

Appendix B

A Comparison of Visual Basic .NET and C#

A NUMBER OF LANGUAGES work with the .NET Framework. Microsoft is releasing the following four languages with its Visual Studio .NET product: C#, Visual Basic .NET, JScript.NET, and Managed C++. The languages likely to be the most popular are Visual Basic .NET (which I discuss in this book) and C#. C# (which you pronounce *C Sharp*) is the language that Microsoft is using for all its own internal development for .NET, so a comparison of C# and Visual Basic .NET is in order. In this appendix, I cover the similarities and differences between C# and Visual Basic .NET. If you want to go over the different concepts in Visual Basic .NET before reading about C#, look at Chapter 2, which is the roadmap for this appendix.

Although by no means an exhaustive list of the differences between C# and Visual Basic .NET, this appendix should give you a good start if you have any interest in programming in C#. Here are a couple of the better sites on the Web about C#:

- ◆ www.c-sharpcorner.com: A site that offers some samples that show you a variety of techniques.
- ◆ www.csharpindex.com: This site provides code samples, tutorials, and decent links to other C# sites.

Syntax

One of the most noticeable differences between Visual Basic .NET and C# is the syntax that each uses. C# comes from the C family of languages, which also includes C++ and Java. If you've done in any work in JavaScript with your Web development, you're sure to recognize the basic syntax in C#, which I describe in the following section.

Basic syntax

In C#, you wrap each of your procedures, namespaces, classes, and conditional statements in *braces* (`{ }`). Inside these constructs, each line ends with a semicolon (`;`). Compare this syntax to Visual Basic .NET, where each procedure and conditional ends with an `End Type_Of_Construct`, and a line ends with a carriage return.

Listing B-1 shows two complete classes in C#. `Class1` is used as the starting point of the application, while the class named `MyClass` is included to show a property definition.

Listing B-1: C# Syntax

```
using System;

namespace HelloWorld
{
    class Class1 {
        [STAThread]
        static void Main(string[] args) {
            MyClass objMyClass = new MyClass(1);
            Console.WriteLine("Hello, World! MyProperty = {0}",objMyClass.MyProperty);
            Console.Read();
        }
    }

    class MyClass {
        private int m_property;
        public MyClass(int intMyProperty) {
            this.MyProperty = intMyProperty;
        }
        public int MyProperty
        {
            get {
                return m_property;
            }
            set {
                m_property = value;
            }
        }
    }
}
```

Although I explain the syntax of the code throughout the appendix, I want to take a moment to explain what is going on here.

- ◆ **namespace:** The namespace indicates the name of the namespace. By default, there is no explicitly named namespace in Visual Basic .NET, because Visual Basic .NET automatically assigns the project name as the namespace name. This is not the case in C#. The namespace block surrounds all of your classes in the file, although you can have multiple namespaces in one file. The namespace here is `HelloWorld`:

```
namespace HelloWorld {  
}
```

- ◆ **class:** Classes are set up in the same manner as namespaces:

```
class MyClass {  
}
```

- ◆ **entry point:** The entry point here, as in Visual Basic .NET, is called `Main()`. If you go back to Listing 2-2 in Chapter 2, you see a line like this:

```
strMessage = msvb.Command()
```

This line is necessary to pull any command line arguments in Visual Basic .NET, as the language has no way of specifying command line parameters. In C#, the `Main()` routine contains an array of arguments, which allows you to pull these arguments without resorting to another object. You can tell the `Main()` routine is a subroutine, and not a function, because it returns `void`. The return type is indicated in front of the routine name:

```
static void Main(string[] args) {  
}
```



C#, unlike Visual Basic .NET, is case-sensitive. `Public` is not a keyword, although `public` is. In Visual Basic .NET, the IDE corrects case for you, but the language is not case-sensitive. If you use Visual Studio .NET, the IDE helps you out if you type in the incorrect case.

Types

As in Visual Basic .NET, you find reference types and value types in C#. Although C# doesn't use the `Module` keyword, the rest of the types in Visual Basic .NET have a direct correlation in C#.



This appendix assumes that you have already read the book and are familiar with Visual Basic .NET. If you have not read the book, you should at least read Chapter 2 before going any further.

