BCA Semester-5
US05CBCA01-Visual Programming through VB.NET

UNIT-2 CONDITIONS, LOOPS AND WINDOWS FORMS

Making decisions with If…Else statements

You can use the If statements to evaluate one or more conditions and execute appropriate code.

Syntax:

If *condition* **Then**

    [statements]

ElseIf *condition-n* **Then**

    [statements]

Else

    [statements]

End If

It is also possible to combine multiple conditions using And/Or.

Example:

```
If perc>=70 Then
    Result=”Distinction”
ElseIf perc>=60 Then
    Result=”First” Then
ElseIf perc>=50 Then
    Result=”Second”
ElseIf perc>=45 Then
    Result=”Pass”
Else
    Result=”Failed”
End If
```

Using Select…Case

To get a value from the user and then respond in a different way, Select Case can be used.

Syntax:

```
Select Case test-expression
‘ Checking against single value
Case value-1
    [statements]
‘ Checking against single value
Case value-2
    [statements]
‘ Checking against multiple values separated by comma
Case value-3, value-4, value-5
    [statements]
‘ Checking against a range with >= And <=
Case test-expression >= value-6 And test-expression <= value-8
    [statements]
Case Else
        [statements]
End Select
```

Example:

```
Select Case Now.DayOfWeek
Case DayOfWeek.Monday
message = “Have a nice week”
Case DayOfWeek.Tuesday, DayOfWeek.Wednesday, _
DayOfWeek.Thursday, DayOfWeek.Friday
message = “Welcome back!”
Case DayOfWeek.Friday, DayOfWeek.Saturday, DayOfWeek.Sunday
message = “Have a nice weekend!”
End Select
MsgBox(message)
```

Loop Structures

Visual Basic supports the following loop structures:

1) For…Next

2) Do…Loop

3) While…End While

For…Next

The For…Next loop is one of the oldest loop structures in programming languages. Unlike the other two loops, the For…Next loop requires that you know how many times the statements in the loop will be executed.

The For…Next loop uses a variable (it's called the loop's counter) that increases or decreases in value during each repetition of the loop.

Syntax:

For counter = start To end [Step increment]

statements

Next [counter]

The keywords in the square brackets are optional. The arguments counter, start, end, and increment are all numeric. The loop is executed as many times as required for the counter to reach (or exceed) the end value.

In executing a For…Next loop, Visual Basic completes the following steps:

1. Sets counter equal to start

2. Tests to see if counter is greater than end. If so, it exits the loop. If increment is negative, Visual Basic checks if counter is less than end. If it is, it exits the loop.

3. Executes the statements in the block

4. Increments counter by the amount specified with the increment argument. If the increment argument isn't specified, counter is incremented by 1.

5. Repeats the statements

The For…Next loop in the following example scans all the elements of the numeric array data and calculates their average.

Example:

```
Dim i As Integer, total As Double

For i = 0 To data.GetUpperBound(0)

total = total + data(i)

Next i

Label1.Text = total / data.Length
```

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 won't be executed, not even once.

Finally, the counter variable need not be listed after the Next statement, but it makes the code easier to read, especially when For…Next loops are nested within each other.

Do…Loop

The Do…Loop executes a block of statements for as long as a condition is True. Visual Basic evaluatesanexpression, and if it's True, the statements are executed. When the end of block is reached, theexpression is evaluated again and, if it's True, the statements are repeated. If the expression is False,the program continues and the statement following the loop is executed.

There are two variations of the Do…Loop statement; both use the same basic model. A loop canbeexecuted either while the condition is True or until the condition becomes True. These two variations use the keywords While and Until to specify how long the statements are executed.

