. Normally, all debug information is set up by the compiler with the filename being the current file processed, line numbers starting at `1`, and column numbers starting at `0`. As the source file is read by the compiler, these values are updated and used to generate the debug information. These values can be modified using the `position` directive.

The `position` directive can be used to modify these internal compiler counters while the source file is being parsed. This is handy for host applications that run bits and pieces of a user's source code in between host code. Instead of throwing an exception and reporting the current file, line, and column in the host file, a `position` directive can be used that will point the debug information back to the user's original source file so that they know where to go back and debug later. ASP.NET makes extensive use of this feature so that the users can find the appropriate line within their aspx file that caused the problem rather than getting referred to some line within the auto-generated code.

After it's specified, a `position` directive will continue to modify the position values within the file until the end of the source input is reached or the special value of `end` is passed to the `position` directive. All of this can seem rather abstract because most programmers never write code that hosts the code of other users. But I think I have an example that might help explain this in a bit more detail. Let's assume that you are writing an algorithm and you want to figure out how long loop iterations are taking. Now it looks as though you need to go back and instrument your code. When you add this instrumentation code, you'd really like to know when exceptions are thrown inside of this code versus inside of your actual program. This can easily be accomplished with the `position` directive. The next two code listings demonstrate the same source file with and without using the position directive. The next code listing is our original program; the code listing immediately following that is a modified version of that same program with some instrumentation code buried:

```
import System;

class AlgorithmHolder {
    public static function Algorithm1() : void {
         return;
    }
    public static function Algorithm2() : void {
         return;
    }
}

var rand:Random = new Random();
var i:int = 0;

for(i = 0; i < 20; i++) {
    switch(rand.Next(0,2)) {
        case 0:
            AlgorithmHolder.Algorithm1();
            break;
        case 1:
            AlgorithmHolder.Algorithm2();
            break;
        default:
            // Never gets called
            AlgorithmHolder.Algorithm1();
            break;
    }
}
```

Now, we'll write some special code that parses this source file and inserts the profiling code into each function. This is an advanced concept and we won't actually make this code; we'll do the insertions by hand to create our profiled source code. Listing 9.3 demonstrates some code that might be created by a code-profiling utility.

LISTING 9.3 Injected Profiler Code and `Position` Directives

```
@cc_on
@set @position(file="Profiler.js";line=1;column=0)
import System;
import System.Collections;

class Profiler {
    private static var hsh:Hashtable;

    public static function CallFunction(name:String) : void {
        if (hsh == null) {
```

LISTING 9.3 continued

```
            hsh = new Hashtable();
        }

        if ( hsh[name] == null ) {
            hsh[name] = 1;
        } else {
            hsh[name]++;
        }
    }

    public static function PrintCallUsage() : void {
        var key:String;

        for(key in hsh.Keys) {
            Console.WriteLine("{0,-20} {1}", key, hsh[key]);
        }
    }
}
// CallFunction Injected Code Failure
// PrintCallUsage Injected Code Failure
@set @position(end)
import System;

class AlgorithmHolder {
    public static function Algorithm1() : void {
@set @position(file="Profiler.js";line=28;column=0)
        Profiler.CallFunction("Algorithm1");
@set @position(end)
         return;
    }
    public static function Algorithm2() : void {
@set @position(file="Profiler.js";line=28;column=0)
        Profiler.CallFunction("Algorithm2");
@set @position(end)
         return;
    }
}

var rand:Random = new Random();
var i:int = 0;

for(i = 0; i < 20; i++) {
    switch(rand.Next(0,2)) {
        case 0:
```

LISTING 9.3 continued

```
            AlgorithmHolder.Algorithm1();
            break;
        case 1:
            AlgorithmHolder.Algorithm2();
            break;
        default:
            // Never gets called
            AlgorithmHolder.Algorithm1();
            break;
    }
}

@set @position(file="Profiler.js";line=29;column=0)
    Profiler.PrintCallUsage();
@set @position(end)
```

Let's run over what just happened in Listing 9.3. First we blasted the contents of `Profiler.js` into the top of the file, and we set it up so that if that function fails we'll get debug information which points back to the `Profiler.js` file rather than to our auto-generated file. We also took the liberty of inserting some profiler calls into each of our functions, and finally at the bottom to print the usage statistics. If one of these functions fails, we also get referred to `Profiler.js` instead of 9-3.js. This is just one sample usage of this particular directive; unless you're a writer of profiling and hosting tools, I don't suggest that you spend too much time pondering the preceding code. Let us instead move on to the special symbols declared within the CLR.

Symbols Defined by the CLR

The CLR gets kind of tricky with conditional compilation. Conditional compilation generally happens at the compiler level and actually omits code from the executable that is being built. What happens when you write a code library that is precompiled and distributed to customers and they need to use conditional compilation to turn features on and off? The short answer is that they can't do so using only the compiler. The longer answer is that the CLR provides a special mechanism for turning code on and off that is already compiled into a library known as the `ConditionalAttribute`. The `ConditionalAttribute` can be applied at the method level to mark methods as callable based on various compile-time conditions. The following code demonstrates the creation of a function that gets called only from code that is compiled with the Database command-line symbol:

```
import System;
import System.Diagnostics;
```

```
public ConditionalAttribute("Database") function DatabaseLogger() : void {
    // Some code to log to the database
}

// This next line will either be inserted or omitted from the code based
// on the Database symbol being defined
DatabaseLogger();
```

This would be exactly identical to using some conditional compilation code in the compiler. The following code demonstrates how conditional compilation code can be used to remove the call to the database logging function:

```
@cc_on
import System;
import System.Diagnostics;

public function DatabaseLogger() : void {
    // Some code to log to the database
}

// This next line will either be inserted or omitted from the code based
// on the Database symbol being defined
@if (@Database)
DatabaseLogger();
@end
```

However, specifying the attribute on the methods tends to be quite a bit cleaner, yet less well defined. If you saw the first example in which the `ConditionalAttribute` was used instead of conditional compilation, then you might realize that if the actual function was located in another assembly and was already precompiled, you needed to specify the compile-time symbol to get it working. The reason for delving into this feature is that the CLR uses it in several locations and actually defines some special symbols that can be used to activate various classes.

The first such symbol is the `DEBUG` symbol. When the `DEBUG` symbol is specified on the command line, additional logging functionality and tracing information are often available. The `System.Diagnostics.Debug` class has many methods that are enabled and useful only when the `DEBUG` symbol is specified on the command line.

The second symbol is the `TRACE` symbol. The `TRACE` symbol behaves identically to the `DEBUG` symbol and simply activates certain classes and their methods. This time the primary class that is activated is the `System.Diagnostics.Trace` class and its methods. Both of these classes are extremely handy when you're debugging your application in that they allow you to insert debugging and tracing logic that gets compiled out in your retail builds without any changes to the code. A code stripping tool is always one option, but when it happens at the compiler level as a service, it is always much cleaner and more maintainable.

That concludes the discussion of conditional compilation symbols and moves us into our next compiler-level feature, the assembly-level attribute.

Assembly-Level Attributes

So far we've seen that attributes can be applied at all levels of our code to affect the various classes, methods, properties, and fields. However, we haven't seen a way of specifying attributes at the assembly level. Assembly-level attributes are important for specifying information that applies to the entire assembly, such as version information, key signing, and strong name information, and any custom attributes you might want to create as a programmer to identify your assembly.

As an example, you might want to create a special attribute that identifies which class should be run when an assembly is loaded into your application. This comes in handy if your application is going to support plug-ins, because it allows the developer to specify which class in his assembly is the plug-in connector class. For a special .NET game, we allowed the developer to create special animals. When we loaded the developer's assembly, we had to know which class was his animal so that we could load it and start him off in the game. Otherwise, we might have loaded one of the helper classes instead and that wouldn't have worked very well. So what is so different about assembly attributes?

Assembly Attribute Syntax

Assembly-level attributes are defined in the global area of the source code. Normally you would apply them at the top of the source file before any other code so that they are easy to find and modify later. Each attribute should be surrounded by brackets and prefixed with the keyword `assembly` followed by a semicolon. The rest of the syntax should be rather familiar to you because you've used attributes for other code items. The following code demonstrates some assembly-level attributes being applied in some JScript .NET source:

```
import System;

[assembly: MyAssemblyAttribute(MyParm1, MyParm2) ]
[assembly: MyAssembly2Attribute(MyParm1, MyParm2) ]

    // Source Code
```

Versioning Attributes

We've mentioned a couple of areas in which assembly-level attributes might be used, but the most prominent is probably the versioning attribute. Version attributes affect the special version page you see when you right-click an assembly in the Windows Explorer and then click the Version tab. Some of the types of information normally provided are the assembly author, a description, a title, any copyright information, and version information.

The important attributes for our discussion are the `AssemblyCompanyAttribute`, the `AssemblyCopyrightAttribute`, the `AssemblyDescriptionAttribute`, the `AssemblyFileVersionAttribute`, the `AssemblyProductAttribute`, the `AssemblyTitleAttribute`, the `AssemblyTrademarkAttribute`, and the `AssemblyVersionAttribute`. That's certainly a long list, so you'll probably want to create some sort of template file that contains each of these items so you can just copy them between projects. I've included one template file with the media for this book, or you can type them yourself from Listing 9.4.

LISTING 9.4 Assembly Version Attributes

```
import System;
import System.Reflection;

[assembly: AssemblyCompanyAttribute("DigiTec Web Consultants")]
[assembly: AssemblyCopyrightAttribute("Copyright DigiTec Web Consultants 2001")]
[assembly: AssemblyDescriptionAttribute("A Super Utility")]
[assembly: AssemblyFileVersionAttribute("1.0.*")]
[assembly: AssemblyProductAttribute("Super Utility")]
[assembly: AssemblyTitleAttribute("Super Utility")]
[assembly: AssemblyTrademarkAttribute("Trademark DigiTec Web Consultants 2001")]
[assembly: AssemblyVersionAttribute("1.0.*")]
```

Strongly Naming Your Assembly Using Attributes

Another feature of the assembly-level attribute is the capability to attach a strong name to your assembly. After this is done, you can ensure that someone can't simply delete your assembly, rename one of their own, and stick it in place and expect it to work. You can actually specify that you want your assembly with your special strong name, and the CLR looks only for that assembly.

To begin this process you have to first generate a key pair. This can be done with the sn.exe utility shipped with the .CLR. If you run the utility with the `-k` option and specify an output filename for the key file, you'll have enough to strongly name and sign your assembly. The following code demonstrates the creation of a strong naming key:

```
sn -k MyApp.snk
```

After the key pair is generated, you simply apply an assembly-level attribute that specifies the key file to use for signing. The compiler handles the rest. The attribute in question is the `AssemblyKeyFileAttribute`, which takes only the path to the key file. This can be a relative or absolute path, but for everything to work the key file must exist in the location specified. The following code uses an assembly-level attribute to apply a strong naming key to the source file and resulting assembly:

```
import System;
import System.Reflection;

[assembly: AssemblyKeyFileAttribute("MyApp.snk") ]
```

After you compile your code, you are able to refer to your assembly using its strong name rather than its normal assembly name. You also can register your assemblies in the GAC, which makes them available to any application on the system. This can be very useful information, especially if you're planning to write and distribute code libraries for use by other developers.

Summary

This chapter had several major features that will remain important throughout the rest of the book. We began the chapter by re-iterating through some of the compiler options we saw earlier in the book to compile the sample code up until now. We expanded on this to talk about all the various options available to you as the programmer and all the different features you can turn either on or off.

You also saw the various code-generation options the compiler offers. These are important because they affect the way the code is parsed and compiled, and they can also affect the compiler's output. If you don't fully understand these options, you might not understand some of the more discrete errors generated by your program because it was compiled incorrectly.

From compilation we moved into conditional compilation and defining command-line symbols and source symbols, and using those various symbols to either enable or disable code at compile-time. You also learned that some symbols have to be defined for some of the CLR classes to work and that you can also use attributes to define methods that should be conditionally compiled rather than wrapping the function calls in conditional compilation syntax.

Finally, we took a look at assembly-level attributes, their strange syntax, and their many uses. You learned that the primary use of the assembly-level attribute was for specifying assembly versioning information. In addition we briefly examined strongly named assemblies and using signing tools and assembly attributes to accomplish this task.

In the next chapter we'll look at the System.IO namespace in full detail. Expect lots of code, lots of diagrams, and lots of study, because File IO is probably the most often-used programming tool for any developer.

CHAPTER

10

File Access

IN THIS CHAPTER

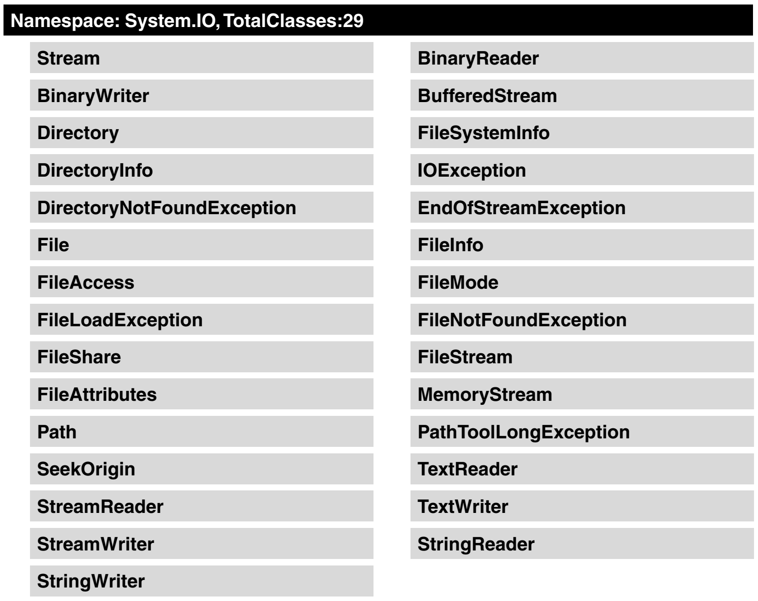
File access and manipulation is an extremely important programming task for most applications. File access is the process of locating files on disk and opening them in the application; file manipulation is the process of reading and writing to files. Large amounts of data needed by the application that are too large to stay in memory need to be accessed from disk. Intermediate calculations might have to be persisted to disk in case a user wants to come back and complete the operations later. More important, storing user preferences and information so that the user experience of the application can be as pleasing as possible has to be done by using file IO. More often than not, great pains are taken to implement different persistence formats, whether in binary, in a textual representation readable by the user, or now in XML. It's difficult to have to write new file access mechanisms every time some small feature changes. With this in mind, the CLR team has provided a very comprehensive set of APIs so that the amount of code needed to read and write files is as small and atomic as possible.

Under JScript .NET, the methods for performing file IO are located in the System.IO namespace. The System.IO namespace has quite a few members, but for the most part only a few of them are ever used. You'll be examining the file-system navigation objects, `File`, `FileInfo`, `Directory`, `DirectoryInfo`, and `Path`. Each of these file-system navigation objects relates in some manner to investigating the underlying file system to find available files, get and set file attributes, and even get information on how to work with underlying file systems that are different from the standard FAT/NTFS that most Windows programmers are used to.

Then you'll discover that all the actual IO objects require a `Stream` object that provides a minimal set of functionality for reading and writing to any information stream on the system. You'll examine how to use this `Stream` class to create higher level IO objects such as `BinaryReader`, `BinaryWriter`, `StreamReader`, and `StreamWriter`. Figure 10.1 shows the System.IO namespace and its many objects.

File, FileInfo, Directory, DirectoryInfo, and Path

The `File`, `FileInfo`, `Directory`, `DirectoryInfo`, and `Path` classes are used in some way to navigate or query the underlying file system for files, directories, and their attributes. Let's start with the `Path` object because you'll probably want to use it to build paths to pass to the rest of the navigation objects.



Namespace: System.IO, TotalClasses:29

Stream	BinaryReader
BinaryWriter	BufferedStream
Directory	FileSystemInfo
DirectoryInfo	IOException
DirectoryNotFoundException	EndOfStreamException
File	FileInfo
FileAccess	FileMode
FileLoadException	FileNotFoundException
FileShare	FileStream
FileAttributes	MemoryStream
Path	PathToolLongException
SeekOrigin	TextReader
StreamReader	TextWriter
StreamWriter	StringReader
StringWriter	

FIGURE 10.1
The System.IO namespace.

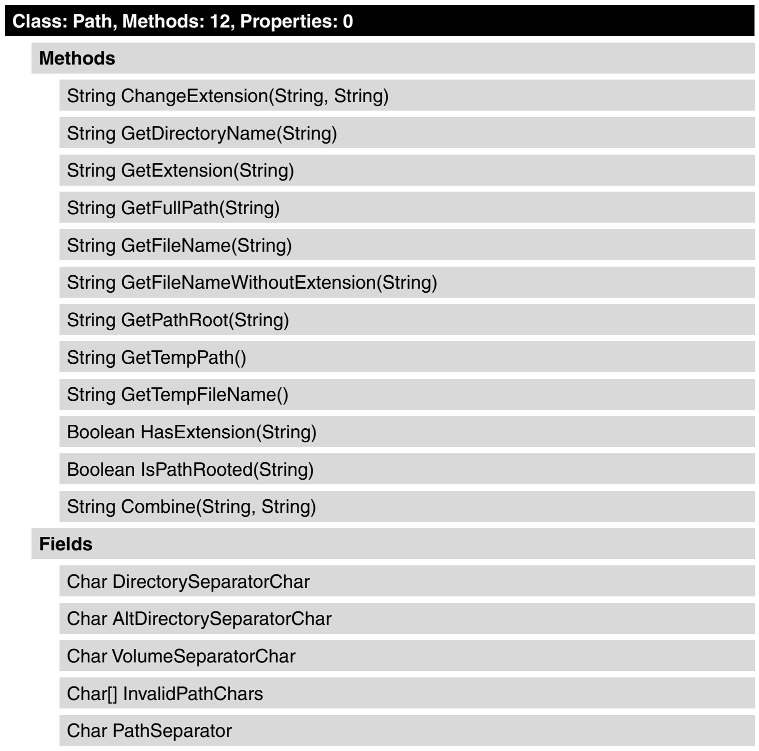
The `Path` Class

Figure 10.2 begins the investigation of the `Path` class with a member map of the class.

Why use the path object to build paths rather than just building them using string concatenation or hard coding them into the file? You can no longer assume that the underlying file system your program is running on supports the path and directory semantics you're used to. By using the `Path` object, you can build paths that are custom tailored to work on whatever the underlying file system might be.

The `Path` class has several static fields that can be used to describe how paths, files, and volumes are each separated, along with which characters are invalid given the underlying file system. The `DirectorySeparatorChar` and `AltDirectorySeparatorChar` fields each describe the symbols used to separate directory names and pathnames in fully qualified or relative paths. The `VolumeSeparatorChar` field describes the character used to split a volume name from the rest of the path. The `PathSeparatorChar` field is generally available only on Windows systems, and is the character used to separate multiple paths. And, finally, the `InvalidPathChars` field can be used to determine which characters can and cannot be used on the target file system. All of this can easily be summed up in a small code listing to show where each of these fields is useful when building paths:

```
C:\Program Files/Microsoft.NET\SomeFile.txt
 ^               ^              ^ DirectorySeparatorChar
 ^               ^ AltDirectorySeparatorChar
 ^ VolumeSeparatorChar
```



Class: Path, Methods: 12, Properties: 0

Methods

- String ChangeExtension(String, String)
- String GetDirectoryName(String)
- String GetExtension(String)
- String GetFullPath(String)
- String GetFileName(String)
- String GetFileNameWithoutExtension(String)
- String GetPathRoot(String)
- String GetTempPath()
- String GetTempFileName()
- Boolean HasExtension(String)
- Boolean IsPathRooted(String)
- String Combine(String, String)

Fields

- Char DirectorySeparatorChar
- Char AltDirectorySeparatorChar
- Char VolumeSeparatorChar
- Char[] InvalidPathChars
- Char PathSeparator

FIGURE 10.2
The `Path` *class member map.*

NOTE

The preceding code listing is being used to demonstrate a Windows directory structure. However, I could have just as easily used UNIX, WinCE, or a Macintosh path instead. If and when the CLR makes it onto operating systems other than Windows, the `Path` class will properly support the creation of paths on those systems using the same code that would be needed to create a path on a Windows system. This file-system abstraction is actually pretty powerful, and all the code written in this chapter will use the `Path` class to build paths.

With the preceding code listing you can see the use of three of the five fields discussed, the `DirectorySeparatorChar`, `AltDirectorySeparatorChar`, and `VolumeSeparatorChar`. `PathSeparatorChar` would be used to combine several paths together. Anyone familiar with the `PATH` environment variable on a Windows system will know that to specify multiple paths, the semicolon (`;`) character is used as the `PathSeparatorChar`. The `InvalidPathChars` could be used to do a replacement on user input strings to replace invalid characters with a default valid character or to alert users that they entered an invalid path and need to supply another. It's easy to see that just by using the special fields on the `Path` class an application could easily be made to work on systems with extremely different file systems. In addition to the fields available, the `Path` class also contains many helpful methods for building and examining paths.

The `Path` methods can be broken down into a couple of smaller groups. There are methods for examining a current path and determining various path-related properties. The `HasExtension` method can be used to determine whether the path ends with a filename that contains an extension. The best use of this method is to determine whether a user-input filename and path needs to have a default extension added, but you could also use it to determine whether the specified path is a directory path or a file path. The `IsPathRooted` method returns information on whether the given path contains a volume root. This could be either a rooted file-system path such as the one used in the previous code listing, or a rooted UNC pathname with a server and sharename. Often it's very important to store fully qualified names, and this method is a quick check.

Another set of `Path` methods is related to creating or modifying a current path. The `ChangeExtension` method can be used to either change or remove an existing file extension on the end of a path. The `Combine` method is useful for combining several smaller pieces of a path into a single path. This is most often the method used to concatenate directory names together to build a full path to a file because it automatically uses the correct separator characters. The `GetDirectoryName`, `GetExtension`, `GetFileName`, `GetFileNameWithoutExtension`, and `GetPathRoot` methods are all pretty self-explanatory. The `GetFullPath` method expands a path to a fully qualified path. This is often used in conjunction with the `IsPathRooted` method because you can check the path to see whether it is rooted, and if not, call this function to return a rooted version of the path.

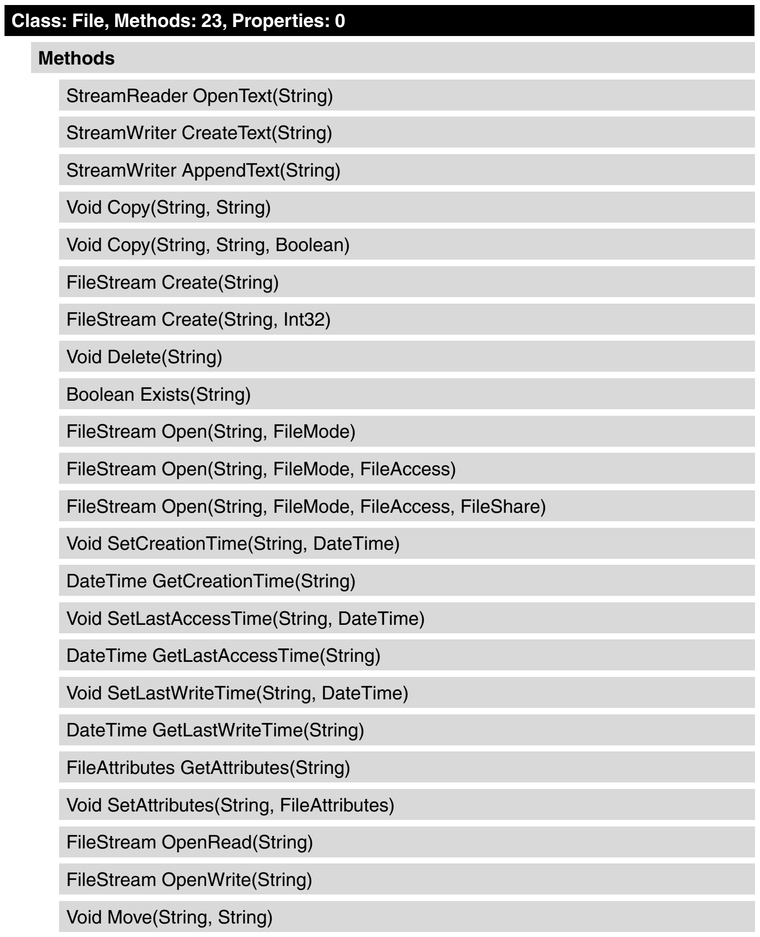
A final set of methods is available for working with temporary files. The `GetTempPath` method is used to gain access to the system's temporary directory. The `GetTempFileName` method not only returns the directory, but also creates a 0-length temporary file and filename. This is extremely handy for storing information while your program is running.

TIP

It's important to know what methods are available to avoid reproducing any of the behavior later in user code.

The `File` Class

The `File` class is used to get information about files. Figure 10.3 illustrates the various properties and methods available to inspect files and get their information.



Class: File, Methods: 23, Properties: 0

- **Methods**
  - StreamReader OpenText(String)
  - StreamWriter CreateText(String)
  - StreamWriter AppendText(String)
  - Void Copy(String, String)
  - Void Copy(String, String, Boolean)
  - FileStream Create(String)
  - FileStream Create(String, Int32)
  - Void Delete(String)
  - Boolean Exists(String)
  - FileStream Open(String, FileMode)
  - FileStream Open(String, FileMode, FileAccess)
  - FileStream Open(String, FileMode, FileAccess, FileShare)
  - Void SetCreationTime(String, DateTime)
  - DateTime GetCreationTime(String)
  - Void SetLastAccessTime(String, DateTime)
  - DateTime GetLastAccessTime(String)
  - Void SetLastWriteTime(String, DateTime)
  - DateTime GetLastWriteTime(String)
  - FileAttributes GetAttributes(String)
  - Void SetAttributes(String, FileAttributes)
  - FileStream OpenRead(String)
  - FileStream OpenWrite(String)
  - Void Move(String, String)

FIGURE 10.3
The `File` Class Member Map.

The `File` class contains all the methods needed to manipulate an individual file. One set of methods is used to work with the file as a whole. These include the `Copy`, `Exists`, `Move`, and `Delete` methods. Each of these can be used to manipulate the file all at once in a single call.

Another set of methods is used to actually gain access to the contents of the file. These include the `Create`, `CreateText`, `Open`, `OpenRead`, `OpenText`, and `OpenWrite` methods. Notice that the `File` object has special methods for gaining access to `Text` files. This just goes to show what some of the more common operations on files will be.

A final set of methods is used to set meta-information for the file. Meta-information includes information about the file that's stored not as part of the data stream, but rather by the underlying file system. This includes access to the `File` attributes and time information for file creation and modification. The `GetAttributes`, `GetCreationTime`, `GetLastAccessTime`, and `GetLastWriteTime` methods are each used to obtain this information about the file. The `SetAttributes`, `SetCreationTime`, `SetLastAccessTime`, and `SetLastWriteTime` methods are in turn used to write this information about a file.

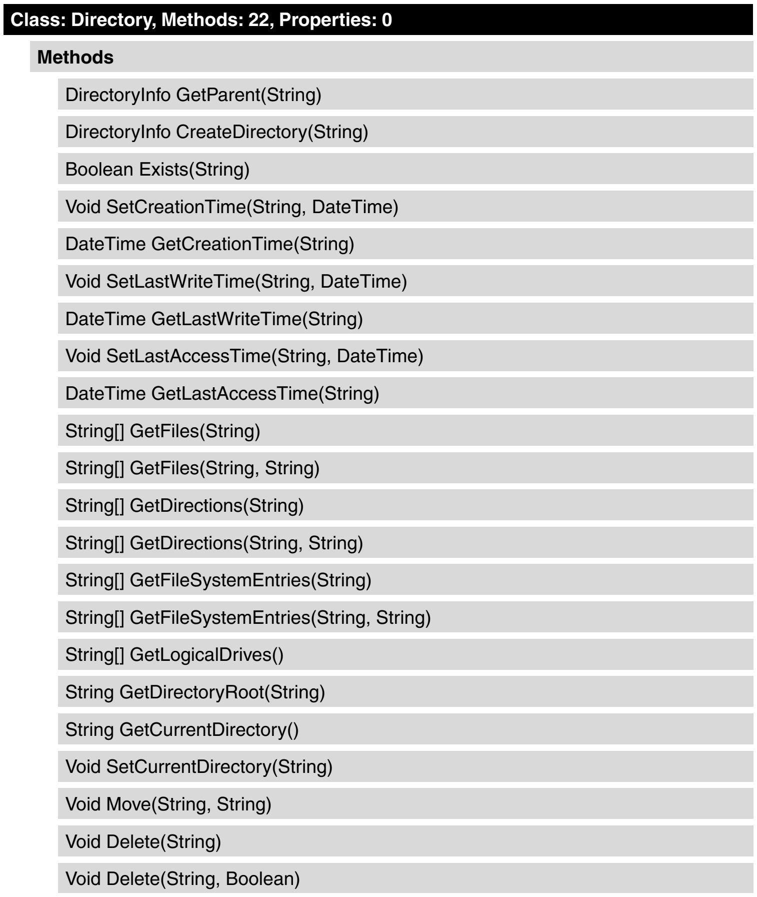
The `Directory` Class

The `Directory` class contains all the various methods needed to investigate the properties of directories. This includes enumerating files and subdirectories, determining access permissions, and other directory-related data. More specific methods and properties can be determined from Figure 10.4.