Table B-1 compares and contrasts the syntax for classes, structures, and enumerators in Visual Basic .NET and in C#.

TABLE B-1 CLASSES, STRUCTURES, AND ENUMERATORS

Classes

C#	You wrap classes in braces, as follows: <pre>public class MyClass { } }</pre>
Visual Basic .NET	Classes end with <code>End Class</code> , as follows: <pre>Public Class MyClass End Class</pre>

Structures

C#	You wrap structures in braces, as follows: <pre>struct MyStructure { public int intFirstInteger; public int intSecondInteger; }</pre>
Visual Basic .NET	Structures end with <code>End Structure</code> , as follows: <pre>Public Structure MyStructure Public intFirstInteger As Integer Public intSecondInteger As Integer End Structure</pre> <p>As mentioned in Chapter 2, structures are value types that mimic the same "structure" of a class, which is a reference type. For Visual Basic 6 developers, a <code>Structure</code> can replace UDTs (user-defined types).</p>

Enumerators

C#	An enumerator is a set list of constants that you declare by using the word <code>enum</code> . In its simplest format, it's a list that starts implicitly with the number 0, as in the following example <pre>public enum daysOfWeek { Sun, Mon, Tues, Wed, Thurs, Fri, Sat }</pre> <p>By implicit here, I mean that the language, or rather the CLR, makes the decision of a starting point for the enumerator. This is the opposite of the next example, where I use explicit code to indicate that Sunday should have a value of 100, and so forth.</p>
-----------	--

Enumerators

```
public enum DaysOfWeek : int
{
    Sun =100,Mon=101,Tues=102,Wed=103,
    Thurs=104,Fri=105,Sat=106
}
```

Visual Basic .NET

The Visual Basic .NET version is very similar. As in C#, you can choose not to initialize your values and have an enumerator that starts with 0 and increases by 1 for each item, as in the following example:

```
Public Enum daysOfWeek
    Sun
    Mon
    Tues
    Wed
    Thurs
    Fri
    Sat
End Enum
```

The better method is to explicitly state a value for each item, even if it's the default method of setting up an enumerator (0-based list), as follows:

```
Public Enum daysOfWeek as Integer
    Sun = 100
    Mon = 101
    Tues = 102
    Wed = 103
    Thurs = 104
    Fri = 105
    Sat = 106
End Enum
```

Data types

One of the goals of the Common Type System (CTS) is to share common data types across all languages in .NET. This goal is the reason for changing the Integer and Long types in Visual Basic .NET. Listing B-2 shows the declaration of different types of variables in C#. Notice that the format for declaration is to name the datatype in front of the variable name rather than use `As DataType`.

Listing B-2: Declaring Variables in C#

```
public int MyInteger;
public short myShort = 100;
public char MyChar = 'A';
const int intMyConst = 1024;
```

You may notice that the words that I use to declare data types in this example are a bit different than those for Visual Basic .NET. You find a few other differences, too, in the keywords that you use for data types. Table B-2 shows the difference in data types in the two languages and how they map to .NET types. Most of the types here are CTS compliant; the main exception is with types that do not have a Visual Basic .NET equivalent.

TABLE B-2 PRIMITIVE TYPES IN C# AND VISUAL BASIC .NET

C#	Visual Basic .NET	.NET Type	Size	Description	Value Range
bool	Boolean	Boolean	8 bits	Boolean	True or False
sbyte		SByte	8 bits	Signed Byte	-128 to 127
byte	Byte	Byte	8 bits	Unsigned Byte	0 to 255
short	Short	Int16	16 bits	Signed short integer	-32,768 to 32,767
ushort		UInt16	16 bits	Unsigned short integer	0 to 65,536
int	Integer	Int32	32 bits	Signed integer	-2,147,483,648 to 2,147,483,647
uint		UInt32	32 bits	Unsigned integer	0 to 4,294,967,296
long	Long	Int64	64 bits	Signed long integer	-9.22337E+18 to (9.22337E +18 -1)
ulong		UInt64	64 bits	Unsigned long integer	0 to 1.84467E+19
float	Single	Single	32 bits	Single precision floating point	-3.40282E+38 to (3.40282E+38 -1)
double	Double	Double	64 bits	Double precision floating point	-1.79769E+308 to (1.79769E+308 -1)
decimal	Decimal	Decimal	96 bits	Precision decimal with 28 significant digits	-7.9228+28 to (7.9228+28 -1)

C#	Visual Basic .NET	.NET Type	Size	Description	Value Range
char	Char	Char	16 bits	Unicode character	Any single Unicode character
string	String	String		Reference type to hold strings	String of Unicode characters



The types not represented by keywords in Visual Basic .NET are not CLS (Common Language Specification) compliant. While you can use the .NET types in Visual Basic .NET, you will step outside of CLS compliance.

