UNIT 2: Conditions, Loops and Windows Forms

If…Else Statements

It is used to execute one or more statement conditionally. You can use single line syntax or multi line syntax.

Syntax : 1) If <condition> then <statement>

Syntax : 2)
```
If <condition> then
        <statement>
End if
```

Syntax : 3)
```
If condition Then
        [statements]
[ElseIf condition-n Then
        [elseifstatements] …]
[Else
        [elsestatements]]
End If
```

If condition is true, VB .NET executes all the statements following the then keyword.

- You can use either single line or multiple-line syntax to execute just 1 statement.
- You can use comparison operators in the condition to generate a logical result that's true or false.
- If *condition* is **True**, the statements immediately following the **Then** keyword in the body of the **If** statement will be executed, and the **If** statement will terminate before the code in any **ElseIf** or **Else** statement is executed.
- If *condition* is **False**, the following **ElseIf** statements are evaluated, if there are any; this statement lets you test additional conditions, and if any are **True**, the corresponding code (*elseifstatements* above) is executed and the **If** statement terminates. If there are no **ElseIf** statements, or if none of their conditions are **True**, the code in the **Else** statement (*elsestatements* above), if there is one, is executed automatically.

Example
```
Dim a, b, c As Integer
    a = InputBox("Enter A: ")
    b = InputBox("Enter B: ")
    c = InputBox("Enter C: ")
```

```
If a>b And a>c Then
   MsgBox("A is Maximum")
ElseIf b>a And b>c Then
   MsgBox("B is Maximum")
Else
   MsgBox("C is Maximum")
End If
```

Select Case statement:

Executes one of several groups of statements depending on the value of an expression.

Syntax:

```
Select Case testexpression
        [Case expressionlist-n
              [statements-n]]…
       [Case Else
              [elsestatements]]
End Select
```

- **Testexpression:** Required It may be any numeric or string expression.
- **Expression list_n:** it may be
  - expression or
  - expression To expression
  - Is expression operator expression
- **statements-n :** one or more statements executed if test expression doesn't match any of the case clause.
- If test expression matches any case expression list expression then statement following that clause is executed upto the next case clause or for the last clause or up to End select.
- After performing select……..case, control executes the next statement after the End select.
- If test expression matches more than 1 expression then only the statements which follow the 1st match are executed.
- The Case Else clause is used to indicated the else statements to be executed if no match is found between test expression and expression list.

- Select case statement can be nested. Each nested select case statement must have a matching End select statement.

Example: Dim no1, no2 as integer

```
              No2 =20
              No1 = val ( Inputbox ("Enter the value of no1:")
              Select case no1
                            Case 1:
                                      Msgbox("The value of no1 is:") & no1
                            Case 2:
                                      Msgbox("The value of no1 is:") & no1
                            Case 3 To 15:
                                      Msgbox("The value of no1 is between 3 to 5")
                            Case  Is > no2:
                                      Msgbox  no1 & "is grater than" & no2
                            Case else:
                                      Msgbox ("nothing");
          End select.
```

With **Statement**

The **With** statement is not a loop, but it can be as useful as a loop. User can use the **With** statement to execute statements using a particular object.

Syntax: With *object*

```
                    [statements]
      End With
```

Example:

```
With TextBox1
   .Height = 1000
   .Width = 3000
   .Text = "Welcome to Visual Basic"
End With
```

***Do* Loop:**

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

Syntax:
```
Do [{While | Until} condition ]
   [statements]
   [Exit Do]
   [statements]
Loop
```
OR

Syntax:
```
Do
   [statements]
   [Exit Do]
   [statements]
Loop [{While | Until} condition]
```

- A loop can be executed either while the condition is true or until the condition becomes true.
- When VB executes the loops, if first evaluates the condition if condition is false, the Do… While or Do…Until loop is skipped & execute the statement followed by the loop.
- The Do…Loop can execute any number of times as long as condition is true.
- If the condition is initially false, the statements may never execute.
- Exit Do statement terminates the loop.