To executea block of statements while a condition isTrue, use the following syntax:

Do While condition

statement-block

Loop

To execute a block of statements until the condition becomes True, use the following syntax:

Do Until condition

statement-block

Loop

When Visual Basic executes these loops, it first evaluates condition. If condition is False, a Do…While loop is skipped (the statements aren't even executed once) but a Do…Until loop is executed. When the Loop statement is reached, Visual Basic evaluates the expression again and repeats the statement block of the Do…While loop if the expression is True, or repeats the statements of the Do…Until loop if the expression is False.

In short, the Do While loop is executed when the condition is True, and the Do Until loop is executed when the condition is False.

The Do…Loop can execute any number of times as long as condition is True or False, as appropriate (zero or nonzero if the condition evaluates to a number). Moreover, the number of iterations need not be known before the loops starts. In fact, the statements may never execute if condition is initially False for While or True for Until.

Here's a typical example of using a Do…Loop. Suppose the string MyText holds a piece of text (perhaps the Text property of a TextBox control), and you want to count the words in the text. (We'll assume that there are no multiple spaces in the text and that the space character separates successive words.) To locate an instance of a character in a string, use the InStr() function, which accepts three arguments:

1. The starting location of the search
2. The text to be searched
3. The character being searched

The following loop repeats for as long as there are spaces in the text. Each time the InStr() function finds another space in the text, it returns the location (a positive number) of the space. When there are no more spaces in the text, the InStr() function returns zero, which signals the end of the loop, as shown:

```
Dim MyTextAs String = "The quick brown fox jumped over the lazy dog"
Dim position, words As Integer
position = 1
Do While position > 0
position = InStr(position + 1, MyText, " ")
words = words + 1
Loop
Console.WriteLine "There are " & words & " words in the text"
```

The Do…Loop is executed while the InStr() function returns a positive number, which happens foras long as there are more words in the text. The variable position holds the location of each successivespace character in the text. The search for the next space starts at the location of the current space plus1 (so that the program won't keep finding the same space). For each space found, the program increments the value of the wordsvariable, which holds the total number of words when the loop ends.

Another variation of the Do loop executes the statements first and evaluates the condition after each execution. This Do loop has the following syntax:

```
Do
statements
Loop While condition
```

```
Do
statements
Loop Until condition
```

The statements in this type of loop execute at least once, since the condition is examined at the end of the loop. Could we have implemented the previous example with one of the last two types of loops? The fact that we had to do something special about zero-length strings suggests that this problem shouldn't be coded with a loop that tests the condition at the end. Since the loop's body will be executed once, the words variable is never going to be zero.

As you can see, you can code loops in several ways with the Do…Loop statement, and the way you use it depends on the problem at hand and your programming style.

While…End While

The While…End While loop executes a block of statements as long as a condition is True. The While loop has the following syntax:

While condition

statement-block

End While

If condition is True, all statements are executed and when the End While statement is reached, control isreturned to the While statement, which evaluates condition again. If condition is still True, the process is repeated. If condition is False, the program resumes with the statement following End While.

The Exit Statement

The Exit statement allows you to exit prematurely from a block of statements in a control structure, from a loop, or even from a procedure. Suppose you have a For…Next loop that calculates the square root of a series of numbers. Because the square root of negative numbers can't be calculated (the Sqrt() function will generate a runtime error), you might want to halt the operation if the array contains an invalid value. To exit the loop prematurely, use the Exit For statement as follows:

```
For i = 0 To UBound(nArray)
If nArray(i) < 0 Then Exit For
nArray(i) = Math.Sqrt(nArray(i))
Next
```

If a negative element is found in this loop, the program exits the loop and continues with the statement following the Next statement.

There are similar Exit statements for the Do loop (Exit Do) and the While loop (Exit While), as well as for functions and subroutines (Exit Function and Exit Sub).

Working with Procedures

An application is made up of small, self-contained segments. The code you write isn't a monolithic listing; it's made up of small segments called procedures, and you work on one procedure at a time.

For example, when you write code for a control's Click event, you concentrate on the event athand—namely, how the program should react to the Click event. What happens when the control is double-clicked, or when another control is clicked, is something you will worry about later in another control's event handler.