The `Directory` class is nearly identical to the `File` class except that it works with directories. Directories are different from files in that they don't contain data and can't be written to. Instead they contain and logically separate other directories and files. Again, let's break the methods down into small groups so that they are easily digestible.

Again, there is a set of methods for working on the directory as a whole. These include the `Exists`, `Move`, and `Delete` methods. Notice that there isn't a copy method. Copying a directory also means copying all the files and subdirectories it contains. This isn't a difficult operation, but generally speaking the replication of data is unwanted.

Several methods exist for gaining access to directory contents. Gaining access to a directory is also a bit different from working with and gaining access to files because you can't open them and retrieve a stream of data. Instead, there are the `CreateDirectory`, `GetDirectories`, `GetFiles`, `GetFileSystemEntries`, and `GetLogicalDrives` methods. `GetDirectories` returns the list of directories as a string array, `GetFiles` returns the list of files, and `GetFileSystemEntries` returns a listing of the other two functions combined. `GetLogicalDrives` actually returns all the root paths for the current system. You never actually open the directory or do any stream manipulation; you simply retrieve lists of the contents of the directory.



Class: Directory, Methods: 22, Properties: 0

Methods

- DirectoryInfo GetParent(String)
- DirectoryInfo CreateDirectory(String)
- Boolean Exists(String)
- Void SetCreationTime(String, DateTime)
- DateTime GetCreationTime(String)
- Void SetLastWriteTime(String, DateTime)
- DateTime GetLastWriteTime(String)
- Void SetLastAccessTime(String, DateTime)
- DateTime GetLastAccessTime(String)
- String[] GetFiles(String)
- String[] GetFiles(String, String)
- String[] GetDirections(String)
- String[] GetDirections(String, String)
- String[] GetFileSystemEntries(String)
- String[] GetFileSystemEntries(String, String)
- String[] GetLogicalDrives()
- String GetDirectoryRoot(String)
- String GetCurrentDirectory()
- Void SetCurrentDirectory(String)
- Void Move(String, String)
- Void Delete(String)
- Void Delete(String, Boolean)

FIGURE 10.4
The `Directory` class member map.

NOTE

The file and directory enumeration functions have two forms. The first form returns all files or all directories. The second form actually takes a search pattern to limit the returned results.

Another set of methods is unique to the `Directory` class and encompasses the getting and setting of the current working directory. The `GetCurrentDirectory` method is used to find out what the current working directory for the application is. This is important because it will be used as the root directory for any files or relative paths that aren't fully qualified and contain a root. You can set the current directory with `SetCurrentDirectory`.

The final set of methods is used to access file-system meta-data. These include methods for getting meta-data, `GetCreationTime`, `GetLastAccessTime`, `GetLastWriteTime`; and the methods for setting the meta-data, `SetCreationTime`, `SetLastAccessTime`, and `SetLastWriteTime`.

Now that you know all the methods on the `Directory` and `File` classes, we can write at least some small pieces of code. You'll remember that the `Directory` object doesn't support a copy function. So here you'll be writing several small functions to encompass the copying of an entire directory. This comes in two forms: shallow copy and deep copy. A shallow copy of a directory is when you copy the directory and any included files, but you don't copy subdirectories and you don't recurse into those directories and copy their files. A deep copy of a directory copies all files, directories, and subdirectories. Both forms are important because you generally don't want to deep copy an entire directory because of space considerations. Listing 10.1 is the code for a shallow directory copy algorithm.

LISTING 10.1 Shallow Directory Copy

```
import System;
import System.IO;

public class DirectoryCopier {
    public static function ShallowCopy(srcDir:String, destDir:String) : int {
        var curFile:int;
        var Files:String[];

        // If Destination exists don't copy
        if ( Directory.Exists(destDir) ) {
            return 0;
        }

        // If Source doesn't exist don't copy
        if ( !Directory.Exists(srcDir) ) {
            return 0;
        }

        // Create the destination directory
        Directory.CreateDirectory(destDir);

        // Get the list of files to copy
        Files = Directory.GetFiles(srcDir);
        for(curFile = 0; curFile < Files.Length; curFile++) {
```

LISTING 10.1 continued

```
            // Remove any directory Information
            var file:String = Path.GetFileName(Files[curFile]);

            // Append directory information and Copy
            File.Copy(
                Path.Combine(srcDir, file),
                Path.Combine(destDir, file));
        }

        return Files.Length;
    }
}

var Args:String[] = Environment.GetCommandLineArgs();
Console.WriteLine(DirectoryCopier.ShallowCopy(Args[1], Args[2]));
```

That should be all the code needed to do a simple shallow directory copy. Now you should be able to use the preceding code in the deep directory copy. After all, this already copies all the files from one directory to another. What you want to do is add an overload for the shallow copy function that doesn't check to see whether the destination directory already exists. In many cases you won't care whether it exists because if it does you simply don't try to create it. Now, you'll add a small piece of code to loop through and shallow copy each directory recursively and I think we'll have a workable function. Listing 10.2 demonstrates a deep directory copying algorithm.

LISTING 10.2 Deep Directory Copy

```
import System;
import System.IO;

public class DirectoryCopier {
    // This is our original function
    public static function ShallowCopy(srcDir:String, destDir:String) : int {
ShallowCopy(srcDir, destDir, true);
    }

    // Here is the overload I promised
    public static function ShallowCopy(srcDir:String, destDir:String,
        createDirectory:Boolean) : int {
var curFile:int;
        var Files:String[];
```

LISTING 10.2 continued

```
        // If Destination doesn't exist then ...
        if ( !Directory.Exists(destDir) && createDirectory ) {
            // Create the destination directory
            Directory.CreateDirectory(destDir);
        } else {
            // If directory doesn't exist and we don't
            // createDirectory then we should fail.
            return 0;
        }

        // If Source doesn't exist don't copy
        if ( !Directory.Exists(srcDir) ) {
            return 0;
        }

        // Get the list of files to copy
        Files = Directory.GetFiles(srcDir);
        for(curFile = 0; curFile < Files.Length; curFile++) {
            // Remove any directory Information
            var file:String = Path.GetFileName(Files[curFile]);

            // Append directory information and Copy
            File.Copy(
                Path.Combine(srcDir, file),
                Path.Combine(destDir, file));
        }

        return Files.Length;
    }

    public static function DeepCopy(srcDir:String, destDir:String) : int {
        var Directories:String[];
        var curDirectory:int;
        var filesCopied:int;

        // First we shallow copy
        filesCopied = ShallowCopy(srcDir, destDir, true);

        Directories = Directory.GetDirectories(srcDir);
        for(curDirectory = 0; curDirectory < Directories.Length;
            curDirectory++) {
            // Remove all but target directory
           var directory:String =
                Path.GetFileName(Directories[curDirectory]);
```

LISTING 10.2 continued

```
            filesCopied += ShallowCopy(
                Path.Combine(srcDir, directory),
                Path.Combine(destDir, directory),
                true);
        }

        return filesCopied;
    }
}

var Args:String[] = Environment.GetCommandLineArgs();
Console.WriteLine(DirectoryCopier.DeepCopy(Args[1], Args[2]));
```

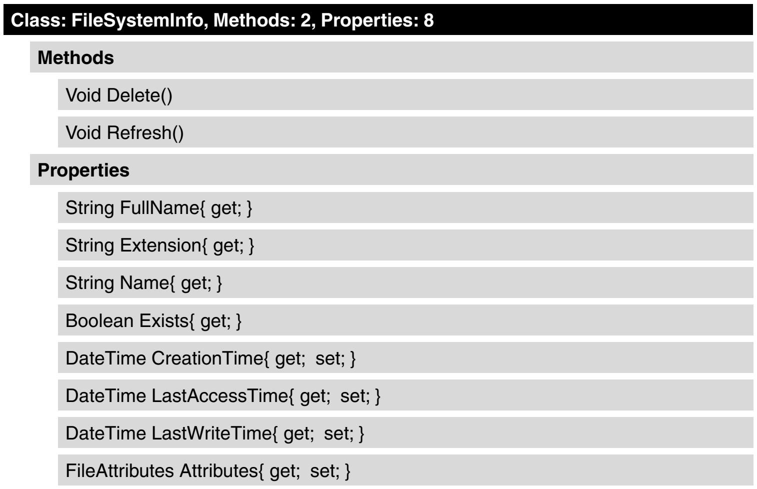
Well, that's it. You just worked through two different directory copy algorithms. Now let's move on to the `FileInfo` and `DirectoryInfo` classes and examine extended file and directory information.

The `FileSystemInfo`, `FileInfo`, and `DirectoryInfo` Classes

Both `FileInfo` and `DirectoryInfo` inherit the base class `FileSystemInfo`. `FileSystemInfo` contains some base properties and methods that work for both files and directories. The fields `FullPath` and `OriginalPath` describe the location of the current `FileSystemInfo` object. `FullPath` is the fully qualified path to the resource, and `OriginalPath` is the path the user originally specified, whether relative or absolute, when creating the `FileSystemInfo` object. Figure 10.5 shows all the properties and methods available in the `FileSystemInfo` class.

Some properties are available that encapsulate all the information for a given file-system object. These include the `Attributes` property, which describes the file-system attributes; the `CreationTime` property; the `LastAccessTime` property; and the `LastWriteTime` property, which returns all the pertinent information on when the given file-system object was accessed and created. The `Name` property returns the name of the file or directory minus any information about parent directories or root directories. The `FullName` property returns the full path and filename of the file or directory.

You can refresh the information in this `FileSystemInfo` object by calling the `Refresh` method. This comes in handy if other processes are accessing and changing the files and you need the most up-to-date information. You can even delete file-system objects using the `Delete` method.



Class: FileSystemInfo, Methods: 2, Properties: 8

Methods

- Void Delete()
- Void Refresh()

Properties

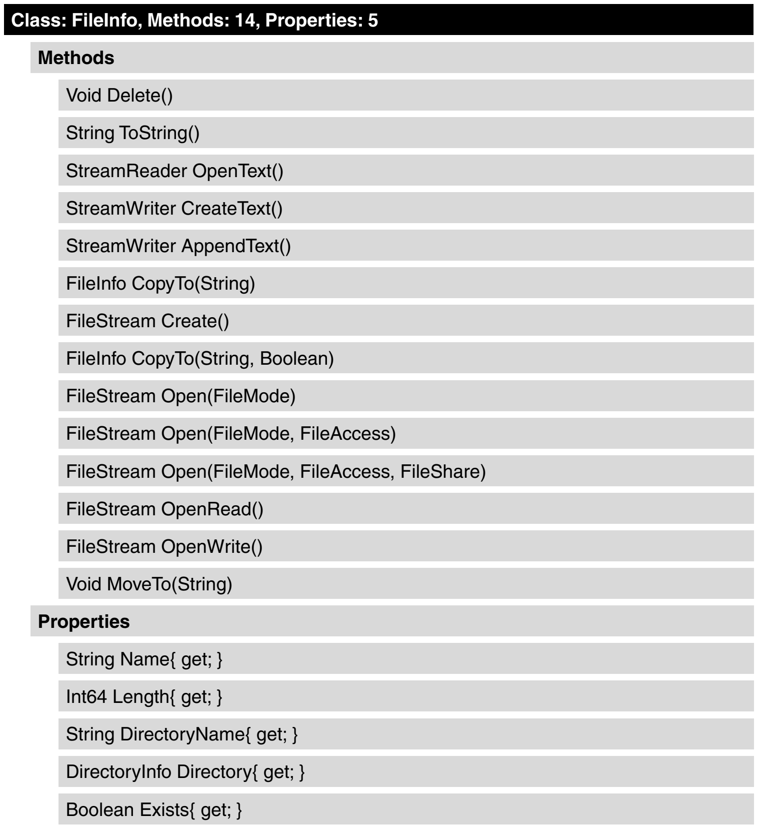
- String FullName{ get; }
- String Extension{ get; }
- String Name{ get; }
- Boolean Exists{ get; }
- DateTime CreationTime{ get; set; }
- DateTime LastAccessTime{ get; set; }
- DateTime LastWriteTime{ get; set; }
- FileAttributes Attributes{ get; set; }

FIGURE 10.5
The `FileSystemInfo` *class member map.*

A neat feature of the `DirectoryInfo` and `FileInfo` classes is that you can create objects with names of files or directories that don't yet exist. The constructors don't check for file existence, but they do check to make sure you have permissions to the file you're getting information for, and that you haven't specified an empty string or null value to the function (see Figure 10.6).

In addition to all the inherited properties from the `FileSystemInfo` class, the `FileInfo` class also has the `Director` property, which returns a `DirectoryInfo` object for the file's parent directory, and the `DirectoryName` property, which returns the string name of the current file's parent directory. You can get the size of the file using the `Length` property. This property can be null if the underlying file system doesn't hold information about the length of files. You'll remember I just mentioned that you can create these objects for files that don't yet exist. You can always test to see whether the file does exist by using the `Exists` property.

Nearly all the methods available on the `File` object you saw earlier in the chapter are also available on the `FileInfo` class. The only difference is that all the methods work on whatever file the `FileInfo` class refers to rather than requiring a file name to be passed in. Files that don't exist can be created with the `Create` method; you can copy files that do exist with the `CopyTo` method. All the `Open` methods from the `File` object are available, you can delete files with `Delete`, and you can move files with the `MoveTo` function. I hope all of these methods are sounding familiar because we covered them earlier in the chapter. There's no reason to go over the same functionality here. Just remember that these methods all work on the underlying file that the `FileInfo` object references and don't require you pass in an explicit path (see Figure 10.7).



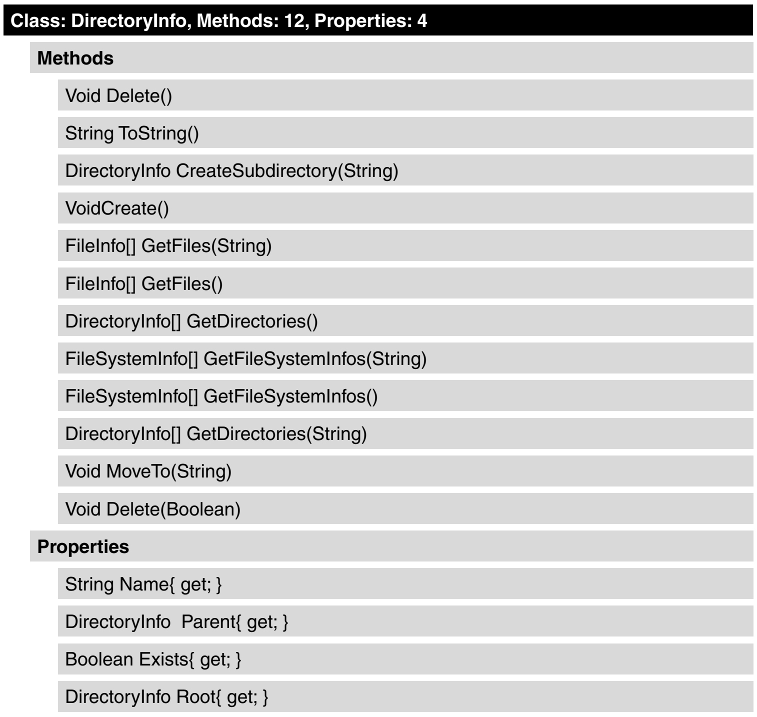
Class: FileInfo, Methods: 14, Properties: 5

- **Methods**
  - Void Delete()
  - String ToString()
  - StreamReader OpenText()
  - StreamWriter CreateText()
  - StreamWriter AppendText()
  - FileInfo CopyTo(String)
  - FileStream Create()
  - FileInfo CopyTo(String, Boolean)
  - FileStream Open(FileMode)
  - FileStream Open(FileMode, FileAccess)
  - FileStream Open(FileMode, FileAccess, FileShare)
  - FileStream OpenRead()
  - FileStream OpenWrite()
  - Void MoveTo(String)
- **Properties**
  - String Name{ get; }
  - Int64 Length{ get; }
  - String DirectoryName{ get; }
  - DirectoryInfo Directory{ get; }
  - Boolean Exists{ get; }

FIGURE 10.6
The `FileInfo` class member map.

The `DirectoryInfo` class is again nearly identical to the `Directory` class. New properties include the `Parent` property, which returns the `DirectoryInfo` for the directory's parent directory, and the `Root` property, which returns the `DirectoryInfo` for the root directory.

There are only a couple of new methods to point out for the `DirectoryInfo` class. The `CreateSubDirectory` method creates a subdirectory within the directory reference by the current `DirectoryInfo` class. You can delete directories using the `Delete` method. An overloaded version lets you pass in a Boolean parameter specifying whether to delete all subdirectories and files before deleting the directory. The normal `Delete` method will fail if the directory is not empty, whereas the overloaded method will succeed if you pass the value `true` to the method.



Class: DirectoryInfo, Methods: 12, Properties: 4

- **Methods**
  - Void Delete()
  - String ToString()
  - DirectoryInfo CreateSubdirectory(String)
  - VoidCreate()
  - FileInfo[] GetFiles(String)
  - FileInfo[] GetFiles()
  - DirectoryInfo[] GetDirectories()
  - FileSystemInfo[] GetFileSystemInfos(String)
  - FileSystemInfo[] GetFileSystemInfos()
  - DirectoryInfo[] GetDirectories(String)
  - Void MoveTo(String)
  - Void Delete(Boolean)
- **Properties**
  - String Name{ get; }
  - DirectoryInfo Parent{ get; }
  - Boolean Exists{ get; }
  - DirectoryInfo Root{ get; }

FIGURE 10.7
The `DirectoryInfoClass` *member map.*

That concludes the examination of the `FileInfo` and `DirectoryInfo` classes. Again the only major differences between `File/Directory` and `FileInfo/DirectoryInfo` is that the `Info` classes are rooted to the path you specify in the constructor for the classes, whereas the `File/Directory` classes support a full suite of static methods that take the path as a parameter.

Many of the methods we've already discussed return stream objects that we'll discuss next. Now you'll take a look at some examples of these objects in action after you learn about the `Stream` class and file IO in the next section.

The `Stream` Class and IO

The `Stream` class is the base class for any class that performs IO. This could be memory IO, disk IO, network IO—it doesn't matter what type of IO as long as it extends from the `Stream` class. We won't go into huge depth on how the `Stream` class works, but we will hit some of the more important functions and properties so that we can later understand some of the more important classes that derive from it.

Every stream object has some basic properties describing how the stream can be manipulated. These include `CanRead`, `CanSeek`, and `CanWrite`. With these properties you can easily determine what types of operations are allowed. For instance, in a network stream, you won't be able to seek through the data; you'll have to read it linearly from beginning to end, and `CanSeek` will be false. When using a file stream from a CD-ROM drive, you won't be able to write to the files because they are read-only. So `CanWrite` will be false. At the lowest level these properties tell you how to use the stream available to you. Whenever the `CanSeek` property is set, two additional properties become active. The `Length` property returns the full length of the stream and is obtained by seeking to the end and getting the `Position` property. The `Position` property is the current location in the stream that is being read from or written to.

CAUTION

If you try to access the `Length` and `Position` properties on a stream where `CanSeek` is false, you're going to end up throwing an exception. If you're unsure about the type of `Stream` that will be coming in, it's always best to use as much error-handling code as possible so that you can clean up any open connections and avoid memory and handle leaks.

With this in mind, the `Stream` class also has several useful methods. You can end a stream with the `Close` method. This effectively releases any stream resources, after which the object can't be used anymore. You can `Read`, `Write`, and `Seek` on the stream. All three of these functions have the capability to update the position variable as you move over the stream. When a stream is being written to, a `Flush` method is also available for forcing buffered data to be written to the underlying device. This could be out to file, out onto the network, or into its final location in memory. All other stream classes extend from this class and must support the methods. For instance, all the CLR readers and writers work on the base stream, but if you pass in a `FileStream` or a `NetworkStream`, they still function properly; they simply don't use any of the extended methods available to them.

The only derived class you should be interested in at this point is the `FileStream` class. All the methods you learned about on the `File` class to create and open files will always return a `FileStream` object. You will be passing this into any of the reader/writer classes we'll talk about next so that the reader/writer class can work on the underlying stream.

IO Read/Write Classes

That completes the discussion of the Stream object and how streams work at the base level. But how do you take advantage of those streams so that you can read and write information to them? Depending on the level of control you want over the output, you'll pick either the StreamReader/StreamWriter classes, which work with textual information, or the BinaryReader/BinaryWriter classes, which work with byte information.

We'll start by examining the StreamReader class, which is capable of reading standard textual streams. You create a StreamReader by passing the underlying Stream object to the constructor. This initializes a new instance of the reader class to work with our file, network, or memory stream. We could additionally specify an encoding for the underlying stream. Examples of different encodings are ASCII, UTF-8, and UTF-16. Depending on the target audience for your application, you'll want to make a decision early on as to what format you'll be using to transfer data. ASCII is fine for our discussion, though, and it is the easiest to work with, requiring no special conversion work. Sometimes encodings are written into the beginning of the stream. This is certainly the case for Unicode files, which can contain a byte order mark. The StreamReader supports an overloaded constructor that takes a Boolean value as the parameter. This Boolean value specifies whether to auto-detect the encoding of the file. Let's go ahead and enumerate some of these constructors with some code:

```
// Read a file using the default encoding UTF-8
var defaultReader:StreamReader = new StreamReader(mystream);

// Read a file using the ASCII encoding
var asciiReader:StreamReader = new StreamReader(mystream, Encoding.ASCII);

// Auto-detect using the byte order marks
var autoReader:StreamReader = new StreamReader(mystream, true);

// Auto-detect, but specify an encoding in case one isn't in the file
var autoBack:StreamReader = new StreamReader(mystream, Encoding.UTF8, true);
```

After you've created a StreamReader, you can always find out which encoding was used for the creation by examining the CurrentEncoding property. The base stream object, in the listing above mystream, can always be retrieved by using the BaseStream property. Each of these can come in handy if you are passed a StreamReader object and need to discover the underlying source. For instance, you might handle things differently in your function if you're working with a NetworkStream versus a FileStream. The following code demonstrates how code can determine the type of the StreamReader.BaseStream property:

```
if ( myStreamReader.BaseStream instanceof FileStream ) {
    // Do some file stream work
} else if ( myStreamReader.BaseStream instanceof NetworkStream) {
    // Do some network stream work
}
```

TIP

You'll want to discover the strange properties of the various streams early on in your learning of the file IO classes. Learning how each of the different streams behaves helps you write more efficient code depending on the type of IO you're performing rather than writing simple generic routines that might cost you in performance later.

The `StreamReader` Class

The `StreamReader` has the `Peek`, `Read`, `ReadLine`, and `ReadToEnd` methods for obtaining data from the stream. You can start by using the `Peek` command to find out whether there is data to be read. This method returns the next character to be read and `-1` if there are no more characters, or if the stream doesn't support seeking operations. In the preceding code listing you could have used `Peek` to read from the `FileStream`; but `NetworkStreams` aren't seekable and you would have had to use other methods of checking whether data were available.

The `Read` method can be used to simply read the next character, in which case it returns the next character as an integer, or `-1` if no more characters are available. So if you wanted to check whether characters were available on a `NetworkStream`, you'd have to use the `Read` method, checking for `-1` and being sure to pass the returned character to your processing functions so that you wouldn't lose the character. The `Read` method has a special overload for reading a byte array instead of simply a byte. The first parameter to this overload is a byte array to hold the data being read. The second parameter should contain the offset into the byte array where the first piece of data will be stored, and the third parameter should be the total number of bytes to read into the array. The return value will be the total number of characters read. This can be `0` if you are at the end of the stream, so it's important to check this value.

All of these methods are great for working with characters or streams that are still being built while your program is running (a `NetworkStream`, for instance, is a constant stream of characters and works well with characters, but a `FileStream` is probably lines of information or lines of records). For working with information in a more human-laid-out format, you can use the `ReadLine` method. `ReadLine` returns a string given the next line of input from the file. It reads to the next newline and returns all characters between the current position and the newline, omitting the newline from the string. When no more data is available, this method returns null.

ReadLine is great when your data is laid out with one piece of data per line or when it just makes more sense to work with the input on a per-line basis than to work with character arrays. The fact that it returns a string instead of a character array also eases processing later when the value is going to be used. When you're working with small files and you need to work with the entire file at once, you can also use the ReadToEnd method, which reads from the current stream position all the way to the end of the stream, returning all data, including newlines, as a string. This will come in handy later in Chapter 12, "Regular Expressions," when you're working with regular expressions and you're trying to match a pattern that spans multiple lines of data.

When you're done with a StreamReader, you always want to close it out using the Close method. This effectively closes the underlying BaseStream as well and releases any system resources that are held open by the operation. One thing to note is that after a Reader/Writer class is bound to a Stream it can't be detached. This means that you can't use the same stream object to create multiple readers.

The StreamWriter Class

The StreamReader is responsible only for reading information out of a file, but what if you want to write to the file? For that there is the StreamWriter class. The StreamWriter is created identically to the StreamReader class, so if you've forgotten the constructors already, you can read the preceding section.

The StreamWriter has the BaseStream property and an Encoding property that is identical to the CurrentEncoding property of the StreamReader seen previously. In addition, there is an AutoFlush property. When many types of IO are being performed, any data written to a stream is queued in an internal buffer. For the data to be written down to the stream, either the buffer must become full or an explicit call to the Flush method must be made. The AutoFlush property overcomes the necessity to call the Flush method by automatically calling Flush every time you write to the stream.

The AutoFlush property is really bad for performance on streams for slow devices such as floppy drives and magnetic tape drives, or on network streams that perform some network IO every time the buffer is flushed. Another thing to note is that Flush is called whenever the StreamWriter is closed, so in most cases the programmer won't have to call the Flush method or set the AutoFlush property.

The StreamWriter also shares some methods with the StreamReader class. The Close method releases the underlying stream resources and flushes any data to the underlying medium. The Flush method is new and allows you to directly control when data is written to disk. Most often you'll pick a time when it won't affect program performance and then call Flush, or you'll do some extensive stream work and call Flush after each logical step. But to flush your buffers, you'll have to fill them with something.

That's the responsibility of the `Write` method. The `Write` method can take several forms. It can write a single character, an array of characters, or a string. Optionally you can specify the starting offset and the number of bytes to write for the array of characters. For the most part, you'll just be writing strings of characters and won't have to worry about dealing with character arrays. Note at this point that there aren't any methods for writing a line of data, so you'll have to write the newlines into the stream yourself.

To gain a bit more control over stream data, you'll have to use the `BinaryReader` and `BinaryWriter` classes. These classes each have many additional methods for manipulating the underlying stream and retrieving data in many more formats than just strings or characters.

`BinaryReader` and `BinaryWriter` share constructors just as the `StreamWriter` and `StreamReader` classes. However, with more control over the underlying stream, we also lose some ease of use. The classes have only two constructors: one that takes the `Stream` object to be used, and one that takes both the `Stream` object and an `Encoding` for the underlying data. The `Encoding` is used only when reading or writing string data, single characters, and arrays of characters. Because there aren't any overloads that take a filename, you'll always have to open the `Stream` first because these classes won't open it for you.

The `BinaryReader` has some methods for controlling the underlying stream we haven't seen before in the `StreamReader`. The first to note is the `FillBuffer` method, which is passed an integer value containing the number of bytes to read into the internal buffer. If the end of the stream is reached before the specified number of bytes is read, an `EndOfStreamException` is thrown. By filling the internal buffer, you can quickly perform read operations on the stream without having to access the underlying medium. For the most part, this won't be necessary and will be used only in special cases in which computationally intensive work is being done on the stream.

The `Peek` method has taken on a new name in the `BinaryReader` as `PeekChar`, and it again returns the next available character, or `-1` if none is available. The `PeekChar` method is crucial when accessing binary files and testing the end of the stream so that you don't throw exceptions. The `Read` method is also back and behaving similarly to the `Read` method of the `StreamReader`. There is an additional overload available to the `BinaryReader` in the form of reading into a byte array. For this overload the `Read` method is called with the byte array to be read into, an offset, and the number of characters to read. The method returns the number of bytes read, which might be `0` if the end of the stream is reached. There are additional atomic methods that are easier to use as long as you're working with the base data types.

Before we continue with the full set of Read methods, I'd like to point out that each of these other methods throws an `EndOfStreamException` if no additional data is available, or if the method you are using requires more data than what is available in the stream. Because it is rather difficult to determine whether the stream contains data valid for a call, in most cases you'll want to just eat the exception and program accordingly. Listing 10.3 shows you an example of properly handling this exception when we make use of the `BinaryReader` and `BinaryWriter` classes.

The `BinaryReader` has a method for all the CLR base data types. For the most part, all the methods follow the pattern of `ReadDataType`, and instead of passing in the structure to hold the return contents, they return the datatype filled with the value from the underlying stream. We'll run through these read methods fairly quickly because they are rather trivial to use. The first method, `ReadBoolean`, reads a single `Boolean` and updates the stream by moving forward one byte.

