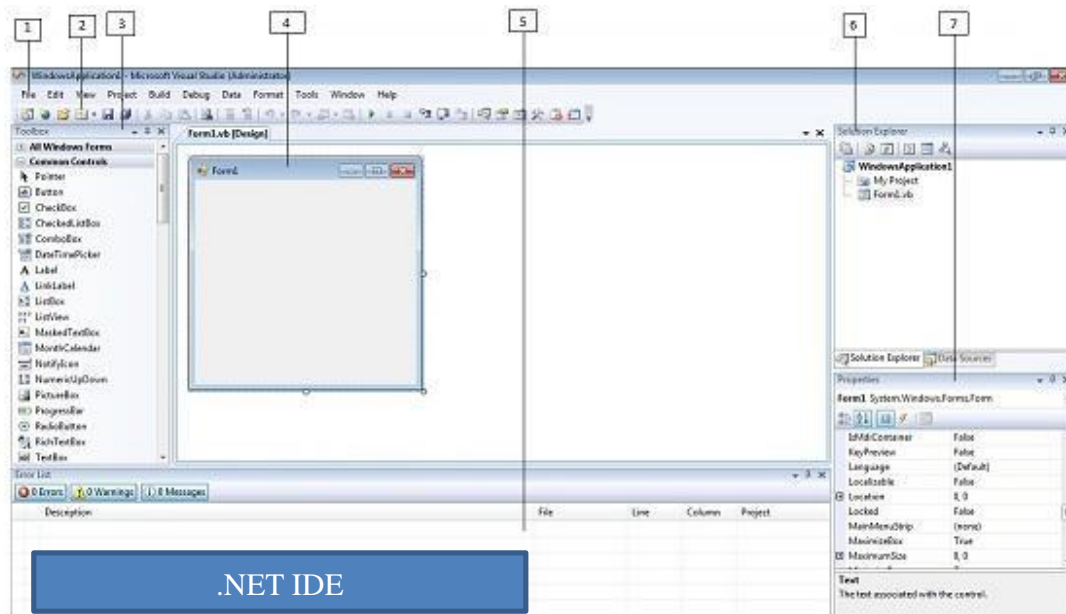


IDE

An integrated development environment (IDE), also known as integrated design environment and integrated debugging environment, is a type of computer software suite that assists computer programmers to develop software.

The integrated development environment (IDE) is important in helping you create, run and debug any .Net programs or applications. You can consider VB.Net IDE as Microsoft Word and VB.Net programs as Word documents.



1. Menu Bar

It consists of menus that help you manipulate VB.Net programs in the project. The menus are listed from left to right as File, Edit, View, Project, Build, Debug, Tools, Window, and Help.

2. Standard Toolbar

Contains buttons that are shortcuts to some commonly used menu items.

3. Toolbox (Ctrl + Alt +X)

The window is very important in the VB.Net IDE. It contains control templates or components that are available for you to use. You can simply drag and drop any control from toolbox to your form

4. Forms Designer (Shift + F7)

We can drag and drop controls in this view of the form. We can also see some type of preview of our form

5. Output Window

The Output window is where many of the tools, including the compiler, send their output. Every time you start an application, a series of messages is displayed in the Output window. These messages are generated by the compiler, and you need not understand them at this point. If the Output window is not visible, choose View > Other Windows > Output from the menu.

6. Solution Explorer (Ctrl + Alt +L)

The window contains a Windows Explorer-like tree view of all the customizable forms and general code (modules) that make up a VB.Net application. The Solution Explorer provides you with an organized view of your project and program files associated with the project. Select the Solution Explorer on the View menu when you cannot find the Solution Explorer in your IDE.