Be wary of breaking CLS compliance, especially with any public members in your classes, because you will potentially restrict your classes to the language you write them in.

Arrays

Arrays are indexed groups of primitive types or objects. This means that you can retrieve an item in an array by using an index number. Table B-3 shows the difference in code for arrays. In all .NET languages, you have the ability to initialize, or set the values of the array, when you declare an array.

TABLE B-3 ARRAYS

C#	Declaring an array: <pre>int[] aryMyArray = new int[25];</pre> Declare and initialize an array: <pre>int[] aryMyArray2 = { 1, 2, 3, 4, 5 }</pre>
Visual Basic .NET	Declaring an array: <pre>Dim aryMyArray(25) As Integer</pre> Declare and initialize an array: <pre>Dim aryMyArray2 As Integer = { 1, 2, 3, 4, 5 }</pre>



You cannot specify the bounds of an array in any .NET language. During part of the cycle, you could do this in Visual Basic .NET, as in the following:

```
Dim aryMyArray3(1-10) As Integer
```

This was due to the clamor from a few Visual Basic developers who did not like this change. I, respectfully, disagree with the ability to set bounds. In general, when I need to set specific lower and upper bounds, I am normally working with an enumerator, not an array.

With arrays, there is another difference between C# and Visual Basic .NET: In Visual Basic .NET, you specify the upper boundary of the array in the declaration, which means that `aryMyArray(25)` contains 26 items, from 0 to 25. In C#, the array contains 25 items, from 0 to 24.

In Visual Basic .NET, you can `ReDim` arrays to a new size. C# doesn't support this capability. If you want a dynamic array in C#, you must use the `ArrayList` in the `System.Collection` namespace.

Operators

Operators are keywords (or symbols) that are used to indicate operations you can perform on variables in .NET languages. They are divided into groups, such as arithmetic (to perform calculations), relational (to relate items), logical, and bitwise operators.

Table B-4 shows the arithmetic operators that you use in C#. The *Overload?* column indicates whether you, as a developer, can overload this operator and create your own implementation. Operator overloading is allowed in C#, but not in Visual Basic .NET.

TABLE B-4 ARITHMETIC OPERATORS

C#	Visual Basic .NET	Description	Overload? (C# only)
+	+	<i>Addition:</i> Adds two numbers.	Yes
-	-	<i>Subtraction:</i> Subtract second number from first.	Yes
*	*	<i>Multiplication:</i> Multiply two numbers.	Yes
/	/	<i>Division:</i> Divide first number by second.	Yes
	\	<i>Integer Division:</i> Divide and round down to an Integer.	Yes
%	Mod	<i>Remainder:</i> Show the remainder.	Yes

C#	Visual Basic .NET	Description	Overload? (C# only)
<<		<i>Bit Shift Left:</i> High order bits are discarded and low order bits are set to 0.	Yes
>>		<i>Bit Shift Right:</i> Low order bits are discarded and high order set to either 0 or 1. If the data type is signed, the high order is reset to 1 each time if it's a negative number.	Yes
++		<i>Increment:</i> Increase number by one. Where the operator sits in the statement varies as the operation is performed. If you tack this operator to the front of a variable, you increment first and then assign. If you attach it to the end of a variable, you assign and then increment, as in the following examples <pre>const int intTemp = 5; // assign then increment intValue = intTemp++ // intValue is 5 // increment and then assign intValue = ++intTemp // intValue is 7</pre>	Yes
--		<i>Decrement:</i> Decrease number by one. Where the operator sits in the statement varies as the operation is performed. If you tack this operator to the front of a variable, you decrement first and then assign. If you attach it to the end of a variable, you assign and then decrement, as in the following examples <pre>const int intTemp = 5; // assign then increment intValue = intTemp-- // intValue is 5 // increment and then assign intValue = --intTemp // intValue is 3</pre>	Yes
+	&	<i>String Concatenation:</i> Used to add one string to the end of another.	Yes

Table B-5 shows the relational operators, which are used to relate one variable to another. Once again, overloading applies to C# and not Visual Basic .NET.