`ReadByte` reads a single `Byte` and updates the stream by moving forward one byte, whereas `ReadBytes` takes the number of `Byte`'s to read and returns an array. The `Read` method is probably more useful for this operation unless you know that a specified number of bytes are located in the file.

`ReadChar` reads a single `Char` in the `Encoding` format specified in the constructor and updates the underlying stream however many bytes were necessary to consume the character. On some character sets this can differ between character reads, so you should never bank on the stream moving ahead a set number of bytes. `ReadChars` reads a specified number of `Chars` as a `Char` array and updates the underlying stream accordingly.

The numeric functions are many, and each reads in as many bytes as required by their datatype. They consist of the `ReadDecimal`, `ReadDouble`, `ReadInt16`, `ReadInt32`, `ReadInt64`, `ReadSByte`, `ReadSingle`, `ReadUInt16`, `ReadUInt32`, and `ReadUInt64`. Again, depending on the datatype, these can update the underlying stream by 1 byte up to 16 bytes for the `Decimal` datatype.

Of course, the familiar `ReadString` is still available and reads a specially formatted string from the file. The string must be prefixed by a length in the seven-bit encoded integer format. This way the size of the string can be read from the file without mistaking any of the string's characters for length information, or any of the length information for characters. The string is read in using the current `Encoding` format.

This seven-bit encoded integer is intriguing because it is an Int32 written in compressed format. What happens is that the first seven bits of the number are written to the stream. If the number is larger than seven bits, the eight bit of the value written to the stream is set, the value being written is shifted left seven bits, and the process starts again. This means that a four-byte integer can be written to the stream in as few as one byte, or as many as five bytes. So in the case of large numbers, there isn't any saved space and actually some space might be wasted; but in the case of encoding string lengths we saw earlier, we actually save quite a bit of space because most strings are short in length.

TIP

If you're interested in using something like the seven-bit encoded integer in your own applications, you'll have to write the function yourself. This is rather unfortunate because this function makes writing out custom data much easier. For instance, you could write a seven-bit encoded integer that specifies the size of the following byte array, rather than having to hard-code the length in code or use a large data type to specify the length on the stream.

Listing 10.3 makes use of all the `Stream` features we've discussed so far. The example reads and writes a textual file using the `StreamReader` and `StreamWriter` classes. Then we will read and write a binary file using the `BinaryReader` and `BinaryWriter` classes. When performing the `BinaryRead`, we'll be sure to read past the end of the stream so that we can cause and catch an exception.

LISTING 10.3 Usage of the Stream Manipulation Classes

```
import System;
import System.IO;
import System.Globalization;
import System.Text;

public class FileReaderWriter {
    public static function StreamWriterFile() : void {
        var fs:FileStream =
            new FileStream("StreamFile.txt", FileMode.Create);
        var sw:StreamWriter = new StreamWriter(fs);

        sw.Write('A'); // Write Character
        sw.Write(Char[](['A', 'B', 'C', 'D', 'E', '\n']));
        sw.Write("Hello World");

        sw.Close();
    }

    public static function StreamReaderFile() : void {
        var sr:StreamReader = new StreamReader("StreamFile.txt");

        Console.WriteLine(sr.Read());
        Console.WriteLine(sr.ReadLine());
        Console.WriteLine(sr.ReadLine());
        Console.WriteLine();
```

Listing 10.3 continued

```
        sr.Close();
    }

    public static function BinaryWriterFile() : void {
        var fs:FileStream = new FileStream("BinFile.bin", FileMode.Create);
        var bw:BinaryWriter = new BinaryWriter(fs, Encoding.UTF8);

        bw.Write(true); // Boolean
        bw.Write(Byte(128)); // Byte
        bw.Write(Byte[]([128,127,126])); // Byte Array
        bw.Write('A'); // Char
        bw.Write(Char[](['A', 'B', 'C', 'D', 'E'])); // Char Array
        bw.Write(Decimal(1.27643));
        bw.Write(Double(1.27643));
        bw.Write(Int16(21));
        bw.Write(Int32(21));
        bw.Write(Int64(21));
        bw.Write(SByte(21));
        bw.Write(Single(1.27643));
        bw.Write("Hello World");
        bw.Write(UInt16(21));
        bw.Write(UInt32(21));
        bw.Write(UInt64(21));

        bw.Close();
    }

    public static function BinaryReaderFile() : void {
        var fs:FileStream = new FileStream("BinFile.bin", FileMode.Open);
        var br:BinaryReader = new BinaryReader(fs, Encoding.UTF8);

        Console.WriteLine(br.ReadBoolean());
        Console.WriteLine(br.ReadByte());
        Console.WriteLine(br.ReadBytes(4).toString());
        Console.WriteLine(br.ReadChar());
        Console.WriteLine(br.ReadChars(5).toString());
        Console.WriteLine(br.ReadDecimal());
        Console.WriteLine(br.ReadDouble());
        Console.WriteLine(br.ReadInt16());
        Console.WriteLine(br.ReadInt32());
        Console.WriteLine(br.ReadInt64());
        Console.WriteLine(br.ReadSByte());
        Console.WriteLine(br.ReadSingle());
        Console.WriteLine(br.ReadString());
```

LISTING 10.3 continued

```
        Console.WriteLine(br.ReadUInt16());
        Console.WriteLine(br.ReadUInt32());
        Console.WriteLine(br.ReadUInt64());
        Console.WriteLine();

        try {
            br.ReadByte();
        } catch(e:EndOfStreamException) {
            Console.WriteLine("We read past the end of the stream");
        }

        br.Close();
    }
}

var Args:String[] = Environment.GetCommandLineArgs();

FileReaderWriter.StreamWriterFile();
FileReaderWriter.StreamReaderFile();

FileReaderWriter.BinaryWriterFile();
FileReaderWriter.BinaryReaderFile();
```

Summary

You should have all the tools you need to perform most of the standard file IO operations your programs will require. We've touched on nearly every aspect of the System.IO namespace. We began with the `Path` object, used to generate cross-platform paths. We found that nearly any file-system operation from concatenating directory names to ripping out file and extension information can be performed using this class.

You then discovered how to manipulate files and directories using two different sets of classes. The `File` and `Directory` classes are full of static methods that work with files and directories on a one-time basis. Then you saw the `FileInfo` and `DirectoryInfo` classes, which can be used to represent a file and can be used many times to perform different sets of operations using the path multiple times.

From here we worked our way to manipulating the contents of the files using the abstract `Stream` class. In this chapter we discussed mainly the `FileStream` class, but many of the same principles will apply in the next chapter when we move on to the `NetworkStream` class and perform network IO. You then discovered that you could create special classes on top of the

`Stream` class to perform various read and write operations. Specifically, we investigated reading and writing textual information using the `StreamReader` and `StreamWriter` classes. You also learned to use the `BinaryReader` and `BinaryWriter` classes for manipulating binary data.

In the next chapter we'll be moving on to the XML persistence format. We'll discuss everything from loading an XML document and navigating the hierarchy, to manipulating the data to create our own configuration files, and finally how to use XSLT to transform our XML from one format to another.

CHAPTER

11

XML

IN THIS CHAPTER

The introduction of eXtensibe Markup Language (XML) has revolutionized how hierarchical data is stored and will continue to shape the way programs are developed in the future. You can easily incorporate each of the application programming interfaces (APIs) you'll discover in this chapter into programs so that you can use XML as a data format rather than coding a custom format.

We'll start this chapter by examining the `System.Xml` namespace and how to load XML content using various XML classes. Each of the classes in the `System.Xml` namespace provides a unique set of access APIs, depending on how you want to load and read the XML document in question. Next, we'll talk about how to enumerate the various elements and attributes that exist in the XML document. Finally, we'll talk about how to write new XML content.

Make sure to read all the tips in this chapter with extra care because they point out lots of tricks and suggestions that should help you spend less time on file input/output and more time to application features.

The `System.Xml` Namespace

The `System.Xml` namespace houses all the XML-related classes for the common language runtime (CLR). These classes are capable of loading and manipulating existing XML documents and writing new XML document content. There are a number of methods for reading and writing XML. The classes in the `System.Xml` namespace provide methods for reading XML in a tree or hierarchical form. There is also a series of classes for reading and writing data in a linear format, starting at the beginning of an XML document and reading or writing straight through to the end.

The following are the classes that will be discussed in this section (see Figure 11.1):

- **`XmlDocument`**—The `XmlDocument` class is used to load XML content into a hierarchical or tree format. Each element in this tree is represented by an `XmlNode` class and is the most basic class for accessing XML data in an `XmlDocument` class.
- **`XmlTextReader`**—The `XmlTextReader` class is used to read through an XML document in a linear manner. This class reads through the document from the beginning to the end and does not take into account the tree-like format of most XML documents.
- **`XmlTextWriter`**—The `XmlTextWriter` class is used to write an XML document. This class is not capable of building an XML document tree through the creation of `XmlNode` elements like the `XmlDocument` class, but it is extremely quick at emitting XML content if all the data is available to be written at the same time.

Namespace: System.Xml, TotalClasses: 52

IHasXmlNode	EntityHandling
IXmlLineInfo	XmlNameTable
NameTable	ReadState
ValidationType	WhitespaceHandling
XmlNode	XmlAttribute
XmlNamedNodeMap	XmlAttributeCollection
XmlLinkedNode	XmlCharacterData
XmlCDataSection	XmlNodeList
XmlComment	XmlConvert
XmlDeclaration	XmlDocument
XmlDocumentFragment	XmlDocumentType
XmlWriter	XmlTextWriter
XmlElement	XmlEntity
XmlReader	XmlTextReader
XmlEntityReference	XmlNodeChangedAction
XmlException	XmlImplementation
XmlNamespaceManager	XmlNodeChangedEventArgs
XmlNodeChangedEventHandler	XmlNodeOrder
XmlNodeReader	XmlNodeType
XmlNotation	XmlParserContext
XmlProcessingInstruction	XmlQualifiedName
XmlResolver	XmlSignificantWhitespace
XmlSpace	XmlText
XmlTokenizedType	XmlUrlResolver
XmlValidatingReader	XmlWhitespace
Formatting	WriteState

FIGURE 11.1
Classes of the `System.Xml` *namespace.*

When you use the `XmlDocument` class, you can use a number of support classes to help create the tree. First, every element in the tree is represented by the `XmlNode` class. `XmlDocument` can even extend the `XmlNode` class to gain the base functionality to become the root node of the document tree. Some of the other classes we'll discuss also extend from the `XmlNode` class and represent such items as XML attributes (that is, the `XmlAttribute` class) and XML elements (that is, the `XmlElement` class). In addition, several collection classes are necessary for iterating over the `XmlDocument` class structure, including the `XmlNodeList` and `XmlAttributeCollection` classes.

When you're comfortable working with ready-built XML documents, you can move on to writing new XML content, using two approaches:

- *Update existing content*—You can add and remove items from the `XmlDocument` class directly and then save the new content to a file. This is the easiest approach because it is object oriented and involves simply creating and adding nodes.
- *Write entirely new documents*—You can start with either a blank `XmlDocument` class or you can make use of the `XmlTextWriter` class. Using the `XmlTextWriter` class is much faster and requires less code for writing new documents. However, it has the shortcoming of not being strongly typed or validating (that is, it can only write string values and it doesn't check your data against a schema).

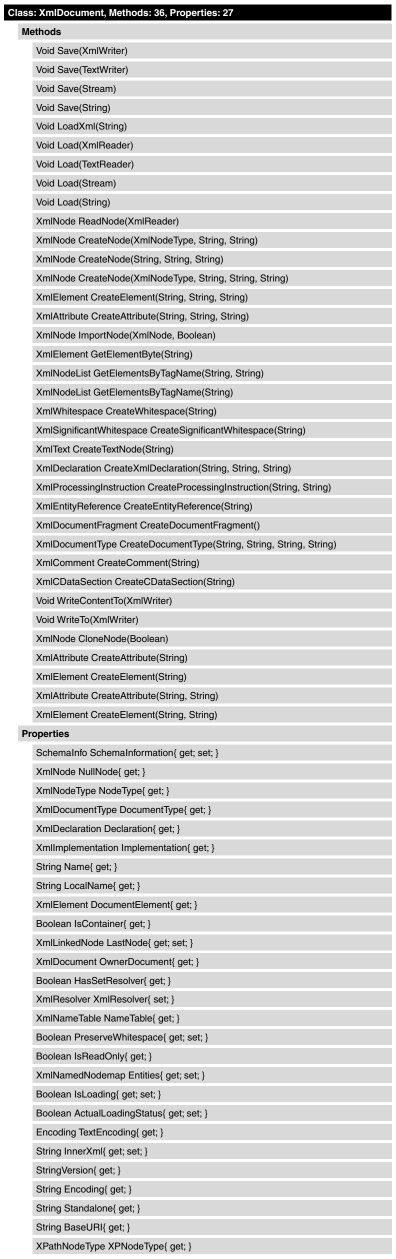
The remainder of this section is dedicated to using `XmlDocument` and `XmlTextReader` to load XML content and enumerate XML content.

Loading XML Content

To load XML documents, you can use several classes in the `System.Xml` namespace. Each class provides different APIs for accessing a loaded document and provides some restrictions on the use of the document. The most widely used class is the `XmlDocument` class because it provides an in-memory tree structure of the document that you can quickly and easily navigate. When an XML document is rather large, the `XmlDocument` class often can't be used because it consumes too much memory. In this case, you can use the `XmlTextReader` class, which provides a stream-style cursor that works with the file on disk rather than working with the file by using in-memory objects. If you only need a couple of the values in an XML document or are trying to process a document into a private format, this is definitely the method to use.

The `XmlDocument` class is shown in its entirety in Figure 11.2. Not all the class members shown are crucial in using the `XmlDocument` class, so some of them are not discussed in this chapter. Some of the class members are discussed in later sections of this chapter.

The `XmlDocument` class has several constructor implementations. However, none of the advanced implementations are useful for generic XML processing. This leaves the default constructor, which all of the following examples use.



```
Class: XmlDocument, Methods: 36, Properties: 27
  Methods
    Void Save(XmlWriter)
    Void Save(TextWriter)
    Void Save(Stream)
    Void Save(String)
    Void LoadXml(String)
    Void Load(XmlReader)
    Void Load(TextReader)
    Void Load(Stream)
    Void Load(String)
    XmlNode ReadNode(XmlReader)
    XmlNode CreateNode(XmlNodeType, String, String)
    XmlNode CreateNode(String, String, String)
    XmlNode CreateNode(XmlNodeType, String, String, String)
    XmlElement CreateElement(String, String, String)
    XmlAttribute CreateAttribute(String, String, String)
    XmlNode ImportNode(XmlNode, Boolean)
    XmlElement GetElementByte(String)
    XmlNodeList GetElementsByTagName(String, String)
    XmlNodeList GetElementsByTagName(String)
    XmlWhitespace CreateWhitespace(String)
    XmlSignificantWhitespace CreateSignificantWhitespace(String)
    XmlText CreateTextNode(String)
    XmlDeclaration CreateXmlDeclaration(String, String, String)
    XmlProcessingInstruction CreateProcessingInstruction(String, String)
    XmlEntityReference CreateEntityReference(String)
    XmlDocumentFragment CreateDocumentFragment()
    XmlDocumentType CreateDocumentType(String, String, String, String)
    XmlComment CreateComment(String)
    XmlCDataSection CreateCDataSection(String)
    Void WriteContentTo(XmlWriter)
    Void WriteTo(XmlWriter)
    XmlNode CloneNode(Boolean)
    XmlAttribute CreateAttribute(String)
    XmlElement CreateElement(String)
    XmlAttribute CreateAttribute(String, String)
    XmlElement CreateElement(String, String)
  Properties
    SchemaInfo SchemaInformation{ get; set; }
    XmlNode NullNode{ get; }
    XmlNodeType NodeType{ get; }
    XmlDocumentType DocumentType{ get; }
    XmlDeclaration Declaration{ get; }
    XmlImplementation Implementation{ get; }
    String Name{ get; }
    String LocalName{ get; }
    XmlElement DocumentElement{ get; }
    Boolean IsContainer{ get; }
    XmlLinkedNode LastNode{ get; set; }
    XmlDocument OwnerDocument{ get; }
    Boolean HasSetResolver{ get; }
    XmlResolver XmlResolver{ set; }
    XmlNameTable NameTable{ get; }
    Boolean PreserveWhitespace{ get; set; }
    Boolean IsReadOnly{ get; }
    XmlNamedNodemap Entities{ get; set; }
    Boolean IsLoading{ get; set; }
    Boolean ActualLoadingStatus{ get; set; }
    Encoding TextEncoding{ get; }
    String InnerXml{ get; set; }
    StringVersion{ get; }
    String Encoding{ get; }
    String Standalone{ get; }
    String BaseURI{ get; }
    XPathNodeType XPNodeType{ get; }
```

FIGURE 11.2

The structure of the `XmlDocument` *class.*

When an `XmlDocument` object is instantiated, it is completely empty. At this point, you can add some XML code by using the `Load` method. The `Load` method takes several forms:

- It can take a `Stream` object, which represents some file or network connection (`XmlDocument.Load(file:Stream)`).
- It can take a `String` object that contains the filename of an XML document (`XmlDocument.Load(fileName:String)`).
- It can take a `TextReader` object, which you know from Chapter 10, "File Access," can be a `StreamReader` or some similar class (`XmlDocument.Load(tr:TextReader)`).
- It can take an `XmlReader` object (`XmlDocument.Load(xr:XmlReader)`).

All these function overloads are helpful, but in this section we'll be using the `String` overload to point to a file on disk.

The `XmlDocument` class also has a `LoadXml` method, which can be used to load a `String` object that contains some XML content. This can be useful in a program that builds some XML content and then needs to load it into an `XmlDocument` class so that the generated XML content can be accessed by your code. It can also be used when the XML content has to be processed before it can be loaded into the `XmlDocument` class. In this case, the default `Load` methods do not work because the XML content they are loading from disk is in an invalid format until the program has processed it for the first time. Listing 11.1 demonstrates the use of the `Load` methods and the `LoadXml` method to load content into an `XmlDocument` object.

LISTING 11.1 Using the `Load` and `LoadXml` Methods on XML Code

```
import System;
import System.Xml;
import System.IO;

public class XmlContent {
    public static function GetXmlFromFile(file:String) : XmlDocument {
        var xDoc:XmlDocument = new XmlDocument();
        xDoc.Load(file);
        return xDoc;
    }

    public static function GetXmlFromStream(file:Stream) : XmlDocument {
        var xDoc:XmlDocument = new XmlDocument();
        xDoc.Load(file);
        return xDoc;
    }
```

LISTING 11.1 continued

```
    public static function
    GetXmlFromTextReader(file:TextReader) : XmlDocument {
        var xDoc:XmlDocument = new XmlDocument();
        xDoc.Load(file);
        return xDoc;
    }

    public static function
    GetXmlFromXmlTextReader(file:XmlTextReader) : XmlDocument {
        var xDoc:XmlDocument = new XmlDocument();
        xDoc.Load(file);
        return xDoc;
    }

    public static function GetXmlFromString(xml:String) : XmlDocument {
        var xDoc:XmlDocument = new XmlDocument();
        xDoc.LoadXml(xml);
        return xDoc;
    }
}

// Here is just wants the name of the file
var xDoc1:XmlDocument = XmlContent.GetXmlFromFile("11-1.xml");

// Pass it in a FileStream which is derived from Stream
var fs1:FileStream = File.Open("11-1.xml", FileMode.Open)
var xDoc2:XmlDocument = XmlContent.GetXmlFromStream(fs1);
fs1.Close();

// We know that a StreamReader is derived from TextReader.  So this works.
var sr1:StreamReader = new StreamReader("11-1.xml");
var xDoc3:XmlDocument = XmlContent.GetXmlFromTextReader(sr1);
sr1.Close();

// XmlTextReader is derived from XmlReader
var xDoc4:XmlDocument =
    XmlContent.GetXmlFromXmlTextReader(new XmlTextReader("11-1.xml"));

// Finally, the document needs to be read into a string
var sr2:StreamReader = new StreamReader("11-1.xml");
var xml:String = sr2.ReadToEnd();
sr2.Close();

var xDoc5:XmlDocument = XmlContent.GetXmlFromString(xml);
```

Listing 11.1 loads five different XmlDocument objects with the contents of the 11-1.xml file. As you can see, this is pretty easy. How much harder is it to use the XmlTextReader class?

The XmlTextReader class is a bit easier to initialize than the XmlDocument class because it provides constructors that enable immediate initialization of the class without having to call support methods such as the Load and LoadXml methods of the XmlDocument class. These constructors enable you to initialize the XmlTextReader object with a single line of code. Three of these constructors, the Stream, String, and TextReader versions, should suit the requirements of a majority of programs. The following list examines each of these three constructors (see Figure 11.3):

- The Stream overload takes a FileStream class or NetworkStream class, similar to the XmlDocument class.
- The String overload takes a uniform resource locator (URL) to the XML document. This URL can either be a file URL or an Internet URL, in which case the document is downloaded from a Web server somewhere on the Internet.
- A TextReader class can also be supplied, in which case a StreamReader class can be used to load the content in question.

These are the most basic of the XmlTextReader overloads. The rest of the XmlTextReader class constructors are quite complex, and you use them only when you need a high degree of control over the XML content being loaded.

The following example shows how easy it is to load XML content into an XmlTextReader class:

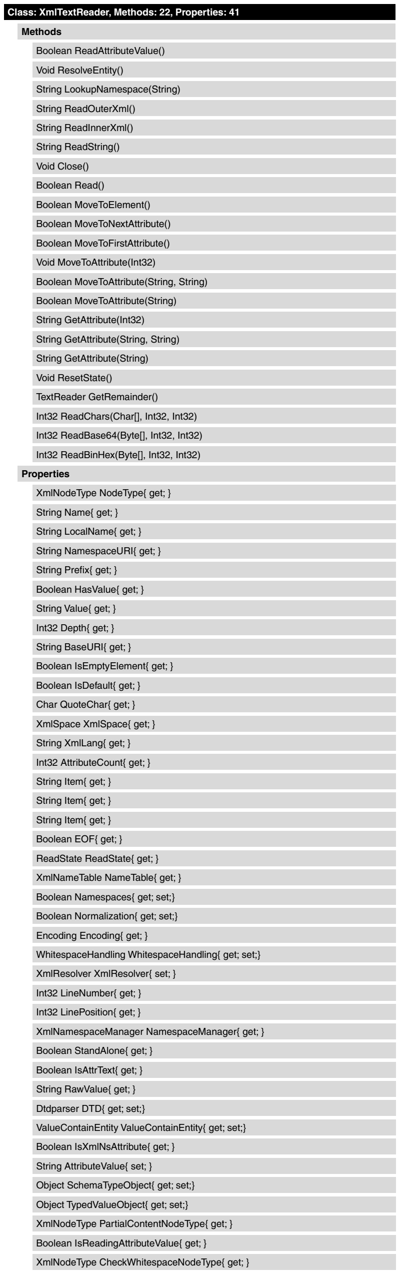
```
import System;
import System.IO;
import System.Xml;

// Using the XML from Code Listing 11.1 as a File URL
var xtr1:XmlTextReader = new XmlTextReader("11-1.xml");

// Using an Internet URL
var xtr2:XmlTextReader = new XmlTextReader("http://www.somexml.org/some.xml");

// Using the Stream overload
var fs1:FileStream = File.Open("11-1.xml", FileMode.Open);
var xtr3:XmlTextReader = new XmlTextReader(fs1);
fs1.Close();

// Using the TextReader overload
var sr1:StreamReader = new StreamReader("11-1.xml");
var xtr4:XmlTextReader = new XmlTextReader(sr1);
sr1.Close();
```



Class: XmlTextReader, Methods: 22, Properties: 41

Methods

- Boolean ReadAttributeValue()
- Void ResolveEntity()
- String LookupNamespace(String)
- String ReadOuterXml()
- String ReadInnerXml()
- String ReadString()
- Void Close()
- Boolean Read()
- Boolean MoveToElement()
- Boolean MoveToNextAttribute()
- Boolean MoveToFirstAttribute()
- Void MoveToAttribute(Int32)
- Boolean MoveToAttribute(String, String)
- Boolean MoveToAttribute(String)
- String GetAttribute(Int32)
- String GetAttribute(String, String)
- String GetAttribute(String)
- Void ResetState()
- TextReader GetRemainder()
- Int32 ReadChars(Char[], Int32, Int32)
- Int32 ReadBase64(Byte[], Int32, Int32)
- Int32 ReadBinHex(Byte[], Int32, Int32)

Properties

- XmlNodeType NodeType{ get; }
- String Name{ get; }
- String LocalName{ get; }
- String NamespaceURI{ get; }
- String Prefix{ get; }
- Boolean HasValue{ get; }
- String Value{ get; }
- Int32 Depth{ get; }
- String BaseURI{ get; }
- Boolean IsEmptyElement{ get; }
- Boolean IsDefault{ get; }
- Char QuoteChar{ get; }
- XmlSpace XmlSpace{ get; }
- String XmlLang{ get; }
- Int32 AttributeCount{ get; }
- String Item{ get; }
- String Item{ get; }
- String Item{ get; }
- Boolean EOF{ get; }
- ReadState ReadState{ get; }
- XmlNameTable NameTable{ get; }
- Boolean Namespaces{ get; set;}
- Boolean Normalization{ get; set;}
- Encoding Encoding{ get; }
- WhitespaceHandling WhitespaceHandling{ get; set;}
- XmlResolver XmlResolver{ set; }
- Int32 LineNumber{ get; }
- Int32 LinePosition{ get; }
- XmlNamespaceManager NamespaceManager{ get; }
- Boolean StandAlone{ get; }
- Boolean IsAttrText{ get; }
- String RawValue{ get; }
- Dtdparser DTD{ get; set;}
- ValueContainEntity ValueContainEntity{ get; set;}
- Boolean IsXmlNsAttribute{ get; }
- String AttributeValue{ set; }
- Object SchemaTypeObject{ get; set;}
- Object TypedValueObject{ get; set;}
- XmlNodeType PartialContentNodeType{ get; }
- Boolean IsReadingAttributeValue{ get; }
- XmlNodeType CheckWhitespaceNodeType{ get; }

FIGURE 11.3

The structure of the `XmlTextReader` *class.*

Just loading a document doesn't allow for much control in a program. In the following section we'll look at some of the ways you can investigate the various elements and attributes.

Enumerating Elements

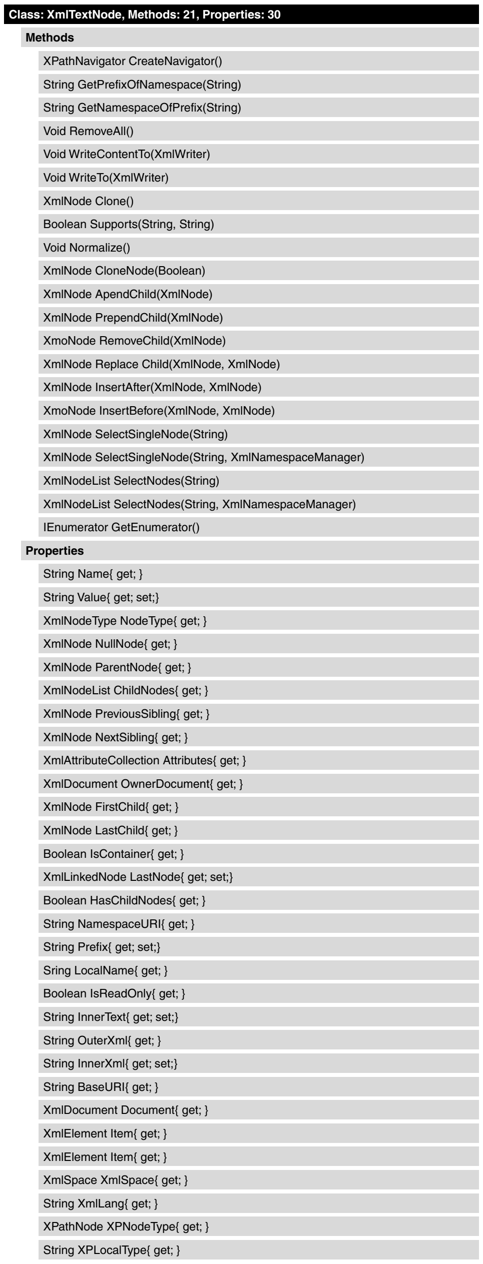
No matter how an XML document is loaded, you have to enumerate and examine its attributes and elements to get at the data in the document. So far we have discussed two types of layout structures for these documents. The `XmlDocument` class uses a tree structure in which collections can be used to get elements and attributes that are grouped together. The `XmlTextReader` class uses a linear layout in which the only way to examine ownership of one element or attribute is to examine properties on the `XmlTextReader` class that are updated as the document is parsed. Because the `XmlDocument` class is easy to traverse, it is discussed here first. Refer to Figure 11.2 at any time to take a look at its properties and methods.