7. Properties Window (F4)

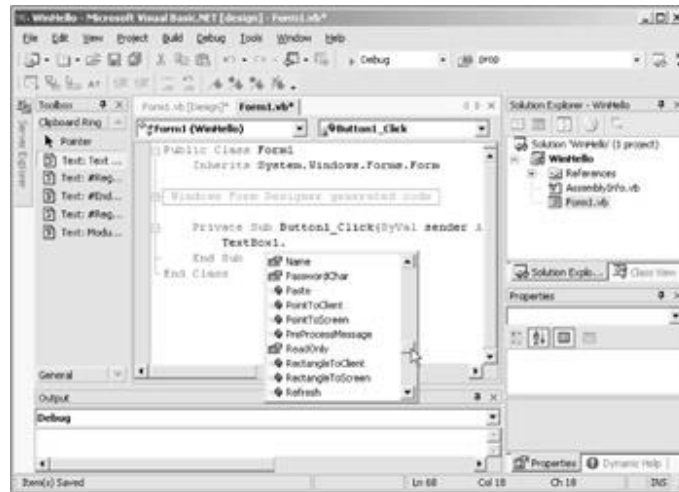
Window The properties window displays the properties for a form or a control. Properties describe attributes such as size, color, and font of a control. Each form or control has its own set of properties. When you click on a control or the form, the properties will be listed in the properties window. There are two columns in the properties window. The first column lists the property names and the second column shows the current value of the property. The value can be changed at the design phase of the form or through the program code. There are lots of properties associated with controls.

Title Bar -It shows the title of the VB.Net project you are currently working on. The default project title is the project name you have specified when you create a new project. If you would like to change the project name or title to other name, you can change it through Project -> Project Properties.

Code editor window (F7) –where we can write the coding of the form or class etc.

IntelliSense

One useful feature of VB .NET code designers is Microsoft's *IntelliSense*. IntelliSense is what's responsible for those boxes that open as you write your code, listing all the possible options and even completing your typing for you. IntelliSense is one of the first things you encounter when you use VB .NET, and you can see an example in Figure 1.25, where I'm looking at all the members of a text box object.

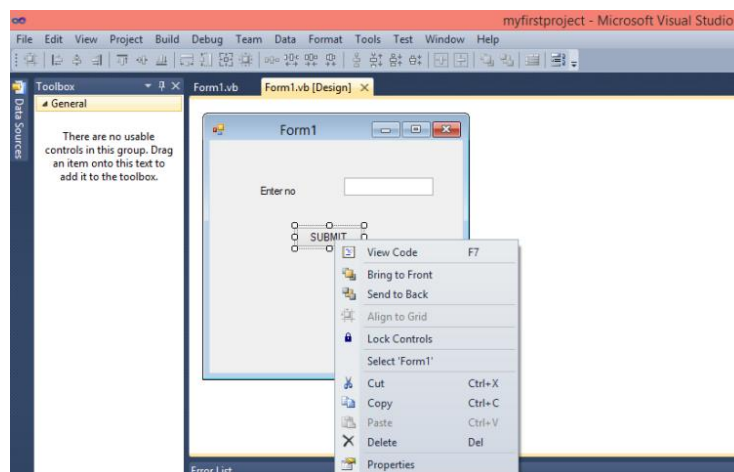


Context Menu

It contains shortcut for frequently performed actions.

To open context menu , select any object and click the right mouse button

Context menu will open according to the selected object



What is Auto Hide icon?

This is a new feature in Visual Studio that hides away the windows not currently in use. The reference to "not currently *using*" suggests that the windows or panels are not in focus; however, they are not closed down. Thus, as you change windows like going from Solution Explorer to Help the one you are leaving slides closed.

Variable:

A variable is something that is used in a program to store data in memory.

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.

The **Dim** statement is used for variable declaration and storage allocation for one or more variables.

Example: **Dim** Counter as Integer

Datatype:

The data type of a programming element refers to what kind of data it can hold and how the data is stored.

Dim b As Byte Dim n As Integer Dim c As Char Dim s As String

b = 1n = 1234567

c = "U"

```
s = "Me"
```

[illegible]

Double (double-precision floating-point)	System.Double	8 bytes	-1.79769313486231570E+308 through -4.94065645841246544E-324 for negative values; 4.94065645841246544E-324 through 1.79769313486231570E+308 for positive values.
Integer	System.Int32	4 bytes	-2,147,483,648 through 2,147,483,647.
Long (long integer)	System.Int64	8 bytes	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807.
Short	System.Int16	2 bytes	-32,768 through 32,767.
Single (single-precision floating-point)	System.Single	4 bytes	-3.4028235E+38 through -1.401298E-45 for negative values; 1.401298E-45 through 3.4028235E+38 for positive values.
User define datatype (structure,enum,array)			
User-Defined Type (structure)	(inherits from System.ValueType)	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.