TABLE B-5 RELATIONAL OPERATORS

C#	Visual Basic .NET	Description	Overload? (C# only)
!	Not	<i>Logical Negation</i> : Returns the opposite of the Boolean value returned from a conditional.	Yes
==	=	<i>Equal</i> : Returns True if both sides are equal.	Yes
!=	<>	<i>Not Equal</i> : Returns True if the two sides are not equal.	Yes
<	<	<i>Less Than</i> : Returns True if left side is smaller than right.	Yes
>	>	<i>Greater Than</i> : Returns True if left side is larger than the right.	Yes
<=	<=	<i>Less Than or Equal</i> : Returns True if the left side is smaller or equal to the right.	Yes
>=	>=	<i>Greater Than or Equal</i> : Returns True if the left side is larger or equal to the right.	Yes

Table B-6 shows the logical and bitwise operators available in each language, as well as which items you can overload in C#. Logical operators are primarily used in conditional statements, such as the following:

```
'Visual Basic .NET
If x = 1 AND y = 2 Then
End If
//C#
if ((x==1)&&(y==2)) {}
```

Bitwise operators have to do with the way computers deal with variables, objects, and so forth. Ultimately everything breaks down to a bit. For example, a computer AND-ing 5 and 12 performs the AND like this:

```
0101 &
1100 =
0100
```

It only passes the bit forward when the bits in both variables are set to 1. Any other combination results in a 0. Table B-6 also shows the syntax differences in Visual Basic .NET and C#, as well as the ability to overload operators.

TABLE B-6 LOGICAL AND BITWISE OPERATORS

C#	Visual Basic .NET	Description	Overload? (C# only)
&&	AND	Logical AND: Returns True when both conditions are true; otherwise returns False.	No
	OR	Logical OR: Returns True if either condition is true; returns False only when both conditions are false.	No
&	AND	Bitwise AND: Returns a 1 if both bits being compared are 1; otherwise returns a 0.	No
	OR	Bitwise OR: Returns a 1 if either bit is 1; otherwise returns a 0.	No
^	XOR	Bitwise XOR: Also known as an exclusive OR. Returns 1 when one and only one of the bits is 1. Returns 0 if both are false or both are true.	No

Working with objects

As is true in Visual Basic .NET, you do the work in C# through classes. You must also remember that everything is an object. In the following sections, I discuss the differences in objects by looking at entry points, constructors, and the use of namespaces.

ENTRY POINT

The entry point into a C# console application is the subroutine `Main()`. Now that sounds a bit familiar, doesn't it? When you use `Main()` as an entry point, you set up the `Main()` routine as `public static void Main()` with a string array for arguments. Although I touched on this in the "Basic syntax" section, I pick through the entry point in more detail in the following list:

- ◆ **public:** The `public` keyword tells you that the scope of the routine is public.
- ◆ **static:** The `static` keyword indicates that this method is a shared method. Every instance of this class uses the same method. Shared is normal for entry points into applications, as the startup should be used by all classes in this application.

- ◆ `void`: You have no return value from this routine. Using `void` as a return value is the manner in which you distinguish between a subroutine and a function in C#. You declare a function with a data type, while you declare a subroutine `void`, which indicates nothing is returned. If you were building a function, you'd simply exchange `void` for the keyword of the type you want to see return from the function.

Now take a look at Table B-7, which shows the code for a shared `public Main` method in both C# and Visual Basic .NET. As mentioned in the “Basic Syntax” section, you can pass arguments directly into a C# `Main()` method, but you can't do so in Visual Basic .NET. You pull the arguments from the `Command` object in Visual Basic .NET, much the same way you did with Visual Basic 6.

TABLE B-7 THE MAIN() METHOD

C#	<pre>public static void Main(string[] strArgs) { // use command line arguments here }</pre>
Visual Basic .NET	<pre>Public Shared Sub Main() Dim strArgs As String = Microsoft. ↵ VisualBasic.Command ' use command line arguments here End Sub</pre>

In the code samples in the table, you can see that the C# method is more efficient as you can pass in an array of variables from the command line. The Visual Basic .NET method is a bit kludgy in comparison, as you must create a variable to pull the arguments into a single string. To place this string in an array, you must separate the string into individual items instead.

CONSTRUCTOR

If you include a constructor in Visual Basic .NET, you will code it in the subroutine `New()`; in C#, you normally create your constructor with the name of the class. Each method offers its own pluses and minuses, but I prefer the C# method because of its ease of maintenance: While `New()` is easy enough, looking for a method with the same name as the class helps me easily zero in on the constructor for a particular class. This is more critical if you have numerous classes in one file. Table B-8 shows the difference in constructors for C# and Visual Basic .NET.

TABLE B-8 CONSTRUCTORS

C#	<pre> public class MyClass { private int intType; public MyClass(int intType) { this.intType = intType; } } </pre>
Visual Basic .NET	<pre> Public Class MyClass Private m_intType As Integer Public Sub New(ByVal intType as Integer) m_intType = intType End Sub End Class </pre>

NAMESPACES

As you do in Visual Basic .NET, you use imported namespaces in C# to lessen the amount of typing necessary to use objects. The word that you use to import namespaces is `using` instead of `Imports`. Aliasing in C# uses the same format as Visual Basic .NET. Table B-9 compares the syntax to create and use namespaces in both C# and Visual Basic .NET. Note that Visual Basic .NET uses the name of the project as the default namespace, if none is supplied.

TABLE B-9 USING NAMESPACES

C#	<p>Declaring a namespace:</p> <pre> namespace MyNameSpace { //code inside Namespace } </pre> <p>Using namespaces:</p> <pre> using System.Console; using VB = Microsoft.VisualBasic; </pre>
-----------	--

Continued

TABLE B-9 USING NAMESPACES *(Continued)*

Visual Basic .NET Declaring a namespace:

```
Namespace MyNameSpace
    'Code inside Namespace
End Namespace
```

Using namespaces:

```
Imports System.Console
Imports VB = Microsoft.VisualBasic
```

Object-Oriented Concepts

Object-oriented programming (OOP) has been a hot topic in IT for quite some time. The capability to express real-life objects in software is a very powerful concept. In this section, I detail various object-oriented concepts and how you express them in C#.

Inheritance

Inheritance is the capability for a new class to implement the properties and methods from a parent class, which is more properly known as a base class, enabling you to gain all the benefits of the parent without needing to rewrite the code. It is also the ability to create an abstract class, or an interface, and inherit only the signatures of the methods and properties; in this case, you have to write the implementation for each method and property. This section deals with implementation inheritance instead of interface inheritance.

The main difference in inheritance from Visual Basic .NET to C# is syntactical. In Visual Basic .NET, you use the keyword `Inherits` on the next line; in C#, you use a semicolon and the name of the class in C#. Table B-10 shows this concept.

TABLE B-10 INHERITANCE

```
C#           public class MyClass {
                public void SayHello() {
                    System.Console.WriteLine("Hello");
                }
            }
            public class MyClass2 : MyClass {
            }
```

Visual Basic .NET	<pre> Public Class MyClass Public Sub SayHello() System.Console.WriteLine End Sub End Class Public Class MyClass2 Inherits MyClass System.Console.WriteLine("Hello!") End Class </pre>
-------------------	---

In each of the examples in the preceding table, you can call `MyClass2.SayHello` and have `Hello` return to the command line.

Overriding

At certain times in inheritance, you want to inherit most of the methods of a class but also want to create a new implementation of certain methods to better fit your new class. For example, you may want to have a method return a different string value (as the example in Table B-11 shows). This capability is known as *overriding*.

Overriding works the same in C# as in Visual Basic .NET, but the words that you use are different. In C#, you use `virtual` on the method that you want to override rather than `Overridable`, and you use `override` in C# instead of `Overrides`. This statement is a bit of oversimplification, however, because you can override simply by creating the `SayHello()` method with the keyword `new`, but if so, you don't know which version of the method to call if you stick these classes into a generic array. Table B-11 shows overriding in C# using a `virtual` method, and the same concept in Visual Basic .NET.

