The .NET Languages: A Quick Translation Guide

BRIAN BISCHOF

Apress™

The .NET Languages: A Quick Translation Guide

ISBN (pbk): 1-893115-48-8

Printed and bound in the United States of America 12345678910

Technical Reviewer: Dan Appleman

Editorial Directors: Dan Appleman, Gary Cornell, Jason Gilmore, Karen Watterson

Marketing Manager: Stephanie Rodriguez

Project Manager: Grace Wong

Copy Editor: Nicole LeClerc

Production Editor: Kari Brooks

Compositor: Susan Glinert

Indexer: Rebecca Plunkett

Cover Designer: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc.,175 Fifth Avenue, New York, NY, 10010

and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com.

Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 901 Grayson Street, Suite 204, Berkeley, CA 94710.

Phone 510-549-5938, fax: 510-549-5939, email info@apress.com, or visit http://www.apress.com.

CHAPTER 2

Program Fundamentals

Table 2-1. Program Fundamentals Equivalent Chart

VB 6.0	VB .NET	C#
' Comment	' Comment	//Comment
		/*Start of comments ... End of comments */
		///XML comments
Dim var As type	Dim var As type	type var;
Dim var As type var = num	Dim var As type = num	type var=num;
Dim var1 As type, var2 As type	Dim var1, var2 As type	type var1, var2;
	Dim var1 As type = num,_ var2 As type = num	type var1=num, var2=num;

Table 2-2. Procedure Declaration Equivalent Chart

VB 6.0	VB .NET	C#

Public Sub procedure()
End Sub
	Public Sub procedure()
	End Sub
		public void procedure()
		{
		}

Public Function procedure() As _
type
procedure = value
End Function
	Public Function procedure() As
	type
	return value
	End Function
		public type procedure()
		{
		return value;
		}

Public Sub procedure(ByRef _ param As type)	Public Sub procedure(ByRef param As type)	public void procedure(ref type param)
ByVal	ByVal	
ByRef	ByRef	ref
		out
Public Sub procedure(Optional _ param As type)	Public Sub procedure(Optional _ param As type)	

Table 2-2. Procedure Declaration Equivalent Chart (Continued)

VB 6.0	VB .NET	C#
`Public Sub procedure (ParamArray _ param() As type)`	`Public Sub procedure (ParamArray _ param() As type)`	`public void procedure(type params[])`
`procedure param1, param2`	`procedure(param1, param2)`	`procedure(param1, param2);`
`var = function(param1, param2)`	`var = function(param1, param2)`	`var = function(param1, param2);`

Overview

Program fundamentals consist of all the little things that are required of almost every program, including writing code across multiple lines, writing comments, declaring variables, and declaring procedures. Although each of these tasks can seem almost trivial when compared to the other aspects of programming, it's essential to have a thorough understanding of how they work.

Starting with the Code Template

When you open a project using Visual Studio .NET, it prompts you to select a Project Template to start with. There are many options to choose from. A couple of the more common choices are VB .NET Web Service and C# Windows Application. With the exception of the chapter on Windows controls, all the examples in this book create console applications in either VB .NET or C#. This is because the book's focus is on the programming language. The best way to show the details of the language is to use examples that don't have the extra code needed when writing a Windows application.

The examples at the end of each chapter always show you the entire code template that is generated by Visual Studio. Nothing has been removed. This section will make it easier to read the examples in each chapter. This section offers a quick introduction to what the different lines of code do. The programming statements are described in more detail in various places throughout the book.

The C# template is much more involved than the VB .NET template. By comparison, a VB .NET console application is simple. It consists of a module declaration and within the module block is the `Main()` procedure. It is your job to fill in the starting code within the `Main()` procedure and add any additional procedures and classes after that.

The C# template gives you a lot more to work with. First of all, there is the `using System` statement. It tells the compiler that you want to use classes from the `System` namespace without typing `System` in front of each class. VB .NET

doesn't need this line because the `System` namespace is built in to the IDE. The VB .NET equivalent for the `using` statement is `Imports`.

