VB.NET

programming language reference

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com


https://www.facebook.com/tutorialspointindia


https://twitter.com/tutorialspoint

About the Tutorial

VB.Net is a simple, modern, object-oriented computer programming language developed by Microsoft to combine the power of .NET Framework and the common language runtime with the productivity benefits that are the hallmark of Visual Basic.

This tutorial will teach you basic VB.Net programming and will also take you through various advanced concepts related to VB.Net programming language.

Audience

This tutorial has been prepared for the beginners to help them understand basic VB.Net programming. After completing this tutorial, you will find yourself at a moderate level of expertise in VB.Net programming from where you can take yourself to next levels.

Prerequisites

VB.Net programming is very much based on BASIC and Visual Basic programming languages, so if you have basic understanding on these programming languages, then it will be a fun for you to learn VB.Net programming language.

Copyright & Disclaimer

Table of Contents

1. Overview

Visual Basic .NET (VB.NET) is an object-oriented computer programming language implemented on the .NET Framework. Although it is an evolution of classic Visual Basic language, it is not backwards-compatible with VB6, and any code written in the old version does not compile under VB.NET.

Like all other .NET languages, VB.NET has complete support for object-oriented concepts. Everything in VB.NET is an object, including all of the primitive types (Short, Integer, Long, String, Boolean, etc.) and user-defined types, events, and even assemblies. All objects inherits from the base class Object.

VB.NET is implemented by Microsoft's .NET framework. Therefore, it has full access to all the libraries in the .Net Framework. It's also possible to run VB.NET programs on Mono, the open-source alternative to .NET, not only under Windows, but even Linux or Mac OSX.

The following reasons make VB.Net a widely used professional language:

- Modern, general purpose.
- Object oriented.
- Component oriented.
- Easy to learn.
- Structured language.
- It produces efficient programs.
- It can be compiled on a variety of computer platforms.
- Part of .Net Framework.

Strong Programming Features VB.Net

VB.Net has numerous strong programming features that make it endearing to multitude of programmers worldwide. Let us mention some of these features:

- Boolean Conditions
- Automatic Garbage Collection
- Standard Library

- Assembly Versioning
- Properties and Events
- Delegates and Events Management
- Easy-to-use Generics
- Indexers
- Conditional Compilation
- Simple Multithreading

2. Environment Setup

In this chapter, we will discuss the tools available for creating VB.Net applications.

We have already mentioned that VB.Net is part of .Net framework and used for writing .Net applications. Therefore before discussing the available tools for running a VB.Net program, let us understand how VB.Net relates to the .Net framework.

The .Net Framework

The .Net framework is a revolutionary platform that helps you to write the following types of applications:

- Windows applications
- Web applications
- Web services

The .Net framework applications are multi-platform applications. The framework has been designed in such a way that it can be used from any of the following languages: Visual Basic, C#, C++, Jscript, and COBOL, etc.

All these languages can access the framework as well as communicate with each other.

The .Net framework consists of an enormous library of codes used by the client languages like VB.Net. These languages use object-oriented methodology.

Following are some of the components of the .Net framework:

- Common Language Runtime (CLR)
- The .Net Framework Class Library
- Common Language Specification
- Common Type System
- Metadata and Assemblies
- Windows Forms
- ASP.Net and ASP.Net AJAX
- ADO.Net

- Windows Workflow Foundation (WF)
- Windows Presentation Foundation
- Windows Communication Foundation (WCF)
- LINQ

For the jobs each of these components perform, please see ASP.Net - Introduction, and for details of each component, please consult Microsoft's documentation.

Integrated Development Environment (IDE) For VB.Net

Microsoft provides the following development tools for VB.Net programming:

- Visual Studio 2010 (VS)
- Visual Basic 2010 Express (VBE)
- Visual Web Developer

The last two are free. Using these tools, you can write all kinds of VB.Net programs from simple command-line applications to more complex applications. Visual Basic Express and Visual Web Developer Express edition are trimmed down versions of Visual Studio and has the same look and feel. They retain most features of Visual Studio. In this tutorial, we have used Visual Basic 2010 Express and Visual Web Developer (for the web programming chapter).

You can download it from here. It gets automatically installed in your machine. Please note that you need an active internet connection for installing the express edition.

Writing VB.Net Programs on Linux or Mac OS

Although the .NET Framework runs on the Windows operating system, there are some alternative versions that work on other operating systems. Mono is an open-source version of the .NET Framework which includes a Visual Basic compiler and runs on several operating systems, including various flavors of Linux and Mac OS. The most recent version is VB 2012.

The stated purpose of Mono is not only to be able to run Microsoft .NET applications cross-platform, but also to bring better development tools to Linux developers. Mono can be run on many operating systems including Android, BSD, iOS, Linux, OS X, Windows, Solaris and UNIX.

3. Program Structure

Before we study basic building blocks of the VB.Net programming language, let us look a bare minimum VB.Net program structure so that we can take it as a reference in upcoming chapters.

VB.Net Hello World Example

A VB.Net program basically consists of the following parts:

- Namespace declaration
- A class or module
- One or more procedures
- Variables
- The Main procedure
- Statements & Expressions
- Comments

Let us look at a simple code that would print the words "Hello World":

```
Imports System
Module Module1
   'This program will display Hello World
   Sub Main()
      Console.WriteLine("Hello World")
      Console.ReadKey()
   End Sub
End Module
```

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

```
Hello, World!
```

Let us look various parts of the above program:

- The first line of the program **Imports System** is used to include the System namespace in the program.

- The next line has a **Module** declaration, the module *Module1*. VB.Net is completely object oriented, so every program must contain a module of a class that contains the data and procedures that your program uses.

- Classes or Modules generally would contain more than one procedure. Procedures contain the executable code, or in other words, they define the behavior of the class. A procedure could be any of the following:
  - Function
  - Sub
  - Operator
  - Get
  - Set
  - AddHandler
  - RemoveHandler
  - RaiseEvent
- The next line ('This program) will be ignored by the compiler and it has been put to add additional comments in the program.

- The next line defines the Main procedure, which is the entry point for all VB.Net programs. The Main procedure states what the module or class will do when executed.

- The Main procedure specifies its behavior with the statement **Console.WriteLine ("Hello World")** *WriteLine* is a method of the *Console* class defined in the *System* namespace. This statement causes the message "Hello, World!" to be displayed on the screen.

- The last line **Console.ReadKey()** is for the VS.NET Users. This will prevent the screen from running and closing quickly when the program is launched from Visual Studio .NET.

Compile & Execute VB.Net Program

If you are using Visual Studio.Net IDE, take the following steps:

- Start Visual Studio.

- On the menu bar, choose File → New → Project.

- Choose Visual Basic from templates

- Choose Console Application.
- Specify a name and location for your project using the Browse button, and then choose the OK button.
- The new project appears in Solution Explorer.
- Write code in the Code Editor.
- Click the Run button or the F5 key to run the project. A Command Prompt window appears that contains the line Hello World.

You can compile a VB.Net program by using the command line instead of the Visual Studio IDE:

- Open a text editor and add the above mentioned code.
- Save the file as **helloworld.vb**
- Open the command prompt tool and go to the directory where you saved the file.
- Type **vbc helloworld.vb** and press enter to compile your code.
- If there are no errors in your code the command prompt will take you to the next line and would generate **helloworld.exe** executable file.
- Next, type **helloworld** to execute your program.
- You will be able to see "Hello World" printed on the screen.

4. Basic Syntax

VB.Net is an object-oriented programming language. In Object-Oriented Programming methodology, a program consists of various objects that interact with each other by means of actions. The actions that an object may take are called methods. Objects of the same kind are said to have the same type or, more often, are said to be in the same class.

When we consider a VB.Net program, it can be defined as a collection of objects that communicate via invoking each other's methods. Let us now briefly look into what do class, object, methods, and instant variables mean.

- **Object** - Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors - wagging, barking, eating, etc. An object is an instance of a class.

- **Class** - A class can be defined as a template/blueprint that describes the behaviors/states that object of its type support.

- **Methods** - A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.

- **Instant Variables** - Each object has its unique set of instant variables. An object's state is created by the values assigned to these instant variables.

A Rectangle Class in VB.Net

For example, let us consider a Rectangle object. It has attributes like length and width. Depending upon the design, it may need ways for accepting the values of these attributes, calculating area and displaying details.

