variable

variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in VB.Net has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

We have already discussed various data types. The basic

Type	Example
Integral types	SByte, Byte, Short, UShort, Integer, UInteger, Long, ULong and Char
Floating point types	Single and Double
Decimal types	Decimal
Boolean types	True or False values, as assigned
Date types	Date

value types provided in VB.Net can be categorized as –

VB.Net also allows defining other value types of variable like **Enum** and reference types of variables like **Class**. We will discuss date types and Classes in subsequent chapters.

Variable Declaration in VB.Net

The **Dim** statement is used for variable declaration and storage allocation for one or more variables. The Dim statement is used at module, class, structure, procedure or block level.

Syntax for variable declaration in VB.Net is −

```
[ < attributelist > ] [ accessmodifier ] [[ Shared ] [
Shadows ] | [ Static ]]
[ ReadOnly ] Dim [ WithEvents ] variablelist
```

Where,

- ***attributelist*** is a list of attributes that apply to the variable. Optional.
- ***accessmodifier*** defines the access levels of the variables, it has values as - Public, Protected, Friend, Protected Friend and Private. Optional.
- ***Shared*** declares a shared variable, which is not associated with any specific instance of a class or structure, rather available to all the instances of the class or structure. Optional.
- ***Shadows*** indicate that the variable re-declares and hides an identically named element, or set of overloaded elements, in a base class. Optional.

- ***Static*** indicates that the variable will retain its value, even when the after termination of the procedure in which it is declared. Optional.
- ***ReadOnly*** means the variable can be read, but not written. Optional.
- ***WithEvents*** specifies that the variable is used to respond to events raised by the instance assigned to the variable. Optional.
- ***Variablelist*** provides the list of variables declared.

Each variable in the variable list has the following syntax and parts –

variablename[([boundslist])] [As [New] datatype] [= initializer]

Where,

- ***variablename*** – is the name of the variable
- ***boundslist*** – optional. It provides list of bounds of each dimension of an array variable.
- ***New*** – optional. It creates a new instance of the class when the Dim statement runs.
- ***datatype*** – Required if Option Strict is On. It specifies the data type of the variable.
- ***initializer*** – Optional if New is not specified. Expression that is evaluated and assigned to the variable when it is created.

Some valid variable declarations along with their definition are shown here –

```
Dim StudentID As Integer
Dim StudentName As String
```

```
Dim Salary As Double
Dim count1, count2 As Integer
Dim status As Boolean
Dim exitButton As New System.Windows.Forms.Button
Dim lastTime, nextTime As Date
```

Variable Initialization in VB.Net

Variables are initialized (assigned a value) with an equal sign followed by a constant expression. The general form of initialization is −

variable_name = value;

for example,

```
Dim pi As Double
pi = 3.14159
```

You can initialize a variable at the time of declaration as follows −

```
Dim StudentID As Integer = 100
Dim StudentName As String = "Bill Smith"
```

Example

Try the following example which makes use of various types of variables −

```
Module variablesNdataypes
   Sub Main()
      Dim a As Short
      Dim b As Integer
      Dim c As Double
```

```
      a = 10
      b = 20
      c = a + b
      Console.WriteLine("a = {0}, b = {1}, c = {2}", a, b,
c)
      Console.ReadLine()
   End Sub
End Module
```

When the above code is compiled and executed, it produces the following result −

```
a = 10, b = 20, c = 30
```

Accepting Values from User

The Console class in the System namespace provides a function **ReadLine** for accepting input from the user and store it into a variable. For example,

```
Dim message As String
message = Console.ReadLine
```

The following example demonstrates it −

```
Module variablesNdataypes
   Sub Main()
      Dim message As String
      Console.Write("Enter message: ")
      message = Console.ReadLine
      Console.WriteLine()
      Console.WriteLine("Your Message: {0}", message)
      Console.ReadLine()
   End Sub
End Module
```

When the above code is compiled and executed, it produces the following result (assume the user inputs Hello World) –

```
Enter message: Hello World
Your Message: Hello World
```

Lvalues and Rvalues

There are two kinds of expressions –

- **lvalue** – An expression that is an lvalue may appear as either the left-hand or right-hand side of an assignment.
- **rvalue** – An expression that is an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned and can not appear on the left-hand side. Following is a valid statement –

```
Dim g As Integer = 20
```

But following is not a valid statement and would generate compile-time error –

```
20 = g
```

VB.Net - Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. VB.Net is rich in built-in operators and provides following types of commonly used operators −

- Arithmetic Operators
- Comparison Operators
- Logical/Bitwise Operators
- Bit Shift Operators
- Assignment Operators
- Miscellaneous Operators

This tutorial will explain the most commonly used operators.

