**Chapter-2**

**The Visual Basic Language**

**2.1 Keywords**

The given below keywords are reserved for use by Visual Basic, and you use them to build your program.

The Visual Basic Keywords

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ByVal | CByte | Char | Const | Date |
| Dim | Else | Error | For | GoSub |
| Imports | Is | Loop | MustOverride | Next |
| Objects | OrElse | Preserve | RaiseEvents | Resume |
| And | Auto | Call | CChar | CInt |
| CShort | Decimal | ElseIf | Event | Friend |
| GoTo | MyBase | Not | Overloads | Private |
| ReadOnly | Return | Boolean | Case | CDate |
| Class | Do | End | Exit | Function |
| Handles | Inherits | Mod | Overridable | Select |
| ByRef | Catch | CStr | Default | Double |
| Enum | False | If | Integer | Namespace |
| NotInheritable | Optional | Overrides | Protected | As |
| Byte | CBool | Delegate | Each | Finally |
| Implements | Interface | Long | New | NotOverridable |
| Public | Static | Try | Short | Single |

**2.2 Statements**

A Visual Basic statement is a complete instruction. It contain

**Keywords**- Words reserved for Visual Basic’s use.

**Operators**-Symbols used to perform operations like +,-.

**Variables**- Symbolic names given to values stored in memory and declared with the **Dim** keyword.

**Literal values**-Simple values like 5 or “hello”.

**Constants**-The same as variable, except that constants are assigned a value that cannot then be altered.

**Expressions**-Combinations of terms and/or keywords that yield a value

EG:

Module Module1

Sub Main()

Dim strMsg As String

strMsg = "I am new to .Net application"

System.Console.WriteLine(strMsg)

System.Console.ReadKey()

End Sub

End Module

**2.3 The Option and Imports Statements**

The Option statement sets a number of options for the code and the Imports Statement imports namespaces into your code, making them more readily available.

**Option Statements**

You use Option statements to set the “ground rules” for your code.

* **Option Explicit**- Set to **On** or **Off. On** is the default. Requires declaration of all variables before they are used.

Option Explicit On

Module Module1

Sub Main()

Dim thisVar As Integer

thisVar = 10

' The following assignment produces a COMPILER ERROR because

' the variable is not declared and Option Explicit is On.

thisInt = 10

End Sub

End Module

* **Option Compare**- Set to **Binary** or **Text**. This specifies if strings are compared using binary or text comparison operations.
* **Option Strict**-Set to **On** or **Off**. Off is the default. When You assign a value of one type to a variable of another type if it is a **On** you will get an error.

Option Strict Off

Module Module1

Sub Main()

Dim longNum As Double

Dim intNum As Integer

longNum = 123.9

intNum = longNum

System.Console.WriteLine(intNum)

System.Console.ReadKey()

End Sub

End Module

o/p:124

[Note: Option Strict On : it gives the error: That is Option Strict On disallows implicit conversions from 'Double' to 'Integer'. ]

**Import Statements**

You use import statements to import a namespace. For ex,the **WriteLine** procedure is built into the System.Console namespace, so it is a method of that namespace.

Imports System.Console

Module Module1

Sub Main()

WriteLine("Hello")

ReadKey()

End Sub

End Module

**2.5 Declaring Variable and Data Types**

In VB.NET variables are declared using Dim keyword. Multiple declarations without repeating the data type are allowed here.

A variable can be initialized at the time of declaration itself. A variable can be defined using primitive data types such as integer, double, using composite data type such as an array or structure and an object type such as label, textbox.

Eg:

Dim intA as Integer

Dim strName as String

**Data Types**

An application needs to store the data entered by the user and display the result. The data used by an application may be of different types. For example, the name of a customer is entered as a string and age of a customer is entered as an integer. The following table lists data types provided by Visual Basic .NET to store different types of data.

|  |  |
| --- | --- |
| **Type** | **Storage size** |
| Boolean | 2 bytes |
| Byte | 1 byte |
| Char | 2 bytes |
| Date | 8 bytes |
| Decimal | 16 bytes |
| Double | 8 bytes |
| Integer | 4 bytes |
| Long | 8 bytes |
| Object | 4 bytes |
| Short | 2 bytes |
| Single | 4 bytes |
| String | 2 bytes |

**2.6 Arrays**

An array can hold a collection of values of the same type. The variables in an array are called elements. Array has a name and these elements can be accessed by using index or subscript.