Nearly every XML document contains a root element that contains the data-related portion of the document. To find the root element, you use the `DocumentElement` property, which returns `null` if no document root element exists or returns an `XmlNode` element that represents the document root element if there is a root element. Examination of the XML content within an `XmlDocument` class begins with the root element for all the examples in this chapter.

With an `XmlNode` element at the root, you can easily examine the rest of a document by recursively stepping through child element and attribute objects. In Figure 11.4, you can see that the `XmlNode` class has quite a few properties and methods for investigating the state of the current `XmlNode` element and its children.

Let's begin by examining the state of the current node. You must first find out the current node's type, which you can do by using the `NodeType` property. Each node in an XML document can be one of a number of types, such as an attribute, a comment, a document, or an element. For a full list of the available `XmlNodeType` types, see Figure 11.5. Depending on the type of the node, various properties and method are available. When a property or method isn't available, `null` is returned, so it is easy to determine whether a certain type of node makes use of a certain property.

When the program finds the type of the node, it can make a decision about how to parse that type of node. There are generic methods for parsing all `XmlNode` objects identically, but you often don't want this functionality. Generally, you want a program to ignore most types of nodes and parse only the most prominent nodes. Therefore, in this section we'll discuss the parsing of element nodes, text nodes, and attribute nodes only.



Class: XmlTextNode, Methods: 21, Properties: 30

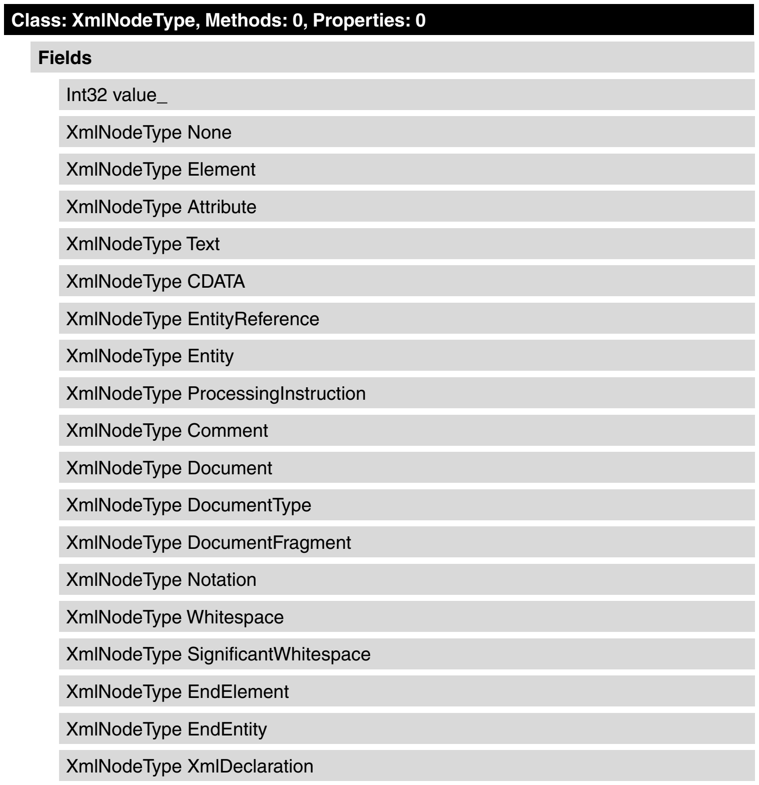
Methods

XPathNavigator CreateNavigator()
String GetPrefixOfNamespace(String)
String GetNamespaceOfPrefix(String)
Void RemoveAll()
Void WriteContentTo(XmlWriter)
Void WriteTo(XmlWriter)
XmlNode Clone()
Boolean Supports(String, String)
Void Normalize()
XmlNode CloneNode(Boolean)
XmlNode ApendChild(XmlNode)
XmlNode PrependChild(XmlNode)
XmoNode RemoveChild(XmlNode)
XmlNode Replace Child(XmlNode, XmlNode)
XmlNode InsertAfter(XmlNode, XmlNode)
XmoNode InsertBefore(XmlNode, XmlNode)
XmlNode SelectSingleNode(String)
XmlNode SelectSingleNode(String, XmlNamespaceManager)
XmlNodeList SelectNodes(String)
XmlNodeList SelectNodes(String, XmlNamespaceManager)
IEnumerator GetEnumerator()

Properties

String Name{ get; }
String Value{ get; set;}
XmlNodeType NodeType{ get; }
XmlNode NullNode{ get; }
XmlNode ParentNode{ get; }
XmlNodeList ChildNodes{ get; }
XmlNode PreviousSibling{ get; }
XmlNode NextSibling{ get; }
XmlAttributeCollection Attributes{ get; }
XmlDocument OwnerDocument{ get; }
XmlNode FirstChild{ get; }
XmlNode LastChild{ get; }
Boolean IsContainer{ get; }
XmlLinkedNode LastNode{ get; set;}
Boolean HasChildNodes{ get; }
String NamespaceURI{ get; }
String Prefix{ get; set;}
Sring LocalName{ get; }
Boolean IsReadOnly{ get; }
String InnerText{ get; set;}
String OuterXml{ get; }
String InnerXml{ get; set;}
String BaseURI{ get; }
XmlDocument Document{ get; }
XmlElement Item{ get; }
XmlElement Item{ get; }
XmlSpace XmlSpace{ get; }
String XmlLang{ get; }
XPathNode XPNodeType{ get; }
String XPLocalType{ get; }

FIGURE 11.4
The structure of the `XmlNode` *class.*



Class: XmlNodeType, Methods: 0, Properties: 0

Fields

- Int32 value_
- XmlNodeType None
- XmlNodeType Element
- XmlNodeType Attribute
- XmlNodeType Text
- XmlNodeType CDATA
- XmlNodeType EntityReference
- XmlNodeType Entity
- XmlNodeType ProcessingInstruction
- XmlNodeType Comment
- XmlNodeType Document
- XmlNodeType DocumentType
- XmlNodeType DocumentFragment
- XmlNodeType Notation
- XmlNodeType Whitespace
- XmlNodeType SignificantWhitespace
- XmlNodeType EndElement
- XmlNodeType EndEntity
- XmlNodeType XmlDeclaration

FIGURE 11.5
`XmlNodeType` *enumeration.*

TIP

In addition to using the `DocumentElement` property, the `XmlDocument` object can directly obtain an `XmlNodeList` object that contains elements that match a given element name by using the `GetElementsByTagName` function. You can use this technique to quickly gain access to objects that have a unique element name.

When the root node obtained from `DocumentElement` is an element, a large number of properties are available. For example, you can use either the `Name` or the `LocalName` property to find the name of the element. You can use the `InnerXml` and `InnerText` properties to obtain information about the contents of the element. You can use the `InnerXml` property to return the actual XML markup contained in the current elements.

The `InnerText` property is a bit different from the other `DocumentElement` properties. It concatenates all the values for the contents of the current element and returns only that value rather than the entire XML markup. This means that some types of nodes have a `Value` property. For elements, the `Value` property returns `null` because an element is a container for other elements. If the element contains a value, a special type of node, called a text node, is generated to contain these values. So to get the values of elements, a program must first find the text nodes contained within those elements.

To better understand these properties, take a look at how they're used in Listing 11.2, which uses the XML content from Listing 11.1. We'll expand on this example as we discuss more properties of the `XmlNode` class and more objects in the `System.Xml` namespace later in the chapter.

LISTING 11.2 Examining Properties in the `DocumentElement` and `XmlNode` Classes

```
import System;
import System.Xml;

// Load the XML document
var xDoc:XmlDocument = new XmlDocument();
xDoc.Load("11-1.xml");

// Get the root element
var xRootNode:XmlNode = xDoc.DocumentElement;

// Start to print out the property values
Console.WriteLine("Name       : " + xRootNode.Name);
Console.WriteLine("LocalName  : " + xRootNode.LocalName);
Console.WriteLine("Value      : " + xRootNode.Value);
Console.WriteLine();

Console.WriteLine("InnerXml   : ");
Console.WriteLine(xRootNode.InnerXml);
Console.WriteLine();

Console.WriteLine("InnerText  : ");
Console.WriteLine(xRootNode.InnerText);
Console.WriteLine();
```

Listing 11.2 demonstrates the use of the document root element and isn't very interesting. In order to get at the real data in the document, you need to traverse the child nodes. The `XmlNode` has a special set of properties just for investigating child nodes. You can use the `HasChildNodes` property to make sure that the node in question has child nodes. If the node does have children, you can access the `FirstChild` property to grab the very first child node in the collection and the `LastChild` property to get the last child node in the collection. Unfortunately, there is not a way to enumerate through the children one by one, using only properties.

TIP

If you're familiar with XML and XPath, then it might be helpful to know that you obtain an `XmlNodeList` object that contains nodes that fit a particular `XPath` expression by using the `SelectNodes` method. `XPath` is beyond the scope of this chapter. For more information on XPath and XML in general, you might want to check the World Wide Web Consortium Web site (`www.w3c.org`) for the language specifications.

You can use the `ChildNodes` property to get all the children. This property returns an `XmlNodeList` collection of child nodes. The `XmlNodeList` class is a very basic collection class, whose only properties of consequence are the `Count` property and the default indexed property, Item, which returns the child node at a given index. Because the `XmlNodeList` object has a default indexed property, you can index into the child nodes just as if `XmlNodeList` were an array.

In addition to the three properties you can use to access child nodes (`FirstChild`, `LastChild`, and `ChildNodes`), you can also grab child nodes by name. `XmlNode` has a default indexed property so you can grab the first child with a given name by using the name in an indexer expression. Again, this means you simply pass the name to the `XmlNode` object as if you were accessing an array. Listing 11.3 expands on Listing 11.1, and it enables you to write information to the console for the child nodes of a document element.

LISTING 11.3 Accessing Child Nodes by Using the `XmlDocument` Class

```
import System;
import System.Xml;

// Load the XML document
var xDoc:XmlDocument = new XmlDocument();
xDoc.Load("11-1.xml");

// Get the root element
var xRootNode:XmlNode = xDoc.DocumentElement;

if ( xRootNode.HasChildNodes ) {
    // Make sure we have child nodes
    var i:int = 0;
```

Listing 11.3 continued

```
    Console.WriteLine("-------- Child Nodes using ChildNodes Collection");
    for(i = 0; i < xRootNode.ChildNodes.Count; i++) {
        Console.WriteLine("Name       : " + xRootNode.ChildNodes[i].Name);
        Console.WriteLine("LocalName  : " +
            xRootNode.ChildNodes[i].LocalName);
        Console.WriteLine("Value      : " + xRootNode.ChildNodes[i].Value);
        Console.WriteLine("Children?  : " +
            xRootNode.ChildNodes[i].HasChildNodes);
        Console.WriteLine();
    }

    // Let's print the First and Last as well
    Console.WriteLine(
        "-------- Child Nodes using FirstChild and LastChild Properties");
    Console.WriteLine("Name       : " + xRootNode.FirstChild.Name);
    Console.WriteLine("LocalName  : " + xRootNode.FirstChild.LocalName);
    Console.WriteLine("Value      : " + xRootNode.FirstChild.Value);
    Console.WriteLine("Children?  : " + xRootNode.FirstChild.HasChildNodes);
    Console.WriteLine();
    Console.WriteLine("Name       : " + xRootNode.LastChild.Name);
    Console.WriteLine("LocalName  : " + xRootNode.LastChild.LocalName);
    Console.WriteLine("Value      : " + xRootNode.LastChild.Value);
    Console.WriteLine("Children?  : " + xRootNode.LastChild.HasChildNodes);
    Console.WriteLine();

    // Let's get one by name
    Console.WriteLine("-------- Child Node access by name");
    Console.WriteLine("Name       : " + xRootNode["LINE"].Name);
    Console.WriteLine("LocalName  : " + xRootNode["LINE"].LocalName);
    Console.WriteLine("Value      : " + xRootNode["LINE"].Value);
    Console.WriteLine("Children?  : " + xRootNode["LINE"].HasChildNodes);
    Console.WriteLine();
}
```

Enumerating Attributes

You might have noticed that there is a bunch of data in the XML file that hasn't been accessed by any of the methods shown so far in this chapter. There are two reasons for this. First, the XML document is well structured and many levels deep, and so far none of the samples have tried to go back far enough into the document structure to grab the data. Second, there hasn't been a set of methods to access attributes.

In order to access the attributes, you need to use the `Attributes` property. Only when the node type of the current node is `XmlNodeType.Element` does the `Attributes` property return an `XmlAttributeCollection` object; otherwise, it simply returns `null`. Just like the `XmlNodeList` class, discussed previously in this chapter, the `XmlAttributeCollection` class has a default indexer that allows it to be used just like an array. The collection has a `Count` property with which you can access the attributes by index. The `XmlAttributeCollection` also allows you to access attributes by name.

The `XmlAttributeCollection` class returns `XmlAttribute` objects. These objects extend from `XmlNode`, so they should be extremely familiar to you. The important properties for an `XmlAtttribute` class are the `Name`, `LocalName`, and `Value` properties you saw earlier in the chapter, in the section "Enumerating Elements." Sometimes it is also important to get back to the element that owns the attribute being used. The `OwnerElement` class returns the owning element as an `XmlElement` class, which extends the `XmlNode` class. Listing 11.4 displays the attributes of the XML elements in the `11-1.xml` file, and it also shows the classes discussed so far working with the `11-1.xml` file.

LISTING 11.4 Using `XmlDocument`

```
import System;
import System.Xml;

// Load the XML document
var xDoc:XmlDocument = new XmlDocument();
xDoc.Load("11-1.xml");

// Get the root element
var xRootNode:XmlNode = xDoc.DocumentElement;
var i:int;

if ( xRootNode.HasChildNodes ) {
    // This is a working usage.  So the important items are being parsed.
    Console.WriteLine("Internal Settings - ");
    for(i = 0; i < xRootNode.ChildNodes.Count; i++) {
        if ( xRootNode.ChildNodes[i].LocalName == "INTERNAL" ) {
            Console.WriteLine("\tName   : " +
                xRootNode.ChildNodes[i].Attributes["Type"].Value);
            Console.WriteLine("\tValue  : " +
                xRootNode.ChildNodes[i].InnerText);
            Console.WriteLine();
        }
    }
```

Listing 11.4 continued

```
    Console.WriteLine("Line Objects - ");
    for(i = 0; i < xRootNode.ChildNodes.Count; i++) {
        if ( xRootNode.ChildNodes[i].LocalName == "LINE" ) {
            Console.WriteLine("\tLabel  : " +
                xRootNode.ChildNodes[i]["LABEL"].InnerText);
            Console.WriteLine("\tType   : " +
                xRootNode.ChildNodes[i]["TYPE"].InnerText);
            Console.WriteLine();
        }
    }
}
```

The code in Listing 11.4 appears to be enough information to parse nearly any XML document. So far, such simple code has proven to be all the power needed to complete most of the requirements of many applications. But you can use a more advanced syntax with the XML classes, which are extremely powerful and complex.

Remember that the `XmlDocument` class loads the entire tree into memory, and thus it can't be used for extremely large XML documents or when many XML documents, whose combined size is large, need to be loaded and used. When you need to deal with large XML documents, you can use the `XmlTextReader` class, which works with the XML file on disk rather than in memory. (A discussion about the full power of the `XmlTextReader` class is beyond the scope of this chapter.)

You can convert a few methods from Listing 11.4 that work with the `XmlDocument` class to work with the `XmlTextReader class`. `XmlTextReader` is a state engine class. This means that as various methods are called to advance the object through the XML document, various properties are updated with the information and thus the state of the reader is changed by the actions of the program. Several properties are required to mimic Listing 11.4. Many of the properties such as the `Name`, `LocalName`, `NodeType`, and `Value` properties, should already be familiar to you. They behave identically to the `XmlNode` class properties discussed earlier in the chapter. Quite a few new properties must be used as well. For example, the `Depth` property tells how many elements deep the reader currently is. If the reader is at the root node, the `Depth` property returns `0`; if the reader is at the root node's children, the property returns `1`; if the reader is at the children of the root node's children, the property returns `2`; and so on. Whenever the current `NodeType` class is `XmlNodeType.Element`, the `AttributeCount` property is as well, to take the place of the `XmlAttributeCollection.Count` property.

One of the most abstract concepts of reading XML content with the `XmlTextReader` class is how the elements are parsed. They are parsed from the top down, as if one element immediately followed another rather than elements actually containing other elements. To move between elements, you call the `Read` method. `Read` returns `true` if the next element is retrieved. To get the contents of an element, you call the `ReadString` method. This method concatenates all textual content within the node and returns a single string value. The final piece of the puzzle is the `GetAttribute` method, which can take either a string or an integer offset of the attribute to retrieve. The same string overload that is used in Listing 11.4 is used in Listing 11.5, which demonstrates all the `XmlTextReader` properties and methods being used together to mimic the functionality of the `XmlDocument` sample in Listing 11.4.

LISTING 11.5 Using `XmlTextReader` in place of `XmlDocument`

```
import System;
import System.Xml;

// Load the XML document
var xRed:XmlTextReader = new XmlTextReader("11-1.xml");
var bInternal = false;
var bLine = false;
var bLineDone = false;

// Get the root element
while(xRed.Read()) {
    if ( xRed.NodeType == XmlNodeType.Element ) {
        if ( xRed.LocalName == "INTERNAL" ) {
            if ( !bInternal ) {
                bInternal = true;
                Console.WriteLine("Internal Settings - ");
            }

            Console.WriteLine("\tName   : " + xRed.GetAttribute("Type"));
            Console.WriteLine("\tValue  : " + xRed.ReadString());
            Console.WriteLine();
        }

        if ( xRed.LocalName == "LINE" ) {
            if ( !bLine ) {
                bLine = true;
                Console.WriteLine("Line Objects - ");
            }

            bLineDone = false;
            while(xRed.Read() && !bLineDone) {
                if ( xRed.LocalName == "LABEL" ) {
                    Console.WriteLine("\tLabel  : " + xRed.ReadString());
                }
```

Listing 11.5 continued

```
                if ( xRed.LocalName == "TYPE" ) {
                    Console.WriteLine("\tType   : " + xRed.ReadString());
                    Console.WriteLine();
                    bLineDone = true;
                }
            }
        }
    }
}
```

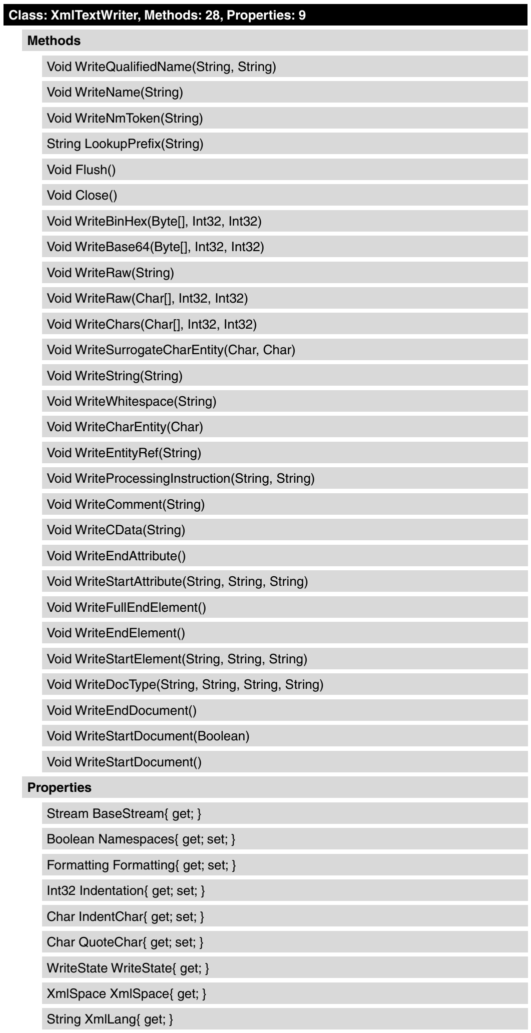
Creating New XML Content

Using `XmlDocument` and `XmlTextReader` to access ready-made XML content is the most common way of incorporating XML in a program. However, you often need to create new XML content.

Creating new XML content can be as easy as loading a text editor and creating a brand new XML-format document with the data needed for a program. You can also use a number of commercial and free programs that are available on the Internet to generate XML documents based on input parameters and some other data store, such as a `.csv` (that is, a file where all values are separated by commas) file or a database. You can also write your own tool to accomplish the same task. The `System.Xml` namespace makes this easy by providing an easy-to-use interface for writing XML documents.

This section focuses on the use of the `XmlTextWriter` class. This class is used to generate a new XML document based on either some data structures that are held in your program or some set of functions that generate data that needs to be written to file. After a document is created, it is difficult to edit the document by using the `XmlTextWriter` class; it is often much easier to load the document into an `XmlDocument` class and update the contents in-line. After you make the needed changes, you can use one of the overloads for `XmlDocument`'s `Save` method for saving the file back to disk.

Looking at Figure 11.6, you can see that `XmlTextWriter` is yet another very powerful and complex class. However, the methods needed to work with the sample data shown up until now are not very complex. In this section, you'll first write the contents of `11-1.xml` by using `XmlTextWriter` and then update the newly produced XML data by using the `XmlDocument` class.



Class: XmlTextWriter, Methods: 28, Properties: 9

Methods

- Void WriteQualifiedName(String, String)
- Void WriteName(String)
- Void WriteNmToken(String)
- String LookupPrefix(String)
- Void Flush()
- Void Close()
- Void WriteBinHex(Byte[], Int32, Int32)
- Void WriteBase64(Byte[], Int32, Int32)
- Void WriteRaw(String)
- Void WriteRaw(Char[], Int32, Int32)
- Void WriteChars(Char[], Int32, Int32)
- Void WriteSurrogateCharEntity(Char, Char)
- Void WriteString(String)
- Void WriteWhitespace(String)
- Void WriteCharEntity(Char)
- Void WriteEntityRef(String)
- Void WriteProcessingInstruction(String, String)
- Void WriteComment(String)
- Void WriteCData(String)
- Void WriteEndAttribute()
- Void WriteStartAttribute(String, String, String)
- Void WriteFullEndElement()
- Void WriteEndElement()
- Void WriteStartElement(String, String, String)
- Void WriteDocType(String, String, String, String)
- Void WriteEndDocument()
- Void WriteStartDocument(Boolean)
- Void WriteStartDocument()

Properties

- Stream BaseStream{ get; }
- Boolean Namespaces{ get; set; }
- Formatting Formatting{ get; set; }
- Int32 Indentation{ get; set; }
- Char IndentChar{ get; set; }
- Char QuoteChar{ get; set; }
- WriteState WriteState{ get; }
- XmlSpace XmlSpace{ get; }
- String XmlLang{ get; }

FIGURE 11.6
The structure of the `XmlTextWriter` *class.*

Several steps are required to write the contents of the `11.1` XML file that has been used for many of the samples and code listings in this chapter:

1. Create `XmlTextWriter`. During this step, you set the file or stream to which you write any XML content.

2. Use the various functions to write content. You can use the `WriteStartElement` and `WriteEndElement` methods to create any needed elements. The `WriteString` method can be used to write out element content. The `WriteStartAttribute` and `WriteEndAttribute` methods can be used for generating attributes, and the `WriteCData` method can be used to write the `CData` section.
3. Save the content. You do this by first ensuring that all content has been written to the underlying file or stream by using the `Flush` function. After all data has been written to the file, you close the file by using the `Close` method.

The easiest overload for the `XmlTextWriter` class is the `String` object's `Encoding` overload, in which a filename is passed to `XmlTextWriter` along with the encoding for the output. If the last parameter is left `null`, then UTF-8 encoding is used. This works perfectly for the example we're discussing in this section. Because `XmlTextWriter` is a stream-style object, the `Close` method should be called to close the XML document that is being written. You might also want to try to call `Flush` right before the document is closed, to ensure that all data is written to the file. The code to do all this is shown in Listing 11.6.

LISTING 11.6 Writing XML Content with `XmlTextWriter`

```
import System;
import System.IO;
import System.Xml;

public class ElementWriter {
    public static function
    WriteInternalElement(w:XmlTextWriter, type:String, value:String) : void {
        w.WriteStartElement("INTERNAL");

        // Write attributes
        w.WriteStartAttribute("Type", null);
        w.WriteString(type);
        w.WriteEndAttribute();

        // Write content
        w.WriteString(value);

        w.WriteEndElement();
    }

    public static function
    WriteLineElement(w:XmlTextWriter, label:String, type:String) : void {
        w.WriteStartElement("LINE");
```

LISTING 11.6 continued

```
        // Write content
        w.WriteStartElement("LABEL");
        w.WriteCData(label);
        w.WriteEndElement();

        w.WriteStartElement("TYPE");
        w.WriteString(type);
        w.WriteEndElement();

        w.WriteEndElement();
    }
}

var w:XmlTextWriter = new XmlTextWriter("11-6.xml", null);
w.Formatting = Formatting.Indented;

w.WriteStartDocument();

// Write Document Content
w.WriteStartElement("PAGE");

// Write Page Content
ElementWriter.WriteInternalElement(w, "StyleSheet", "Default.css");
ElementWriter.WriteInternalElement(w, "Branch", "Input");
ElementWriter.WriteInternalElement(w, "BaseName", "Commercial");
ElementWriter.WriteInternalElement(w, "RegionTable", "LakeMartin");
ElementWriter.WriteInternalElement(w, "RegionID", "292");

ElementWriter.WriteLineElement(w, "Overview", "Panel");
ElementWriter.WriteLineElement(w, "Property ID", "Display");

w.WriteEndElement();

w.WriteEndDocument();

w.Flush();
w.Close();
```

After the `XmlTextWriter` object has been created, the elements in question need to be generated. The first item in the XML file is the XML document tag, which can be written by using the `WriteStartDocument` and `WriteEndDocument` functions. These two functions wrap the entire application because the document needs to be started before any other elements are written, and it is closed after all elements are written. This concept is important because the ordering of start/end function calls determines the hierarchy of the document.

The first element in the document is the `<PAGE>` element, which you can write by using the `WriteStartElement` and `WriteEndElement` functions. For example, the `String` overload takes the new element's name and doesn't perform any special namespace processing. The same methods are used to write the `<INTERNAL>` and `<LINE>` elements.

To write the attributes, you can use the `WriteStartAttribute` and `WriteEndAttribute` functions. However, these functions don't provide any way to set the attribute value. For this, you use the `WriteString` method, so that the value of the attribute can be written to disk. The `WriteString` method can also be used to write the text nodes of the elements.

TIP

The `WriteString` method is only capable of writing `String` datatypes. However, XML is capable of supporting many different datatypes other than `String`. Each of these datatypes has a special XML format that enables it to appear in a validated document. You can use the `XmlConvert` class to convert other datatypes to their XML string equivalents when writing XML documents.

There is one item in the XML document that can't be written by using the standard element- and value-writing functions: the `CDATA` section, which you can write by using the `WriteCData` function. With just this limited set of methods, you can generate the content in `11-1.xml`. Listing 11.6 makes use of all the elements discussed so far to generate the file `11-6.xml`, which is nearly identical to the `11-1.xml` file that rest of the samples in this chapter use. The only difference is that in `11-6.xml`, only the elements used in the previous samples are written, which means quite a bit of junk in the old XML file isn't being written.

What happens if `RegionID` needs to be changed because the region in real life is either assigned a new ID or the region splits and gains two Region IDs? For one thing, you could simply regenerate the file. However, sometimes it makes more sense to perform an update on the file, especially if you no longer have the code that originally generated this file. To accomplish this, you can generate a program that uses `XmlTextReader` and `XmlTextWriter` in conjunction with one another to insert and update values that are being read as they are written.

A more appropriate way of changing `RegionID` would be to take advantage of the three sets of `XmlDocument` methods associated with modifying and appending to a loaded document:

- The first set of methods is used to update values and elements that are already in the document.
- The second set of methods is used to add new content to the XML document.
- The third set of methods enables you to save any changes that have been made.

The first set of methods is located on the `XmlNode` class directly. The `Value` property is read/write, which means to update an attributes value, you simply have to set a new value simply. For elements, which don't have values, you need to do a little more than set the `Value` property. If the only item in the element is textual content, you can set the `Value` property on the `FirstSibling` property for the element. Because all the objects being changed in the XML code shown in Listing 11.7 follow this pattern, we can assume that these methods will work for the majority of operations.