Arithmetic Operators

Following table shows all the arithmetic operators supported by VB.Net. Assume variable **A** holds 2 and variable **B** holds 7, then −

Show Examples

Operator	Description	Example
^	Raises one operand to the power of another	B^A will give 49
+	Adds two operands	A + B will give 9
-	Subtracts second operand from the	A - B will give -5

	first	
*	Multiplies both operands	A * B will give 14
/	Divides one operand by another and returns a floating point result	B / A will give 3.5
\	Divides one operand by another and returns an integer result	B \ A will give 3
MOD	Modulus Operator and remainder of after an integer division	B MOD A will give 1

Operator	Description	Example
=	Checks if the values of two operands are equal or not; if yes, then condition becomes true.	(A = B) is not true.
<>	Checks if the values of two operands are equal or not; if values are not equal, then condition becomes true.	(A <> B) is true.
>	Checks if the value of left operand is greater than the value of right operand; if yes, then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand; if yes, then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand; if yes, then condition becomes true.	(A <= B) is true.

Comparison Operators

Following table shows all the comparison operators supported by VB.Net. Assume variable **A** holds 10 and variable **B**holds 20, then −

Show Examples

Apart from the above, VB.Net provides three more comparison operators, which we will be using in forthcoming chapters; however, we give a brief description here.

- **Is** Operator – It compares two object reference variables and determines if two object references refer to the same object without performing value comparisons. If object1 and object2 both refer to the exact same object instance, result is **True**; otherwise, result is False.
- **IsNot** Operator – It also compares two object reference variables and determines if two object references refer to different objects. If object1 and object2 both refer to the exact same object instance, result is **False**; otherwise, result is True.
- **Like** Operator – It compares a string against a pattern.

Logical/Bitwise Operators

Following table shows all the logical operators supported by VB.Net. Assume variable A holds Boolean value True and variable B holds Boolean value False, then –

Show Examples

Operator	Description	Example
And	It is the logical as well as bitwise AND operator. If both the operands are true, then condition becomes	(A And B) is

	true. This operator does not perform short-circuiting, i.e., it evaluates both the expressions.	False.
Or	It is the logical as well as bitwise OR operator. If any of the two operands is true, then condition becomes true. This operator does not perform short-circuiting, i.e., it evaluates both the expressions.	(A Or B) is True.
Not	It is the logical as well as bitwise NOT operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	Not(A And B) is True.
Xor	It is the logical as well as bitwise Logical Exclusive OR operator. It returns True if both expressions are True or both expressions are False; otherwise it returns False. This operator does not perform short-circuiting, it always evaluates both expressions and there is no short-circuiting counterpart of this operator.	A Xor B is True.
AndAlso	It is the logical AND operator. It works only on Boolean data. It performs short-circuiting.	(A AndAlso B) is False.

OrElse	It is the logical OR operator. It works only on Boolean data. It performs short-circuiting.	(A OrElse B) is True.
IsFalse	It determines whether an expression is False.	
IsTrue	It determines whether an expression is True.	

Bit Shift Operators

We have already discussed the bitwise operators. The bit shift operators perform the shift operations on binary values. Before coming into the bit shift operators, let us understand the bit operations.

Bitwise operators work on bits and perform bit-by-bit

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

operations. The truth tables for &, |, and ^ are as follows –

Assume if A = 60; and B = 13; now in binary format they will be as follows –

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

We have seen that the Bitwise operators supported by VB.Net are And, Or, Xor and Not. The Bit shift operators are >> and << for left shift and right shift, respectively.

Assume that the variable A holds 60 and variable B holds 13, then –

Operator	Description	Example
And	Bitwise AND Operator copies a bit to the result if it exists in both operands.	(A AND B) will give 12, which is 0000 1100
Or	Binary OR Operator copies a bit if it exists in either operand.	(A Or B) will give 61,

		which is 0011 1101
Xor	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A Xor B) will give 49, which is 0011 0001
Not	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(Not A) will give -61, which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240, which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits	A >> 2 will give 15, which is

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left specified by the right operand.	C = A + B will assign value of A + B will 0000 1111

Show Examples

Assignment Operators

There are following assignment operators supported by VB.Net −

Show Examples

	side operand	B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assigns the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assigns the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assigns the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assigns the result to left operand (floating point division)	C /= A is equivalent to C = C / A
\=	Divide AND assignment operator, It divides left operand with the right operand and assigns the result to left operand (Integer division)	C \= A is equivalent to C = C \A

^=	Exponentiation and assignment operator. It raises the left operand to the power of the right operand and assigns the result to left operand.	C^=A is equivalent to C = C ^ A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Concatenates a String expression to a String variable or property and assigns the result to the variable or property.	Str1 &= Str2 is same as Str1 = Str1 & Str2

Miscellaneous Operators

There are few other important operators supported by VB.Net.

Operator	Description	Example
AddressOf	Returns the address of a procedure.	`AddHandler Button1.Click, AddressOf Button1_Click`
Await	It is applied to an operand in an	`Dim result As res`

	asynchronous method or lambda expression to suspend execution of the method until the awaited task completes.	`= Await AsyncMethodThatReturnsResult()` `Await AsyncMethod()`
GetType	It returns a Type object for the specified type. The Type object provides information about the type such as its properties, methods, and events.	`MsgBox(GetType(Integer).ToString())`

Function Expression	It declares the parameters and code that define a function lambda expression.	`Dim add5 = Function(num As Integer) num + 5`
		`'prints 10`
		`Console.WriteLine(add5(5))`

If	It uses short-circuit evaluation to conditionally return one of two values. The If operator can be called with three arguments or with	`Dim num = 5` `Console.WriteLine(If(num >= 0, "Positive", "Negative"))`

	two arguments.	

Show Examples

Operators Precedence in VB.Net

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator −

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Show Examples

Operator	Precedence
Await	Highest
Exponentiation (^)	
Unary identity and negation (+, -)	

Multiplication and floating-point division (*, /)	
Integer division (\)	
Modulus arithmetic (Mod)	
Addition and subtraction (+, -)	
Arithmetic bit shift (<<, >>)	
All comparison operators (=, <>, <, <=, >, >=, Is, IsNot, Like, TypeOf...Is)	
Negation (Not)	
Conjunction (And, AndAlso)	
Inclusive disjunction (Or, OrElse)	
Exclusive disjunction (Xor)	

VB.Net - If...Then...Else Statement

An **If** statement can be followed by an optional **Else** statement, which executes when the Boolean expression is false.

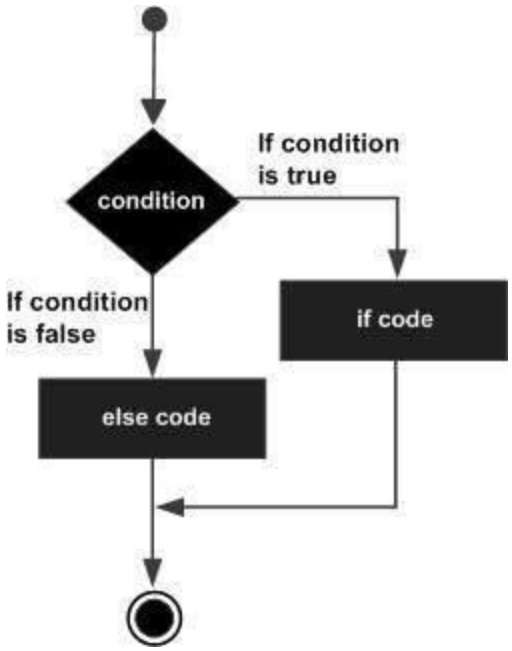
Syntax

The syntax of an If...Then... Else statement in VB.Net is as follows −

```
If(boolean_expression)Then
   'statement(s) will execute if the Boolean expression is
true
Else
  'statement(s) will execute if the Boolean expression is
false
End If
```

If the Boolean expression evaluates to **true**, then the if block of code will be executed, otherwise else block of code will be executed.

Flow Diagram



Example

```
Module decisions
   Sub Main()
       'local variable definition '
      Dim a As Integer = 100

      ' check the boolean condition using if statement
      If (a < 20) Then
          ' if condition is true then print the following
          Console.WriteLine("a is less than 20")
      Else
          ' if condition is false then print the following
          Console.WriteLine("a is not less than 20")
      End If
      Console.WriteLine("value of a is : {0}", a)
      Console.ReadLine()
   End Sub
End Module
```

When the above code is compiled and executed, it produces the following result –

```
a is not less than 20
value of a is : 100
```

The If...Else If...Else Statement

An **If** statement can be followed by an optional **Else if...Else** statement, which is very useful to test various conditions using single If...Else If statement.

When using If... Else If... Else statements, there are few points to keep in mind.

- An If can have zero or one Else's and it must come after an Else If's.
- An If can have zero to many Else If's and they must come before the Else.
- Once an Else if succeeds, none of the remaining Else If's or Else's will be tested.

Syntax

The syntax of an if...else if...else statement in VB.Net is as follows –

```
If(boolean_expression 1)Then
   ' Executes when the boolean expression 1 is true
ElseIf( boolean_expression 2)Then
   ' Executes when the boolean expression 2 is true
ElseIf( boolean_expression 3)Then
   ' Executes when the boolean expression 3 is true
Else
   ' executes when the none of the above condition is true
End If
```

Example

```
Module decisions
   Sub Main()
      'local variable definition '
      Dim a As Integer = 100
      ' check the boolean condition '
      If (a = 10) Then
         ' if condition is true then print the following '
         Console.WriteLine("Value of a is 10") '
      ElseIf (a = 20) Then
         'if else if condition is true '
         Console.WriteLine("Value of a is 20") '
      ElseIf (a = 30) Then
         'if else if condition is true
         Console.WriteLine("Value of a is 30")
      Else
         'if none of the conditions is true
         Console.WriteLine("None of the values is
matching")
      End If
      Console.WriteLine("Exact value of a is: {0}", a)
      Console.ReadLine()
   End Sub
End Module
```

When the above code is compiled and executed, it produces the following result –

```
None of the values is matching
Exact value of a is: 100
```

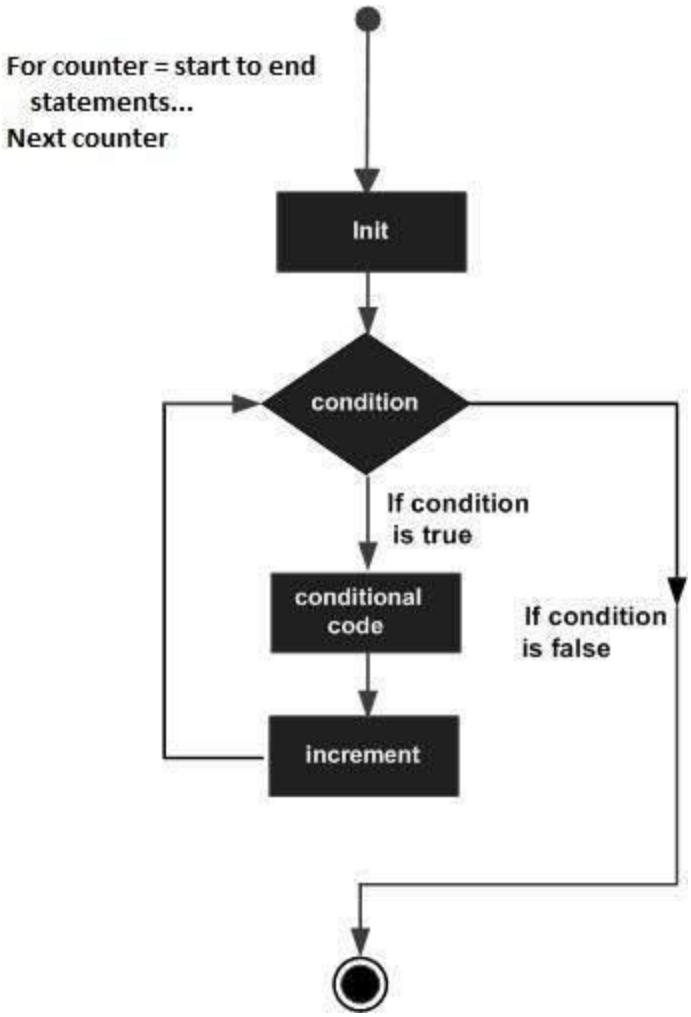
VB.Net - For...Next Loop

It repeats a group of statements a specified number of times and a loop index counts the number of loop iterations as the loop executes.