An array can have single, double and multi dimension. Lower bound of the array for all dimension starts from 0.Upper bound can be defined by using the code. Array is a reference type and can be static or dynamic array. The size of the dynamic array changes during the execution of the code.

**Static Arrays**

Arrays can be declared using Dim keyword. You can initialize the values in the array. If array is declared as Object type, we can store different type of data types also.

Dim Data(30)

Dim Strings(10) As String

Dim TwoDArray(20, 40) As Integer

Eg:Module Module1

Sub main()

Dim a(2) As Integer

Dim i, sum As Integer

a(0) = 100

a(1) = 20

a(2) = 99

For i = 0 To 2

sum = sum + a(i)

Next

System.Console.WriteLine("Sum is:" & sum)

System.Console.ReadKey()

End Sub

End Module

o/p:Sum is:219

**Dynamic Arrays**

If you don't know the size of the array, dynamic arrays would be very helpful for storing the data. The ReDim and Preserve statements are used for this purpose. **For Dynamic array we should not provide the size when we declare the array. After the declaration, we cannot change the data type and can specify size using ReDim statement. Preserve keyword is used to preserve the data without losing when changing the dimension of the array.** Using ReDim Preserve keyword, only last dimension can be resized.

You can resize an array by using the ReDim statement. For example, to resize the Marks array to hold 20 elements, you can use the following code:

**ReDim marks(20)**

However, when you use the ReDim statement, the existing contents are erased. You can use the Preserve keyword to ensure that the existing contents of an array are not lost when you use the ReDim statement.

**ReDim Preserve marks(20)**

If you use the Preserve keyword while resizing an array, only the last dimension of the array can be resizes. Consider the following example

Dim marks (10,5) As Integer

ReDim Preserve marks(10,10) “Ok, The last dimension is resized

ReDim marks(15,5) “Error, The first dimension is resized

Eg:

Module Module1

Sub main()

Dim a(2) As Integer

Dim i As Integer

a(0) = 100

a(1) = 20

a(2) = 99

ReDim a(5)

a(3) = 11

a(4) = 90

For i = 0 To 4

System.Console.WriteLine("Array value:" & a(i))

Next

System.Console.ReadKey()

End Sub

End Module

o/p: Array value:0

Array value:0

Array value:0

Array value:11

Array value:90

Note:It does not preserve the previous values

**Using Preserve keyword**

**Eg:**

Module Module1

Sub main()

Dim a(2) As Integer

Dim i As Integer

a(0) = 100

a(1) = 20

a(2) = 99

ReDim Preserve a(5)

a(3) = 11

a(4) = 90

For i = 0 To 4

System.Console.WriteLine("Array value:" & a(i))

Next

System.Console.ReadKey()

End Sub

End Module

o/p: Array value:100

Array value:20

Array value:99

Array value:11

Array value:90

**2.8 Operators**

VB.NET comes with plenty of built-in operators, which manipulate your data. For ex, adding the values in var1 and var2 with the addition operator +, and storing the result in the var3 with the assignment operator=.

Eg:

Module Module1

Sub main()

Dim var1 As Integer = 1234

Dim var2 As Integer = 2345

Dim var3 As Integer

var3 = var1 + var2

System.Console.WriteLine(var3)

System.Console.ReadKey()

End Sub

End Module

O/P:3579

**Arithmetic operators**

* ^ Exponentiation
* \* Multiplication
* / Division
* \ Integer division
* Mod modulus
* + Addition
* -Subtraction

**Assignment operators**

For ex. Temp=76 stores the value 76 in the variable Temp.

* = Assignment
* ^= Exponentiation followed by assignment
* \*= Multiplication followed by assignment
* /= Division followed by assignment
* \= integer division followed by assignment
* += Addition followed by assignment
* -= Subtraction followed by assignment
* &= Concatenation followed by assignment

**Comparison Operators**

These operators yield true or false values. For ex, 5>3 yields a value of true.

* < Less than
* <= Less than or equal to
* > Greater than
* >= Greater than or Equal to
* = Equal to
* <> Not equal to
* Is –True if two object references refer to the same object.
* Like- Performs string pattern matching

**String Concatenation operators**

* & String concatenation
* + String concatenation

**Logical/Bitwise operators**

**And**-Performs an **And** operation true if both operands are true, false otherwise.

**Not**- Reverse the logical value of its operand, from true to false and false to true.

**Or**- Operators performs an Or operation True if either operand is true, false otherwise.

**Xor**- True either operand but not both is true and false otherwise.

**AddressOf**- Gets the address of a procedure.

**GetType**- Gets information about a type.

**Note**

+= is a combination of + and =, which means that you can write val=val+1 as val+=1.