LISTING 11.7 Modifying XML Content by Using `XmlDocument`

```
import System;
import System.IO;
import System.Xml;

var xDoc:XmlDocument = new XmlDocument();
var xRootElement:XmlNode;
var i:int;
var xLastInternal:XmlNode;

// Load the document from the previous Code Listing
xDoc.Load("11-6.xml");

// Now let's modify some existing values
xRootElement = xDoc.DocumentElement;
for(i = 0; i < xRootElement.ChildNodes.Count; i++) {
    if ( xRootElement.ChildNodes[i].LocalName == "INTERNAL" ) {
        xRootElement.ChildNodes[i].Attributes["Type"].Value =
            "Attribute - " + i;
        xRootElement.ChildNodes[i].FirstChild.Value = "Old Element - " + i;

        xLastInternal = xRootElement.ChildNodes[i];
    }
}

// At the same time we need to add a new INTERNAL node
var xInternalNode:XmlNode =
    xDoc.CreateNode(XmlNodeType.Element, "INTERNAL", null);
```

LISTING 11.7 continued

```
var xTextNode:XmlNode = xDoc.CreateNode(XmlNodeType.Text,  null, null);
var xTypeAttr:XmlAttribute = xDoc.CreateAttribute("Type");

xTextNode.Value = "New Element";
xInternalNode.AppendChild(xTextNode);

xTypeAttr.Value = "New Attribute";
xInternalNode.Attributes.Append(xTypeAttr);

xRootElement.InsertAfter(xInternalNode, xLastInternal);

var w:XmlTextWriter = new XmlTextWriter("11-7.xml", null);
w.Formatting = Formatting.Indented;
w.Indentation = 1;
w.IndentChar = '\t';
xDoc.WriteTo(w);
w.Close();
```

The second set of methods is made available by the `XmlDocument`, `XmlNode`, and `XmlAttribute` classes. The `XmlDocument` class exports the `CreateNode` methods. The most common overload takes three parameters in the form `String`, `String`, `String` which takes as parameters the node type string, the node name, and the node namespace. Rather than use a string as the first parameter, you can pass in an `XmlNodeType` object. This is the method our example uses because it's faster to process an enumeration type than to parse a string.

You can also use the `CreateAttribute` method to generate `XmlAttribute` objects. All the `XmlAttributeCollection` methods for inserting items use these objects rather than `XmlNode` objects, so it is important to remember the distinction between these two functions and use `CreateAttribute` instead of `CreateNode` for making attribute nodes. The `CreateAttribute` method has an overload that takes the name of the attribute as a string. This is the overload used in Listing 11.7 and is the easiest overload for creating simple attributes.

TIP

In addition to creating new elements and attributes, you can also use the `Clone` method of the `XmlNode` class to create an exact copy of the current node. After the node is cloned, you can change various values. By using this technique, you don't have to build a complex node from scratch; you can first create a copy and then update its properties and insert it into the document as a new child node.

With an `XmlNode` object in hand, the next step is to either add additional elements to the node you are creating, set any properties, or add the node somewhere in the document collection. The `XmlNode` object has a series of methods for placing nodes in the `ChildNodes` collection. You can append nodes to the end of the collection by using `AppendChild`, which can be inserted either before or after another node with the `InsertAfter` and `InsertBefore` methods. You can also load the node to the beginning of the collection by using `PrependChild`, replace another node by using `ReplaceChild`, or remove a single child or all children by using `RemoveChild` and `RemoveAll`. Listing 11.7 uses the `InsertAfter` function and places additional `<INTERNAL>` elements directly in the document.

Adding attributes is a bit different than adding element nodes because element nodes are accessed by the `ChildNodes` property and attributes are accessed through the `Attributes` property. First, you must obtain the `XmlAttributeCollection` object by accessing the `Attributes` property. You add nodes by using a set of methods that are identical in operation to those found in the `XmlNode` class, including the `Append`, `InsertAfter`, `InsertBefore`, `Prepend`, `Remove`, `RemoveAll`, and `RemoveAt` methods. When these functions require a parameter, they use an `XmlAttribute` object, which you can create by using the `CreateAttribute` method of the `XmlDocument` class.

TIP

You can consume a series of `XmlDocument` class events as objects are added and removed. Programs can use these events by passing `XmlDocument` instances into library calls to determine whether the library modifies the `XmlDocument` in any way.

The final set of methods is used to save the modified XML content. Saving the document finalizes the modifications. If the file isn't saved after it has been modified, the modified version is not reflected in the file on disk and is destroyed when the program exits. You can do this by using the `Save` method of the `XmlDocument` class. The easiest of the Save method overloads is the `String` overload, which takes the filename to save to. In addition to the `Save` methods, the `XmlDocument` class also contains the `WriteTo` and `WriteContentTo` methods, which can be used to write entire documents using an `XmlWriter`-derived class to format the output XML code. Listing 11.7 makes use of the `XmlDocument` modification and save methods to update the `11-6.xml` file with additional content and change values of several of the existing elements.

TIP

You should use this section as a reference rather than as a once-through tool. If you're really interested in learning all the intricacies of XML, then use the .NET Software Development Kit documentation to look up some of the classes and features that are beyond the scope of this chapter.

Summary

XML has been a hot technology topic for quite some time, and therefore, quite a few programming methodologies use it. XML provides cross-platform compatibility and interoperation of other disparate technologies. In only a few years, it has become the de facto standard format for data stored outside a database that is both structured and human readable.

In this chapter you have learned multiple ways to read and load XML documents, as well as several ways to write XML documents and modify their contents.

In Chapter 12, "Regular Expressions," you'll discover a much older programming tool, known as the regular expression. You'll learn about the internal JScript .NET support of this programming feature and about how the common language runtime offers even more power and support.

Regular Expressions

CHAPTER

12

IN THIS CHAPTER

In nearly every type of application, some sort of string or user input parsing is required. This can involve parsing items off of the command line, processing user input to ensure that it is in a valid format, or searching for important data in documents or HTML pages. Each of these items quantifies an excellent place to use regular expressions when the item being searched for isn't exact. Regular expressions have the capability to match patterns in data rather than static strings that are hard-coded into a program or strings that are input by the user, but they still cannot match a group of different inputs. Regular expressions should be seen as just another programming tool, and that is how they are used in this chapter.

In JScript, rather than making regular expressions an external tool or library, they are accounted for directly in the language syntax. The JScript .NET language has direct support for regular expressions in some of the built-in objects such as the `String` object and the global `RegExp` object. JScript .NET also maintains syntax for returning a regular expression object using an inline format, so in this manner regular expressions are present directly in source rather than being created using the `new` operator and the normal object-creation mechanisms.

In addition to having a huge amount of regular expression support in the JScript .NET language, a huge amount of support is available through the CLR (Common Language Runtime). In many ways the CLR support for regular expressions is far superior to that implemented in the CLR, so most of the code samples and documentation focus on the CLR classes rather than the JScript .NET native support. It should be noted that in future versions of JScript .NET, more care will be taken to integrate the JScript regular expressions with the CLR classes, making the JScript inline syntax much more attractive.

After the basic classes for regular expression manipulation are discussed in detail, the chapter moves on to matching basic patterns using the CLR classes. This includes using both the instance classes and static methods to complete matches, and the differences between the two. Regular expressions can also be used for general string replacements. The associated syntax for this, discussed in the section "Using `Regex` for String Replacement," will help build the documentation for the CLR regular expression classes.

JScript `Regex` Support

JScript .NET has great built-in support for regular expressions, used to find arbitrary patterns in text rather than static strings. Regular expressions can be created inline with code using special inline syntax. Many of the built-in JScript objects also take either a string or a regular expression for parameters. A good example is the `String.replace` method, which can take either a string or a regular expression for the match property with a normal string property being used as the string replacement. This section begins with a discussion of this inline regular expression support and how useful it can be in JScript .NET.

In addition to the inline regular expression, JScript .NET also supports a special global regular expression object (`RegExp`). This object can be queried to provide the results for the previous match. This is often useful when you're using inline expressions because the match object is lost in the method call. This global object ensures that there is always an object from which to get match results.

Having built-in support for regular expressions also means having a format the regular expressions must appear in. There is a true standard for regular expressions, and JScript is compliant with a subset of the standard. This subset includes most metacharacters, look-ahead assertions, and back references. Don't be concerned if you don't understand these principles yet; I'll explain many of them throughout the chapter.

Inline Regular Expression Support

Inline regular expression support in JScript .NET basically means that the language has special syntax for denoting a regular expression inline with the code. For languages that don't support an inline format, you have to create an object and pass in parameters each time you want to create a pattern. JScript supports both inline regular expressions and object-oriented regular expressions because the `RegExp` global object can be used to generate new regular expression objects, but generally the inline format will work the best.

The inline format uses forward slashes to surround text that denotes a regular expression pattern. Any regular expression engine options can be placed after the second forward slash. The result of this expression will be assigned to a variable within the program or passed as a function parameter. Following is a sample inline expression:

```
/(\d+)/i; // Match a number with multiple digits. Case-insensitive option
```

But what exactly does a JScript .NET inline regular expression generate? It generates a `Microsoft.JScript.RegExpObject`. This object represents the inline regular expression and handles all the logic that JScript .NET performs behind the scenes.

In addition to the regular expression object being created by using an inline syntax, they can be created by using the `new` operator and the `RegExp` object. The `RegExp` operator takes two constructor parameters, the first being the regular expression and the second being the regular expression engine options. The following code demonstrates strongly typing the results of the inline regular expression to a variable and strongly typing the result of an explicit `RegExp` object creation:

```
import Microsoft.JScript;
var typedInline:RegExpObject = /(\d+)/i;
var typedExplicit:RegExpObject = new RegExp("(\d+)", "i");
```

The regular expression object has only three useful methods: the `compile` method, the `exec` method, and the `test` method. First, a regular expression that doesn't change and needs to be used multiple times can be compiled. The `compile` method converts the regular expression into an internal format that greatly speeds execution when matching against more than one string. To perform a pattern match, the `exec` method is used. Using a regular expression object, the `exec` method simply takes one parameter being the string to match against. If a match fails, this function returns null. Otherwise, it returns a JScript array of results. Array index 0 is the entire matched string, whereas indexes 1 through the end of the array are any subpatterns that were specified when the regular expression was created.

Sometimes the overhead of a full pattern match isn't necessary. If there is a large regular expression, it takes a lot less time to test whether the pattern simply matches, rather than creating the results array and returning all the subpatterns. For this purpose there is the `test` method of the regular expression object. The `test` method again takes a string to match against, just like the `exec` method, but instead of a full results array it returns either `true` or `false`. The following listing demonstrates the use of the `test` method to check for digits in a string:

```
import System;

var re = /^\d+$/;  // Make sure there are only digits
if ( re.test("123") ) {
    Console.WriteLine("Only digits");
} else {
    Console.WriteLine("Uh-oh, there are some non digits");
}
```

NOTE

Because `test` is faster than `exec`, it would seem logical to first test a string, and only if it matches run the `exec` method to get the results. This is a poor programming decision—the only time `test` is faster than `exec` is when the pattern match actually succeeds. This means `test` should be used only when the existence of a pattern in a string needs to be checked and not when subpatterns are being extracted from the string.

LISTING 12.1 Using the `exec` Method and Returned Array

```
import System;

var re = /.*?(\d+).*?/;
var results = re.exec("Let's match numbers - 123");
```

LISTING 12.1 continued

```
if ( results != null ) {
    Console.WriteLine("Total Captures (+1 for the Matched Text)");
    Console.WriteLine(results.length);
    Console.WriteLine();

    Console.WriteLine(
        "Full Matched Text (everything that matched the entire pattern)");
    Console.WriteLine(results[0]);
    Console.WriteLine();

    Console.WriteLine("Only the captured text");
    Console.WriteLine(results[1]);
    Console.WriteLine();
}
```

In addition to using the `exec` method, the `String.match` method can be passed a regular expression. Here the match occurs using the string that the `match` method was called on. This can be extremely handy if a regular expression is going to be used only once. Just as with the `exec` method, the `match` method returns an array or null depending on the success of the pattern match.

Another method of the `String` object `search` can be used to find a pattern match and return the offset at which it first occurs in the string. If a match is found, the offset is returned from the function; `-1` is returned if no match is found. This can be useful for picking large strings apart into smaller pieces, but you'll see later that regular expressions can be used to do this just as well.

There is also a `replace` method that takes a regular expression to match, along with a replacement string. This topic is discussed more in the section "Using `Regex` for String Replacement," because there are some special considerations involved in replacing pattern-matched strings.

After every `String.search`, `String.replace`, `String.match`, `test`, and `exec` method, the global regular expression object, or `RegExp`, is updated with a bunch of statistics about the match, along with a limited number of the captures from the match. This object is investigated next.

Global Regular Expression Object

The global regular expression object, or `RegExp`, always contains the results of the preceding pattern match. This object is similar to the `RegExp` instance objects that are created, but the global object is distinctly different. It can't be created but is always available within your program. Until a successful match occurs, all the properties of the `RegExp` object are undefined.

After a match occurs, the properties are updated with the results of that match. If one match succeeds and another fails, the RegExp isn't updated; it always contains the information for the last successful pattern match.

So exactly what type of information does it contain? Most important, it contains the last nine subpattern matches. That makes the RegExp object perfect for pulling out subexpression results and matches that can be used elsewhere in your program. It also contains information about where the match began in the string via the index property and where it left off in the string lastIndex. The index and lastIndex properties are useful for determining where the next pattern match will pick up if the engine flag of g was used in the match.

The global RegExp object also contains all the information about the string associated with the match. The input property is a copy of the input string to the match function. The leftContext property is a copy of all the text to the left of the pattern match. So anything in the string that isn't in the match but occurs before the match will appear in this variable. In turn, the rightContext property displays any text that is to the right side of the match but not included in the match. The string that was matched can always be retrieved from the lastMatch property. And although I don't know why it exists, there is also a way to get the last subpattern match by using the lastParen property.

TIP

As long as there are fewer than nine subpattern matches in the regular expression, it's probably best to just use the test method and then examine the RegExp object to examine the results of any subcaptures or return values.

I mentioned before that the last nine subpattern matches were available. These can be accessed using the $1 through $9 properties. However, this won't work if the source file is compiled using the /fast+ flag, which is the compiler default. To get at these properties, either the /fast- flag must be specified at the command line or the array indexers have to be used with the string value of "$1" being a sample index. Listing 12.2 demonstrates the RegExp object in action by executing a recurring regular expression and using index/lastIndex properties, and then executing a pattern match with multiple subpatterns and using the array access method of the $1 through $9 properties.

LISTING 12.2 Using the RegExp Object to Examine Matches

```
import System;

var multiMatch:String = "This is a hello world sample.";
```

LISTING 12.2 continued

```
var subMatch:String =
    "     There has to be something to match.  Words should be enough.";

DoMultiMatch();
DoSubMatches();

function DoMultiMatch(){
    var re = /\w+/g; // Match words

    Console.WriteLine("Performing a match multiple times");
    while(re.test(multiMatch)) {
        Console.WriteLine(
            RegExp.index + "-" + RegExp.lastIndex +
            "\t" + RegExp.lastMatch);
    }
    Console.WriteLine();
    Console.WriteLine();
}

function DoSubMatches() {
    var re = /(\w+) (\w+) (\w+) (\w+)/i;

    Console.WriteLine("Performing a match with sub pattern matches");
    re.test(subMatch);

    Console.WriteLine("Matched Sub Pattern #1: " + RegExp["$1"]);
    Console.WriteLine("Matched Sub Pattern #2: " + RegExp["$2"]);
    Console.WriteLine("Matched Sub Pattern #3: " + RegExp["$3"]);
    Console.WriteLine("Matched Sub Pattern #4: " + RegExp["$4"]);
    Console.WriteLine("Matched Sub Pattern #5: " + RegExp["$5"]);
    Console.WriteLine("Matched Sub Pattern #6: " + RegExp["$6"]);
    Console.WriteLine("Matched Sub Pattern #7: " + RegExp["$7"]);
    Console.WriteLine("Matched Sub Pattern #8: " + RegExp["$8"]);
    Console.WriteLine("Matched Sub Pattern #8: " + RegExp["$9"]);
    Console.WriteLine("Last Matched Pattern  : " + RegExp.lastParen);

    Console.WriteLine();
    Console.WriteLine("Match Location: " + RegExp.index + "-" +
        RegExp.lastIndex);
    Console.WriteLine("Last Match:             begin-->" +
        RegExp.lastMatch + "<-- end");
    Console.WriteLine("Unmatched Left String:  begin-->" +
        RegExp.leftContext + "<-- end");
    Console.WriteLine("Unmatched Right String: begin-->" +
        RegExp.rightContext + "<-- end");
```

LISTING 12.2 continued

```
    Console.WriteLine();
    Console.WriteLine();
}
```

Listing 12.3 is the same demonstration as Listing 12.2, except that you'll be using the compiler `flag /fast-` when compiling so that the `$1` through `$9` operators can be accessed using the dot (`.`) operator.

LISTING 12.3 Using the `RegExp` Object with Fast Mode Disabled

```
import System;

var multiMatch:String = "This is a hello world sample.";
var subMatch:String =
    "     There has to be something to match.  Words should be enough.";

DoMultiMatch();
DoSubMatches();

function DoMultiMatch(){
    var re = /\w+/g; // Match words

    Console.WriteLine("Performing a match multiple times");
    while(re.test(multiMatch)) {
        Console.WriteLine(RegExp.index + "-" + RegExp.lastIndex + "\t" +
            RegExp.lastMatch);
    }
    Console.WriteLine();
    Console.WriteLine();
}

function DoSubMatches() {
    var re = /(\w+) (\w+) (\w+) (\w+)/i;

    Console.WriteLine("Performing a match with sub pattern matches");
    re.test(subMatch);

    Console.WriteLine("Matched Sub Pattern #1: " + RegExp.$1);
    Console.WriteLine("Matched Sub Pattern #2: " + RegExp.$2);
    Console.WriteLine("Matched Sub Pattern #3: " + RegExp.$3);
    Console.WriteLine("Matched Sub Pattern #4: " + RegExp.$4);
    Console.WriteLine("Matched Sub Pattern #5: " + RegExp.$5);
    Console.WriteLine("Matched Sub Pattern #6: " + RegExp.$6);
    Console.WriteLine("Matched Sub Pattern #7: " + RegExp.$7);
```

LISTING 12.3 continued

```
    Console.WriteLine("Matched Sub Pattern #8: " + RegExp.$8);
    Console.WriteLine("Matched Sub Pattern #8: " + RegExp.$9);
    Console.WriteLine("Last Matched Pattern  : " + RegExp.lastParen);

    Console.WriteLine();
    Console.WriteLine("Match Location: " + RegExp.index + "-" +
        RegExp.lastIndex);
    Console.WriteLine("Last Match:             begin-->" +
        RegExp.lastMatch + "<-- end");
    Console.WriteLine("Unmatched Left String:  begin-->" +
        RegExp.leftContext + "<-- end");
    Console.WriteLine("Unmatched Right String: begin-->" +
        RegExp.rightContext + "<-- end");

    Console.WriteLine();
    Console.WriteLine();
}
```

Listing 12.2 and Listing 12.3 both demonstrate how to use the properties available on the global `RegExp` object to get results of a pattern match. Everything about a match can be determined, including all information about where the match began within the string, where it ended, the text preceding the match, and the text following the match. The next section, "JScript `Regex` Patterns," focuses on how to create the patterns used in Listing 12.2 and Listing 12.3.

JScript `Regex` Patterns

Every regular expression implementation has tables of metacharacters; tables of engine options that change the overall behavior of a pattern; and another series of syntaxes for modifying captures, the meaning of metacharacters, and whether or not matched patterns are stored or erased from the results.

At the end, Listing 12.3 will display a large number of sample matches, and behaviors will be demonstrated based on the following tables. Table 12.1 is the engine option table. It has options for modifying the overall matching behavior of a regular expression. This overall behavior can be changed without having to modify the regular expression itself.

TABLE 12.1 JScript Regular Expression Engine Options

Option	*Behavior*
`g`	This option enables the global search for all occurrences of a pattern. When the `g` option is enabled, a regular expression can be started again from the end of the last match. When matches are no longer available, the `lastIndex` for the regular expression object gets set and the matches start from the beginning again.
`i`	This option enables case-insensitive matching over the pattern. When this option is enabled, the case of the text doesn't matter.
`m`	This option enables multiline searches on patterns. When this option is enabled, the `^` metacharacter matches both the beginning of the string to be searched and any locations immediately following a newline. The `$` metacharacter matches both the end of the string and the location immediately preceding the newline.

Table 12.2 lists metacharacters available using JScript .NET regular expressions. These are characters that have a special meaning in a match rather than the standard character value. For instance, the character `"w"` in a regular expression matches exactly the character `"w"`, or `"w"` and `"W"` if the engine option of `i` is specified. However, the metacharacter `\w` matches any word character and is equivalent to the match group `[A-Za-z0-9_]`. Listing 12.4 shows examples of regular expressions that use several of the metacharacters in the table.

TABLE 12.2 JScript .NET Regular Expression Metacharacters

Character	*Description*
`\`	Marks the next character as a special character, a literal, a back reference, or an octal escape. For example: `w` matches "w" `\w` matches a word character `\\` matches "\" `\(` matches "(" `\1` matches the first subpattern match
`^`	Matches the position at the beginning of the input string. If the `RegExp` object's `Multiline` property is set, `^` also matches the position following `\n` or `\r`.
`$`	Matches the position at the end of the input string. If the `RegExp` object's `Multiline` property is set, `$` also matches the position preceding `\n` or `\r`.

TABLE 12.2 continued

Character	*Description*
`*`	Matches the preceding subexpression zero or more times. For example, `zo*` matches "z" and "zoo". `*` is equivalent to `{0,}`.
`+`	Matches the preceding subexpression one or more times. For example, `zo+` matches "zo" and "zoo", but not "z". `+` is equivalent to `{1,}`.
`?`	Matches the preceding subexpression zero or one time. For example, `do(es)?` matches the "do" in "do" or "does". `?` is equivalent to `{0,1}`.
`{`*n*`}`	*n* is a nonnegative integer. Matches exactly *n* times. For example, `o{2}` does not match the "o" in "Bob" but matches the two o's in "food".
`{`*n*`,}`	*n* is a nonnegative integer. Matches at least *n* times. For example, `o{2,}` does not match the "o" in "Bob" and matches all the o's in "foooood". `o{1,}` is equivalent to `o+`. `o{0,}` is equivalent to `o*`.
`{`*n*`,`*m*`}`	*m* and *n* are nonnegative integers, where *n* `<=` *m*. Matches at least *n* and at most *m* times. For example, `o{1,3}` matches the first three o's in "fooooood". `o{0,1}` is equivalent to `o?`. Note that you cannot put a space between the comma and the numbers.
`?`	When this character immediately follows any of the other quantifiers (`*`, `+`, `?`, `{`*n*`}`, `{`*n*`,}`, `{`*n*`,`*m*`}`), the matching pattern is nongreedy. A nongreedy pattern matches as little of the searched string as possible, whereas the default greedy pattern matches as much of the searched string as possible. For example, in the string "oooo", `o+?` matches a single "o" and `o+` matches all o's.
`.`	Matches any single character except `\n`. To match any character including the `\n`, use a pattern such as `[.\n]`.
`(`*pattern*`)`	Matches *pattern* and captures the match. To match parentheses characters "( )" use `\(` or `\)`.
`(?:`*pattern*`)`	Matches *pattern* but does not capture the match; that is, it is a noncapturing match that is not stored for possible later use. This is useful for combining parts of a pattern with the "or" character (`|`).
`(?=`*pattern*`)`	Positive lookahead matches the search string at any point where a string matching *pattern* begins. This is a noncapturing match; that is, the match is not captured for possible later use. For example, `Windows (?=95|98|NT|2000)` matches "Windows" in "Windows 2000" but not "Windows" in "Windows 3.1". Lookaheads do not consume characters; that is, after a match occurs, the search for the next match begins immediately following the last match, not after the characters that composed the lookahead.

Table 12.2 continued

Character	*Description*
`(?!`*`pattern`*`)`	Negative lookahead matches the search string at any point where a string not matching *`pattern`* begins. This is a noncapturing match; that is, the match is not captured for possible later use. For example, `Windows (?!95|98|NT|2000)` matches "Windows" in "Windows 3.1" but does not match "Windows" in "Windows 2000". Lookaheads do not consume characters; that is, after a match occurs, the search for the next match begins immediately following the last match, not after the characters that composed the lookahead.
`x|y`	Matches either *x* or *y*. For example, `z|food` matches "z" or "food". `(z|f)ood` matches "zood" or "food".
`[`*`xyz`*`]`	A character set. Matches any one of the enclosed characters. For example, `[abc]` matches the "a" in "plain".
`[^`*`xyz`*`]`	A negative character set. Matches any character not enclosed. For example, `[^abc]` matches the "p" in "plain".
`[`*`a-z`*`]`	A range of characters. Matches any character in the specified range. For example, `[a-z]` matches any lowercase alphabetic character in the range "a" through "z".
`[^`*`a-z`*`]`	A negative range of characters. Matches any character not in the specified range. For example, `[^a-z]` matches any character not in the range "a" through "z".
`\b`	Matches a word boundary, that is, the position between a word and a space. For example, `er\b` matches the "er" in "never" but not the "er" in "verb".
`\B`	Matches a nonword boundary. For example, `er\B` matches the "er" in "verb" but not the "er" in "never".
`\c`*`x`*	Matches the control character indicated by *`x`*. For example, `\cM` matches a Ctrl+M or carriage-return character. The value of `x` must be in the range of A-Z or a-z. If not, `c` is assumed to be a literal "c" character.
`\d`	Matches a digit character. Equivalent to `[0-9]`.
`\D`	Matches a nondigit character. Equivalent to `[^0-9]`.
`\f`	Matches a form-feed character. Equivalent to `\x0c` and `\cL`.
`\n`	Matches a newline character. Equivalent to `\x0a` and `\cJ`.
`\r`	Matches a carriage-return character. Equivalent to `\x0d` and `\cM`.
`\s`	Matches any whitespace character including space, tab, form-feed, and the like. Equivalent to `[ \f\n\r\t\v]`.

Table 12.2 continued

Character	*Description*
`\S`	Matches any nonwhitespace character. Equivalent to `[^ \f\n\r\t\v]`.
`\t`	Matches a tab character. Equivalent to `\x09` and `\cI`.
`\v`	Matches a vertical tab character. Equivalent to `\x0b` and `\cK`.
`\w`	Matches any word character including underscore. Equivalent to `[A-Za-z0-9_]`.
`\W`	Matches any nonword character. Equivalent to `[^A-Za-z0-9_]`.
`\x`*`n`*	Matches *`n`*, where *`n`* is a hexadecimal escape value. Hexadecimal escape values must be exactly two digits long. For example, `\x41` matches "A". `\x041` is equivalent to `\x04` and `"1"`. Allows ASCII codes to be used in regular expressions.
`\`*`num`*	Matches *`num`*, where *`num`* is a positive integer. A reference back to captured matches. For example, `(.)\1` matches two consecutive identical characters.
`\`*`n`*	Identifies either an octal escape value or a back reference. If `\`*`n`* is preceded by at least *`n`* captured subexpressions, *`n`* is a back reference. Otherwise, *`n`* is an octal escape value if *`n`* is an octal digit (0–7).
`\`*`nm`*	Identifies either an octal escape value or a back reference. If `\`*`nm`* is preceded by at least *`nm`* captured subexpressions, *`nm`* is a back reference. If `\`*`nm`* is preceded by at least *`n`* captures, *`n`* is a back reference followed by literal *`m`*. If neither of the preceding conditions exists, `\`*`nm`* matches octal escape value *`nm`* when *`n`* and *`m`* are octal digits (0–7).
`\`*`nml`*	Matches octal escape value *`nml`* when *`n`* is an octal digit (0–3) and *`m`* and *`l`* are octal digits (0–7).
`\un`	Matches *`n`*, where *`n`* is a Unicode character expressed as four hexadecimal digits. For example, `\u00A9` matches the copyright symbol (©).

From the preceding table, any JScript-supported regular expression can be generated. To put all of this together, the following sample demonstrates some command-line processing regular expressions that might be useful for command-line applications. Because most of the applications in this book are command-line based, it could certainly be used to enhance any of them. Listing 12.4 demonstrates the use of JScript .NET regular expressions to parse a command line and generate some format that can be used later in an application.

LISTING 12.4 Using JScript .NET Regular Expressions for Parsing

```
import System;
import System.Collections;

public class CommandLineProcessor {
    public static function
    ProcessCommandLine(Args:String[], SkipFirst:Boolean) : Hashtable {
        var commands:Hashtable = new Hashtable();

        for(var i:int = (SkipFirst) ? 1 : 0; i < Args.Length; i++) {

            var reSimplePlus = /^[\/\-](\w+):/;
            if ( reSimplePlus.test(Args[i]) ) {
                var options:ArrayList = new ArrayList();
                var command:String = RegExp["$1"];
                var optionString:String = RegExp.rightContext;

                var reSimpleMore = /(\w+?)(?:,|$)/g;
                while(reSimpleMore.test(optionString)) {
                    options.Add(RegExp["$1"]);
                }

                commands[command] = options;
                continue;
            }

            var reSimple = /^[\/\-](\w+)/;
            if ( reSimple.test(Args[i]) ) {
                // Load it with true, it's simple
                commands[RegExp["$1"]] = true;
                continue;
            }

            Console.WriteLine(Args[i] + " broke through the processor.");
        }

        return commands;
    }

    public static function PrintCommandLineHash(commands:Hashtable) : void {
        var Key:String;
        var Option:String;

        for(Key in commands.Keys) {
            if ( commands[Key] instanceof System.Boolean ) {
```

LISTING 12.4 continued

```
                Console.WriteLine(Key + ": " + Boolean(commands[Key]));
            } else {
                var options:ArrayList = ArrayList(commands[Key]);
                Console.WriteLine(Key + ": ");
                for(Option in options) {
                    Console.WriteLine("\t" + Option);
                }
            }
        }
    }
}

var Args:String[] = Environment.GetCommandLineArgs();
var Commands:Hashtable = CommandLineProcessor.ProcessCommandLine(Args, true);
CommandLineProcessor.PrintCommandLineHash(Commands);

[ic:output]
>12-4 /SimpleCommand /Complex:OneParm -SuperComplex:123,654,ThirdParm
SimpleCommand: true
SuperComplex:
        123
        654
        ThirdParm
Complex:
        OneParm
```

Listing 12.4 shows the use of three separate regular expressions used to parse command-line options. The regular expressions used here along with the metacharacters and engine options are going to help in the sections to come as text moves from JScript regular expressions to CLR regular expressions. The CLR regular expressions are both more complete in features and faster in execution speed.

Using .NET Regular Expressions

The CLR has a full-featured regular expressions engine that fully conforms to standards and also contains many additional features that are quite useful. This full implementation is located in the `System.Text.RegularExpressions` namespace displayed in Figure 12.1. The benefits of the CLR engine are many—so many, in fact, that the JScript .NET implementation is actually built on top of the CLR implementation. Because JScript .NET has to conform to the ECMA standards, the `RegExp` object was programmed to do any internal processing required, and then create a CLR regular expression on the backend.

Namespace: System.Text.RegularExpressions, TotalClasses: 12	
Regex	MatchEvaluator
Capture	CaptureCollection
RegexCompilationInfo	Group
GroupCollection	RegexRunner
Match	MatchCollection
RegexOptions	RegexRunnerFactory

FIGURE 12.1
The `System.Text.RegularExpressions` namespace.

The parsing engine implemented for CLR regular expressions is fast enough to build simple string processing expressions or even parsers for custom computer languages. ASP.NET page processing and parsing was built entirely out of regular expressions, and all parsing is done at page compile-time using the classes available in the `RegularExpressions` namespace.

To fully understand the CLR regular expressions, you need to examine each of the objects available in the `System.Text.RegularExpressions` namespace, along with the relationships between them. This section of the book discusses four primary classes: the `Regex`, which represents a parsing entity; the `Match`, which represents the results of a full regular expression hit on the string; the `Group`, which represents a matched group between parentheses; and, one step further, the `Capture`, which represents an individual match of an item within a matching group. It all starts with the `Regex`, so that is where this section begins.

`Regex` Class

The `Regex` class represents the entire pattern to be matched. The `Regex` also contains any engine options such as the global option (`g`) or the case-insensitive option (`i`). Because the `Regex` class contains information on the regular expression engine and the pattern to be matched, it is the start of every pattern match. The `Regex` class can be used in two ways. The first way is as an instance class. This means a `Regex` class can be generated from a pattern and some engine options, and then it can be used multiple times to match patterns. This is the way it should be used if the pattern is going to be executed multiple times, because the pattern is compiled to an internal format only once.

The second use of the `Regex` object utilizes static methods and requires that the pattern be passed in each time the methods are used. This is very efficient if a pattern is going to be used only once. For the most part, a regular expression will be used more than once, so the instance methods are the most important.

A new `Regex` can be generated either by using the string pattern in the constructor or by passing in both the pattern and several of the `RegexOption` enumeration values. For the most part, the pattern can use any of the features discussed in the section "JScript `Regex` Patterns." To avoid complicating things here with new features that aren't necessarily required to use regular expressions, I'm not going to use any features more advanced than have already been pointed out in tables or discussed. One additional feature I do want to point out is the named group. A named group is a way of attaching a real name to a group match. These names can be used to retrieve the captured group from the `Match` rather than using the ordinal. This improves the ease of modifying regular expressions without having to greatly modify the code that uses the results. The following code demonstrates creating a basic named group, and it also demonstrates the two `Regex` constructors:

```
import System.Text.RegularExpressions;

var reNoOptions:Regex = new Regex("(?<digits>\d+)");
var reOptions:Regex = new Regex("(?<digits>\d+)", RegexOptions.IgnoreCase);
```

Now that I've started using the `RegexOptions` in some of the examples, I should probably introduce the next table. Table 12.3 lists CLR `Regex` options.

TABLE 12.3 CLR `Regex` Options

Member Name	*Description*
`Compiled`	Specifies that the regular expression is compiled to an assembly. This yields faster execution but increases startup time.
`ECMAScript`	Enables ECMAScript-compliant behavior for the expression. This flag can be used only in conjunction with the `Multiline`, `IgnoreCase`, and `Compiled` flags. The use of this flag with any other flags results in an exception.
`ExplicitCapture`	Specifies that the only valid captures are explicitly named or numbered groups of the form `(?<name>...)`. This allows unnamed parentheses to act as noncapturing groups without the syntactic clumsiness of the expression `(?:...)`.
`IgnoreCase`	Specifies case-insensitive matching.
`IgnorePatternWhitespace`	Eliminates unescaped whitespace from the pattern and enables comments marked with `#`.
`Multiline`	Multiline mode. Changes the meaning of `^` and `$` so that they match at the beginning and end, respectively, of any line, and not just the beginning and end of the entire string.

TABLE 12.3 continued

Member Name	*Description*
`None`	Specifies that no options are set.
`RightToLeft`	Specifies that the search will be from right to left instead of from left to right.
`Singleline`	Specifies single-line mode. Changes the meaning of the dot (`.`) so that it matches every character (instead of every character except `\n`).

After the `Regex` object has been created, it is time to start using it. All the methods discussed are going to be instance methods. However, each of these methods also has a static function equivalent that takes a pattern as the second parameter immediately following the input parameter. A string can be tested for a match against a regular expression by using the `IsMatch` method. This method can take just the input string to test, or it can take both the input string and an offset at which to start matching. The return value is either `true` or `false` depending on whether the match was successful.

Two methods are available for actually returning a match. The `Match` method returns the first match in a string. The parameters for the `Match` function are the same as those for the `IsMatch` method. When more than one match needs to be retrieved, the `Matches` method can be used instead. This returns all the matches of a particular pattern within a string. This is similar to using the (`g`) option in JScript. The (`g`) option isn't available in the CLR engine, so the `Matches` method is used instead to simply return an array of all the matches. Again, the function prototypes are identical to both the `Match` and the `IsMatch` methods. Because each of these two functions returns `Match` objects, it is time to examine the `Match` class.

`Match` Class

The `Match` class represents a full match of the regular expression. One special property of the `Match` class is that it inherits from the `Group` class. This means that it also represents a group and has all the special properties of the `Group` class, including the `Value` property that represents a group match. However, because the `Match` object does represent the entire match, the `Value` property represents the entire matched string. In turn, the `Match` object contains the `Groups` property that returns a `Group` collection. The item at offset `0` in this collection is the same as `Match.Value`. This can be really confusing at first because there are multiple ways to get the same values, but I'll try to keep it as simple and consistent as possible.

Starting with the `Match` object from the beginning, you have to use the `Success` property quite often. The `Success` property needs to be investigated for a `Match` object to ensure that the pattern really did match. If the pattern didn't match, it will return `false`, and the program should perform the necessary processing to get a new value.

If the match is successful, the `Groups` property can be investigated to get subpatterns from the `Match`. The `Groups` property can be indexed using either an ordinal index or a string index by using the name of the group, assuming that it was a named group. There aren't any more important properties on the `Match` object. Any important properties are inherited from either the `Group` object or the `Capture` object. So let's move on to these two classes.

`Group` and `Capture` Classes

The `Group` and `Capture` objects are very similar. A `Group` object is the actual group match. This can be a named pattern match using a string identity. It can be a numbered pattern match. Or it can be a match that wasn't given an explicit name or number, but was assigned a default ordinal based on the location in the string. This is normally as far as a programmer needs to journey to get all the results from a regular expression.

The `Group` object does have a `Captures` property, though. This property returns a `CaptureCollection` class that can be indexed by ordinal. Just as with the `GroupCollection` for the `Match` object, the `Capture` at ordinal `0` will be the same thing as `Group.Value`. Again, it really isn't that important to get down this low in the object hierarchy when examining most matches. One benefit of the `Capture` object, however, is the `Index` and the `Length` properties. The `Group` and `Match` objects inherit both of these properties, and they represent the start and length of where the `Match`, `Group`, or `Capture` occurred in the string.

This should be enough information to get working on some basic pattern matches. The following section is dedicated to just matching basic patterns, examining all the properties and the methods discussed in this section, and using the static methods in addition to the instance methods.

Basic Pattern Matching

Up until now there has been a lot of talk and not much code or action. This section is devoted to matching various patterns and working only with code examples. Each item in this section is preceded by a short paragraph describing what is going on and then jumps straight into a code listing.

The code samples will include executing normal pattern matches, using the named groups features, testing for pattern success without returning results, and getting all successful matches using two features, one of which wasn't documented in earlier sections. A final code listing is dedicated to demonstrating many of these features using the static functions rather than the instance functions.

Executing Pattern Matches

As seen earlier, executing a pattern match entails first creating a new `Regex` object with the pattern and `RegexOptions`. Then the `Match` function is called with an input string to retrieve a `Match` object with the results of the pattern match against that string. Listing 12.5 demonstrates using the `Match` function and properties of the `Match` object to show the results of a pattern match.

LISTING 12.5 Investigating a `Match` Object

```
import System;
import System.Text.RegularExpressions;

var re:Regex = new Regex("(\\d+) (\\w+)");
var m:Match = re.Match("123 JScript.NET");

Console.WriteLine("Match Results:");
Console.WriteLine("Success: " + m.Success);
Console.WriteLine("Value: " + m.Value);
Console.WriteLine();

Console.WriteLine("Group Results:");
Console.WriteLine("Groups[0]: " + m.Groups[0].Value);
Console.WriteLine("Groups[1]: " + m.Groups[1].Value);
Console.WriteLine("Groups[2]: " + m.Groups[2].Value);
```

Getting Named Pattern Groups Back

Listing 12.5 does a really good job of matching the string in question. However, if a third match needs to be added between the first and second matches, the group numbers are going to get thrown out of whack. If some code relies on these group numbers, they get out of order and things might start to break. Listing 12.6 demonstrates adding in grouped matches so that reordering items in the regular expression won't affect code later in the application.

LISTING 12.6 Using Named Groups to Protect Code

```
import System;
import System.Text.RegularExpressions;

var re:Regex = new Regex("(?<digits>\\d+) (?<word>[\\w\\.]+)");
var m:Match = re.Match("123 JScript.NET");

Console.WriteLine("Match Results:");
Console.WriteLine("Success: " + m.Success);
```

LISTING 12.6 continued

```
Console.WriteLine("Value: " + m.Value);
Console.WriteLine();

Console.WriteLine("Group Results:");
Console.WriteLine("Groups[0]: " + m.Groups[0].Value);
Console.WriteLine("Groups[1]: " + m.Groups[1].Value);
Console.WriteLine("Groups[2]: " + m.Groups[2].Value);
Console.WriteLine();
Console.WriteLine("Groups[\"digits\"]: " + m.Groups["digits"].Value);
Console.WriteLine("Groups[\"word\"]: " + m.Groups["word"].Value);

// Reordered
re = new Regex("(?<word>[\\w\\.]+) (?<digits>\\d+)");
m = re.Match("JScript.NET 123");

Console.WriteLine("Match Results:");
Console.WriteLine("Success: " + m.Success);
Console.WriteLine("Value: " + m.Value);
Console.WriteLine();

Console.WriteLine("Group Results:");
Console.WriteLine("Groups[0]: " + m.Groups[0].Value);
Console.WriteLine("Groups[1]: " + m.Groups[1].Value);
Console.WriteLine("Groups[2]: " + m.Groups[2].Value);
Console.WriteLine();
Console.WriteLine("Groups[\"digits\"]: " + m.Groups["digits"].Value);
Console.WriteLine("Groups[\"word\"]: " + m.Groups["word"].Value);
```

Testing for Pattern Success

Testing for pattern success is sometimes the only thing required by a regular expression. Certain optimizations can be made in the regular expression if the engine knows that captures won't have to be saved. Because objects don't have to be created to hold the results of the match, memory can be saved and the pattern can be matched more quickly. The following code demonstrates testing an input string against a pattern using the `IsMatch` function:

```
import System;
import System.Text.RegularExpressions;

var re:Regex = new Regex("^\\d+$");
Console.WriteLine(Only digits?: " + re.Match("123"));
Console.WriteLine(Only digits?: " + re.Match("ab c123"));
Console.WriteLine(Only digits?: " + re.Match("1 23"));
Console.WriteLine(Only digits?: " + re.Match("134 abc"));
```

Getting All Successful Matches

Sometimes a pattern needs to match many times throughout a string. Maybe all the words need to be pulled out of the string, but not the digits. Maybe only full numbers need to be parsed out. Regardless of what circumstances warrant grabbing all the matches, Listing 12.7 demonstrates two methods for iterating over all the matches of a pattern within an input string.

LISTING 12.7 Getting All Matches Using `Matches` and `NextMatch`

```
import System;
import System.Text.RegularExpressions;

var re:Regex = new Regex("(?<digits>\\d+)");
var mArray:MatchCollection = re.Matches("123 456 abc 789 xyz 321");

for(var i:int = 0; i < mArray.Count; i++) {
    Console.WriteLine(mArray[i].Groups["digits"].Value);
}

var m:Match = re.Match("123 456 abc 789 xyz 321");
Console.WriteLine();
while(m.Success) {
    Console.WriteLine(m.Groups["digits"].Value);
    m = m.NextMatch();
}
```

Static Regular Expression Syntax

In addition to the instance methods that have been used in all the code listings up until now, there is also a set of static methods that can be used without having to create a `Regex` object first. These are very useful when a pattern will be run only a single time or when a user is supplying the regular expression to the program. Listing 12.8 demonstrates using a basic match, named groups, and testing regular expressions using the static methods.

LISTING 12.8 Using the Static `Regex` Methods

```
import System;
import System.Text.RegularExpressions;

var s:String = "123 JScript.NET";
var p:String = "(?<digits>\\d+) (?<word>[\\w\\.]+)";

if ( Regex.IsMatch(s, p) ) {
    var m:Match = Regex.Match(s, p);
```

LISTING 12.8 continued

```
    Console.WriteLine("Match Results:");
    Console.WriteLine("Success: " + m.Success);
    Console.WriteLine("Value: " + m.Value);
    Console.WriteLine();

    Console.WriteLine("Group Results:");
    Console.WriteLine("Groups[0]: " + m.Groups[0].Value);
    Console.WriteLine("Groups[1]: " + m.Groups[1].Value);
    Console.WriteLine("Groups[2]: " + m.Groups[2].Value);
    Console.WriteLine();
    Console.WriteLine("Groups[\"digits\"]: " + m.Groups["digits"].Value);
    Console.WriteLine("Groups[\"word\"]: " + m.Groups["word"].Value);
}
```

Using `Regex` for String Replacement

Regular expressions come in very handy for generic string replacements. Not only are they fast and efficient at replacing simple static strings, but they also are extremely fast and efficient for replacing ranges and patterns of strings. This section begins by examining the `Replace` function of the `Regex` object. This function can be used for both basic replacements and back reference replacements. For the samples in this section, the static `Replace` function will be used. This function takes the input string, the regular expression pattern, and the replacement string. The replacement string is the subject of this section.

Performing Basic String Replacement

With basic string replacements, the result is replaced with a simple string. This string has no necessary attachment to the string being matched against in any way. This is normally used to remove noise from a string by removing whitespace, doubled up characters, or digits from within textual content. The following code listing demonstrates some basic replacements, mainly removal of whitespace and digits from a string:

```
import System;
import System.Text.RegularExpressions;

var whitespace:String = "\t    Some Text    ";
whitespace = Regex.Replace(whitespace, "^\\s*", "");
whitespace = Regex.Replace(whitespace, "\\s*$", "");
Console.WriteLine("begin -->" + whitespace + "<-- end");

var noisy:String = "Here is some te8xt with7 buri1ed numbers.";
noisy = Regex.Replace(noisy, "\\d+", "");
Console.WriteLine(noisy);
```

Reusing Matched Patterns for Replacements

Some more complex patterns require that the matched string be used in the replacement. Using the CLR, there are two methods for specifying that these matches be used in the replacement. The first method is replacement by index. To specify one of the matches, the number of the match should be preceded by the dollar sign (`$`) in the replacement string. The second method of replacement is by name. For any named groups, the name of the match should be preceded by the dollar sign and an open French (curly) brace, and followed by a close French brace (for example, `${digits}`). Listing 12.9 demonstrates the use of matched patterns in the replacement strings.

LISTING 12.9 Using Group Captures in Replacement

```
import System;
import System.Text.RegularExpressions;

var strJScriptCode:String = "public function MyFunction() : void {";

// The following is going to format the JScript code using some
// HTML tags to color code keywords.

strJScriptCode = Regex.Replace(strJScriptCode,
    "(public|function|void)",
    "<font color=\"blue\">$1</font>");

strJScriptCode = Regex.Replace(strJScriptCode,
    "(?<ops>:|{|\\(|\\))",
    "<font color=\"red\"><b>${ops}</b></font>");

Console.WriteLine(strJScriptCode);
```

I recommend outputting the results of Listing 12.9 to an HTML file and then viewing the results. You'll see that the regular expressions used can do a fairly decent job of color-coding a JScript file, assuming that additional keywords are added to the first replacement and additional operators are added to the second replacement.

Summary

For those who use regular expressions in their common, everyday programming, the benefits of this chapter don't have to be explained. For others, the abstract syntax looks exactly like another programming language, and after all, you're having a hard enough time learning JScript .NET and all the new CLR classes. So why should you use regular expressions? Regular expressions enable data validation by pattern rather than static values. Expressions can be used to obtain pieces of a total string, and possibly even a single important value out of an entire file. Expressions can even be used to enable general replacement of values with a new value that is created from or dependent on the value matched. These features can take hundreds of lines of code if they are implemented without using regular expressions.

The chapter started with a discussion of the JScript .NET language support for regular expressions. For most people those will probably provide enough features to complete most programming jobs. We soon found that JScript .NET support for regular expressions is built on top of the CLR classes. This realization leads to the knowledge that you can get more performance out of the CLR classes.

Next the `System.Text.RegularExpressions` namespace was examined along with four common regular expressions objects: the `Regex` object, responsible for handling pattern matches at the highest level; the `Match` object, responsible for representing an overall match against a string; and the `Group` and `Capture` objects, which represent matching groups in the regular expression and individual captures for metacharacter groups.

This led to several examples of matching patterns, matching all instances of a pattern in a given string, testing for matches, and named group patterns. Along with simple pattern matches, you found that string replacement was also possible using regular expressions. Two types of replacement were discussed at this point, static string replacement and pattern replacements. The later pattern replacements proved very useful in modifying or marking up keywords and characters.

The next chapter moves far from the world of regular expressions. It focuses on Web-enabling your application. This includes downloading program updates and grabbing HTML content. We'll also discuss using the Net classes for asynchronous Web access, allowing the program to perform some other task while the download happens in the background.

CHAPTER

13

Accessing the Internet

IN THIS CHAPTER

Accessing the Internet from within an application used to be a common but daunting programming task. The APIs for accessing the Internet either didn't exist or were difficult to use and implement in a meaningful way. Microsoft has done many things to make accessing the Internet easier than it originally was. The Inet APIs available to C/C++ programmers made it significantly easier to make basic Web requests including both HTTP and FTP connections. Internet Explorer began to export a COM model that could be used from within any COM-compliant programming language to actually embed a Web browser within the application. These advances made Web programming much easier.

With JScript .NET all the old mechanisms still exist. The IE object model can be used to provide a Web GUI, or the Inet APIs can be called using a special CLR method known as `PInvoke` (or Platform Invoke). In addition to the old methods, there are many new methods for accessing the Web. The CLR has classes for working with DNS and resolving hostnames to IP addresses, classes for making basic Web requests, classes for working with cookies, classes for requesting files from the Web, and even more classes that receive responses initiated by a Web request.

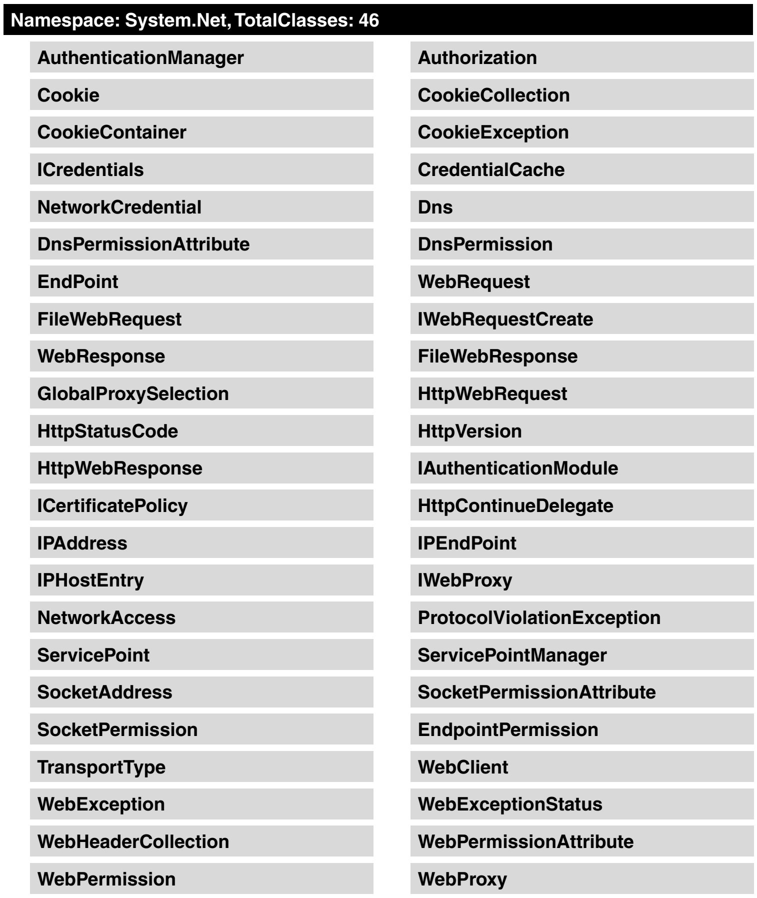
All of these classes can work in either synchronous or asynchronous mode. The asynchronous mode is actually easy to use and implement, so it has a definite advantage in making more responsive programs that perform network operations in the background.

This chapter doesn't discuss all the features available in the Net classes, but it does review the `System.Net` namespace with emphasis on the classes used to initiate Web requests and retrieve responses. `WebRequest`, `WebResponse`, `HttpWebRequest`, and `HttpWebResponse` are the classes necessary to access Web page content.

The first examples show synchronous operation of the Net classes. The section "Asynchronous Web Requests" demonstrates ways to make the same Web applications using the asynchronous programming model.

The `System.Net` Namespace

The `System.Net` namespace is another one of those namespaces for which a large number of confusing classes exist for using advanced features of the namespace. For the most part, only a few classes are required for performing basic network requests. The classes important to this chapter are `WebRequest`, `WebResponse`, `HttpWebRequest`, and `HttpWebResponse`. Figure 13.1 shows an example of the System.Net namespace.



Namespace: System.Net, TotalClasses: 46	
AuthenticationManager	Authorization
Cookie	CookieCollection
CookieContainer	CookieException
ICredentials	CredentialCache
NetworkCredential	Dns
DnsPermissionAttribute	DnsPermission
EndPoint	WebRequest
FileWebRequest	IWebRequestCreate
WebResponse	FileWebResponse
GlobalProxySelection	HttpWebRequest
HttpStatusCode	HttpVersion
HttpWebResponse	IAuthenticationModule
ICertificatePolicy	HttpContinueDelegate
IPAddress	IPEndPoint
IPHostEntry	IWebProxy
NetworkAccess	ProtocolViolationException
ServicePoint	ServicePointManager
SocketAddress	SocketPermissionAttribute
SocketPermission	EndpointPermission
TransportType	WebClient
WebException	WebExceptionStatus
WebHeaderCollection	WebPermissionAttribute
WebPermission	WebProxy

FIGURE 13.1
The `System.Net` *namespace.*

Every Web operation starts with the `WebRequest` class. Calling the `Create` method along with the URI for the remote Web site or network file will create a new `WebRequest` class. The `WebRequest` class is special in that when the `Create` method is called, it does not return an instance of a `WebRequest` object, but rather returns an instance of the class that is registered for the protocol prefix in the URI. If the `http://` protocol prefix were used, the class returned from a call to `Create` would be an `HttpWebRequest` class.

This can be rather confusing, but it enables the programmer to use a single class when programming to handle any style of URI with which a user might identify a remote resource. If the user passed in a `file://` protocol URI, the `FileWebRequest` would be returned instead of the `HttpWebRequest`. Any class that is registered with a prefix must inherit from the `WebRequest` base class. The `WebRequest` base class has enough methods to perform any basic network requests. Each of the derived classes might in turn, contain methods or properties specific to that type of Web transaction, but for the most part only the methods on the `WebRequest` object need to be used.

Several useful methods are available on the `WebRequest` object. The `GetResponse` method can be used to actually initiate the Web request and retrieve an answer from the server. Depending on the protocol, this response can be very different. It can be a Web page, a file of some sort, or possibly some directory services information. For asynchronous programming, the `BeginGetResponse` and `EndGetResponse` are used in place of the `GetResponse` method. The begin method initiates the network transfer and accepts a special callback function. After the response is retrieved, the callback function is called and the end method can be used to finalize the transaction. If a request is taking too long in an asynchronous mode, the `Abort` method can also be called to stop the transaction from completing and cancel the request. Most of the time this method isn't used because the `Timeout` property is more useful. The `Timeout` property takes the maximum time in milliseconds before the request will be terminated. If the timeout is reached, an exception is thrown, indicating that there was a failure to retrieve the response.

In addition to the properties on the `WebRequest` class, there are some properties available only on the HTTP implementation that might come in handy. The `UserAgent` property can be used to make the server think that the connection is coming from a well-known browser or a brand-new one. The `AllowRedirect` property can be set to follow redirects automatically. This can be handy when you're writing some sort of Web spider application that lists all the pages available on a particular Web site. Even the request `Method` can be modified from the default of `GET` and changed to something like `POST` or `TRACE`. Again, the basic methods of the `WebRequest` are generally enough to write generic functions to handle any protocol, but often you can get a better response by using the more advanced features of the derived classes.

After the request is created, the `GetResponse` method is generally called. This method returns a `WebResponse` object. Now this response object behaves in the same manner as the request object did. Instead of returning a true `WebResponse`, it returns the derived class associated with the protocol. Now, after a `WebResponse` object is obtained, the request and response have fully completed, all requests have been sent, all response data has been received, and the connection to the remote server has been shut down. Properties on the `WebResponse` object can be examined to determine the success of the operation. The `ContentLength` property returns the amount of data received, the `Headers` property retrieves a collection of headers returned by the server, and the ResponseUri confirms the name of the page actually received. Remember that if redirections are on, this URI might be different from the originally requested URI.

If the request was successful and data was retrieved, the GetResponseStream method has to be called to get the Stream object for the data buffer. Any of the IO classes for working with streams discussed in Chapter 10, "File Access," can be used here to examine the data. After the data has been processed, the Close method *must* be called to shut down the response object and clean up resources. If the Close method is not called, a resource leak will result and future Web requests will block and wait for an available socket that will be tied up until the next GC pass (Garbage Collection). This is probably the most important concept in using the Net classes because not closing response objects will starve a program for available resources to initiate future requests.

Again, the HttpWebResponse is more full featured than its parent, WebResponse. Some of the more useful properties include the following: The Server property can determine which server returned the request in question. The StatusCode property can be examined to determine whether an HTTP error occurred such as a 404 File Not Found error or maybe a 500 Internal Server Error. The StatusCode returns only the numeric portion of the error message, but the StatusDescription returns a human-readable form of the StatusCode.

That should be enough information to work through some basic Web transactions. Let's move on to making Web requests and processing the responses.

Simple Web Requests

Requesting Web content is becoming a very common occurrence in many applications. Accessing the Web can be done to download new data for the application (XML seems to be very popular for this), possibly to download new files such as music files or updates to the program, or just to grab some content for the user. Because all the required methods for making Web requests were discussed in the first section of this chapter, this section demonstrates the actual programming concepts for accessing Web content. The examples will all accept URLs on the command line, and there won't be any URLs hard-coded in the listings. This means that the samples won't work unless they are properly called with a URL on the command line. So be sure to get a good list of test URLs together to test each of the code examples. This section discusses the use of basic Web requests using the WebRequest class followed by a more advanced use that checks for HTTP conditions and uses the HttpWebRequest. There is also a FileWebRequest that works with files on the local machine, but this won't be discussed because basic file IO, discussed in Chapter 10 will handle this situation in a more thorough manner.

The most basic type of Web request involves pulling down an HTML page. This is actually pretty common. Often, pulling down a Web page from within a Web browser can invoke the wrath of the banner-ad gods, so having an alternative method of getting just the page wanted by the user can be beneficial. This type of tool is also beneficial when you're creating Web pages because it is useful to see what type of content is available to users with different browsers (the user agent property can be used to trick the server into returning content for a specific browser). Listing 13.1 demonstrates downloading a specific page—the user agent will be set to a custom agent, but this can be changed to mimic a more popular browser like Netscape or IE—and saving the HTML content out to a page specified by the user.