Example:
```
i=5                          i=1
Do While i>0                 Do until i<=5
        Msgbox  i                    me.point i
        i=i-1                        i=i+1
loop                         loop
```

For Loop

The **For** loop is probably the most popular of all Visual Basic loops. The Do loop doesn't need a *loop index*, but the **For** loop does; a loop index counts the number of loop iterations as the loop executes.

Syntax:
```
For index = start To end [Step step]
    [statements]
    [Exit For]
    [statements]
Next [index]
```

- The keywords in square brackets are optional.
- The keyword counter, start, end, increment all one of numeric type.
- **Index:** Required Numeric variable. The variable can't be a Boolean or an array element.
- **End:** Required find value of counter.
- **Step:** Optional Counter is changed each time through the loop. If not specified then step defaults to one.
- **Statement:** Optional one or more statement between for and Next that the executed the specified number of times.
- **Exit For:** to terminate the loop without executing the statements after exit for keyword.
- The loop is executed as many times as required for the counter to reach the end value.
- The increment argument can be either positive or negative. If start is greater than end, the value of increment must be negative. If not the loop's body can't be executed.

Example:
```
for i=0 To 10                    for i=10 To 1 step -1
     msgbox i                           msgbox i
   Next                          Next.
```

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

```
For i = 0 To n Step 2
   MsgBox(i)
Next
```

For Each…Next Loop

You use the **For Each…Next** loop to loop over elements in an array or a Visual Basic collection. This loop automatically loops over all the elements in the array or collection.

Syntax:

```
For Each element In group
    [statements]
    [Exit For]
    [statements]
Next [element]
```

Example:

```
Dim intIDArray(3), intArrayItem As Integer
intIDArray(0) = 0
intIDArray(1) = 1
intIDArray(2) = 2
intIDArray(3) = 3
For Each intArrayItem In intIDArray
   System.Console.WriteLine(intArrayItem)
Next intArrayItem
```

While Loop

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

Syntax:

```
While condition
   [statements]
End While
```

- If condition is true, all the statements are executed & when the wend statement is reached, control is returned to the While statement which evaluate condition again.
- If condition is still true, the process is repeated.
- If condition is false, the program executes the statement following Wend statement.

Example:

```
i=5
While i>=0
        total=total+i
        i=i-1
End while
```

➔ <u>Working with Procedure</u> :--

- ❖ Dividing your code into procedure allows you to break it up into more modular units. As your program become longer that's invaluable as it stops everything from becoming too cultured

In visual basic , all executable code must be in procedure

There are two typed of procedure
1) sub procedure (sub routine)
2) Functions

<u>Procedure:</u>

- ➢ A procedure is a set of one or more program statements that can be run or call by referring to the procedure name.
- ➢ We can divide the big program into small procedures.
- ➢ Due to procedure the application is easier to debug and easier to find error.
- ➢ The main advantage of procedure is its reusability.
- ➢ Procedures are categorized into two categories:
  1) In-built procedure
  2) User-defined procedure
- ➢ There are 4 types of procedure in VB.Net.
  1) Sub procedure: It does not return value.
  2) Event handling procedure: These are Sub procedures that execute in response to an event triggered by user action.
  3) Function procedures: they return a value.
  4) Property procedures: used in object oriented concept.

Sub Procedure: Also known as Sub Routine.

- Sub procedure may or may not have arguments. Arguments are not compulsory.
- A sub procedure does not return the value.
- By Val is by default argument type.
- To declare procedure Sub keyword is used.

Sub Statement: Declares the name, parameters, and code that define a **Sub** procedure.