C# uses the `Namespace` keyword to define the application's namespace. It defaults to the name of your application, but you can type in a different name. A namespace declares the scope for objects within it and helps you organize your code. VB .NET doesn't use the `Namespace` keyword by default (though it is supported in the language). Instead, the namespace is set via the IDE menu. Select Project ➤ Properties and you can enter the VB .NET application's namespace in the dialog box.

Next in the C# template are the XML comments. This is simply an example of how XML comments look and it gives you some starter comments. VB .NET doesn't have these comments in its template because it doesn't have the capability to do XML comments.

C# defines a default class for you. VB .NET uses a module instead. VB .NET modules are similar to a class, but they are shared and accessible anywhere (i.e., global scope) within their namespace.

C# defines the `Main()` procedure inside the default class. It uses the `static` modifier because the `Main()` procedure isn't associated with an instance of the class. VB .NET puts the `Main()` procedure within the module. The `Main()` procedure is where an application starts. There can only be one per application.

Case Sensitivity

If a compiler is case sensitive, it treats as unique procedures and variables with the same name if the case of the letters do not match.

VB 6.0 and VB .NET are not case sensitive. A variable called `FirstName` is the same as the variable called `firstName`. Because of this, the IDE for both languages automatically converts the case of your procedure and variable names to match how they were declared. This ensures that you don't mistakenly think that two variables are different.

C# is case sensitive. If you are coming from a VB 6.0 background, you might have the habit of only worrying about entering the proper case when declaring a variable. You know that after you declare a variable properly you can just enter it as lowercase and the IDE will fix it for you. This doesn't work with C#.

A common way of declaring variables in C# is to use camelCasing for private members of a class and use PascalCasing for anything exposed externally. camelCasing uses uppercase for the first letter of every word except the first word. For example, `firstName` and `socialSecurityNumber` use camelCasing. PascalCasing uses uppercase for the first letter of every word. For example, `FirstName` and `SocialSecurityNumber` is PascalCasing. Although VB .NET isn't case sensitive, I find

this naming convention useful in VB .NET because I can immediately recognize which variables are arguments in the procedure declaration.

Writing Code across Multiple Lines

Writing your code so that it crosses multiple lines can make it more readable. Given the width constraints of monitors, it can be helpful to have a programming statement use two or more lines. In VB this is done using the underscore character (_). In C# this isn't an issue because the compiler uses a semicolon (;) to mark the end of a line.

VB 6.0/VB .NET

Use the underscore character to mark that a statement will continue to the next line. It is important to remember, and easy to miss, that the underscore must be preceded with a space. A common error is to type the underscore immediately after typing the last character on a line.

```
'Demonstrate concatenating two strings together with the "&". The strings are
'on different lines. The "_" is used to tell the compiler to use the
'second line.
var = "This is a very, very, very long string " & _
      "that takes two lines to write."
```

C#

Because each statement ends with a semicolon, no line continuation character is needed to mark that a statement continues to the next line. The exception to this is the // comment tag that only applies to the line it is on. This is discussed in the next section.

```
//Demonstrate concatenating two strings together with the "+". The strings are
//on different lines.
var = "This is a very, very, very long string " +
      "that takes two lines to write.";
```

Comments

Comments play an essential part in documenting your code. This ensures that your program is understandable and maintainable by other programmers (as well as yourself). Although many programs have separately written programmer's documentation, nothing can take the place of a bunch of short, in-line comments placed at critical points in your program code.

What can be even more important than comments is writing self-documenting code. There isn't anything language specific about self-documenting code. It's a self-imposed discipline that every good programmer uses. It simply consists of taking an extra second or two to use variable names and procedure names that describe what their purpose is.