Let us look at an implementation of a Rectangle class and discuss VB.Net basic syntax on the basis of our observations in it:

```
Imports System

Public Class Rectangle

    Private length As Double

    Private width As Double


    'Public methods

    Public Sub AcceptDetails()
```

```
        length = 4.5
        width = 3.5
    End Sub

    Public Function GetArea() As Double
        GetArea = length * width
    End Function
    Public Sub Display()
        Console.WriteLine("Length: {0}", length)
        Console.WriteLine("Width: {0}", width)
        Console.WriteLine("Area: {0}", GetArea())

    End Sub

    Shared Sub Main()
        Dim r As New Rectangle()
        r.Acceptdetails()
        r.Display()
        Console.ReadLine()
    End Sub
End Class
```

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

```
Length: 4.5
Width: 3.5
Area: 15.75
```

In previous chapter, we created a Visual Basic module that held the code. Sub Main indicates the entry point of VB.Net program. Here, we are using Class that contains both code and data. You use classes to create objects. For example, in the code, r is a Rectangle object.

An object is an instance of a class:

```
Dim r As New Rectangle()
```

A class may have members that can be accessible from outside class, if so specified. Data members are called fields and procedure members are called methods.

Shared methods or **static** methods can be invoked without creating an object of the class. Instance methods are invoked through an object of the class:

```
Shared Sub Main()
   Dim r As New Rectangle()
   r.Acceptdetails()
   r.Display()
   Console.ReadLine()
End Sub
```

Identifiers

An identifier is a name used to identify a class, variable, function, or any other user-defined item. The basic rules for naming classes in VB.Net are as follows:

- A name must begin with a letter that could be followed by a sequence of letters, digits (0 - 9) or underscore. The first character in an identifier cannot be a digit.
- It must not contain any embedded space or symbol like ? - +! @ # % ^ & * ( ) [ ] { } . ; : " ' / and \. However, an underscore (_) can be used.
- It should not be a reserved keyword.

VB.Net Keywords

The following table lists the VB.Net reserved keywords:

AddHandler	AddressOf	Alias	And	AndAlso	As	Boolean
ByRef	Byte	ByVal	Call	Case	Catch	CBool
CByte	CChar	CDate	CDec	CDbl	Char	CInt
Class	CLng	CObj	Const	Continue	CSByte	CShort
CSng	CStr	CType	CUInt	CULng	CUShort	Date

Decimal	Declare	Default	Delegate	Dim	DirectCast	Do
Double	Each	Else	ElseIf	End	End If	Enum
Erase	Error	Event	Exit	False	Finally	For
Friend	Function	Get	GetType	GetXML Namespace	Global	GoTo
Handles	If	Implements	Imports	In	Inherits	Integer
Interface	Is	IsNot	Let	Lib	Like	Long
Loop	Me	Mod	Module	MustInherit	MustOverride	MyBase
MyClass	Namespace	Narrowing	New	Next	Not	Nothing
Not Inheritable	Not Overridable	Object	Of	On	Operator	Option
Optional	Or	OrElse	Overloads	Overridable	Overrides	ParamArray
Partial	Private	Property	Protected	Public	RaiseEvent	ReadOnly
ReDim	REM	Remove Handler	Resume	Return	SByte	Select
Set	Shadows	Shared	Short	Single	Static	Step
Stop	String	Structure	Sub	SyncLock	Then	Throw
To	True	Try	TryCast	TypeOf	UInteger	While
Widening	With	WithEvents	WriteOnly	Xor		

5. Data Types

Data types refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

Data Types Available in VB.Net

VB.Net provides a wide range of data types. The following table shows all the data types available:

Data Type	Storage Allocation	Value Range
Boolean	Depends on implementing platform	True or False
Byte	1 byte	0 through 255 (unsigned)
Char	2 bytes	0 through 65535 (unsigned)
Date	8 bytes	0:00:00 (midnight) on January 1, 0001 through 11:59:59 PM on December 31, 9999
Decimal	16 bytes	0 through +/- 79,228,162,514,264,337,593,543,950,335 (+/-7.9...E+28) with no decimal point; 0 through +/- 7.9228162514264337593543950335 with 28 places to the right of the decimal
Double	8 bytes	-1.79769313486231570E+308 through -4.94065645841246544E-324, for negative values 4.94065645841246544E-324 through 1.79769313486231570E+308, for positive values

Integer	4 bytes	-2,147,483,648 through 2,147,483,647 (signed)
Long	8 bytes	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807(signed)
Object	4 bytes on 32-bit platform 8 bytes on 64-bit platform	Any type can be stored in a variable of type Object
SByte	1 byte	-128 through 127 (signed)
Short	2 bytes	-32,768 through 32,767 (signed)
Single	4 bytes	-3.4028235E+38 through -1.401298E-45 for negative values; 1.401298E-45 through 3.4028235E+38 for positive values
String	Depends on implementing platform	0 to approximately 2 billion Unicode characters
UInteger	4 bytes	0 through 4,294,967,295 (unsigned)
ULong	8 bytes	0 through 18,446,744,073,709,551,615 (unsigned)
User-Defined	Depends on implementing platform	Each member of the structure has a range determined by its data type and independent of the ranges of the other members
UShort	2 bytes	0 through 65,535 (unsigned)

Example

The following example demonstrates use of some of the types:

```
Module DataTypes
   Sub Main()
      Dim b As Byte
      Dim n As Integer
      Dim si As Single
      Dim d As Double
      Dim da As Date
      Dim c As Char
      Dim s As String
      Dim bl As Boolean
      b = 1
      n = 1234567
      si = 0.12345678901234566
      d = 0.12345678901234566
      da = Today
      c = "U"c
      s = "Me"
      If ScriptEngine = "VB" Then
         bl = True
      Else
         bl = False
      End If
      If bl Then
         'the oath taking
          Console.Write(c & " and," & s & vbCrLf)
          Console.WriteLine("declaring on the day of: {0}", da)
          Console.WriteLine("We will learn VB.Net seriously")
          Console.WriteLine("Lets see what happens to the floating point
variables:")
          Console.WriteLine("The Single: {0}, The Double: {1}", si, d)
      End If
      Console.ReadKey()
```

```
   End Sub

End Module
```

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

```
U and, Me
declaring on the day of: 12/4/2012 12:00:00 PM
We will learn VB.Net seriously
Lets see what happens to the floating point variables:
The Single:0.1234568, The Double: 0.123456789012346
```

The Type Conversion Functions in VB.Net

VB.Net provides the following in-line type conversion functions:

S.N	Functions & Description
1	**CBool(expression)** Converts the expression to Boolean data type.
2	**CByte(expression)** Converts the expression to Byte data type.
3	**CChar(expression)** Converts the expression to Char data type.
4	**CDate(expression)** Converts the expression to Date data type
5	**CDbl(expression)** Converts the expression to Double data type.
6	**CDec(expression)** Converts the expression to Decimal data type.

7	**CInt(expression)** Converts the expression to Integer data type.
8	**CLng(expression)** Converts the expression to Long data type.
9	**CObj(expression)** Converts the expression to Object type.
10	**CSByte(expression)** Converts the expression to SByte data type.
11	**CShort(expression)** Converts the expression to Short data type.
12	**CSng(expression)** Converts the expression to Single data type.
13	**CStr(expression)** Converts the expression to String data type.
14	**CUInt(expression)** Converts the expression to UInt data type.
15	**CULng(expression)** Converts the expression to ULng data type.
16	**CUShort(expression)** Converts the expression to UShort data type.

Example

The following example demonstrates some of these functions:

```
Module DataTypes
   Sub Main()
      Dim n As Integer
      Dim da As Date
      Dim bl As Boolean = True
      n = 1234567
      da = Today
      Console.WriteLine(bl)
      Console.WriteLine(CSByte(bl))
      Console.WriteLine(CStr(bl))
      Console.WriteLine(CStr(da))
      Console.WriteLine(CChar(CChar(CStr(n))))
      Console.WriteLine(CChar(CStr(da)))
      Console.ReadKey()
   End Sub
End Module
```

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

```
True
-1
True
12/4/2012
1
1
```

6. Variables

A 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 value types provided in VB.Net can be categorized as:

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

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
```

7. Constants and Enumerations

The **constants** refer to fixed values that the program may not alter during its execution. These fixed values are also called literals.

Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal. There are also enumeration constants as well.

The constants are treated just like regular variables except that their values cannot be modified after their definition.

An **enumeration** is a set of named integer constants.

Declaring Constants

In VB.Net, constants are declared using the **Const** statement. The Const statement is used at module, class, structure, procedure, or block level for use in place of literal values.

The syntax for the Const statement is:

```
[ < attributelist> ] [ accessmodifier ] [ Shadows ]
Const constantlist
```

Where,

- ***attributelist***: specifies the list of attributes applied to the constants; you can provide multiple attributes separated by commas. Optional.
- ***accessmodifier***: specifies which code can access these constants. Optional. Values can be either of the: Public, Protected, Friend, Protected Friend, or Private.
- ***Shadows***: this makes the constant hide a programming element of identical name in a base class. Optional.
- ***Constantlist***: gives the list of names of constants declared. Required.

Where, each constant name has the following syntax and parts:

```
constantname [ As datatype ] = initializer
```

- ***constantname***: specifies the name of the constant
- ***datatype***: specifies the data type of the constant

- ***initializer***: specifies the value assigned to the constant

For example,

```
' The following statements declare constants.
Const maxval As Long = 4999
Public Const message As String = "HELLO"
Private Const piValue As Double = 3.1415
```

Example

The following example demonstrates declaration and use of a constant value:

```
Module constantsNenum
   Sub Main()
      Const PI = 3.14149
      Dim radius, area As Single
      radius = 7
      area = PI * radius * radius
      Console.WriteLine("Area = " & Str(area))
      Console.ReadKey()
   End Sub
End Module
```

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

```
Area = 153.933
```

Print and Display Constants in VB.Net

VB.Net provides the following print and display constants:

Constant	Description
vbCrLf	Carriage return/linefeed character combination.
vbCr	Carriage return character.
vbLf	Linefeed character.

vbNewLine	Newline character.
vbNullChar	Null character.
vbNullString	Not the same as a zero-length string (""); used for calling external procedures.
vbObjectError	Error number. User-defined error numbers should be greater than this value. For example: Err.Raise(Number) = vbObjectError + 1000
vbTab	Tab character.
vbBack	Backspace character.

Declaring Enumerations

An enumerated type is declared using the **Enum** statement. The Enum statement declares an enumeration and defines the values of its members. The Enum statement can be used at the module, class, structure, procedure, or block level.

The syntax for the Enum statement is as follows:

```
[ < attributelist > ] [ accessmodifier ]  [ Shadows ]
Enum enumerationname [ As datatype ]
   memberlist
End Enum
```

Where,

- ***attributelist***: refers to the list of attributes applied to the variable. Optional.
- ***asscessmodifier***: specifies which code can access these enumerations. Optional. Values can be either of the: Public, Protected, Friend, or Private.
- ***Shadows***: this makes the enumeration hide a programming element of identical name in a base class. Optional.
- ***enumerationname***: name of the enumeration. Required
- ***datatype***: specifies the data type of the enumeration and all its members.

- ***memberlist***: specifies the list of member constants being declared in this statement. Required.

Each member in the memberlist has the following syntax and parts:

```
[< attribute list>] member name [ = initializer ]
```

Where,

- ***name***: specifies the name of the member. Required.
- ***initializer***: value assigned to the enumeration member. Optional.

For example,

```
Enum Colors
   red = 1
   orange = 2
   yellow = 3
   green = 4
   azure = 5
   blue = 6
   violet = 7
End Enum
```

Example

The following example demonstrates declaration and use of the Enum variable *Colors*:

```
Module constantsNenum
   Enum Colors
      red = 1
      orange = 2
      yellow = 3
      green = 4
      azure = 5
      blue = 6
      violet = 7
   End Enum
   Sub Main()
```

```
      Console.WriteLine("The Color Red is : " & Colors.red)
      Console.WriteLine("The Color Yellow is : " & Colors.yellow)
      Console.WriteLine("The Color Blue is : " & Colors.blue)
      Console.WriteLine("The Color Green is : " & Colors.green)
      Console.ReadKey()
   End Sub
End Module
```

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

```
The Color Red is: 1
The Color Yellow is: 3
The Color Blue is: 6
The Color Green is: 4
```

8. Modifiers

The modifiers are keywords added with any programming element to give some especial emphasis on how the programming element will behave or will be accessed in the program

For example, the access modifiers: Public, Private, Protected, Friend, Protected Friend, etc., indicate the access level of a programming element like a variable, constant, enumeration, or a class.

List of Available Modifiers in VB.Net

The following table provides the complete list of VB.Net modifiers:

S.N	Modifier	Description
1	Ansi	Specifies that Visual Basic should marshal all strings to American National Standards Institute (ANSI) values regardless of the name of the external procedure being declared.
2	Assembly	Specifies that an attribute at the beginning of a source file applies to the entire assembly.
3	Async	Indicates that the method or lambda expression that it modifies is asynchronous. Such methods are referred to as async methods. The caller of an async method can resume its work without waiting for the async method to finish.
4	Auto	The *charsetmodifier* part in the Declare statement supplies the character set information for marshaling strings during a call to the external procedure. It also affects how Visual Basic searches the external file for the external procedure name. The Auto modifier specifies that Visual Basic should marshal strings according to .NET Framework rules.
5	ByRef	Specifies that an argument is passed by reference, i.e., the called procedure can change the value of a

		variable underlying the argument in the calling code. It is used under the contexts of: Declare Statement Function Statement Sub Statement
6	ByVal	Specifies that an argument is passed in such a way that the called procedure or property cannot change the value of a variable underlying the argument in the calling code. It is used under the contexts of: Declare Statement Function Statement Operator Statement Property Statement Sub Statement
7	Default	Identifies a property as the default property of its class, structure, or interface.
8	Friend	Specifies that one or more declared programming elements are accessible from within the assembly that contains their declaration, not only by the component that declares them. Friend access is often the preferred level for an application's programming elements, and Friend is the default access level of an interface, a module, a class, or a structure.
9	In	It is used in generic interfaces and delegates.
10	Iterator	Specifies that a function or Get accessor is an iterator. Aniterator performs a custom iteration over a collection.
11	Key	The Key keyword enables you to specify behavior for properties of anonymous types.

12	Module	Specifies that an attribute at the beginning of a source file applies to the current assembly module. It is not same as the Module statement.
13	MustInherit	Specifies that a class can be used only as a base class and that you cannot create an object directly from it.
14	MustOverride	Specifies that a property or procedure is not implemented in this class and must be overridden in a derived class before it can be used.
15	Narrowing	Indicates that a conversion operator (CType) converts a class or structure to a type that might not be able to hold some of the possible values of the original class or structure.
16	NotInheritable	Specifies that a class cannot be used as a base class.
17	NotOverridable	Specifies that a property or procedure cannot be overridden in a derived class.
18	Optional	Specifies that a procedure argument can be omitted when the procedure is called.
19	Out	For generic type parameters, the Out keyword specifies that the type is covariant.
20	Overloads	Specifies that a property or procedure redeclares one or more existing properties or procedures with the same name.
21	Overridable	Specifies that a property or procedure can be overridden by an identically named property or procedure in a derived class.
22	Overrides	Specifies that a property or procedure overrides an identically named property or procedure inherited from a base class.

23	ParamArray	ParamArray allows you to pass an arbitrary number of arguments to the procedure. A ParamArray parameter is always declared using ByVal.
24	Partial	Indicates that a class or structure declaration is a partial definition of the class or structure.
25	Private	Specifies that one or more declared programming elements are accessible only from within their declaration context, including from within any contained types.
26	Protected	Specifies that one or more declared programming elements are accessible only from within their own class or from a derived class.
27	Public	Specifies that one or more declared programming elements have no access restrictions.
28	ReadOnly	Specifies that a variable or property can be read but not written.
29	Shadows	Specifies that a declared programming element redeclares and hides an identically named element, or set of overloaded elements, in a base class.
30	Shared	Specifies that one or more declared programming elements are associated with a class or structure at large, and not with a specific instance of the class or structure.
31	Static	Specifies that one or more declared local variables are to continue to exist and retain their latest values after termination of the procedure in which they are declared.
32	Unicode	Specifies that Visual Basic should marshal all strings to Unicode values regardless of the name of the external procedure being declared.

33	Widening	Indicates that a conversion operator (CType) converts a class or structure to a type that can hold all possible values of the original class or structure.
34	WithEvents	Specifies that one or more declared member variables refer to an instance of a class that can raise events.
35	WriteOnly	Specifies that a property can be written but not read.

9. Statements

A **statement** is a complete instruction in Visual Basic programs. It may contain keywords, operators, variables, literal values, constants, and expressions.

Statements could be categorized as:

- **Declaration statements** - these are the statements where you name a variable, constant, or procedure, and can also specify a data type.

- **Executable statements** - these are the statements, which initiate actions. These statements can call a method or function, loop or branch through blocks of code or assign values or expression to a variable or constant. In the last case, it is called an Assignment statement.

Declaration Statements

The declaration statements are used to name and define procedures, variables, properties, arrays, and constants. When you declare a programming element, you can also define its data type, access level, and scope.

The programming elements you may declare include variables, constants, enumerations, classes, structures, modules, interfaces, procedures, procedure parameters, function returns, external procedure references, operators, properties, events, and delegates.

Following are the declaration statements in VB.Net:

S.N	Statements and Description	Example

1	**Dim Statement**
	Declares and allocates storage space for one or more variables.
		`Dim number As Integer`
		`Dim quantity As Integer = 100`
		`Dim message As String = "Hello!"`

2	**Const Statement** Declares and defines one or more constants.	`Const maximum As Long = 1000`

		`Const naturalLogBase As Object` `= CDec(2.7182818284)`

3	**Enum Statement**
	Declares an enumeration and defines the values of its members.
		`Enum CoffeeMugSize`
		`Jumbo`
		`ExtraLarge`
		`Large`
		`Medium`
		`Small`
		`End Enum`

4	**Class Statement**
	Declares the name of a class and introduces the definition of the variables, properties, events, and procedures that the class comprises.
		`Class Box`
		`Public length As Double`
		`Public breadth As Double`
		`Public height As Double`
		`End Class`

5	**Structure Statement**
	Declares the name of a structure and introduces the definition of the variables, properties, events, and procedures that the structure comprises.
		`Structure Box`
		`Public length As Double`
		`Public breadth As Double`
		`Public height As Double`
		`End Structure`

6	**Module Statement**
	Declares the name of a module and introduces the definition of the variables, properties, events, and procedures that the module comprises.
		`Public Module myModule`
		`Sub Main()`
		`Dim user As String =`
		`InputBox("What is your name?")`
		`MsgBox("User name is" & user)`
		`End Sub`
		`End Module`

7	**Interface Statement**
	Declares the name of an interface and introduces the definitions of the members that the interface comprises.
		`Public Interface MyInterface`
		`Sub doSomething()`
		`End Interface`

8	**Function Statement**
	Declares the name, parameters, and code that define a Function procedure.
		`Function myFunction`
		`(ByVal n As Integer) As Double`
		`Return 5.87 * n`
		`End Function`

9	**Sub Statement**
	Declares the name, parameters, and code that define a Sub procedure.
		`Sub mySub(ByVal s As String)`
		`Return`
		`End Sub`

10	**Declare Statement**
	Declares a reference to a procedure implemented in an external file.
		`Declare Function getUserName`
		`Lib "advapi32.dll"`
		`Alias "GetUserNameA"`
		`(`

		ByVal lpBuffer As String, ByRef nSize As Integer) As Integer
11	**Operator Statement** Declares the operator symbol, operands, and code that define an operator procedure on a class or structure.	Public Shared Operator + (ByVal x As obj, ByVal y As obj) As obj Dim r As New obj ' implemention code for r = x + y Return r End Operator
12	**Property Statement** Declares the name of a property, and the property procedures used to store and retrieve the value of the property.	ReadOnly Property quote() As String Get Return quoteString End Get End Property
13	**Event Statement** Declares a user-defined event.	Public Event Finished()
14	**Delegate Statement** Used to declare a delegate.	Delegate Function MathOperator(ByVal x As Double, ByVal y As Double) As Double

Executable Statements

An executable statement performs an action. Statements calling a procedure, branching to another place in the code, looping through several statements, or evaluating an expression are executable statements. An assignment statement is a special case of an executable statement.

Example

The following example demonstrates a decision making statement:

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

      ' 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")
      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 less than 20;
value of a is : 10
```