**2.9 Decision Making**

**2.9.1 if..else**

These are also called as conditional statements. According to value of the variable, control flow can be changed for the execution. There are two statements are there in this category. They **are If - Else and Select case.**

**If *condition* Then**

**[*statements*]**

**[ElseIf *condition-n* Then**

**[*elseifstatements*]]**

**[Else**

**[*elsestatements*]]**

**End If**

Here, [] denotes these are optional.

Eg:

Module Module1

Sub Main()

Dim intInput As Integer

System.Console.WriteLine("Enter an integer…")

intInput = Val(System.Console.ReadLine())

If intInput = 1 Then

System.Console.WriteLine("Thank you.")

ElseIf intInput = 2 Then

System.Console.WriteLine("That's fine.")

ElseIf intInput = 3 Then

System.Console.WriteLine("Too big.")

Else

System.Console.WriteLine("Not a number I know.")

End If

End Sub

End Module

**2.9.2 Select Case** This can be the replacement for multiple IF-Else statements. If we want to compare with multiple values of particular variable, this statement is the most appropriate one. There is no break statement in this structure. Range values are allowed in the case expression. Is operator is used to test against condition like greater than or less than values.

**Select Case *testexpression***

**[Case *expressionlist-n***

**[*statements-n*]]…**

**[Case Else**

**[*elsestatements*]]**

**End Select**

Eg:

Module Module1

Sub Main()

Dim intInput As Integer

System.Console.WriteLine("Enter an integer…")

intInput = Val(System.Console.ReadLine())

Select Case intInput

Case 1

System.Console.WriteLine("Thank you.")

Case 2

System.Console.WriteLine("That's fine.")

Case 3

System.Console.WriteLine("OK.")

Case 4 To 7

System.Console.WriteLine("In the range 4 to 7.")

Case Is > 7

System.Console.WriteLine("Definitely too big.")

Case Else

System.Console.WriteLine("Not a number I know.")

End Select

End Sub

End Module

**2.10 Loop**

You use loops to execute a series of statements repeatedly.

**2.10.1 Do** Loop

The Do loop keeps executing its enclosed statements while or until condition is true. You can also terminate a Do loop at any time with an Exit Do statement. The Do loop has two versions: you can either evaluate a condition at the beginning or at the end.

**Syntax:**At the beginning:

**Do [{While | Until} *condition* ]**

**[*statements*]**

**[Exit Do]**

**[*statements*]**

**Loop**

or at the end:

**Do**

**[*statements*]**

**[Exit Do]**

**[*statements*]**

**Loop [{While | Until} *condition*]**

Eg:

Module Module1

Sub Main()

Dim strInput As String

Do Until UCase(strInput) = "STOP"

System.Console.WriteLine("What should I do?")

strInput = System.Console.ReadLine()

Loop

End Sub

End Module

**2.10.2 For** Loop

This is the most popular loop in VB.NET. For loop needs the loop index and it counts the iteration of the loop. For loop can be nested also. The default is step 1 if the user has not written the value in the code.

Syntax:

**For *index* = *start* To *end* [Step *step*]**

**[*statements*]**

**[Exit For]**

**[*statements*]**

**Next [*index*]**

The ***index*** variable is originally set to *start* automatically when the loop begins. Each time through the loop, ***index*** is incremented by ***step*** (***step*** is set to a default of **1** if you don't specify a value) and when ***index*** equals end, the loop ends.

Eg:

Module Module1

Sub Main()

Dim intLoopIndex As Integer

For intLoopIndex = 0 To 3

System.Console.WriteLine("Hello from Visual Basic")

Next intLoopIndex

End Sub

End Module

Hello from Visual Basic

Hello from Visual Basic

Hello from Visual Basic

Hello from Visual Basic

If I were to use a **step** size of 2:

For intLoopIndex = 0 To 3 Step 2

System.Console.WriteLine("Hello from Visual Basic")

Next intLoopIndex

I'd see this result:

Hello from Visual Basic

Hello from Visual Basic

**2.10.3 For Each..Next** Loop

You use the For Each..Next loop to loop over elements in an array.