This is basically "divide and conquer" approach wherein each task is performed by a separate procedure that is written and tested separately from the others.

A Procedure is a block or set of statements which performs a specific task.Use of procedures makes the code modular. Each section of code performs a specific single task.Further, the same code can be used at many places without the need of re-typing or copy-paste.

In VB.NET all the executable code must be in procedures. There are two types of procedure:

(1) Sub-procedure (does not return any value)
(2) Function (returns a value)

Sub-procedures

A sub-procedure is a block of statements that carries out a well-defined task. The block of statements is placed within a set of Sub…End Sub statements and can be invoked by name. The following sub-procedure displays the current date in a message box and can be called by its name, ShowDate():

```
Sub ShowDate()

MsgBox(Now())

End Sub
```

Normally, the task a sub-procedure performs is more complicated than this; nevertheless, even this isa block of code isolated from the rest of the application. All the event handlers in Visual Basic, forexample, are coded as sub-procedures. The actions that must be performed each time a button is clickedare coded in the button's Click procedure.

The statements in a sub-procedure are executed, and when the End Sub statement is reached,control returns to the calling program. It's possible to exit a sub-procedure prematurely, with the ExitSub statement. For example, some condition may stop the sub-procedure from successfully completingits task.

All variables declared within a sub-procedure are local to that sub-procedure. When the sub-procedure exits,all variables declared in it cease to exist.

Most procedures also accept and act upon arguments. The ShowDate() subroutine displays the current date on a message box. If you want to display any other date, you'd have to pass an argument to the subroutine telling it to act on a different value, like this:

```
Sub ShowDate(ByVal birthDate As Date)

MsgBox(birthDate)

End Sub
```

birthDate is a variable that holds the date to be displayed; its type is Date. The ByVal keyword meansthat the subroutine sees a copy of the variable, not the variable itself. What this means practically is thatthe subroutine can't change the value of the birthDate variable.

To pass an argument by ref, ByRef keyword is used.

When you pass an array as argument to a procedure, the array is always passed by reference—even if you specify the ByVal keyword. The reason for this is that it would take the machine some time to create a copy of the array. Since the copy of the array must also live in memory, passing too many arrays back and forth by value would deplete your system's memory.

When you pass objects as arguments, they're passed by reference, even if you have specified the ByVal keyword. The procedure can access and modify the members of the object passed as argument, and the new value will be visible in the procedure that made the call.

To display the current date on a message box, you must call the ShowDate subroutine as followsfrom within your program:

ShowDate()

To display another date with the second implementation of the subroutine, use a statement likethe following:

Dim myBirthDate = #1/1/1985#

ShowDate(myBirthDate)

Or, you can pass the value to be displayed directly without the use of an intermediate variable:

ShowDate(#2/9/1960#)

Functions

A function is similar to a subroutine, but a function returns a value. Subroutines perform a task and don't report anything to the calling program; functions commonly carry out calculations and report the result.

The value you passback to the calling program from a function is called the return value, and its type must match the typeof the function. Functions accept arguments, just like subroutines. The statements that make up a function are placed in a set of Function…End Function statements, as shown here:

Function NextDay() **As Date**

Dim theNextDay As Date

theNextDay = DateAdd(DateInterval.Day, 1, Now())

Return(theNextDay)

End Function

The result of a function is returned to the calling program with the Return statement. In our example, the Return statement happens to be the last statement in the function, but it could appear anywhere; it could even appear several times in the function's code. The first time a Return statement is executed, the function terminates and control is returned to the calling program.

Returning Multiple Values

If you want to write a function that returns more than a single result, you will most likely pass additional arguments by reference and set their values from within the function's code.