10. Directives

The VB.Net compiler directives give instructions to the compiler to preprocess the information before actual compilation starts. All these directives begin with #, and only white-space characters may appear before a directive on a line. These directives are not statements.

VB.Net compiler does not have a separate preprocessor; however, the directives are processed as if there was one. In VB.Net, the compiler directives are used to help in conditional compilation. Unlike C and C++ directives, they are not used to create macros.

Compiler Directives in VB.Net

VB.Net provides the following set of compiler directives:

- The #Const Directive
- The #ExternalSource Directive
- The #If...Then...#Else Directives
- The #Region Directive

The #Const Directive

This directive defines conditional compiler constants. Syntax for this directive is:

```
#Const constname = expression
```

Where,

- ***constname***: specifies the name of the constant. Required.
- ***expression***: it is either a literal, or other conditional compiler constant, or a combination including any or all arithmetic or logical operators except **Is**.

For example,

```
#Const state = "WEST BENGAL"
```

Example

The following code demonstrates a hypothetical use of the directive:

```
Module mydirectives
#Const age = True
Sub Main()
   #If age Then
      Console.WriteLine("You are welcome to the Robotics Club")
   #End If
   Console.ReadKey()
End Sub
End Module
```

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

```
You are welcome to the Robotics Club
```

The #ExternalSource Directive

This directive is used for indicating a mapping between specific lines of source code and text external to the source. It is used only by the compiler and the debugger has no effect on code compilation.

This directive allows including external code from an external code file into a source code file.

Syntax for this directive is:

```
#ExternalSource( StringLiteral , IntLiteral )
    [ LogicalLine ]
#End ExternalSource
```

The parameters of #ExternalSource directive are the path of external file, line number of the first line, and the line where the error occurred.

Example

The following code demonstrates a hypothetical use of the directive:

```
Module mydirectives
    Public Class ExternalSourceTester


        Sub TestExternalSource()
```

```
        #ExternalSource("c:\vbprogs\directives.vb", 5)
            Console.WriteLine("This is External Code. ")
        #End ExternalSource

        End Sub
    End Class

    Sub Main()
        Dim t As New ExternalSourceTester()
        t.TestExternalSource()
        Console.WriteLine("In Main.")
        Console.ReadKey()

    End Sub
```

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

```
This is External Code.
In Main.
```

The #If...Then...#Else Directives

This directive conditionally compiles selected blocks of Visual Basic code.

Syntax for this directive is:

```
#If expression Then
   statements
[ #ElseIf expression Then
   [ statements ]
...
#ElseIf expression Then
   [ statements ] ]
[ #Else
   [ statements ] ]
#End If
```

For example,

```
#Const TargetOS = "Linux"
#If TargetOS = "Windows 7" Then
   ' Windows 7 specific code
#ElseIf TargetOS = "WinXP" Then
   ' Windows XP specific code
#Else
   ' Code for other OS
#End if
```

Example

The following code demonstrates a hypothetical use of the directive:

```
Module mydirectives
#Const classCode = 8

   Sub Main()
   #If classCode = 7 Then
        Console.WriteLine("Exam Questions for Class VII")
   #ElseIf classCode = 8 Then
        Console.WriteLine("Exam Questions for Class VIII")
   #Else
        Console.WriteLine("Exam Questions for Higher Classes")
   #End If
        Console.ReadKey()

    End Sub
End Module
```

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

```
Exam Questions for Class VIII
```

The #Region Directive

This directive helps in collapsing and hiding sections of code in Visual Basic files.

Syntax for this directive is:

```
#Region "identifier_string"
#End Region
```

For example,

```
#Region "StatsFunctions"
    ' Insert code for the Statistical functions here.
#End Region
```

11. 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:

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 first	A - B will give -5
*	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

Example

Try the following example to understand all the arithmetic operators available in VB.Net:

```
Module operators
   Sub Main()
      Dim a As Integer = 21
      Dim b As Integer = 10
      Dim p As Integer = 2
      Dim c As Integer
      Dim d As Single
      c = a + b
      Console.WriteLine("Line 1 - Value of c is {0}", c)
      c = a - b
      Console.WriteLine("Line 2 - Value of c is {0}", c)
      c = a * b
      Console.WriteLine("Line 3 - Value of c is {0}", c)
      d = a / b
      Console.WriteLine("Line 4 - Value of d is {0}", d)
      c = a \ b
      Console.WriteLine("Line 5 - Value of c is {0}", c)
      c = a Mod b
      Console.WriteLine("Line 6 - Value of c is {0}", c)
      c = b ^ p
      Console.WriteLine("Line 7 - Value of c is {0}", c)
      Console.ReadLine()
   End Sub
End Module
```

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

```
Line 1 - Value of c is 31
Line 2 - Value of c is 11
Line 3 - Value of c is 210
Line 4 - Value of d is 2.1
Line 5 - Value of c is 2
```

```
Line 6 - Value of c is 1
Line 7 - Value of c is 100
```

Comparison Operators

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

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.