A problem for a lot of programmers entering the professional programming world is that many are accustomed to using variables and procedure names such as `X` and `Test`. This is fine if you are working on a simple program for a homework assignment or the program is out of a *Learn to Program in X Days* book. After you finish writing the program, you will probably never look at it again. But when you work on projects that will be used for months and years to come, many other programmers will be looking at your code. It's important to make it easy for them to immediately understand what your program does. Personally, I know that when I look at a program 6 months after I wrote it, the self-documenting code makes it so much easier to remember what I did and why.

Comments in both versions of VB are very simple. Use an apostrophe (`'`) in front of the text that you want to comment out. This can either be at the beginning of a line or after a line of code. The compiler ignores everything after the apostrophe.

Comments in C# are much more robust. You have the option of using single-line comments, multiline comment blocks, and XML comments. Single-line comments are two forward slashes (`//`). They can be placed at the beginning of a line or at the end of a line of code. The compiler ignores everything after the two slashes.

C# also has multiline comment blocks that are used for making very detailed comments. They are represented by `/*` to start the block and `*/` to finish the block. They are useful for temporarily blocking out a large section of code for testing purposes.

XML comments in C# are a new concept to make it easier to document your code. They are designed so that you can have "intelligent" comments that use tags to describe the different parts of your program. You can view these comments using an XML reader (e.g., Internet Explorer). Because this is a totally new concept, it remains to be seen how C# programmers will take to it. This chapter does not teach you the concepts of writing XML comments, but Table 2-3 provides an easy lookup chart for the standard tags that Microsoft recommends.

VB 6.0/VB .NET

Comments use a single apostrophe and the compiler ignores everything that comes after it.

```
'This is a comment on a line by itself
Dim Month As String   'This is a comment after a line of code
```

C#

Single-line comments use two forward slashes (`//`).

```
//This is a comment on a line by itself
string month;    //This is a comment after a line of code
```

Multiline comment blocks use `/*` and `*/` to begin and end a block of comments.

```
/* This is the start of some comments.
    This is the end of the comments. */
/* Comment out a block code for testing purposes
string month;
month = "July";
*/
```

XML comments are defined using three forward slashes (`///`) and they must follow the standards for well-defined XML.

```
/// <summary>
/// This is a sample of XML code.
/// </summary>
///<code>
string month;
month = "September";
///</code>
```

You can create any tag you feel is necessary. However, there are some standard tags you should use to make your comments consistent with other programmers' work.[1]

Table 2-3. Standard Tags for Documenting Comments Using XML

XML TAG	DESCRIPTION
`<code>`	Mark a block of code
`<example>`	An example to demonstrate how to use the code
`<exception>`	Document a custom exception class
`<include>`	Refer to comments in another file that describe the types and members in your source code
`<list>`	A bulleted item list

1. From Microsoft's "C# Programmer's Reference," which you can find in the MSDN Help file. Use the index and type in **XML Documentation**.

Table 2-3. Standard Tags for Documenting Comments Using XML (Continued)

XML TAG	DESCRIPTION
<para>	A paragraph
<param>	Describe a parameter
<paramref>	Indicate that a word is a parameter
<permission>	Describe the access to a member
<see>	Specify a link to other text
<seealso>	Text that you want to appear in a See Also section
<summary>	Summarize an object
<remarks>	Specify overview information about a class or type
<returns>	A description of the return value
<value>	Describe a property

Declaring Variables

Declaring variables is a way of telling the compiler which data type a variable represents[2]. Each of the three languages does this differently. There are three major syntax differences to be aware of: declaring a single variable, declaring multiple variables on one line, and initializing a variable.

The syntax for declaring a variable in VB 6.0 and VB .NET uses the `Dim` and `As` keywords. C# uses a shorter notation by only stating the data type followed by the variable name. Thus, VB and C# are in the reverse order. This can make it a little confusing if you are VB programmer who is starting to learn C# programming. But you will probably adjust to it very quickly.