Syntax:

**For Each *element* In *group***

**[*statements*]**

**[Exit For]**

**[*statements*]**

**Next [*element*]**

You can get a look at this loop in action with an example like this, in which I'm displaying all the elements of an array:

Module Module1

Sub Main()

Dim intIDArray(3), intArrayItem As Integer

intIDArray(0) = 0

intIDArray(1) = 1

intIDArray(2) = 2

intIDArray(3) = 3

For Each intArrayItem In intIDArray

System.Console.WriteLine(intArrayItem)

Next intArrayItem

End Sub

End Module

And here's the result of this code:

0

1

2

3

**2.10.4 While** Loop

**While** loops keep looping while the condition they test remains true, so you use a **While** loop if you have a condition that will become false when you want to stop looping.

Syntax:

**While *condition***

**[*statements*]**

**End While**

And here's an example putting **While** to work:

Sub CheckWhile()

Dim intCounter As Integer = 0

Dim intNumber As Integer = 10

While intNumber > 6

intNumber -= 1

intCounter += 1

End While

MsgBox("The loop ran " & intCounter & " times.")

End Sub

And here's what you see when you run this code:

The loop ran 4 times.

**2.10.5 With Statement**

You use the With statement to execute statements using a particular object.

Syntax:

**With *object***

**[*statements*]**

**End With**

EG:

With TextBox1

.Height = 1000

.Width = 3000

.Text = "Welcome to Visual Basic"

End With

**Chapter 3:**

**Procedures, scope and exception handling**

**Sub Procedures and Functions**

Procedures are made up of series of Visual Basic statements that, when called, are executed. After the call is finished, control returns to the statement that called the procedure. You can pass data to procedures and the code in the procedure can work on that data.

There are two types of procedures in Visual Basic .NET: Sub procedures and functions.

Sub procedures do not return a value, while functions do.

Eg1:

Module Module1

Sub Main()

System.Console.WriteLine("Hello from Visual Basic")

End Sub

End Module

Eg2:

Module Module1

Sub Main()

DisplayMessage()

End Sub

Sub DisplayMessage()

System.Console.WriteLine("Hello from Visual Basic")

End Sub

End Module

The keyword **ByVal** indicates that the text string is passed ***by value***, which means a copy of the string is passed.. The other possibility is **ByRef**, which means that the argument will be **passed *by reference***. When you pass a variable by reference the *location* of the variable is passed to the procedure, which means you have direct access to that variable back in the calling code.

Eg3:

Module Module1

Sub Main()

DisplayMessage("Hello from Visual Basic")

End Sub

Sub DisplayMessage(ByVal strText As String)

System.Console.WriteLine(strText)

End Sub

End Module

You can also create functions, which return values.

Module Module1

Sub Main()

Dim intValue As Integer = 2

System.Console.WriteLine("{0}+{1}={2}", intValue, intValue, Addem(intValue, intValue))

End Sub

Function Addem(ByVal int1 As Integer, ByVal int2 As Integer) As Long

Return int1 + int2

End Function

End Module

You return a value from a function with the **Return** statement

**Understanding Scope:**

The *scope* of an element in your code is all the code that can refer to it without qualifying its name.

In VB .NET, *where you declare an element determines its scope*, and an element can have scope at one of the following levels:

* ***Block scope***—available only within the code block in which it is declared
* ***Procedure scope***—available only within the procedure in which it is declared
* ***Module scope***—available to all code within the module, class, or structure in which it is declared
* ***Namespace scope***—available to all code in the namespace

Declaring a variable in a procedure gives it procedure scope, and so on.

* **Public** —The **Public** statement declares elements to be accessible from anywhere within the same project, from other projects that reference the project, and from an assembly built from the project. You can use **Public** only at module, namespace, or file level. This means you can declare a **Public** element in a source file or inside a module, class, or structure, but not within a procedure.
* **Protected** —The **Protected** statement declares elements to be accessible only from within the same class, or from a class derived from this class. You can use **Protected** only at class level, and only when declaring a member of a class.
* **Friend** —The **Friend** statement declares elements to be accessible from within the same project, but not from outside the project. You can use **Friend** only at module, namespace, or file level. This means you can declare a **Friend** element in a source file or inside a module, class, or structure, but not within a procedure.
* **Protected Friend** —The **Protected** statement with the **Friend** keyword declares elements to be accessible either from derived classes or from within the same project, or both. You can use **Protected Friend** only at class level, and only when declaring a member of a class.
* **Private** —The **Private** statement declares elements to be accessible only from within the same module, class, or structure. You can use **Private** only at module, namespace, or file level. This means you can declare a **Private** element in a source file or inside a module, class, or structure, but not within a procedure.

Eg 1:

Module Module1

Sub Main()

Dim intValue As Integer = 1

If intValue = 1 Then

Dim strText As String = "No worries."

System.Console.WriteLine(strText)

End If

System.Console.WriteLine(strText) 'Will not work!

End Sub

End Module

**Eg 2:** Module Module1

Sub Main()

System.Console.WriteLine(Module2.Function1())

End Sub

End Module

Module Module2

Public Function Function1() As String 'OK

Return "Hello from Visual Basic"

End Function

End Module

**Immediate Solutions: Creating Sub Procedures**

We know all about Sub procedures: They're the handy blocks of code that can organize your code into single-purpose sections to make programming easier. Unlike functions, Sub procedures do not return values, but like functions, you can pass values to Sub procedures in an argument list.

You declare Sub procedures with the **Sub** statement:

[ <*attrlist*> ] [{ Overloads | Overrides | Overridable | NotOverridable |

MustOverride | Shadows | Shared }]

[{ Public | Protected | Friend | Protected Friend | Private }]

Sub *name* [(*arglist*)]

[ *statements* ]

[ Exit Sub ]

[ *statements* ]

End Sub

Here are the parts of this statement:

* ***attrlist***-List of attributes for this procedure. You separate multiple attributes with commas.
* **Overloads**-Specifies that this Sub procedure overloads one (or more) procedures defined with the same name in a base class. In this case, the argument list must be different from the argument list of every procedure that is to be overloaded (that is, the lists must differ in the number of arguments, their data types, or both). You cannot specify both **Overloads** and **Shadows** in the same procedure declaration.
* **Overrides**-Specifies that this Sub procedure overrides a procedure with the same name in a base class. Note that the number and data types of the arguments must match those of the procedure in the base class.
* **Overridable**-Specifies that this Sub procedure can be overridden by a procedure with the same name in a derived class.
* **NotOverridable**-Specifies that this Sub procedure may not be overridden in a derived class.
* **MustOverride**-Specifies that this Sub procedure is not implemented. Instead, this procedure must be implemented in a derived class. If it is not, that class will not be creatable.
* **Shadows**-Makes this Sub procedure a shadow of an identically named programming element in a base class. A shadowed element is unavailable in the derived class that shadows it. You can use **Shadows** only at module, namespace, or file level (but not inside a procedure). This means you can declare shadowing variables in a source file or inside a module, class, or structure, but not inside a procedure. Note that you cannot specify both **Overloads** and **Shadows** in the same procedure declaration.
* **Shared**-Specifies that this Sub procedure is a shared procedure. As a shared procedure, it is not associated with a specific instance of a class or structure, and you can call it by qualifying it either with the class or structure name, or with the variable name of a specific instance of the class or structure.
* **Public**-Procedures declared **Public** have public access. There are no restrictions on the accessibility of public procedures.
* **Protected**-Procedures declared **Protected** have protected access. They are accessible only from within their own class or from a derived class. Protected access can be specified only on members of classes.
* **Friend**-Procedures declared **Friend** have friend access. They are accessible from within the program that contains their declaration and from anywhere else in the same assembly.
* **Protected Friend**-Procedures declared **Protected Friend** have both protected and friend accessibility. They can be used by code in the same assembly, as well as by code in derived classes.
* **Private**-Procedures declared **Private** have private access. They are accessible only within their declaration context, including from any nested procedures.
* ***name***-Name of the Sub procedure.
* ***arglist***-List of expressions (which can be single variables or simple values) representing arguments that are passed to the Sub procedure when it is called. Multiple arguments are separated by commas. Note that in VB .NET, if you supply an argument list, you must enclose it in parentheses.
* ***statements***-The block of statements to be executed within the Sub procedure.

Each argument in the *arglist* part has the following syntax and parts:

[ <*attrlist*> ] [ Optional ] [{ ByVal | ByRef }] [ ParamArray ] *argname*[( )]

[ As *argtype* ] [ = *defaultvalue* ]

Here are the parts of the *arglist*:

* ***attrlist***-List of attributes that apply to this argument. Multiple attributes are separated by commas.
* **Optional**-Specifies that this argument is not required when the procedure is called. Note that if you use this keyword, all following arguments in *arglist* must also be optional and be declared using the **Optional** keyword. Every optional argument declaration must supply a *defaultvalue*. Also, **Optional** cannot be used for any argument if you also use **ParamArray**.
* **ByVal**-Specifies passing by value. In this case, the procedure cannot replace or reassign the underlying variable element in the calling code (unless the argument is a reference type). **ByVal** is the default in Visual Basic.
* **ByRef**-Specifies passing by reference. In this case, the procedure can modify the underlying variable in the calling code the same way the calling code itself can.
* **ParamArray**-Used as the last argument in *arglist* to indicate that the final argument is an optional array of elements of the specified type. The **ParamArray** keyword allows you to pass an arbitrary number of arguments to the procedure. A **ParamArray** argument is always passed **ByVal**.
* ***argname***-Name of the variable representing the argument.
* ***argtype***-This part is optional unless **Option Strict** is set to **On**, and holds the data type of the argument passed to the procedure. Can be **Boolean**, **Byte**, **Char**, **Date**, **Decimal**, **Double**, **Integer**, **Long**, **Object**, **Short**, **Single**, or **String**; or the name of an enumeration, structure, class, or interface.
* ***defaultvalue***-Required for **Optional** arguments. Any constant or constant expression that evaluates to the data type of the argument. Note that if the type is **Object**, or a class, interface, array, or structure, the default value can be only **Nothing**.

Each attribute in the *attrlist* part has the following syntax and parts:

<*attrname* [({ *attrargs* | *attrinit* })]>

Here are the parts of *attrlist*:

* ***attrname***-Name of the attribute.
* ***attrargs***-List of positional arguments for this attribute. Multiple arguments are separated by commas.
* ***attrinit***-List of field or property initializers for this attribute. Multiple initializers are separated by commas.

|  |  |  |
| --- | --- | --- |
|  | Tip | When you use **ByVal** (the default in VB .NET), you pass a copy of a variable to a procedure; when you use **ByRef**, you pass a reference to the variable, and if you make changes to that reference, the original variable is changed. |

You call a Sub procedure using the procedure name followed by the argument list. The **Exit Sub** keywords cause an immediate exit from a Sub procedure. Finally, **End Sub** ends the procedure definition. Here's an example we saw in the In Depth section of this chapter, where I'm passing a text string, "Hello from Visual Basic", to the **DisplayMessage** Sub procedure, which displays that message in a console application:

Module Module1

Sub Main()

DisplayMessage("Hello from Visual Basic")

End Sub

Sub DisplayMessage(ByVal strText As String)

System.Console.WriteLine(strText)

End Sub

End Module

**Creating Functions**

Unlike Sub procedures (see the previous topic), functions can return values, as discussed in the In Depth section of this chapter. You use the **Function** statement to create a function:

[ <*attrlist*> ] [{ Overloads | Overrides | Overridable | NotOverridable |

MustOverride | Shadows | Shared }]

[{ Public | Protected | Friend | Protected Friend | Private }] Function

*name*[(*arglist*)] [ As *type* ]

[ *statements* ]

[ Exit Function ]

[ *statements* ]

End Function

|  |  |  |
| --- | --- | --- |
|  | Tip | When you use **ByVal** (the default in VB .NET), you pass a copy of a variable to a procedure; when you use **ByRef**, you pass a reference to the variable, and if you make changes to that reference, the original variable is changed. |

The various parts of this statement are the same as for Sub procedures (see the previous topic) except for the **As *type*** clause, which specifies the type of the return value from the function; here's how to set the *type* item:

* *type*-This is optional unless **Option Strict** is **On**. Data type of the value returned by the **Function** procedure can be **Boolean**, **Byte**, **Char**, **Date**, **Decimal**, **Double**, **Integer**, **Long**, **Object**, **Short**, **Single**, or **String**; or the name of an enumeration, structure, class, or interface.

|  |  |  |
| --- | --- | --- |
|  | Tip | If you use **Exit Function** without assigning a value to name, the function returns the default value appropriate to argtype. This is **0** for **Byte**, **Char**, **Decimal**, **Double**, **Integer**, **Long**, **Short**, and **Single**; **Nothing for Object**, **String**, and all arrays; **False** for **Boolean**; and **#1/1/0001 12:00 AM#** for **Date**. |

The **Return** statement simultaneously assigns the return value and exits the function; any number of **Return** statements can appear anywhere in the procedure. (You also can mix **Exit Function** and **Return** statements.) Here's an example function-**Addem**-we saw in the In Depth section of this chapter, which adds two integer values passed to it:

Module Module1

Sub Main()

Dim intValue As Integer = 2

System.Console.WriteLine("{0}+{1}={2}", \_

intValue, intValue, Addem(intValue, intValue))

End Sub

Function Addem(ByVal int1 As Integer, ByVal int2 As Integer) As Long

Return int1 + int2

End Function

End Module

## Commenting Your Procedures

In general, you should add a new comment when you declare a new and important variable, or if you wish to make clear some implementation method. Ideally, procedures should only have one purpose, and they should be named clearly enough so that excessive comments are not needed.

In addition, procedures should begin with a comment describing what the procedure does, and that comment should be broken up into various sections. The Microsoft recommendations for those sections appears in Table 3.1.

| Table 3.1: Procedure starting comment block sections. | |
| --- | --- |
| **Section heading** | **Comment description** |
| Purpose | What the procedure does. |
| Assumptions | List of each external variable, control, open file, or other element that is not obvious. |
| Effects | List of each affected external variable, control, or file and the effect it has (only if this is not obvious). |
| Inputs | Each argument that may not be obvious. Arguments are on a separate line with inline comments. |
| Returns | Explanation of the values returned by functions. |

Here's an example, showing how to set up a comment preceding a function named **dblSquare()**:

'\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

' dblSquare()

' Purpose: Squares a number

' Inputs: sngSquareMe, the value to be squared

' Returns: The input value squared

'\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function dblSquare() (sngSquareMe As Integer) As Double

dblSquare = sngSquareMe \* sngSquareMe 'Use \*, not ^2, for speed

End Function

| ***Related solution:*** | ***Found on page:*** |
| --- | --- |
| Commenting Your Code | 93 |

## Passing a Variable Number of Arguments

Usually, you cannot call a procedure with more arguments than the procedure declaration specifies. When you need an indefinite number of arguments, you can declare a *parameter array*, which allows a procedure to accept an array of values for an argument. You do not have to know the number of elements in the parameter array when you define the procedure. The array size is determined by each call to the procedure.

|  |  |  |
| --- | --- | --- |
|  | Note | In Visual Basic .NET, **ParamArray** arguments are always passed using **ByVal**. All of the arguments in the array must be of the data type of the **ParamArray** argument. |

Here's an example; in this case, I'll pass different numbers of arguments to a Sub procedure, **DisplayMessage**. As you can see in the way **DisplayMessage** is declared, all arguments after the first one go into the parameter array, and I can loop over the parameter array to get all arguments passed to us (note that I use the **UBound** function, which we first saw in the previous chapter, to determine the upper bound of the array—and notice also that I'm passing a compete array of text strings to **DisplayMessage** with no problem):

Module Module1

Sub Main()

DisplayMessage("First message:", "Hi")

DisplayMessage("Second message:", "Hello", "there")

Dim TextArray() As String = {"Hello", "from", "Visual", \_

"Basic"}

DisplayMessage("Third message:", TextArray)

Resume Next

End Sub

Sub DisplayMessage(ByVal Title As String, ByVal ParamArray \_

MessageText() As String)

Dim intLoopIndex As Integer

System.Console.WriteLine(Title)

For intLoopIndex = 0 To UBound(MessageText)

System.Console.WriteLine(MessageText(intLoopIndex))

Next intLoopIndex

End Sub

End Module

Here's what you see when this code runs:

First message:

Hi

Second message:

Hello

there

Third message:

Hello

from

Visual

Basic

## Specifying Optional Procedure Arguments

**You also can make arguments *optional* in VB .NET procedures if you use the Optional keyword when declaring those arguments.** Note that if you make one argument optional, all the following arguments must also be optional, and you have to specify a *default value* for each optional argument (although you can set them to the keyword **Nothing** if you wish). You specify a default value with **=** **default\_value** in the procedure's argument list. Here's an example where I'm making the string argument you pass to a Sub procedure named **DisplayMessage** optional, and giving that argument the default value "**Hello from Visual Basic**":

Module Module1

Sub Main()

DisplayMessage()

End Sub

Sub DisplayMessage(Optional ByVal strText As String = "Hello from Visual Basic")

System.Console.WriteLine(strText)

End Sub

End Module

Now when I call **DisplayMessage** with no arguments, as in the code above, the default value is used and this code displays:

Hello from Visual Basic

|  |  |  |
| --- | --- | --- |
|  | Tip | VB6 had a function named **IsMissing** that would test if an optional argument had been given a value or not, but now that all optional arguments have default values, **IsMissing** has been removed. You can, however, use the **IsNothing** function to check if an argument has been set to **Nothing**. |

## Preserving a Variable's Values between Procedure Calls

You've written a function named **Counter** to keep track of the number of times the user clicks a particular button. Each time through a loop, you call the **Counter** function to increment the count, but when the program ends, it just displays 1 counts. Why? Let's look at the code:

Module Module1

Sub Main()

Dim intLoopIndex As Integer, intValue = 0

For intLoopIndex = 0 To 4

intValue = Counter()

Next intLoopIndex

System.Console.WriteLine(intValue)

End Sub

Function Counter() As Integer

Dim intCountValue As Integer

intCountValue += 1

Return intCountValue

End Function

End Module

**The problem here is that the counter variable, intCountValue, in the Counter function is reinitialized each time the Counter function is called** (because a new copy of all the variables local to procedures is allocated each time you call that procedure). The solution is to declare **intCountValue** as ***static*. This means it will retain its value between calls to the Counter function.** Here's the working code:

Module Module1

Sub Main()

Dim intLoopIndex As Integer, intValue = 0

For intLoopIndex = 0 To 4

intValue = Counter()

Next intLoopIndex

System.Console.WriteLine(intValue)

End Sub

Function Counter() As Integer

Static intCountValue As Integer

intCountValue += 1

Return intCountValue

End Function

End Module

Running this code displays a value of **5**, as it should.

|  |  |  |
| --- | --- | --- |
|  | Tip | You can also make **intCountValue** preserve its value between procedure calls by making it a module-level variable-just declare it outside any procedure. But note that you should restrict the scope of your variables as much as possible (to avoid inadvertent conflicts with variables of the same name), so making this variable a static variable in a procedure is probably a better choice. |

|  |  |  |
| --- | --- | --- |
|  | Note | You were able to declare a whole function static in VB6, which meant that all the variables in it would be static, but you can't do that in VB .NET. |

## Understanding Scope

The *scope* of a variable or constant is the set of all code that can refer to it without qualifying its name. A variable's scope is determined by where the variable is declared. It's usually a good idea to make the scope of variables or constants as narrow as possible (block scope is the narrowest). This helps conserve memory and minimizes the chances of your code referring to the wrong item. I'll take a look at the different kinds of scope in VB .NET here.

#### Block Scope

As discussed in the In Depth section of this chapter, a block is a series of statements terminated by an **End**, **Else**, **Loop**, or **Next** statement, and an element declared within a block can be used only within that block. Here's what block scope looks like in an example from the In Depth section of this chapter. In this case, I'll declare a variable, **strText** in an **If** statement. That variable can be used *inside* the **If** statement's block, but not *outside* (VB .NET will tag the second use here as a syntax error):

Module Module1

Sub Main()

Dim intValue As Integer = 1

If intValue = 1 Then

Dim strText As String = "No worries."

System.Console.WriteLine(strText)

End If

System.Console.WriteLine(strText) 'Will not work!

End Sub

End Module

#### Procedure Scope

An element declared within a procedure is not available outside that procedure, and only the procedure that contains the declaration can use it. Elements at this level are also called *local* elements, and you declare them with the **Dim** or **Static** statement.

Note also that if an element is declared inside a procedure but outside any block within that procedure, the element can be thought of as having block scope, where the block is the entire procedure.

#### Module Scope

When discussing scope, Visual Basic uses the term *module level* to apply equally to modules, classes, and structures. You can declare elements at this level by placing the declaration statement outside of any procedure or block within the module, class, or structure.

When you make a declaration at the module level, the accessibility you choose determines the scope. The namespace that contains the module, class, or structure also affects the scope.

Elements for which you declare **Private** accessibility are available for reference to every procedure in that module, but not to any code in a different module. The **Dim** statement at module level defaults to **Private** accessibility, so it is equivalent to using the **Private** statement. However, you can make the scope and accessibility more obvious by using **Private**. In this example from the In Depth section of this chapter, I've declared **Function1** as private to **Module2**, so it's inaccessible in **Module1** (VB .NET will tag **Module2.Function1** below as a syntax error):

Module Module1

Sub Main()

System.Console.WriteLine(Module2.Function1()) 'Will not work!

End Sub

End Module

Module Module2

Private Function Function1() As String

Return "Hello from Visual Basic"

End Function

End Module

#### Namespace Scope

If you declare an element at module level using the **Friend** or **Public** statement, it becomes available to all procedures throughout the entire namespace in which it is declared. Note that an element accessible in a namespace is also accessible from inside any namespace nested inside that namespace.

|  |  |  |
| --- | --- | --- |
|  | Note | If your project does not contain any **namespace** statements, everything in the project is in the same namespace. In this case, namespace scope can be thought of as procedure scope. |