Data Type Conversion Function

CBool(expression)- Converts the expression to Boolean data type.

```
Dim a, b, c As Integer

Dim check As Boolean

a = 5

b = 5

' The following line of code sets check to True.

check = CBool(a = b)

c = 0

' The following line of code sets check to False.

check = CBool(c)
```

CByte(expression)- Converts the expression to Byte data type.

```
Dim aDouble As Double  
  
Dim aByte As Byte  
  
aDouble = 125.5678  
  
' The following line of code sets aByte to 126.  
  
aByte = CByte(aDouble)
```

CChar(expression) - Converts the expression to Char data type.

```
Dim aString As String  
  
Dim aChar As Char  
  
' CChar converts only the first character of the string.  
  
aString = "BCD"  
  
' The following line of code sets aChar to "B".  
  
aChar = CChar(aString)
```

CDate(expression) - Converts the expression to Date data type

```
Dim aDateString, aTimeString As String  
  
Dim aDate, aTime As Date  
  
aDateString = "February 12, 1969"  
  
aTimeString = "4:35:47 PM"  
  
' The following line of code sets aDate to a Date value.  
  
aDate = CDate(aDateString)  
  
' The following line of code sets aTime to Date value.  
  
aTime = CDate(aTimeString)
```

CDBl(expression)- Converts the expression to Double data type.

```
Dim aDec As Decimal
```

```
Dim aDbl As Double
```

```
' The following line of code uses the literal type character D to make aDec a Decimal.
```

```
aDec = 234.456784D
```

```
' The following line of code sets aDbl to 1.9225456288E+1.
```

```
aDbl = CDBl(aDec * 8.2D * 0.01D)
```

CDec(expression)- Converts the expression to Decimal data type.

```
Dim aDouble As Double
```

```
Dim aDecimal As Decimal
```

```
aDouble = 10000000.0587
```

```
' The following line of code sets aDecimal to 10000000.0587.
```

```
aDecimal = CDec(aDouble)
```

CInt(expression)- Converts the expression to Integer data type.

```
Dim aDbl As Double
```

```
Dim anInt As Integer
```

```
aDbl = 2345.5678
```

```
' The following line of code sets anInt to 2346.
```

```
anInt = CInt(aDbl)
```

CLng(expression)- Converts the expression to Long data type.

```
Dim aDbl1, aDbl2 As Double
```

```
Dim aLng1, aLng2 As Long  
aDb11 = 25427.45  
aDb12 = 25427.55  
' The following line of code sets aLng1 to 25427.  
aLng1 = CLng(aDb11)  
' The following line of code sets aLng2 to 25428.  
aLng2 = CLng(aDb12)
```

CObj(expression)- Converts the expression to Object type.

```
Dim aDouble As Double  
Dim anObject As Object  
aDouble = 2.7182818284  
' The following line of code sets anObject to a pointer to aDouble.  
anObject = CObj(aDouble)
```

CShort(expression)- Converts the expression to Short data type.

```
Dim aByte As Byte  
Dim aShort As Short  
aByte = 100  
' The following line of code sets aShort to 100.  
aShort = CShort(aByte)
```


CSng(expression)- Converts the expression to Single data type.

```
Dim aDouble1, aDouble2 As Double  
Dim aSingle1, aSingle2 As Single  
aDouble1 = 75.3421105  
aDouble2 = 75.3421567  
' The following line of code sets aSingle1 to 75.34211.  
aSingle1 = CSng(aDouble1)  
' The following line of code sets aSingle2 to 75.34216.  
aSingle2 = CSng(aDouble2)
```

CStr(expression)- Converts the expression to String data type.

```
Dim aDouble As Double  
Dim aString As String  
aDouble = 437.324  
' The following line of code sets aString to "437.324".  
aString = CStr(aDouble)
```

CTYPE Function

It used to convert one type to another type. Instead of remember all conversion function, remember only CTYPE function.

Syntax: CType(expression,typename)

Example:

```
dim a as integer  
dim b as integer
```

b=66.7
a=CType(b,Integer)

Operators:

Operators	Description
<u>Arithmetic Operators</u>	+, -, *, /, \, MOD, ^
<u>Assignment Operators</u>	=, ^=, *=, /=, \=, +=, -=, <<=, >>=, &=
<u>Comparison Operators</u>	>, <, =, <>, >=, <=
<u>Concatenation Operators</u>	+, &
<u>Logical/Bitwise Operators</u>	And, Or, Not, Xor
<u>Bit Shift Operators</u>	>>, <<

Boxing and Unboxing

Boxing and unboxing is an important concept in VB.NET's type system. With Boxing and Unboxing one can link between value-types and reference-types by allowing any value of a value-type to be converted to and from type object.

Boxing

- Boxing is a mechanism in which value type is converted into reference type.
- It is implicit conversion process

Unboxing

- Unboxing is a mechanism in which reference type is converted into value.
- It is explicit conversion process.

Example:

Dim I as Integer = 10

Dim o as Object = I **'Boxing**

Dim J As integer=CInt(o) **'Unboxing**

Constant

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

const s1 as String = "hello"

Comments in vb.net

Rem or ' (single quote) can be used to comment the line.

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.

Option Explicit and Option Strict

Option Explicit

- **Option Explicit** statement ensures whether the compiler requires all variables to be explicitly declared or not before it use in the program.
Option Explicit [On Off]
- The **Option Explicit** has two modes. **On** and **Off** mode.
- when ON , you have to declare all the variable before you use it in the program . If not , it will generate a compile-time error whenever a variable that has not been declared is encountered .
- when OFF , Vb.Net automatically create a variable whenever it sees a variable without proper declaration.
- By default the *Option Explicit is On*
- With the Option Explicit On , you can reduce the possible errors that result from misspelled variable names.
- Because in Option Explicit On mode you have to declare each variable in the program for storing data.

Option Explicit On	Option Explicit Off
Public Class Form1	Option Explicit Off Public Class Form1

Private Sub Button1_Click(ByVal sender As System.Object, _ ByVal e As System.EventArgs) Handles Button1.Click Dim someVariable As String someVariable = "Option Explicit ON" MsgBox(someVariable) End Sub End Class	Private Sub Button1_Click(ByVal sender As System.Object, _ ByVal e As System.EventArgs) Handles Button1.Click someVariable = "Option Explicit ON" MsgBox(someVariable) End Sub End Class
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Option Strict

- **Option Strict** is prevents program from automatic variable conversions, that is implicit data type conversions .

Option Strict [On Off]

- By default *Option Strict is Off*

Public Class Form1 Private Sub Button1_Click(ByVal sender As System.Object, _ ByVal e As System.EventArgs) Handles Button1.Click Dim longNum As Long Dim intNum As Integer longNum = 12345 intNum = longNum MsgBox(intNum) End Sub End Class	Option Strict On Public Class Form1 Private Sub Button1_Click(ByVal sender As System.Object, _ ByVal e As System.EventArgs) Handles Button1.Click Dim longNum As Long Dim intNum As Integer longNum = 12345 intNum = CInt(longNum) MsgBox(intNum) End Sub End Class
<i>Option Strict is Off</i>	<i>Option Strict is ON</i>

Option Compare

The **Option Compare** statement controls whether string comparisons are carried out using binary comparisons or text comparisons. If no such statement is specified in a file, the compilation environment controls which type of comparison will be used.

Syntax: option compare {binary | text }

- When `Option Compare` is not used in a module, the default comparison method is Binary.

- When `Option Compare` is used, it must appear at the start of the module's declarations section, before any procedures.
- **Binary comparison** the default text comparison method in Visual Basic—uses the internal binary code of each character to determine the sort order of the characters. For example, "A" < "a".
- **Text comparison** uses the locale settings of the current system to determine the sort order of the characters. Text comparison is case insensitive. For example, "A" = "a".

Option Infer Statement

Enables the use of local type inference in declaring variables.

Syntax

`Option Infer { On | Off }`

When you set `Option Infer` to `On`, you can declare local variables without explicitly stating a data type. The compiler infers the data type of a variable from the type of its initialization expression.

`Option Infer` is turned on. The variable in the declaration `Dim someVar = 2` is declared as an integer by type inference. The following screenshot shows IntelliSense when `Option Infer` is on:



In the following illustration, `Option Infer` is turned off. The variable in the declaration `Dim someVar = 2` is declared as an Object by type inference. In this example,



[!NOTE] When a variable is declared as an Object, the run-time type can change while the program is running. Visual Basic performs operations called *boxing* and *unboxing* to convert between an Object and a value type, which makes execution slower.

Type Checking Function:

VB.NET provides number of data verification or data type checking function as below:

IsDate()

- Returns True if the value of variable is date value; Otherwise, it returns False

```
Dim MyVar, MyCheck
```

```
MyVar = "04/28/2014" Assign valid date value.
```

```
MyCheck = IsDate(MyVar) ' Returns True.
```

```
MsgBox(MyCheck)
```

```
MyVar = "April 28, 2014" Assign valid date value.
```

```
MyCheck = IsDate(MyVar) ' Returns True.
```

```
MsgBox(MyCheck)
```

```
MyVar = "13/32/2014" Assign invalid date value.
```

```
MyCheck = IsDate(MyVar) ' Returns False.
```

```
MsgBox(MyCheck)
```

```
MyVar = "04.28.14" Assign valid time value.
```

```
MyCheck = IsDate(MyVar) ' Returns True.
```

```
MsgBox(MyCheck)
```

```
MyVar = "04.28.2014" Assign invalid time value.
```

```
MyCheck = IsDate(MyVar) ' Returns False.
```

```
MsgBox(MyCheck)
```

IsNothing()

- Returns True if the object variable that currently has no assigned value; Otherwise, it returns False

```
Dim objtemp As Object
```

```
Dim bolans As Boolean
```

```
bolans = IsNothing(objtemp)
```

```
MsgBox(bolans) ' return true
```

```
objtemp = "dolly"
```

```
bolans = IsNothing(objtemp)
```

```
MsgBox(bolans) ' return false
```

IsNumeric()

- Returns True if the value is numeric; Otherwise returns false

```
Dim MyVar, MyCheck
MyVar = "53" Assign value.
MyCheck = IsNumeric(MyVar) ' Returns True.
MsgBox(MyCheck)
```

```
MyVar = "459.95" Assign value.
MyCheck = IsNumeric(MyVar) ' Returns True.
MsgBox(MyCheck)
```

```
MyVar = "45 Help" Assign value.
MyCheck = IsNumeric(MyVar) ' Returns False.
MsgBox(MyCheck)
```

IsArray()

- Tests whether an object variable points to an array

```
Dim s() As Integer = {1, 2}
Dim t As Object
t = s
MsgBox(IsArray(t)) ' return true
```

```
Dim strArr() As String
Console.WriteLine(IsArray(strArr)) ' return false An uninitialized array
```

Enumerations

When you are in a situation to have a number of constants that are logically related to each other, you can define them together these constants in an enumerator list. An enumerated type is declared using the enum keyword.

Syntax:

```
Enum enumerationname [ As datatype ]
    memberlist
End Enum
```

Enum declaration:

```
Enum dayaction As Integer
```

```
awake = 0
asleep = 1
coding = 2
```

```
End Enum
```

An enumeration has a name, an underlying data type, and a set of members. Each member represents a constant. It is useful when you have a set of values that are functionally significant and fixed.

Retrieve and check the Enum value:

```
Private action As dayaction = 2
```

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Button1.Click
```

```
MessageBox.Show(action.ToString())
End Sub
```

String Function:

1. The Len Function

The length function is used to find out the number of characters in any given string.

Syntax

```
Len(string)
```

2. The Mid function

The mid function is used to Return a substring containing a specified number of characters from a string.

Syntax

```
Mid (string, start[, length])
```

string - String expression from which characters are returned.

start - Long. Character position in string at which the part to be taken begins.

length - Length is Optional. Number of characters to return.

3. The Left Function

The Left function extract the left portion of a string.

Syntax

```
Left("string", n)
```

4. The Right Function

The Right function extract the right portion of a string.

SyntaxRight("string", n)

5. The Replace Function

The replace function is used to replacing some text in a string with some other text.

Syntax

```
Replace( string, searchtext, replacetext )
```

6. The Trim function

The trim function trims the empty spaces on both side of the String.

Syntax

```
Trim ("String")
```

7. The Ltrim Function

The Ltrim function trims the empty spaces of the left portion of the string.

Syntax

```
Ltrim("string")
```

8. The Rtrim Function

The Rtrim function trims the empty spaces of the Right portion of the string.

Syntax

```
Rtrim("string")
```

9. The Ucase and the Lcase Functions

The Ucase function converts all the characters of a string to capital letters. On the other hand, the Lcase function converts all the characters of a string to small letters.

Syntax

Ucase("string")

Lcase("string")

EXAMPLE

```
Module Module1
```

```
Sub Main()
```

```
Dim leng As String = Len(" Rohatashkumar")
```

```
Console.WriteLine("length is :" &leng)
```

```
Dim middle As String = Mid("Rohatash Kumar", 3, 4)
```

```
Console.WriteLine("Mid is :" & middle)
```

```
Dim leftf As String = Left("Rohatash Kumar", 3)
```

```
Console.WriteLine("Left is:" &leftf)
```

```
Dim right As String = Right("rohatashkumar", 6)
```

```
Console.WriteLine("Right is :" &right)
```

```
Dim spaces As String = Right("rohatashkumar", 7)
```

```
Console.WriteLine("Space is :" & spaces)
```

```
Dim replaces As String = Replace("rohatashkumar", "hat", "nmo")
```

```
Console.WriteLine("Replace is :" & replaces)
```

```
Dim trimt As String = Trim(" rohatashkumar ")
```

```
Console.WriteLine("Trim is :" &trimt)
```

```
Dim ltriml As String = LTrim(" rohatashkumar ")
```

```
Console.WriteLine("ltrim is :" &ltriml)
```

```
Dim rtrimr As String = RTrim(" rohatashkumar ")
```

```
Console.WriteLine("rtrim is :" &rtrimr)
```

```
Dim ucaseu As String = UCase("rohatashkumar")
```

```
Console.WriteLine("Ucase is :" &ucaseu)
```

```
Dim lcaseL As String = LCase("ROHATASH KUMAR")
```

```
Console.WriteLine("Ucase is :" &lcaseL)
```

```
End Sub
```

```
End Module
```

OUTPUT

```
length is :15
Mid is :hata
Left is:Roh
Right is : kumar
Space is :h kumar
Replace is :ronmoash kumar
Trim is :rohatash kumar
ltrim is :rohatash kumar
rtrim is :   rohatash kumar
Ucase is :ROHATASH KUMAR
Ucase is :rohatash kumar
Press any key to continue . . .
```

Date Time Class

- VB.NET's *DateTime* structure represents an instant in time and is usually expressed as a particular date and time of the day.
- **Properties**
 - Date: returns the date component of the DateTime value.
 - Day: returns the day of the month component of the DateTime value.
 - DayOfWeek: returns the day of the week component of the DateTime value.
 - DayOfYear: returns the day of the year component of the DateTime value.
 - Hour: returns the hour component of the DateTime value.
 - Millisecond: returns the milliseconds component of the DateTime value.
 - Minute: returns the minute component of the DateTime value.
 - Month: returns the month component of the DateTime value.
 - Now: returns a DateTime value that is the current local date and time on this computer.
 - Second: returns the seconds component of the DateTime value.
 - TimeOfDay: returns the time of day of the DateTime value.
 - Today: returns the current system date.
 - Year: returns the year component of the DateTime value.
- **Methods**
 - Add: adds the value of the specified TimeSpan to the DateTime value.
 - AddDays: adds the specified number of days to the DateTime value.
 - AddHours: adds the specified number of hours to the DateTime value.
 - AddMilliseconds: adds the specified number of milliseconds to the DateTime value.
 - AddMinutes: adds the specified number of minutes to the DateTime value.
 - AddMonths: adds the specified number of months to the DateTime value.
 - AddSeconds: adds the specified number of seconds to the DateTime value.
 - AddYears: adds the specified number of years to the DateTime value.
 - DaysInMonth: returns the number of days in the specified month of the specified year.

- **IsLeapYear**: returns an indication of whether the specified year is a leap year.
- **Subtract**: subtracts the specified time or duration from the **DateTime** value.
- **ToLocalTime**: converts the current Coordinated Universal Time (UTC) to local time.
- **ToLongDateString**: converts the value of this instance to its equivalent long date string representation.
- **ToLongTimeString**: converts the value of this instance to its equivalent long time string representation.
- **ToShortDateString**: converts the value of this instance to its equivalent short date string representation.
- **ToShortTimeString**: converts the value of this instance to its equivalent short time string representation.

Date Functions

DateAdd()

It is used to returns a date with a date,time value added with a specified time intreval.

Syntax: **DateAdd(interval, number, date)**

A date is added after an intreval of 10 days to the current date value.

DateDiff()

It is used to return a long value specifying the number of time intrevals between the specified date values.

Syntax: **Datediff(interval,date1,date2)**

The time intrevals between two same dates of different years are found using the '**DateDiff()**'.

DatePart()

returns an integer value containing the specified component of the Date value.

Syntax: **DatePart(interval,date)**

the date value entered is in the **mm:dd:yy** format

DateSerial()

returns an date value for the specified year, month and day with the time set to the midnight.

If the month value is '0' or '-1' the month december or november of the previous year is taken.

If the month value is '1', january month of the calculated year is taken, if '13' january of the following year is taken.

If the Day value is '1' refers to the first day of the calculated month, '0' for the last day of previous month, '-1' the penultimate day of the previous month.

Syntax: **DateSerial(Year,Month,Day)**

DateValue()

returns an date value containing the date information as a string

Syntax: DateValue(Date)

the date information alone is displayed as a String using the **DateValue** function

IsDate()

checks if the given expression is a valid date and returns a boolean true or false.

Syntax: IsDate(Expression)

If the date given as the argument is valid, the **IsDate** function returns **True**.

Today()

Return today's date

Syntax: Today()

Now()

Return today's date with time

Syntax: now()

Month()

returns the month of the year as an integer value in the range of 1-12.

Syntax: Month(Date)

the month for the given date is returned using the Month function.

TimeSerial()

returns an date value with the specified hour, minute, second with the date information is set to January 1 of year 1.

Syntax: Timeserial(hour, minute, second)

Format()

Returns a string formatted according to instructions contained in a format string expression.

Syntax: format(date,format)

Example:

DateAdd

```
MsgBox("10 Days after the current date is::" &DateAdd(DateInterval.Day, 10, Now))
```

DateDiff

```
Dim d1 As Date = #2/4/2009#
```

```
Dim d2 As Date = #2/4/2010#
```

```
Dim res As Long
res = DateDiff(DateInterval.Day, d1, d2)
MsgBox("The number of time intervals between the dates is::" & res)
```

DatePart

```
MsgBox("The date part of '01/10/2010' is::" & DatePart("d", "01/10/2010"))
```

DateSerial

```
Dim a As Date
a = DateSerial(2010, 2, 21)
MsgBox(a)
```

DateValue

```
MsgBox("Date information as a String is::"&DateValue("5/10/2010 12:00:01 AM"))
```

TimeSerial

```
MsgBox("Time displayed using Timeserial() is:: " & TimeSerial(4, 30, 23))
```

isdate

```
Dim curdat As Date
curdat = "5/31/2010"
MsgBox("Is '5/31/2010' a valid date::" & IsDate(curdat))
```

Today

```
Dim curdat As Date
curdat = Today()
MsgBox(curdat)
```

now

```
Dim curdat As Date
curdat = Now()
MsgBox(curdat)
```

Month

```
Dim dat As Date
dat = "6/23/2010"
```

```
MsgBox("Month value of the given date is::" & Month(dat))
```

Format

```
MsgBox(Format(Now, "M-d-yy"))  
MsgBox(Format(Now, "MM-dd-yyyy"))  
MsgBox(Format(Now, "MMMM-d-yyy- dddd"))  
MsgBox(Format(Now, "hh:mm:ss"))
```

Design time and Run time

- The time during which you build an application in the development environment by adding controls, setting controls or form properties, and so on. For ex. Setting the password char property to textbox (*) design time
- The time, during which code is running. during run time, you can't edit the code

With...End With

Executes a series of statements that repeatedly refer to a single object so that the statements can use a simplified syntax when accessing members of the object.

To make this type more efficient and easier to read, we use this block.

The use of it do not require calling again and again the name of the object

It allows us to set multiple properties and methods quickly and easily

Remember this is not a type of loop

Syntax	Example (on button click)
With object name [statements] End With	With Button1 .Text = "Click ME" .ForeColor = Color.Aqua .BackColor = Color.Yellow .Height = 50 .Width = 100 EndWith

Procedure

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

- Sub procedures or Subs
- Functions

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

1. Sub procedures or Subs or Sub Routine

- A Sub procedure is a series of Visual Basic statements enclosed by the Sub and End Sub statements. The Sub procedure performs a task and then returns control to the calling code, but it does not return a value to the calling code.
- Each time the procedure is called, its statements are executed, starting with the first executable statement after the Sub statement and ending with the first End Sub, Exit Sub, or Return statement encountered.
- You can define a Sub procedure in modules, classes, and structures. By default, it is `Public`, which means you can call it from anywhere in your application that has access to the module, class, or structure in which you defined it

A Sub procedure can take arguments, such as constants, variables, or expressions, which are passed to it by the calling code.

The **Sub** statement is used to declare the name, parameter and the body of a sub procedure. The syntax for the Sub statement is –

```
[Modifiers] Sub SubName [(ParameterList)]  
    [Statements]  
End Sub
```

Where,

- **Modifiers** – specify the access level of the procedure; possible values are - Public, Private, Protected, Friend, Protected Friend and information regarding overloading, overriding, sharing, and shadowing.
- **SubName** – indicates the name of the Sub
- **ParameterList** – specifies the list of the parameters

<code>Sub tellOperator(ByVal task As String) Dim stamp AsDate stamp = TimeOfDay()</code>	<code>tellOperator("file update")</code>
--------------------------------------------------------------------------------------------------	------------------------------------------

<pre>MsgBox("Starting "& task &" at "&CStr(stamp)) EndSub</pre>	
-----------------------------------------------------------------------------	--

2. Functions

- A Function procedure is a series of Visual Basic statements enclosed by the Function and End Function statements. The Function procedure performs a task and then returns control to the calling code. When it returns control, it also returns a value to the calling code.
- Each time the procedure is called, its statements run, starting with the first executable statement after the Function statement and ending with the first End Function, Exit Function, or Return statement encountered.
- You can define a Function procedure in a module, class, or structure. It is Public by default, which means you can call it from anywhere in your application that has access to the module, class, or structure in which you defined it.
- A Function procedure can take arguments, such as constants, variables, or expressions, which are passed to it by the calling code.

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

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

Where,

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

<pre>Function FindMax(ByVal num1 As Integer, ByVal num2 As Integer) As Integer 'local variable declaration */ Dim result As Integer If (num1 > num2) Then result = num1 Else</pre>	<p>Calling function</p> <pre>MsgBox("the maximum number is "&FindMax(10, 5))</pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------

<pre> result = num2 EndIf FindMax = result EndFunction </pre>	
---------------------------------------------------------------	--

Optional argument

<pre> Sub notify(ByVal company AsString, OptionalByValdesgAsString = "manager") Ifdesg = "manager"Then MsgBox("i am manager") Else MsgBox("i am not manager") EndIf EndSub </pre>	<pre> Call notify("abc") Call notify("abc", "worker") </pre>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------

Function overloading

Function overloading is where two or more functions can have the same name but different parameters.

Function overloading can be considered as an example of compile time polymorphism feature in Oop's.

<pre> PublicOverloadsFunction add(ByVal a AsInteger, ByVal b AsInteger) MsgBox("You are in function add(a,b)") Return a + b EndFunction PublicOverloadsFunction add(ByVal a AsInteger, ByVal b AsInteger, ByVal c AsInteger) MsgBox("You are in function add(a, b, c)") Return a + b + c EndFunction </pre>	<pre> MsgBox(add(4, 2)) MsgBox(add(4, 5, 1)) </pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------

ByVal&ByRef Methods for passing arguments to Subroutine & Functions

- The **ByVal** keyword indicates that an argument is passed in such a way that the called procedure or property cannot change the value of a variable passed as the argument in the calling code.
- The **ByRef** keyword indicates that an argument is passed in such a way that the called procedure can change the value of a variable passed as the argument in the calling code.

Dim value AsInteger = 1	
Pass by value Sub Example1(ByVal test AsInteger) test = 10 EndSub	Pass by reference Sub Example2(ByRef test AsInteger) test = 10 EndSub
' The integer value doesn't change here when passed ByVal. Example1(value) MsgBox("by val "& value)	' The integer value DOES change when passed ByRef. Example2(value) MsgBox("by ref "& value)
Output : 1	Output :10