LISTING 13.1 Saving a Request to a File

```
import System;
import System.IO;
import System.Net;

public class SaveHtmlPage {
    public static function Main(Args:String[]) : void {
        if ( Args.Length < 3 ) {
            return;
        }

        // Perform the Web Request
        var wr:WebRequest = WebRequest.Create(Args[1]);
        var wrs:WebResponse = wr.GetResponse();

        // Open the output File and get Network Stream
        var file:Byte[] = new Byte[wrs.ContentLength];
        var fs:FileStream = new FileStream(Args[2], FileMode.Create);
        var s:Stream = wrs.GetResponseStream();

        s.Read(file, 0, int(wrs.ContentLength));
        s.Close()

        fs.Write(file, 0, int(wrs.ContentLength));
        fs.Flush();
        fs.Close();

        wrs.Close();
    }
}

SaveHtmlPage.Main(Environment.GetCommandLineArgs());
```

Saving the results of a request to a file is pretty common. In Listing 13.1 either a file URL or an HTTP URL could have been specified, meaning that the sample can save files from network shares or the local file system. The problem with this general mechanism for saving the results of URL requests is that, at least in the case of HTTP requests, various error conditions can occur, but a response will still be received from the server. If that happens to be an error message in HTML format, that is what will be saved to the file. Wouldn't it be nice to print something to the console and not save the file if something goes wrong? Listing 13.2 demonstrates performing conditional actions when the request is an HTTP request.

LISTING 13.2 Saving a Request to a File with HTTP Conditions

```
import System;
import System.IO;
import System.Net;

public class SaveHtmlPage {
    public static function Main(Args:String[]) : void {
        if ( Args.Length < 3 ) {
            return;
        }

        // Perform the Web Request
        var wr:WebRequest = WebRequest.Create(Args[1]);
        var wrs:WebResponse = wr.GetResponse();

        if ( !(wrs instanceof HttpWebResponse) ||
             HttpWebResponse(wrs).StatusCode == 200 ) {
            // Open the output File and get Network Stream
            var file:Byte[] = new Byte[wrs.ContentLength];
            var fs:FileStream = new FileStream(Args[2], FileMode.Create);
            var s:Stream = wrs.GetResponseStream();

            s.Read(file, 0, int(wrs.ContentLength));
            s.Close()

            fs.Write(file, 0, int(wrs.ContentLength));
            fs.Flush();
            fs.Close();
        }

        wrs.Close();
    }
}

SaveHtmlPage.Main(Environment.GetCommandLineArgs());
```

Asynchronous Web Requests

The capability to have two code operations occurring at the same time is known as asynchronous programming. In addition to making synchronous requests for files, it is often beneficial to make requests that are processed by the application in the background. This could be the case when the user can still manipulate and work with the application while the download is in progress, or when the user should have the ability to abort the download in progress. In Listing 13.3 a file will be download asynchronously while a console counter prints every second the file is downloading. The application will exit after the file has been downloaded.

LISTING 13.3 Downloading a File Asynchronously

```
import System;
import System.IO;
import System.Net;
import System.Threading;

public class SaveHtmlPage {
    private static var pendingResult:Boolean = false;

    public static function Main(Args:String[]) : void {
        if ( Args.Length < 2 ) {
            return;
        }

        // Perform the Web Request
        var wr:WebRequest = WebRequest.Create(Args[1]);

        pendingResult = true;
        wr.BeginGetResponse(SaveHtmlPage.Download_Callback, wr);

        while(pendingResult) {
            Console.WriteLine("Counting once every second");
            System.Threading.Thread.Sleep(1000);
        }

        return;
    }

    private static function Download_Callback(ar:IAsyncResult) : void {
        try {
            var wrs:WebResponse =
                WebRequest(ar.AsyncState).EndGetResponse(ar);
            wrs.Close();
        } catch(exc:Exception) {
```

LISTING 13.3 continued

```
            Console.WriteLine("Unable to obtain a valid response.");
        }
        pendingResult = false;
    }
}

SaveHtmlPage.Main(Environment.GetCommandLineArgs());
```

Summary

This chapter discussed the concepts of CLR networking in a very minimal manner. The topics discussed are enough to migrate basic Web interactions into any program you might want to create. The chapter began with the concepts of `WebRequest` and `WebResponse` objects. These objects will always be available through the lifetime of the CLR. What will change are the protocol handlers such as the `HttpWebRequest` and the `FileWebRequest`. Expect some handlers for ftp and other common Web protocols in the near future.

With these basic concepts behind us, we moved into creating basic Web requests in the form of downloading content to disk. Even though the returned response stream was used only for obtaining a byte array and writing it to file, it certainly could be used for more advanced processing. More advanced processing might include making sure that the file is written using the correct encoding format. The byte array retrieved from the `WebResponse` can be converted to a string to be processed using regular expressions (see Chapter 12, "Regular Expressions") or other string processing techniques. You also saw that the generic response object could always be casted to the real response object for the protocol handler being used. This gives access to many more parameters and methods useful only for that particular protocol. In the case of the HTTP protocol, the `StatusCode` can always be used to determine the true success of the request rather than just whether the server exists.

In addition to the synchronous Web request operation, you saw an asynchronous option. This enables a program to do background processing of requests and responses. It can be used to process multiple requests at the same time, or it can be used to provide a crisp response to the user even though the program is busy performing the download. Asynchronous downloads also give the user the power to cancel the download in progress if it is taking too long. All the principles make for a better user experience and a better program.

In the next chapter we'll continue our discussion of the Web-based technologies as we examine using JScript .NET to program some basic ASP.NET pages. If you have a Web server on your machine, you can even use the next chapter to build pages for use with the examples in this chapter.

CHAPTER

14

Using JScript .NET in ASP.NET

IN THIS CHAPTER

VBScript and JScript were traditionally the languages of choice in Active Server Pages (ASP), mainly because they were the only two languages that Microsoft supported and developed for its server-side Web programming platform. With ASP.NET, many more languages are becoming available to the server-side platform, including C#, VB .NET, and JScript .NET.

ASP.NET has many benefits over its precursor, ASP. The most important benefit of ASP.NET is that it is precompiled, which means all the content is running as fast as possible on the target central processing unit rather than being interpreted at runtime, as are the scripting languages in ASP. Another important benefit of ASP.NET is that it offers many user services, such as Hypertext Markup Language (HTML) and Web controls. These controls can be programmed by using an object hierarchy, starting at the page level. In addition, ASP.NET supports custom authentication technologies, built-in support for form processing, and error handling that had to be hand-coded in ASP.

This chapter helps you get some of the basic features of ASP.NET working under JScript .NET. It begins with a discussion of ASP.NET top-level directives, which are parsed by the ASP.NET engine and which enable various features that are normally turned off.

When you have a basic ASP.NET page up and running, you need to create code blocks, which are used in ASP.NET to manipulate the server-side object model and provide all the server-side code that occurred in ASP either at the top of a page or in-line with a page. In ASP.NET, code blocks can be centralized and moved either solely to the top of the page or to a separate code-behind file. In this manner, code can be separated from presentation and the code and presentation don't have to coexist within the same file.

The chapter ends with a discussion of the basic event handlers in ASP.NET. These event handlers allow you to process a page at different stages, with different types of access levels to various portions of the server-side object model.

ASP.NET Page-Level Directives

ASP.NET programming begins with a series of top-level directives known as *page-level directives*. ASP.NET parses these directives and enables various features during the compilation process. These directives can be used to import namespaces, register controls with tag prefixes, set up code-behind, or notify ASP.NET that trace or debug mode should be enabled.

NOTE

Code-behind is a feature that is primarily used by the Visual Studio .NET ASP.NET Web Designer. When a page is set up for code-behind, all the code used to manipulate the page is placed in a source file that is separate from the HTML and ASP.NET markup.

This allows for the code and presentation to be separated.

In addition to splitting the code from the page and design markup, code-behind also enables code designers to more easily parse and change the code or markup on an individual basis. The code can also be compiled into an assembly without compiling the page content.

Code-behind is discussed in detail later in this chapter.

You can create a very basic ASP.NET page that has no page-level directives and no server-side code. The only requirements of an ASP.NET page are that it be created with the `.aspx` file extension and that it run on a Web server that supports ASP.NET. The page directives discussed in this section include the `Page` directive, and the `Import`, `Assembly`, and `Register` directives.

The `Page` Directive

The `Page` directive is used to control basic page-level options such as the programming language used, the debug and tracing flags, and the character encoding. The code snippet below demonstrates the basic format of any directives. In the case of `Page` directives, *`DirectivePrefix`* is `Page`. The `Page` directive is used in almost every ASP.NET page, to specify various compile-time attributes, including the debug and trace options.

```
<%@ DirectivePrefix attribute="value" [attribute="value" ...] %>
```

The `Page` directive attributes start with the `Language` attribute, which you use to specify the default language for the rest of the ASP.NET file. After the language is set at the page level, it can't be changed by a code block later. If any code blocks specify an alternate language, they will cause errors. In this chapter, the `Language` attribute is set to either JScript or JScript .NET.

You use the `Debug` and `Trace` attributes to control debugging information. Setting the `Debug` attribute to `true` enables debugging information to be generated; all errors are then shown with line numbers and offsets in the source file. This is extremely helpful and is identical to specifying the `/debug+` switch to the command-line compiler. Setting the `Trace` attribute to `true` enables a rather large printout of information at the bottom of the rendered ASP.NET page. This information includes the rendered sizes of various controls, the amount of time it took to render each control on the page, and a list of all of the various server collections.

ASP.NET includes two states: session state, which enables you to save information for a client across page requests, and view state, which enables a page to save control values and information as the page is posted back. Session state, is enabled by default, but you can disable it by setting the `EnableSessionState` attribute to false. Disabling session state causes a very small increase in page performance. View state is also enabled by default. View state is a new feature in ASP.NET, in which controls can remember state between form posts. This enables round-trips to the server, so you don't have to write code to read in the form values and then write them back to the controls or hidden form variables. You can disable view state by setting the `EnableViewState` attribute to `false`.

A couple ASP.NET attributes that you will use later are the `Inherits` and `Src` attributes. The `Inherits` attribute allows the current ASP.NET file to inherit from a class that is programmed and compiled into a dynamic link library or a class that is located in the file to which the `Src` attribute points. This is known as *code-behind* because the code file is separate from the display page file. The `Src` attribute in turn points to some source code file that contains the class to which the `Inherits` attribute points. For the purposes of this chapter, these two attributes must be specified together. The following example demonstrates the use of each of these attributes in a `Page` directive:

```
<%@ Page Language="JScript" Debug="true" Trace="true"
    EnableViewState="false" EnableSessionState = "false"
    Inherits="MyFormClass" Src="CodeBehind.js" %>

<body>
    Hello World
</body>
```

The `Assembly`, `Import`, and `Register` Directives

You use the `Assembly`, `Import`, and `Register` directives to control various parts of the compilation process. The `Import` directive is capable of importing namespaces and is similar to using an `import` statement in JScript .NET code. The `Assembly` directive is used for including an assembly in the command line, and it is similar to the `/r` compiler option. The `Register` directive is completely new in ASP.NET, and it is used to link a namespace of classes to a special tag name so that the classes can be used as server controls.

The `Import` directive is capable of importing additional namespaces for use in an ASP.NET page. These namespaces might be the namespaces of special library code that need to be used in the page or namespaces for special controls to use in the ASP.NET page. In any case, the `Namespace` attribute is assigned the name of the namespace in question. By default, many namespaces are already included by the ASP.NET runtime, but it doesn't cause an error if you include `Import` statements for namespaces that are already imported by ASP.NET. The following is an example of importing namespace that shows all the namespaces that are automatically imported by ASP.NET:

```
<%@ Import Namespace="System" %>
<%@ Import Namespace="System.Collections" %>
<%@ Import Namespace="System.Collections.Specialized" %>
<%@ Import Namespace="System.Configuration" %>
<%@ Import Namespace="System.IO" %>
<%@ Import Namespace="System.Text" %>
<%@ Import Namespace="System.Text.RegularExpressions" %>
<%@ Import Namespace="System.Web" %>
<%@ Import Namespace="System.Web.Caching" %>
<%@ Import Namespace="System.Web.Security" %>
<%@ Import Namespace="System.Web.SessionState" %>
<%@ Import Namespace="System.Web.UI" %>
<%@ Import Namespace="System.Web.UI.HtmlControls" %>
<%@ Import Namespace="System.Web.UI.WebControls" %>
```

Sometimes just importing a namespace isn't enough. For example, ASP.NET may need to know in which assembly the namespace resides. This is a job for the `Assembly` directive. The assembly directive has two attributes, `Name` and `Src`. The `Name` attribute can be filled with the name of the assembly to load, without any extension information (such as `.exe` or `.dll`). You can also use the `Assembly` directive to specify a source file to be compiled, by using the `Src` attribute. The following example demonstrates both of these attributes being used with an `Assembly` directive:

```
<%@ Assembly Name="System" %>
<%@ Assembly Src="MyNewAssembly.js" %>
```

Namespaces can also be imported and mapped to tag prefixes. When a namespace is mapped to a tag prefix, you can use special syntax in the ASP.NET file to create an instance of that control on the page. In addition to importing an entire namespace and attaching a tag prefix, you can import a single control and map it to a tag prefix by using a tag name. The following example shows how a control might be included on a page, using the tag prefix and tag name:

```
<body>
    <tagprefix:tagname runat="server" />
</body>
```

To register an entire namespace of classes to a tag prefix using the `Register` directive, you make use of the `TagPrefix`, `Namespace`, and `Assembly` attributes. `TagPrefix` is the string that should be associated with the namespace, `Namespace` is the namespace that needs to be imported, and `Assembly` is the assembly that contains the namespace in question. ASP.NET registers a tag prefix by default. The following is an example of code you could use to import the ASP.NET controls if they were not registered by default:

```
<%@ Register TagPrefix="asp"
    Namespace="System.Web.UI.WebControls"
    Assembly="System.Web" %>
<%@ Register TagPrefix="asp"
    Namespace="System.Web.UI.HtmlControls"
    Assembly="System.Web" %>
```

In order to use the `Register` directive to import a single control and map it to a tag prefix/tag name pair, you'll have to use the `TagPrefix`, `TagName`, and `Assembly` attributes. This is very common in code because one of the most popular features of ASP.NET is the user control, and all user controls have to be imported in this manner. The `TagPrefix` attribute, `TagName` attribute, and `Src` attribute are used to denote this type of `Register` directive. The `Src` tag should point to the user control, which is generally a control that with the `.ascx` file extension. The following is an example of how to import a user control and map it to a tag prefix/tag name pair:

```
<%@ Register TagPrefix="Acme" TagName="NewsControl" Assembly="News.ascx" %>
<body>
    <Acme:NewsControl runat="server" />
</body>
```

Server-Side Code Blocks, In-line Code, and Code-Behind

By using the various directives described in the section "ASP.NET Page-Level Directives," you were able to generate a couple small sample pages. Those code samples would probably compile and run as ASP.NET pages, if the required assemblies and controls were in place on the machine, but they don't do anything important. In order to interact with a page, you have to create some server-side code blocks. There are several ways to do this, including the use of in-line code, server side code blocks, and code-behind, as discussed in the following subsections.

Creating Server-Side Code Blocks and In-line Code

The easiest way to create server-side code is to generate ASP-style in-line code blocks. You can do this by setting the language in the `Page` directive to JScript so that the compiler knows which language is being used in the in-line blocks. After you have established the language for the page in the `Page` directive, you can use the begin server-side code tag (`<%`) and the end server-side code tag (`%>`)to inject code into the HTML page. You could specify simple server-side in-line code as in the following example:

```
<%@ Page Language="JScript" %>
<body>
    <% for(int i = 1; i < 8; i++) { %>
        <font size="<%=i%>">Welcome to JScript .NET</font>
    <% } %>
</body>
```

The in-line code syntax example is actually considered bad coding practice in ASP.NET. In ASP there wasn't a server-side event or control model, so the only way to work with various HTML elements was to put code in-line within a page to generate the content. With ASP.NET, code can be placed within a single code block at the top of the page or within a separate file altogether and still control portions of the HTML rendering and output process. To do this, you use a server-side script instead of the in-line code syntax. This tag is basically the same as the client-side script blocks commonly used in Dynamic HTML (DHTML) scripting. The only difference between the client-side script block and the server-side script block is the introduction of the `runat` attribute. If the `runat` attribute is given the value of `server`, then the ASP.NET runtime knows to compile this code so that it can be used on the server side rather than being passed to the client side.

A server-side script block can also contain the target language. The language being used within the page must be set either at the page level or within each script block on the page. If the page and script blocks specify different languages, an error occurs. If two script blocks contain different languages, an error also occurs. The following example demonstrates a server-side code block that specifies language:

```
<script language="JScript" runat="server">
</script>
```

In order for code within server-side code blocks to affect the HTML output and rendering stage, some of the controls must be made to run on the server. The concept that controls within an HTML page are somehow instantiated on the server is a new idea in ASP.NET. To turn a normal HTML control into a server control, you simply need to add the `runat="server"` attribute/value pair to the start tag. In many cases ASP.NET maps the control in question to one of the `HtmlControl` controls. If it can't find one that matches perfectly, it uses the `HtmlGenericControl` control, which is capable of working with any existing HTML tags. The following example demonstrates the creation of some server-side controls:

```
<body>
    <!-- HTML Controls -->
    <span id="MessageSpan" runat="server"></span>
    <form id="ServerForm" runat="server">
    </form>

    <!-- Some of the ASP.NET controls -->
    <asp:Label id="Message" runat="server" />
    <asp:DataGrid id="MyGrid" runat="server" />
</body>
```

Code-Behind

With code-behind, you write code outside the document against which you're coding. When you use this technique, a single server-side application might consist of a display page or an ASP.NET page and also a source code page that resides either in the same location as the ASP.NET page or in a precompiled assembly.

Code-behind is primarily used by the Visual Studio .NET Web forms designers and isn't generally meant to be programmed by hand from the ground up. The merits for the classic text editor programmer of having everything in one page far outweigh the benefits of having code in another file and having to have both files open at the same time. In Visual Studio .NET, the reason for splitting code from the HTML and ASP.NET markup is obvious. It's easier for the designer to auto-generate content separately from code and keep the two in sync visually as changes are made rather than changing the code and text for the pages.

This section focuses on the basic concepts of setting up code-behind by hand. If you have Visual Studio .NET, you might want to take a look at the power of the designer and the ease-of-use value it adds to ASP.NET development.

Code-behind is completed in two steps. First, you use the `Page` directive to point the ASP.NET page to its code-behind class and source file. You do this by using the `Inherits` and `Src` attributes, discussed earlier in the chapter, in the section "ASP.NET Page-Level Directives." The second step in code-behind is to create a source file for the `Page` directive attributes to reference. The source file for code-behind needs to contain a class that extends `System.Web.UI.Page`, and the name of this class is be the name of the class used in the `Inherits` attribute. Listings 14.1 and 14.2 demonstrate the most basic code-behind files possible.

LISTING 14.1 An ASP.NET Page that Uses Code-Behind

```
<%@ Page Inherits="MyClass" Src="CodeBehind.js" %>
```

LISTING 14.2 The Code Source for a Code-Behind Page

```
import System;
import System.Web;
import System.Web.UI;

public class MyClass extends System.Web.UI.Page {
}
```

Writing Basic Event Handlers

So far none of the examples in this chapter have actually done anything in regard to the ASP.NET server-side object model. But in order to break into the processing of an ASP.NET page, various events have to be handled. These events let you know, among other things, when it is safe to access server-side control collections and when it is safe to start rendering page content. Without these events, ASP.NET programming is impossible.

ASP.NET provides a special event hookup mechanism: You make some functions by using a special naming syntax, and ASP.NET automatically sets up those functions as callback functions for the various events fired by ASP.NET. This format takes the form of *`ObjectName_EventName`*, where *`ObjectName`* is generally `Page` and *`EventName`* is one of the events on the `Page` object. The following sections describe the `Page.Load`, `Page.PreRender`, `Page.Init`, and `Page.Error` events.

Trapping the `Page.Load` Event

The `Page.Load` event is the most important event in ASP.NET programming because it is where you can first access the server-side object model and thus affect controls on the page. It is also the first place where you have access to view state information for the various controls after a postback has occurred.

Trapping the `Page.Load` event is as easy as declaring a function with the correct signature and method name. ASP.NET looks for a function whose name has the syntax `Page_`*`EventName`*. This method can be used for both the `Load` event discussed here and any other events (as discussed later in this chapter, in the section "Using Other Page Object Events") that the `Page` object supports. The following example demonstrates server-side code trapping the `Page.Load` event and the syntax required to provide a method with the correct signature:

```
<script language="jscript" runat="server">
    function Page_Load(sender:Object, e:EventArgs) : void {
    }
</script>
```

After the `Page.Load` event is captured, you have access to all the server-side controls on the page. You can call functions or set properties on these server-side controls to affect their output. The following example demonstrates how to use a `Label` control to send a custom message to the client:

```
<script language="jscript" runat="server">
    function Page_Load(sender:Object, e:EventArgs) : void {
        Message.Text = "Message From Code";
    }
</script>

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

You should be able to clearly see now how much power server-side controls and code blocks give you. You can use server-side controls to customize page output. Many controls even have a `Visible` property that lets you disable rendering for the control and truly customize what is visible to the client.

Using Other `Page` Object Events

In addition to the `Load` event, the `Page` object also contains the `Init` event (for the phase that occurs before controls are loaded and the server-side object model is ready), the `PreRender` event (for the phase that occurs after the `Load` event and immediately before controls are set to render content), and the `Error` event (which is raised whenever an unhandled runtime error occurs on the page).

Depending on the location or time during page processing in which an event is called, various things are either available or not available. The `Init` event is called while the controls on the page are still being initialized, when the server-side object model isn't ready yet. This means that you don't have access to server-side controls yet, but you can still do data initialization or whatever else might be required for use later, when the other events are called. The following example demonstrates how to trap the `Page.Init` event and set up a page-level variable for use when the `Page.Load` event is called:

```
<script language="jscript" runat="server">
    public var dsn:String;

    function Page_Init(sender:Object, e:EventArgs) : void {
        dsn = "Database Connection String";
    }
    function Page_Load(sender:Object, e:EventArgs) : void {
        // Code that uses the DSN.  Other controls that know about the
        // variable on the page can also use the variable.  That is why
        // it was set in the Init phase and not the Load phase.
    }
</script>
```

The `PreRender` event is called after the `Load` event but immediately before the HTML output is rendered. This event is useful for making final modifications to controls and to view state right before the render phase is called. This is the final place such modifications can be made in order for them to be persisted in the view state. If changes are made while the page is actually rendering, the changes are evident in the output HTML but the view state does not reflect those changes after the page is posted back. The following example demonstrates how to access a control right before the page is rendered by trapping the `PreRender` event:

```
<script language="jscript" runat="server">
    function Page_Load(sender:Object, e:EventArgs) : void {
        Message.Text = "Message Set During Load Event";
    }
    function Page_PreRender(sender:Object, e:EventArgs) : void {
        Message.Text = "Message ReSet During the PreRender Event";
    }
```

```
</script>

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

In addition to the callback functions for hooking into the page rendering process, there are also events that are called for different states in the page. A good example of this is the runtime error state. Normally, when a runtime error occurs and is not trapped in the page's code, ASP.NET generates an error message and sends the message to the client. By trapping the `Error` event, the page can customize what happens when an error occurs by redirecting to a custom page. The following example demonstrates how to redirect to a custom page when a runtime error occurs:

```
<script language="jscript" runat="server">
    function Page_Load(sender:Object, e:EventArgs) : void {
        // Code that causes an error
    }
    function Page_Error(sender:Object, e:EventArgs) : void {
        Response.Redirect("MyPrettyErrorPage.htm");
    }
</script>

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

Summary

This chapter is in not meant to be a comprehensive ASP.NET tutorial, but to introduce you to the various server-side programming tools available in the new ASP.NET programming model and JScript .NET. In this chapter you have learned how to create basic ASP.NET applications and how to customize the ASP.NET page to include all the various features and options that make it such a powerful new tool. You have also learned how to create in-line script blocks and server-side script blocks. In addition, you have learned a bit about code-behind code blocks.

You have also learned in this chapter how to use the ASP.NET event model to tap into the page-rendering process and how to use the server-side control and object model.

Chapter 15, "Windows Forms Programming in JScript .NET," discusses the Windows forms programming model, the exciting world of user interface programming, and the how to create windowed applications.

CHAPTER

15

Windows Forms Programming in JScript .NET

IN THIS CHAPTER

For most programmers the GUI application is probably the most coveted and misunderstood application. Everyone wants to make something with a pretty and convenient user interface for the user—something with a WOW effect, often taking more time to generate the WOW than to get the job done. What most programmers don't like to admit is that GUI programming takes the majority of time in any project. It's the GUI that takes multiple revisions to get right, often at the cost of the functionality of the application. For this reason I've waited until the end of the book to introduce Windows Forms programming. Nearly every task can be completed in a more timely manner when a console application is used, so the basic concepts of JScript and the CLR (common language runtime) are more easily described using console application code.

Now it is time to finish the book on a UI note. This chapter introduces the features of Windows Forms that make up the most basic portions of UI programming. We'll start with a basic Windows Forms application consisting of `Form`, using some of the properties, and the mechanism for launching and displaying that form. We'll make use of the Windows executable compiler target to create these applications without a console—this will add a more professional Windows effect.

With a basic operational form, the next step is to add buttons, labels, and text controls. Here you'll learn about various control events and the event programming model.

One of the greatest features of Windows Forms is the layout manager. This manager automatically arranges controls based on the order in which the controls were added to the form and according to some additional docking and layout properties available on all Windows Forms controls.

Any Windows Forms applications wouldn't be complete without the menus. Menus are extremely easy to implement using Windows Forms. We'll discover how to use the `MainMenu` control to implement a top-level menu, how to use the `MenuItem` control to add individual menus, and how to add submenus. Menus don't work unless the proper events are trapped, so we'll look at how to handle menu events in different ways, depending on which type of menu the `MenuItem` represents.

With menus come dialogs. Dialogs are often used as the implementation for Open and Save menus, so we'll take a look at the `OpenFileDialog` and `SaveFileDialog` classes for opening and saving files. The implementation of both of these dialogs will make use of file IO, so refer to Chapter 10, "File Access," if needed.

This should be all the information needed to make JPad. I'm not going to lie and say that this is going to be the end-all editor and you'll never use another. This will be the simplest text editor possible and will implement very few features. It certainly won't be something that will be useful later without quite a bit more work. It will, however, demonstrate how much code it takes to make even the most basic GUI application.

Setting Up a Basic Form

Getting a basic form (windowed application) running is actually fairly trivial. The various Windows Forms controls all have such a large number of properties and methods that I won't be able to show diagrams throughout the text for each of the classes and for the Windows Forms namespace. You should try to use the Windows Forms documentation along with the discussions on Windows Forms features in this chapter, and to try out some of the properties and methods that are missed in the text.

The first step in a Windows Forms application is to make a new form class. This class should be derived from `System.Windows.Forms.Form` or from some other class that is descended from the `Form` class. This is actually the minimum amount of work required to get a new form ready to run, but we'll do some additional work to make it a bit more interesting. These additional steps include setting up a basic function for initializing the new form, customizing the look and startup location of the form, and setting up a main method for launching the form under an STAThread (Single Threaded Apartment).

The new class should have some sort of constructor that initializes all the various controls and UI for the form. This is where events are hooked up, menus are added, controls are added, and basic form properties are initialized. The Windows Forms designer does this by creating an `InitializeComponent()` function, which is called from within the constructor. This makes it easy to separate any basic control and layout code from more complex initializations such as opening database connections or setting up networking or Web requests.

Because `InitializeComponent` is where all the basic form properties get set up, there should be something to put in there. Every form has various properties that affect the size, layout, and state of the form after it is made visible. Using the `Text` property can set the title for the form. The size of the form can be set by using the `Size` property and assigning a new instance of the `Size` class to the property. It might be easier to remember the `Height` and `Width` properties because they have more standard names, and they are equally useful in setting up a form.

Setting up the basic window's location is achieved by either setting the `Location` property with a new `Point` class, or setting both the `Left` and `Top` properties. To allow Windows to place the form, the `StartPosition` is assigned one of the values in the `FormStartPosition` enumeration. The available values are `CenterParent`, `CenterScreen`, `Manual`, `WindowsDefaultBounds`, and `WindowsDefaultLocation`. The `CenterParent` and `WindowsDefaultBounds` will both affect the window's size as well as its location, so more care needs to be taken when using these two options in order to get the correct results. The `WindowState` can also be specified using a `FormWindowsState`. The `WindowState` specifies whether a form is in a normal state, a maximized state, or a minimized state. With these properties the form should be sufficiently initialized to look nice onscreen. But how is a form launched?

To launch a form, some global startup code needs to be generated. Normally, this is taken care of by creating an object in global code and running the necessary methods. However, there is a special attribute that needs to be applied to the function that launches the form. For this reason it is best to create a static function and apply the `STAThreadAttribute` to the function declaration. This forces the threading model into an appropriate state to run Windows Forms. Inside of this static method, the `Application.Run` method is called on a new instance of the form that has been created. Listing 15.1 demonstrates the creation of a new form class, setting all the properties discussed here in this section, and then finally launches this form and displays it onscreen.

LISTING 15.1 Launching a Basic Windows Form