Syntax: [**Private** | **Public**] **Sub** name [**(**parameterlist**)**]

```
[statements]
    [Exit Sub]
    [statements]
End Sub
```

- **name :** It specifies name of the procedure.
- **parameterlist :** It is optional. It specifies the list of local variable names representing the parameters of this procedure. Specifies the parameters a procedure expects when it is called.
- Multiple parameters are separated by commas. The following is the syntax for one parameter. [optional] [**ByVal** | **ByRef**] varname[**( )**] [**As** type] [= defaultvalue]
- **Optional:** Optional. Specifies that this parameter is not required when the procedure is called.
- **ByVal** : It is Optional. Specifies that the procedure cannot replace or reassign the variable element underlying the corresponding argument in the calling code.
- **ByRef** : It is Optional. Specifies that the procedure can modify the underlying variable element in the calling code the same way the calling code itself can.
- **parametername :** It is Required. Name of the local variable representing the parameter.
- **parametertype :** Required if **Option Strict** is **On**. Data type of the local variable representing the parameter.
- **defaultvalue :** Required for **Optional** parameters. Any constant or constant expression that evaluates to the data type of the parameter. If the type is **Object**, or a class, interface, array, or structure, the default value can only be **Nothing**.

Example:

```
   Dim ans As Double
   Private Sub sMsg()
      MsgBox("Hello")
   End Sub
   Private Sub sum1(ByRef n1 As Integer)
      MsgBox(n1 + 20)
   End Sub
Private Sub butOk_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles butOk.Click
      Call sMsg()
      Call sum1(Val(Me.txtN1.Text))
End Sub
```

<u>Working with Modules</u>:-

- ❖ Visual Basic uses the term module level to apply equally to modules,classes and structures you can declare elements at this level by placing the declaration statement out side of any procedure or block within the module class or structure .
- ❖ When you make a declaration at the module level the accessibility you choose determines the scope. The namespace that contains the Module , class or Structure also affects the scope
- ❖ Elements for which you declare private accessibility are available for reference to every procedure in that module . but , not to any code in a different module

The Dim statement at Module level defaults to Private accessibility . o , it is equivalent to using the private statement. However , you can make the scope and accessibility more obvious by using private.

<u>Class</u> :--

- A class means collection of methods/functions. Method/function accepts parameters, process set of codes which you have written in the module/function and returns the output to the caller. Collection of class is called Class Library. When you complie the Class Library it becomes a DLL.

- A class is simply an abstract model used to define new data types. A class may contain any combination of encapsulated data (fields or member variables), operations that can be performed on the data (methods) and accessors to data (properties). For example, there is a class String in the System namespace of .Net Framework Class Library (FCL). This class contains an array of characters (data) and provide different operations (methods) that can be applied to its data like *ToLowerCase()*, *Trim()*, *Substring()*, etc. It also has some properties like *Length* (used to find the length of the string).

- A class in VB.Net is declared using the keyword Class and its members are enclosed with the *End Class* marker

❖ **NameSpace:** If you declared an element at module level using the friend or public statement it becomes available to all procedures throughout the entire namespace in which it is declared.

Not that , an element accessible in a namespace is also accessible from inside any namespace nested inside that namespace .

A namespace is used in .NET Framework to define the scope of a set of related object. The namespace scope lets you organize code and gives you a way to create globally unique types. Whether or not you explicitly declare a namespace in a source file, the compiler adds a default namespace. This unnamed namespace, sometimes referred to as the global namespace, is present in every file.

Namespaces organize the objects defined in an assembly. Assemblies can contain multiple namespaces, which can in turn contain other namespaces. Namespaces prevent ambiguity and simplify references when using large groups of objects such as class libraries.

For example, the .NET Framework defines the ListBox class in the System.Windows.Forms namespace. The following code fragment shows how to declare a variable using the fully qualified name for this class:

Ex: Dim LBox As System.Windows.Forms.ListBox

Introduction of Windows Forms :--

Windows Forms is the new platform for Microsoft Windows application development, based on the .NET Framework. This framework provides a clear, object-oriented, extensible set of classes that enable you to develop rich Windows applications. Additionally, Windows Forms can act as the local user interface in a multi-tier distributed solution.