Similar to variables, a custom function has a name, which must be unique in its scope. If you declare a function in a form, the function name must be unique in the form. If you declare a function as Public or Friend, its name must be unique in the project. Functions have the same scope rules as variables and can be prefixed by many

of the same keywords. In effect, you can modify the default scope of a function with the keywords Public, Private, Protected, Friend, and Protected Friend.

Passing a variable number of arguments

Visual Basic supports the **ParamArray** keyword, which allows you to pass a variable number of arguments to a procedure/function.

```
Private Function average (ByVal ParamArray a() As Integer) As Decimal

    Dim n As Integer
    Dim sum As Integer
    Dim avg As Decimal
       sum = 0

    For Each n In a
       sum = sum + n
    Next
       avg = sum / a.Length
    Return avg
End Function
```

Functions Returning Array

In VB.NET functions can also return arrays. This is an interesting possibility that allows you to write functions that return not only multiple values, but also an unknown number of values. Earlier in the chapter you saw how to return multiple values from a function as arguments, passed to the function by reference.

To implement a function that returns an array, you must do the following:

1. Specify a type for the function's return value, and add a pair of parentheses after the type's name. Don't specify the dimensions of the array to be returned here; the array will be declared formally in the function.

```
Function FindAll(ByVal a As Integer, ByVal b As Integer) As Decimal()
```

2. In the function's code, declare an array of the same type and specify its dimensions. If the function should return four values, use a declaration like this one:

```
Dim r(4) As Decimal
```

The Results array will be used to store the results and must be of the same type as the function—its name can be anything.

3. To return the Results array, simply use it as argument to the Return statement:

```
Return r
```

4. In the calling procedure, you must declare an array of the same type without dimensions:

```
Dim results() As Decimal
```

5. Finally, you must call the function and assign its return value to this array:

```
Results = FindAll(4,5)
```

Assume that we want to define a function that takes two intergers as argument and find their minimum, maximum, sum and average. Then, the function stores these results in an array and returns entire array. Such a function can be defined as follows:

```
Function FindAll(ByVal a As Integer, ByVal b As Integer) As Decimal()
    Dim r(4) As Decimal
r(0) = Math.Min(a, b)
r(1) = Math.Max(a, b)
r(2) = a + b
r(3) = (a + b) / 2
    Return r
  End Function
```

It can be called as follows:

```
Dim results() As Decimal
:::
results = FindAll(4, 5)
```

Classes in VB.NET

Classes are at the very heart of Visual Studio. Just about everything you do with VB.NET is a class, and you already know how to use classes. The .NET Framework itself is a large collection of classes, and you can import any of them into your applications. You simply declare a variable of the specific class type, initialize it, and then use it in your code.

The 3,500 (or so) classes that come with the .NET Framework give you access to all the objects used by the operating system. All you have to do is use them in your application. You don't have to see the code, and you don't have to know anything about them. In fact, you're reusing code that Microsoft has already written.

As you have noticed, even a Form is a Class, and it includes the controls on the form and the code behind them. All the applications you've written so far are enclosed in a set of Class…End Class statements.

When you create a variable of any type, you're creating an instance of a class. The variable lets you access the functionality of the class through its properties and methods. Even the basic data types are implemented as classes (the System.Integer class, System.Double, and so on).

To define a class, you only need to use the Class statement as follows:

```
Public Class classname
:::
End Class
```

You can create an object of this class as follows:

```
Dim object As New classname()
```

OR

```
Dim object As classname = New classname()
```

Members of class

There are four types of members of a class: Fields, Properties, Methods and Events.

Fields

The fields of a class are public data members of a class which can be directly accessed.

```
Public Class Student
        Public RollNo As Integer
End Class
```

Now, you can refer to that field in an object of this class using o*bject.field* syntax:

```
Dim s1 As New Student()
```

S1.RollNo =1

You can use the Shared keyword to create class (i.e static) data members.

Public **Shared** Total_Marks As Integer = 800

Such a class data member can be accessed using classname.field.

Student.Total_Marks

You can also make a field constant using Const keyword.