Declaring multiple variables on a single line in VB .NET has changed a lot from VB 6.0. Previously, every variable had to have its own data type assigned to it. Otherwise, it would be treated as a variant. In VB .NET this has been changed so that all variables listed between the `Dim` and the `As` keywords are of the same type. In C#, all variables listed between the data type and the semicolon are of the same type.

VB 6.0 does not allow you to initialize variables on the declaration line—this has to be done on another line of code. VB .NET now makes this possible by placing the equal sign (=) and a value after the data type. The C# syntax places the equal sign between the variable name and the semicolon.

2. Data types are described in Chapter 3.

VB 6.0

Declare a variable by listing the data type after the `As` keyword.

```
'Declare an integer variable
Dim var1 As Integer
'Declare and initialize an integer variable
Public var2 As Integer
var2 = 5
'Declare two variables: var3 is a Variant and var4 is a Single
Private var3, var4 As Single
'Declare two variables, all of which are of type Single
Dim var5 As Single, var6 As Single
```

VB .NET

Declare a variable by listing the data type after the `As` keyword. You can initialize the variable by assigning a value to it after the data type name. This won't work when you initialize more than one variable of the same type on one line.

```
'Declare an integer variable
Dim var1 As Integer
'Declare and initialize an integer variable
Public var2 As Integer = 5
'Declare two variables, both are of type Single
Private var3, var4 As Single
'Declare and initialize two variables
Private var5 As Integer = 5, var4 As Single = 6.5
```

C#

Declare a variable by putting the data type before the variable name. You can initialize the variable by assigning a value to it after the variable name. This will work when you initialize more than one variable of the same type on one line.

```
//Declare an integer variable
int var1;
//Declare and initialize an integer variable
public int var2 = 5;
//Declare and initialize two variables, both of which are of type float
private float var3=1.25, var4=5.6;
```

Modifying a Variable's Scope

The scope of a variable refers to who can access it and how long it exists in your program. This is useful for limiting the accessibility of a variable so that it can be used within a module or class, but not outside of it.

A variable exists as long as the context in which it was declared exists. For example, a variable declared within a procedure exists as long as the procedure is running. Once the procedure ends, all variables within the procedure are released. The same applies if a variable is declared within a class, or within a code block such as the `For Next` loop.

The `Static` keyword in VB 6.0 and VB .NET modifies a variable so that it lives for the entire life of the program. When the procedure ends, the static variable keeps its value in memory. When entering the procedure again, the variable will have the same value as when the procedure last ended. The scope is `Private` because it is only visible within the procedure it was declared in.

When declaring variables in a class, you can use the `Shared` modifier with variables in VB .NET. In a C# class, the `static` modifier is equivalent to both the `Static` and `Shared` modifiers in VB .NET. This is discussed in Chapter 6.

The standard access modifiers for setting the scope of a variable are `Public`, `Private`, `Friend` and `internal`, `Dim`, and `Static`.

- The `Public` modifier allows a variable to be accessed from anywhere in the program or from a program using the assembly. There are no restrictions.
- The `Private` modifier only allows a variable to be accessed from within its declaration context.
- The `Friend` (VB .NET) and `internal` (C#) modifiers allow a variable to be used from anywhere within its assembly. Programs using the assembly can't access it.
- The `Dim` (VB only) modifier has the same scope as `Private`. It cannot be used when declaring a procedure, class, or module.
- The `Static` modifier (VB only) exists for the entire life of the program. It has the same scope as `Private`.

VB 6.0

List the access modifier before the variable name.

```
Static var As type
```

VB .NET

List the access modifier before the variable name.

```
Friend var As type
```

C#

List the access modifier before the data type.

```
private type var;
```

Declaring Procedures

Declaring procedures is how you separate the different functionality within your code. The syntax for returning a value and passing parameters is different for each language.