TABLE B-11 OVERRIDING

C#	<pre> public class MyClass { public virtual void SayHello() { System.Console.WriteLine("Hello"); } } public class MyClass2 : MyClass { public override void SayHello() { System.Console.WriteLine("Hello Again!"); } } </pre>
----	--

Continued

TABLE B-11 OVERRIDING *(Continued)*

```

    }
}

Visual Basic .NET Public Class MyClass
                   Public Overridable Sub SayHello()
                       System.Console.WriteLine("Hello")
                   End Sub
End Class

                   Public Class MyClass2
                       Inherits MyClass

                       Public Overrides Sub SayHello()
                           System.Console.WriteLine("Hello Again!")
                       End Sub
                   End Class

```

Overloading

Overloading is the capability of a language to create multiple methods with the same name but different signatures (sets of input parameters). When your component calls the overloaded method, the parameters are called to ensure that the correct version of the method is called.

Overloading a method is very simple in C#: You simply type the new version of the method, using the same name, but a different signature. Unlike Visual Basic .NET, you do not have the option of adding the `Overloads` keyword to more explicitly show that the method is overloaded. Table B-12 shows overloading in both C# and Visual Basic .NET. In each example, I have overloaded the constructor; in Visual Basic .NET, I have also overloaded another method without using the keyword `Overloads`.

TABLE B-12 OVERLOADING

```

C#           public class MyClass {
                public MyClass(int intType) {
                }
                public MyClass(string strType) {
                }
            }

```

```
Visual Basic .NET Public Class MyClass
                    Public Overloads Sub New(ByVal intType As Integer)
                    End Sub
                    Public Overloads Sub New(ByVal strType As String)
                    End Sub
                    Public Sub NoKeyword(ByVal intType As Integer)
                    End Sub
                    Public Sub NoKeyword(ByVal strType As String)
                    End Sub
                End Class
```

One thing that you can do in C# that you can't do in Visual Basic .NET is to overload operators. This capability is useful if you need to change unary operators (+, -, !, ~, ++, --, true, false), binary operators (+, -, *, /, %, &, ^, |, <<, >>), or logical operators (==, !=, <, >, <=, =>). Listing B-3 shows a useful way to overload an operator.

Listing B-3: Overloading Operators in C#

```
public override bool Equals(object obj)
{
    // set up own equal here
    MyObject objM02 = (MyObject) obj;

    if(MOID != objM02.MOID)
        return(false);
    return(true);
}
public static bool operator==(MyObject objM01, MyObject objM02)
{
    return(objM01.Equals(objM02);
}
public static bool operator!=(MyObject objM01, MyObject objM02)
{
    return(!objM01.Equals(objM02);
}
```

The useful thing about overloading operators is that you can decide what constitutes equality and what doesn't. If you simply use the `Equals` method for the objects, they evaluate as equal only if both variables reference the same object. In

data-driven applications, a good use of overriding the equals operator (==) would be a test of the equality of primary keys.

Interfaces

Interfaces are contracts between your class and the interface you're implementing. If you inherit an interface, you agree to implement all the properties and methods of the interface in the class that inherits, or implements, it. Interfaces contain no implementation of methods—just the signature. Table B-13 shows how to create interfaces in C# and in Visual Basic .NET.

TABLE B-13 INTERFACES

```
C#           interface IMyInterface {
                string MyMethod();
                string MyMethod2();
            }
            public class MyClass : IMyInterface {
                public string MyMethod() {
                    // Implementation here
                }
                public string MyMethod2() {
                    // Implementation here
                }
            }

Visual Basic .NET Public Interface IMyInterface
                    Function MyMethod() As String
                    Function MyMethod2() As String
                End Interface

                Public Class MyClass
                    Implements IMyInterface

                    Public Function MyMethod() As String
                        'Implementation Here
                    End Function

                    Public Function MyMethod2() As String
                        'Implementation Here
                    End Function
                End Class
```

Code Branching and Loops

One of the most important concepts in programming is the idea of *code flow*. Through *branches*, you run specific code based on the current state of your application variables, while *loops* enable you to run the same piece of code over and over again. In this section, I detail the syntax for branches and loops in C#.

Code branching

Code branches are sections of code where a condition is evaluated, and code is run based on the outcome of this evaluation. The two basic code branches are

- ◆ **if:** The `if` keyword is used to indicate that the statement following has to be evaluated to determine if it is true. If it is, the code is run; if not, the code doesn't run. You can extend the `if` code branching statement with `else` and `else if` branches. The `else` branch is code that runs when the statement (or all of the statements, in the case of a block with `else if` branches) is false. The `else if` branch allows you to add additional Boolean comparisons. If you find you are using a great number of `else if` statements, you should use a `switch` instead.
- ◆ **switch:** The `switch` keyword is used to evaluate when you have more choices than simply true or false. It is used to compare strings or numerics, rather than Booleans.

Table B-14 shows the comparison of the `if` code branch in both C# and Visual Basic .NET (note that the keyword is lowercase in C# and uppercase in Visual Basic .NET). If C#, you can shortcut the statement slightly by dropping the braces, as shown in the second C# example.

TABLE B-14 IF BRANCHES

```
C#           if (intMyValue == 0)
              {
                intMyValue = 1;
              }
              else if (intMyValue == 1)
              {
                intMyValue = 2;
              }
              else
```

Continued

TABLE B-14 IF BRANCHES (*Continued*)

```

    {
        intMyValue = 3;
    }

```

The same can also be written in shorthand:

```

if (intMyValue == 0)
    intMyValue = 1;
else if (intMyValue == 1)
{
    intMyValue = 2;
}
else
    intMyValue = 3;

```

```

Visual Basic .NET If intMyValue = 0 then
                    intMyValue = 1
ElseIf intMyValue = 1 then
                    intMyValue = 2
Else
                    intMyValue = 3
End If

```

`switch` branches are quite a bit different from C# to Visual Basic .NET. First, you notice the different keywords that you use (`switch` in C# and `Select Case` in Visual Basic .NET). Then you notice the differences in syntax. Finally, you notice that each statement in a `switch ... case` must have a `break` or `goto` statement. The compiler doesn't permit you to let code fall through to the next condition. If you need to do so, use `goto` to move to the next case after completing the case you're on.

Table B-15 shows a `switch` branch in C# and compares it to a `Select Case`, the Visual Basic .NET equivalent of `switch`.

TABLE B-15 SWITCH BRANCHES

```

C#                switch(intMyValue)
                  {
                    case 0:
                      intMyValue = 1;
                      break;

```

```
        case 1:
            intMyValue = 2;
            break;
        default:
            intMyValue = 3;
            break;
    }
```

```
Visual Basic .NET Select Case intMyValue
    Case 0
        intMyValue = 1
    Case 1
        intMyValue = 2
    Case Else
        intMyValue = 3
End Select
```

Loops

Sometimes you need to run the same piece of code over and over again until it meets a certain condition. Repetition of code is the reason for *loops*.

The `for` loop is an iterative loop. The basic format in C# is `for (initializer; condition; iterator)`. You use the *initializer* to set the test value; the *condition* is a Boolean expression that breaks the loop; and the *iterator* changes the value until the *condition* is met. The `foreach` loop is a special form of the `for` loop for iterating through a collection of objects. Table B-16 shows the `for` loop in C# and Visual Basic .NET, while Table B-17 covers the `foreach` loop.

TABLE B-16 FOR LOOP

```
C#          for (int intCounter=0;intCounter<101;intCounter++)
            {
                System.Console.WriteLine(intCounter);
            }
```

```
Visual Basic .NET For intCounter = 1 to 100
                    System.Console.WriteLine(intCounter)
                    IntCounter += 1
                Next intCounter
```

TABLE B-17 FOREACH LOOP

C#	<pre>foreach (game objGame in objGameArray) { System.Console.WriteLine(game.ToString()); }</pre>
Visual Basic .NET	<pre>For Each objGame In objGameArray System.Console.WriteLine(objGame.ToString()) Next objGame</pre>

Another useful type of loop is commonly known in Visual Basic as a `Do` loop (this is a `do` loop in C#, which uses a different case). This type of loop continues until the condition stated in the loop is met. There are two iterations of a `do` loop:

- ◆ `while`: Runs the code until the condition is true. If the condition is true at the offset, no code runs.
- ◆ `do ... while`: Runs one time regardless of whether the condition is true.