The syntax for this loop construct is −

```
For counter [ As datatype ] = start To end [ Step step ]
   [ statements ]
   [ Continue For ]
   [ statements ]
   [ Exit For ]
   [ statements ]
Next [ counter ]
```

Flow Diagram



Example

```
Module loops
   Sub Main()
      Dim a As Byte
      ' for loop execution
      For a = 10 To 20
          Console.WriteLine("value of a: {0}", a)
      Next
      Console.ReadLine()
   End Sub
```

```
End Module
```

When the above code is compiled and executed, it produces the following result –

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
value of a: 20
```

If you want to use a step size of 2, for example, you need to display only even numbers, between 10 and 20 –

```
Module loops
   Sub Main()
      Dim a As Byte
      ' for loop execution
      For a = 10 To 20 Step 2
         Console.WriteLine("value of a: {0}", a)
      Next
      Console.ReadLine()
   End Sub
End Module
```

When the above code is compiled and executed, it produces the following result –

```
value of a: 10
value of a: 12
value of a: 14
value of a: 16
value of a: 18
value of a: 20
```

VB.Net - Each...Next Loop

It repeats a group of statements for each element in a collection. This loop is used for accessing and manipulating all elements in an array or a VB.Net collection.

The syntax for this loop construct is –

```
For Each element [ As datatype ] In group
   [ statements ]
   [ Continue For ]
   [ statements ]
   [ Exit For ]
   [ statements ]
Next [ element ]
```

Example

```
Module loops
   Sub Main()
```

```
      Dim anArray() As Integer = {1, 3, 5, 7, 9}
      Dim arrayItem As Integer
     'displaying the values

      For Each arrayItem In anArray
         Console.WriteLine(arrayItem)
      Next
      Console.ReadLine()
   End Sub
End Module
```

When the above code is compiled and executed, it produces the following result –

```
1
3
5
7
9
```

VB.Net - Do Loop

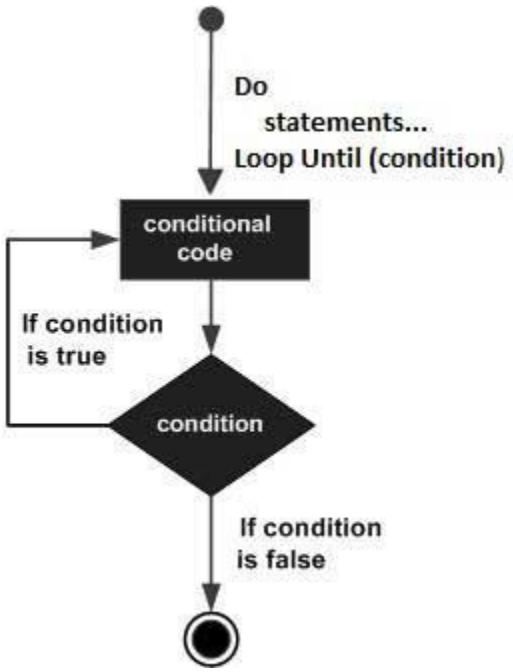
It repeats the enclosed block of statements while a Boolean condition is True or until the condition becomes True. It could be terminated at any time with the Exit Do statement.

The syntax for this loop construct is –

```
Do { While | Until } condition
```

```
    [ statements ]
    [ Continue Do ]
    [ statements ]
    [ Exit Do ]
    [ statements ]
Loop
-or-
Do
    [ statements ]
    [ Continue Do ]
    [ statements ]
    [ Exit Do ]
    [ statements ]
Loop { While | Until } condition
```

Flow Diagram



Example

Module loops

```
   Sub Main()
      ' local variable definition
      Dim a As Integer = 10
      'do loop execution
      Do
          Console.WriteLine("value of a: {0}", a)
          a = a + 1
      Loop While (a < 20)
      Console.ReadLine()
   End Sub
End Module
```

When the above code is compiled and executed, it produces the following result –

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

The program would behave in same way, if you use an Until statement, instead of While –

```
Module loops
   Sub Main()
      ' local variable definition
      Dim a As Integer = 10
```

```
      'do loop execution

      Do
         Console.WriteLine("value of a: {0}", a)
         a = a + 1
      Loop Until (a = 20)
      Console.ReadLine()
   End Sub
End Module
```

When the above code is compiled and executed, it produces the following result –

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

VB.Net - While... End While Loop

It executes a series of statements as long as a given condition is True.

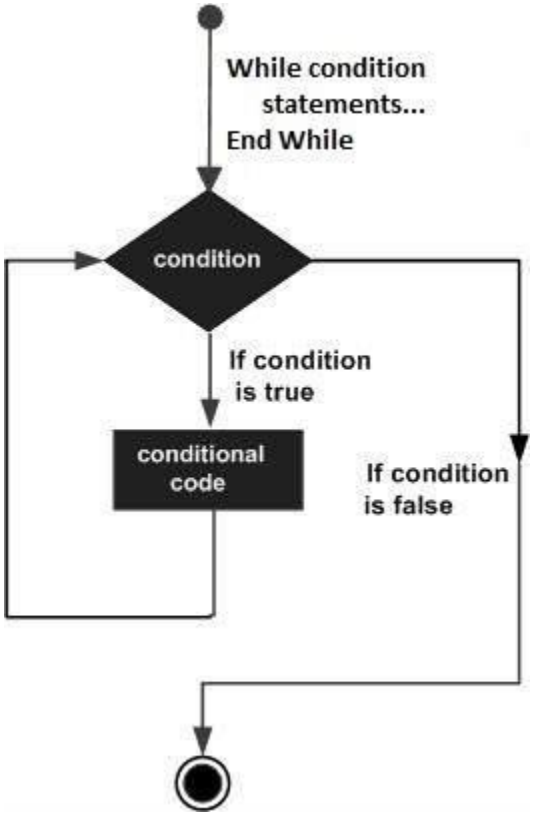
The syntax for this loop construct is –

```
While condition
   [ statements ]
   [ Continue While ]
   [ statements ]
   [ Exit While ]
   [ statements ]
End While
```

Here, statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is logical true. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

Flow Diagram



Here, key point of the *While* loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```
Module loops
   Sub Main()
      Dim a As Integer = 10
      ' while loop execution '

      While a < 20
         Console.WriteLine("value of a: {0}", a)
         a = a + 1
      End While
      Console.ReadLine()
   End Sub
End Module
```

When the above code is compiled and executed, it produces the following result –

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Casting

Casting is the process of converting one data type to another, for example, from an Integer type to a String type. Some operations in VB.NET require specific data types to work. Casting creates the type you need. The first article in this two-part series, Casting and Data Type Conversions in VB.NET, introduces casting. This article describes the three operators you can use to cast in VB.NET - DirectCast, CType and TryCast - and compares their performance.

Performance is one of the big differences between the three casting operators according to Microsoft and other articles. For example, Microsoft is usually careful to warn that, "DirectCast ... can provide somewhat better performance than CType *when converting to and from data type Object*." (Emphasis added.)

I decided to write some code to check.

But first a word of caution. Dan Appleman, one of the founders of the technical book publisher Apress and a reliable technical guru, once told me that benchmarking performance is much harder to do correctly than most people realize. There are factors like machine performance, other processes that might be running in parallel, optimization like memory caching or compiler optimization, and errors in your assumptions about what the code is actually doing. In these benchmarks, I have

tried to eliminate "apples and oranges" comparison errors and all tests have been run with the release build. But there still might be errors in these results. If you notice any, please let me know.

The three casting operators are:

- DirectCast
- CType
- TryCast

VB.Net - Functions

A procedure is a group of statements that together perform a task when called. After the procedure is executed, the control returns to the statement calling the procedure. VB.Net has two types of procedures −

- Functions
- Sub procedures or Subs

Functions return a value, whereas Subs do not return a value.

Defining a Function

The Function statement is used to declare the name, parameter and the body of a function. The syntax for the Function statement is −

[Modifiers] Function FunctionName [(ParameterList)] As ReturnType
   [Statements]
End Function

Where,

- ***Modifiers*** – specify the access level of the function; possible values are: Public, Private, Protected, Friend, Protected Friend and information regarding overloading, overriding, sharing, and shadowing.
- ***FunctionName*** – indicates the name of the function
- ***ParameterList*** – specifies the list of the parameters
- ***ReturnType*** – specifies the data type of the variable the function returns

Example

Following code snippet shows a function *FindMax* that takes two integer values and returns the larger of the two.

```
Function FindMax(ByVal num1 As Integer, ByVal num2 As Integer) As Integer
   ' local variable declaration */
   Dim result As Integer

   If (num1 > num2) Then
      result = num1
   Else
      result = num2
   End If
```

```
   FindMax = result
End Function
```

Function Returning a Value

In VB.Net, a function can return a value to the calling code in two ways –

- By using the return statement
- By assigning the value to the function name

The following example demonstrates using the *FindMax* function –

```
Module myfunctions
   Function FindMax(ByVal num1 As Integer, ByVal num2
As Integer) As Integer
      ' local variable declaration */
      Dim result As Integer

      If (num1 > num2) Then
         result = num1
      Else
         result = num2
      End If
      FindMax = result
   End Function
   Sub Main()
      Dim a As Integer = 100
      Dim b As Integer = 200
      Dim res As Integer

      res = FindMax(a, b)
      Console.WriteLine("Max value is : {0}", res)
```

```
      Console.ReadLine()
   End Sub
End Module
```

When the above code is compiled and executed, it produces the following result –

```
Max value is : 200
```

Recursive Function

A function can call itself. This is known as recursion. Following is an example that calculates factorial for a given number using a recursive function –

```
Module myfunctions
   Function factorial(ByVal num As Integer) As Integer
      ' local variable declaration */
      Dim result As Integer

      If (num = 1) Then
         Return 1
      Else
         result = factorial(num - 1) * num
         Return result
      End If
   End Function
   Sub Main()
      'calling the factorial method
      Console.WriteLine("Factorial of 6 is : {0}",
factorial(6))
      Console.WriteLine("Factorial of 7 is : {0}",
factorial(7))
      Console.WriteLine("Factorial of 8 is : {0}",
factorial(8))
```

```
      Console.ReadLine()
   End Sub
End Module
```

When the above code is compiled and executed, it produces the following result –

```
Factorial of 6 is: 720
Factorial of 7 is: 5040
Factorial of 8 is: 40320
```

Param Arrays

At times, while declaring a function or sub procedure, you are not sure of the number of arguments passed as a parameter. VB.Net param arrays (or parameter arrays) come into help at these times.

The following example demonstrates this –

```
Module myparamfunc
   Function AddElements(ParamArray arr As Integer()) As
Integer
      Dim sum As Integer = 0
      Dim i As Integer = 0

      For Each i In arr
         sum += i
      Next i
      Return sum
   End Function
   Sub Main()
      Dim sum As Integer
      sum = AddElements(512, 720, 250, 567, 889)
      Console.WriteLine("The sum is: {0}", sum)
      Console.ReadLine()
```

```
   End Sub
End Module
```

When the above code is compiled and executed, it produces the following result –

```
The sum is: 2938
```

Passing Arrays as Function Arguments

You can pass an array as a function argument in VB.Net. The following example demonstrates this –

```
Module arrayParameter
   Function getAverage(ByVal arr As Integer(), ByVal size
As Integer) As Double
      'local variables
      Dim i As Integer
      Dim avg As Double
      Dim sum As Integer = 0

      For i = 0 To size - 1
         sum += arr(i)
      Next i
      avg = sum / size
      Return avg
   End Function
   Sub Main()
      ' an int array with 5 elements '
      Dim balance As Integer() = {1000, 2, 3, 17, 50}
      Dim avg As Double
      'pass pointer to the array as an argument
      avg = getAverage(balance, 5)
      ' output the returned value '
      Console.WriteLine("Average value is: {0} ", avg)
```

```
    Console.ReadLine()
  End Sub
End Module
```

When the above code is compiled and executed, it produces the following result –

Average value is: 214.4

Subroutine

A subroutine is a code construct.
A code construct is a pattern of coding. Examples;- For Loops, Do Whiles Loops, Sub ... End Sub, Function .... End Function
A subroutine is a self contained section of code, that;-
Has Inputs: Possible
Has Output: Never

```
1 Private Sub PrintHelloWorld()
2 Console.WriteLine("Hello World")
3 End Sub
```

Where you put your code to handle a button click is particular type of subroutine called **An Event Handler** Indicated by the Handles Button1.Click.

```
1Private Sub Button1_Click(ByVal sender As System.Obje
```

```
 ct, ByVal e As System.EventArgs) HandlesButton1.Click
2
3  End Sub
```