A form is a bit of screen real estate, usually rectangular, that you can use to present information to the user and to accept input from the user. Forms can be standard windows, multiple document interface (MDI) windows, dialog boxes, or display surfaces for graphical routines. The easiest way to define the user interface for a form is to place controls on its surface. Forms are objects that expose properties which define their appearance, methods which define their behavior, and events which define their interaction with the user. By setting the properties of the form and writing code to respond to its events, you customize the object to meet the requirements of your application.

As with all objects in the .NET Framework, forms are instances of classes. The form you create with the Windows Forms Designer is a class, and when you display an instance of the form at run time, this class is the template used to create the form. The framework also allows you to inherit from existing forms to add functionality or modify existing behavior. When you add a form to your project, you can choose whether it inherits from the **Form** class provided by the framework, or from a form you have previously created.

Form Life Cycle

- **Move:** This event occurs when the form is moved. Although by default, when a form is instantiated and launched, the user does not move it, yet this event is triggered before the Load event occurs.
- **Load:** This event occurs before a form is displayed for the first time.
- **VisibleChanged:** This event occurs when the Visible property value changes.
- **Activated:** This event occurs when the form is activated in code or by the user.
- **Shown:** This event occurs whenever the form is first displayed.
- **Paint:** This event occurs when the control is redrawn.
- **Deactivate:** This event occurs when the form loses focus and is not the active form.
- **Closing:** This event occurs when the form is closing.
- **Closed:** This event occurs when the form is being closed.

MsgBox Function

Syntax: MsgBox(*Prompt* [, *Buttons* As MsgBoxStyle = MsgBoxStyle.OKOnly [, *Title*])

- ***Prompt***—A string expression displayed as the message in the dialog box. The maximum length is about 1,024 characters (depending on the width of the characters used).
- ***Buttons***—The sum of values specifying the number and type of buttons to display, the icon style to use, the identity of the default button, and the modality of the message box. If you omit *Buttons*, the default value is zero. See below.
- ***Title***—String expression displayed in the title bar of the dialog box. Note that if you omit *Title*, the application name is placed in the title bar.

You can find the possible constants to use for the *Buttons* argument in Table

MsgBox constants.

Constant	**Value**	**Description**
OKOnly	0	Shows OK button only.
OKCancel	1	Shows OK and Cancel buttons.
AbortRetryIgnore	2	Shows Abort, Retry, and Ignore buttons.
YesNoCancel	3	Shows Yes, No, and Cancel buttons.
YesNo	4	Shows Yes and No buttons.
RetryCancel	5	Shows Retry and Cancel buttons.
Critical	16	Shows Critical Message icon.

MsgBox constants.

Constant	Value	Description
Question	32	Shows Warning Query icon.
Exclamation	48	Shows Warning Message icon.
Information	64	Shows Information Message icon.
DefaultButton1	0	First button is default.
DefaultButton2	256	Second button is default.
DefaultButton3	512	Third button is default.
ApplicationModal	0	Application modal, which means the user must respond to the message box before continuing work in the current application.
SystemModal	4096	System modal, which means all applications are unavailable until the user dismisses the message box.
MsgBoxSetForeground	65536	Specifies the message box window as the foreground window.
MsgBoxRight	524288	Text will be right-aligned.
MsgBoxRtlReading	1048576	Specifies text should appear as right-to-left on RTL systems such as Hebrew and Arabic.

Example:

```
Private Sub Button1_Click
   Dim Result As Integer
   Result = MsgBox("This is a message box!", MsgBoxStyle.OKCancel +
      MsgBoxStyle.Information + MsgBoxStyle.SystemModal, "Message Box")
   End If
End Sub
```

InputBox Function

User can use the **InputBox** function to get a string of text from the user.

Syntax : InputBox(Prompt [, Title [, DefaultResponseg[,XPos [, YPos]]]])

- **Prompt**— A string expression displayed as the message in the dialog box. The maximum length is about 1,024 characters (depending on the width of the characters used).

- **Title—** String expression displayed in the title bar of the dialog box. Note that if you omit **Title**, the application name is placed in the title bar.
- **DefaultResponse—** A string expression displayed in the text box as the default response if no other input is provided. Note that if you omit **DefaultResponse**, the displayed text box is empty.
- **XPos—** The distance in pixels of the left edge of the dialog box from the left edge of the screen. Note that if you omit **XPos**, the dialog box is centered horizontally.
- **YPos—** The distance in pixels of the upper edge of the dialog box from the top of the screen. Note that if you omit **YPos**, the dialog box is positioned vertically about one-third of the way down the screen.
- Input boxes let you display a prompt and read a line of text typed by the user, and the InputBox function returns the string result.

Example:

```
Private Sub Button3_Click
   Dim Result As String
   Result = InputBox("Enter your text!")
   TextBox1.Text = Result
End Sub
```

Function Statement:

Declares the name, parameters, and code that define a **Function** procedure.

Syntax:

[**Public** | **Private**] **Function** name [**(**arglist**)**] [**As** returntype]

    [statements]
    [name = expression]
    [**Exit Function**]
    [statements]
    [name = expression]

End Function

- **name :** Required. Name of the procedure.
- **Parameterlist:** Optional. List of local variable names representing the parameters of this procedure.
- **returntype :** Required if **Option Strict** is **On**. Data type of the value returned by this procedure.
- Specifies the parameters a procedure expects when it is called. Multiple parameters are separated by commas. The following is the syntax for one parameter.

[Optional] [{ ByVal | ByRef }] *parametername*[( )] [As *parametertype*] [= *defaultvalue*]

- **Optional** : Optional. Specifies that this parameter is not required when the procedure is called.
- **ByVal** : Optional. Specifies that the procedure cannot replace or reassign the variable element underlying the corresponding argument in the calling code.
- **ByRef** : Optional. Specifies that the procedure can modify the underlying variable element in the calling code the same way the calling code itself can.
- ***parametername*** **:** Required. Name of the local variable representing the parameter.

- ***parametertype* :** Required if **Option Strict** is **On**. Data type of the local variable representing the parameter.
- ***defaultvalue* :** Required for **Optional** parameters. Any constant or constant expression that evaluates to the data type of the parameter. If the type is **Object**, or a class, interface, array, or structure, the default value can only be **Nothing**.

Example:

```
Private Function res1(ByVal n1 As Double) As Double
      Return (n1 * 3.14)
End Function
Private Sub butOk_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles butOk.Click
              ans = res1(Val(Me.txtN1.Text))
              MsgBox("Result is:" & ans)
End Sub
```

<u>About SDI</u>

- Most applications in Windows 95 or later use a Single Document Interface. Each window of the application holds a single document, so if the user wants to open more documents with that application, he must open a new window. also the default mode when building an application with VB.Net. An example of an SDI application is Windows Notepad.

Advantages of SDI

- An SDI interface works very well with multiple monitors and multiple virtual desktops. It also allows users to switch between multiple open documents using the native Windows taskbar and task manager, rather than through special code that must be written into your application.

<u>About MDI</u>

- Multiple Document Interfaces were more popular in versions of Windows prior to Windows 95. With an MDI, each window within an application holds multiple documents, usually in sub-windows. Each time the user wants to open a new document, rather than opening a new window, the document opens within the existing window and shares it with all other open documents. An example of an MDI application is a tabbed Web browser like Firefox, where users have an option to open documents in multiple tabs within the same window.

Advantages of MDI

- MDI applications can often handle multiple documents more readily than SDI programs. For example, many MDI text editors allow the user to open multiple text files side by side in the same window, making it easy to compare and look up information from a second document while working on the first.