Try the following example to understand all the relational operators available in VB.Net:

```
Module operators
   Sub Main()
      Dim a As Integer = 21
      Dim b As Integer = 10
```

```
      If (a = b) Then
          Console.WriteLine("Line 1 - a is equal to b")
      Else
          Console.WriteLine("Line 1 - a is not equal to b")
      End If
      If (a < b) Then
          Console.WriteLine("Line 2 - a is less than b")
      Else
          Console.WriteLine("Line 2 - a is not less than b")
      End If
      If (a > b) Then
          Console.WriteLine("Line 3 - a is greater than b")
      Else
          Console.WriteLine("Line 3 - a is not greater than b")
      End If
      ' Lets change value of a and b
      a = 5
      b = 20
      If (a <= b) Then
          Console.WriteLine("Line 4 - a is either less than or equal to
b")
      End If
      If (b >= a) Then
          Console.WriteLine("Line 5 - b is either greater than  or equal
to b")
      End If
      Console.ReadLine()
   End Sub
End Module
```

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

```
Line 1 - a is not equal to b
Line 2 - a is not less than b
Line 3 - a is greater than b
Line 4 - a is either less than or equal to b
```

```
Line 5 - b is either greater than or equal to b
```

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.

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:

Operator	Description	Example
And	It is the logical as well as bitwise AND operator. If both the operands are true, then condition becomes true. This operator does not perform short-circuiting, i.e., it evaluates both the expressions.	(A And B) is 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.	

Example

Try the following example to understand all the logical/bitwise operators available in VB.Net:

```
Module logicalOp

    Sub Main()
        Dim a As Boolean = True
        Dim b As Boolean = True
        Dim c As Integer = 5
```

```
        Dim d As Integer = 20
        'logical And, Or and Xor Checking
        If (a And b) Then
            Console.WriteLine("Line 1 - Condition is true")
        End If
        If (a Or b) Then
            Console.WriteLine("Line 2 - Condition is true")
        End If
        If (a Xor b) Then
            Console.WriteLine("Line 3 - Condition is true")
        End If
        'bitwise And, Or and Xor Checking
        If (c And d) Then
            Console.WriteLine("Line 4 - Condition is true")
        End If
        If (c Or d) Then
            Console.WriteLine("Line 5 - Condition is true")
        End If
        If (c Or d) Then
            Console.WriteLine("Line 6 - Condition is true")
        End If
        'Only logical operators
        If (a AndAlso b) Then
            Console.WriteLine("Line 7 - Condition is true")
        End If
        If (a OrElse b) Then
            Console.WriteLine("Line 8 - Condition is true")
        End If

        ' lets change the value of  a and b
        a = False
        b = True
        If (a And b) Then
            Console.WriteLine("Line 9 - Condition is true")
```

```
        Else
            Console.WriteLine("Line 9 - Condition is not true")
        End If
        If (Not (a And b)) Then
            Console.WriteLine("Line 10 - Condition is true")
        End If
        Console.ReadLine()
    End Sub
End Module
```

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

```
Line 1 - Condition is true
Line 2 - Condition is true
Line 3 - Condition is true
Line 4 - Condition is true
Line 5 - Condition is true
Line 6 - Condition is true
Line 7 - Condition is true
Line 8 - Condition is true
Line 9 - Condition is not true
Line 10 - Condition 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 operations. The truth tables for &, |, and ^ are as follows:

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

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	A << 2 will give 240, which is 1111 0000

	number of bits specified by the right operand.	
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15, which is 0000 1111

Example

Try the following example to understand all the bitwise operators available in VB.Net:

```
Module BitwiseOp
   Sub Main()
      Dim a As Integer = 60        ' 60 = 0011 1100
      Dim b As Integer = 13        ' 13 = 0000 1101
      Dim c As Integer = 0
      c = a And b       ' 12 = 0000 1100
      Console.WriteLine("Line 1 - Value of c is {0}", c)
      c = a Or b       ' 61 = 0011 1101
      Console.WriteLine("Line 2 - Value of c is {0}", c)
      c = a Xor b       ' 49 = 0011 0001
      Console.WriteLine("Line 3 - Value of c is {0}", c)
      c = Not a          ' -61 = 1100 0011
      Console.WriteLine("Line 4 - Value of c is {0}", c)
      c = a << 2     ' 240 = 1111 0000
      Console.WriteLine("Line 5 - Value of c is {0}", c)
      c = a >> 2    ' 15 = 0000 1111
      Console.WriteLine("Line 6 - Value of c is {0}", c)
      Console.ReadLine()
   End Sub
End Module
```

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

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
```

```
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15
```

Assignment Operators

There are following assignment operators supported by VB.Net:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + 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

Example

Try the following example to understand all the assignment operators available in VB.Net:

```
Module assignment
   Sub Main()
      Dim a As Integer = 21
      Dim pow As Integer = 2
      Dim str1 As String = "Hello! "
      Dim str2 As String = "VB Programmers"
      Dim c As Integer
      c = a
      Console.WriteLine("Line 1 - =  Operator Example, _
       Value of c = {0}", c)
      c += a
      Console.WriteLine("Line 2 - +=  Operator Example, _
       Value of c = {0}", c)
      c -= a
      Console.WriteLine("Line 3 - -=  Operator Example, _
       Value of c = {0}", c)
      c *= a
      Console.WriteLine("Line 4 - *=  Operator Example, _
       Value of c = {0}", c)
      c /= a
      Console.WriteLine("Line 5 - /=  Operator Example, _
```

```
        Value of c = {0}", c)
       c = 20
       c ^= pow
       Console.WriteLine("Line 6 - ^=  Operator Example, _
        Value of c = {0}", c)
       c <<= 2
       Console.WriteLine("Line 7 - <<=  Operator Example,_
        Value of c = {0}", c)
       c >>= 2
       Console.WriteLine("Line 8 - >>=  Operator Example,_
        Value of c = {0}", c)
       str1 &= str2
       Console.WriteLine("Line 9 - &=  Operator Example,_
        Value of str1 = {0}", str1)
       Console.ReadLine()
   End Sub
End Module
```

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

```
Line 1 - =  Operator Example, Value of c = 21
Line 2 - += Operator Example, Value of c = 42
Line 3 - -= Operator Example, Value of c = 21
Line 4 - *= Operator Example, Value of c = 441
Line 5 - /= Operator Example, Value of c = 21
Line 6 - ^= Operator Example, Value of c = 400
Line 7 - <<= Operator Example, Value of c = 1600
Line 8 - >>= Operator Example, Value of c = 400
Line 9 - &= Operator Example, Value of str1 = Hello! VB Programmers
```

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 asynchronous method or lambda expression to suspend execution of the method until the awaited task completes.	`Dim result As res`
		`= 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 two arguments.	`Dim num = 5`
		`Console.WriteLine(If(num >= 0,`
		`"Positive", "Negative"))`

Example

The following example demonstrates some of these operators:

```
Module assignment
   Sub Main()
```

```
      Dim a As Integer = 21
      Console.WriteLine(GetType(Integer).ToString())
      Console.WriteLine(GetType(Double).ToString())
      Console.WriteLine(GetType(String).ToString())
      Dim multiplywith5 = Function(num As Integer) num * 5
      Console.WriteLine(multiplywith5(5))
      Console.WriteLine(If(a >= 0, "Positive", "Negative"))
      Console.ReadLine()
   End Sub
End Module
```

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

```
System.Int32
System.Double
System.String
25
Positive
```

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.

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)	Lowest

Example

The following example demonstrates operator precedence in a simple way:

```
Module assignment
   Sub Main()
      Dim a As Integer = 20
      Dim b As Integer = 10
      Dim c As Integer = 15
      Dim d As Integer = 5
      Dim e As Integer
      e = (a + b) * c / d      ' ( 30 * 15 ) / 5
      Console.WriteLine("Value of (a + b) * c / d is : {0}", e)
      e = ((a + b) * c) / d    ' (30 * 15 ) / 5
```

```
      Console.WriteLine("Value of ((a + b) * c) / d is  : {0}", e)
      e = (a + b) * (c / d)   ' (30) * (15/5)
      Console.WriteLine("Value of (a + b) * (c / d) is  : {0}", e)
      e = a + (b * c) / d     '  20 + (150/5)
      Console.WriteLine("Value of a + (b * c) / d is  : {0}", e)
      Console.ReadLine()
   End Sub
End Module
```

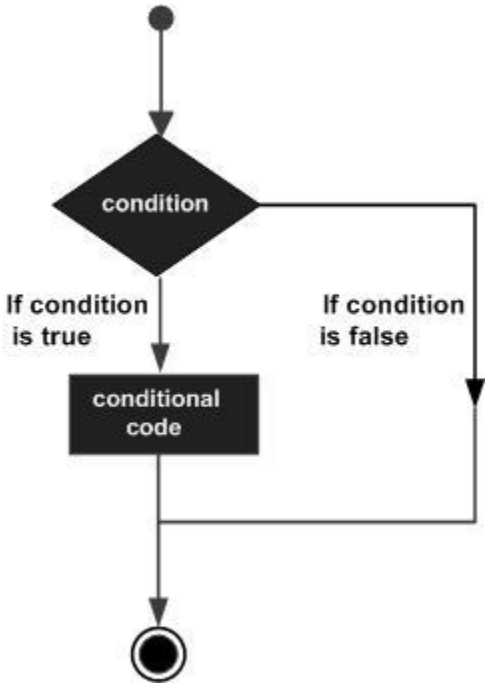
When the above code is compiled and executed, it produces the following result:

```
Value of (a + b) * c / d is : 90
Value of ((a + b) * c) / d is  : 90
Value of (a + b) * (c / d) is  : 90
Value of a + (b * c) / d is  : 50
```

12. Decision Making

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages:



VB.Net provides the following types of decision making statements. Click the following links to check their details.

