VB.NET - QUICK GUIDE

 Copyright © tutorialspoint.com

VB.NET - OVERVIEW

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

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

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

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

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

Strong Programming Features VB.Net

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

- Boolean Conditions
- Automatic Garbage Collection
- Standard Library
- Assembly Versioning
- Properties and Events
- Delegates and Events Management
- Easy to use Generics
- Indexers
- Conditional Compilation

- Simple Multithreading

VB.NET - ENVIRONMENT

Integrated Development Environment (IDE) For VB.Net

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

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

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

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

Writing VB.Net Programs on Linux or Mac OS

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

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

VB.NET - PROGRAM STRUCTURE

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

VB.Net Hello World Example

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

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

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

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

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

```
Hello, World!
```

Let us look various parts of the above program:

- The first line of the program **Imports System** is used to include the System namespace in the program.
- The next line has a **Module** declaration, the module *Module1*. VB.Net is completely object oriented, so every program must contain a module of a class that contains the data and procedures that your program uses.
- Classes or Modules generally would contain more than one procedure. Procedures contain the executable code, or in other words, they define the behavior of the class. A procedure could be any of the following:
    - Function
    - Sub
    - Operator
    - Get
    - Set
    - AddHandler
    - RemoveHandler
    - RaiseEvent
- The next line('This program) will be ignored by the compiler and it has been put to add additional comments in the program.
- The next line defines the Main procedure, which is the entry point for all VB.Net programs. The Main procedure states what the module or class will do when executed.
- The Main procedure specifies its behavior with the statement

  Console.WriteLine("Hello World")

  WriteLine is a method of the *Console* class defined in the *System* namespace. This statement causes the message "Hello, World!" to be displayed on the screen.
- The last line **Console.ReadKey()** is for the VS.NET Users. This will prevent the screen from running and closing quickly when the program is launched from Visual Studio .NET.

Compile & Execute VB.Net Program:

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

- Start Visual Studio.
- On the menu bar, choose File, New, Project.
- Choose Visual Basic from templates
- Choose Console Application.
- Specify a name and location for your project using the Browse button, and then choose the OK button.
- The new project appears in Solution Explorer.
- Write code in the Code Editor.
- Click the Run button or the F5 key to run the project. A Command Prompt window appears that contains the line Hello World.

VB.NET - BASIC SYNTAX

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

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

- **Object** - Objects have states and behaviors. Example: A dog has states-color, name, breed as well as behaviors - wagging, barking, eating etc. An object is an instance of a class.
- **Class** - A class can be defined as a template/ blue print that describe the behaviors/states that object of its type support.
- **Methods** - A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
- **Instant Variables** - Each object has its unique set of instant variables. An object's state is created by the values assigned to these instant variables.

A Rectangle Class in VB.Net

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

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

```
Imports System
Public Class Rectangle
    Private length As Double
    Private width As Double

    'Public methods
    Public Sub AcceptDetails()
        length = 4.5
        width = 3.5
    End Sub

    Public Function GetArea() As Double
        GetArea = length * width
```

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

    End Sub

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

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

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

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

An object is an instance of a class:

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

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

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

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

Identifiers

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

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

VB.Net Keywords

The following table lists the VB.Net reserved keywords:

AddHandler	AddressOf	Alias	And	AndAlso	As	Boolean
ByRef	Byte	ByVal	Call	Case	Catch	CBool
CByte	CChar	CDate	CDec	CDbl	Char	CInt
Class	CLng	CObj	Const	Continue	CSByte	CShort
CSng	CStr	CType	CUInt	CULng	CUShort	Date
Decimal	Declare	Default	Delegate	Dim	DirectCast	Do
Double	Each	Else	ElseIf	End	End If	Enum
Erase	Error	Event	Exit	False	Finally	For
Friend	Function	Get	GetType	GetXML Namespace	Global	GoTo
Handles	If	Implements	Imports	In	Inherits	Integer
Interface	Is	IsNot	Let	Lib	Like	Long
Loop	Me	Mod	Module	MustInherit	MustOverride	MyBase
MyClass	Namespace	Narrowing	New	Next	Not	Nothing
Not Inheritable	Not Overridable	Object	Of	On	Operator	Option
Optional	Or	OrElse	Overloads	Overridable	Overrides	ParamArray
Partial	Private	Property	Protected	Public	RaiseEvent	ReadOnly
ReDim	REM	Remove Handler	Resume	Return	SByte	Select
Set	Shadows	Shared	Short	Single	Static	Step
Stop	String	Structure	Sub	SyncLock	Then	Throw
To	True	Try	TryCast	TypeOf	UInteger	While
Widening	With	WithEvents	WriteOnly	Xor		

VB.NET - DATA TYPES

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

Data Types Available in VB.Net

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

Data Type	Storage Allocation	Value Range

Boolean	Depends on implementing platform	**True** or **False**
Byte	1 byte	0 through 255 (unsigned)
Char	2 bytes	0 through 65535 (unsigned)
Date	8 bytes	0:00:00 (midnight) on January 1, 0001 through 11:59:59 PM on December 31, 9999
Decimal	16 bytes	0 through +/-79,228,162,514,264,337,593,543,950,335 (+/-7.9...E+28)with no decimal point; 0 through +/-7.9228162514264337593543950335 with 28 places to the right of the decimal
Double	8 bytes	-1.79769313486231570E+308 through -4.94065645841246544E-324, for negative values 4.94065645841246544E-324 through 1.79769313486231570E+308, for positive values
Integer	4 bytes	-2,147,483,648 through 2,147,483,647 (signed)
Long	8 bytes	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807(signed)
Object	4 bytes on 32-bit platform 8 bytes on 64-bit platform	Any type can be stored in a variable of type Object
SByte	1 byte	-128 through 127 (signed)
Short	2 bytes	-32,768 through 32,767 (signed)
Single	4 bytes	-3.4028235E+38 through -1.401298E-45 for negative values; 1.401298E-45 through 3.4028235E+38 for positive values
String	Depends on implementing platform	0 to approximately 2 billion Unicode characters
UInteger	4 bytes	0 through 4,294,967,295 (unsigned)
ULong	8 bytes	0 through 18,446,744,073,709,551,615 (unsigned)
User-Defined	Depends on implementing platform	Each member of the structure has a range determined by its data type and independent of the ranges of the other members
UShort	2 bytes	0 through 65,535 (unsigned)

Example

The following example demonstrates use of some of the types:

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

End Module
```

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

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

The Type Conversion Functions in VB.Net

VB.Net provides the following inline type conversion functions:

S.N	Functionss & Description
1	**CBool(expression)** Converts the expression to Boolean data type.
2	**CByte(expression)** Converts the expression to Byte data type.
3	**CChar(expression)** Converts the expression to Char data type.
4	**CDate(expression)**

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

Example:

The following example demonstrates some of these functions:

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

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

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

VB.NET - VARIABLES

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in VB.Net has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

We have already discussed various data types. The basic value types provided in VB.Net can be categorized as:

Type	Example
Integral types	SByte, Byte, Short, UShort, Integer, UInteger, Long, ULong and Char
Floating point types	Single and Double
Decimal types	Decimal
Boolean types	True or False values, as assigned
Date types	Date

VB.Net also allows defining other value types of variable like **Enum** and reference types of variables like **Class**. We will discuss date types and Classes, in subsequent chapters.

Variable Declaration in VB.Net

The **Dim** statement is used for variable declaration and storage allocation for one or more variables. The Dim statement is used at module, class, structure, procedure or block level.

Syntax for variable declaration in VB.Net is:

```
[ < attributelist> ] [ accessmodifier ] [[ Shared ] [ Shadows ] | [ Static ]]
[ ReadOnly ] Dim [ WithEvents ] variablelist
```

Where,

- ***attributelist*** is a list of attributes that apply to the variable. Optional.
- ***accessmodifier*** defines the access levels of the variables, it has values as - Public, Protected, Friend, Protected Friend and Private. Optional.
- ***Shared*** declares a shared variable, which is not associated with any specific instance of a class or structure, rather available to all the instances of the class or structure. Optional.
- ***Shadows*** indicate that the variable re-declares and hides an identically named element, or set of overloaded elements, in a base class. Optional.

- ***Static*** indicates that the variable will retain its value, even when the after termination of the procedure in which it is declared. Optional.
- ***ReadOnly*** means the variable can be read, but not written. Optional.
- ***WithEvents*** specifies that the variable is used to respond to events raised by the instance assigned to the variable. Optional.
- ***Variablelist*** provides the list of variables declared.

Each variable in the variable list has the following syntax and parts:

```
variablename[ ( [ boundslist ] ) ] [ As [ New ] datatype ] [ = initializer ]
```

Where,

- ***variablename***: is the name of the variable
- ***boundslist***: optional. It provides list of bounds of each dimension of an array variable.
- ***New***: optional. It creates a new instance of the class when the Dim statement runs.
- ***datatype***: Required if Option Strict is On. It specifies the data type of the variable.
- ***initializer***: Optional if New is not specified. Expression that is evaluated and assigned to the variable when it is created.

Some valid variable declarations along with their definition are shown here:

```
Dim StudentID As Integer
Dim StudentName As String
Dim Salary As Double
Dim count1, count2 As Integer
Dim status As Boolean
Dim exitButton As New System.Windows.Forms.Button
Dim lastTime, nextTime As Date
```

Variable Initialization in VB.Net

Variables are initialized (assigned a value) with an equal sign followed by a constant expression. The general form of initialization is:

```
variable_name = value;
```

for example,

```
Dim pi As Double
pi = 3.14159
```

You can initialize a variable at the time of declaration as follows:

```
Dim StudentID As Integer = 100
Dim StudentName As String = "Bill Smith"
```

Example

Try following example which makes use of various types of variables:

```
Module variablesNdataypes
   Sub Main()
      Dim a As Short
      Dim b As Integer
      Dim c As Double
      a = 10
      b = 20
      c = a + b
      Console.WriteLine("a = {0}, b = {1}, c = {2}", a, b, c)
      Console.ReadLine()
   End Sub
End Module
```

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

```
a = 10, b = 20, c = 30
```

Accepting Values from User

The Console class in the System namespace provides a function **ReadLine** for accepting input from the user and store it into a variable. For example,

```
Dim message As String
message = Console.ReadLine
```

The following example demonstrates it:

```
Module variablesNdataypes
   Sub Main()
      Dim message As String
      Console.Write("Enter message: ")
      message = Console.ReadLine
      Console.WriteLine()
      Console.WriteLine("Your Message: {0}", message)
      Console.ReadLine()
   End Sub
End Module
```

When the above code is compiled and executed, it produces following result(Assume the user inputs Hello World):

```
Enter message: Hello World
Your Message: Hello World
```

VB.NET - CONSTANTS AND ENUMERATIONS

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

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

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

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

Declaring Constants

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

The syntax for the Const statement is:

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

Where,

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

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

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

- ***constantname***: specifies the name of the constant
- ***datatype***: specifies the data type of the constant
- ***initializer***: specifies the value assigned to the constant

For example,

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

Example

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

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

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

```
Area = 153.933
```

Print and Display Constants in VB.Net

VB.Net provides the following print and display constants:

Constant	Description

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

Declaring Enumerations

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

The syntax for the Enum statement is as follows:

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

Where,

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

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

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

Where,

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

For example,

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

Example

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

```
Module constantsNenum
   Enum Colors
      red = 1
      orange = 2
      yellow = 3
      green = 4
      azure = 5
      blue = 6
      violet = 7
   End Enum
   Sub Main()
      Console.WriteLine("The Color Red is : " & Colors.red)
      Console.WriteLine("The Color Yellow is : " & Colors.yellow)
      Console.WriteLine("The Color Blue is : " & Colors.blue)
      Console.WriteLine("The Color Green is : " & Colors.green)
      Console.ReadKey()
   End Sub
End Module
```

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

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

VB.NET - MODIFIERS

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

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

List of Available Modifiers in VB.Net

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

S.N	Modifier	Description
1	Ansi	Specifies that Visual Basic should marshal all strings to American National Standards Institute (ANSI) values regardless of the name of the external procedure being declared.
2	Assembly	Specifies that an attribute at the beginning of a source file applies to the entire

		assembly.
3	Async	Indicates that the method or lambda expression that it modifies is asynchronous. Such methods are referred to as async methods. The caller of an async method can resume its work without waiting for the async method to finish..
4	Auto	The *charsetmodifier* part in the **Declare** statement supplies the character set information for marshaling strings during a call to the external procedure. It also affects how Visual Basic searches the external file for the external procedure name. The Auto modifier specifies that Visual Basic should marshal strings according to .NET Framework rules.
5	ByRef	Specifies that an argument is passed by reference, i.e., the called procedure can change the value of a variable underlying the argument in the calling code. It is used under the contexts of: • Declare Statement • Function Statement • Sub Statement
6	ByVal	Specifies that an argument is passed in such a way that the called procedure or property cannot change the value of a variable underlying the argument in the calling code. It is used under the contexts of: • Declare Statement • unction Statement • Operator Statement • Property Statement • Sub Statement
7	Default	Identifies a property as the default property of its class, structure, or interface.
8	Friend	Specifies that one or more declared programming elements are accessible from within the assembly that contains their declaration, not only by the component that declares them. **Friend** access is often the preferred level for an application's programming elements, and Friend is the default access level of an interface, a module, a class, or a structure.
9	In	It is used in generic interfaces and delegates.
10	Iterator	Specifies that a function or Get accessor is an iterator. An **iterator** performs a custom iteration over a collection.
11	Key	The Key keyword enables you to specify behavior for properties of anonymous types.
12	Module	Specifies that an attribute at the beginning of a source file applies to the current assembly module. It is not same as the Module statement.
13	MustInherit	Specifies that a class can be used only as a base class and that you cannot create

		an object directly from it.
14	MustOverride	Specifies that a property or procedure is not implemented in this class and must be overridden in a derived class before it can be used.
15	Narrowing	Indicates that a conversion operator (CType) converts a class or structure to a type that might not be able to hold some of the possible values of the original class or structure.
16	NotInheritable	Specifies that a class cannot be used as a base class.
17	NotOverridable	Specifies that a property or procedure cannot be overridden in a derived class.
18	Optional	Specifies that a procedure argument can be omitted when the procedure is called.
19	Out	For generic type parameters, the Out keyword specifies that the type is covariant.
20	Overloads	Specifies that a property or procedure redeclares one or more existing properties or procedures with the same name.
21	Overridable	Specifies that a property or procedure can be overridden by an identically named property or procedure in a derived class.
22	Overrides	Specifies that a property or procedure overrides an identically named property or procedure inherited from a base class.
23	ParamArray	ParamArray allows you to pass an arbitrary number of arguments to the procedure. A ParamArray parameter is always declared using ByVal.
24	Partial	Indicates that a class or structure declaration is a partial definition of the class or structure.
25	Private	Specifies that one or more declared programming elements are accessible only from within their declaration context, including from within any contained types.
26	Protected	Specifies that one or more declared programming elements are accessible only from within their own class or from a derived class.
27	Public	Specifies that one or more declared programming elements have no access restrictions.
28	ReadOnly	Specifies that a variable or property can be read but not written.
29	Shadows	Specifies that a declared programming element redeclares and hides an identically named element, or set of overloaded elements, in a base class.
30	Shared	Specifies that one or more declared programming elements are associated with a class or structure at large, and not with a specific instance of the class or structure.
31	Static	Specifies that one or more declared local variables are to continue to exist and retain their latest values after termination of the procedure in which they are declared.

32	Unicode	Specifies that Visual Basic should marshal all strings to Unicode values regardless of the name of the external procedure being declared.
33	Widening	Indicates that a conversion operator (CType) converts a class or structure to a type that can hold all possible values of the original class or structure.
34	WithEvents	Specifies that one or more declared member variables refer to an instance of a class that can raise events.
35	WriteOnly	Specifies that a property can be written but not read.

VB.NET - STATEMENTS

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

Statements could be categorized as:

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

Declaration Statements

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

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

Following are the declaration statements in VB.Net:

S.N	Statements and Description	Example

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

2	**Const Statement**
	Declares and defines one or more constants.
		`Const maximum As Long = 1000`
		`Const naturalLogBase As Object`
		`= CDec(2.7182818284)`

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

		Small End Enum
4	**Class Statement** Declares the name of a class and introduces the definition of the variables, properties, events, and procedures that the class comprises.	Class Box Public length As Double Public breadth As Double Public height As Double End Class
5	**Structure Statement** Declares the name of a structure and introduces the definition of the variables, properties, events, and procedures that the structure comprises.	Structure Box Public length As Double Public breadth As Double Public height As Double End Structure
6	**Module Statement** Declares the name of a module and introduces the definition of the variables, properties, events, and procedures that the module comprises.	Public Module myModule Sub Main() Dim user As String = InputBox("What is your name?") MsgBox("User name is" & user) End Sub End Module
7	**Interface Statement** Declares the name of an interface and introduces the definitions of the members that the interface comprises.	Public Interface MyInterface Sub doSomething() End Interface
8	**Function Statement** Declares the name, parameters, and code that define a Function procedure.	Function myFunction (ByVal n As Integer) As Double Return 5.87 * n End Function
9	**Sub Statement** Declares the name, parameters, and code that define a Sub procedure.	Sub mySub(ByVal s As String) Return End Sub
10	**Declare Statement** Declares a reference to a procedure implemented in an external file.	Declare Function getUserName Lib "advapi32.dll" Alias "GetUserNameA" (ByVal lpBuffer As String, ByRef nSize As Integer) As Integer
11	**Operator Statement** Declares the operator symbol, operands, and code that define an operator procedure on a class or structure.	Public Shared Operator + (ByVal x As obj, ByVal y As obj) As obj Dim r As New obj

		```' implemention code for r = x + y Return r End Operator```
12	**Property Statement** Declares the name of a property, and the property procedures used to store and retrieve the value of the property.	```ReadOnly Property quote() As String Get Return quoteString End Get End Property```
13	**Event Statement** Declares a user-defined event.	```Public Event Finished()```
14	**Delegate Statement** Used to declare a delegate.	```Delegate Function MathOperator(ByVal x As Double, ByVal y As Double) As Double```

Executable Statements

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

Example

The following example demonstrates a decision making statement:

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

      ' check the boolean condition using if statement '
      If (a < 20) Then
         ' if condition is true then print the following '
         Console.WriteLine("a is less than 20")
      End If
      Console.WriteLine("value of a is : {0}", a)
      Console.ReadLine()
   End Sub
End Module
```

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

```
a is less than 20;
value of a is : 10
```

VB.NET - DIRECTIVES

The VB.Net compiler directives give instruction to the compiler to preprocess the information before actual compilation starts.

All these directives begin with #, and only white-space characters may appear before a directive on a line. These

directives are not statements.

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

Compiler Directives in VB.Net

VB.Net provides the following set of compiler directives:

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

The #Const Directive

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

```
#Const constname = expression
```

Where,

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

For example,

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

Example

The following code demonstrates a hypothetical use of the directive:

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

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

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

The #ExternalSource Directive

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

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

Syntax for this directive is:

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

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

Example

The following code demonstrates a hypothetical use of the directive:

```
Module mydirectives
    Public Class ExternalSourceTester

        Sub TestExternalSource()

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

        End Sub
    End Class

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

    End Sub
```

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

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

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

This directive conditionally compiles selected blocks of Visual Basic code.

Syntax for this directive is:

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

For example,

```
#Const TargetOS = "Linux"
#If TargetOS = "Windows 7" Then
   ' Windows 7 specific code
#ElseIf TargetOS = "WinXP" Then
   ' Windows XP specific code
#Else
```

```
   ' Code for other OS
#End if
```

Example

The following code demonstrates a hypothetical use of the directive:

```
Module mydirectives
#Const classCode = 8

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

    End Sub
End Module
```

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

```
Exam Questions for Class VIII
```

The #Region Directive

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

Syntax for this directive is:

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

For example,

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

VB.NET - OPERATORS

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. VB.Net is rich in built-in operators and provides following type of commonly used operators:

- Arithmetic Operators
- Comparison Operators
- Logical/Bitwise Operators
- Bit Shift Operators
- Assignment Operators
- Miscellaneous Operators

This tutorial will explain the most commonly used operators.

Arithmetic Operators

Following table shows all the arithmetic operators supported by VB.Net. Assume variable **A** holds 2 and variable **B** holds 7 then:

Show Examples

Operator	Description	Example
^	Raises one operand to the power of another	B^A will give 49
+	Adds two operands	A + B will give 9
-	Subtracts second operand from the first	A - B will give -5
*	Multiply both operands	A * B will give 14
/	Divide one operand by another and returns a floating point result	B / A will give 3.5
\	Divide one operand by another and returns an integer result	B \ A will give 3
MOD	Modulus Operator and remainder of after an integer division	B MOD A will give 1

Comparison Operators

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

Show Examples

Operator	Description	Example
==	Checks if the value of two operands is equal or not, if yes then condition becomes true.	(A == B) is not true.
<>	Checks if the value of two operands is equal or not, if values are not equal then condition becomes true.	(A <> B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Apart from the above, VB.Net provides three more comparison operators, which we will be using in forthcoming chapters; however, we give a brief description here.

- **Is** Operator - It compares two object reference variables and determines if two object references refer to the same object without performing value comparisons. If object1 and object2 both refer to the exact same object instance, result is **True**; otherwise, result is False.
- **IsNot** Operator - It also compares two object reference variables and determines if two object references refer to different objects. If object1 and object2 both refer to the exact same object instance, result is **False**; otherwise, result is True.
- **Like** Operator - It compares a string against a pattern.

Logical/Bitwise Operators

Following table shows all the logical operators supported by VB.Net. Assume variable A holds Boolean value True and variable B holds Boolean value False then:

Show Examples

Operator	Description	Example
And	It is the logical as well as bitwise AND operator. If both the operands are true then condition becomes true. This operator does not perform short-circuiting, i.e., it evaluates both the expressions.	(A And B) is False.
Or	It is the logical as well as bitwise OR operator. If any of the two operands is true then condition becomes true. This operator does not perform short-circuiting, i.e., it evaluates both the expressions.	(A Or B) is True.
Not	It is the logical as well as bitwise NOT operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	Not(A And B) is True.
Xor	It is the logical as well as bitwise Logical Exclusive OR operator. It returns True if both expressions are True or both expressions are False; otherwise it returns False. This operator does not perform short-circuiting, it always evaluates both expressions and there is no short-circuiting counterpart of this operator	A Xor B is True.
AndAlso	It is the logical AND operator. It works only on Boolean data. It performs short-circuiting.	(A AndAlso B) is False.
OrElse	It is the logical OR operator. It works only on Boolean data. It performs short-circuiting.	(A OrElse B) is True.
IsFalse	It determines whether an expression is False.	
IsTrue	It determines whether an expression is True.	

Bit Shift Operators

We have already discussed the bitwise operators. The bit shift operators perform the shift operations on binary values.

Before coming into the bit shift operators, let us understand the bit operations.

Bitwise operators work on bits and perform bit by bit operation. The truth tables for &, |, and ^ are as follows:

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; Now in binary format they will be as follows:

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

We have seen that the Bitwise operators supported by VB.Net are And, Or, Xor and Not. The Bit shift operators are >> and << for left shift and right shift respectively.

Assume that the variable A holds 60 and variable B holds 13 then:

Show Examples

Operator	Description	Example
And	Bitwise AND Operator copies a bit to the result if it exists in both operands.	(A AND B) will give 12 which is 0000 1100
Or	Binary OR Operator copies a bit if it exists in either operand.	(A Or B) will give 61 which is 0011 1101
Xor	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A Xor B) will give 49 which is 0011 0001
Not	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(Not A) will give -60 which is 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved	A >> 2 will give 15 which is 0000

	right by the number of bits specified by the right operand.	1111

Assignment Operators

There are following assignment operators supported by VB.Net:

Show Examples

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand(floating point division)	C /= A is equivalent to C = C / A
\=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand (Integer division)	C \= A is equivalent to C = C \A
^=	Exponentiation and assignment operator. It raises the left operand to the power of the right operand and assigns the result to left operand.	C^=A is equivalent to C = C ^ A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Concatenates a String expression to a String variable or property and assigns the result to the variable or property.	Str1 &= Str2 is same as Str1 = Str1 & Str2

Miscellaneous Operators

There are few other important operators supported by VB.Net.

Show Examples

Operator	Description	Example
AddressOf	Returns the address of a procedure.	`AddHandler Button1.Click, AddressOf Button1_Click`

Await	It is applied to an operand in an asynchronous method or lambda expression to suspend execution of the method until the awaited task completes.	`Dim result As res = Await AsyncMethodThatReturnsResult() Await AsyncMethod()`
GetType	It returns a Type object for the specified type. The Type object provides information about the type such as its properties, methods, and events.	`MsgBox(GetType(Integer).ToString())`
Function Expression	It declares the parameters and code that define a function lambda expression.	`Dim add5 = Function(num As Integer) num + 5 'prints 10 Console.WriteLine(add5(5))`
If	It uses short-circuit evaluation to conditionally return one of two values. The If operator can be called with three arguments or with two arguments.	`Dim num = 5 Console.WriteLine(If(num >= 0, "Positive", "Negative"))`

Operators Precedence in VB.Net

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example x = 7 + 3 * 2; Here x is assigned 13, not 20 because operator * has higher precedence than + so it first get multiplied with 3*2 and then adds into 7.

Here operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Show Examples

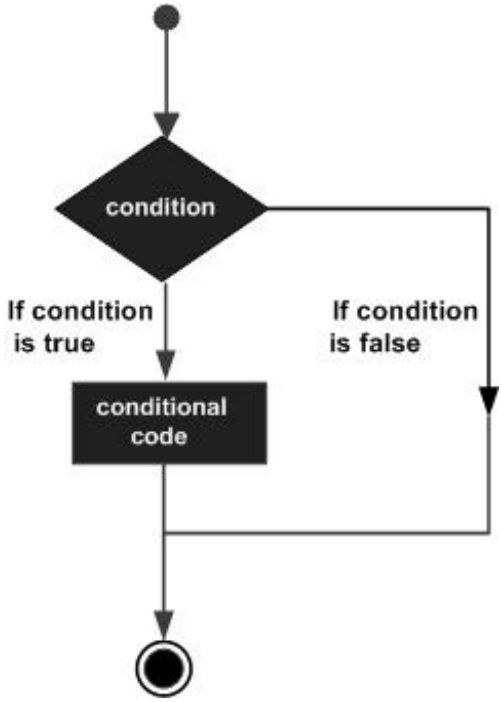
Operator	**Precedence**
Await	Highest
Exponentiation (^)	
Unary identity and negation (+, -)	
Multiplication and floating-point division (*, /)	
Integer division (\)	
Modulus arithmetic (Mod)	
Addition and subtraction (+, -)	
Arithmetic bit shift (<<, >>)	
All comparison operators (=, <>, <, <=, >, >=, Is, IsNot, Like,	

TypeOf...Is)	
Negation (Not)	
Conjunction (And, AndAlso)	
Inclusive disjunction (Or, OrElse)	
Exclusive disjunction (Xor)	Lowest

VB.NET - DECISION MAKING

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

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



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

Statement	Description
If ... Then statement	An **If...Then statement** consists of a boolean expression followed by one or more statements.
If...Then...Else statement	An **If...Then statement** can be followed by an optional **Else statement**, which executes when the boolean expression is false.
nested If statements	You can use one **If** or **Else if** statement inside another **If** or **Else if** statement(s).
Select Case statement	A **Select Case** statement allows a variable to be tested for equality against a list of values.

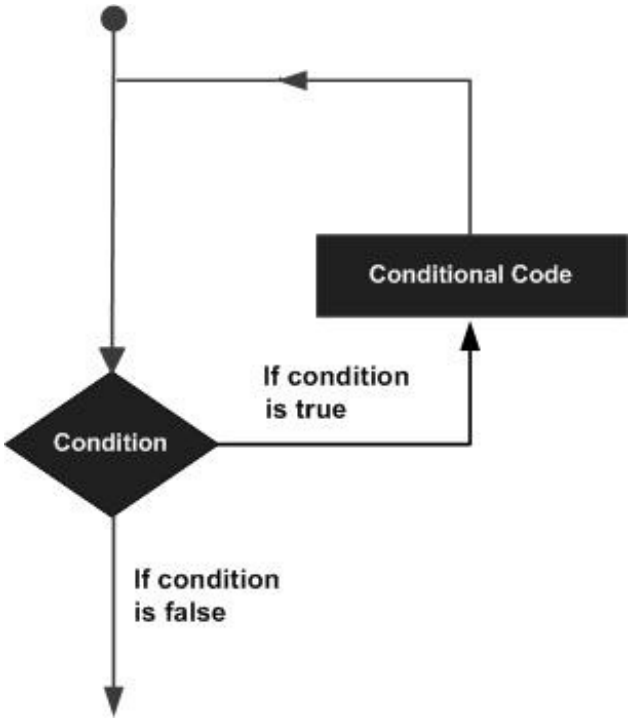
nested Select Case statements	You can use one **select case** statement inside another **select case** statement(s).

VB.NET - LOOPS

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

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

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



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

Loop Type	Description
Do Loop	It repeats the enclosed block of statements while a Boolean condition is True or until the condition becomes True. It could be terminated at any time with the Exit Do statement.
For...Next	It repeats a group of statements a specified number of times and a loop index counts the number of loop iterations as the loop executes.
For Each...Next	It repeats a group of statements for each element in a collection. This loop is used for accessing and manipulating all elements in an array or a VB.Net collection.
While... End While	It executes a series of statements as long as a given condition is True.
With... End With	It is not exactly a looping construct. It executes a series of statements that repeatedly refers to a single object or structure.

Nested loops	You can use one or more loop inside any another While, For or Do loop.

Loop Control Statements:

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

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

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

VB.NET - STRINGS

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

Creating a String Object

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

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

The following example demonstrates this:

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

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

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

```
      Dim message As String = String.Join(" ", sarray)
      Console.WriteLine("Message: {0}", message)

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

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

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

Properties of the String Class

The String class has the following two properties:

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

Methods of the String Class

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

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.

VB.NET - DATE & TIME

Most of the software you write needs 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 contains 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 produced 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 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 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 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, orT	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 following result:

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

VB.NET - ARRAYS

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 10 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 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 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. Multi-dimensional arrays are also called rectangular array.

You can declare a 2 dimensional array of strings as:

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

or, a three dimensional array of Integer variables:

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

The following program demonstrates creating and using a two 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 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 follwoing 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 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** eturns a string that represents the current object. (Inherited from Object.)

For complete list of Array class properties and methods, please consult 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 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
```

VB.NET - 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 C#.

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 detail.

Class	Description and Useage
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.

VB.NET - 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, where 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,

Tutorial content goes here.....

- ***Modifiers***: specifiy 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 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 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 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 following result:

```
Average value is: 214.4
```

VB.NET - 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***: specifiy 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 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 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 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
```

VB.NET - 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.
- ***Partia***l 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 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 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 explain 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 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 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 explain 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 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 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 class or interface.

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 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 following result:

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

VB.Net supports multiple inheritance.

VB.NET - 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 and 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 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 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 following result:

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

VB.NET - 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 class that are used for performing various operation 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 to 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 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** class helps to accomplish it.
Reading from and Writing into Binary files It involves reading from and writing into binary files. The **BinaryReader** and **BinaryWriter** class helps to accomplish this.
Manipulating the Windows file system It gives a VB.Net programamer the ability to browse and locate Windows files and directories.

VB.NET - 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 also 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 to change.
- **Value** is the new property setting.

For example,

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

You can set any of the form properties using Properties Windows. 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 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.

VB.NET - DIALOG BOXES

There are many built-in dialog boxes to be used in Windows forms for various tasks like opening and saving files, printing a page, providing choices for colors, fonts, page set up etc. to the user of an application. These built-in dialog boxes reduce the developer's time and work load.

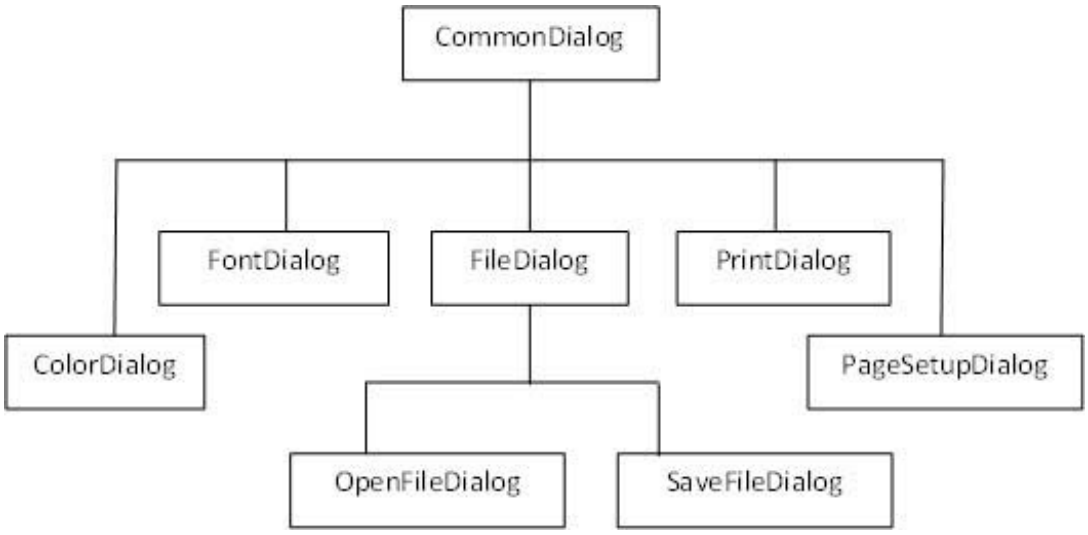
All of these dialog box control classes inherit from the **CommonDialog** class and override the *RunDialog()* function of the base class to create the specific dialog box.

The RunDialog() function is automatically invoked when a user of a dialog box calls its *ShowDialog()* function.

The **ShowDialog** method is used to display all the dialog box controls at run time. It returns a value of the type of **DialogResult** enumeration. The values of DialogResult enumeration are:

- **Abort** - returns DialogResult.Abort value, when user clicks an Abort button.
- **Cancel**- returns DialogResult.Cancel, when user clicks a Cancel button.
- **Ignore** - returns DialogResult.Ignore, when user clicks an Ignore button.
- **No** - returns DialogResult.No, when user clicks a No button.
- **None** - returns nothing and the dialog box continues running.
- **OK** - returns DialogResult.OK, when user clicks an OK button
- **Retry** - returns DialogResult.Retry , when user clicks an Retry button
- **Yes** - returns DialogResult.Yes, when user clicks an Yes button

The following diagram shows the common dialog class inheritance:



All these above mentioned classes have corresponding controls that could be added from the Toolbox during design time. You can include relevant functionality of these classes to your application, either by instantiating the class programmatically or by using relevant controls.

When you double click any of the dialog controls in the toolbox or drag the control onto the form, it appears in the Component tray at the bottom of the Windows Forms Designer, they do not directly show up on the form.

The following table lists the commonly used dialog box controls. Click the following links to check their detail:

S.N.	Control & Description
1	ColorDialog It represents a common dialog box that displays available colors along with controls that enable the user to define custom colors.
2	FontDialog It prompts the user to choose a font from among those installed on the local computer and lets the user select the font, font size, and color.
3	OpenFileDialog It prompts the user to open a file and allows the user to select a file to open.

4	SaveFileDialog It prompts the user to select a location for saving a file and allows the user to specify the name of the file to save data.
5	PrintDialog It lets the user to print documents by selecting a printer and choosing which sections of the document to print from a Windows Forms application.

VB.NET - ADVANCED FORMS

In this chapter, let us study the following concepts:

- Adding menus and sub menus in an application
- Adding the cut, copy and paste functionalities in a form
- Anchoring and docking controls in a form
- Modal forms

Adding Menus and Sub Menus in an Application

Traditionally, the *Menu*, *MainMenu*, *ContextMenu*, and *MenuItem* classes were used for adding menus, sub-menus and context menus in a Windows application.

Now, the **MenuStrip**, the **ToolStripMenuItem**, **ToolStripDropDown** and **ToolStripDropDownMenu** controls replace and add functionality to the Menu related controls of previous versions. However, the old control classes are retained for both backward compatibility and future use.

Let us create a typical windows main menu bar and sub menus using the old versions controls first. Since these controls are still much used in old applications.

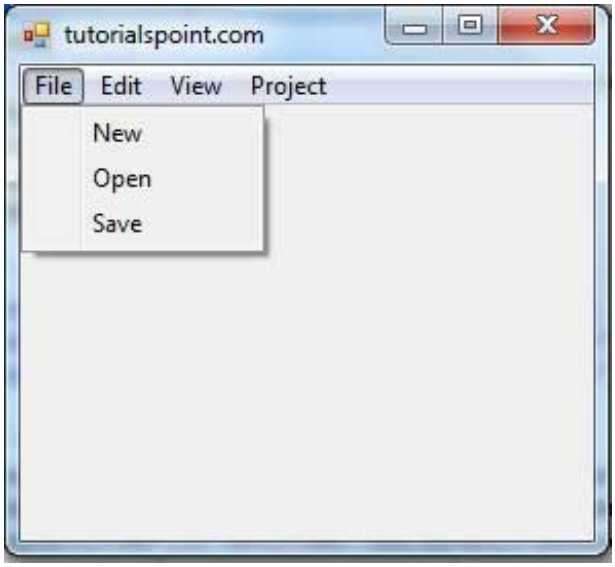
Following is an example which shows how we create a menu bar with menu items: File, Edit, View and Project. The File menu has the sub menus New, Open and Save.

Let's double click on the Form and put the follow code in the opened window.

```
Public Class Form1
   Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
      'defining the main menu bar
      Dim mnuBar As New MainMenu()
      'defining the menu items for the main menu bar
      Dim myMenuItemFile As New MenuItem("&File")
      Dim myMenuItemEdit As New MenuItem("&Edit")
      Dim myMenuItemView As New MenuItem("&View")
      Dim myMenuItemProject As New MenuItem("&Project")

      'adding the menu items to the main menu bar
      mnuBar.MenuItems.Add(myMenuItemFile)
      mnuBar.MenuItems.Add(myMenuItemEdit)
      mnuBar.MenuItems.Add(myMenuItemView)
      mnuBar.MenuItems.Add(myMenuItemProject)

     ' defining some sub menus
      Dim myMenuItemNew As New MenuItem("&New")
      Dim myMenuItemOpen As New MenuItem("&Open")
      Dim myMenuItemSave As New MenuItem("&Save")

      'add sub menus to the File menu
      myMenuItemFile.MenuItems.Add(myMenuItemNew)
```

```
      myMenuItemFile.MenuItems.Add(myMenuItemOpen)
      myMenuItemFile.MenuItems.Add(myMenuItemSave)

      'add the main menu to the form
      Me.Menu = mnuBar

      ' Set the caption bar text of the form.
      Me.Text = "tutorialspoint.com"
   End Sub
End Class
```

When the above code is executed and run using Start button available at the Microsoft Visual Studio tool bar, it will show following window:



Windows Forms contains a rich set of classes for creating your own custom menus with modern appearance and look and feel. The **MenuStrip**, **ToolStripMenuItem**, **ContextMenuStrip** controls are used to create menu bars and context menus efficiently.

Click the following links to check their detail:

S.N.	Control & Description
1	MenuStrip It provides a menu system for a form.
2	ToolStripMenuItem It Represents a selectable option displayed on a **MenuStrip** or **ContextMenuStrip**. The ToolStripMenuItem control replaces and adds functionality to the MenuItem control of previous versions.
2	ContextMenuStrip It represents a shortcut menu.

Adding the Cut, Copy and Paste Functionalities in a Form

The methods exposed by the **ClipBoard** class are used for adding the cut, copy and paste functionalities in an application. The ClipBoard class provides methods to place data on and retrieve data from the system Clipboard.

It has the following commonly used methods:

S.N	Method Name & Description
1	**Clear** Removes all data from the Clipboard.
2	**ContainsData** Indicates whether there is data on the Clipboard that is in the specified format or can be converted to that format.
3	**ContainsImage** Indicates whether there is data on the Clipboard that is in the Bitmap format or can be converted to that format.
4	**ContainsText** Indicates whether there is data on the Clipboard in the Text or UnicodeText format, depending on the operating system.
5	**GetData** Retrieves data from the Clipboard in the specified format.
6	**GetDataObject** Retrieves the data that is currently on the system Clipboard.
7	**GetImage** Retrieves an image from the Clipboard.
8	**GetText** Retrieves text data from the Clipboard in the Text or UnicodeText format, depending on the operating system.
9	**GetText(TextDataFormat)** Retrieves text data from the Clipboard in the format indicated by the specified TextDataFormat value.
10	**SetData** Clears the Clipboard and then adds data in the specified format.
11	**SetText(String)** Clears the Clipboard and then adds text data in the Text or UnicodeText format, depending on the operating system.

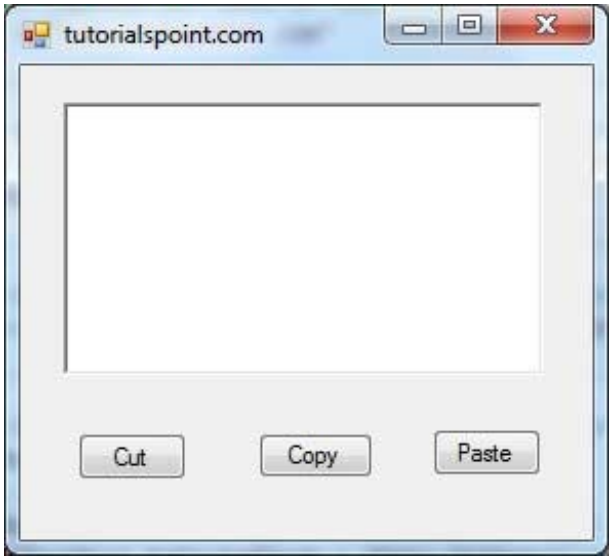
Following is an example which shows how we cut, copy and paste data using methods of the Clipboard class. Take the following steps:

1. Add a rich text box control and three button controls on the form.
2. Change the text property of the buttons to Cut, Copy and Paste respectively.
3. Double click on the buttons to add the following code in the code editor:

```
Public Class Form1
   Private Sub Form1_Load(sender As Object, e As EventArgs) _
    Handles MyBase.Load
      ' Set the caption bar text of the form.
      Me.Text = "tutorialspoint.com"
   End Sub
   Private Sub Button1_Click(sender As Object, e As EventArgs) _
```

```
      Handles Button1.Click
      Clipboard.SetDataObject(RichTextBox1.SelectedText)
      RichTextBox1.SelectedText = ""
   End Sub
   Private Sub Button2_Click(sender As Object, e As EventArgs) _
       Handles Button2.Click
      Clipboard.SetDataObject(RichTextBox1.SelectedText)
   End Sub
   Private Sub Button3_Click(sender As Object, e As EventArgs) _
    Handles Button3.Click
      Dim iData As IDataObject
      iData = Clipboard.GetDataObject()
      If (iData.GetDataPresent(DataFormats.Text)) Then
          RichTextBox1.SelectedText = iData.GetData(DataFormats.Text)
      Else
          RichTextBox1.SelectedText = " "
      End If
   End Sub
End Class
```

When the above code is executed and run using **Start** button available at the Microsoft Visual Studio tool bar, it will show following window:



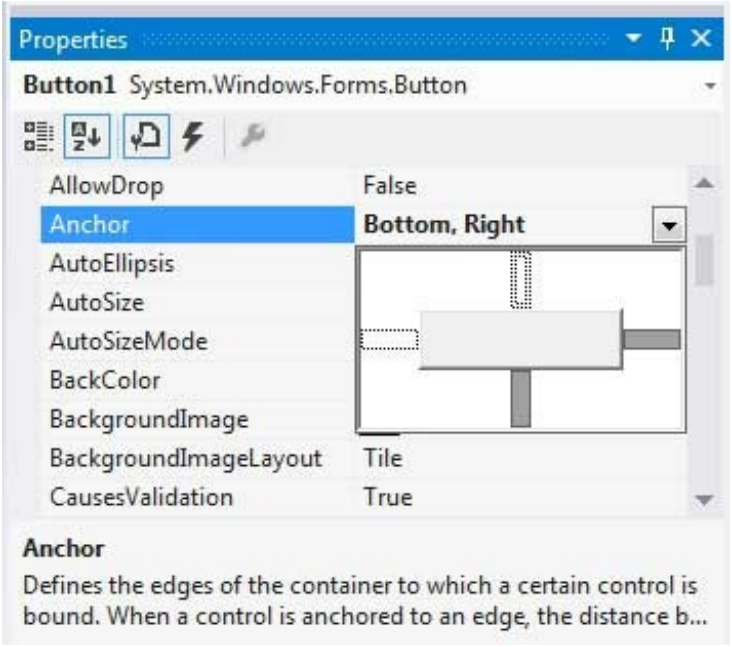
Enter some text and check how the buttons work.

Anchoring and Docking Controls in a Form

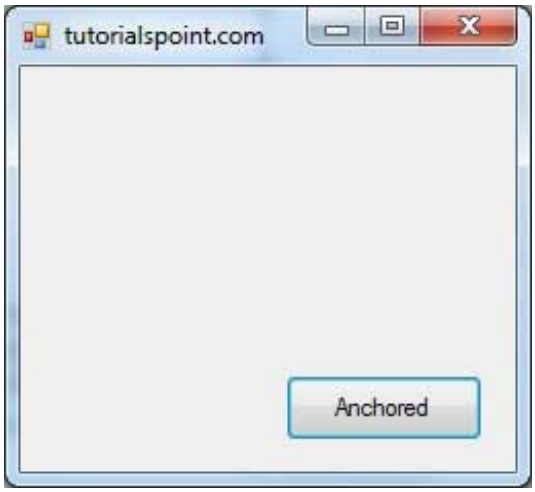
Anchoring allows you to set an anchor position for a control to the edges of its container control, for example, the form. The **Anchor** property of the Control class allows you to set values of this property. The Anchor property gets or sets the edges of the container to which a control is bound and determines how a control is resized with its parent.

When you anchor a control to a form, the control maintains its distance from the edges of the form and its anchored position, when the form is resized.

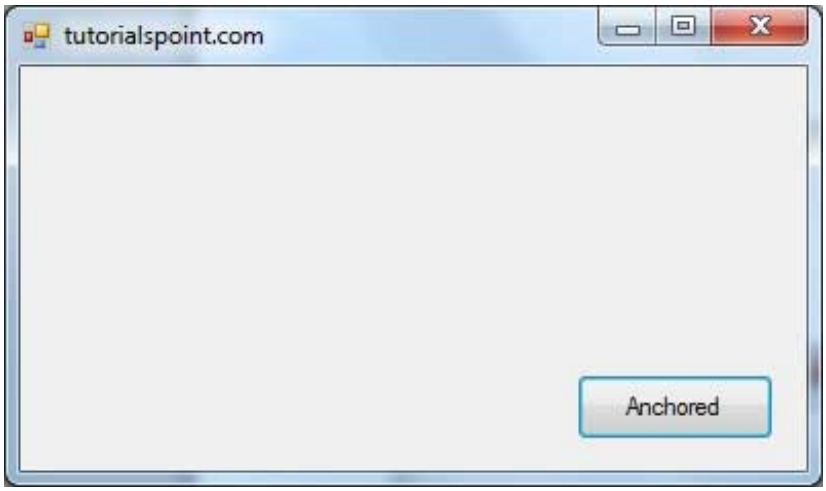
You can set the Anchor property values of a control from the Properties window:



For example, let us add a Button control on a form. And set its anchor property to Bottom, Right. Run this form to see the original position of the Button control with respect to the form.



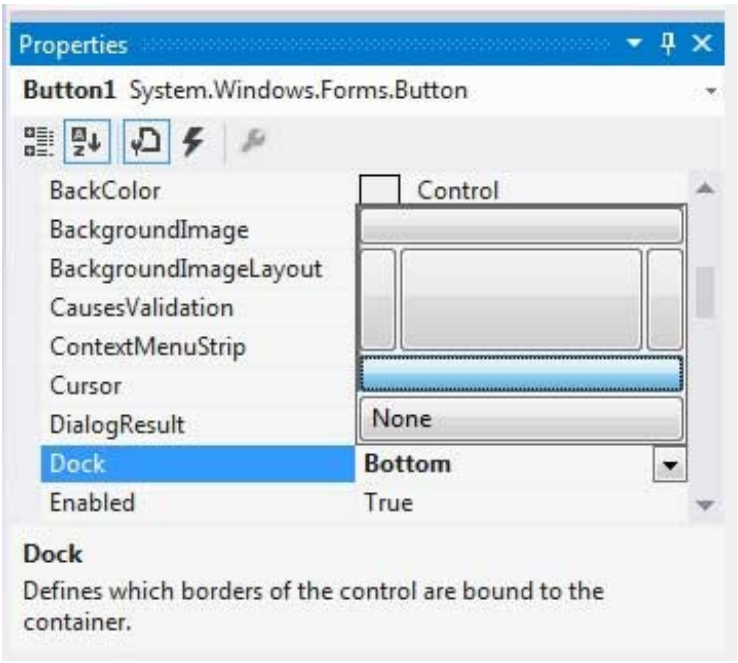
Now when you stretch the form, the distance between the Button and the bottom, right corner of the form remains same.



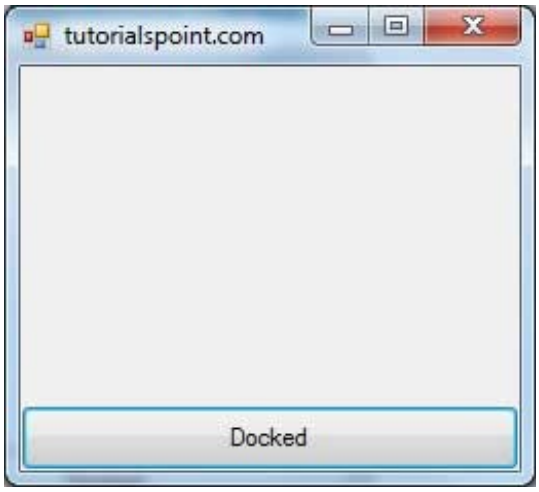
Docking of a control means docking it to one of the edges of its container. In docking, the control fills certain area of the container completely.

The **Dock** property of the Control class does this. The Dock property gets or sets which control borders are docked to its parent control and determines how a control is resized with its parent.

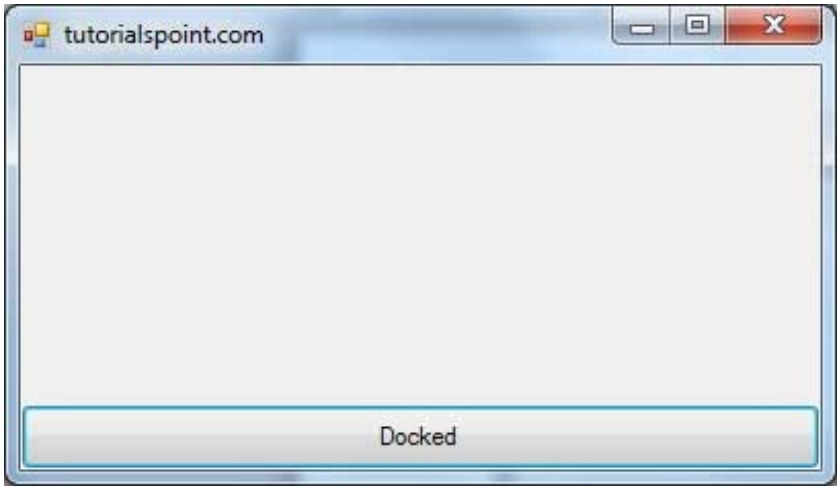
You can set the Dock property values of a control from the Properties window:



For example, let us add a Button control on a form. And set its Dock property to Bottom. Run this form to see the original position of the Button control with respect to the form.



Now when you stretch the form, the Button resizes itself with the form.



Modal Forms

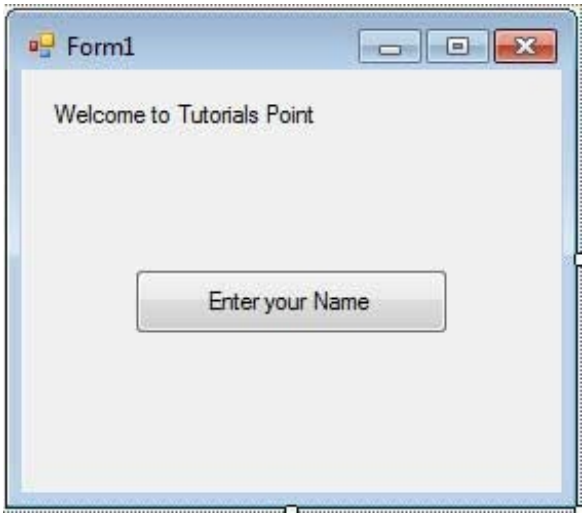
Modal Forms are those form that need to be closed or hidden before you can continue working with the rest of the application. All dialog boxes are modal forms. A MessageBox is also a modal form.

You can call a modal form by two ways:

- Calling the **ShowDialog** method
- Calling the **Show** method

Let us take up an example in which we will create a modal form, a dialog box. Take the following steps:

1. Add a form, Form1 to your application, and add two labels and a button control to Form1
2. Change the text properties of the first label and the button to 'Welcome to Tutorials Point' and 'Enter your Name' respectively. Keep the text properties of the second label as blank.



3. Add a new Windows Form, Form2, two buttons, one label, and a text box to Form2.
4. Change the text properties of the buttons to OK and Cancel respectively. Change the text properties of the label to 'Enter your name:'
5. Set the *FormBorderStyle* property of Form2 to *FixedDialog*, for giving it a dialog box border.
6. Set the *ControlBox* property of Form2 to False.

7. Set the *ShowInTaskbar* property of Form2 to False.

8. Set the *DialogResult* property of the OK button to OK and the Cancel button to Cancel.

Add the following code snippets in the Form2_Load method of Form2:

```
Private Sub Form2_Load(sender As Object, e As EventArgs) _
   Handles MyBase.Load
   AcceptButton = Button1
   CancelButton = Button2
End Sub
```

Add the following code snippets in the Button1_Click method of Form1:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) _
   Handles Button1.Click
   Dim frmSecond As Form2 = New Form2()
   If frmSecond.ShowDialog() = DialogResult.OK Then
      Label2.Text = frmSecond.TextBox1.Text
   End If
End Sub
```

When the above code is executed and run using **Start** button available at the Microsoft Visual Studio tool bar, it will show following window:

Clicking on the 'Enter your Name' button displays the second form:

Clicking on the OK button, takes control and information back from the modal form to the previous form:

VB.NET - EVENT HANDLING

Events are basically a user action like key press, clicks, mouse movements etc., or some occurrence like system generated notifications. Applications need to respond to events when they occur.

Clicking on a button, or entering some text in a text box, or clicking on a menu item all are examples of events. An event is an action that calls a function or may cause another event.

Event handlers are functions that tell how to respond to an event.

VB.Net is an event-driven language. There are mainly two types of events:

- Mouse events
- Keyboard events

Handling Mouse Events

Mouse events occur with mouse movements in forms and controls. Following are the various mouse events related with a Control class:

- **MouseDown** - it occurs when a mouse button is pressed
- **MouseEnter** - it occurs when the mouse pointer enters the control

- **MouseHover** - it occurs when the mouse pointer hovers over the control
- **MouseLeave** - it occurs when the mouse pointer leaves the control
- **MouseMove** - it occurs when the mouse pointer moves over the control
- **MouseUp** - it occurs when the mouse pointer is over the control and the mouse button is released
- **MouseWheel** - it occurs when the mouse wheel moves and the control has focus

The event handlers of the mouse events get an argument of type **MouseEventArgs**. The MouseEventArgs object is used for handling mouse events. It has the following properties:

- **Buttons** - indicates the mouse button pressed
- **Clicks** - indicates the number of clicks
- **Delta** - indicates the number of detents the mouse wheel rotated
- **X** - indicates the x-coordinate of mouse click
- **Y** - indicates the y-coordinate of mouse click

Example

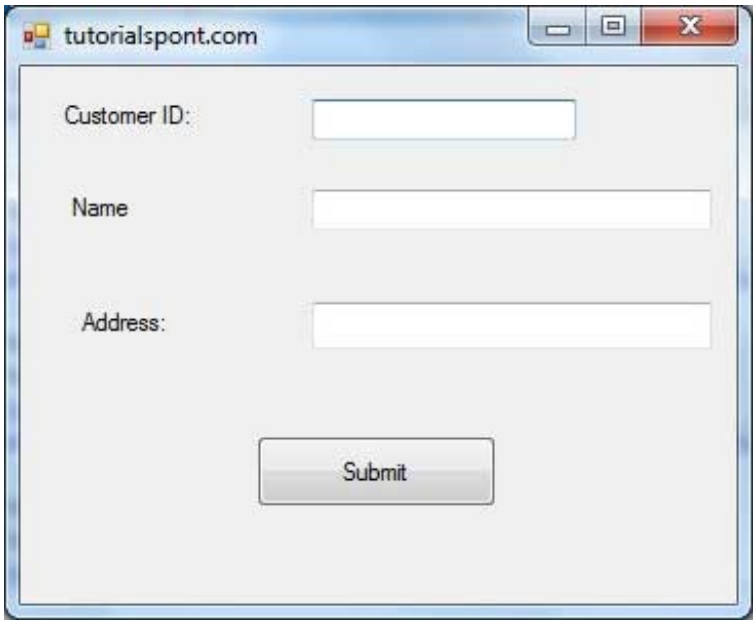
Following is an example which shows how to handle mouse events. Take the following steps:

1. Add three labels, three text boxes and a button control in the form.
2. Change the text properties of the labels to - Customer ID, Name and Address respectively.
3. Change the name properties of the text boxes to txtID, txtName and txtAddress respectively.
4. Change the text property of the button to 'Submit'
5. Add the following code in the code editor window:

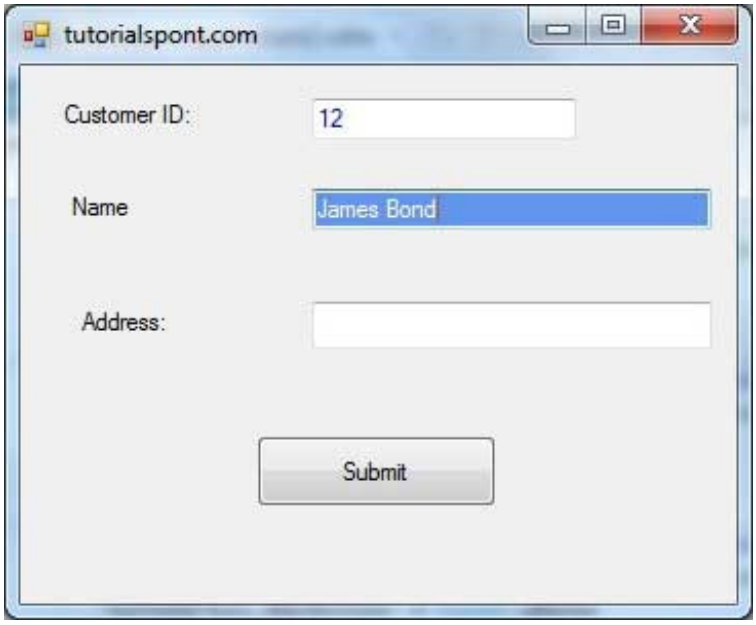
```
Public Class Form1
   Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
      ' Set the caption bar text of the form.
      Me.Text = "tutorialspont.com"
   End Sub

   Private Sub txtID_MouseEnter(sender As Object, e As EventArgs)_
        Handles txtID.MouseEnter
      'code for handling mouse enter on ID textbox
      txtID.BackColor = Color.CornflowerBlue
      txtID.ForeColor = Color.White
   End Sub
   Private Sub txtID_MouseLeave(sender As Object, e As EventArgs) _
        Handles txtID.MouseLeave
      'code for handling mouse leave on ID textbox
      txtID.BackColor = Color.White
      txtID.ForeColor = Color.Blue
   End Sub
   Private Sub txtName_MouseEnter(sender As Object, e As EventArgs) _
       Handles txtName.MouseEnter
      'code for handling mouse enter on Name textbox
      txtName.BackColor = Color.CornflowerBlue
      txtName.ForeColor = Color.White
   End Sub
   Private Sub txtName_MouseLeave(sender As Object, e As EventArgs) _
      Handles txtName.MouseLeave
```

```
      'code for handling mouse leave on Name textbox
      txtName.BackColor = Color.White
      txtName.ForeColor = Color.Blue
   End Sub
   Private Sub txtAddress_MouseEnter(sender As Object, e As EventArgs) _
      Handles txtAddress.MouseEnter
      'code for handling mouse enter on Address textbox
      txtAddress.BackColor = Color.CornflowerBlue
      txtAddress.ForeColor = Color.White
   End Sub
   Private Sub txtAddress_MouseLeave(sender As Object, e As EventArgs) _
        Handles txtAddress.MouseLeave
      'code for handling mouse leave on Address textbox
      txtAddress.BackColor = Color.White
      txtAddress.ForeColor = Color.Blue
   End Sub

   Private Sub Button1_Click(sender As Object, e As EventArgs) _
       Handles Button1.Click
      MsgBox("Thank you " & txtName.Text & ", for your kind cooperation")
   End Sub
End Class
```

When the above code is executed and run using **Start** button available at the Microsoft Visual Studio tool bar, it will show following window:



Try to enter text in the text boxes and check the mouse events:



Handling Keyboard Events

Following are the various keyboard events related with a Control class:

- **KeyDown** - occurs when a key is pressed down and the control has focus
- **KeyPress** - occurs when a key is pressed and the control has focus
- **KeyUp** - occurs when a key is released while the control has focus

The event handlers of the KeyDown and KeyUp events get an argument of type **KeyEventArgs**. This object has the following properties:

- **Alt** - it indicates whether the ALT key is pressed/p>
- **Control** - it indicates whether the CTRL key is pressed
- **Handled** - it indicates whether the event is handled
- **KeyCode** - stores the keyboard code for the event
- **KeyData** - stores the keyboard data for the event
- **KeyValue** - stores the keyboard value for the event
- **Modifiers** - it indicates which modifier keys (Ctrl, Shift, and/or Alt) are pressed
- **Shift** - it indicates if the Shift key is pressed

The event handlers of the KeyDown and KeyUp events get an argument of type **KeyEventArgs**. This object has the following properties:

- **Handled** - indicates if the KeyPress event is handled
- **KeyChar** - stores the character corresponding to the key pressed

Example

Let us continue with the previous example to show how to handle keyboard events. The code will verify that the user enters some numbers for his customer ID and age.

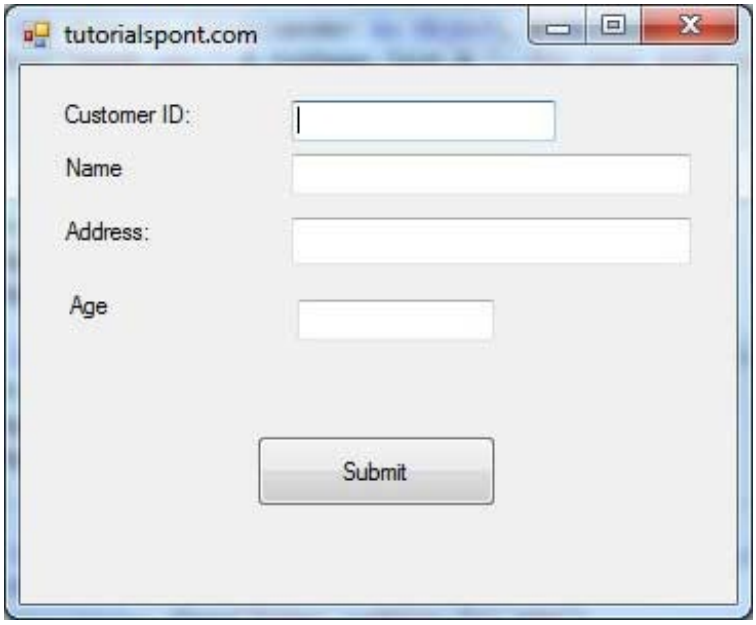
1. Add a label with text Property as 'Age' and add a corresponding text box named txtAge.
2. Add the following codes for handling the KeyUP evens of the text box txtID.

```
Private Sub txtID_KeyUP(sender As Object, e As KeyEventArgs) _
   Handles txtID.KeyUp
   If (Not Char.IsNumber(ChrW(e.KeyCode))) Then
      MessageBox.Show("Enter numbers for your Customer ID")
      txtID.Text = " "
   End If
End Sub
```

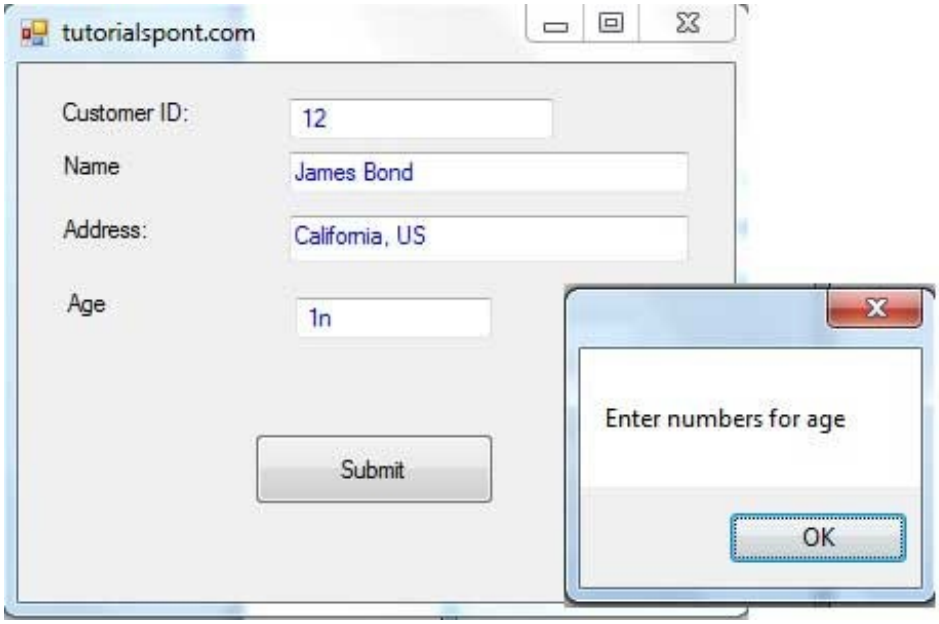
3. Add the following codes for handling the KeyUP evens of the text box txtID.

```
Private Sub txtAge_KeyUP(sender As Object, e As KeyEventArgs) _
   Handles txtAge.KeyUp
   If (Not Char.IsNumber(ChrW(e.keyCode))) Then
      MessageBox.Show("Enter numbers for age")
      txtAge.Text = " "
   End If
End Sub
```

When the above code is executed and run using **Start** button available at the Microsoft Visual Studio tool bar, it will show following window:



If you leave the text for age or ID as blank or enter some non-numeric data, it gives a warning message box and clears the respective text:



VB.NET - REGULAR EXPRESSIONS

A **regular expression** is a pattern that could be matched against an input text. The .Net framework provides a regular expression engine that allows such matching. A pattern consists of one or more character literals, operators, or constructs.

Constructs for Defining Regular Expressions

There are various categories of characters, operators, and constructs that lets you to define regular expressions. Click the follwoing links to find these constructs.

- Character escapes
- Character classes
- Anchors
- Grouping constructs
- Quantifiers
- Backreference constructs
- Alternation constructs
- Substitutions
- Miscellaneous constructs

The Regex Class

The Regex class is used for representing a regular expression.

The Regex class has the following commonly used methods:

S.N	Methods & Description
1	**Public Function IsMatch (input As String) As Boolean**

	Indicates whether the regular expression specified in the Regex constructor finds a match in a specified input string.
2	**Public Function IsMatch (input As String, startat As Integer) As Boolean** Indicates whether the regular expression specified in the Regex constructor finds a match in the specified input string, beginning at the specified starting position in the string.
3	**Public Shared Function IsMatch (input As String, pattern As String) As Boolean** Indicates whether the specified regular expression finds a match in the specified input string.
4	**Public Function Matches (input As String) As MatchCollection** Searches the specified input string for all occurrences of a regular expression.
5	**Public Function Replace (input As String, replacement As String) As String** In a specified input string, replaces all strings that match a regular expression pattern with a specified replacement string.
6	**Public Function Split (input As String) As String()** Splits an input string into an array of substrings at the positions defined by a regular expression pattern specified in the Regex constructor.

For the complete list of methods and properties, please consult Microsoft documentation.

Example 1

The following example matches words that start with 'S':

```
Imports System.Text.RegularExpressions
Module regexProg
   Sub showMatch(ByVal text As String, ByVal expr As String)
      Console.WriteLine("The Expression: " + expr)
      Dim mc As MatchCollection = Regex.Matches(text, expr)
      Dim m As Match
      For Each m In mc
          Console.WriteLine(m)
      Next m
   End Sub
   Sub Main()
      Dim str As String = "A Thousand Splendid Suns"
      Console.WriteLine("Matching words that start with 'S': ")
      showMatch(str, "\bS\S*")
      Console.ReadKey()
   End Sub
End Module
```

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

```
Matching words that start with 'S':
The Expression: \bS\S*
Splendid
Suns
```

Example 2

The following example matches words that start with 'm' and ends with 'e':

```
Imports System.Text.RegularExpressions
Module regexProg
   Sub showMatch(ByVal text As String, ByVal expr As String)
      Console.WriteLine("The Expression: " + expr)
```

```
      Dim mc As MatchCollection = Regex.Matches(text, expr)
      Dim m As Match
      For Each m In mc
          Console.WriteLine(m)
      Next m
   End Sub
   Sub Main()
      Dim str As String = "make a maze and manage to measure it"
      Console.WriteLine("Matching words that start with 'm' and ends with 'e': ")
      showMatch(str, "\bm\S*e\b")
      Console.ReadKey()
   End Sub
End Module
```

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

```
Matching words start with 'm' and ends with 'e':
The Expression: \bm\S*e\b
make
maze
manage
measure
```

Example 3

This example replaces extra white space:

```
Imports System.Text.RegularExpressions
Module regexProg
   Sub Main()
      Dim input As String = "Hello    World   "
      Dim pattern As String = "\\s+"
      Dim replacement As String = " "
      Dim rgx As Regex = New Regex(pattern)
      Dim result As String = rgx.Replace(input, replacement)
      Console.WriteLine("Original String: {0}", input)
      Console.WriteLine("Replacement String: {0}", result)
      Console.ReadKey()
   End Sub
End Module
```

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

```
Original String: Hello   World
Replacement String: Hello World
```

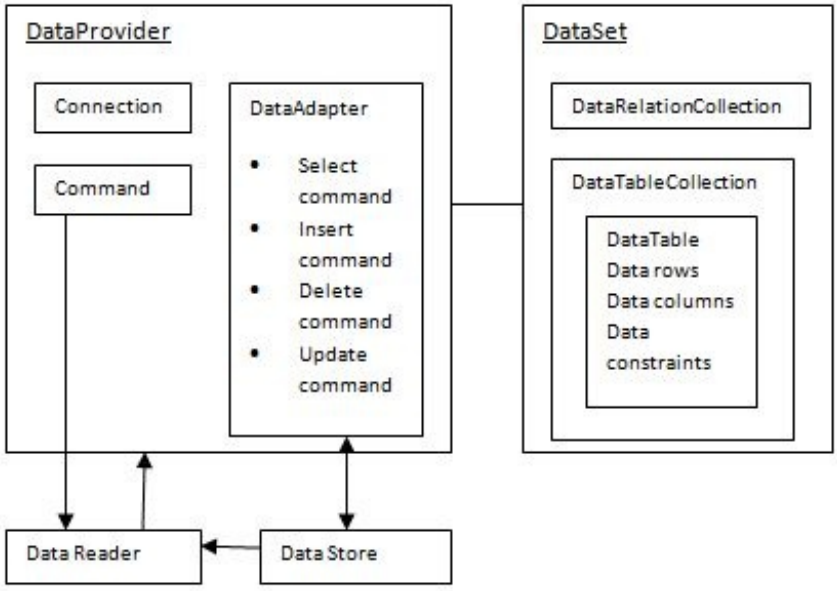
VB.NET - DATABASE ACCESS

Applications communicate with a database, firstly, to retrieve the data stored there and present it in a user-friendly way, and secondly, to update the database by inserting, modifying and deleting data.

Microsoft ActiveX Data Objects.Net (ADO.Net) is a model, a part of the .Net framework that is used by the .Net applications for retrieving, accessing and updating data.

ADO.Net Object Model

ADO.Net object model is nothing but the structured process flow through various components. The object model can be pictorially described as:



The data residing in a data store or database is retrieved through the **data provider**. Various components of the data provider, retrieves data for the application and update data.

An application accesses data either through a dataset or a data reader.

- **Data sets** store data in a disconnected cache and the application retrieve data from it.
- **Data readers** provide data to the application in a read-only and forward-only mode.

Data Provider

A data provider is used for connecting to a database, executing commands and retrieving data, storing it in a dataset, reading the retrieved data and updating the database.

The data provider in ADO.Net consists of the following four objects:

S.N	Objects & Description
1	**Connection** This component is used to set up a connection with a data source.
2	**Command** A command is a SQL statement or a stored procedure used to retrieve, insert, delete or modify data in a data source.
3	**DataReader** Data reader is used to retrieve data from a data source in a read-only and forward-only mode.
4	**DataAdapter** This is integral to the working of ADO.Net since data is transferred to and from a database through a data adapter. It retrieves data from a database into a dataset and updates the database. When changes are made to the dataset, the changes in the database are actually done by the data adapter.

There are following different types of data providers included in ADO.Net

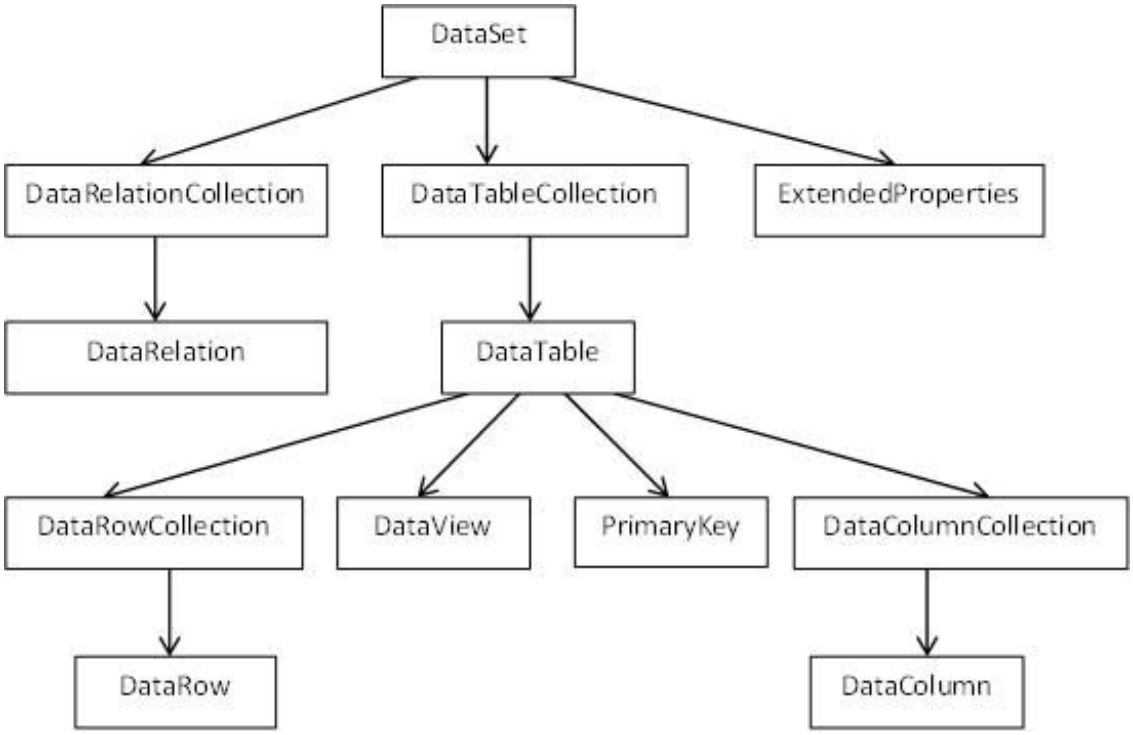
- The .Net Framework data provider for SQL Server - provides access to Microsoft SQL Server.

- The .Net Framework data provider for OLE DB - provides access to data sources exposed by using OLE DB.
- The .Net Framework data provider for ODBC - provides access to data sources exposed by ODBC.
- The .Net Framework data provider for Oracle - provides access to Oracle data source.
- The EntityClient provider - enables accessing data through Entity Data Model (EDM) applications.

DataSet

DataSet is an in-memory representation of data. It is a disconnected, cached set of records that are retrieved from a database. When a connection is established with the database, the data adapter creates a dataset and stores data in it. After the data is retrieved and stored in a dataset, the connection with the database is closed. This is called the 'disconnected architecture'. The dataset works as a virtual database, containing tables, rows, and columns.

The following diagram shows the dataset object model:



The DataSet class is present in the **System.Data** namespace. The following table describes all the components of DataSet:

S.N	Components & Description
1	**DataTableCollection** It contains all the tables retrieved from the data source.
2	**DataRelationCollection** It contains relationships and the links between tables in a data set.
3	**ExtendedProperties** It contains additional information, like the SQL statement for retrieving data, time of retrieval etc.
4	**DataTable** It represents a table in the DataTableCollection of a dataset. It consists of the dataRow and DataColumn

	objects. The DataTable objects are case-sensitive.
5	**DataRelation** It represents a relationship in the DataRelationshipCollection of the dataset. It is used to relate two DataTable objects to each other through the DataColumn objects.
6	**DataRowCollection** It contains all the rows in a DataTable.
7	**DataView** It represents a fixed customized view of a DataTable for sorting, filtering, searching, editing and navigation.
8	**PrimaryKey** It represents the column that uniquely identifies a row in a DataTable.
9	**DataRow** It represents a row in the DataTable. The DataRow object and its properties and methods are used to retrieve, evaluate, insert, delete, and update values in the DataTable. The NewRow method is used to create a new row and the Add method adds a row to the table.
10	**DataColumnCollection** It represents all the columns in a DataTable.
11	**DataColumn** It consists of the number columns that comprise a DataTable.

Connecting to a Database

The .Net Framework provides two types of Connection classes:

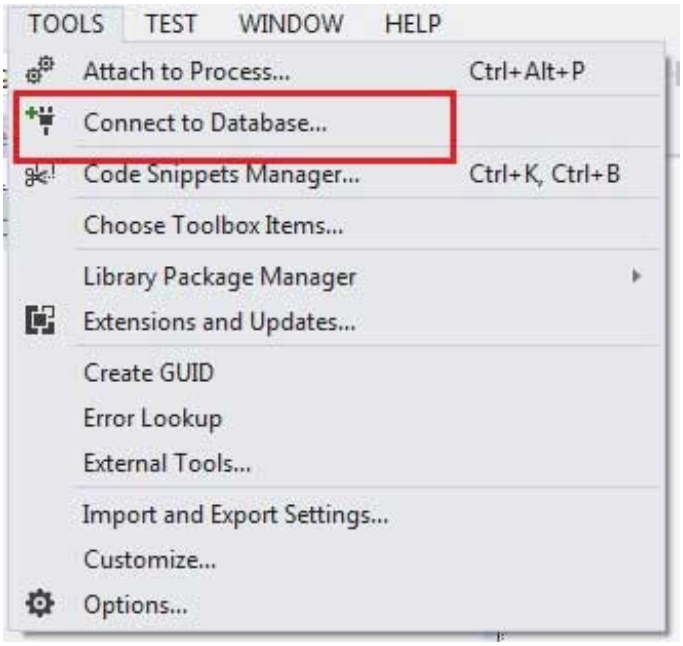
- **SqlConnection** - designed for connecting to Microsoft SQL Server.
- **OleDbConnection** - designed for connecting to a wide range of databases, like Microsoft Access and Oracle.

Example 1

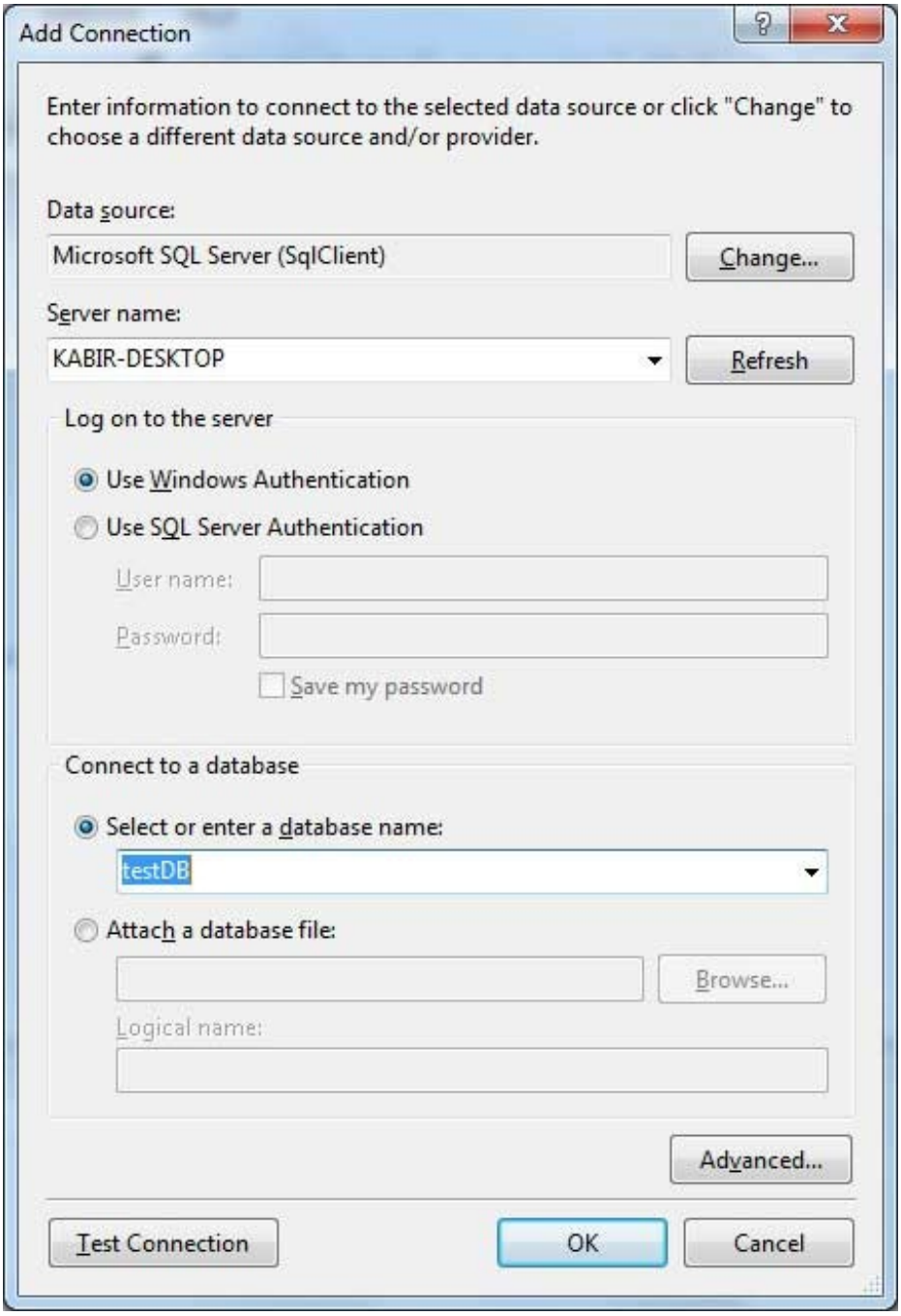
We have a table stored in Microsoft SQL Server, named Customers, in a database named testDB. Please consult 'SQL Server' tutorial for creating databases and database tables in SQL Server.

Let us connect to this database. Take the following steps:

1. Select TOOLS -> Connect to Database

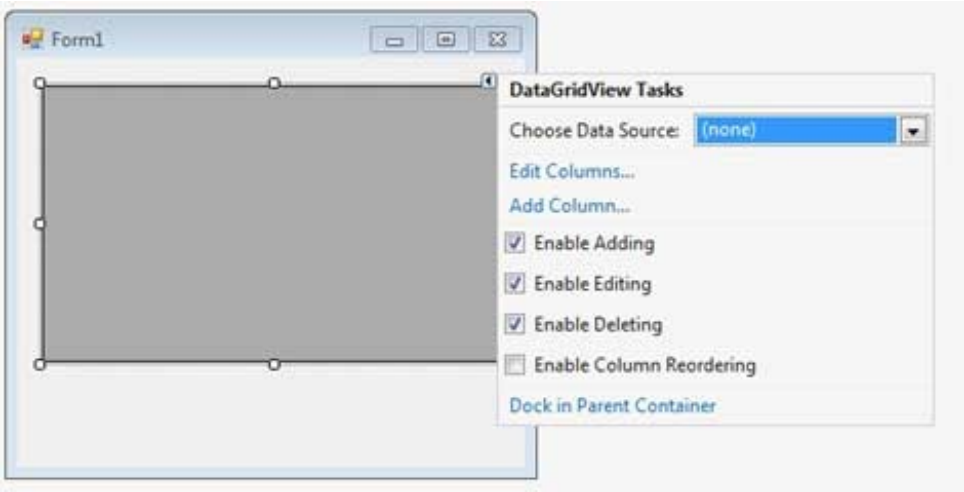


2. Select a server name and the database name in the Add Connection dialog box.



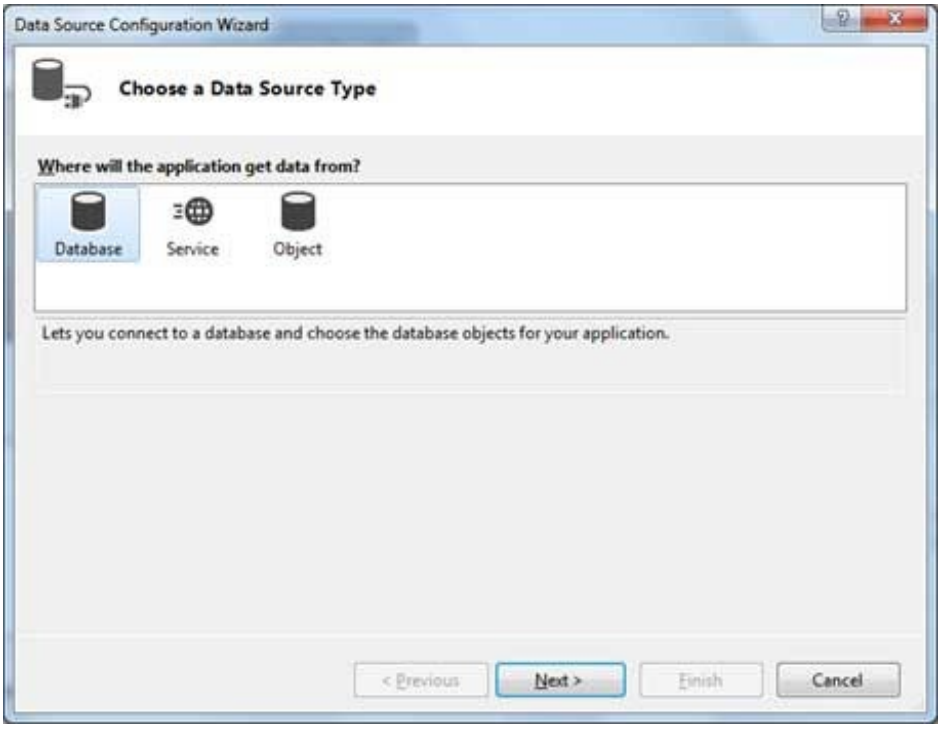
3. Click on the Test Connection button to check if the connection succeeded.

4. Add a DataGridView on the form.

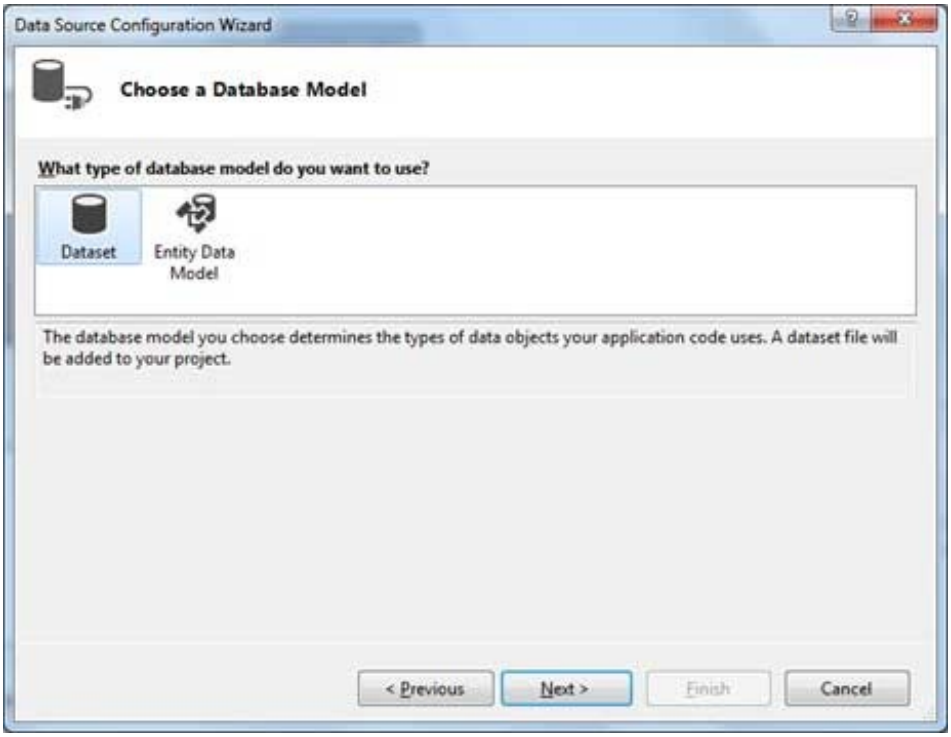


5. Click on the Choose Data Source combo box.
6. Click on the Add Project Data Source link.

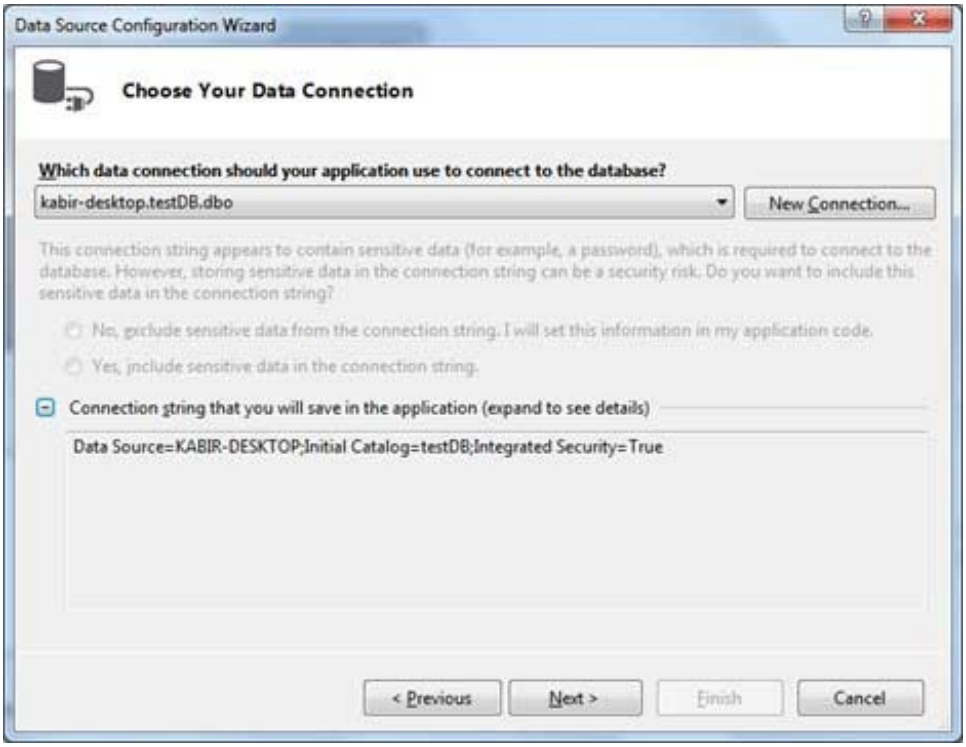
7. This opens the Data Source Configuration Wizard.
8. Select Database as the data source type



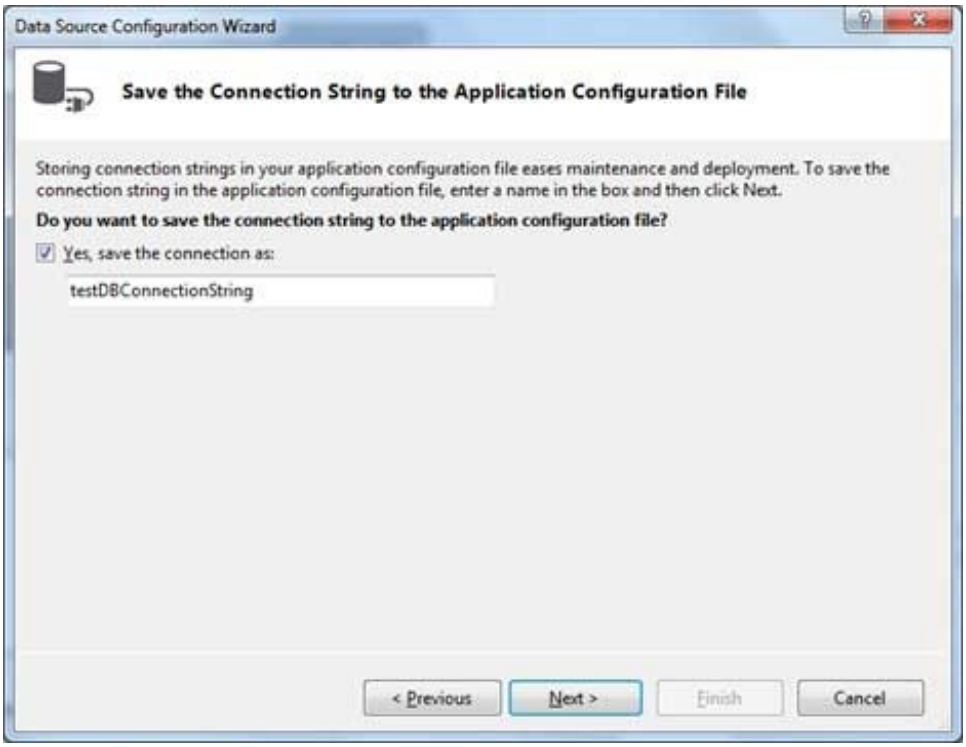
9. Choose Dataset as the database model.



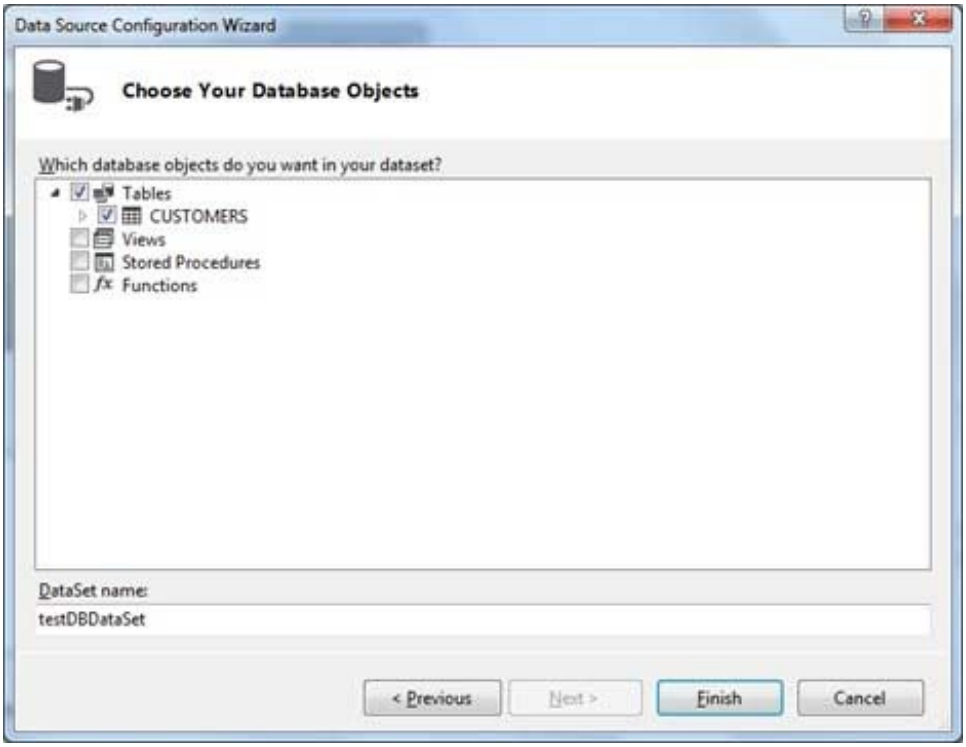
10. Choose the connection already set up



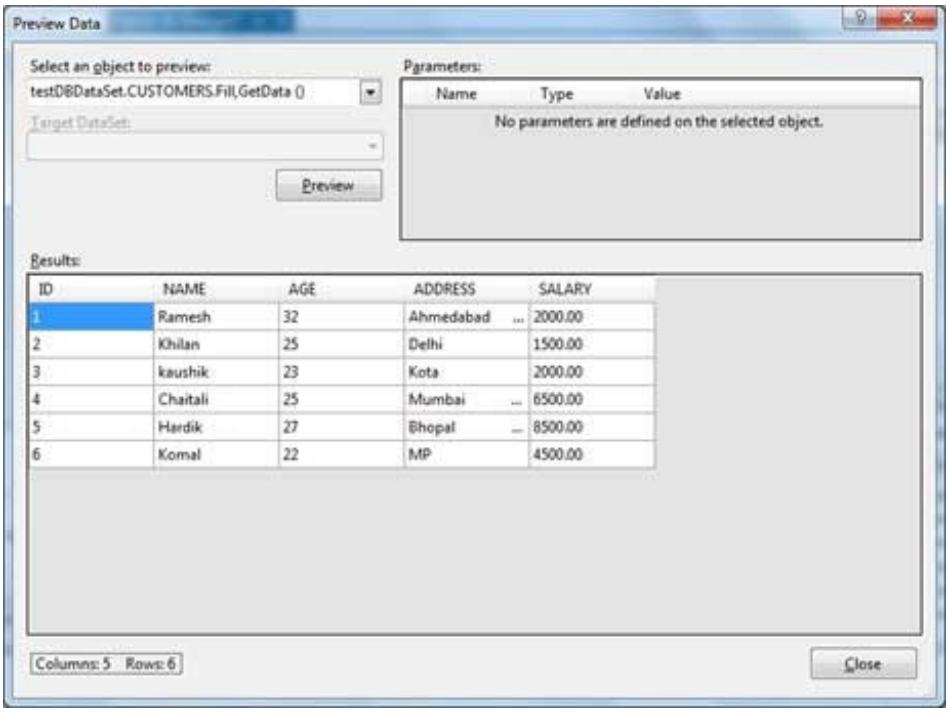
11. Save the connection string.



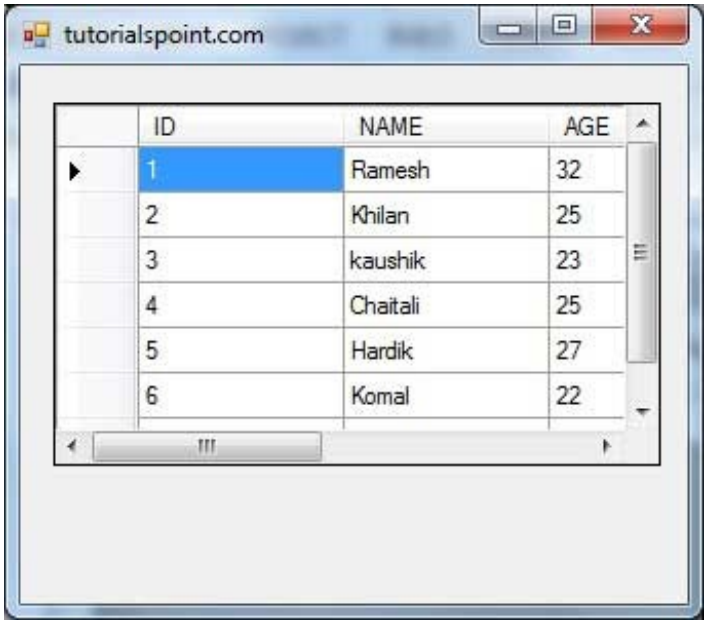
12. Choose the database object, Customers table in our example, and click the Finish button.



13. Select the Preview Data link to see the data in the Results grid:



When the application is run using **Start** button available at the Microsoft Visual Studio tool bar, it will show following window:



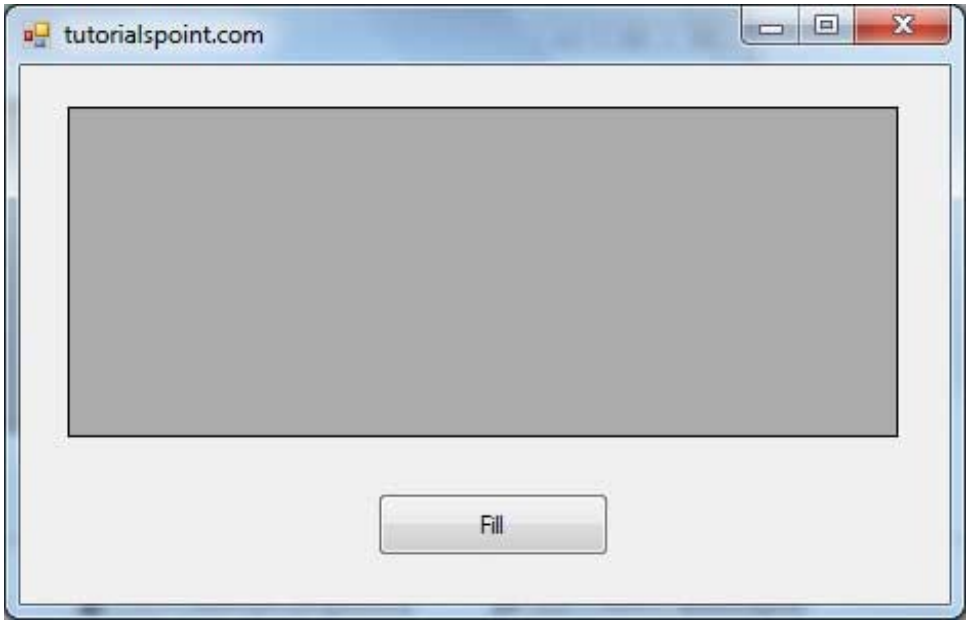
Example 2

In this example, let us access data in a DataGridView control using code. Take the following steps:

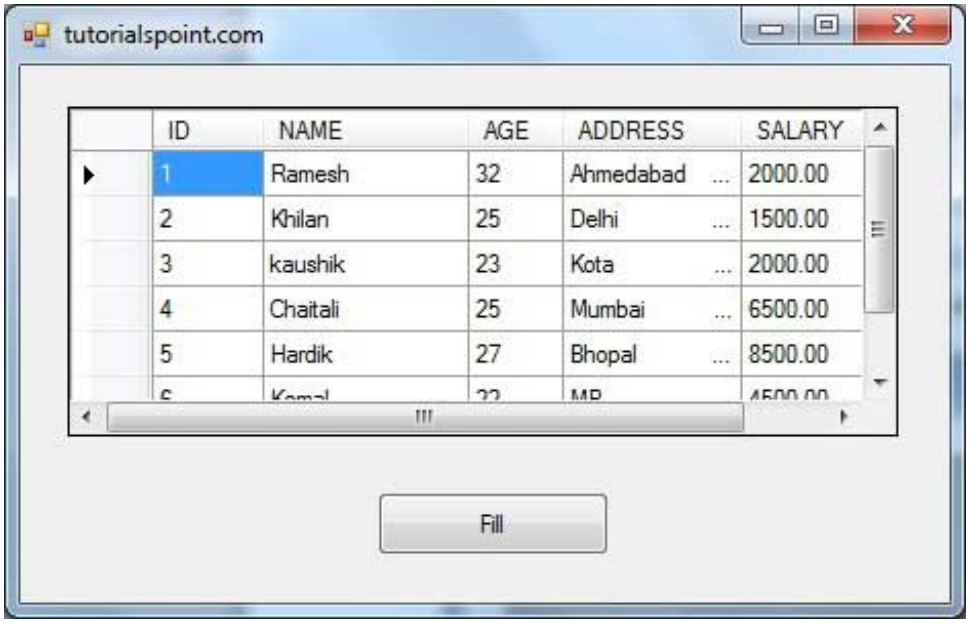
1. Add a DataGridView control and a button in the form.
2. Change the text of the button control to 'Fill'
3. Double click the button control to add the required code for the Click event of the button, as shown below:

```
Imports System.Data.SqlClient
Public Class Form1
   Private Sub Form1_Load(sender As Object, e As EventArgs) _
   Handles MyBase.Load
        'TODO: This line of code loads data into the 'TestDBDataSet.CUSTOMERS' table.   You can
move, or remove it, as needed.
      Me.CUSTOMERSTableAdapter.Fill(Me.TestDBDataSet.CUSTOMERS)
      ' Set the caption bar text of the form.
      Me.Text = "tutorialspoint.com"
   End Sub
   Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
      Dim connection As SqlConnection = New sqlconnection()
      connection.ConnectionString = "Data Source=KABIR-DESKTOP; _
          Initial Catalog=testDB;Integrated Security=True"
      connection.Open()
      Dim adp As SqlDataAdapter = New SqlDataAdapter _
      ("select * from Customers", connection)
      Dim ds As DataSet = New DataSet()
      adp.Fill(ds)
      DataGridView1.DataSource = ds.Tables(0)
   End Sub
End Class
```

When the above code is executed and run using **Start** button available at the Microsoft Visual Studio tool bar, it will show following window:



Clicking the Fill button, displays the table on the data grid view control:



Creating Table, Columns and Rows

We have discussed that the DataSet components like DataTable, DataColumn and DataRow allows us to create tables, columns and rows respectively.

The following example demonstrates the concept:

Example 3

So far, we have used tables and databases already existing in our computer. In this example, we will create a table, add column, rows and data into it and display the table using a DataGridView object.

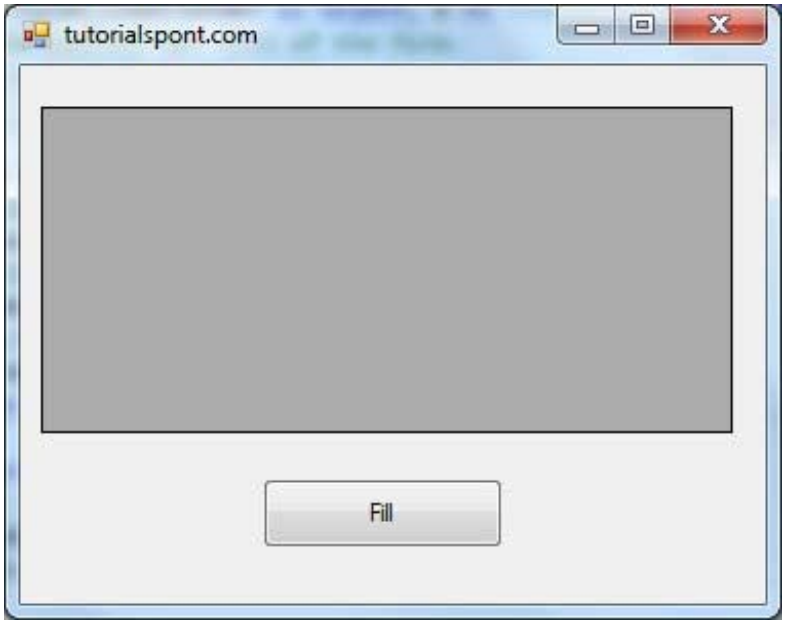
Take the following steps:

- Add a DataGridView control and a button in the form.
- Change the text of the button control to 'Fill'

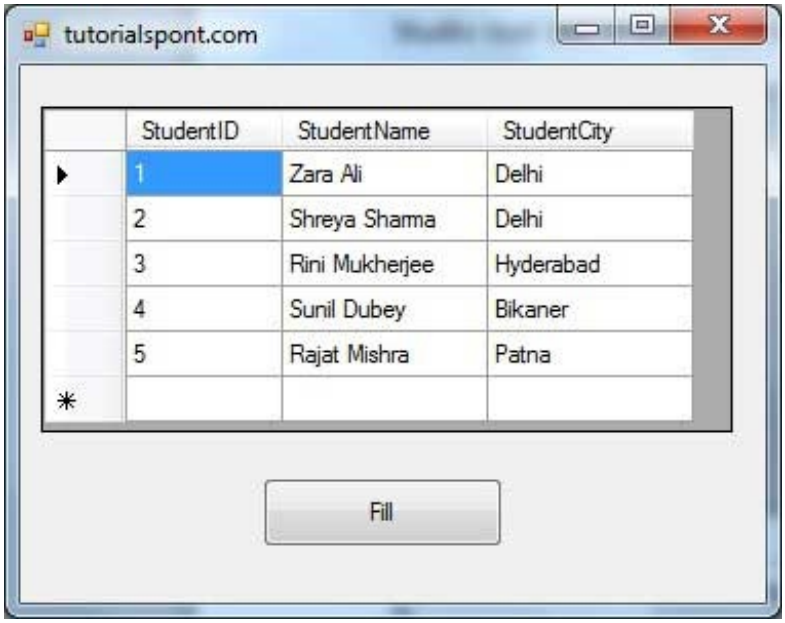
- Add the following code in the code editor

```
Public Class Form1
   Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
      ' Set the caption bar text of the form.
      Me.Text = "tutorialspont.com"
   End Sub
   Private Function CreateDataSet() As DataSet
      'creating a DataSet object for tables
      Dim dataset As DataSet = New DataSet()
      ' creating the student table
      Dim Students As DataTable = CreateStudentTable()
      dataset.Tables.Add(Students)
      Return dataset
   End Function
   Private Function CreateStudentTable() As DataTable
      Dim Students As DataTable
      Students = New DataTable("Student")
      ' adding columns
      AddNewColumn(Students, "System.Int32", "StudentID")
      AddNewColumn(Students, "System.String", "StudentName")
      AddNewColumn(Students, "System.String", "StudentCity")
      ' adding rows
      AddNewRow(Students, 1, "Zara Ali", "Kolkata")
      AddNewRow(Students, 2, "Shreya Sharma", "Delhi")
      AddNewRow(Students, 3, "Rini Mukherjee", "Hyderabad")
      AddNewRow(Students, 4, "Sunil Dubey", "Bikaner")
      AddNewRow(Students, 5, "Rajat Mishra", "Patna")
      Return Students
   End Function
   Private Sub AddNewColumn(ByRef table As DataTable, _
   ByVal columnType As String, ByVal columnName As String)
      Dim column As DataColumn = _
       table.Columns.Add(columnName, Type.GetType(columnType))
   End Sub

   'adding data into the table
   Private Sub AddNewRow(ByRef table As DataTable, ByRef id As Integer,_
   ByRef name As String, ByRef city As String)
      Dim newrow As DataRow = table.NewRow()
      newrow("StudentID") = id
      newrow("StudentName") = name
      newrow("StudentCity") = city
      table.Rows.Add(newrow)
   End Sub
   Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
      Dim ds As New DataSet
      ds = CreateDataSet()
      DataGridView1.DataSource = ds.Tables("Student")
   End Sub
End Class
```

When the above code is executed and run using **Start** button available at the Microsoft Visual Studio tool bar, it will show following window:



Clicking the Fill button, displays the table on the data grid view control:



VB.NET - EXCEL SHEET

VB.Net provides support for interoperability between the COM object model of Microsoft Excel 2010 and your application.

To avail this interoperability in your application, you need to import the namespace **Microsoft.Office.Interop.Excel** in your Windows Form Application.

Creating an Excel Application from VB.Net

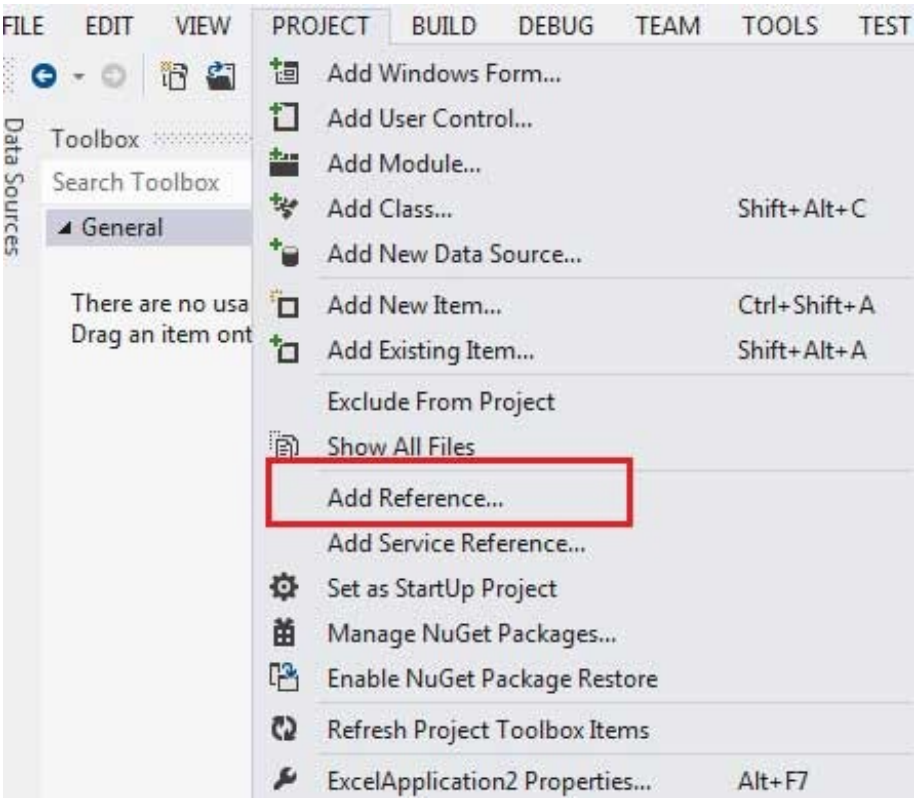
Let's start with creating a Window Forms Application by following the following steps in Microsoft Visual Studio: **File -> New Project -> Windows Forms Applications**

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

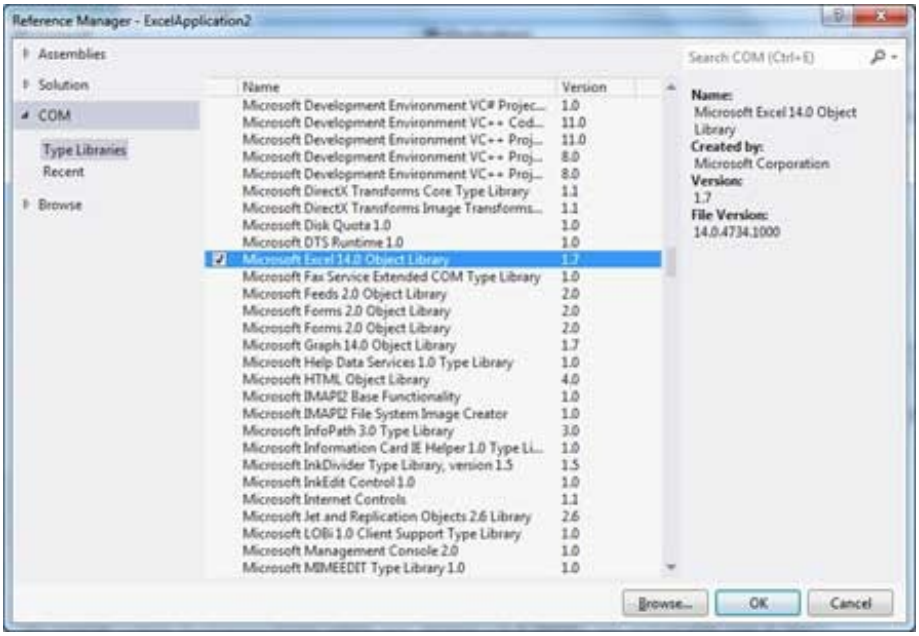
Insert a Button control Button1, in the form.

Add a reference to Microsoft Excel Object Library to your project. To do this:

1. Select Add Reference from the Project Menu.



2. On the COM tab, locate Microsoft Excel Object Library, and then click Select.



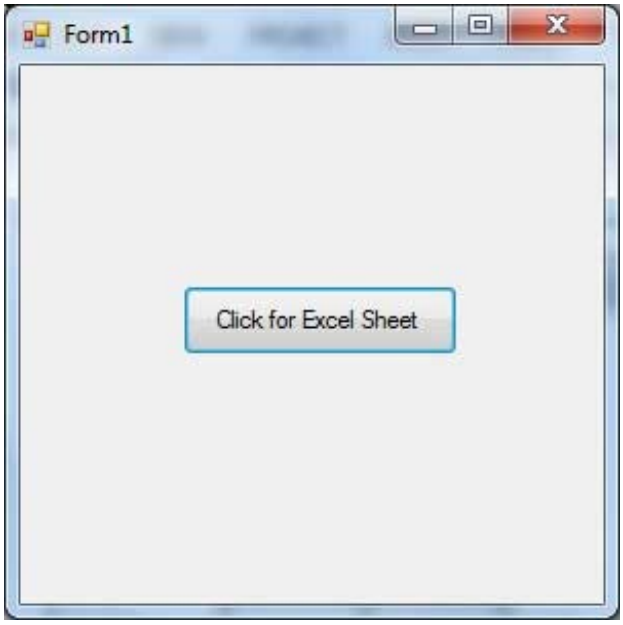
3. Click OK.

Double click the code window, and populate the Click event of Button1, as shown below.

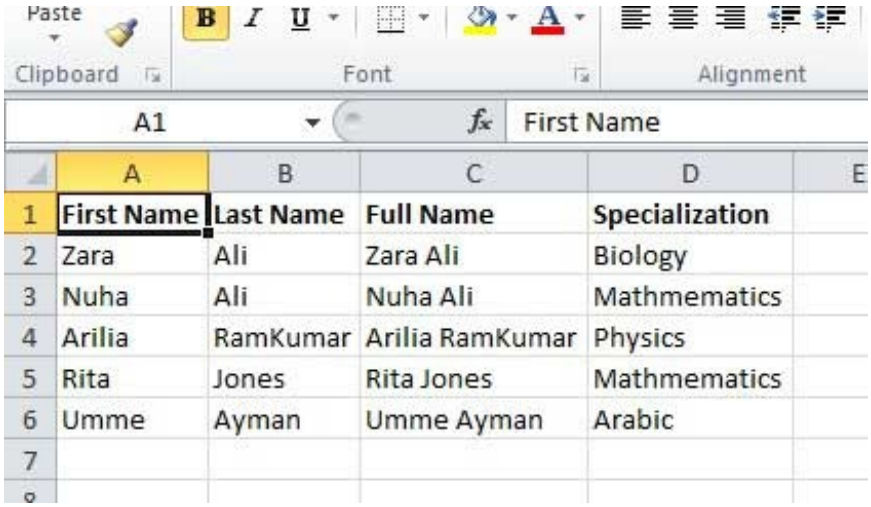
```
'  Add the following code snippet on top of Form1.vb
Imports Excel = Microsoft.Office.Interop.Excel
Public Class Form1
   Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
      Dim appXL As Excel.Application
      Dim wbXl As Excel.Workbook
```

```
      Dim shXL As Excel.Worksheet
      Dim raXL As Excel.Range
      ' Start Excel and get Application object.
      appXL = CreateObject("Excel.Application")
      appXL.Visible = True
      ' Add a new workbook.
      wbXl = appXL.Workbooks.Add
      shXL = wbXl.ActiveSheet
      ' Add table headers going cell by cell.
      shXL.Cells(1, 1).Value = "First Name"
      shXL.Cells(1, 2).Value = "Last Name"
      shXL.Cells(1, 3).Value = "Full Name"
      shXL.Cells(1, 4).Value = "Specialization"
      ' Format A1:D1 as bold, vertical alignment = center.
      With shXL.Range("A1", "D1")
          .Font.Bold = True
          .VerticalAlignment = Excel.XlVAlign.xlVAlignCenter
      End With
      ' Create an array to set multiple values at once.
      Dim students(5, 2) As String
      students(0, 0) = "Zara"
      students(0, 1) = "Ali"
      students(1, 0) = "Nuha"
      students(1, 1) = "Ali"
      students(2, 0) = "Arilia"
      students(2, 1) = "RamKumar"
      students(3, 0) = "Rita"
      students(3, 1) = "Jones"
      students(4, 0) = "Umme"
      students(4, 1) = "Ayman"
      ' Fill A2:B6 with an array of values (First and Last Names).
      shXL.Range("A2", "B6").Value = students
       ' Fill C2:C6 with a relative formula (=A2 & " " & B2).
      raXL = shXL.Range("C2", "C6")
      raXL.Formula = "=A2 & "" "" & B2"
       ' Fill D2:D6 values.
      With shXL
          .Cells(2, 4).Value = "Biology"
          .Cells(3, 4).Value = "Mathmematics"
          .Cells(4, 4).Value = "Physics"
          .Cells(5, 4).Value = "Mathmematics"
          .Cells(6, 4).Value = "Arabic"
      End With
      ' AutoFit columns A:D.
      raXL = shXL.Range("A1", "D1")
      raXL.EntireColumn.AutoFit()
       ' Make sure Excel is visible and give the user control
      ' of Excel's lifetime.
      appXL.Visible = True
      appXL.UserControl = True
       ' Release object references.
      raXL = Nothing
      shXL = Nothing
      wbXl = Nothing
      appXL.Quit()
      appXL = Nothing
      Exit Sub
Err_Handler:
      MsgBox(Err.Description, vbCritical, "Error: " & Err.Number)
   End Sub
End Class
```

When the above code is executed and run using **Start** button available at the Microsoft Visual Studio tool bar, it will show following window:



Clicking on the Button, would display the following excel sheet. You will be asked to save the workbook.



	A	B	C	D	E
1	First Name	Last Name	Full Name	Specialization	
2	Zara	Ali	Zara Ali	Biology	
3	Nuha	Ali	Nuha Ali	Mathmematics	
4	Arilia	RamKumar	Arilia RamKumar	Physics	
5	Rita	Jones	Rita Jones	Mathmematics	
6	Umme	Ayman	Umme Ayman	Arabic	
7					

VB.NET - SEND EMAIL

VB.Net allows sending E-mails from your application. The **System.Net.Mail** namespace contains classes used for sending emails to a Simple Mail Transfer Protocol (SMTP) server for delivery.

The following table lists some of these commonly used classes:

S.N	Class	Description
1	**Attachment**	Represents an attachment to an e-mail.
2	**AttachmentCollection**	Stores attachments to be sent as part of an e-mail message.
3	**MailAddress**	Represents the address of an electronic mail sender or recipient.
4	**MailAddressCollection**	Store e-mail addresses that are associated with an e-mail message.
5	**MailMessage**	Represents an e-mail message that can be sent using the SmtpClient class.
6	**SmtpClient**	Allows applications to send e-mail by using the Simple Mail Transfer Protocol

		(SMTP).
7	**SmtpException**	Represents the exception that is thrown when the SmtpClient is not able to complete a Send or SendAsync operation.

The SmtpClient Class

The SmtpClient class allows applications to send e-mail by using the Simple Mail Transfer Protocol (SMTP).

Following are some commonly used properties of the SmtpClient class:

S.N	Property	Description
1	**ClientCertificates**	Specify which certificates should be used to establish the Secure Sockets Layer (SSL) connection.
2	**Credentials**	Gets or sets the credentials used to authenticate the sender.
3	**EnableSsl**	Specify whether the SmtpClient uses Secure Sockets Layer (SSL) to encrypt the connection.
4	**Host**	Gets or sets the name or IP address of the host used for SMTP transactions..
5	**Port**	Gets or sets the port used for SMTP transactions.
6	**Timeout**	Gets or sets a value that specifies the amount of time after which a synchronous Send call times out.
7	**UseDefaultCredentials**	Gets or sets a Boolean value that controls whether the DefaultCredentials are sent with requests.

Following are some commonly used methods of the SmtpClient class:

S.N	Method & Description
1	**Dispose** Sends a QUIT message to the SMTP server, gracefully ends the TCP connection, and releases all resources used by the current instance of the SmtpClient class.
2	**Dispose(Boolean)** Sends a QUIT message to the SMTP server, gracefully ends the TCP connection, releases all resources used by the current instance of the SmtpClient class, and optionally disposes of the managed resources.
3	**OnSendCompleted** Raises the SendCompleted event.
4	**Send(MailMessage)** Sends the specified message to an SMTP server for delivery.
5	**Send(String, String, String, String)** Sends the specified e-mail message to an SMTP server for delivery. The message sender, recipients, subject, and message body are specified using String objects.

6	**SendAsync(MailMessage, Object)** Sends the specified e-mail message to an SMTP server for delivery. This method does not block the calling thread and allows the caller to pass an object to the method that is invoked when the operation completes.
7	**SendAsync(String, String, String, String, Object)** Sends an e-mail message to an SMTP server for delivery. The message sender, recipients, subject, and message body are specified using String objects. This method does not block the calling thread and allows the caller to pass an object to the method that is invoked when the operation completes.
8	**SendAsyncCancel** Cancels an asynchronous operation to send an e-mail message.
9	**SendMailAsync(MailMessage)** Sends the specified message to an SMTP server for delivery as an asynchronous operation.
10	**SendMailAsync(String, String, String, String)** Sends the specified message to an SMTP server for delivery as an asynchronous operation. . The message sender, recipients, subject, and message body are specified using String objects.
11	**ToString** Returns a string that represents the current object.

The following example demonstrates how to send mail using the SmtpClient class. Following points are to be noted in this respect:

- You must specify the SMTP host server that you use to send e-mail. The **Host** and **Port** properties will be different for different host server. We will be using gmail server.
- You need to give the **Credentials** for authentication, if required by the SMTP server.
- You should also provide the email address of the sender and the e-mail address or addresses of the recipients using the **MailMessage.From** and **MailMessage.To** properties respectively.
- You should also specify the message content using the **MailMessage.Body** property.

Example

In this example, let us create a simple application that would send an email. Take the following steps:

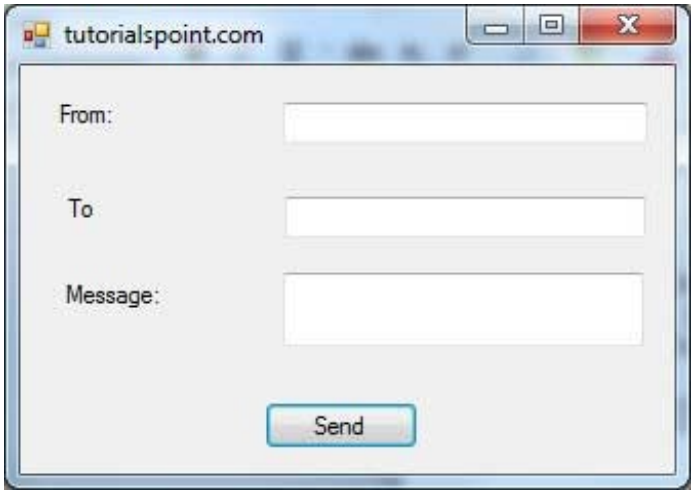
1. Add three labels, three text boxes and a button control in the form.
2. Change the text properties of the labels to - 'From', 'To:' and 'Message:' respectively.
3. Change the name properties of the texts to txtFrom, txtTo and txtMessage respectively.
4. Change the text property of the button control to 'Send'
5. Add the following code in the code editor.

```
Imports System.Net.Mail
Public Class Form1
   Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
      ' Set the caption bar text of the form.
      Me.Text = "tutorialspoint.com"
   End Sub
```

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
   Try
       Dim Smtp_Server As New SmtpClient
       Dim e_mail As New MailMessage()
       Smtp_Server.UseDefaultCredentials = False
       Smtp_Server.Credentials = New Net.NetworkCredential("username@gmail.com", "password")
       Smtp_Server.Port = 587
       Smtp_Server.EnableSsl = True
       Smtp_Server.Host = "smtp.gmail.com"

       e_mail = New MailMessage()
       e_mail.From = New MailAddress(txtFrom.Text)
       e_mail.To.Add(txtTo.Text)
       e_mail.Subject = "Email Sending"
       e_mail.IsBodyHtml = False
       e_mail.Body = txtMessage.Text
       Smtp_Server.Send(e_mail)
       MsgBox("Mail Sent")

   Catch error_t As Exception
       MsgBox(error_t.ToString)
   End Try

End Sub
```

You must provide your gmail address and real password for credentials.

When the above code is executed and run using **Start** button available at the Microsoft Visual Studio tool bar, it will show following window which you will use to send your emails, try it yourself.



VB.NET - XML PROCESSING

The Extensible Markup Language (XML) is a markup language much like HTML or SGML. This is recommended by the World Wide Web Consortium and available as an open standard.

The **System.Xml** namespace in the .Net Framework contains classes for processing XML documents. Following are some of the commonly used classes in the System.Xml namespace.

S.N	Class	Description
1	**XmlAttribute**	Represents an attribute. Valid and default values for the attribute are defined in a document type definition (DTD) or schema.
2	**XmlCDataSection**	Represents a CDATA section.
3	**XmlCharacterData**	Provides text manipulation methods that are used by several classes.

4	**XmlComment**	Represents the content of an XML comment.
5	**XmlConvert**	Encodes and decodes XML names and provides methods for converting between common language runtime types and XML Schema definition language (XSD) types. When converting data types the values returned are locale independent.
6	**XmlDeclaration**	Represents the XML declaration node <?xml version='1.0'…?>.
7	**XmlDictionary**	Implements a dictionary used to optimize Windows Communication Foundation (WCF)'s XML reader/writer implementations.
8	**XmlDictionaryReader**	An abstract class that the Windows Communication Foundation (WCF) derives from XmlReader to do serialization and deserialization.
9	**XmlDictionaryWriter**	Represents an abstract class that Windows Communication Foundation (WCF) derives from XmlWriter to do serialization and deserialization.
10	**XmlDocument**	Represents an XML document.
11	**XmlDocumentFragment**	Represents a lightweight object that is useful for tree insert operations.
12	**XmlDocumentType**	Represents the document type declaration.
13	**XmlElement**	Represents an element.
14	**XmlEntity**	Represents an entity declaration, such as <!ENTITY… >.
15	**XmlEntityReference**	Represents an entity reference node.
16	**XmlException**	Returns detailed information about the last exception.
17	**XmlImplementation**	Defines the context for a set of XmlDocument objects.
18	**XmlLinkedNode**	Gets the node immediately preceding or following this node.
19	**XmlNode**	Represents a single node in the XML document.
20	**XmlNodeList**	Represents an ordered collection of nodes.
21	**XmlNodeReader**	Represents a reader that provides fast, non-cached forward only access to XML data in an XmlNode.
22	**XmlNotation**	Represents a notation declaration, such as <!NOTATION... >.
23	**XmlParserContext**	Provides all the context information required by the XmlReader to parse an XML fragment.
24	**XmlProcessingInstruction**	Represents a processing instruction, which XML defines to keep processor-specific information in the text of the document.
25	**XmlQualifiedName**	Represents an XML qualified name.
26	**XmlReader**	Represents a reader that provides fast, noncached, forward-only access to XML data.

27	**XmlReaderSettings**	Specifies a set of features to support on the XmlReader object created by the Create method.
28	**XmlResolver**	Resolves external XML resources named by a Uniform Resource Identifier (URI).
29	**XmlSecureResolver**	Helps to secure another implementation of XmlResolver by wrapping the XmlResolver object and restricting the resources that the underlying XmlResolver has access to.
30	**XmlSignificantWhitespace**	Represents white space between markup in a mixed content node or white space within an xml:space= 'preserve' scope. This is also referred to as significant white space.
31	**XmlText**	Represents the text content of an element or attribute.
32	**XmlTextReader**	Represents a reader that provides fast, non-cached, forward-only access to XML data.
33	**XmlTextWriter**	Represents a writer that provides a fast, non-cached, forward-only way of generating streams or files containing XML data that conforms to the W3C Extensible Markup Language (XML) 1.0 and the Namespaces in XML recommendations.
34	**XmlUrlResolver**	Resolves external XML resources named by a Uniform Resource Identifier (URI).
35	**XmlWhitespace**	Represents white space in element content.
36	**XmlWriter**	Represents a writer that provides a fast, non-cached, forward-only means of generating streams or files containing XML data.
37	**XmlWriterSettings**	Specifies a set of features to support on the XmlWriter object created by the XmlWriter.Create method.

XML Parser APIs

The two most basic and broadly used APIs to XML data are the SAX and DOM interfaces.

- **Simple API for XML (SAX)** : Here you register callbacks for events of interest and then let the parser proceed through the document. This is useful when your documents are large or you have memory limitations, it parses the file as it reads it from disk, and the entire file is never stored in memory.
- **Document Object Model (DOM) API** : This is World Wide Web Consortium recommendation wherein the entire file is read into memory and stored in a hierarchical (tree-based) form to represent all the features of an XML document.

SAX obviously can't process information as fast as DOM can when working with large files. On the other hand, using DOM exclusively can really kill your resources, especially if used on a lot of small files.

SAX is read-only, while DOM allows changes to the XML file. Since these two different APIs literally complement each other there is no reason why you can't use them both for large projects.

For all our XML code examples, let's use a simple XML file movies.xml as an input:

```
<?xml version="1.0"?>
```

```
<collection shelf="New Arrivals">
<movie title="Enemy Behind">
   <type>War, Thriller</type>
   <format>DVD</format>
   <year>2003</year>
   <rating>PG</rating>
   <stars>10</stars>
   <description>Talk about a US-Japan war</description>
</movie>
<movie title="Transformers">
   <type>Anime, Science Fiction</type>
   <format>DVD</format>
   <year>1989</year>
   <rating>R</rating>
   <stars>8</stars>
   <description>A schientific fiction</description>
</movie>
   <movie title="Trigun">
   <type>Anime, Action</type>
   <format>DVD</format>
   <episodes>4</episodes>
   <rating>PG</rating>
   <stars>10</stars>
   <description>Vash the Stampede!</description>
</movie>
<movie title="Ishtar">
   <type>Comedy</type>
   <format>VHS</format>
   <rating>PG</rating>
   <stars>2</stars>
   <description>Viewable boredom</description>
</movie>
</collection>
```

Parsing XML with SAX API

In SAX model, you use the **XmlReader** and **XmlWriter** classes to work with the XML data.

The **XmlReader** class is used to read XML data in a fast, forward-only and non-cached manner. It reads an XML document or a stream.

Example 1

This example demonstrates reading XML data from the file movies.xml.

Take the following steps:

1. Add the movies.xml file in the bin\Debug folder of your application.
2. Import the System.Xml namespace in Form1.vb file.
3. Add a label in the form and change its text to 'Movies Galore'.
4. Add three list boxes and three buttons to show the title, type and description of a movie from the xml file.
5. Add the following code using the code editor window.

```
Imports System.Xml
Public Class Form1

   Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
      ' Set the caption bar text of the form.
      Me.Text = "tutorialspoint.com"
   End Sub
   Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

```
      ListBox1().Items.Clear()
      Dim xr As XmlReader = XmlReader.Create("movies.xml")
      Do While xr.Read()
          If xr.NodeType = XmlNodeType.Element AndAlso xr.Name = "movie" Then
              ListBox1.Items.Add(xr.GetAttribute(0))
          End If
      Loop
   End Sub
   Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
      ListBox2().Items.Clear()
      Dim xr As XmlReader = XmlReader.Create("movies.xml")
      Do While xr.Read()
          If xr.NodeType = XmlNodeType.Element AndAlso xr.Name = "type" Then
              ListBox2.Items.Add(xr.ReadElementString)
          Else
              xr.Read()
          End If
      Loop
   End Sub
   Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
      ListBox3().Items.Clear()
      Dim xr As XmlReader = XmlReader.Create("movies.xml")
      Do While xr.Read()
          If xr.NodeType = XmlNodeType.Element AndAlso xr.Name = "description" Then
              ListBox3.Items.Add(xr.ReadElementString)
          Else
              xr.Read()
          End If
      Loop
   End Sub
End Class
```

Execute and run the above code using **Start** button available at the Microsoft Visual Studio tool bar. Clicking on the buttons would display, title, type and description of the movies from the file.

The **XmlWriter** class is used to write XML data into a stream, a file or a TextWriter object. It also works in a forward-only, non-cached manner.

Example 2

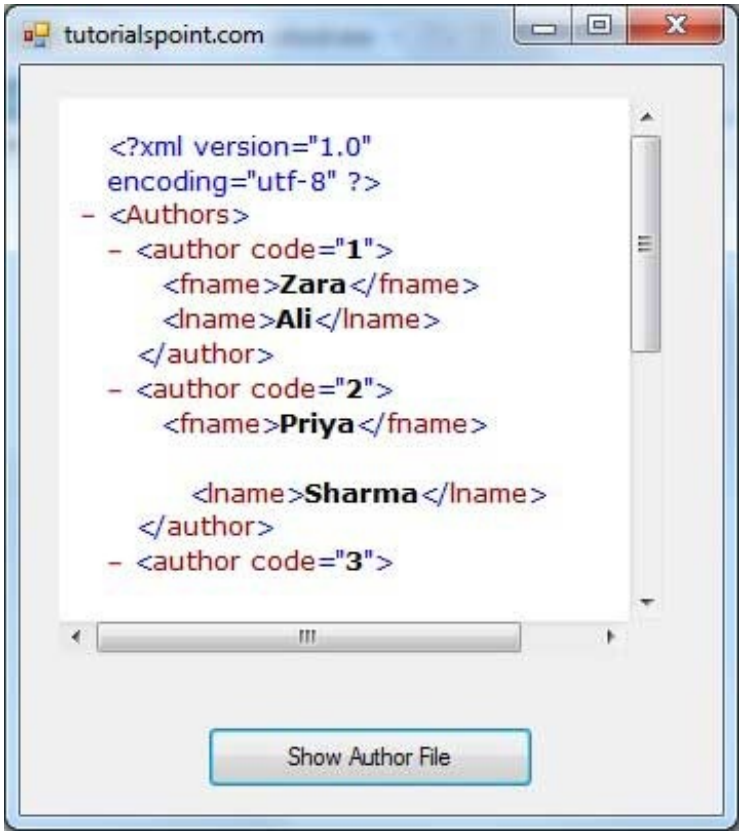
Let us create an XML file by adding some data at runtime. Take the following steps:

1. Add a WebBrowser control and a button control in the form.
2. Change the Text property of the button to Show Authors File.

3. Add the following code in the code editor.

```
Imports System.Xml
Public Class Form1
   Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
      ' Set the caption bar text of the form.
      Me.Text = "tutorialspoint.com"
   End Sub
   Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
      Dim xws As XmlWriterSettings = New XmlWriterSettings()
      xws.Indent = True
      xws.NewLineOnAttributes = True
      Dim xw As XmlWriter = XmlWriter.Create("authors.xml", xws)
      xw.WriteStartDocument()
      xw.WriteStartElement("Authors")
      xw.WriteStartElement("author")
      xw.WriteAttributeString("code", "1")
      xw.WriteElementString("fname", "Zara")
      xw.WriteElementString("lname", "Ali")
      xw.WriteEndElement()
      xw.WriteStartElement("author")
      xw.WriteAttributeString("code", "2")
      xw.WriteElementString("fname", "Priya")
      xw.WriteElementString("lname", "Sharma")
      xw.WriteEndElement()
      xw.WriteStartElement("author")
      xw.WriteAttributeString("code", "3")
      xw.WriteElementString("fname", "Anshuman")
      xw.WriteElementString("lname", "Mohan")
      xw.WriteEndElement()
      xw.WriteStartElement("author")
      xw.WriteAttributeString("code", "4")
      xw.WriteElementString("fname", "Bibhuti")
      xw.WriteElementString("lname", "Banerjee")
      xw.WriteEndElement()
      xw.WriteStartElement("author")
      xw.WriteAttributeString("code", "5")
      xw.WriteElementString("fname", "Riyan")
      xw.WriteElementString("lname", "Sengupta")
      xw.WriteEndElement()
      xw.WriteEndElement()
      xw.WriteEndDocument()
      xw.Flush()
      xw.Close()
      WebBrowser1.Url = New Uri(AppDomain.CurrentDomain.BaseDirectory + "authors.xml")
   End Sub
End Class
```

Execute and run the above code using **Start** button available at the Microsoft Visual Studio tool bar. Clicking on the Show Author File would display the newly created authors.xml file on the web browser.



Parsing XML with DOM API

According to the Document Object Model (DOM), an XML document consists of nodes and attributes of the nodes. The **XmlDocument** class is used to implement the XML DOM parser of the .Net Framework. It also allows you to modify an existing XML document by inserting, deleting or updating data in the document.

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

S.N	Method Name & Description
1	**AppendChild** Adds the specified node to the end of the list of child nodes, of this node.
2	**CreateAttribute(String)** Creates an XmlAttribute with the specified Name.
3	**CreateComment** Creates an XmlComment containing the specified data.
4	**CreateDefaultAttribute** Creates a default attribute with the specified prefix, local name and namespace URI.
5	**CreateElement(String)** Creates an element with the specified name.
6	**CreateNode(String, String, String)** Creates an XmlNode with the specified node type, Name, and NamespaceURI.
7	**CreateNode(XmlNodeType, String, String)** Creates an XmlNode with the specified XmlNodeType, Name, and NamespaceURI.

8	**CreateNode(XmlNodeType, String, String, String)** Creates a XmlNode with the specified XmlNodeType, Prefix, Name, and NamespaceURI.
9	**CreateProcessingInstruction** Creates an XmlProcessingInstruction with the specified name and data.
10	**CreateSignificantWhitespace** Creates an XmlSignificantWhitespace node.
11	**CreateTextNode** Creates an XmlText with the specified text.
12	**CreateWhitespace** Creates an XmlWhitespace node.
13	**CreateXmlDeclaration** Creates an XmlDeclaration node with the specified values.
14	**GetElementById** Gets the XmlElement with the specified ID.
15	**GetElementsByTagName(String)** Returns an XmlNodeList containing a list of all descendant elements that match the specified Name.
16	**GetElementsByTagName(String, String)** Returns an XmlNodeList containing a list of all descendant elements that match the specified LocalName and NamespaceURI.
17	**InsertAfter** Inserts the specified node immediately after the specified reference node.
18	**InsertBefore** Inserts the specified node immediately before the specified reference node.
19	**Load(Stream)** Loads the XML document from the specified stream.
20	**Load(String)** Loads the XML document from the specified URL.
21	**Load(TextReader)** Loads the XML document from the specified TextReader.
22	**Load(XmlReader)** Loads the XML document from the specified XmlReader.
23	**LoadXml** Loads the XML document from the specified string.
24	**PrependChild** Adds the specified node to the beginning of the list of child nodes for this node.
25	**ReadNode** Creates an XmlNode object based on the information in the XmlReader. The reader must be positioned on a node or attribute.

26	**RemoveAll** Removes all the child nodes and/or attributes of the current node.
27	**RemoveChild** Removes specified child node.
28	**ReplaceChild** Replaces the child node oldChild with newChild node.
29	**Save(Stream)** Saves the XML document to the specified stream.
30	**Save(String)** Saves the XML document to the specified file.
31	**Save(TextWriter)** Saves the XML document to the specified TextWriter.
32	**Save(XmlWriter)** Saves the XML document to the specified XmlWriter.

Example 3

In this example, let us insert some new nodes in the xml document authors.xml and then show all the authors' first names in a list box.

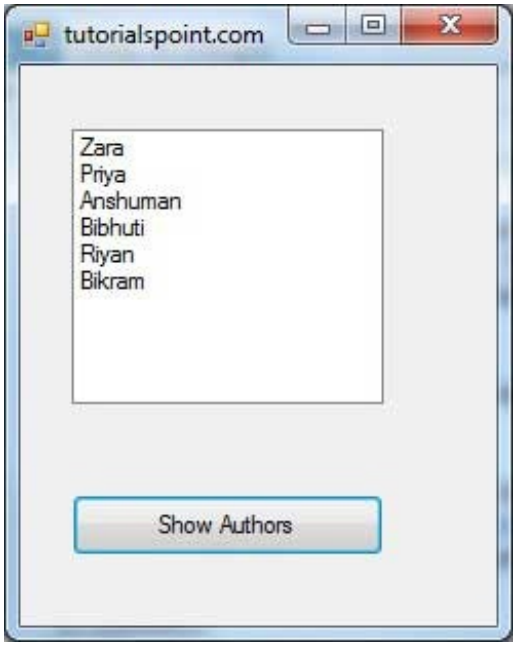
Take the following steps:

- Add the authors.xml file in the bin/Debug folder of your application(it should be there if you have tried the last example)
- Import the System.Xml namespace
- Add a list box and a button control in the form and set the text property of the button control to Show Authors.
- Add the following code using the code editor.

```
Imports System.Xml
Public Class Form1
   Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
      ' Set the caption bar text of the form.
      Me.Text = "tutorialspoint.com"
   End Sub
   Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
      ListBox1.Items.Clear()
      Dim xd As XmlDocument = New XmlDocument()
      xd.Load("authors.xml")
      Dim newAuthor As XmlElement = xd.CreateElement("author")
      newAuthor.SetAttribute("code", "6")
      Dim fn As XmlElement = xd.CreateElement("fname")
      fn.InnerText = "Bikram"
      newAuthor.AppendChild(fn)
      Dim ln As XmlElement = xd.CreateElement("lname")
      ln.InnerText = "Seth"
      newAuthor.AppendChild(ln)
      xd.DocumentElement.AppendChild(newAuthor)
      Dim tr As XmlTextWriter = New XmlTextWriter("movies.xml", Nothing)
      tr.Formatting = Formatting.Indented
      xd.WriteContentTo(tr)
      tr.Close()
```