In VB, procedures that don't return a value are *subroutines.* Procedures that do return a value are *functions.* C# doesn't differentiate between the two because every procedure must be declared as returning some data type. If you don't want the procedure to return a value, use the `void` data type. The `void` data type returns nothing.

The C# equivalent of a VB subroutine is a procedure with type `void`. The C# equivalent of a VB function is a procedure with any other data type.

Functions in VB 6.0 require that you assign the function name to the value that is to be returned. VB .NET still allows you to return values this way, and both VB .NET and C# now use the `Return` keyword to return a value. Simply list the value to return after the `Return` keyword and it will be passed back to the calling procedure.

VB 6.0

Procedures that don't return data are declared using the `Sub` keyword. Procedures that do return data use the `Function` keyword and after the procedure parameters you must list the data type that will be returned. Assign the return value to the function name.

```
Public Sub procedure()
End Sub

Public Function procedure() As type
    Dim var as type
    var = num
    procedure = var
End Function
```

VB .NET

Procedures that don't return data are declared using the `Sub` keyword. Procedures that do return data use the `Function` keyword and after the procedure parameters you must list the data type that will be returned. Either assign the return value to the function name or list the return value after the `Return` keyword.

```
Public Sub procedure()
End Sub

Public Function procedure() As type
    Dim var as type
    var = num
    procedure = var
End Function

'Demonstrate returning a value with the Return keyword
Public Function procedure() As type
    Dim var as type
    var = num
    Return var
End Function
```

C#

Every procedure has to have a return data type listed before the procedure name. If you don't want to return a value, use the `void` data type. List the return value after the `return` keyword.

```
public void procedure()
{
}

public type procedure()
{
    type var;
    var = num;
    return var;
}
```

Passing Arguments to Procedures

Passing data to procedures is done with parameter declarations. The data that is passed to the procedures are called *arguments*. The biggest change for VB

programmers is that in VB 6.0, parameters are passed by reference as the default. This could cause problems because if you mistakenly change the value of a parameter without realizing it, you won't get an error. Instead, the data in the calling procedure would be changed and this would probably result in a hard-to-find bug. In .NET, both languages pass parameter data by value as the default. In fact, C# doesn't have a keyword for designating a parameter being passed by value. It only requires that you specify which parameters are passed by reference.

VB uses the `ByVal` keyword to specify a parameter as being passed by value. It uses the `ByRef` keyword to specify a parameter being passed by reference.

C# uses the `ref` keyword to specify a parameter as being passed by reference. It uses the `out` keyword to specify a parameter as only being used to pass data to the calling procedure. `out` differs from the `ref` keyword in that the calling procedure doesn't have to initialize the argument before passing it to the procedure. Normally, C# requires any argument passed to a procedure to be initialized first. The `out` keyword is useful when you want a procedure to return many values and you don't want to bother with initializing them.

A requirement in C# is that both the `ref` and `out` keywords must be used in the procedure declaration and in the procedure call. VB .NET doesn't have this requirement.

VB 6.0/VB .NET

Use the `ByVal` keyword to pass parameters by value. Use the `ByRef` keyword to pass parameters by reference. If not specified, the VB 6.0 default is `ByRef`. The VB .NET default is `ByVal`.

```
' VB .NET: param1 and param3 are both passed by value; param2 is passed by reference
' VB 6.0: param1 and param2 are both passed by reference; param3 is by value
Sub procedure(param1 As type, ByRef param2 As type, ByVal param3 As type)
```

C#

Passing parameters by value requires no extra coding. Use the `ref` keyword to pass parameters by reference. Use the `out` keyword to specify parameters whose only purpose is to return a value. The `ref` and `out` keywords must also be specified when calling the procedure.

```
//param1 is passed by value and param2 is passed by reference
void procedure(type param1, ref type param2, out type param3)

//Calling this procedure would look like this
procedure(val1, ref val2, out val3);
```

Optional Parameters

Optional parameters give you flexibility with how much data is required to be passed to a procedure. VB 6.0 and VB .NET use the `Optional` keyword to specify optional parameters. C# doesn't have optional parameters. As a substitute, you can use the parameter array that is described in the next section. You can also override methods as is discussed in Chapter 6.

Because a parameter is optional, you need a way to find out if the calling procedure passed an argument or not. In VB 6.0 this is done using the `IsMissing()` function. If `IsMissing()` returns `True`, no argument was passed and you can assign a default value to the parameter. The `IsMissing()` function isn't in VB .NET because it requires optional parameters to be given a default value in the procedure declaration.

A procedure can have as many optional parameters as needed, but they must be the last parameters listed. No standard parameters can be listed after them.

VB 6.0

Optional parameters use the `Optional` keyword.

```
Sub procedure(Optional param As type)
    If IsMissing(param) Then
        param = default
    End If
End Sub
```

VB .NET

Optional parameters use the `Optional` keyword. It is required that each optional parameter has a default value assigned to it.

```
Sub procedure(Optional param As type = default)

End Sub
```

C#

There are no optional parameters in C#.

Parameter Arrays

Parameter arrays give you the benefit of being able to pass from zero to an indefinite number of parameters to a procedure without having to declare each one individually.

Parameter arrays are declared in VB using the `ParamArray` keyword and in C# using the `params` keyword. There are two restrictions with parameter arrays: There can only be one parameter array declared per procedure, and it must be the last parameter listed.

A drawback to using a parameter array is that every element in the array is declared of the same type. To get around this obstacle, you can declare the array to be of type `Object`.

VB 6.0

Use the `ParamArray` keyword to declare a parameter array.

```
Sub procedure(ParamArray param() As type)
```

VB .NET

Use the `ParamArray` keyword to declare a parameter array.

```
Sub procedure(ParamArray param() As type)
```

C#

Use the `params` keyword to declare a parameter array.

```
void procedure(params type[] param)
```

Modifying a Procedure's Accessibility

You can determine which parts of a program can call a procedure by using an access modifier. The access modifier is placed at the beginning of the line, in front of the procedure declaration. The standard access modifiers for a procedure are `Public`, `Private`, and `Friend` and `internal`.[3]

- The `Public` modifier allows a procedure to be accessed from anywhere in the program or from a program using this assembly. There are no restrictions. This is the default for VB .NET procedures.

- The `Private` modifier allows a procedure to be accessed only from within its own declaration context. This includes being used within nested procedures. This is the default for C# procedures.

- The `Friend` (VB .NET) and `internal` (C#) modifiers allow a procedure to be accessed only from within the assembly.

3. There are other access modifiers available for classes. These are discussed in Chapter 6 and Chapter 8.

An interesting difference between VB .NET and C# is that they have different default modifiers. In VB .NET, procedures and functions declared in a module or class without an access modifier are treated as `Public`. In C#, methods declared in a class without an access modifier are `private` by default.

VB 6.0

List the access modifier before the procedure name.

```
Public Sub procedure()

Private Function procedure() As type
```

VB .NET

List the access modifier before the procedure name. A procedure without an access modifier will be declared as `Public`.

```
Sub procedure()    'This is Public

Public Sub procedure()

Private Function procedure() As type
```

C#

List the access modifier before the procedure data type. A procedure without an access modifier will be declared as `Private`.

```
void procedure()    //This is Private

public void procedure()

private type procedure()
```

Calling Procedures

Once you have declared a procedure, you need to call it so that its functionality can be used. Although this seems fairly simple, you need to be aware of the proper use of parentheses as well as how to properly pass arguments.

Before examining how to pass arguments, let's look at a change from VB 6.0 to VB .NET. It is now required in .NET to use parentheses when calling a procedure. This simplifies things because you will no longer get compiler errors stating that you need to have parentheses or that you are using them where

they aren't required. Now parentheses are always required, so you don't have to think about it. C# also requires parentheses when calling a procedure.

When passing arguments to a procedure, VB has the advantage over C#. Normally, when you call a procedure you pass the arguments in the order that they are listed in the procedure declaration. This applies to both VB and C#. The advantage that VB has over C# is that in VB you can pass arguments by name. In other words, you do not have to pass arguments in the order that the procedure declaration says you should. This is very helpful when working with optional parameters because you can state which parameters you want to use and leave the others alone. You don't have to use an empty comma list and worry about inserting the correct number of commas. Simply list which parameters get which arguments. This makes your code look much cleaner. Because C# doesn't have optional parameters, it doesn't need the capability to pass arguments by name.

It should be noted that regardless of the language, when calling procedures that have a parameter array, you do not need to pass the procedure an actual array. You only need to pass it arguments separated by commas. The calling procedure will store these arguments in an array, but it is only expecting standard single-value arguments.

VB 6.0

Call subroutines without using parentheses. Call functions using parentheses. When passing arguments by name use := after the argument and give it a value.

```
procedure param1, param2
var = function(param1, param2)
procedure param2 := value
```

VB .NET

Call subroutines and functions using parentheses. When passing arguments by name use := after the argument and give it a value.

```
procedure (param1, param2)
var = function(param1, param2)
procedure (param2 := value)
```

C#

Call all procedures using parentheses.

```
procedure(param1, param2);
var = procedure(param1, param2);
```

Example 2-1. Calculate Test Score Functions

This example takes up to three test scores and displays the high score and the average score. Using reference parameters and an `out` parameter is demonstrated in the `CalcHighScore()` procedure. It returns the high score. The parameter array is demonstrated in the `CalcAvgScore()` function. It returns the average score via the function name.

VB 6.0

```
'Calculate Test Scores sample application using VB 6.0
'Copyright (c)2001 by Bischof Systems, Inc.

Sub Main()
    ProcessScores 80, 90, 75
End Sub

Public Sub ProcessScores(ByVal Score1 As Integer, ByVal Score2 As Integer, _
    ByVal Score3 As Integer)
    Dim HighScore As Integer
    Dim avgScore As Single
    'Write the scores we are working with
    DisplayScores Score1, Score2, Score3
    'Get the high score
    CalcHighScore Score1, Score2, Score3, HighScore
    Debug.Print "The high score is " & HighScore
    'Get the average score
    avgScore = CalcAvgScore(Score1, Score2, Score3)
    Debug.Print "The average score is " & Format(avgScore, "##.00")
End Sub

'Display all scores using a parameter array
Private Sub DisplayScores(ParamArray Scores())
    Dim CurrentScore As Integer
    Debug.Print "The scores being used are: "
    For CurrentScore = 0 To UBound(Scores)
        Debug.Print Scores(CurrentScore) & "  "
    Next
    Debug.Print
End Sub
```

```
'Use an out parameter to return the high score
Private Sub CalcHighScore(ByVal Score1 As Integer, ByVal Score2 As Integer, _
    ByVal Score3 As Integer, ByRef HighScore As Integer)
    'Assume score1 is the high score
    HighScore = Score1
    'Is Score2 higher?
    If (Score2 > Score1) Then
        HighScore = Score2
    End If
    'Does Score3 beat them all?
    If (Score3 > HighScore) Then
        HighScore = Score3
    End If
End Sub

'Use a function to calculate the average score
Private Function CalcAvgScore(ParamArray Scores()) As Single
    Dim TotalScore, CurrentScore As Integer
    For CurrentScore = 0 To UBound(Scores)
        TotalScore = TotalScore + Scores(CurrentScore)
    Next
    CalcAvgScore = TotalScore / (UBound(Scores) + 1)
End Function
```

VB .NET

```
'Calculate Test Scores sample application using VB .NET
'Copyright (c)2001 by Bischof Systems, Inc.

Module Module1

    Sub Main()
        ProcessScores(80, 90, 75)
        Console.ReadLine()
    End Sub

    Public Sub ProcessScores(ByVal Score1 As Integer, ByVal Score2 As Integer, _
        ByVal Score3 As Integer)
        Dim highScore As Integer
        Dim avgScore As Single
        'Write the scores we are working with
        DisplayScores(Score1, Score2, Score3)
        'Get the high score
        CalcHighScore(Score1, Score2, Score3, highScore)
```

```
        Console.WriteLine("The high score is {0}", highScore)
        'Get the average score
        avgScore = CalcAvgScore(Score1, Score2, Score3)
        Console.WriteLine("The average score is {0:N2}", avgScore)
    End Sub

    'Display all scores using a parameter array
    Private Sub DisplayScores(ByVal ParamArray Scores() As Integer)
        Dim currentScore As Integer
        Console.Write("The scores being used are: ")
        For currentScore = 0 To Scores.Length - 1
            Console.Write(Scores(currentScore).ToString & "  ")
        Next
        Console.WriteLine()
    End Sub

    'Use an out parameter to return the high score
    Private Sub CalcHighScore(ByVal Score1 As Integer, ByVal Score2 As Integer, _
        ByVal Score3 As Integer, ByRef HighScore As Integer)
        'Assume score1 is the high score
        HighScore = Score1
        'Is Score2 higher?
        If (Score2 > Score1) Then
            HighScore = Score2
        End If
        'Does Score3 beat them all?
        If (Score3 > HighScore) Then
            HighScore = Score3
        End If
    End Sub

    'Use a function to calculate the average score
    Private Function CalcAvgScore(ByVal ParamArray Scores() As Integer) As Single
        Dim TotalScore, CurrentScore As Integer
        For CurrentScore = 0 To Scores.Length - 1
            TotalScore += Scores(CurrentScore)
        Next
        Return CSng(TotalScore / Scores.Length)
    End Function
End Module
```

C#

```
//Calculate Test Scores sample application using C#
//Copyright (c)2001 by Bischof Systems, Inc.

using System;

namespace C_Fundamentals_TestScores
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            ProcessScores(80, 90, 75);
            Console.ReadLine();
        }
        static public void ProcessScores(int Score1, int Score2, int Score3)
        {
            int highScore;
            float avgScore;
            //Write the scores we are working with
            DisplayScores(Score1, Score2, Score3);
            //Get the high score
            CalcHighScore(Score1, Score2, Score3, out highScore);
            Console.WriteLine("The high score is {0}", highScore);
            //Get the average score
            avgScore = CalcAvgScore(Score1, Score2, Score3);
            Console.WriteLine("The average score is {0:N2}", avgScore);
        }

        //Display all scores using a parameter array
        static internal void DisplayScores(params int[] Scores)
        {
            Console.Write("The scores being used are: ");
            for (int currentScore = 0; currentScore < Scores.Length;
                currentScore++)
            {
                Console.Write(Scores[currentScore] + "  ");
            }
            Console.WriteLine();
        }
```

```
        //Use an out parameter to return the high score
        static private void CalcHighScore(int Score1, int Score2, int Score3,
            out int highScore)
        {
            //Assume Score1 is the high score
            highScore=Score1;
            //Is Score2 higher?
            if (Score2 > Score1)
            {
                highScore = Score2;
            }
            //Does Score3 beat them all?
            if (Score3 > highScore)
            {
                highScore = Score3;
            }
        }

        //Use a function to calculate the average score
        static private float CalcAvgScore(params int[] Scores)
        {
            int totalScore=0;
            for (int currentScore = 0; currentScore < Scores.Length;
                currentScore++)
            {
                totalScore += Scores[currentScore];
            }
            return totalScore/(float)Scores.Length;
        }
    }
}
```

Example 2-1 Output

```
The scores being used are: 80  90  75
The high score is 90
The average score is 81.67
```