Statement	Description
If ... Then statement	An **If...Then statement** consists of a boolean expression followed by one or more statements.
If...Then...Else statement	An **If...Then statement** can be followed by an optional **Else statement**, which executes when the boolean expression is false.

nested If statements	You can use one **If** or **Else if** statement inside another **If** or **Else if** statement(s).
Select Case statement	A **Select Case** statement allows a variable to be tested for equality against a list of values.
nested Select Case statements	You can use one **select case** statement inside another **select case** statement(s).

If...Then Statement

It is the simplest form of control statement, frequently used in decision making and changing the control flow of the program execution. Syntax for if-then statement is:

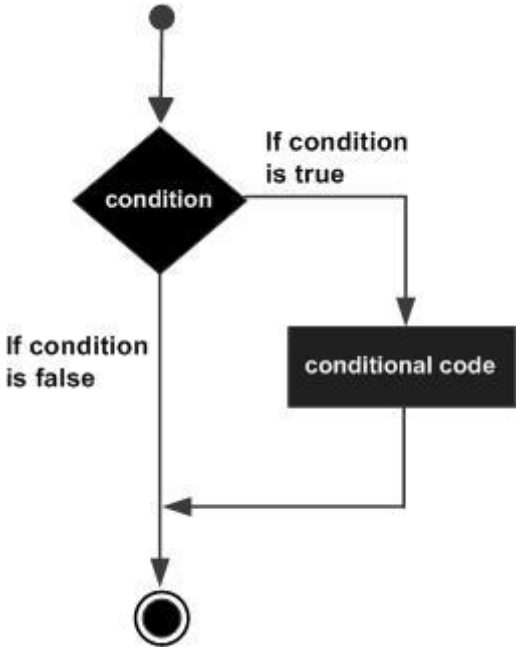
```
If condition Then
[Statement(s)]
End If
```

Where, *condition* is a Boolean or relational condition and Statement(s) is a simple or compound statement. Example of an If-Then statement is:

```
If (a <= 20) Then
   c= c+1
End If
```

If the condition evaluates to true, then the block of code inside the If statement will be executed. If condition evaluates to false, then the first set of code after the end of the If statement (after the closing End If) will be executed.

Flow Diagram



Example

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

        ' 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")
        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 less than 20
value of a is : 10
```

If...Then...Else Statement

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

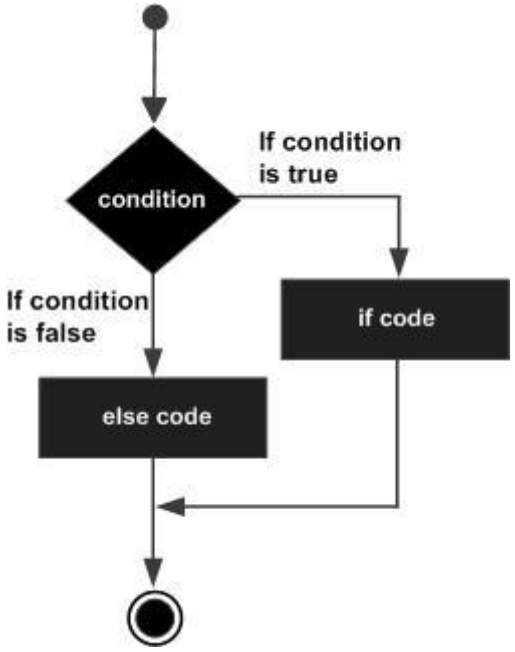
Syntax

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
```

Nested If Statements

It is always legal in VB.Net to nest If-Then-Else statements, which means you can use one If or ElseIf statement inside another If ElseIf statement(s).

Syntax

The syntax for a nested If statement is as follows:

```
If( boolean_expression 1)Then
   'Executes when the boolean expression 1 is true
   If(boolean_expression 2)Then
         'Executes when the boolean expression 2 is true
   End If
End If
```

You can nest ElseIf...Else in the similar way as you have nested If statement.

Example

```
Module decisions
   Sub Main()
      'local variable definition
      Dim a As Integer = 100
      Dim b As Integer = 200
      ' check the boolean condition
      If (a = 100) Then
          ' if condition is true then check the following
          If (b = 200) Then
              ' if condition is true then print the following
              Console.WriteLine("Value of a is 100 and b is 200")
          End If
      End If
      Console.WriteLine("Exact value of a is : {0}", a)
      Console.WriteLine("Exact value of b is : {0}", b)
      Console.ReadLine()
   End Sub
End Module
```

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

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

Select Case Statement

A **Select Case** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each select case.

Syntax

The syntax for a Select Case statement in VB.Net is as follows:

```
Select [ Case ] expression
    [ Case expressionlist
        [ statements ] ]
    [ Case Else
        [ elsestatements ] ]
End Select
```

Where,

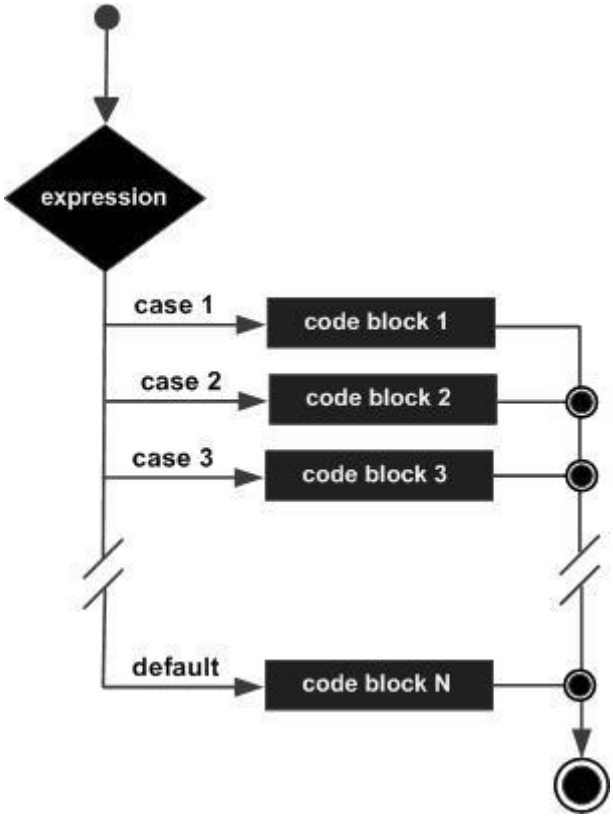
- ***expression***: is an expression that must evaluate to any of the elementary data type in VB.Net, i.e., Boolean, Byte, Char, Date, Double, Decimal, Integer, Long, Object, SByte, Short, Single, String, UInteger, ULong, and UShort.

- ***expressionlist***: List of expression clauses representing match values for *expression*. Multiple expression clauses are separated by commas.

- ***statements***: statements following Case that run if the select expression matches any clause in *expressionlist*.

- ***elsestatements***: statements following Case Else that run if the select expression does not match any clause in the *expressionlist* of any of the Case statements.

Flow Diagram



Example

```
Module decisions
   Sub Main()
      'local variable definition
      Dim grade As Char
      grade = "B"
      Select grade
          Case "A"
              Console.WriteLine("Excellent!")
          Case "B", "C"
              Console.WriteLine("Well done")
          Case "D"
              Console.WriteLine("You passed")
          Case "F"
              Console.WriteLine("Better try again")
          Case Else
```

```
            Console.WriteLine("Invalid grade")
      End Select

      Console.WriteLine("Your grade is  {0}", grade)
      Console.ReadLine()
   End Sub
End Module
```

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

```
Well done
Your grade is B
```

Nested Select Case Statement

It is possible to have a select statement as part of the statement sequence of an outer select statement. Even if the case constants of the inner and outer select contain common values, no conflicts will arise.