Public **Const** Total_MarksAs Integer = 800

Using fields like this can give you direct access to the fields, but that unusual in OOP. An easy way of securing data is to use *properties.*

Properties

Properties are accessed and changed like fields, but are handled through Property Get and Property Set procedures, which provide more control on how values are set or returned.

A property is defined as follows:

```
Private Property name As String

Get
        Return Name
End Get
Set (ByVal Value As String)
        Name = Value
End Set
End Property
```

Methods

Methods represent operations that can be performed by object or on objects of a class. You can define methods by adding sub-procedures or functions in a class. For example,

```
Sub Display ()
        Label1.Text = s1.RollNo
        Label2.Text = s1.Name
End Sub
```

A method can also be a shared (i.e. static). Such a method can be defined using Shared keyword. A shared method can access only shared members of a class.

Events

An event allows the user to define the task to be performed when something happens. It is basically the code which is executed in response to some interaction.

Events are implemented using sub-procedures only.

Modules in VB.NET

A module is a collection of procedures and functions which perform specific task. A module normally acts as a ready-to-use library.Modules are mainly designed to contain code only but they can also have members like a class.

You can create a module with a Module…End Module statement.

Module *Modulename*

:::::

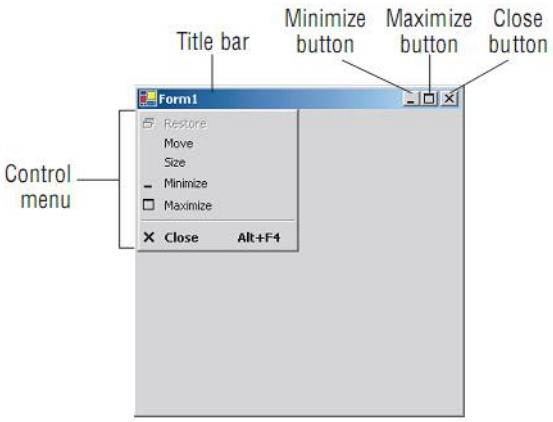
End Module

Difference between class & module

- Members of modules are implicitly shared.
- You can not create objects of a module.
- Modules do not support inheritance.
- Modules can not implement interfaces.
- A module can be declared only inside a namespace.
- Modules can not be nested inside other types.
- If two or more modules have member with same name, member-name must be prefixed with module name to distinguish.

Working with forms

- In Visual Basic, the form is the container for all the controls that make up the user interface.
- When a Visual Basic application is executing, each window it displays on the desktop is a form.
- In VB.NET, the support for form is made through System.Windows.Forms.Form class.
- The terms form and window describe the same entity. A window is what the user sees on the desktop when the application is running. A form is the same entity at design time. The proper term is a Windows form.
- The forms that constitute the visible interface of your application are called Windows forms; this term includes both the regular forms and dialog boxes, which are simple forms you use for very specific actions, such as to prompt the user for a specific piece of data or to display critical information.
- A dialog box is a form with a small number of controls, no menus, and usually an OK and a Cancel button to close it.
- Forms have built-in functionality that is always available without any programming effort on your part. You can move a form around, resize it, and even cover it with other forms. You do sowith the mouse, or with the keyboard through the Control menu.
- Forms are not passive containers; they're "intelligent" objects that are aware of the controls placed on them and can actually manipulate the controls at runtime.
- At the top of the form is title bar, which displays the form's title. At right in the title bar is control box, including minimize/maximize and close buttons.
- Under the title bar comes a menu bar, if it is added at all. Under the menu bar, a set of small icons-called tool bar can be there.



- A form is known as Me from its own code.
- The Form class has one class member-ActiveForm, many object members and events.
- ActiveForm refers to currently active form in the application. There may many forms in an application, but only one of them can be active at any time.

Important Properties

Sr. No.	Property	Use
1	AcceptButton	Gets or sets the button on the form that is pressed when the user uses theEnter key.
2	ActiveMdiChild	Gets the currently active multiple document interface (MDI) child window.
3	BackColor	Gets or sets the background color for this form.
4	BackgroundImage	Gets or sets the background image in the form.
5	CancelButton	Indicates the button control that is pressed when the user presses the ESCkey.

6	ControlBox	Gets or sets a value indicating if a control box is displayed.
7	Controls	Gets or sets the collection of controls contained within the form.
8	Enabled	Gets or sets a value indicating if the form is enabled.
9	ForeColor	Gets or sets the foreground color of the form.
10	FormBorderStyle	Gets or sets the border style of the form. Fixed3D— A fixed, three-dimensional border. FixedDialog— A thick, fixed dialog-style border. FixedSingle—A fixed, single-line border. FixedToolWindow—A tool window border that is not resizable. None—No border. Sizable—A resizable border. SizableToolWindow— A resizable tool window border.
11	Icon	Gets or sets the icon for the form.
12	IsMdiChild	Indicates if the form is an MDI child form.
13	MaximizeBox	Gets or sets a value indicating if the maximize button is displayed in thecaption bar of the form.
14	MaximumSize	Returns the maximum size the form can be resized to.
15	MdiChildren	Returns an array of forms of the MDI child forms that are parented to thisform.
16	MdiParent	Gets or sets the current MDI parent form of this form.
17	MinimizeBox	Gets or sets a value indicating if the minimize button is displayed in thecaption bar of the form.
18	Modal	Gets a value indicating if this form is displayed modally.
19	StartPosition	Gets or sets the starting position of the form at run time. CenterParent- The form is centered within the boundaries of its parent form. CenterScreen- The form is centered in the current display Manual – The Location & Size properties of the form decided its starting position.
20	Text	Gets or sets the text associated with this form.
21	Visible	Gets or sets a value indicating if the form is visible.
22	WindowState	Gets or sets the form's window state.

Important Events

Sr. No.	Property	Description
1	Click	Occurs when the form is clicked.
2	Closed	Occurs when the form is closed.
3	Closing	Occurs when the form is closing.
4	Deactivate	Occurs when the form loses focus and is not the active form.
5	GotFocus	Occurs when the form receives focus.
6	KeyDown	Occurs when a key is pressed down while the form has focus.
7	KeyPress	Occurs when a key is pressed while the form has focus.
8	KeyUp	Occurs when a key is released while the form has focus.
9	Load	Occurs before a form is displayed for the first time.
10	LostFocus	Occurs when the form loses focus.
11	MouseDown	Occurs when the mouse pointer is over the form and a mouse button ispressed.
12	MouseEnter	Occurs when the mouse pointer enters the form.
13	MouseHover	Occurs when the mouse pointer hovers over the form.
14	MouseLeave	Occurs when the mouse pointer leaves the form.
15	MouseMove	Occurs when the mouse pointer is moved over the form.
16	MouseUp	Occurs when the mouse pointer is over the form and a mouse button isreleased.
17	Move	Occurs when the form is moved.

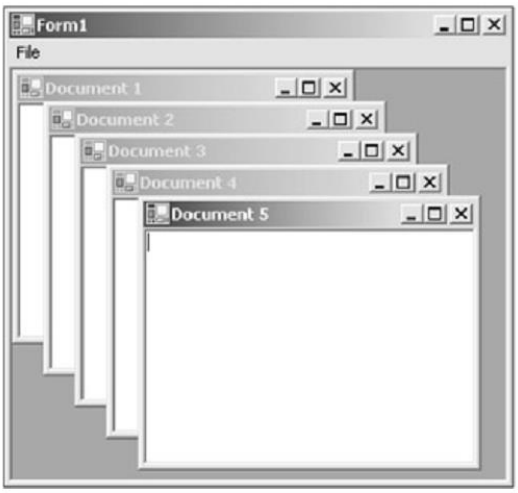
18	Paint	Occurs when the form is redrawn.
19	Resize	Occurs when the form is resized.

Important Methods

Method	Description
Activate	Activates the form (gives it focus and makes it active).
Close	Closes the form.
Dispose	Releases the resources used by the form.
Hide	Hides the form
Show	Shows the form

Working with MDI Forms

- Besides standard forms, Visual Basic also supports Multiple Document Interface (MDI) forms. Such an MDI form is shown below:



- You can see that an MDI form closely resembles a standard form, with one major difference—the client area of an MDI form acts as a kind of corral for other forms. That is, an MDI form, also called an MDI parent form can display MDI children in it, which is how the multiple document interface works.
- In the figure given above we have five documents open in the MDI form.
- That's the third type of form you can have in Visual Basic—MDI child forms. These forms appear in MDI child windows, but otherwise are very similar to standard forms.
- You make forms into MDI parents and children by setting the **IsMdiContainer** and **MdiParent** properties.
- Thus, there are three types of Windows forms available to us in Visual Basic: standard forms, MDI forms, and MDI child forms.

Using the MsgBox Function

- Syntax:

Public Function MsgBox(Prompt As Object [, Buttons As MsgBoxStyle =MsgBoxStyle.OKOnly [, Title As Object = Nothing]]) As MsgBoxResultArguments

- Here are the arguments you pass to this function:

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.

If you want the message box prompt to be more than one line of text, you can force separate lines of text using a carriage return character (Chr(13)), a linefeed character (Chr(10)), or a carriage return/linefeed together (Chr(13) &Chr(10)) between each line.

- You can find the possible constants to use for the Buttons argument in table given below:

Constant	**Description**
OKOnly	Shows OK button only.
OKCancel	Shows OK and Cancel buttons.
AbortRetryIgnore	Shows Abort, Retry, and Ignore buttons.
YesNoCancel	Shows Yes, No, and Cancel buttons.
YesNo	Shows Yes and No buttons.
RetryCancel	Shows Retry and Cancel buttons.
Critical	Shows Critical Message icon.
Question	Shows Warning Query icon.
Exclamation	Shows Warning Message icon.
Information	Shows Information Message icon.
DefaultButton1	First button is default.
ApplicationModal	Application modal, which means the user must respond to the message box before continuing work in the current application.
SystemModal	System modal, which means all applications are unavailable until the user dismisses the message box.

- Here are the possible MsgBox Result return-values, indicating which button in the message box the user clicked:

Button clicked	Constant returned by MsgBox( ) function
OK	vbOK
Cancel	vbCancel
Abort	vbAbort
Retry	vbRetry
Ignore	vbIgnore
Yes	vbYes
No	vbNo

- To see which button is clicked by the user & then take action accordingly, following code may be used:

```
Dim result As Integer ‘ Module-level variable

result = MsgBox("Do you want to save?", MsgBoxStyle.YesNoCancel)

    If result = vbYes Then
        Label1.Text = "Yes"
ElseIf result = vbNo Then
        Label1.Text = "No"
```

```
        ElseIf result = vbCancel Then
            Label1.Text = "Cancel"
        End If
```

Using the InputBox Function

- You can use the InputBox function to get a string of text from the user. Here's the syntax for this function:

Public Function InputBox(Prompt As String [, Title As String = "" [, DefaultResponse As String = "" [, XPos As Integer = -1 [, YPos As Integer = -1]]]]) As String

- And here are the arguments for this function:

Prompt— A string expression displayed as the message in the dialog box. The maximum length is about1,024 characters.

Title— String expression displayed in the title bar of the dialog box. Note that if you omit Title, theapplication name is placed in the title bar.

DefaultResponse— A string expression displayed in the text box as the default response if no other inputis 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 thatif 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 ifyou 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.

```
Dim Result As String
Result = InputBox("Enter your text!")
```