```
import System;
import System.Drawing;
import System.Windows.Forms;

public class BasicForm extends Form {

    public function BasicForm() {
        InitializeComponent();
    }

    private function InitializeComponent() : void {
        this.Text = "Basic Windows Forms";
        this.Height = 400;
        this.Width = 500;
        this.WindowState = FormWindowState.Normal;
        this.StartPosition = FormStartPosition.CenterScreen;
    }

    public STAThreadAttribute() static function Main(Args:String[]) : void {
        Application.Run(new BasicForm());
    }
}

BasicForm.Main(Environment.GetCommandLineArgs());
```

Buttons, Labels, and Text Controls

I've never heard anyone talk about how neat a blank window was. It was certainly never considered a WOW feature (well, I once made a fully functional round window when using custom regions wasn't widely known about, and that was kind of a WOW). To get the user's attention, it might be a bit better to add some buttons, labels, and text controls.

To accomplish this feat, the form is going to have to change a bit more. In this section, Listing 15.1 is built upon in order to add the new controls. This task consists of a few steps:

1. Add each of the controls as a private field of the form. You do this so that the controls can be accessed anywhere in the form.
2. Initialize the fields in the `InitializeComponent` section of the form. You also need to add them to the form's `Controls` collection or they won't show up when the form is shown.
3. Make sure that the components are displayed on the resulting form. This is the most crucial step because often placement of some controls will affect other controls. This can sometimes push controls outside of the bounds of the form window and make them inaccessible. If you are using the form layout manager, this isn't usually an issue, but if you are using custom layouts, you'll want to make sure that your code works and places the controls correctly.

In addition to initializing and setting simple properties on the new controls, you can set some events. However, in this sample, you are only going to trap the button's `Click` event. To accomplish this task, the event hookup mechanism is used to add an event handler to the `Click` event. After the event is called, it should probably do some work with the `TextBox` control and the `Label` control. Listing 15.2 demonstrates creating and adding controls to a form, laying those controls out on the form using a manual layout, and finally trapping the `Click` method of the button to update the `Label` control based on the text in the `TextBox` control.

LISTING 15.2 Implementing Form Controls and Trapping Events

```
import System;
import System.Drawing;
import System.Windows.Forms;

public class BasicForm extends Form {
    private var textInput:System.Windows.Forms.TextBox;
    private var labelDisplay:System.Windows.Forms.Label;
    private var button1:System.Windows.Forms.Button;
    private var totalControls:int;

    public function BasicForm() {
        InitializeComponent();
    }

    private function Button1_Click(sender:Object, e:EventArgs) : void {
        this.labelDisplay.Text = "Display: " + this.textInput.Text;
    }
```

LISTING 15.2 continued

```
    private function InitializeComponent() : void {
        this.Text = "Basic Windows Forms";
        this.Height = 160;
        this.Width = 300;
        this.WindowState = FormWindowState.Normal;
        this.StartPosition = FormStartPosition.CenterScreen;

        this.totalControls = 3; // This is used for manual layout

        this.textInput = new TextBox();
        this.textInput.Text = "Initial Text";
        this.textInput.Height = (this.ClientSize.Height - ((totalControls+1) *
➥10)) / totalControls;
        this.textInput.Width = this.ClientSize.Width - 20;
        this.textInput.Left = 10;
        this.textInput.Top = 10;

        this.labelDisplay = new Label();
        this.labelDisplay.Text = "Display: " + this.textInput.Text;
        this.labelDisplay.Height = (this.ClientSize.Height - ((totalControls+1) *
➥10)) / totalControls;
        this.labelDisplay.Width = this.ClientSize.Width - 20;
        this.labelDisplay.Left = 10;
        this.labelDisplay.Top = this.textInput.Top + this.textInput.Height + 10;

        this.button1 = new Button();
        this.button1.Text = "Update Label";
        this.button1.Height = (this.ClientSize.Height - ((totalControls+1) * 10))
➥/ totalControls;
        this.button1.Width = this.ClientSize.Width - 20;
        this.button1.Left = 10;
        this.button1.Top = this.labelDisplay.Top + this.labelDisplay.Height + 10;
        this.button1.add_Click(this.Button1_Click);

        this.Controls.Add(this.textInput);
        this.Controls.Add(this.labelDisplay);
        this.Controls.Add(this.button1);
    }
```

LISTING 15.2 continued

```
    public STAThreadAttribute() static function Main(Args:String[]) : void {
        Application.Run(new BasicForm());
    }
}

BasicForm.Main(Environment.GetCommandLineArgs());
```

Be sure to compile and run Listing 15.2 and interact with the controls so that you understand exactly what you've created. Most of the new code is related to creating the button, label, and text box to interact with the user. This is all linked together with the `Click` event on the button which updates the label with any text that the user has typed into the text box control. You'll find that most of the code actions in Windows Forms programming are started in response to some user action on a control.

Using the Docking and Layout Feature

You now have a working application that displays some controls, and uses event-based programming to update properties on the controls in response to some action by the user. Even if the work is only updating a label with some information typed into a text box, it is certainly better than nothing. In addition to adding the controls, Listing 15.2 includes quite a bit of layout code in the `InitializeComponent` method in order to position the button, label, and text box. Programming this layout code can make an application hard to manage. Imagine if 10 or 15 more controls were added—in no time at all, the simple calculations would become quite complex. For this reason Windows Forms has a special layout manager that works based on the `Dock` and `Anchor` properties of the controls.

The first property is the `Dock` property. The `Dock` property is responsible for controlling which edges of the screen a control is docked to. This can enable some fairly complex automatic layouts in that when a form is resized, the layout manager will also resize and move the controls to make sure that the new position conforms to the `Dock` property. The possible positions are `Left`, `Right`, `Top`, `Bottom`, `Fill`, and `None`. `None` is the default value of the `Dock` property. Another feature of the `Dock` property is that if multiple controls are docked to the same edge, they won't cover one another. They will bump flush with one another. This enables easy layout of all controls based on common edges.

There is one additional layout property: the `Anchor` property. It is responsible for rooting a control in place rather than stretching a control flush with the edges of the container. If a control is placed 10 pixels from the left side of the form and the `Anchor` property is set to `AnchorStyles.Left`, then no matter how the control is resized or moved, the control will always remain 10 pixels from the left edge. The possible values are `Top`, `Bottom`, `Left`, `Right`, and None. These values can be combined to anchor the control on more than one side. Listing 15.3 demonstrates the use of the Dock and Anchor properties for layout purposes.

LISTING 15.3 The Dock and Anchor Properties

```
import System;
import System.Drawing;
import System.Windows.Forms;

public class BasicForm extends Form {
    private var textInput:System.Windows.Forms.TextBox;
    private var labelDisplay:System.Windows.Forms.Label;
    private var button1:System.Windows.Forms.Button;
    private var totalControls:int;

    public function BasicForm() {
        InitializeComponent();
    }

    private function Button1_Click(sender:Object, e:EventArgs) : void {
        this.labelDisplay.Text = "Display: " + this.textInput.Text;
    }

    private function InitializeComponent() : void {
        this.Text = "Basic Windows Forms";
        this.Height = 160;
        this.Width = 300;
        this.WindowState = FormWindowState.Normal;
        this.StartPosition = FormStartPosition.CenterScreen;

        this.totalControls = 3; // This is used for manual layout

        this.textInput = new TextBox();
        this.textInput.Text = "Initial Text";
        this.textInput.Top = 10;
        this.textInput.Left = 10;
        this.textInput.Width = this.ClientSize.Width - 20;
        this.textInput.Anchor = AnchorStyles.Top | AnchorStyles.Left |
            AnchorStyles.Right;

        this.labelDisplay = new Label();
        this.labelDisplay.Text = "Display: " + this.textInput.Text;
        this.labelDisplay.Width = this.ClientSize.Width - 20;
        this.labelDisplay.Top = this.textInput.Bottom + 10;
        this.labelDisplay.Left = 10;
```

LISTING 15.3 continued

```
        this.labelDisplay.Anchor = AnchorStyles.Left | AnchorStyles.Right |
            AnchorStyles.Top;

        this.button1 = new Button();
        this.button1.Text = "Update Label";
        this.button1.Width = this.ClientSize.Width - 20;
        this.button1.Left = 10;
        this.button1.Anchor = AnchorStyles.Left | AnchorStyles.Right;
        this.button1.Dock = DockStyle.Bottom;
        this.button1.add_Click(this.Button1_Click);

        this.Controls.Add(this.textInput);
        this.Controls.Add(this.labelDisplay);
        this.Controls.Add(this.button1);
    }

    public STAThreadAttribute() static function Main(Args:String[]) : void {
        Application.Run(new BasicForm());
    }
}

BasicForm.Main(Environment.GetCommandLineArgs());
```

You'll notice that Listing 15.3 doesn't have nearly as much layout code as Listing 15.2. This is because you've used the layout manager properties, `Dock` and `Anchor`, instead of the static layout properties, `Left`, `Right`, `Width`, and `Height`. An additional bonus to using the layout manager is that even when the form is resized, the controls will automatically position to work with the new form size.

Implementing Simple Menus

The next step in a great program is to implement some menus. Every graphical application has a limited amount of screen space to display command and control elements. The menus enable the application to maintain hidden command items. They also offer a method for switching between different UI displays because they are always available at the top of the application window. In addition to top-level or main menus, context menus can be enabled that pop up based on the control the mouse is hovering over. These can also be very handy because they put commands only a mouse click away. Context or pop-up menus are outside the scope of this chapter, so we'll focus only on creating the main menu and creating some submenus.

Creating a main menu is as simple as creating some additional form fields to hold the new menu objects. First a `MainMenu` object must be created and assigned to the `Menu` property on the form. The `MainMenu` object is a specialized menu that contains special drawing and docking functionality so that it can exist at the top of the application. The `MainMenu` object contains two methods for adding menu items. A single item can be adding using the `Add` method of the `MenuItems` property and passing in a new `MenuItem` class. An array of `MenuItems` can be added using the `AddRange` method of the `MenuItems` property.

Before you can add some menu items, you first need to create them. The simplest menus can be created using only the `MenuItem` constructors. More complex features are available through the `MenuItem` properties, but we won't be using these. So all that is required to generate a menu are a menu name and the event handler used for the menu click callback. There is one special note about the menu name. The menu name can precede the letter of the Alt-activated hotkey. This can be used to quickly access the menu using the keyboard. Additionally, the sample code will be using a third parameter in the constructor. The third parameter is a member of the `Shortcut` enumeration. Shortcuts allow the user to access specific commands with the Alt keystrokes. For example, in Listing 15.4 pressing Ctrl+O, which is a very common shortcut for an Open menu in an application, will access the Open menu. Listing 15.4 demonstrates the creation of a `MainMenu` with a File menu and several File menu commands.

LISTING 15.4 Creating Menus Using Windows Forms

```
import System;
import System.Drawing;
import System.Windows.Forms;

public class BasicForm extends Form {
    private var mainMenu:System.Windows.Forms.MainMenu;
    private var mnuFile:System.Windows.Forms.MenuItem;
    private var mnuOpen:System.Windows.Forms.MenuItem;
    private var mnuSave:System.Windows.Forms.MenuItem;
    private var mnuExit:System.Windows.Forms.MenuItem;

    public function BasicForm() {
        InitializeComponent();
    }

    private function Form_Open(sender:Object, e:EventArgs) : void {
    }

    private function Form_Save(sender:Object, e:EventArgs) : void {
    }
```

LISTING 15.4 continued

```
    private function Form_Exit(sender:Object, e:EventArgs) : void {
        this.Close();
    }

    private function InitializeComponent() : void {
        this.Text = "Basic Windows Forms";
        this.Height = 160;
        this.Width = 300;
        this.WindowState = FormWindowState.Normal;
        this.StartPosition = FormStartPosition.CenterScreen;

        this.mainMenu = new MainMenu();

        this.mnuFile = new MenuItem("&File");
        this.mnuOpen = new MenuItem("&Open", this.Form_Open, Shortcut.CtrlO);
        this.mnuSave = new MenuItem("&Save", this.Form_Save, Shortcut.CtrlS);
        this.mnuExit = new MenuItem("E&xit", this.Form_Exit, Shortcut.CtrlX);

        this.mainMenu.MenuItems.Add(this.mnuFile);
        this.mnuFile.MenuItems.AddRange(
            MenuItem[](
                [this.mnuOpen, this.mnuSave,
                 new MenuItem("-"), this.mnuExit]
            ));
        this.Menu = this.mainMenu;
    }

    public STAThreadAttribute() static function Main(Args:String[]) : void {
        Application.Run(new BasicForm());
    }
}

BasicForm.Main(Environment.GetCommandLineArgs());
```

Creating menus is a very common programming task, and most users will expect a familiar menu on your application. Through Listing 15.4 you've learned to create menus, apply Alt-key activation shortcuts, and apply keyboard shortcuts. You'll also notice that the Open and Save menu items don't currently have any event implementations. The next section expands on these two menu items to implement Open and Save dialogs.

Using the Common Dialog Controls

Many common user operations include working with the file system—more important, opening and saving files. In the past, programmers have had to generate the dialogs for these operations from scratch. This task can be quite complex when the dialogs have to traverse the file system, give access to common directories, and also perform identically on network shares. Eventually, common dialog controls were shipped with the operating system. This happened in Windows 95, and programmers have been using these dialog controls ever since. In the more recent Windows 2000 and Windows XP, the dialogs contain 20 to 30 different controls and have gained such user recognition that programming custom dialogs for these operations is considered a GUI programming no-no.

With these concepts in mind, Listing 15.5 works with the `OpenFileDialog` and the `SaveFileDialog`. The syntax for working with each of these is nearly identical, so examining one of the controls will be enough to work with both of them. Because the dialogs will both be used quite often in the application, they will both be initialized in the `InitializeComponent` function the same as the rest of the controls that have been added to the form. To initialize each control, the `InitialDirectory` must be set so that the controls know which directory to use the first time they are opened. This can often be set to the current working directory for the application. In the sample code the location of the assembly file or executable is used.

The `Filter` property contains the listing of file filters to use. These are specified in the form display name followed by the pipe symbol (`|`) followed by the file mask. More than one filter can be specified by using another pipe symbol and continuing with this format for as many display and pattern pairs as needed. The `FilterIndex` property can be used to set which filter is selected by default when the dialog loads. This index is 1-based and not 0-based. Generally, the first item in the pattern is the value of the `FilterIndex` property. The final property to set is the `RestoreDirectory` property. When this is set, the dialog will automatically open to the last directory the user selected in the dialog.

After the dialogs are initialized, they still have to be opened with some code. Both the Save and the Open dialogs have a `ShowDialog` function. This function opens the dialog in question and processes the user input. If the user picks a file and clicks OK, the dialog will return the result `DialogResult.OK`. This result can be tested for and used to determine whether the program should perform some additional action. This action can be to open or save by using the file in the `FileName` property, or one of the special dialog methods can be used for special functionality. One such method is the `OpenFile` method that can be used to get a `Stream` object for the file the user selected. This `Stream` can be read from or written to depending on whether the program used the `SaveFileDialog` or the `OpenFileDialog`. Listing 15.5 demonstrates the basic code needed to use the Open and Save dialog controls. Listing 15.6, in the section

"Wrapping Up in JPad," will further demonstrate some code that will use the results of the Open and Save dialogs to actually open and save files.

LISTING 15.5 Using Common Dialogs

```
import System;
import System.Drawing;
import System.Reflection;
import System.Windows.Forms;

public class BasicForm extends Form {
    private var mainMenu:System.Windows.Forms.MainMenu;
    private var mnuFile:System.Windows.Forms.MenuItem;
    private var mnuOpen:System.Windows.Forms.MenuItem;
    private var mnuSave:System.Windows.Forms.MenuItem;
    private var mnuExit:System.Windows.Forms.MenuItem;

    private var labelFile:System.Windows.Forms.Label;

    private var openDialog:System.Windows.Forms.OpenFileDialog;
    private var saveDialog:System.Windows.Forms.SaveFileDialog;

    public function BasicForm() {
        InitializeComponent();
    }

    private function Form_Open(sender:Object, e:EventArgs) : void {
        if ( openDialog.ShowDialog() == DialogResult.OK ) {
            this.labelFile.Text = "Open: " + openDialog.FileName;
        }
    }

    private function Form_Save(sender:Object, e:EventArgs) : void {
        if ( saveDialog.ShowDialog() == DialogResult.OK ) {
            this.labelFile.Text = "Save: " + saveDialog.FileName;
        }
    }

    private function Form_Exit(sender:Object, e:EventArgs) : void {
        this.Close();
    }

    private function InitializeComponent() : void {
        this.Text = "Basic Windows Forms";
        this.Height = 160;
        this.Width = 300;
```

Listing 15.5 continued

```
        this.WindowState = FormWindowState.Normal;
        this.StartPosition = FormStartPosition.CenterScreen;

        this.openDialog = new OpenFileDialog();
        this.openDialog.InitialDirectory =
➥Assembly.GetExecutingAssembly().Location;
        this.openDialog.Filter = "Text Files (*.txt)|*.txt|All Files
➥(*.*)|*.*";
        this.openDialog.FilterIndex = 1;
        this.openDialog.RestoreDirectory = true;

        this.saveDialog = new SaveFileDialog();
        this.saveDialog.InitialDirectory =
➥Assembly.GetExecutingAssembly().Location;
        this.saveDialog.Filter = "Text Files (*.txt)|*.txt|All Files
➥(*.*)|*.*";
        this.saveDialog.FilterIndex = 1;
        this.saveDialog.RestoreDirectory = true;

        this.labelFile = new Label();
        this.labelFile.Dock = DockStyle.Fill;

        this.mainMenu = new MainMenu();

        this.mnuFile = new MenuItem("&File");
        this.mnuOpen = new MenuItem("&Open", this.Form_Open, Shortcut.CtrlO);
        this.mnuSave = new MenuItem("&Save", this.Form_Save, Shortcut.CtrlS);
        this.mnuExit = new MenuItem("E&xit", this.Form_Exit, Shortcut.CtrlX);

        this.mainMenu.MenuItems.Add(this.mnuFile);
        this.mnuFile.MenuItems.AddRange(
            MenuItem[](
                [this.mnuOpen, this.mnuSave,
                 new MenuItem("-"), this.mnuExit]
            ));
        this.Menu = this.mainMenu;

        this.Controls.Add(this.labelFile);
    }
```

LISTING 15.5 continued

```
    public STAThreadAttribute() static function Main(Args:String[]) : void {
        Application.Run(new BasicForm());
    }
}

BasicForm.Main(Environment.GetCommandLineArgs());
```

Wrapping Up in JPad

Sometimes it is important to wrap up a series of ideas into a final application or idea that demonstrates a proof of concept. The proof here is that a real application can be generated from everything shown in this chapter. In Listing 15.6 you will integrate a simple text box with the code from Listing 15.5 that will enable a user to open, edit, and save text files.

You should probably have some challenges to program that aren't already in Listing 15.6. It would be nice to make sure that the user knows the current state the application is in—for instance, whether or not the text has been modified. If it has been modified and the user tries to save, it would be nice to let the user know this and ask whether he wants to cancel the current action or proceed. It might also be nice to display the currently opened file as part of the form title, and make sure that it updates after a save as well. These are just a few of the advancements JPad could use. Listing 15.6 demonstrates all the principles needed for the most basic of text editors.

LISTING 15.6 JPad

```
import System;
import System.Drawing;
import System.IO;
import System.Reflection;
import System.Windows.Forms;

public class BasicForm extends Form {
    private var mainMenu:System.Windows.Forms.MainMenu;
    private var mnuFile:System.Windows.Forms.MenuItem;
    private var mnuOpen:System.Windows.Forms.MenuItem;
    private var mnuSave:System.Windows.Forms.MenuItem;
    private var mnuExit:System.Windows.Forms.MenuItem;

    private var textFile:System.Windows.Forms.TextBox;

    private var openDialog:System.Windows.Forms.OpenFileDialog;
    private var saveDialog:System.Windows.Forms.SaveFileDialog;
```

LISTING 15.6 continued

```
    public function BasicForm() {
        InitializeComponent();
    }

    private function Form_Open(sender:Object, e:EventArgs) : void {
        if ( openDialog.ShowDialog() == DialogResult.OK ) {
            var sr:StreamReader = new StreamReader(openDialog.FileName);
            this.textFile.Text = sr.ReadToEnd();
            sr.Close();
        }
    }

    private function Form_Save(sender:Object, e:EventArgs) : void {
        if ( saveDialog.ShowDialog() == DialogResult.OK ) {
            var sw:StreamWriter = new StreamWriter(saveDialog.FileName);
            sw.Write(this.textFile.Text);
            sw.Close();
        }
    }

    private function Form_Exit(sender:Object, e:EventArgs) : void {
        this.Close();
    }

    private function InitializeComponent() : void {
        this.Text = "Basic Windows Forms";
        this.Height = 160;
        this.Width = 300;
        this.WindowState = FormWindowState.Normal;
        this.StartPosition = FormStartPosition.CenterScreen;

        this.openDialog = new OpenFileDialog();
        this.openDialog.InitialDirectory =
        ➥Assembly.GetExecutingAssembly().Location;
        this.openDialog.Filter = "Text Files
        ➥ (*.txt)|*.txt|All Files (*.*)|*.*";
        this.openDialog.FilterIndex = 1;
        this.openDialog.RestoreDirectory = true;

        this.saveDialog = new SaveFileDialog();
        this.saveDialog.InitialDirectory =
        ➥Assembly.GetExecutingAssembly().Location;
        this.saveDialog.Filter =
        ➥ "Text Files (*.txt)|*.txt|All Files (*.*)|*.*";
```

LISTING 15.6 continued

```
        this.saveDialog.FilterIndex = 1;
        this.saveDialog.RestoreDirectory = true;

        this.textFile = new TextBox();
        this.textFile.AcceptsReturn = true;
        this.textFile.Multiline = true;
        this.textFile.Dock = DockStyle.Fill;

        this.mainMenu = new MainMenu();

        this.mnuFile = new MenuItem("&File");
        this.mnuOpen = new MenuItem("&Open", this.Form_Open, Shortcut.CtrlO);
        this.mnuSave = new MenuItem("&Save", this.Form_Save, Shortcut.CtrlS);
        this.mnuExit = new MenuItem("E&xit", this.Form_Exit, Shortcut.CtrlX);

        this.mainMenu.MenuItems.Add(this.mnuFile);
        this.mnuFile.MenuItems.AddRange(
            MenuItem[](
                [this.mnuOpen, this.mnuSave,
                 new MenuItem("-"), this.mnuExit]
            ));
        this.Menu = this.mainMenu;

        this.Controls.Add(this.textFile);
    }

    public STAThreadAttribute() static function Main(Args:String[]) : void {
        Application.Run(new BasicForm());
    }
}

BasicForm.Main(Environment.GetCommandLineArgs());
```

Summary

Okay, okay, I must admit that this chapter was a great load of fun, and I probably should have done some more Windows programming throughout the book. However, it took this entire chapter to show even the most basic Windows Forms programming techniques. After reading this chapter, you should be able to create a basic form or at least use Listing 15.1 as a template for a base program. You should generally try to keep a template around so that you don't have to code everything from scratch. You should also keep common form layouts saved for templates so that you can reuse them later as windows or dialogs in your application.

The section "Buttons, Labels, and Text Controls" demonstrated adding controls and initializing their state. This was the first time events were used in this chapter. Windows Forms is a highly event-driven programming model, so learning about events was necessary. Events were again used for the menus later in the chapter. This section also showed how extremely difficult laying out even a small amount of controls on a form can be. For this reason it is recommended that the auto layout manager be used. The section "Using the Docking and Layout Feature" showed a simple use of the layout manager by examining the `Dock` and `Anchor` properties on each of the controls. These properties are used even when resizing the form.

The section "Implementing Simple Menus" moved away from the controls on the form and more into the commands that can be executed from menus. The section explained the small difference between a `MainMenu` and a `MenuItem` and how to use each of these items to make basic menus. These menus were then used in the section "Using the Common Dialog Controls" to open some common dialogs. You learned that the common dialogs are useful for opening and saving files and giving users a familiar experience. Using the common dialogs certainly made it much easier to navigate the file system than having to custom-code enough UI to give the user the same abilities as the common dialogs.

With each of these principles, the JPad application was created. It doesn't show any new principles, but it does show the integration of some System.IO code along with the windows code to make an application that actually does something.

This chapter ends the book. I recommend that you use Chapters 10 through 13 as references for some of the more common CLR namespaces. You can also look at the compiler reference in Chapter 9, "JScript .NET Compiler Features," or the shorter version available in Appendix A, "JScript .NET Compiler Reference."

APPENDIX

A

JScript .NET Compiler Reference

The JScript .NET compiler has quite a few options that can aid in various programming tasks. Many of these options are discussed in depth in Chapter 9, "JScript .NET Compiler Features," but for those who need a refresher, a quick reference is provided here. This is the basic command-line syntax for the JScript .NET compiler:

```
jsc.exe /t:{target option} /r:{references} [options] <source files>
```

Not every program will require that a target be specified and that some references be specified, because the JScript compiler will automatically set some defaults for you. For instance, the default target is a console executable. If you don't specify any reference lines and you don't turn off the `autoref` functionality, then usually the required assembly references will be generated automatically by the compiler. Table A.1 lists additional command-line options and features.

Table A.1 JScript .NET Compiler Options Guide

Option	*Description*
`/out:<target file>`	The name of the target assembly that the JScript .NET compiler will generate. This is generally a file ending in the .exe or .dll file extension.
`/target:exe` `/t:exe`	Specifies that the compiler should generate a console executable. Console executables are launched with an attached console that can be read from and written to using the `System.Console` class.
`/target:winexe` `/t:winexe`	Specifies that the compiler should generate a windows executable. The application will launch without a console window, and the application should generate its own UI or simply remain hidden.
`/target:library` `/t:library`	Specifies that the compiler should generate a library assembly. A library is generally used to hold commonly used code. Libraries can be used as references in other applications.

`/reference:<assembly>`
`/r:<assembly>`
	The compiler will reference types and metadata from the referenced assembly.

`/lib:<path>`	Specifies alternative library paths so that the compiler can find referenced assemblies. This is often used to change which version of an assembly is being referenced by changing the directory in which the assemblies are found.
`/autoref[+,-]`	Can be turned either off or on, and is on by default. The `autoref` instructs the compiler to try to automatically reference assemblies based on the import statements used in the source code.

TABLE A.1 continued

Option	*Description*
`/debug[+,-]`	Instructs the compiler to generate debug symbols for the compiled assembly. Debug symbols enable stack traces to contain the filenames and line numbers for functions to make finding errors much easier. The `debug` option is off by default.
`/fast[+,-]`	Disables certain language features to enable better code generation. Code generated with the `fast` option generally creates faster code but can often emit errors if legacy JScript features are used. An example of a disabled language feature when using the `fast` option is expando classes. By default, the `fast` option is off.
`/warnaserror[+,-]`	Treats warnings as errors. Any warning causes the compilation step to fail.
`/warn:<`*`level`*`>`	Sets the warning level (0–4). Various warnings have different warning levels. Setting a higher level generally means enabling more warnings, whereas a lower level enables fewer warnings.
`@<`*`file`*`>`	The compiler will read additional commands out of the command file. This is a good way to share commands between multiple programs. It also enables shorter command lines in case you compile your program by hand every time.
`/?` `/help`	Displays the JScript .NET compiler help output.

`/define:<`*`symbol`*`>`
`/d:<`*`symbol`*`>`
	Defines conditional compilation symbols. Commonly used symbols include the `DEBUG` and `TRACE` symbols, which are required for some CLR debug capabilities.

`/nologo`	Prevents the compiler from outputting the Microsoft logo and copyright information.
`/print[+,-]`	Enables or disables the use of the `print()` function. `print` is an implementation-independent method of outputting text. In JScript .NET this function directly evaluates to the `Console.WriteLine` function.
`/codepage:<`*`id`*`>`	Enables the use of a special code page when opening source files. This option enables the use of localized source files in different languages.

TABLE A.1 continued

Option	*Description*
`/lcid:<id>`	Enables the use of a special `lcid` for messages and the default code page.
`/nostdlib[+,-]`	Disables the import of the standard library (mscorlib.dll). This option also turns the default for the `autoref` switch to off.
`/utf8output[+,-]`	Instructs the compiler to emit an assembly using the UTF-8 character encoding.
`/versionsafe[+,-]`	Specifies the default versioning attribute for members not marked `override` or `new`. If `versionsafe` is turned on, members will be marked for `override`; otherwise, they will be marked with `new`.

INDEX

SYMBOLS