Example

```
Module decisions
   Sub Main()
      'local variable definition
      Dim a As Integer = 100
      Dim b As Integer = 200
      Select a
          Case 100
              Console.WriteLine("This is part of outer case ")
              Select Case b
                  Case 200
                      Console.WriteLine("This is part of inner case ")
              End Select
      End Select
      Console.WriteLine("Exact value of a is : {0}", a)
      Console.WriteLine("Exact value of b is : {0}", b)
      Console.ReadLine()
   End Sub
```

```
End Module
```

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

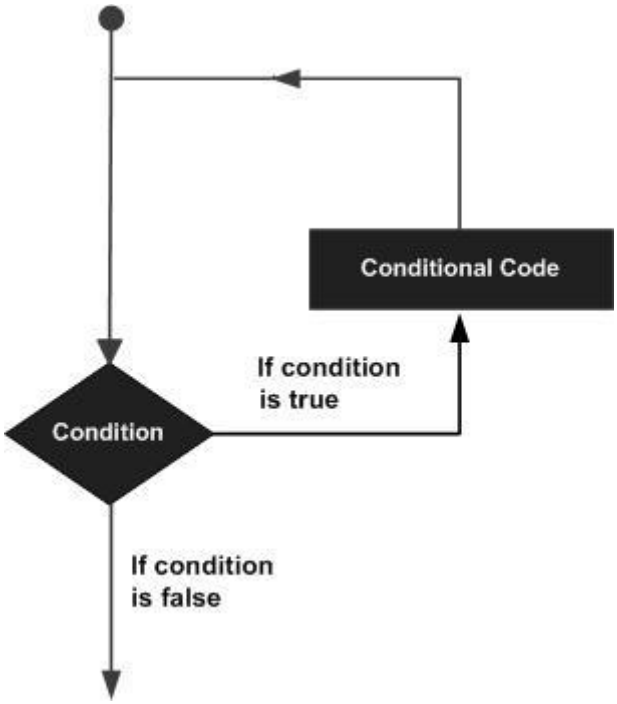
```
This is part of outer case
This is part of inner case
Exact value of a is : 100
Exact value of b is : 200
```

13. Loops

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages:



VB.Net provides following types of loops to handle looping requirements. Click the following links to check their details.

Loop Type	Description
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.

For...Next	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.
For Each...Next	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.
While... End While	It executes a series of statements as long as a given condition is True.
With... End With	It is not exactly a looping construct. It executes a series of statements that repeatedly refer to a single object or structure.
Nested loops	You can use one or more loops inside any another While, For or Do loop.

Do Loop

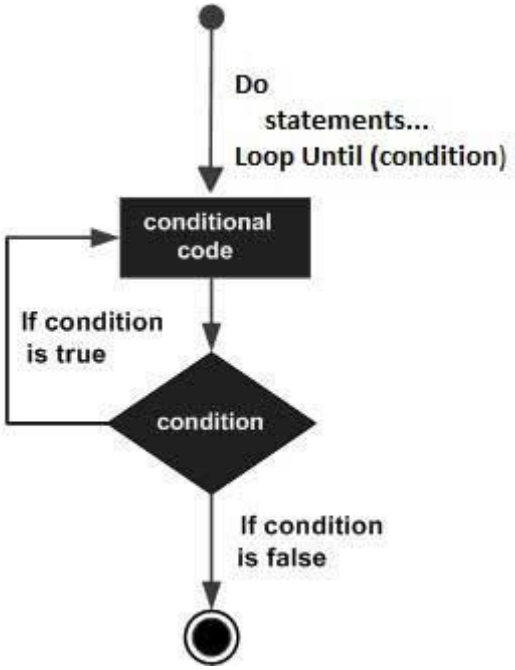
It repeats the enclosed block of statements while a Boolean condition is True or until the condition becomes True. It could be terminated at any time with the Exit Do statement.

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
```

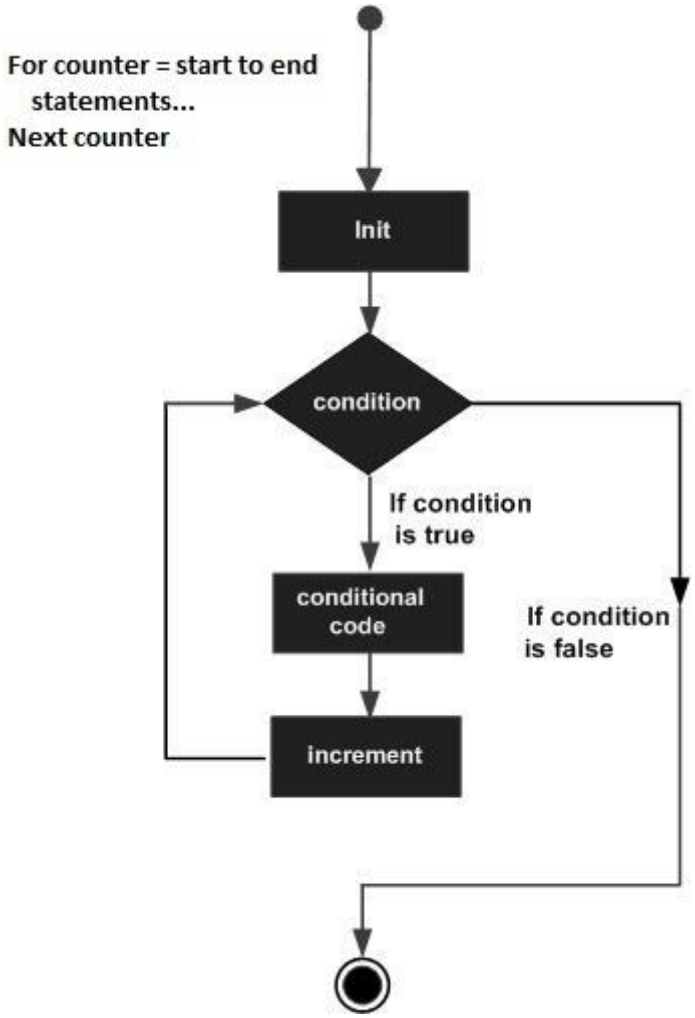
For...Next Loop

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
```

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
```

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:

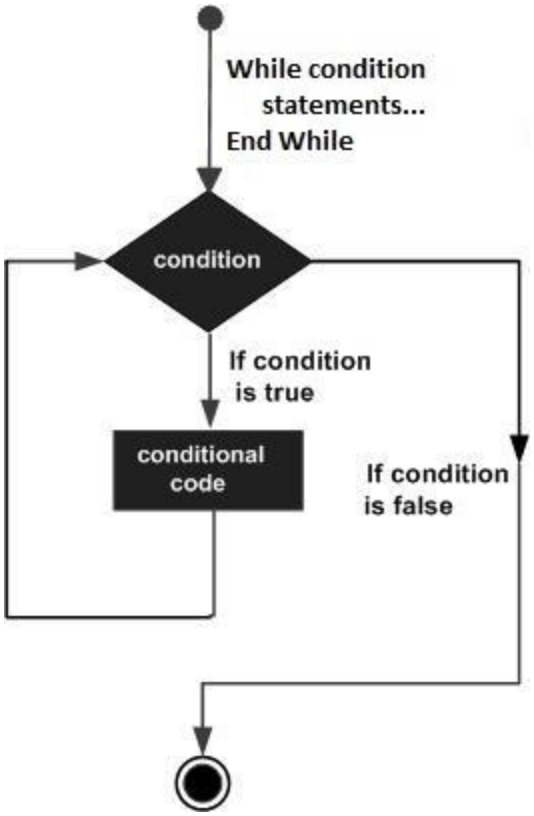
```
While condition
```

```
    [ 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
```

With... End With Statement

It is not exactly a looping construct. It executes a series of statements that repeatedly refers to a single object or structure.

The syntax for this loop construct is:

```
With object
    [ statements ]
End With
```

Example

```
Module loops
   Public Class Book
      Public Property Name As String
      Public Property Author As String
      Public Property Subject As String
   End Class
   Sub Main()
      Dim aBook As New Book
      With aBook
          .Name = "VB.Net Programming"
          .Author = "Zara Ali"
          .Subject = "Information Technology"
      End With
```

```
      With aBook
          Console.WriteLine(.Name)
          Console.WriteLine(.Author)
          Console.WriteLine(.Subject)
      End With
      Console.ReadLine()
   End Sub
End Module
```

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

```
VB.Net Programming
Zara Ali
Information Technology
```

Nested Loops

VB.Net allows using one loop inside another loop. Following section shows few examples to illustrate the concept.

Syntax

The syntax for a **nested For loop** statement in VB.Net is as follows:

```
For counter1 [ As datatype1 ] = start1 To end1 [ Step step1 ]
    For counter2 [ As datatype2 ] = start2 To end2 [ Step step2 ]
    ...
    Next [ counter2 ]
Next [ counter 1]
```

The syntax for a **nested While loop** statement in VB.Net is as follows:

```
While condition1
    While condition2
    ...
    End While
End While
```

The syntax for a **nested Do...While loop** statement in VB.Net is as follows:

```
Do { While | Until } condition1
    Do { While | Until } condition2
    ...
    Loop
Loop
```

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example, a for loop can be inside a while loop or vice versa.

Example

The following program uses a nested for loop to find the prime numbers from 2 to 100:

```
Module loops
   Sub Main()
      ' local variable definition
      Dim i, j As Integer
      For i = 2 To 100
          For j = 2 To i
              ' if factor found, not prime
              If ((i Mod j) = 0) Then
                  Exit For
              End If
          Next j
          If (j > (i \ j)) Then
              Console.WriteLine("{0} is prime", i)
          End If
      Next i
      Console.ReadLine()
   End Sub
End Module
```

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

```
2 is prime
3 is prime
5 is prime
```

```
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
```

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

VB.Net provides the following control statements. Click the following links to check their details.

Control Statement	Description

Exit statement	Terminates the **loop** or **select case** statement and transfers execution to the statement immediately following the loop or select case.
Continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
GoTo statement	Transfers control to the labeled statement. Though it is not advised to use GoTo statement in your program.

Exit Statement

The Exit statement transfers the control from a procedure or block immediately to the statement following the procedure call or the block definition. It terminates the loop, procedure, try block or the select block from where it is called.

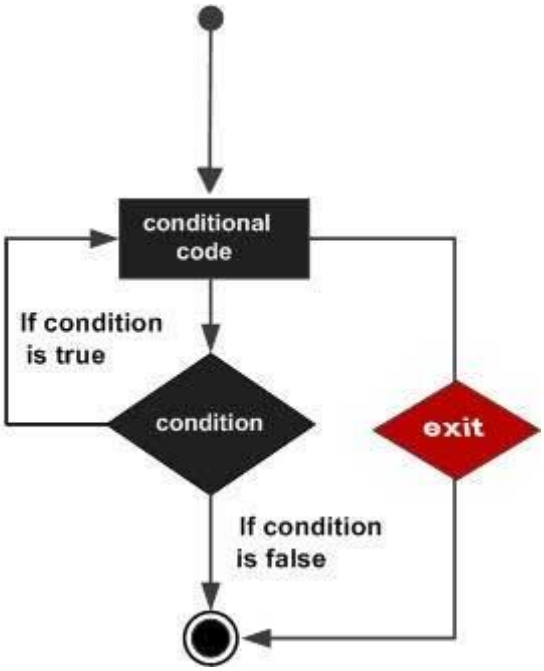
If you are using nested loops (i.e., one loop inside another loop), the Exit statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax

The syntax for the Exit statement is:

```
Exit { Do | For | Function | Property | Select | Sub | Try | While }
```

Flow Diagram



Example

```
Module loops
   Sub Main()
      ' local variable definition
      Dim a As Integer = 10
      ' while loop execution '
      While (a < 20)
          Console.WriteLine("value of a: {0}", a)
          a = a + 1
          If (a > 15) Then
              'terminate the loop using exit statement
              Exit While
          End If
      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
```

Continue Statement

The Continue statement causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. It works somewhat like the Exit statement. Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code in between.

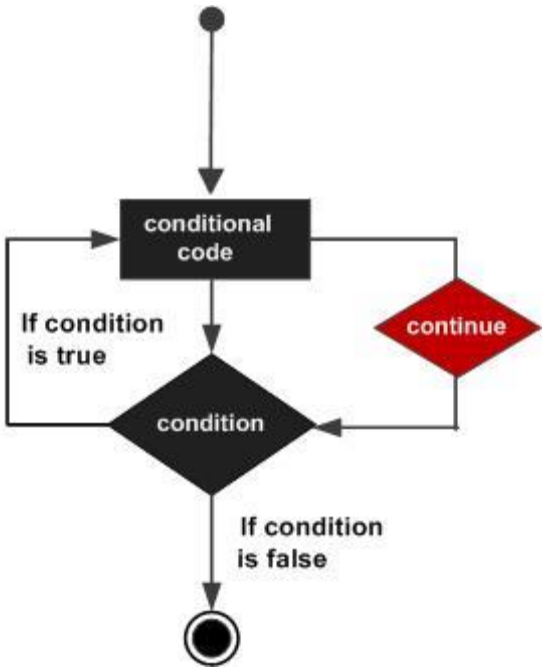
For the For...Next loop, Continue statement causes the conditional test and increment portions of the loop to execute. For the While and Do...While loops, continue statement causes the program control to pass to the conditional tests.

Syntax

The syntax for a Continue statement is as follows:

```
Continue { Do | For | While }
```

Flow Diagram



Example

```
Module loops
   Sub Main()
      ' local variable definition
      Dim a As Integer = 10
      Do
          If (a = 15) Then
              ' skip the iteration '
              a = a + 1
              Continue Do
          End If
          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: 16
value of a: 17
value of a: 18
value of a: 19
```

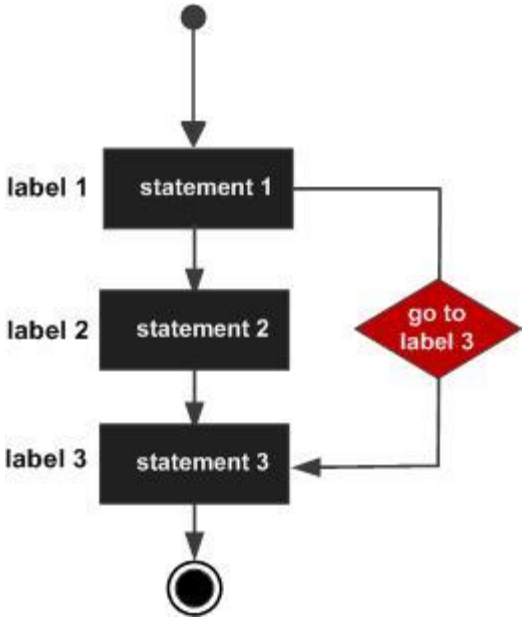
GoTo Statement

The GoTo statement transfers control unconditionally to a specified line in a procedure.

The syntax for the GoTo statement is:

```
GoTo label
```

Flow Diagram



Example

```
Module loops
   Sub Main()
      ' local variable definition
      Dim a As Integer = 10
Line1:
      Do
          If (a = 15) Then
              ' skip the iteration '
              a = a + 1
              GoTo Line1
          End If
          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: 16
value of a: 17
value of a: 18
value of a: 19
```

14. Strings

In VB.Net, you can use strings as array of characters, however, more common practice is to use the String keyword to declare a string variable. The string keyword is an alias for the **System.String** class.

Creating a String Objec

You can create string object using one of the following methods:

- By assigning a string literal to a String variable
- By using a String class constructor
- By using the string concatenation operator (+)
- By retrieving a property or calling a method that returns a string
- By calling a formatting method to convert a value or object to its string representation

The following example demonstrates this:

```
Module strings
   Sub Main()
      Dim fname, lname, fullname, greetings As String
      fname = "Rowan"
      lname = "Atkinson"
      fullname = fname + " " + lname
      Console.WriteLine("Full Name: {0}", fullname)

     'by using string constructor
      Dim letters As Char() = {"H", "e", "l", "l", "o"}
      greetings = New String(letters)
      Console.WriteLine("Greetings: {0}", greetings)

      'methods returning String
      Dim sarray() As String = {"Hello", "From", "Tutorials", "Point"}
      Dim message As String = String.Join(" ", sarray)
```

```
      Console.WriteLine("Message: {0}", message)


      'formatting method to convert a value
      Dim waiting As DateTime = New DateTime(2012, 12, 12, 17, 58, 1)
      Dim chat As String = String.Format("Message sent at {0:t} on
{0:D}", waiting)
      Console.WriteLine("Message: {0}", chat)
      Console.ReadLine()
   End Sub
End Module
```

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

```
Full Name: Rowan Atkinson
Greetings: Hello
Message: Hello From Tutorials Point
Message: Message sent at 5:58 PM on Wednesday, December 12, 2012
```

Properties of the String Class

The String class has the following two properties:

S.N	Property Name & Description
1	**Chars** Gets the *Char* object at a specified position in the current *String* object.
2	**Length** Gets the number of characters in the current String object.

Methods of the String Class

The String class has numerous methods that help you in working with the string objects. The following table provides some of the most commonly used methods: