S.N	Method Name & Description
1	**Public Shared Function Compare (strA As String, strB As String) As Integer** Compares two specified string objects and returns an integer that indicates their relative position in the sort order.
2	**Public Shared Function Compare (strA As String, strB As String, ignoreCase As Boolean) As Integer** Compares two specified string objects and returns an integer that indicates their relative position in the sort order. However, it ignores case if the Boolean parameter is true.
3	**Public Shared Function Concat (str0 As String, str1 As String) As String** Concatenates two string objects.
4	**Public Shared Function Concat (str0 As String, str1 As String, str2 As String) As String** Concatenates three string objects.
5	**Public Shared Function Concat (str0 As String, str1 As String, str2 As String, str3 As String) As String** Concatenates four string objects.
6	**Public Function Contains (value As String) As Boolean** Returns a value indicating whether the specified string object occurs within this string.
7	**Public Shared Function Copy (str As String) As String** Creates a new String object with the same value as the specified string.
8	**pPublic Sub CopyTo (sourceIndex As Integer, destination As Char(), destinationIndex As Integer, count As Integer)**

	Copies a specified number of characters from a specified position of the string object to a specified position in an array of Unicode characters.
9	**Public Function EndsWith (value As String) As Boolean** Determines whether the end of the string object matches the specified string.
10	**Public Function Equals (value As String) As Boolean** Determines whether the current string object and the specified string object have the same value.
11	**Public Shared Function Equals (a As String, b As String) As Boolean** Determines whether two specified string objects have the same value.
12	**Public Shared Function Format (format As String, arg0 As Object) As String** Replaces one or more format items in a specified string with the string representation of a specified object.
13	**Public Function IndexOf (value As Char) As Integer** Returns the zero-based index of the first occurrence of the specified Unicode character in the current string.
14	**Public Function IndexOf (value As String) As Integer** Returns the zero-based index of the first occurrence of the specified string in this instance.
15	**Public Function IndexOf (value As Char, startIndex As Integer) As Integer** Returns the zero-based index of the first occurrence of the specified Unicode character in this string, starting search at the specified character position.

16	**Public Function IndexOf (value As String, startIndex As Integer) As Integer** Returns the zero-based index of the first occurrence of the specified string in this instance, starting search at the specified character position.
17	**Public Function IndexOfAny (anyOf As Char()) As Integer** Returns the zero-based index of the first occurrence in this instance of any character in a specified array of Unicode characters.
18	**Public Function IndexOfAny (anyOf As Char(), startIndex As Integer) As Integer** Returns the zero-based index of the first occurrence in this instance of any character in a specified array of Unicode characters, starting search at the specified character position.
19	**Public Function Insert (startIndex As Integer, value As String) As String** Returns a new string in which a specified string is inserted at a specified index position in the current string object.
20	**Public Shared Function IsNullOrEmpty (value As String) As Boolean** Indicates whether the specified string is null or an Empty string.
21	**Public Shared Function Join (separator As String, ParamArray value As String()) As String** Concatenates all the elements of a string array, using the specified separator between each element.
22	**Public Shared Function Join (separator As String, value As String(), startIndex As Integer, count As Integer) As String** Concatenates the specified elements of a string array, using the specified separator between each element.

23	**Public Function LastIndexOf (value As Char) As Integer** Returns the zero-based index position of the last occurrence of the specified Unicode character within the current string object.
24	**Public Function LastIndexOf (value As String) As Integer** Returns the zero-based index position of the last occurrence of a specified string within the current string object.
25	**Public Function Remove (startIndex As Integer) As String** Removes all the characters in the current instance, beginning at a specified position and continuing through the last position, and returns the string.
26	**Public Function Remove (startIndex As Integer, count As Integer) As String** Removes the specified number of characters in the current string beginning at a specified position and returns the string.
27	**Public Function Replace (oldChar As Char, newChar As Char) As String** Replaces all occurrences of a specified Unicode character in the current string object with the specified Unicode character and returns the new string.
28	**Public Function Replace (oldValue As String, newValue As String) As String** Replaces all occurrences of a specified string in the current string object with the specified string and returns the new string.
29	**Public Function Split (ParamArray separator As Char()) As String()** Returns a string array that contains the substrings in the current string object, delimited by elements of a specified Unicode character array.

30	**Public Function Split (separator As Char(), count As Integer) As String()** Returns a string array that contains the substrings in the current string object, delimited by elements of a specified Unicode character array. The int parameter specifies the maximum number of substrings to return.
31	**Public Function StartsWith (value As String) As Boolean** Determines whether the beginning of this string instance matches the specified string.
32	**Public Function ToCharArray As Char()** Returns a Unicode character array with all the characters in the current string object.
33	**Public Function ToCharArray (startIndex As Integer, length As Integer) As Char()** Returns a Unicode character array with all the characters in the current string object, starting from the specified index and up to the specified length.
34	**Public Function ToLower As String** Returns a copy of this string converted to lowercase.
35	**Public Function ToUpper As String** Returns a copy of this string converted to uppercase.
36	**Public Function Trim As String** Removes all leading and trailing white-space characters from the current String object.

The above list of methods is not exhaustive, please visit MSDN library for the complete list of methods and String class constructors.

Examples

The following example demonstrates some of the methods mentioned above:

Comparing Strings

```
#include <include.h>
Module strings
   Sub Main()
      Dim str1, str2 As String
      str1 = "This is test"
      str2 = "This is text"
      If (String.Compare(str1, str2) = 0) Then
          Console.WriteLine(str1 + " and " + str2 +
                            " are equal.")
      Else
          Console.WriteLine(str1 + " and " + str2 +
                            " are not equal.")
      End If
      Console.ReadLine()
   End Sub
End Module
```

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

```
This is test and This is text are not equal.
```

String Contains String

```
Module strings
   Sub Main()
      Dim str1 As String
      str1 = "This is test"
      If (str1.Contains("test")) Then
          Console.WriteLine("The sequence 'test' was found.")
      End If
      Console.ReadLine()
   End Sub
```

```
End Module
```

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

```
The sequence 'test' was found.
```

Getting a Substring

```
Module strings
   Sub Main()
      Dim str As String
      str = "Last night I dreamt of San Pedro"
      Console.WriteLine(str)
      Dim substr As String = str.Substring(23)
      Console.WriteLine(substr)
      Console.ReadLine()
   End Sub
End Module
```

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

```
Last night I dreamt of San Pedro
San Pedro.
```

Joining Strings

```
Module strings
   Sub Main()
      Dim strarray As String() = {"Down the way where the nights are
gay",
                           "And the sun shines daily on the mountain
top",
                            "I took a trip on a sailing ship",
                           "And when I reached Jamaica",
                           "I made a stop"}
      Dim str As String = String.Join(vbCrLf, strarray)
      Console.WriteLine(str)
      Console.ReadLine()
```

```
   End Sub
End Module
```

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

```
Down the way where the nights are gay
And the sun shines daily on the mountain top
I took a trip on a sailing ship
And when I reached Jamaica
I made a stop
```

15. Date & Time

Most of the softwares you write need implementing some form of date functions returning current date and time. Dates are so much part of everyday life that it becomes easy to work with them without thinking. VB.Net also provides powerful tools for date arithmetic that makes manipulating dates easy.

The **Date** data type contains date values, time values, or date and time values. The default value of Date is 0:00:00 (midnight) on January 1, 0001. The equivalent .NET data type is **System.DateTime**.

The **DateTime** structure represents an instant in time, typically expressed as a date and time of day

```
'Declaration
<SerializableAttribute> _
Public Structure DateTime _
     Implements IComparable, IFormattable, IConvertible, ISerializable,
     IComparable(Of DateTime), IEquatable(Of DateTime)
```

You can also get the current date and time from the DateAndTime class.

The **DateAndTime** module contains the procedures and properties used in date and time operations.

```
'Declaration
<StandardModuleAttribute> _
Public NotInheritable Class DateAndTime
```

Note:

Both the DateTime structure and the DateAndTime module contain properties like **Now** and **Today**, so often beginners find it confusing. The DateAndTime class belongs to the Microsoft.VisualBasic namespace and the DateTime structure belongs to the System namespace. Therefore, using the later would help you in porting your code to another .Net language like C#. However, the DateAndTime class/module contains all the legacy date functions available in Visual Basic.

Properties and Methods of the DateTime Structure

The following table lists some of the commonly used **properties** of the **DateTime** Structure:

S.N	Property	Description
1	**Date**	Gets the date component of this instance.
2	**Day**	Gets the day of the month represented by this instance.
3	**DayOfWeek**	Gets the day of the week represented by this instance.
4	**DayOfYear**	Gets the day of the year represented by this instance.
5	**Hour**	Gets the hour component of the date represented by this instance.
6	**Kind**	Gets a value that indicates whether the time represented by this instance is based on local time, Coordinated Universal Time (UTC), or neither.
7	**Millisecond**	Gets the milliseconds component of the date represented by this instance.
8	**Minute**	Gets the minute component of the date represented by this instance.
9	**Month**	Gets the month component of the date represented by this instance.
10	**Now**	Gets a **DateTime** object that is set to the current date and time on this computer, expressed as the local time.

11	**Second**	Gets the seconds component of the date represented by this instance.
12	**Ticks**	Gets the number of ticks that represent the date and time of this instance.
13	**TimeOfDay**	Gets the time of day for this instance.
14	**Today**	Gets the current date.
15	**UtcNow**	Gets a **DateTime** object that is set to the current date and time on this computer, expressed as the Coordinated Universal Time (UTC).
16	**Year**	Gets the year component of the date represented by this instance.

The following table lists some of the commonly used **methods** of the **DateTime** structure:

S.N	Method Name & Description
1	**Public Function Add (value As TimeSpan) As DateTime** Returns a new DateTime that adds the value of the specified TimeSpan to the value of this instance.
2	**Public Function AddDays (value As Double) As DateTime** Returns a new DateTime that adds the specified number of days to the value of this instance.
3	**Public Function AddHours (value As Double) As DateTime** Returns a new DateTime that adds the specified number of hours to the value of this instance.

4	**Public Function AddMinutes (value As Double) As DateTime** Returns a new DateTime that adds the specified number of minutes to the value of this instance.
5	**Public Function AddMonths (months As Integer) As DateTime** Returns a new DateTime that adds the specified number of months to the value of this instance.
6	**Public Function AddSeconds (value As Double) As DateTime** Returns a new DateTime that adds the specified number of seconds to the value of this instance.
7	**Public Function AddYears (value As Integer) As DateTime** Returns a new DateTime that adds the specified number of years to the value of this instance.
8	**Public Shared Function Compare (t1 As DateTime,t2 As DateTime) As Integer** Compares two instances of DateTime and returns an integer that indicates whether the first instance is earlier than, the same as, or later than the second instance.
9	**Public Function CompareTo (value As DateTime) As Integer** Compares the value of this instance to a specified DateTime value and returns an integer that indicates whether this instance is earlier than, the same as, or later than the specified DateTime value.
10	**Public Function Equals (value As DateTime) As Boolean** Returns a value indicating whether the value of this instance is equal to the value of the specified DateTime instance.
11	**Public Shared Function Equals (t1 As DateTime, t2 As DateTime) As Boolean**

	Returns a value indicating whether two DateTime instances have the same date and time value.
12	**Public Overrides Function ToString As String** Converts the value of the current DateTime object to its equivalent string representation.

The above list of methods is not exhaustive, please visit Microsoft documentation for the complete list of methods and properties of the DateTime structure.

Creating a DateTime Object

You can create a DateTime object in one of the following ways:

- By calling a DateTime constructor from any of the overloaded DateTime constructors.
- By assigning the DateTime object a date and time value returned by a property or method.
- By parsing the string representation of a date and time value.
- By calling the DateTime structure's implicit default constructor.

The following example demonstrates this:

```
Module Module1
   Sub Main()
      'DateTime constructor: parameters year, month, day, hour, min, sec
      Dim date1 As New Date(2012, 12, 16, 12, 0, 0)
      'initializes a new DateTime value
      Dim date2 As Date = #12/16/2012 12:00:52 AM#
      'using properties
      Dim date3 As Date = Date.Now
      Dim date4 As Date = Date.UtcNow
      Dim date5 As Date = Date.Today
      Console.WriteLine(date1)
      Console.WriteLine(date2)
      Console.WriteLine(date3)
```

```
      Console.WriteLine(date4)
      Console.WriteLine(date5)
      Console.ReadKey()
   End Sub
End Module
```

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

```
12/16/2012 12:00:00 PM
12/16/2012 12:00:52 PM
12/12/2012 10:22:50 PM
12/12/2012 12:00:00 PM
```

Getting the Current Date and Time

The following programs demonstrate how to get the current date and time in VB.Net:

Current Time

```
Module dateNtime
   Sub Main()
      Console.Write("Current Time: ")
      Console.WriteLine(Now.ToLongTimeString)
      Console.ReadKey()
   End Sub
End Module
```

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

```
Current Time: 11 :05 :32 AM
```

Current Date

```
Module dateNtime
   Sub Main()
      Console.WriteLine("Current Date: ")
      Dim dt As Date = Today
      Console.WriteLine("Today is: {0}", dt)
```

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

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

```
Today is: 12/11/2012 12:00:00 AM
```

Formatting Date

A Date literal should be enclosed within hash signs (# #), and specified in the format M/d/yyyy, for example #12/16/2012#. Otherwise, your code may change depending on the locale in which your application is running.

For example, you specified Date literal of #2/6/2012# for the date February 6, 2012. It is alright for the locale that uses mm/dd/yyyy format. However, in a locale that uses dd/mm/yyyy format, your literal would compile to June 2, 2012. If a locale uses another format say, yyyy/mm/dd, the literal would be invalid and cause a compiler error.

To convert a Date literal to the format of your locale or to a custom format, use the **Format** function of String class, specifying either a predefined or user-defined date format.

The following example demonstrates this.

```
Module dateNtime
   Sub Main()
      Console.WriteLine("India Wins Freedom: ")
      Dim independenceDay As New Date(1947, 8, 15, 0, 0, 0)
      ' Use format specifiers to control the date display.
      Console.WriteLine(" Format 'd:' " & independenceDay.ToString("d"))
      Console.WriteLine(" Format 'D:' " & independenceDay.ToString("D"))
      Console.WriteLine(" Format 't:' " & independenceDay.ToString("t"))
      Console.WriteLine(" Format 'T:' " & independenceDay.ToString("T"))
      Console.WriteLine(" Format 'f:' " & independenceDay.ToString("f"))
      Console.WriteLine(" Format 'F:' " & independenceDay.ToString("F"))
      Console.WriteLine(" Format 'g:' " & independenceDay.ToString("g"))
      Console.WriteLine(" Format 'G:' " & independenceDay.ToString("G"))
      Console.WriteLine(" Format 'M:' " & independenceDay.ToString("M"))
      Console.WriteLine(" Format 'R:' " & independenceDay.ToString("R"))
```

```
      Console.WriteLine(" Format 'y:' " & independenceDay.ToString("y"))
      Console.ReadKey()
   End Sub
End Module
```

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

```
India Wins Freedom:
Format 'd:' 8/15/1947
Format 'D:' Friday, August 15, 1947
Format 't:' 12:00 AM
Format 'T:' 12:00:00 AM
Format 'f:' Friday, August 15, 1947 12:00 AM
Format 'F:' Friday, August 15, 1947 12:00:00 AM
Format 'g:' 8/15/1947 12:00 AM
Format 'G:' 8/15/1947 12:00:00 AM
Format 'M:' 8/15/1947 August 15
Format 'R:' Fri, 15 August 1947 00:00:00 GMT
Format 'y:' August, 1947
```

Predefined Date/Time Formats

The following table identifies the predefined date and time format names. These may be used by name as the style argument for the **Format** function:

Format	Description
General Date, or G	Displays a date and/or time. For example, 1/12/2012 07:07:30 AM.
Long Date, Medium Date, or D	Displays a date according to your current culture's long date format. For example, Sunday, December 16, 2012.
Short Date, or d	Displays a date using your current culture's short date format. For example, 12/12/2012.

Long Time, Medium Time, or T	Displays a time using your current culture's long time format; typically includes hours, minutes, seconds. For example, 01:07:30 AM.
Short Time or t	Displays a time using your current culture's short time format. For example, 11:07 AM.
F	Displays the long date and short time according to your current culture's format. For example, Sunday, December 16, 2012 12:15 AM.
F	Displays the long date and long time according to your current culture's format. For example, Sunday, December 16, 2012 12:15:31 AM.
G	Displays the short date and short time according to your current culture's format. For example, 12/16/2012 12:15 AM.
M, m	Displays the month and the day of a date. For example, December 16.
R, r	Formats the date according to the RFC1123Pattern property.
S	Formats the date and time as a sortable index. For example, 2012-12-16T12:07:31.
U	Formats the date and time as a GMT sortable index. For example, 2012-12-16 12:15:31Z.

U	Formats the date and time with the long date and long time as GMT. For example, Sunday, December 16, 2012 6:07:31 PM.
Y, y	Formats the date as the year and month. For example, December, 2012.

For other formats like user-defined formats, please consult Microsoft Documentation.

Properties and Methods of the DateAndTime Class

The following table lists some of the commonly used **properties** of the **DateAndTime** Class:

S.N	Property	Description
1	**Date**	Returns or sets a String value representing the current date according to your system.
2	**Now**	Returns a Date value containing the current date and time according to your system.
3	**TimeOfDay**	Returns or sets a Date value containing the current time of day according to your system.
4	**Timer**	Returns a Double value representing the number of seconds elapsed since midnight.
5	**TimeString**	Returns or sets a String value representing the current time of day according to your system.
6	**Today**	Gets the current date.

The following table lists some of the commonly used **methods** of the **DateAndTime** class:

S.N	Method Name & Description

1	**Public Shared Function DateAdd (Interval As DateInterval, Number As Double, DateValue As DateTime) As DateTime** Returns a Date value containing a date and time value to which a specified time interval has been added.
2	**Public Shared Function DateAdd (Interval As String,Number As Double,DateValue As Object) As DateTime** Returns a Date value containing a date and time value to which a specified time interval has been added.
3	**Public Shared Function DateDiff (Interval As DateInterval, Date1 As DateTime, Date2 As DateTime, DayOfWeek As FirstDayOfWeek, WeekOfYear As FirstWeekOfYear) As Long** Returns a Long value specifying the number of time intervals between two Date values.
4	**Public Shared Function DatePart (Interval As DateInterval, DateValue As DateTime, FirstDayOfWeekValue As FirstDayOfWeek, FirstWeekOfYearValue As FirstWeekOfYear) As Integer** Returns an Integer value containing the specified component of a given Date value.
5	**Public Shared Function Day (DateValue As DateTime) As Integer** Returns an Integer value from 1 through 31 representing the day of the month.
6	**Public Shared Function Hour (TimeValue As DateTime) As Integer** Returns an Integer value from 0 through 23 representing the hour of the day.
7	**Public Shared Function Minute (TimeValue As DateTime) As Integer** Returns an Integer value from 0 through 59 representing the minute of the hour.

8	**Public Shared Function Month (DateValue As DateTime) As Integer** Returns an Integer value from 1 through 12 representing the month of the year.
9	**Public Shared Function MonthName (Month As Integer, Abbreviate As Boolean) As String** Returns a String value containing the name of the specified month.
10	**Public Shared Function Second (TimeValue As DateTime) As Integer** Returns an Integer value from 0 through 59 representing the second of the minute.
11	**Public Overridable Function ToString As String** Returns a string that represents the current object.
12	**Public Shared Function Weekday (DateValue As DateTime, DayOfWeek As FirstDayOfWeek) As Integer** Returns an Integer value containing a number representing the day of the week.
13	**Public Shared Function WeekdayName (Weekday As Integer, Abbreviate As Boolean, FirstDayOfWeekValue As FirstDayOfWeek) As String** Returns a String value containing the name of the specified weekday.
14	**Public Shared Function Year (DateValue As DateTime) As Integer** Returns an Integer value from 1 through 9999 representing the year.

The above list is not exhaustive. For complete list of properties and methods of the DateAndTime class, please consult Microsoft Documentation.

The following program demonstrates some of these and methods:

```
Module Module1
   Sub Main()

      Dim birthday As Date
      Dim bday As Integer
      Dim month As Integer
      Dim monthname As String
      ' Assign a date using standard short format.
      birthday = #7/27/1998#
      bday = Microsoft.VisualBasic.DateAndTime.Day(birthday)
      month = Microsoft.VisualBasic.DateAndTime.Month(birthday)
      monthname = Microsoft.VisualBasic.DateAndTime.MonthName(month)
      Console.WriteLine(birthday)
      Console.WriteLine(bday)
      Console.WriteLine(month)
      Console.WriteLine(monthname)
      Console.ReadKey()
   End Sub
End Module
```

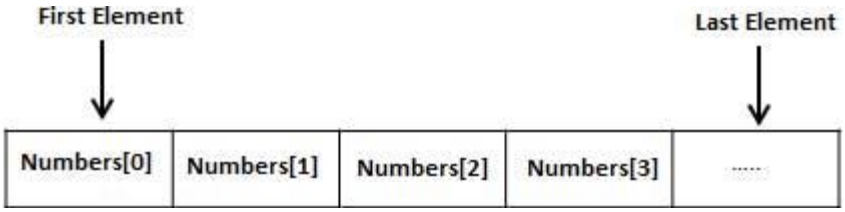
When the above code is compiled and executed, it produces the following result:

```
7/27/1998 12:00:00 AM
27
7
July
```

16. Arrays

An array stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Creating Arrays in VB.Net

To declare an array in VB.Net, you use the Dim statement. For example,

```
Dim intData(30)    ' an array of 31 elements
Dim strData(20) As String    ' an array of 21 strings
Dim twoDarray(10, 20) As Integer   'a two dimensional array of integers
Dim ranges(10, 100)     'a two dimensional array
```

You can also initialize the array elements while declaring the array. For example,

```
Dim intData() As Integer = {12, 16, 20, 24, 28, 32}
Dim names() As String = {"Karthik", "Sandhya", _
"Shivangi", "Ashwitha", "Somnath"}
Dim miscData() As Object = {"Hello World", 12d, 16ui, "A"c}
```

The elements in an array can be stored and accessed by using the index of the array. The following program demonstrates this:

```
Module arrayApl
   Sub Main()
      Dim n(10) As Integer  ' n is an array of 11 integers '
      Dim i, j As Integer
      ' initialize elements of array n '
      For i = 0 To 10
```

```
            n(i) = i + 100 ' set element at location i to i + 100
        Next i
        ' output each array element's value '
        For j = 0 To 10
            Console.WriteLine("Element({0}) = {1}", j, n(j))
        Next j
        Console.ReadKey()
    End Sub
End Module
```

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

```
Element(0) = 100
Element(1) = 101
Element(2) = 102
Element(3) = 103
Element(4) = 104
Element(5) = 105
Element(6) = 106
Element(7) = 107
Element(8) = 108
Element(9) = 109
Element(10) = 110
```

Dynamic Arrays

Dynamic arrays are arrays that can be dimensioned and re-dimensioned as par the need of the program. You can declare a dynamic array using the **ReDim** statement.

Syntax for ReDim statement:

```
ReDim [Preserve] arrayname(subscripts)
```

Where,

- The **Preserve** keyword helps to preserve the data in an existing array, when you resize it.

- **arrayname** is the name of the array to re-dimension.

- **subscripts** specifies the new dimension.

```
Module arrayApl
   Sub Main()
      Dim marks() As Integer
      ReDim marks(2)
      marks(0) = 85
      marks(1) = 75
      marks(2) = 90
      ReDim Preserve marks(10)
      marks(3) = 80
      marks(4) = 76
      marks(5) = 92
      marks(6) = 99
      marks(7) = 79
      marks(8) = 75
      For i = 0 To 10
          Console.WriteLine(i & vbTab & marks(i))
      Next i
      Console.ReadKey()
   End Sub
End Module
```

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

```
0    85
1    75
2    90
3    80
4    76
5    92
6    99
7    79
8    75
9    0
10   0
```

Multi-Dimensional Arrays

VB.Net allows multidimensional arrays. Multidimensional arrays are also called rectangular arrays.

You can declare a 2-dimensional array of strings as:

```
Dim twoDStringArray(10, 20) As String
```

or, a 3-dimensional array of Integer variables:

```
Dim threeDIntArray(10, 10, 10) As Integer
```

The following program demonstrates creating and using a 2-dimensional array:

```
Module arrayApl
   Sub Main()
      ' an array with 5 rows and 2 columns
      Dim a(,) As Integer = {{0, 0}, {1, 2}, {2, 4}, {3, 6}, {4, 8}}
      Dim i, j As Integer
      ' output each array element's value '
      For i = 0 To 4
          For j = 0 To 1
              Console.WriteLine("a[{0},{1}] = {2}", i, j, a(i, j))
          Next j
      Next i
      Console.ReadKey()
   End Sub
End Module
```

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

```
a[0,0]: 0
a[0,1]: 0
a[1,0]: 1
a[1,1]: 2
a[2,0]: 2
a[2,1]: 4
a[3,0]: 3
a[3,1]: 6
```

```
a[4,0]: 4
a[4,1]: 8
```

Jagged Array

A Jagged array is an array of arrays. The following code shows declaring a jagged array named *scores* of Integers:

```
Dim scores As Integer()() = New Integer(5)(){}
```

The following example illustrates using a jagged array:

```
Module arrayApl
   Sub Main()
      'a jagged array of 5 array of integers
      Dim a As Integer()() = New Integer(4)() {}
      a(0) = New Integer() {0, 0}
      a(1) = New Integer() {1, 2}
      a(2) = New Integer() {2, 4}
      a(3) = New Integer() {3, 6}
      a(4) = New Integer() {4, 8}
      Dim i, j As Integer
      ' output each array element's value
      For i = 0 To 4
          For j = 0 To 1
              Console.WriteLine("a[{0},{1}] = {2}", i, j, a(i)(j))
          Next j
      Next i
      Console.ReadKey()
   End Sub
End Module
```

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

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
```

```
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

The Array Class

The Array class is the base class for all the arrays in VB.Net. It is defined in the System namespace. The Array class provides various properties and methods to work with arrays.

Properties of the Array Class

The following table provides some of the most commonly used **properties** of the **Array** class:

S.N	Property Name & Description
1	**IsFixedSize** Gets a value indicating whether the Array has a fixed size.
2	**IsReadOnly** Gets a value indicating whether the Array is read-only.
3	**Length** Gets a 32-bit integer that represents the total number of elements in all the dimensions of the Array.
4	**LongLength** Gets a 64-bit integer that represents the total number of elements in all the dimensions of the Array.
5	**Rank** Gets the rank (number of dimensions) of the Array.

Methods of the Array Class

The following table provides some of the most commonly used **methods** of the **Array** class:

S.N	Method Name & Description
1	**Public Shared Sub Clear (array As Array, index As Integer, length As Integer)** Sets a range of elements in the Array to zero, to false, or to null, depending on the element type.
2	**Public Shared Sub Copy (sourceArray As Array, destinationArray As Array, length As Integer)** Copies a range of elements from an Array starting at the first element and pastes them into another Array starting at the first element. The length is specified as a 32-bit integer.
3	**Public Sub CopyTo (array As Array, index As Integer)** Copies all the elements of the current one-dimensional Array to the specified one-dimensional Array starting at the specified destination Array index. The index is specified as a 32-bit integer.
4	**Public Function GetLength (dimension As Integer) As Integer** Gets a 32-bit integer that represents the number of elements in the specified dimension of the Array.
5	**Public Function GetLongLength (dimension As Integer) As Long** Gets a 64-bit integer that represents the number of elements in the specified dimension of the Array.
6	**Public Function GetLowerBound (dimension As Integer) As Integer** Gets the lower bound of the specified dimension in the Array.
7	**Public Function GetType As Type**

	Gets the Type of the current instance (Inherited from Object).
8	**Public Function GetUpperBound (dimension As Integer) As Integer** Gets the upper bound of the specified dimension in the Array.
9	**Public Function GetValue (index As Integer) As Object** Gets the value at the specified position in the one-dimensional Array. The index is specified as a 32-bit integer.
10	**Public Shared Function IndexOf (array As Array,value As Object) As Integer** Searches for the specified object and returns the index of the first occurrence within the entire one-dimensional Array.
11	**Public Shared Sub Reverse (array As Array)** Reverses the sequence of the elements in the entire one-dimensional Array.
12	**Public Sub SetValue (value As Object, index As Integer)** Sets a value to the element at the specified position in the one-dimensional Array. The index is specified as a 32-bit integer.
13	**Public Shared Sub Sort (array As Array)** Sorts the elements in an entire one-dimensional Array using the IComparable implementation of each element of the Array.
14	**Public Overridable Function ToString As String** Returns a string that represents the current object (Inherited from Object).

For a complete list of Array class properties and methods, please refer the Microsoft documentation.

Example

The following program demonstrates use of some of the methods of the Array class:

```
Module arrayApl
   Sub Main()
      Dim list As Integer() = {34, 72, 13, 44, 25, 30, 10}
      Dim temp As Integer() = list
      Dim i As Integer
      Console.Write("Original Array: ")
      For Each i In list
          Console.Write("{0} ", i)
      Next i
      Console.WriteLine()
      ' reverse the array
      Array.Reverse(temp)
      Console.Write("Reversed Array: ")
      For Each i In temp
          Console.Write("{0} ", i)
      Next i
      Console.WriteLine()
      'sort the array
      Array.Sort(list)
      Console.Write("Sorted Array: ")
      For Each i In list
          Console.Write("{0} ", i)
      Next i
      Console.WriteLine()
      Console.ReadKey()
   End Sub
End Module
```

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

```
Original Array: 34 72 13 44 25 30 10
Reversed Array: 10 30 25 44 13 72 34
Sorted Array: 10 13 25 30 34 44 72
```

17. Collections

Collection classes are specialized classes for data storage and retrieval. These classes provide support for stacks, queues, lists, and hash tables. Most collection classes implement the same interfaces.

Collection classes serve various purposes, such as allocating memory dynamically to elements and accessing a list of items on the basis of an index, etc. These classes create collections of objects of the Object class, which is the base class for all data types in VB.Net.

Various Collection Classes and Their Usage

The following are the various commonly used classes of the **System.Collection** namespace. Click the following links to check their details.

Class	Description and Usage
ArrayList	It represents ordered collection of an object that can be **indexed** individually. It is basically an alternative to an array. However, unlike array, you can add and remove items from a list at a specified position using an **index** and the array resizes itself automatically. It also allows dynamic memory allocation, add, search and sort items in the list.
Hashtable	It uses a **key** to access the elements in the collection. A hash table is used when you need to access elements by using key, and you can identify a useful key value. Each item in the hash table has a **key/value** pair. The key is used to access the items in the collection.
SortedList	It uses a **key** as well as an **index** to access the items in a list.

	A sorted list is a combination of an array and a hash table. It contains a list of items that can be accessed using a key or an index. If you access items using an index, it is an ArrayList, and if you access items using a key, it is a Hashtable. The collection of items is always sorted by the key value.
Stack	It represents a **last-in, first out** collection of object. It is used when you need a last-in, first-out access of items. When you add an item in the list, it is called **pushing** the item, and when you remove it, it is called **popping** the item.
Queue	It represents a **first-in, first out** collection of object. It is used when you need a first-in, first-out access of items. When you add an item in the list, it is called **enqueue**, and when you remove an item, it is called **deque**.
BitArray	It represents an array of the **binary representation** using the values 1 and 0. It is used when you need to store the bits but do not know the number of bits in advance. You can access items from the BitArray collection by using an **integer index**, which starts from zero.

ArrayList

It represents an ordered collection of an object that can be indexed individually. It is basically an alternative to an array. However, unlike array, you can add and remove items from a list at a specified position using an **index** and the array resizes itself automatically. It also allows dynamic memory allocation, adding, searching and sorting items in the list.

Properties and Methods of the ArrayList Class

The following table lists some of the commonly used **properties** of the **ArrayList** class:

Property	Description
Capacity	Gets or sets the number of elements that the ArrayList can contain.
Count	Gets the number of elements actually contained in the ArrayList.
IsFixedSize	Gets a value indicating whether the ArrayList has a fixed size.
IsReadOnly	Gets a value indicating whether the ArrayList is read-only.
Item	Gets or sets the element at the specified index.

The following table lists some of the commonly used **methods** of the **ArrayList** class:

S.N.	Method Name & Purpose
1	**Public Overridable Function Add (value As Object) As Integer** Adds an object to the end of the ArrayList.
2	**Public Overridable Sub AddRange (c As ICollection)** Adds the elements of an ICollection to the end of the ArrayList.
3	**Public Overridable Sub Clear** Removes all elements from the ArrayList.
4	**Public Overridable Function Contains (item As Object) As Boolean** Determines whether an element is in the ArrayList.

5	**Public Overridable Function GetRange (index As Integer, count As Integer) As ArrayList** Returns an ArrayList, which represents a subset of the elements in the source ArrayList.
6	**Public Overridable Function IndexOf (value As Object) As Integer** Returns the zero-based index of the first occurrence of a value in the ArrayList or in a portion of it.
7	**Public Overridable Sub Insert (index As Integer, value As Object)** Inserts an element into the ArrayList at the specified index.
8	**Public Overridable Sub InsertRange (index As Integer, c As ICollection)** Inserts the elements of a collection into the ArrayList at the specified index.
9	**Public Overridable Sub Remove (obj As Object)** Removes the first occurrence of a specific object from the ArrayList.
10	**Public Overridable Sub RemoveAt (index As Integer)** Removes the element at the specified index of the ArrayList.
11	**Public Overridable Sub RemoveRange (index As Integer, count As Integer)** Removes a range of elements from the ArrayList.
12	**Public Overridable Sub Reverse** Reverses the order of the elements in the ArrayList.
13	**Public Overridable Sub SetRange (index As Integer, c As ICollection)** Copies the elements of a collection over a range of elements in the ArrayList.

14	**Public Overridable Sub Sort** Sorts the elements in the ArrayList.
15	**Public Overridable Sub TrimToSize** Sets the capacity to the actual number of elements in the ArrayList.

Example

The following example demonstrates the concept:

```
Sub Main()
   Dim al As ArrayList = New ArrayList()
   Dim i As Integer
   Console.WriteLine("Adding some numbers:")
   al.Add(45)
   al.Add(78)
   al.Add(33)
   al.Add(56)
   al.Add(12)
   al.Add(23)
   al.Add(9)
   Console.WriteLine("Capacity: {0} ", al.Capacity)
   Console.WriteLine("Count: {0}", al.Count)
   Console.Write("Content: ")
   For Each i In al
       Console.Write("{0} ", i)
   Next i
   Console.WriteLine()
   Console.Write("Sorted Content: ")
   al.Sort()
   For Each i In al
       Console.Write("{0} ", i)
   Next i
   Console.WriteLine()
   Console.ReadKey()
```

```
   End Sub
End Module
```

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

```
Adding some numbers:
Capacity: 8
Count: 7
Content: 45 78 33 56 12 23 9
Content: 9 12 23 33 45 56 78
```

Hashtable

The Hashtable class represents a collection of **key-and-value pairs** that are organized based on the hash code of the key. It uses the key to access the elements in the collection.

A hashtable is used when you need to access elements by using **key**, and you can identify a useful key value. Each item in the hashtable has a key/value pair. The key is used to access the items in the collection.

Properties and Methods of the Hashtable Class

The following table lists some of the commonly used **properties** of the **Hashtable** class:

Property	Description
Count	Gets the number of key-and-value pairs contained in the Hashtable.
IsFixedSize	Gets a value indicating whether the Hashtable has a fixed size.
IsReadOnly	Gets a value indicating whether the Hashtable is read-only.
Item	Gets or sets the value associated with the specified key.
Keys	Gets an ICollection containing the keys in the Hashtable.
Values	Gets an ICollection containing the values in the Hashtable.

The following table lists some of the commonly used **methods** of the **Hashtable** class:

S.N	Method Name & Purpose
1	**Public Overridable Sub Add (key As Object, value As Object)** Adds an element with the specified key and value into the Hashtable.
2	**Public Overridable Sub Clear** Removes all elements from the Hashtable.
3	**Public Overridable Function ContainsKey (key As Object) As Boolean** Determines whether the Hashtable contains a specific key.
4	**Public Overridable Function ContainsValue (value As Object) As Boolean** Determines whether the Hashtable contains a specific value.
5	**Public Overridable Sub Remove (key As Object)** Removes the element with the specified key from the Hashtable.

Example

The following example demonstrates the concept:

```
Module collections
   Sub Main()
      Dim ht As Hashtable = New Hashtable()
      Dim k As String
      ht.Add("001", "Zara Ali")
      ht.Add("002", "Abida Rehman")
      ht.Add("003", "Joe Holzner")
      ht.Add("004", "Mausam Benazir Nur")
      ht.Add("005", "M. Amlan")
      ht.Add("006", "M. Arif")
      ht.Add("007", "Ritesh Saikia")
      If (ht.ContainsValue("Nuha Ali")) Then
          Console.WriteLine("This student name is already in the list")
```

```
      Else
          ht.Add("008", "Nuha Ali")
      End If
      ' Get a collection of the keys.
      Dim key As ICollection = ht.Keys
      For Each k In key
          Console.WriteLine(" {0} : {1}", k, ht(k))
      Next k
      Console.ReadKey()
   End Sub
End Module
```

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

```
006: M. Arif
007: Ritesh Saikia
008: Nuha Ali
003: Joe Holzner
002: Abida Rehman
004: Mausam Banazir Nur
001: Zara Ali
005: M. Amlan
```

SortedList

The SortedList class represents a collection of key-and-value pairs that are sorted by the keys and are accessible by key and by index.

A sorted list is a combination of an array and a hashtable. It contains a list of items that can be accessed using a key or an index. If you access items using an index, it is an ArrayList, and if you access items using a key, it is a Hashtable. The collection of items is always sorted by the key value.

Properties and Methods of the SortedList Class

The following table lists some of the commonly used **properties** of the **SortedList** class:

Property	Description
Capacity	Gets or sets the capacity of the SortedList.
Count	Gets the number of elements contained in the SortedList.
IsFixedSize	Gets a value indicating whether the SortedList has a fixed size.
IsReadOnly	Gets a value indicating whether the SortedList is read-only.
Item	Gets and sets the value associated with a specific key in the SortedList.
Keys	Gets the keys in the SortedList.
Values	Gets the values in the SortedList.

The following table lists some of the commonly used **methods** of the **SortedList** class:

S.N	Method Name & Purpose
1	**Public Overridable Sub Add (key As Object, value As Object)** Adds an element with the specified key and value into the SortedList.
2	**Public Overridable Sub Clear** Removes all elements from the SortedList.
3	**Public Overridable Function ContainsKey (key As Object) As Boolean** Determines whether the SortedList contains a specific key.
4	**Public Overridable Function ContainsValue (value As Object) As Boolean**

	Determines whether the SortedList contains a specific value.
5	**Public Overridable Function GetByIndex (index As Integer) As Object** Gets the value at the specified index of the SortedList.
6	**Public Overridable Function GetKey (index As Integer) As Object** Gets the key at the specified index of the SortedList.
7	**Public Overridable Function GetKeyList As IList** Gets the keys in the SortedList.
8	**Public Overridable Function GetValueList As IList** Gets the values in the SortedList.
9	**Public Overridable Function IndexOfKey (key As Object) As Integer** Returns the zero-based index of the specified key in the SortedList.
10	**Public Overridable Function IndexOfValue (value As Object) As Integer** Returns the zero-based index of the first occurrence of the specified value in the SortedList.
11	**Public Overridable Sub Remove (key As Object)** Removes the element with the specified key from the SortedList.
12	**Public Overridable Sub RemoveAt (index As Integer)** Removes the element at the specified index of SortedList.
13	**Public Overridable Sub TrimToSize** Sets the capacity to the actual number of elements in the SortedList.

Example

The following example demonstrates the concept:

```
Module collections
   Sub Main()
      Dim sl As SortedList = New SortedList()
      sl.Add("001", "Zara Ali")
      sl.Add("002", "Abida Rehman")
      sl.Add("003", "Joe Holzner")
      sl.Add("004", "Mausam Benazir Nur")
      sl.Add("005", "M. Amlan")
      sl.Add("006", "M. Arif")
      sl.Add("007", "Ritesh Saikia")
      If (sl.ContainsValue("Nuha Ali")) Then
          Console.WriteLine("This student name is already in the list")
      Else
          sl.Add("008", "Nuha Ali")
      End If
       ' Get a collection of the keys.
      Dim key As ICollection = sl.Keys
      Dim k As String
      For Each k In key
          Console.WriteLine(" {0} : {1}", k, sl(k))
      Next k
      Console.ReadKey()
   End Sub
End Module
```

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

```
001: Zara Ali
002: Abida Rehman
003: Joe Holzner
```

```
004: Mausam Banazir Nur
005: M. Amlan
006: M. Arif
007: Ritesh Saikia
008: Nuha Ali
```

Stack

It represents a last-in, first-out collection of objects. It is used when you need a last-in, first-out access of items. When you add an item in the list, it is called pushing the item, and when you remove it, it is called popping the item.

Properties and Methods of the Stack Class

The following table lists some of the commonly used **properties** of the **Stack** class:

Property	Description
Count	Gets the number of elements contained in the Stack.

The following table lists some of the commonly used **methods** of the **Stack** class:

S.N	Method Name & Purpose
1	**Public Overridable Sub Clear** Removes all elements from the Stack.
2	**Public Overridable Function Contains (obj As Object) As Boolean** Determines whether an element is in the Stack.
3	**Public Overridable Function Peek As Object** Returns the object at the top of the Stack without removing it.
4	**Public Overridable Function Pop As Object** Removes and returns the object at the top of the Stack.

5	**Public Overridable Sub Push (obj As Object)** Inserts an object at the top of the Stack.
6	**Public Overridable Function ToArray As Object()** Copies the Stack to a new array.

Example

The following example demonstrates use of Stack:

```
Module collections
   Sub Main()
      Dim st As Stack = New Stack()
      st.Push("A")
      st.Push("M")
      st.Push("G")
      st.Push("W")
      Console.WriteLine("Current stack: ")
      Dim c As Char
      For Each c In st
          Console.Write(c + " ")
      Next c
      Console.WriteLine()
      st.Push("V")
      st.Push("H")
      Console.WriteLine("The next poppable value in stack: {0}",
st.Peek())
      Console.WriteLine("Current stack: ")
      For Each c In st
          Console.Write(c + " ")
      Next c
      Console.WriteLine()
      Console.WriteLine("Removing values ")
      st.Pop()
      st.Pop()
```

```
      st.Pop()
      Console.WriteLine("Current stack: ")
      For Each c In st
          Console.Write(c + " ")
      Next c
      Console.ReadKey()
   End Sub
End Module
```

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

```
Current stack:
W G M A
The next poppable value in stack: H
Current stack:
H V W G M A
Removing values
Current stack:
G M A
```

Queue

It represents a first-in, first-out collection of object. It is used when you need a first-in, first-out access of items. When you add an item in the list, it is called **enqueue**, and when you remove an item, it is called **deque**

Properties and Methods of the Queue Class

The following table lists some of the commonly used **properties** of the **Queue** class:

Property	Description
Count	Gets the number of elements contained in the Queue.

The following table lists some of the commonly used **methods** of the **Queue** class:

S.N	Method Name & Purpose

1	**Public Overridable Sub Clear** Removes all elements from the Queue.
2	**Public Overridable Function Contains (obj As Object) As Boolean** Determines whether an element is in the Queue.
3	**Public Overridable Function Dequeue As Object** Removes and returns the object at the beginning of the Queue.
4	**Public Overridable Sub Enqueue (obj As Object)** Adds an object to the end of the Queue.
5	**Public Overridable Function ToArray As Object()** Copies the Queue to a new array.
6	**Public Overridable Sub TrimToSize** Sets the capacity to the actual number of elements in the Queue.

Example

The following example demonstrates use of Queue:

```
Module collections
   Sub Main()
      Dim q As Queue = New Queue()
      q.Enqueue("A")
      q.Enqueue("M")
      q.Enqueue("G")
      q.Enqueue("W")
      Console.WriteLine("Current queue: ")
      Dim c As Char
      For Each c In q
          Console.Write(c + " ")
      Next c
      Console.WriteLine()
```

```
        q.Enqueue("V")
        q.Enqueue("H")
        Console.WriteLine("Current queue: ")
        For Each c In q
            Console.Write(c + " ")
        Next c
        Console.WriteLine()
        Console.WriteLine("Removing some values ")
        Dim ch As Char
        ch = q.Dequeue()
        Console.WriteLine("The removed value: {0}", ch)
        ch = q.Dequeue()
        Console.WriteLine("The removed value: {0}", ch)
        Console.ReadKey()
    End Sub
End Module
```

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

```
Current queue:
A M G W
Current queue:
A M G W V H
Removing some values
The removed value: A
The removed value: M
```

BitArray

The BitArray class manages a compact array of bit values, which are represented as Booleans, where true indicates that the bit is on (1) and false indicates the bit is off (0).

It is used when you need to store the bits but do not know the number of bits in advance. You can access items from the BitArray collection by using an integer index, which starts from zero.

Properties and Methods of the BitArray Class

The following table lists some of the commonly used **properties** of the **BitArray** class:

Property	Description
Count	Gets the number of elements contained in the BitArray.
IsReadOnly	Gets a value indicating whether the BitArray is read-only.
Item	Gets or sets the value of the bit at a specific position in the BitArray.
Length	Gets or sets the number of elements in the BitArray.

The following table lists some of the commonly used **methods** of the **BitArray** class:

S.N	Method Name & Purpose
1	**Public Function And (value As BitArray) As BitArray** Performs the bitwise AND operation on the elements in the current BitArray against the corresponding elements in the specified BitArray.
2	**Public Function Get (index As Integer) As Boolean** Gets the value of the bit at a specific position in the BitArray.
3	**Public Function Not As BitArray** Inverts all the bit values in the current BitArray, so that elements set to true are changed to false, and elements set to false are changed to true.
4	**Public Function Or (value As BitArray) As BitArray**

	Performs the bitwise OR operation on the elements in the current BitArray against the corresponding elements in the specified BitArray.
5	**Public Sub Set (index As Integer, value As Boolean)** Sets the bit at a specific position in the BitArray to the specified value.
6	**Public Sub SetAll (value As Boolean)** Sets all bits in the BitArray to the specified value.
7	**Public Function Xor (value As BitArray) As BitArray** Performs the bitwise eXclusive OR operation on the elements in the current BitArray against the corresponding elements in the specified BitArray.

Example

The following example demonstrates the use of BitArray class:

```
Module collections
   Sub Main()
      'creating two  bit arrays of size 8
      Dim ba1 As BitArray = New BitArray(8)
      Dim ba2 As BitArray = New BitArray(8)
      Dim a() As Byte = {60}
      Dim b() As Byte = {13}
      'storing the values 60, and 13 into the bit arrays
      ba1 = New BitArray(a)
      ba2 = New BitArray(b)
      'content of ba1
      Console.WriteLine("Bit array ba1: 60")
      Dim i As Integer
      For i = 0 To ba1.Count
          Console.Write("{0 } ", ba1(i))
      Next i
      Console.WriteLine()
```

```
        'content of ba2
        Console.WriteLine("Bit array ba2: 13")
        For i = 0 To ba2.Count
            Console.Write("{0 } ", ba2(i))
        Next i
        Console.WriteLine()
        Dim ba3 As BitArray = New BitArray(8)
        ba3 = ba1.And(ba2)
        'content of ba3
        Console.WriteLine("Bit array ba3 after AND operation: 12")
        For i = 0 To ba3.Count
            Console.Write("{0 } ", ba3(i))
        Next i
        Console.WriteLine()
        ba3 = ba1.Or(ba2)
        'content of ba3
        Console.WriteLine("Bit array ba3 after OR operation: 61")
        For i = 0 To ba3.Count
            Console.Write("{0 } ", ba3(i))
        Next i
        Console.WriteLine()
        Console.ReadKey()
    End Sub
End Module
```

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

```
Bit array ba1: 60
False False True True True True False False
Bit array ba2: 13
True False True True False False False False
Bit array ba3 after AND operation: 12
False False True True False False False False
Bit array ba3 after OR operation: 61
True False True True False False False False
```

18. Functions

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

- Functions
- Sub procedures or Subs

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

Defining a Function

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

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

Where,

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

Example

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

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

```
        result = num1
    Else
        result = num2
    End If
    FindMax = result
End Function
```

Function Returning a Value

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

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

The following example demonstrates using the *FindMax* function:

```
Module myfunctions
    Function FindMax(ByVal num1 As Integer, ByVal num2 As Integer) As
Integer
       ' local variable declaration */
       Dim result As Integer
       If (num1 > num2) Then
           result = num1
       Else
           result = num2
       End If
       FindMax = result
    End Function
    Sub Main()
       Dim a As Integer = 100
       Dim b As Integer = 200
       Dim res As Integer
       res = FindMax(a, b)
       Console.WriteLine("Max value is : {0}", res)
       Console.ReadLine()
    End Sub
End Module
```

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

```
Max value is : 200
```

Recursive Function

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

```
Module myfunctions
   Function factorial(ByVal num As Integer) As Integer
      ' local variable declaration */
      Dim result As Integer
      If (num = 1) Then
          Return 1
      Else
          result = factorial(num - 1) * num
          Return result
      End If
   End Function
   Sub Main()
      'calling the factorial method
      Console.WriteLine("Factorial of 6 is : {0}", factorial(6))
      Console.WriteLine("Factorial of 7 is : {0}", factorial(7))
      Console.WriteLine("Factorial of 8 is : {0}", factorial(8))
      Console.ReadLine()
   End Sub
End Module
```

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

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

Param Arrays

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

The following example demonstrates this:

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

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

```
The sum is: 2938
```

Passing Arrays as Function Arguments

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

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

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

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

```
Average value is: 214.4
```

19. Sub Procedures

As we mentioned in the previous chapter, Sub procedures are procedures that do not return any value. We have been using the Sub procedure Main in all our examples. We have been writing console applications so far in these tutorials. When these applications start, the control goes to the Main Sub procedure, and it in turn, runs any other statements constituting the body of the program.

Defining Sub Procedures

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

Example

The following example demonstrates a Sub procedure *CalculatePay* that takes two parameters *hours* and *wages* and displays the total pay of an employee:

```
Module mysub
   Sub CalculatePay(ByVal hours As Double, ByVal wage As Decimal)
      'local variable declaration
      Dim pay As Double
      pay = hours * wage
      Console.WriteLine("Total Pay: {0:C}", pay)
   End Sub
   Sub Main()
```

```
      'calling the CalculatePay Sub Procedure
      CalculatePay(25, 10)
      CalculatePay(40, 20)
      CalculatePay(30, 27.5)
      Console.ReadLine()
   End Sub
End Module
```

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

```
Total Pay: $250.00
Total Pay: $800.00
Total Pay: $825.00
```

Passing Parameters by Value

This is the default mechanism for passing parameters to a method. In this mechanism, when a method is called, a new storage location is created for each value parameter. The values of the actual parameters are copied into them. So, the changes made to the parameter inside the method have no effect on the argument.

In VB.Net, you declare the reference parameters using the **ByVal** keyword. The following example demonstrates the concept:

```
Module paramByval
   Sub swap(ByVal x As Integer, ByVal y As Integer)
      Dim temp As Integer
      temp = x ' save the value of x
      x = y    ' put y into x
      y = temp 'put temp into y
   End Sub
   Sub Main()
      ' local variable definition
      Dim a As Integer = 100
      Dim b As Integer = 200
      Console.WriteLine("Before swap, value of a : {0}", a)
      Console.WriteLine("Before swap, value of b : {0}", b)
```

```
      ' calling a function to swap the values '
      swap(a, b)

      Console.WriteLine("After swap, value of a : {0}", a)
      Console.WriteLine("After swap, value of b : {0}", b)
      Console.ReadLine()
   End Sub
End Module
```

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

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

It shows that there is no change in the values though they had been changed inside the function.

Passing Parameters by Reference

A reference parameter is a reference to a memory location of a variable. When you pass parameters by reference, unlike value parameters, a new storage location is not created for these parameters. The reference parameters represent the same memory location as the actual parameters that are supplied to the method.

In VB.Net, you declare the reference parameters using the **ByRef** keyword. The following example demonstrates this:

```
Module paramByref
   Sub swap(ByRef x As Integer, ByRef y As Integer)
      Dim temp As Integer
      temp = x ' save the value of x
      x = y    ' put y into x
      y = temp 'put temp into y
   End Sub
   Sub Main()
      ' local variable definition
      Dim a As Integer = 100
      Dim b As Integer = 200
```

```
      Console.WriteLine("Before swap, value of a : {0}", a)
      Console.WriteLine("Before swap, value of b : {0}", b)
      ' calling a function to swap the values '
      swap(a, b)
      Console.WriteLine("After swap, value of a : {0}", a)
      Console.WriteLine("After swap, value of b : {0}", b)
      Console.ReadLine()
   End Sub
End Module
```

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

```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 200
After swap, value of b : 100
```

20. Classes & Objects

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

Objects are instances of a class. The methods and variables that constitute a class are called members of the class.

Class Definition

A class definition starts with the keyword **Class** followed by the class name; and the class body, ended by the End Class statement. Following is the general form of a class definition:

```
[ <attributelist> ] [ accessmodifier ] [ Shadows ] [ MustInherit |
NotInheritable ] [ Partial ] _

Class name [ ( Of typelist ) ]

    [ Inherits classname ]

    [ Implements interfacenames ]

    [ statements ]

End Class
```

Where,

- ***attributelist*** is a list of attributes that apply to the class. Optional.
- ***accessmodifier*** defines the access levels of the class, it has values as - Public, Protected, Friend, Protected Friend and Private. Optional.
- ***Shadows*** indicate that the variable re-declares and hides an identically named element, or set of overloaded elements, in a base class. Optional.
- ***MustInherit*** specifies that the class can be used only as a base class and that you cannot create an object directly from it, i.e., an abstract class. Optional.
- ***NotInheritable*** specifies that the class cannot be used as a base class.
- ***Partial*** indicates a partial definition of the class.
- ***Inherits*** specifies the base class it is inheriting from.

- ***Implements*** specifies the interfaces the class is inheriting from.

The following example demonstrates a Box class, with three data members, length, breadth, and height:

```
Module mybox
   Class Box
      Public length As Double   ' Length of a box
      Public breadth As Double   ' Breadth of a box
      Public height As Double    ' Height of a box
   End Class
   Sub Main()
      Dim Box1 As Box = New Box()        ' Declare Box1 of type Box
      Dim Box2 As Box = New Box()        ' Declare Box2 of type Box
      Dim volume As Double = 0.0     ' Store the volume of a box here
      ' box 1 specification
      Box1.height = 5.0
      Box1.length = 6.0
      Box1.breadth = 7.0
       ' box 2 specification
      Box2.height = 10.0
      Box2.length = 12.0
      Box2.breadth = 13.0
      'volume of box 1
      volume = Box1.height * Box1.length * Box1.breadth
      Console.WriteLine("Volume of Box1 : {0}", volume)
      'volume of box 2
      volume = Box2.height * Box2.length * Box2.breadth
      Console.WriteLine("Volume of Box2 : {0}", volume)
      Console.ReadKey()
   End Sub
End Module
```

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

```
Volume of Box1 : 210
```

```
Volume of Box2 : 1560
```

Member Functions and Encapsulation

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member and has access to all the members of a class for that object.

Member variables are attributes of an object (from design perspective) and they are kept private to implement encapsulation. These variables can only be accessed using the public member functions.

Let us put above concepts to set and get the value of different class members in a class:

```
Module mybox
   Class Box
      Public length As Double   ' Length of a box
      Public breadth As Double   ' Breadth of a box
      Public height As Double    ' Height of a box
      Public Sub setLength(ByVal len As Double)
          length = len
      End Sub
      Public Sub setBreadth(ByVal bre As Double)
          breadth = bre
      End Sub
      Public Sub setHeight(ByVal hei As Double)
          height = hei
      End Sub
      Public Function getVolume() As Double
          Return length * breadth * height
      End Function
   End Class
   Sub Main()
      Dim Box1 As Box = New Box()        ' Declare Box1 of type Box
      Dim Box2 As Box = New Box()        ' Declare Box2 of type Box
      Dim volume As Double = 0.0     ' Store the volume of a box here
```

```
      ' box 1 specification
       Box1.setLength(6.0)
       Box1.setBreadth(7.0)
       Box1.setHeight(5.0)

       'box 2 specification
       Box2.setLength(12.0)
       Box2.setBreadth(13.0)
       Box2.setHeight(10.0)

       ' volume of box 1
       volume = Box1.getVolume()
       Console.WriteLine("Volume of Box1 : {0}", volume)

       'volume of box 2
       volume = Box2.getVolume()
       Console.WriteLine("Volume of Box2 : {0}", volume)
       Console.ReadKey()
    End Sub
End Module
```

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

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

Constructors and Destructors

A class **constructor** is a special member Sub of a class that is executed whenever we create new objects of that class. A constructor has the name **New** and it does not have any return type.

Following program explains the concept of constructor:

```
Class Line
   Private length As Double    ' Length of a line
   Public Sub New()   'constructor
```

```
      Console.WriteLine("Object is being created")
   End Sub
   Public Sub setLength(ByVal len As Double)
      length = len
   End Sub

   Public Function getLength() As Double
      Return length
   End Function
   Shared Sub Main()
      Dim line As Line = New Line()
      'set line length
      line.setLength(6.0)
      Console.WriteLine("Length of line : {0}", line.getLength())
      Console.ReadKey()
   End Sub
End Class
```

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

```
Object is being created
Length of line : 6
```

A default constructor does not have any parameter, but if you need, a constructor can have parameters. Such constructors are called **parameterized constructors**. This technique helps you to assign initial value to an object at the time of its creation as shown in the following example:

```
Class Line
   Private length As Double    ' Length of a line
   Public Sub New(ByVal len As Double)   'parameterised constructor
      Console.WriteLine("Object is being created, length = {0}", len)
      length = len
   End Sub
   Public Sub setLength(ByVal len As Double)
      length = len
   End Sub
```

```
   Public Function getLength() As Double

      Return length
   End Function
   Shared Sub Main()
      Dim line As Line = New Line(10.0)
      Console.WriteLine("Length of line set by constructor : {0}",
line.getLength())
      'set line length
      line.setLength(6.0)
      Console.WriteLine("Length of line set by setLength : {0}",
line.getLength())
      Console.ReadKey()
   End Sub
End Class
```

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

```
Object is being created, length = 10
Length of line set by constructor : 10
Length of line set by setLength : 6
```

A **destructor** is a special member Sub of a class that is executed whenever an object of its class goes out of scope.

A **destructor** has the name **Finalize** and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories, etc. Destructors cannot be inherited or overloaded.

Following example demonstrates the concept of destructor:

```
Class Line
   Private length As Double    ' Length of a line
   Public Sub New()   'parameterised constructor
      Console.WriteLine("Object is being created")
   End Sub
   Protected Overrides Sub Finalize()  ' destructor
      Console.WriteLine("Object is being deleted")
   End Sub
```

```
   Public Sub setLength(ByVal len As Double)
      length = len
   End Sub
   Public Function getLength() As Double
      Return length
   End Function
   Shared Sub Main()
      Dim line As Line = New Line()
      'set line length
      line.setLength(6.0)
      Console.WriteLine("Length of line : {0}", line.getLength())
      Console.ReadKey()
   End Sub
End Class
```

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

```
Object is being created
Length of line : 6
Object is being deleted
```

Shared Members of a VB.Net Class

We can define class members as static using the Shared keyword. When we declare a member of a class as Shared, it means no matter how many objects of the class are created, there is only one copy of the member.

The keyword **Shared** implies that only one instance of the member exists for a class. Shared variables are used for defining constants because their values can be retrieved by invoking the class without creating an instance of it.

Shared variables can be initialized outside the member function or class definition. You can also initialize Shared variables inside the class definition.

You can also declare a member function as Shared. Such functions can access only Shared variables. The Shared functions exist even before the object is created.

The following example demonstrates the use of shared members:

```
Class StaticVar
   Public Shared num As Integer
   Public Sub count()
```

```
      num = num + 1
   End Sub
   Public Shared Function getNum() As Integer
      Return num
   End Function
   Shared Sub Main()
      Dim s As StaticVar = New StaticVar()
      s.count()
      s.count()
      s.count()
      Console.WriteLine("Value of variable num: {0}",
StaticVar.getNum())
      Console.ReadKey()
   End Sub
End Class
```

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

```
Value of variable num: 3
```

Inheritance

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

Base & Derived Classes

A class can be derived from more than one class or interface, which means that it can inherit data and functions from multiple base classes or interfaces.

The syntax used in VB.Net for creating derived classes is as follows:

```
<access-specifier> Class <base_class>
...
```

```
End Class
Class <derived_class>: Inherits <base_class>
...
End Class
```

Consider a base class Shape and its derived class Rectangle:

```
' Base class
Class Shape
   Protected width As Integer
   Protected height As Integer
   Public Sub setWidth(ByVal w As Integer)
      width = w
   End Sub
   Public Sub setHeight(ByVal h As Integer)
      height = h
   End Sub
End Class
' Derived class
Class Rectangle : Inherits Shape
   Public Function getArea() As Integer
      Return (width * height)
   End Function
End Class
Class RectangleTester
   Shared Sub Main()
      Dim rect As Rectangle = New Rectangle()
      rect.setWidth(5)
      rect.setHeight(7)
      ' Print the area of the object.
      Console.WriteLine("Total area: {0}", rect.getArea())
      Console.ReadKey()
   End Sub
End Class
```

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

```
Total area: 35
```

Base Class Initialization

The derived class inherits the base class member variables and member methods. Therefore, the super class object should be created before the subclass is created. The super class or the base class is implicitly known as **MyBase** in VB.Net

The following program demonstrates this:

```
' Base class
Class Rectangle
   Protected width As Double
   Protected length As Double
   Public Sub New(ByVal l As Double, ByVal w As Double)
      length = l
      width = w
   End Sub
   Public Function GetArea() As Double
      Return (width * length)
   End Function
   Public Overridable Sub Display()
      Console.WriteLine("Length: {0}", length)
      Console.WriteLine("Width: {0}", width)
      Console.WriteLine("Area: {0}", GetArea())
   End Sub
   'end class Rectangle
End Class
'Derived class
Class Tabletop : Inherits Rectangle
   Private cost As Double
   Public Sub New(ByVal l As Double, ByVal w As Double)
      MyBase.New(l, w)
   End Sub
   Public Function GetCost() As Double
      Dim cost As Double
```

```
      cost = GetArea() * 70
      Return cost
   End Function
   Public Overrides Sub Display()
      MyBase.Display()
      Console.WriteLine("Cost: {0}", GetCost())
   End Sub
    'end class Tabletop
End Class
Class RectangleTester
   Shared Sub Main()
      Dim t As Tabletop = New Tabletop(4.5, 7.5)
      t.Display()
      Console.ReadKey()
   End Sub
End Class
```

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

```
Length: 4.5
Width: 7.5
Area: 33.75
Cost: 2362.5
```

VB.Net supports multiple inheritance.

21. Exception Handling

An exception is a problem that arises during the execution of a program. An exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. VB.Net exception handling is built upon four keywords: **Try**, **Catch**, **Finally** and **Throw**.

- **Try**: A Try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more Catch blocks.

- **Catch**: A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The Catch keyword indicates the catching of an exception.

- **Finally**: The Finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.

- **Throw**: A program throws an exception when a problem shows up. This is done using a Throw keyword.

Syntax

Assuming a block will raise an exception, a method catches an exception using a combination of the Try and Catch keywords. A Try/Catch block is placed around the code that might generate an exception. Code within a Try/Catch block is referred to as protected code, and the syntax for using Try/Catch looks like the following:

```
Try
    [ tryStatements ]
    [ Exit Try ]
[ Catch [ exception [ As type ] ] [ When expression ]
    [ catchStatements ]
    [ Exit Try ] ]
[ Catch ... ]
[ Finally
    [ finallyStatements ] ]
```

```
End Try
```

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

Exception Classes in .Net Framework

In the .Net Framework, exceptions are represented by classes. The exception classes in .Net Framework are mainly directly or indirectly derived from the **System.Exception** class. Some of the exception classes derived from the System.Exception class are the **System.ApplicationException** and **System.SystemException** classes.

The **System.ApplicationException** class supports exceptions generated by application programs. So the exceptions defined by the programmers should derive from this class. The **System.SystemException** class is the base class for all predefined system exception.

The following table provides some of the predefined exception classes derived from the Sytem.SystemException class:

Exception Class	Description
System.IO.IOException	Handles I/O errors.
System.IndexOutOfRangeException	Handles errors generated when a method refers to an array index out of range.
System.ArrayTypeMismatchException	Handles errors generated when type is mismatched with the array type.
System.NullReferenceException	Handles errors generated from deferencing a null object.
System.DivideByZeroException	Handles errors generated from dividing a dividend with zero.
System.InvalidCastException	Handles errors generated during typecasting.
System.OutOfMemoryException	Handles errors generated from insufficient free memory.

System.StackOverflowException	Handles errors generated from stack overflow.

Handling Exceptions

VB.Net provides a structured solution to the exception handling problems in the form of try and catch blocks. Using these blocks the core program statements are separated from the error-handling statements.

These error handling blocks are implemented using the **Try**, **Catch** and **Finally** keywords. Following is an example of throwing an exception when dividing by zero condition occurs:

```
Module exceptionProg
   Sub division(ByVal num1 As Integer, ByVal num2 As Integer)
      Dim result As Integer
      Try
          result = num1 \ num2
      Catch e As DivideByZeroException
          Console.WriteLine("Exception caught: {0}", e)
      Finally
          Console.WriteLine("Result: {0}", result)
      End Try
   End Sub
   Sub Main()
      division(25, 0)
      Console.ReadKey()
  End Sub
End Module
```

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

```
Exception caught: System.DivideByZeroException: Attempted to divide by
zero.
at ...
Result: 0
```

Creating User-Defined Exceptions

You can also define your own exception. User-defined exception classes are derived from the **ApplicationException** class. The following example demonstrates this:

```
Module exceptionProg
   Public Class TempIsZeroException : Inherits ApplicationException
      Public Sub New(ByVal message As String)
          MyBase.New(message)
      End Sub
   End Class
   Public Class Temperature
      Dim temperature As Integer = 0
      Sub showTemp()
          If (temperature = 0) Then
              Throw (New TempIsZeroException("Zero Temperature found"))
          Else
              Console.WriteLine("Temperature: {0}", temperature)
          End If
      End Sub
   End Class
   Sub Main()
      Dim temp As Temperature = New Temperature()
      Try
          temp.showTemp()
      Catch e As TempIsZeroException
          Console.WriteLine("TempIsZeroException: {0}", e.Message)
      End Try
      Console.ReadKey()
   End Sub
End Module
```

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

```
TempIsZeroException: Zero Temperature found
```

Throwing Objects

You can throw an object if it is either directly or indirectly derived from the System.Exception class. You can use a throw statement in the catch block to throw the present object as:

```
Throw [ expression ]
```

The following program demonstrates this:

```
Module exceptionProg
   Sub Main()
      Try
          Throw New ApplicationException("A custom exception _
             is being thrown here...")
      Catch e As Exception
          Console.WriteLine(e.Message)
      Finally
          Console.WriteLine("Now inside the Finally Block")
      End Try
      Console.ReadKey()
   End Sub
End Module
```

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

```
A custom exception is being thrown here...
Now inside the Finally Block
```

22. File Handling

A **file** is a collection of data stored in a disk with a specific name and a directory path. When a file is opened for reading or writing, it becomes a **stream**.

The stream is basically the sequence of bytes passing through the communication path. There are two main streams: the **input stream** and the **output stream**. The **input stream** is used for reading data from file (read operation) and the **output stream** is used for writing into the file (write operation).

VB.Net I/O Classes

The System.IO namespace has various classes that are used for performing various operations with files, like creating and deleting files, reading from or writing to a file, closing a file, etc.

The following table shows some commonly used non-abstract classes in the System.IO namespace:

I/O Class	Description
BinaryReader	Reads primitive data from a binary stream.
BinaryWriter	Writes primitive data in binary format.
BufferedStream	A temporary storage for a stream of bytes.
Directory	Helps in manipulating a directory structure.
DirectoryInfo	Used for performing operations on directories.
DriveInfo	Provides information for the drives.
File	Helps in manipulating files.
FileInfo	Used for performing operations on files.

FileStream	Used to read from and write to any location in a file.
MemoryStream	Used for random access of streamed data stored in memory.
Path	Performs operations on path information.
StreamReader	Used for reading characters from a byte stream.
StreamWriter	Is used for writing characters to a stream.
StringReader	Is used for reading from a string buffer.
StringWriter	Is used for writing into a string buffer.

The FileStream Class

The **FileStream** class in the System.IO namespace helps in reading from, writing to and closing files. This class derives from the abstract class Stream.

You need to create a **FileStream** object to create a new file or open an existing file. The syntax for creating a **FileStream** object is as follows:

```
Dim <object_name> As FileStream = New FileStream(<file_name>, <FileMode
Enumerator>, <FileAccess Enumerator>, <FileShare Enumerator>)
```

For example, for creating a FileStream object **F** for reading a file named **sample.txt**:

```
Dim f1 As FileStream = New FileStream("test.dat", FileMode.OpenOrCreate,
FileAccess.ReadWrite)
```

Parameter	Description
FileMode	The **FileMode** enumerator defines various methods for opening files. The members of the FileMode enumerator are:

	• **Append**: It opens an existing file and puts cursor at the end of file, or creates the file, if the file does not exist. • **Create**: It creates a new file. • **CreateNew**: It specifies to the operating system that it should create a new file. • **Open**: It opens an existing file. • **OpenOrCreate**: It specifies to the operating system that it should open a file if it exists, otherwise it should create a new file. • **Truncate**: It opens an existing file and truncates its size to zero bytes.
FileAccess	**FileAccess** enumerators have members: **Read**, **ReadWrite**, and **Write**.
FileShare	**FileShare** enumerators have the following members: • **Inheritable**: It allows a file handle to pass inheritance to the child processes • **None**: It declines sharing of the current file • **Read**: It allows opening the file for reading • **ReadWrite**: It allows opening the file for reading and writing • **Write**: It allows opening the file for writing

Example

The following program demonstrates use of the **FileStream** class:

```
Imports System.IO
Module fileProg
```

```
   Sub Main()
      Dim f1 As FileStream = New FileStream("test.dat", _
              FileMode.OpenOrCreate, FileAccess.ReadWrite)
      Dim i As Integer
      For i = 0 To 20
          f1.WriteByte(CByte(i))
      Next i
      f1.Position = 0
      For i = 0 To 20
          Console.Write("{0} ", f1.ReadByte())
      Next i
      f1.Close()
      Console.ReadKey()
   End Sub
End Module
```

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

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 -1
```

Advanced File Operations in VB.Net

The preceding example provides simple file operations in VB.Net. However, to utilize the immense powers of System.IO classes, you need to know the commonly used properties and methods of these classes.

We will discuss these classes and the operations they perform in the following sections. Please click the links provided to get to the individual sections:

Topic and Description
Reading from and Writing into Text files It involves reading from and writing into text files. The **StreamReader** and**StreamWriter** classes help to accomplish it.
Reading from and Writing into Binary files It involves reading from and writing into binary files. The **BinaryReader** and **BinaryWriter** classes help to accomplish this.

Manipulating the Windows file system
It gives a VB.Net programmer the ability to browse and locate Windows files and directories.

Reading from and Writing to Text Files

The **StreamReader** and **StreamWriter** classes are used for reading from and writing data to text files. These classes inherit from the abstract base class Stream, which supports reading and writing bytes into a file stream.

The StreamReader Class

The **StreamReader** class also inherits from the abstract base class TextReader that represents a reader for reading series of characters. The following table describes some of the commonly used **methods** of the StreamReader class:

S.N	Method Name & Purpose
1	**Public Overrides Sub Close** It closes the StreamReader object and the underlying stream and releases any system resources associated with the reader.
2	**Public Overrides Function Peek As Integer** Returns the next available character but does not consume it.
3	**Public Overrides Function Read As Integer** Reads the next character from the input stream and advances the character position by one character.

Example

The following example demonstrates reading a text file named Jamaica.txt. The file reads:

```
Down the way where the nights are gay
And the sun shines daily on the mountain top
I took a trip on a sailing ship
And when I reached Jamaica
I made a stop
Imports System.IO
```

```
Module fileProg
   Sub Main()

      Try
          ' Create an instance of StreamReader to read from a file.
          ' The using statement also closes the StreamReader.
          Using sr As StreamReader = New StreamReader("e:/jamaica.txt")
              Dim line As String
              ' Read and display lines from the file until the end of
              ' the file is reached.
              line = sr.ReadLine()
              While (line <> Nothing)
                  Console.WriteLine(line)
                  line = sr.ReadLine()
              End While
          End Using
      Catch e As Exception
          ' Let the user know what went wrong.
          Console.WriteLine("The file could not be read:")
          Console.WriteLine(e.Message)
      End Try
      Console.ReadKey()
   End Sub
End Module
```

Guess what it displays when you compile and run the program!

The StreamWriter Class

The **StreamWriter** class inherits from the abstract class TextWriter that represents a writer, which can write a series of character.

The following table shows some of the most commonly used methods of this class:

S.N	Method Name & Purpose
1	**Public Overrides Sub Close** Closes the current StreamWriter object and the underlying stream.

2	**Public Overrides Sub Flush** Clears all buffers for the current writer and causes any buffered data to be written to the underlying stream.
3	**Public Overridable Sub Write (value As Boolean)** Writes the text representation of a Boolean value to the text string or stream. (Inherited from TextWriter.)
4	**Public Overrides Sub Write (value As Char)** Writes a character to the stream.
5	**Public Overridable Sub Write (value As Decimal)** Writes the text representation of a decimal value to the text string or stream.
6	**Public Overridable Sub Write (value As Double)** Writes the text representation of an 8-byte floating-point value to the text string or stream.
7	**Public Overridable Sub Write (value As Integer)** Writes the text representation of a 4-byte signed integer to the text string or stream.
8	**Public Overrides Sub Write (value As String)** Writes a string to the stream.
9	**Public Overridable Sub WriteLine** Writes a line terminator to the text string or stream.

The above list is not exhaustive. For complete list of methods please visit Microsoft's documentation

Example

The following example demonstrates writing text data into a file using the StreamWriter class:

```
Imports System.IO

Module fileProg

   Sub Main()

      Dim names As String() = New String() {"Zara Ali", _
```

```
         "Nuha Ali", "Amir Sohel", "M Amlan"}
      Dim s As String
      Using sw As StreamWriter = New StreamWriter("names.txt")
          For Each s In names
              sw.WriteLine(s)
          Next s
      End Using
      ' Read and show each line from the file.
      Dim line As String
      Using sr As StreamReader = New StreamReader("names.txt")
          line = sr.ReadLine()
          While (line <> Nothing)
              Console.WriteLine(line)
              line = sr.ReadLine()
          End While
      End Using
      Console.ReadKey()
   End Sub
End Module
```

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

```
Zara Ali
Nuha Ali
Amir Sohel
M Amlan
```

Binary Files

The **BinaryReader** and **BinaryWriter** classes are used for reading from and writing to a binary file.

The BinaryReader Class

The **BinaryReader** class is used to read binary data from a file. A **BinaryReader** object is created by passing a **FileStream** object to its constructor.

The following table shows some of the commonly used **methods** of the **BinaryReader**class.

S.N	Method Name & Purpose
1	**Public Overridable Sub Close** It closes the BinaryReader object and the underlying stream.
2	**Public Overridable Function Read As Integer** Reads the characters from the underlying stream and advances the current position of the stream.
3	**Public Overridable Function ReadBoolean As Boolean** Reads a Boolean value from the current stream and advances the current position of the stream by one byte.
4	**Public Overridable Function ReadByte As Byte** Reads the next byte from the current stream and advances the current position of the stream by one byte.
5	**Public Overridable Function ReadBytes (count As Integer) As Byte()** Reads the specified number of bytes from the current stream into a byte array and advances the current position by that number of bytes.
6	**Public Overridable Function ReadChar As Char** Reads the next character from the current stream and advances the current position of the stream in accordance with the Encoding used and the specific character being read from the stream.
7	**Public Overridable Function ReadChars (count As Integer) As Char()** Reads the specified number of characters from the current stream, returns the data in a character array, and advances the current position in accordance with the Encoding used and the specific character being read from the stream.
8	**Public Overridable Function ReadDouble As Double** Reads an 8-byte floating point value from the current stream and advances the current position of the stream by eight bytes.

9	**Public Overridable Function ReadInt32 As Integer** Reads a 4-byte signed integer from the current stream and advances the current position of the stream by four bytes.
10	**Public Overridable Function ReadString As String** Reads a string from the current stream. The string is prefixed with the length, encoded as an integer seven bits at a time.

The BinaryWriter Class

The **BinaryWriter** class is used to write binary data to a stream. A BinaryWriter object is created by passing a FileStream object to its constructor.

The following table shows some of the commonly used methods of the BinaryWriter class.

S.N	Function Name & Description
1	**Public Overridable Sub Close** It closes the BinaryWriter object and the underlying stream.
2	**Public Overridable Sub Flush** Clears all buffers for the current writer and causes any buffered data to be written to the underlying device.
3	**Public Overridable Function Seek (offset As Integer, origin As SeekOrigin) As Long** Sets the position within the current stream.
4	**Public Overridable Sub Write (value As Boolean)** Writes a one-byte Boolean value to the current stream, with 0 representing false and 1 representing true.
5	**Public Overridable Sub Write (value As Byte)** Writes an unsigned byte to the current stream and advances the stream position by one byte.
6	**Public Overridable Sub Write (buffer As Byte())** Writes a byte array to the underlying stream.
7	**Public Overridable Sub Write (ch As Char)**

	Writes a Unicode character to the current stream and advances the current position of the stream in accordance with the Encoding used and the specific characters being written to the stream.
8	**Public Overridable Sub Write (chars As Char())** Writes a character array to the current stream and advances the current position of the stream in accordance with the Encoding used and the specific characters being written to the stream.
9	**Public Overridable Sub Write (value As Double)** Writes an eight-byte floating-point value to the current stream and advances the stream position by eight bytes.
10	**Public Overridable Sub Write (value As Integer)** Writes a four-byte signed integer to the current stream and advances the stream position by four bytes.
11	**Public Overridable Sub Write (value As String)** Writes a length-prefixed string to this stream in the current encoding of the BinaryWriter and advances the current position of the stream in accordance with the encoding used and the specific characters being written to the stream.

For complete list of methods, please visit Microsoft's documentation.

Example

The following example demonstrates reading and writing binary data:

```
Imports System.IO
Module fileProg
   Sub Main()
      Dim bw As BinaryWriter
      Dim br As BinaryReader
      Dim i As Integer = 25
      Dim d As Double = 3.14157
      Dim b As Boolean = True
      Dim s As String = "I am happy"
      'create the file
      Try
```

```
            bw = New BinaryWriter(New FileStream("mydata",
FileMode.Create))
        Catch e As IOException
            Console.WriteLine(e.Message + "\n Cannot create file.")
            Return
        End Try
        'writing into the file
        Try
            bw.Write(i)
            bw.Write(d)
            bw.Write(b)
            bw.Write(s)
        Catch e As IOException
            Console.WriteLine(e.Message + "\n Cannot write to file.")
            Return
        End Try
        bw.Close()
        'reading from the file
        Try
            br = New BinaryReader(New FileStream("mydata", FileMode.Open))
        Catch e As IOException
            Console.WriteLine(e.Message + "\n Cannot open file.")
            Return
        End Try
        Try
             i = br.ReadInt32()
            Console.WriteLine("Integer data: {0}", i)
            d = br.ReadDouble()
            Console.WriteLine("Double data: {0}", d)
            b = br.ReadBoolean()
            Console.WriteLine("Boolean data: {0}", b)
            s = br.ReadString()
            Console.WriteLine("String data: {0}", s)
        Catch e As IOException
```

```
            Console.WriteLine(e.Message + "\n Cannot read from file.")
            Return
        End Try
        br.Close()
        Console.ReadKey()
    End Sub
End Module
```

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

```
Integer data: 25
Double data: 3.14157
Boolean data: True
String data: I am happy
```

Windows File System

VB.Net allows you to work with the directories and files using various directory and file-related classes like, the **DirectoryInfo** class and the **FileInfo** class.

The DirectoryInfo Class

The **DirectoryInfo** class is derived from the **FileSystemInfo** class. It has various methods for creating, moving, and browsing through directories and subdirectories. This class cannot be inherited.

Following are some commonly used **properties** of the **DirectoryInfo** class:

S.N	Property Name & Description
1	**Attributes** Gets the attributes for the current file or directory.
2	**CreationTime** Gets the creation time of the current file or directory.
3	**Exists** Gets a Boolean value indicating whether the directory exists.
4	**Extension** Gets the string representing the file extension.

5	**FullName** Gets the full path of the directory or file.
6	**LastAccessTime** Gets the time the current file or directory was last accessed.
7	**Name** Gets the name of this DirectoryInfo instance.

Following are some commonly used **methods** of the **DirectoryInfo** class:

S.N	Method Name & Purpose
1	**Public Sub Create** Creates a directory.
2	**Public Function CreateSubdirectory (path As String) As DirectoryInfo** Creates a subdirectory or subdirectories on the specified path. The specified path can be relative to this instance of the DirectoryInfo class.
3	**Public Overrides Sub Delete** Deletes this DirectoryInfo if it is empty.
4	**Public Function GetDirectories As DirectoryInfo()** Returns the subdirectories of the current directory.
5	**Public Function GetFiles As FileInfo()** Returns a file list from the current directory.

For complete list of properties and methods please visit Microsoft's documentation.

The FileInfo Class

The **FileInfo** class is derived from the **FileSystemInfo** class. It has properties and instance methods for creating, copying, deleting, moving, and opening of files, and helps in the creation of FileStream objects. This class cannot be inherited.

Following are some commonly used **properties** of the **FileInfo** class:

S.N	Property Name & Description
1	**Attributes** Gets the attributes for the current file.
2	**CreationTime** Gets the creation time of the current file.
3	**Directory** Gets an instance of the directory, which the file belongs to.
4	**Exists** Gets a Boolean value indicating whether the file exists.
5	**Extension** Gets the string representing the file extension.
6	**FullName** Gets the full path of the file.
7	**LastAccessTime** Gets the time the current file was last accessed.
8	**LastWriteTime** Gets the time of the last written activity of the file.
9	**Length** Gets the size, in bytes, of the current file.
10	**Name** Gets the name of the file.

Following are some commonly used **methods** of the **FileInfo** class:

S.N	Method Name & Purpose
1	**Public Function AppendText As StreamWriter** Creates a StreamWriter that appends text to the file represented by this instance of the FileInfo.

2	**Public Function Create As FileStream** Creates a file.
3	**Public Overrides Sub Delete** Deletes a file permanently.
4	**Public Sub MoveTo (destFileName As String)** Moves a specified file to a new location, providing the option to specify a new file name.
5	**Public Function Open (mode As FileMode) As FileStream** Opens a file in the specified mode.
6	**Public Function Open (mode As FileMode, access As FileAccess) As FileStream** Opens a file in the specified mode with read, write, or read/write access.
7	**Public Function Open (mode As FileMode, access As FileAccess, share As FileShare) As FileStream** Opens a file in the specified mode with read, write, or read/write access and the specified sharing option.
8	**Public Function OpenRead As FileStream** Creates a read-only FileStream
9	**Public Function OpenWrite As FileStream** Creates a write-only FileStream.

For complete list of properties and methods, please visit Microsoft's documentation

Example

The following example demonstrates the use of the above-mentioned classes:

```
Imports System.IO
Module fileProg
   Sub Main()
      'creating a DirectoryInfo object
      Dim mydir As DirectoryInfo = New DirectoryInfo("c:\Windows")
      ' getting the files in the directory, their names and size
      Dim f As FileInfo() = mydir.GetFiles()
```

```
      Dim file As FileInfo
      For Each file In f
          Console.WriteLine("File Name: {0} Size: {1}  ", file.Name,
file.Length)
      Next file
      Console.ReadKey()
   End Sub
End Module
```

When you compile and run the program, it displays the names of files and their size in the Windows directory.

23. Basic Controls

An object is a type of user interface element you create on a Visual Basic form by using a toolbox control. In fact, in Visual Basic, the form itself is an object. Every Visual Basic control consists of three important elements:

Properties which describe the object,

Methods cause an object to do something and

Events are what happens when an object does something.

Control Properties

All the Visual Basic Objects can be moved, resized, or customized by setting their properties. A property is a value or characteristic held by a Visual Basic object, such as Caption or Fore Color.

Properties can be set at design time by using the Properties window or at run time by using statements in the program code.

```
Object. Property = Value
```

Where,

Object is the name of the object you're customizing.

Property is the characteristic you want t,o change.

Value is the new property setting.

For example,

```
Form1.Caption = "Hello"
```

You can set any of the form properties using Properties Window. Most of the properties can be set or read during application execution. You can refer to Microsoft documentation for a complete list of properties associated with different controls and restrictions applied to them.

Control Methods

A method is a procedure created as a member of a class and they cause an object to do something. Methods are used to access or manipulate the characteristics of an object or a variable. There are mainly two categories of methods you will use in your classes:

- If you are using a control such as one of those provided by the Toolbox, you can call any of its public methods. The requirements of such a method depend on the class being used.

- If none of the existing methods can perform your desired task, you can add a method to a class.

For example, the MessageBox control has a method named Show, which is called in the code snippet below:

```
Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs)

     Handles Button1.Click

        MessageBox.Show("Hello, World")

    End Sub

End Class
```

Control Events

An event is a signal that informs an application that something important has occurred. For example, when a user clicks a control on a form, the form can raise a **Click** event and call a procedure that handles the event. There are various types of events associated with a Form like click, double click, close, load, resize, etc.

Following is the default structure of a form **Load** event handler subroutine. You can see this code by double clicking the code which will give you a complete list of the all events associated with Form control:

```
Private Sub Form1_Load(sender As Object, e As EventArgs) Handles
MyBase.Load

 'event handler code goes here

End Sub
```

Here, **Handles MyBase.Load** indicates that **Form1_Load()** subroutine handles **Load** event. Similar way, you can check stub code for click, double click. If you want to initialize some variables like properties, etc., then you will keep such code inside Form1_Load() subroutine. Here, important point to note is the name of the event handler, which is by default Form1_Load, but you can change this name based on your naming convention you use in your application programming.

Basic Controls

VB.Net provides a huge variety of controls that help you to create rich user interface. Functionalities of all these controls are defined in the respective control classes. The control classes are defined in the **System.Windows.Forms** namespace.

The following table lists some of the commonly used controls:

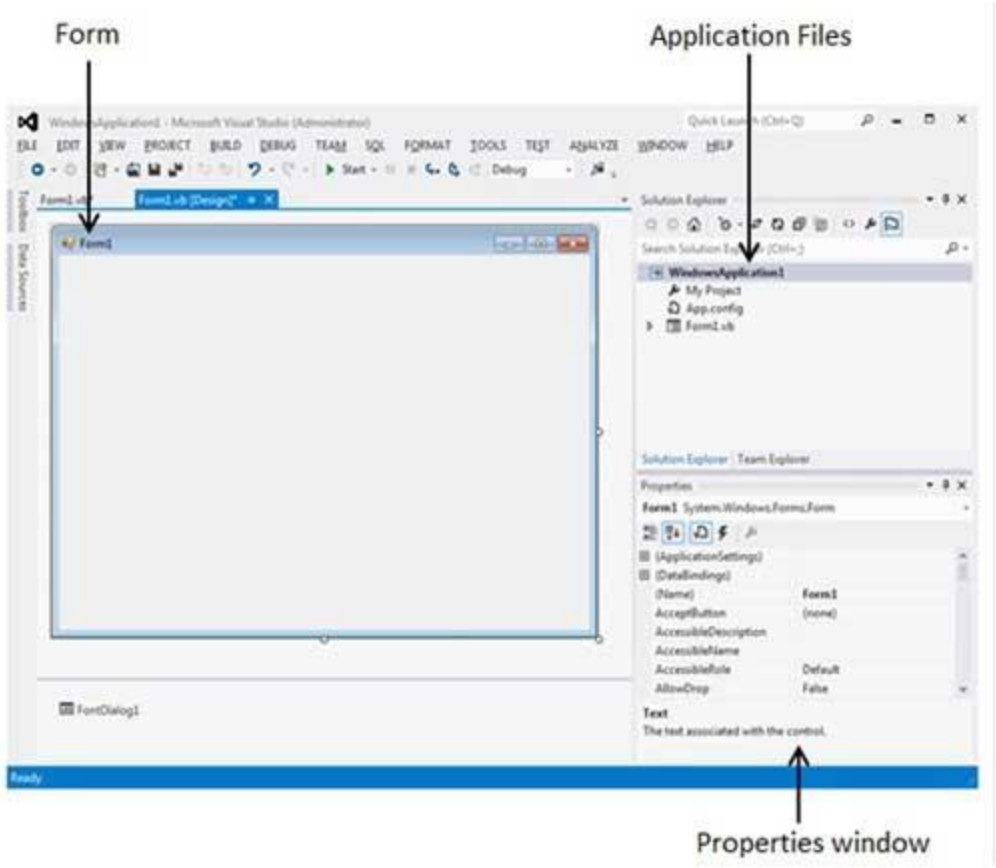
S.N.	Widget & Description
1	Forms The container for all the controls that make up the user interface.
2	TextBox It represents a Windows text box control.
3	Label It represents a standard Windows label.
4	Button It represents a Windows button control.
5	ListBox It represents a Windows control to display a list of items.
6	ComboBox It represents a Windows combo box control.
7	RadioButton It enables the user to select a single option from a group of choices when paired with other RadioButton controls.
8	CheckBox It represents a Windows CheckBox.
9	PictureBox It represents a Windows picture box control for displaying an image.
10	ProgressBar It represents a Windows progress bar control.

11	ScrollBar It Implements the basic functionality of a scroll bar control.
12	DateTimePicker It represents a Windows control that allows the user to select a date and a time and to display the date and time with a specified format.
13	TreeView It displays a hierarchical collection of labeled items, each represented by a TreeNode.
14	ListView It represents a Windows list view control, which displays a collection of items that can be displayed using one of four different views.

Forms

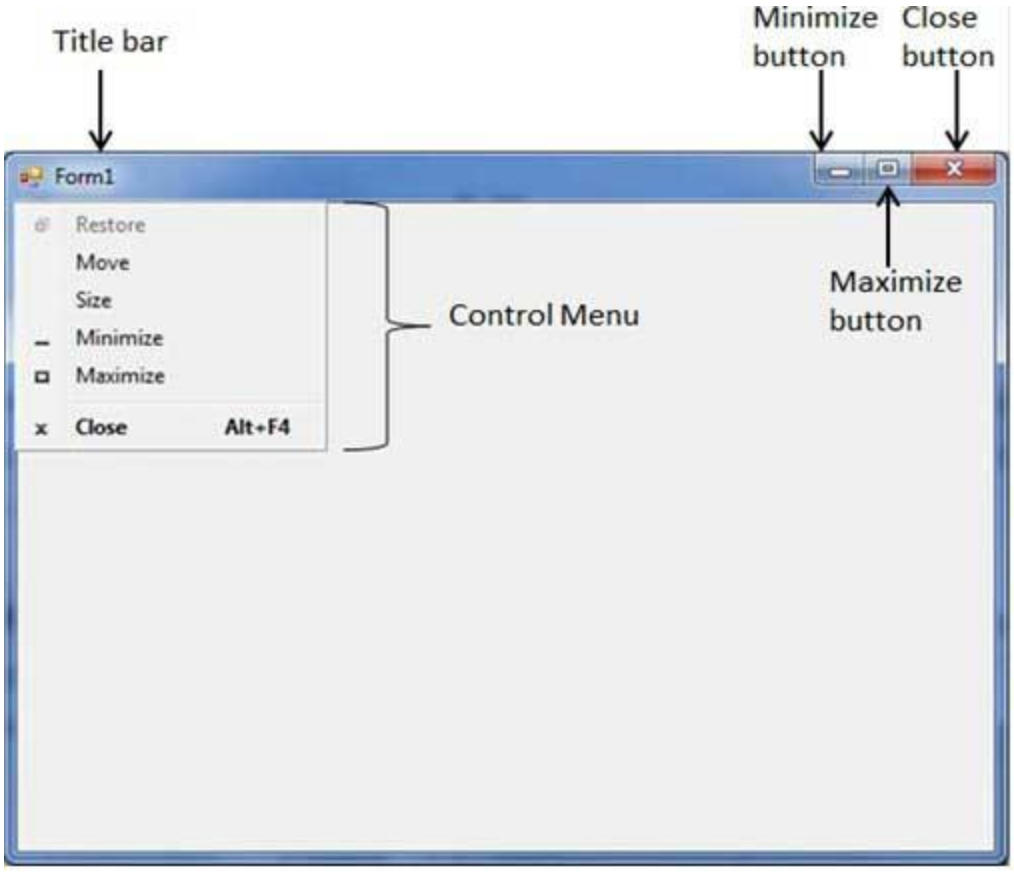
Let's start with creating a Window Forms Application by following the following steps in Microsoft Visual Studio: **File -> New Project -> Windows Forms Applications**

Finally, select OK, Microsoft Visual Studio creates your project and displays following window Form with a name **Form1**.



Visual Basic Form is the container for all the controls that make up the user interface. Every window you see in a running visual basic application is a form, thus the terms form and window describe the same entity. Visual Studio creates a default form for you when you create a **Windows Forms Application**.

Every form will have title bar on which the form's caption is displayed and there will be buttons to close, maximize and minimize the form shown below:



If you click the icon on the top left corner, it opens the control menu, which contains the various commands to control the form like to move control from one place to another place, to maximize or minimize the form or to close the form.

Form Properties

Following table lists down various important properties related to a form. These properties can be set or read during application execution. You can refer to Microsoft documentation for a complete list of properties associated with a Form control:

S.N	Properties	Description
1	**AcceptButton**	The button that's automatically activated when you press Enter, no matter which control has the focus at the time. Usually the OK button on a form is set as AcceptButton for a form.
2	**CancelButton**	The button that's automatically activated when you hit the Esc key.

		Usually, the Cancel button on a form is set as CancelButton for a form.
3	**AutoScale**	This Boolean property determines whether the controls you place on the form are automatically scaled to the height of the current font. The default value of this property is True. This is a property of the form, but it affects the controls on the form.
4	**AutoScroll**	This Boolean property indicates whether scroll bars will be automatically attached to the form if it is resized to a point that not all its controls are visible.
5	**AutoScrollMinSize**	This property lets you specify the minimum size of the form, before the scroll bars are attached.
6	**AutoScrollPosition**	The AutoScrollPosition is the number of pixels by which the two scroll bars were displaced from their initial locations.
7	**BackColor**	Sets the form background color.
8	**BorderStyle**	The BorderStyle property determines the style of the form's border and the appearance of the form: • **None**: Borderless window that can't be resized. • **Sizable**: This is default value and will be used for resizable window that's used for displaying regular forms. • **Fixed3D**: Window with a visible border, "raised" relative to the main area. In this case, windows can't be resized. • **FixedDialog**: A fixed window, used to create dialog boxes. • **FixedSingle**: A fixed window with a single line border.