Table B-18 shows the format for `do` and `while` loops.

TABLE B-18 DO LOOPS

C#	<p>This loop runs only while the condition is false:</p> <pre>while (intCounter < 101) { intCounter++; }</pre> <p>This loop always runs once:</p> <pre>do { intCounter++ } while (intCounter < 101)</pre>
Visual Basic .NET	<p>These loops run only while the condition is false:</p> <pre>Do While intCounter < 101 intCounter += intCounter Loop</pre> <pre>While intCounter < 101 intCounter += intCounter</pre>

```
End While
```

```
Do Until intCounter > 100
    intCounter += intCounter
Loop
```

This loops always runs once:

```
Do
    intCounter += intCounter
Loop While intCounter < 101
```

```
Do
    intCounter += intCounter
Loop Until intCounter > 100
```

Error Handling

Error handling is structured in C# and uses the same Try ... Catch ... Finally methodology (try ... catch ...finally in C#) as in Visual Basic .NET. Table B-19 shows structured error handling in the two languages.

TABLE B-19 ERROR HANDLING

```
C#          try {
              // code that may cause an exception
            }
            catch(DivideByZeroException e) {
              // Catch Divide By Zero Exceptions
            }
            catch(Exception e) {
              // Catch generic exception and throw it up
              // the stack
              throw e;
            }
            finally {
              // Runs every time
            }
          }
```

Continued

TABLE B-19 ERROR HANDLING *(Continued)*

```
Visual Basic .NET Try
    'Code that may cause an exception
Catch e As DivideByZeroException
    'Catch Divide By Zero Exceptions
Catch e As Exception
    'Catch generic exception and throw it
    'up the stack
Throw e
Finally
    'Runs every time
End Try
```

Structured error handling is part of the .NET Framework. The fact that structured error handling is part of the framework is the reason why it behaves identically in both languages. As such, one caveat exists with error handling in both languages: If you want to set up a variety of catches to handle different exceptions, you need to pay attention to the class hierarchy. Any exception that derives from another exception must appear in code before the parent class. For this reason, `Exception` is always the last caught. Look at the .NET documentation to determine which exception classes derive from another exception class.

An easy way to think of this is that the more specific exceptions need to be handled before the more generic ones. The ultimate catch all of exceptions is the `Exception` object, so it should always appear last in the list.

Features Not Present in Visual Basic .NET

This section shows a few features of C# that aren't present in Visual Basic .NET. If you want to use these features, you must code in C# and not Visual Basic .NET.

Unsafe code

Unsafe code is code that runs outside the confines of the CLR. Examples of unsafe code include any code that directly manipulates memory. You cannot run unsafe code in Visual Basic .NET, so I'm going to show the C# syntax and leave it at that. Plenty of books on the market feature C# in all its glory, and I encourage you to examine at least one of them. Listing B-4 shows an example of unsafe code.

Listing B-4: Unsafe Code in C#

```
unsafe public void MyUnsafe1() {
    // entire method is unsafe
}

public void MySafeWithUnsafe() {
    unsafe {
        // this area is unsafe
    }
}
```

XML documentation

XML documentation is a C# feature that's not present in Visual Basic .NET. By typing `///` into a C# file in Visual Studio .NET, you get a block like the one shown in Listing B-5.

It's not completely true that XML documentation is not available in Visual Basic .NET, because you have ways to produce XML from the IL that Visual Basic .NET creates. The only mechanism built into Visual Studio .NET, however, is via a sample macro that ships with the Visual Studio .NET.

If you've ever worked with Java, the XML documentation feature of C# sounds quite a bit like JavaDoc. In the IDE, typing three whacks, or forward slashes (`/`), creates a *summary tag*. The summary tag is simply an XML tag, formatted as `<summary>`. You can write your own summary of any routines, classes, and so on between the opening and closing `<summary>` tags.

The documentation feature doesn't end there. If you type the three whacks over a method, you create the arguments as XML `<param>`, or parameter, tags. Listing B-5 shows a small example of a function with arguments.

Listing B-5: XML Documentation

```
/// <summary>
///
/// </summary>
/// <param name="intMyRef"> </param>
public int MyRoutine(ref int intMyRef) {
}
```