```
      Dim nl As XmlNodeList = xd.GetElementsByTagName("fname")
      For Each node As XmlNode In nl
          ListBox1.Items.Add(node.InnerText)
      Next node
   End Sub
End Class
```

Execute and run the above code using **Start** button available at the Microsoft Visual Studio tool bar. Clicking on the Show Author button would display the first names of all the authors including the one we have added at runtime.



VB.NET - WEB PROGRAMMING

A dynamic web application consists of either or both of the following two types of programs:

- **Server-side scripting** - these are programs executed on a web server, written using server-side scripting languages like ASP (Active Server Pages) or JSP (Java Server Pages).
- **Client-side scripting** - these are programs executed on the browser, written using scripting languages like JavaScript, VBScript etc.

ASP.Net is the .Net version of ASP, introduced by Microsoft, for creating dynamic web pages by using server-side scripts. ASP.Net applications are compiled codes, written using the extensible and reusable components or objects present in .Net framework. These codes can use the entire hierarchy of classes in .Net framework.

The ASP.Net application codes could be written in either of the following languages:

- Visual Basic .Net
- C#
- Jscript
- J#

In this chapter, we will give a very brief introduction to writing ASP.Net applications using VB.Net. For detailed discussion, please consult the ASP.Net Tutorial.

ASP.Net Built-in Objects

ASP.Net has some built-in objects that run on a web server. These objects have methods, properties and collections that are used in application development.

The following table lists the ASP.Net built-in objects with a brief description:

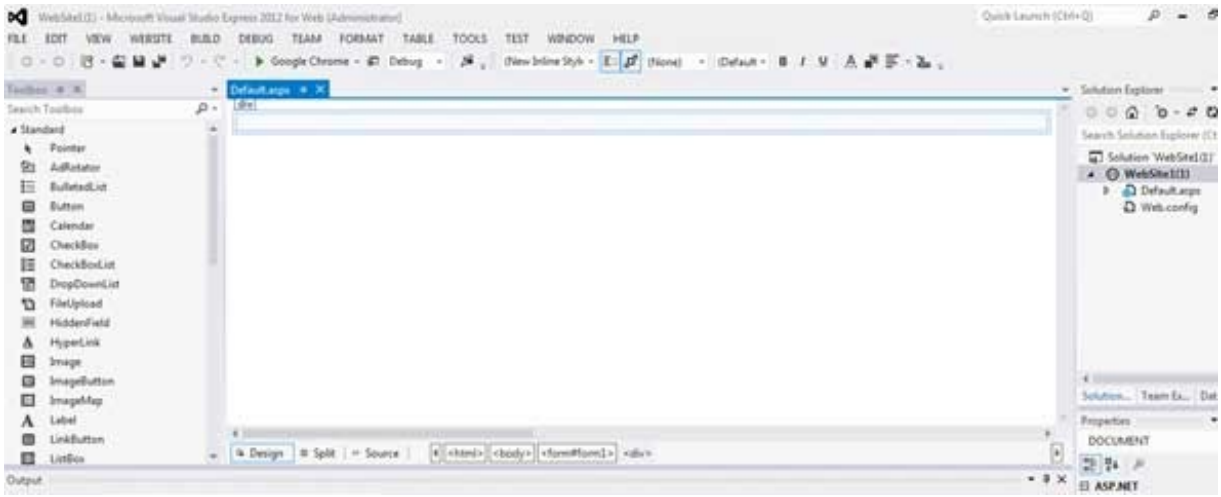
Object	Description
Application	Describes the methods, properties, and collections of the object that stores information related to the entire Web application, including variables and objects that exist for the lifetime of the application. You use this object to store and retrieve information to be shared among all user of an application. For example, you can use an Application object to create an e-commerce page.
Request	Describes the methods, properties, and collections of the object that stores information related to the HTTP request. This includes forms, cookies, server variables, and certificate data. You use this object to access the information sent in a request from a browser to the server. For example, you can use a Request object to access information entered by a user in an HTML form.
Response	Describes the methods, properties, and collections of the object that stores information related to the server's response. This includes displaying content, manipulating headers, setting locales, and redirecting requests. You use this object to send information to the browser. For example, you use a Response object to send output from your scripts to a browser.
Server	Describes the methods and properties of the object that provides methods for various server tasks. With these methods you can execute code, get error conditions, encode text strings, create objects for use by the Web page, and map physical paths. You use this object to access various utility functions on the server. For example, you may use the Server object to set a time out for a script.
Session	Describes the methods, properties, and collections of the object that stores information related to the user's session, including variables and objects that exist for the lifetime of the session. You use this object to store and retrieve information about particular user sessions. For example, you can use Session object to keep information about the user and his preference and keep track of pending operations.

ASP.Net Programming Model

ASP.Net provides two types of programming models:

- **Web Forms** - this enables you to create the user interface and the application logic that would be applied to various components of the user interface.
- **WCF Services** - this enables you to remote access some server-side functionalities.

For this chapter, you need to use Visual Studio Web Developer, which is free. The IDE is almost same as you have already used for creating the Windows Applications.



Web Forms

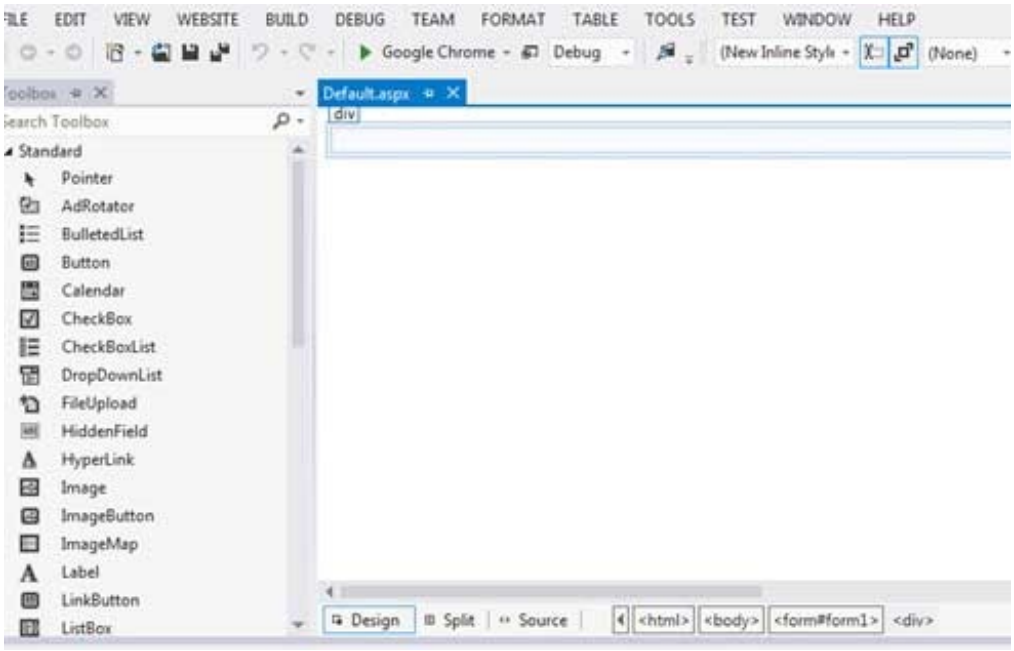
Web forms consists of:

- User interface
- Application logic

User interface consists of static HTML, or XML elements and ASP.Net server controls. When you create a web application, HTML or XML elements and server controls are stored in a file with **.aspx** extension. This file is also called the page file.

The application logic consists of code applied to the user interface elements in the page. You write this code in any of .Net language like, VB.Net, or C#.

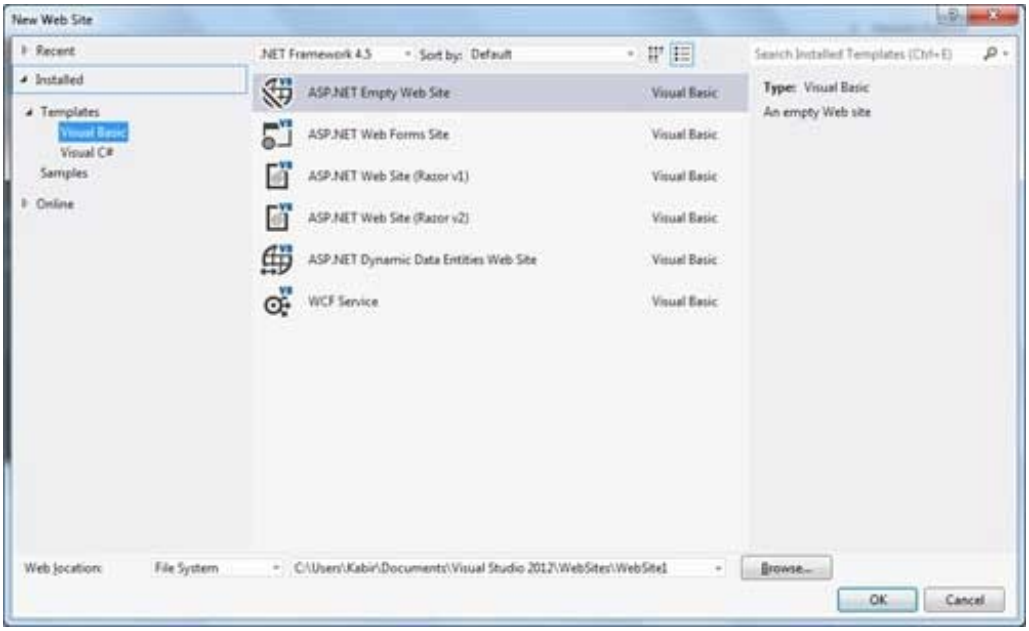
The following figure shows a Web Form in Design view:



Example

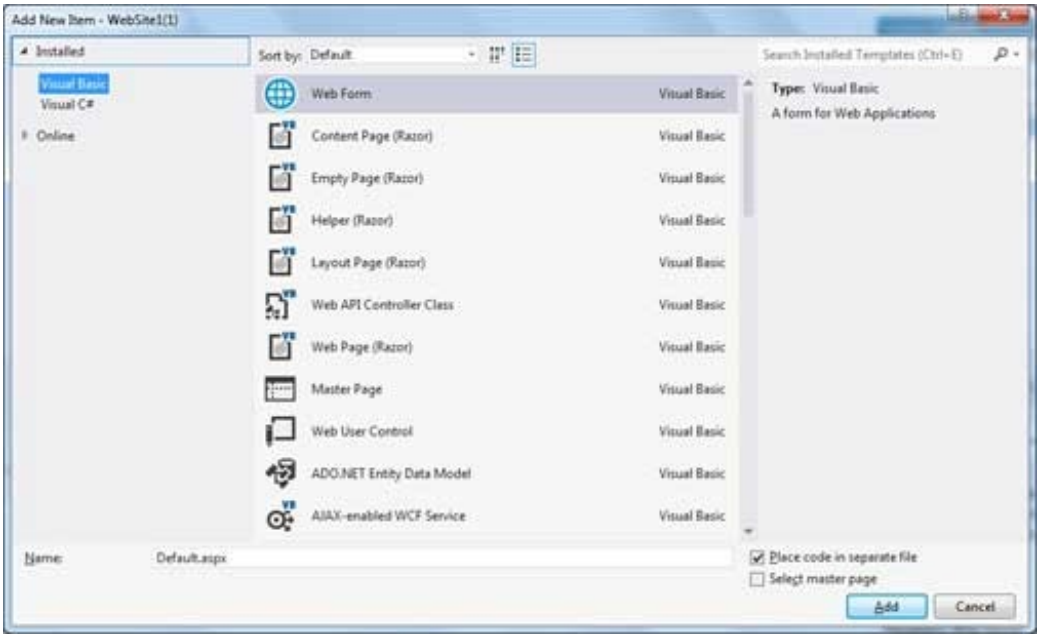
Let us create a new web site with a web form, which will show the current date and time, when a user clicks a button. Take the following steps:

1. Select File -> New -> Web Site. The New Web Site Dialog Box appears.



2. Select the ASP.Net Empty Web Site templates. Type a name for the web site and select a location for saving the files.

3. You need to add a Default page to the site. Right click the web site name in the Solution Explorer and select Add New Item option from the context menu. The Add New Item dialog box is displayed:

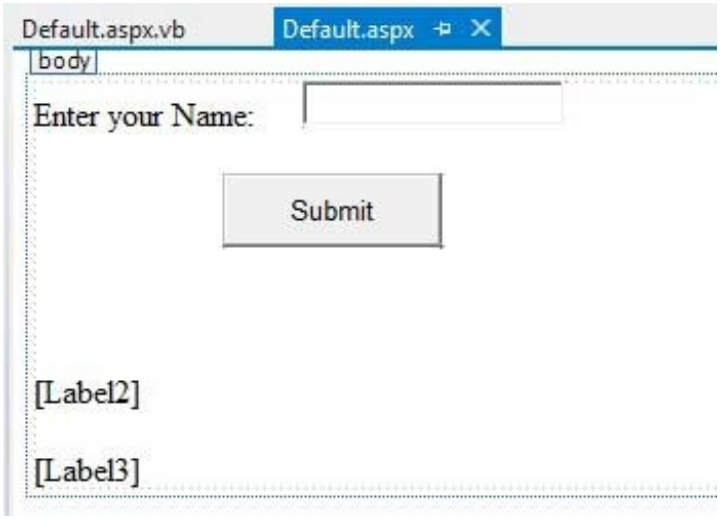


4. Select Web Form option and provide a name for the default page. We have kept it as Default.aspx. Click the Add button.

5. The Default page is shown in Source view

6. Set the title for the Default web page by adding a value to the

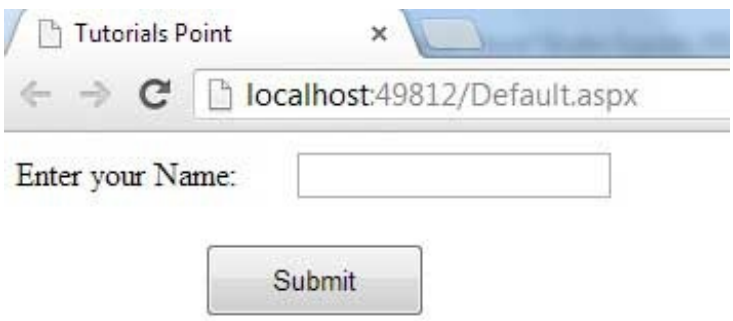
7. To add controls on the web page, go to the design view. Add three labels, a text box and a button on the form.



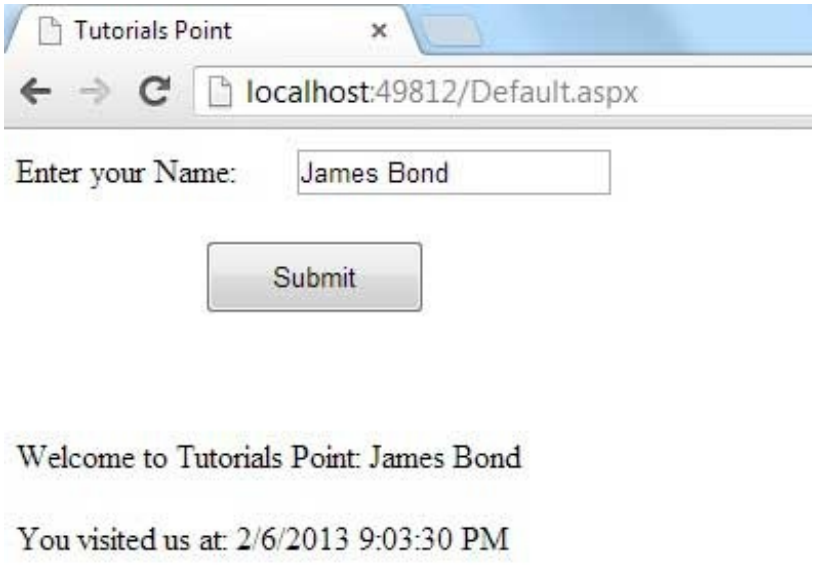
8. Double-click the button and add the following code to the Click event of the button:

```
Protected Sub Button1_Click(sender As Object, e As EventArgs) _
Handles Button1.Click
   Label2.Visible = True
   Label2.Text = "Welcome to Tutorials Point: " + TextBox1.Text
   Label3.Text = "You visited us at: " + DateTime.Now.ToString()
End Sub
```

When the above code is executed and run using Start button available at the Microsoft Visual Studio tool bar, the following page opens in the browser:



Enter your name and click on the Submit button:



Web Services

A web service is a web application which is basically a class consisting of methods that could be used by other applications. It also follows a code-behind architecture like the ASP.Net web pages, although it does not have an user interface.

The previous versions of .Net Framework used this concept of ASP.Net Web Service, which had .asmx file extension. However, from .Net Framework 4.0 onwards, the Windows Communication Foundation (WCF) technology has evolved as the new successor of Web Services, .Net Remoting and some other related technologies. It has rather clubbed all these technologies together. In the next section, we will provide a brief introduction to Windows Communication Foundation(WCF).

If you are using previous versions of .Net Framework, you can still create traditional web services. Please consult ASP.Net - Web Services tutorial for detailed description.

Windows Communication Foundation

Windows Communication Foundation or WCF provides an API for creating distributed service oriented applications, known as WCF Services.

Like Web services, WCF services also enable communication between applications. However, unlike web services, the communication here is not limited to HTTP only. WCF can be configured to be used over HTTP, TCP, IPC, and Message Queues. Another strong point in favour of WCF is, it provides support for duplex communication, whereas with web services we could achieve simplex communication only.

From beginners' point of view, writing a WCF service is not altogether so different from writing a Web Service. To keep the things simple, we will see how to:

- Create a WCF Service
- Create a Service Contract and define the operations
- Implement the contract
- Test the Service
- Utilize the Service

Example

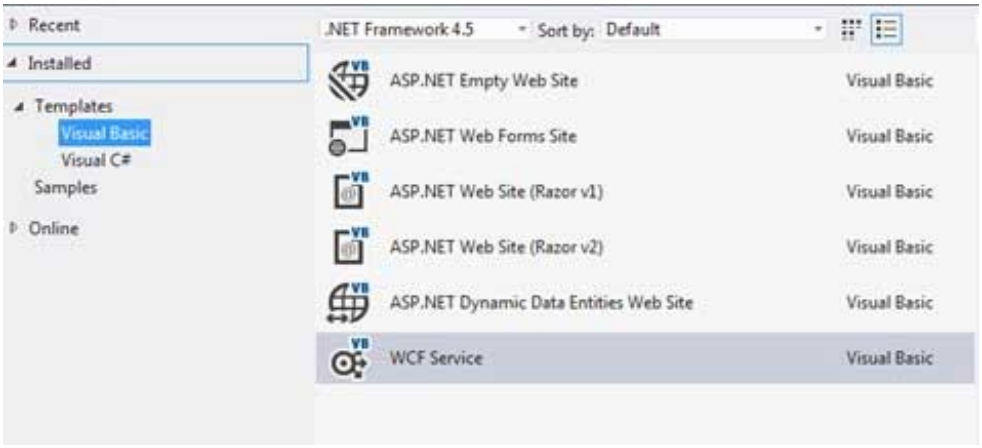
To understand the concept let us create a simplistic service that will provide stock price information. The clients can query about the name and price of a stock based on the stock symbol. To keep this example simple, the values are hardcoded in a two-dimensional array. This service will have two methods:

- GetPrice Method - it will return the price of a stock, based on the symbol provided.
- GetName Method - it will return the name of the stock, based on the symbol provided.

Creating a WCF Service

Take the following steps:

1. Open VS Express for Web 2012
2. Select New Web Site to open the New Web Site dialog box.
3. Select WCF Service template from list of templates:



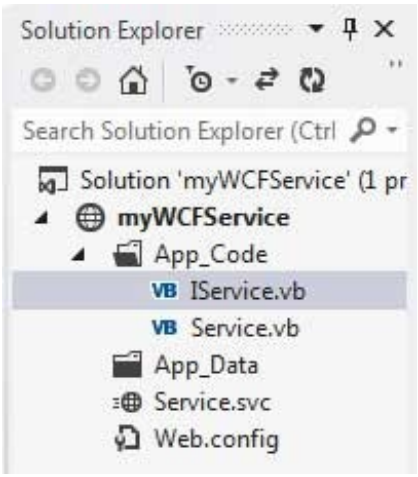
4. Select File System from the Web location drop-down list.
5. Provide a name and location for the WCF Service and click OK.

A new WCF Service is created.

Creating a Service Contract and Defining the Operations

A service contract defines the operation that a service performs. In the WCF Service application, you will find two files automatically created in the App_Code folder in the Solution Explorer

- IService.vb - this will have the service contract; in simpler words, it will have the interface for the service, with the definitions of methods the service will provide, which you will implement in your service.
- Service.vb - this will implement the service contract.



Replace the code of the IService.vb file with the given code:

```
Public Interface IService
    <OperationContract()>
    Function GetPrice(ByVal symbol As String) As Double

    <OperationContract()>
    Function GetName(ByVal symbol As String) As String
End Interface
```

Implementing the Contract

In the Service.vb file, you will find a class named **Service** which will implement the Service Contract defined in the **IService** interface.

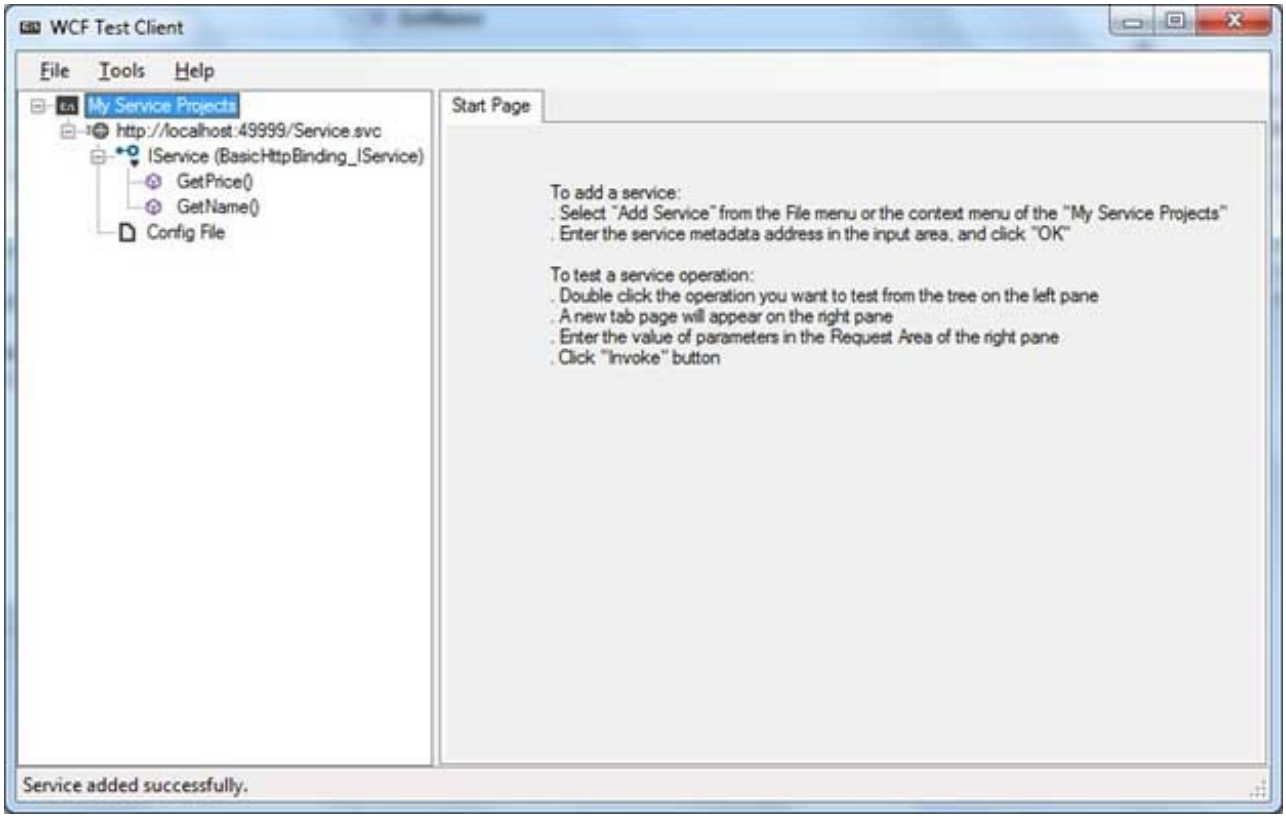
Replace the code of IService.vb with the following code:

```
' NOTE: You can use the "Rename" command on the context menu to change the class name
"Service" in code, svc and config file together.
Public Class Service
   Implements IService
   Public Sub New()
   End Sub
   Dim stocks As String(,) =
   {
   {"RELIND", "Reliance Industries", "1060.15"},
   {"ICICI", "ICICI Bank", "911.55"},
   {"JSW", "JSW Steel", "1201.25"},
   {"WIPRO", "Wipro Limited", "1194.65"},
   {"SATYAM", "Satyam Computers", "91.10"}
   }

   Public Function GetPrice(ByVal symbol As String) As Double _
   Implements IService.GetPrice

      Dim i As Integer
      'it takes the symbol as parameter and returns price
      For i = 0 To i = stocks.GetLength(0) - 1

          If (String.Compare(symbol, stocks(i, 0)) = 0) Then
              Return Convert.ToDouble(stocks(i, 2))
          End If
      Next i
      Return 0
   End Function

   Public Function GetName(ByVal symbol As String) As String _
   Implements IService.GetName

      ' It takes the symbol as parameter and
```

```
      ' returns name of the stock
      Dim i As Integer
      For i = 0 To i = stocks.GetLength(0) - 1

          If (String.Compare(symbol, stocks(i, 0)) = 0) Then
              Return stocks(i, 1)
          End If
      Next i
      Return "Stock Not Found"
   End Function
End Class
```

Testing the Service

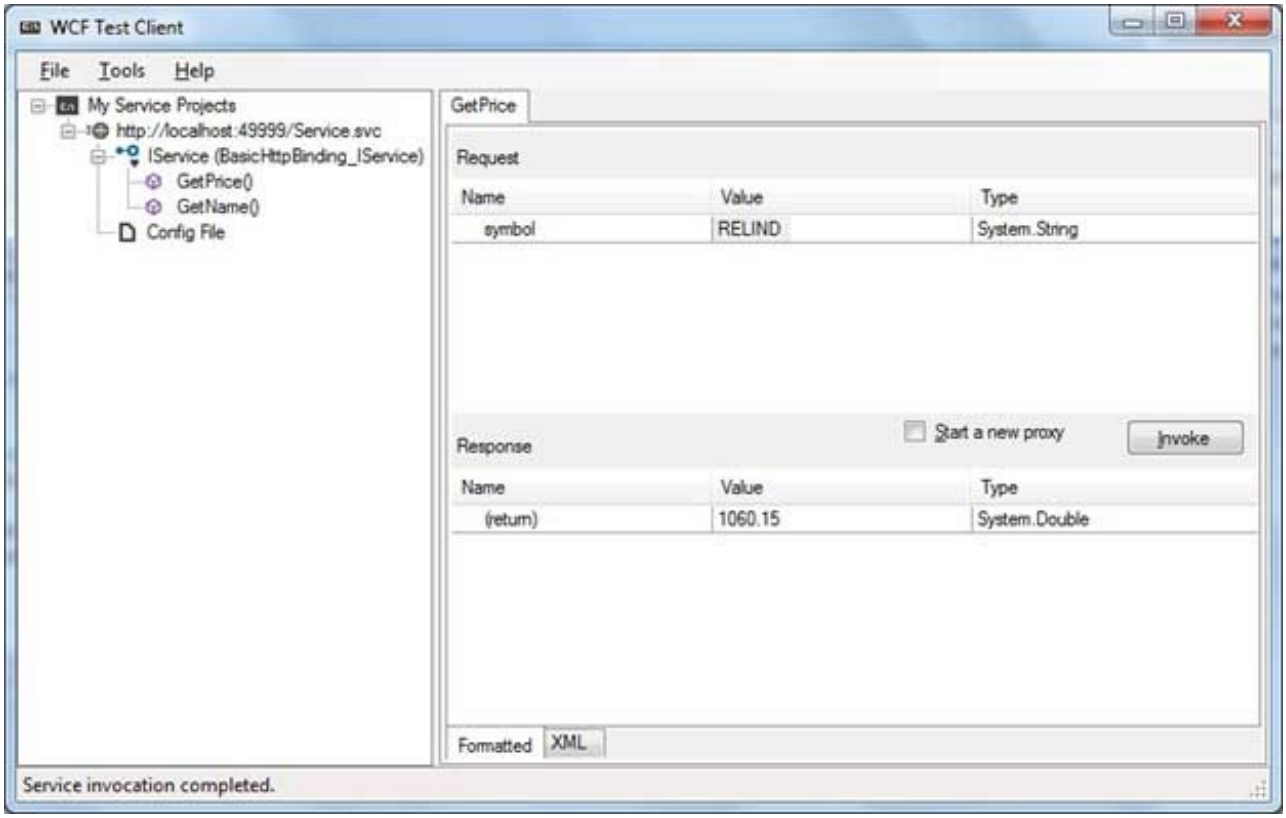
To run the WCF Service, so created, select the Debug->Start Debugging option from the menu bar. The output would be:



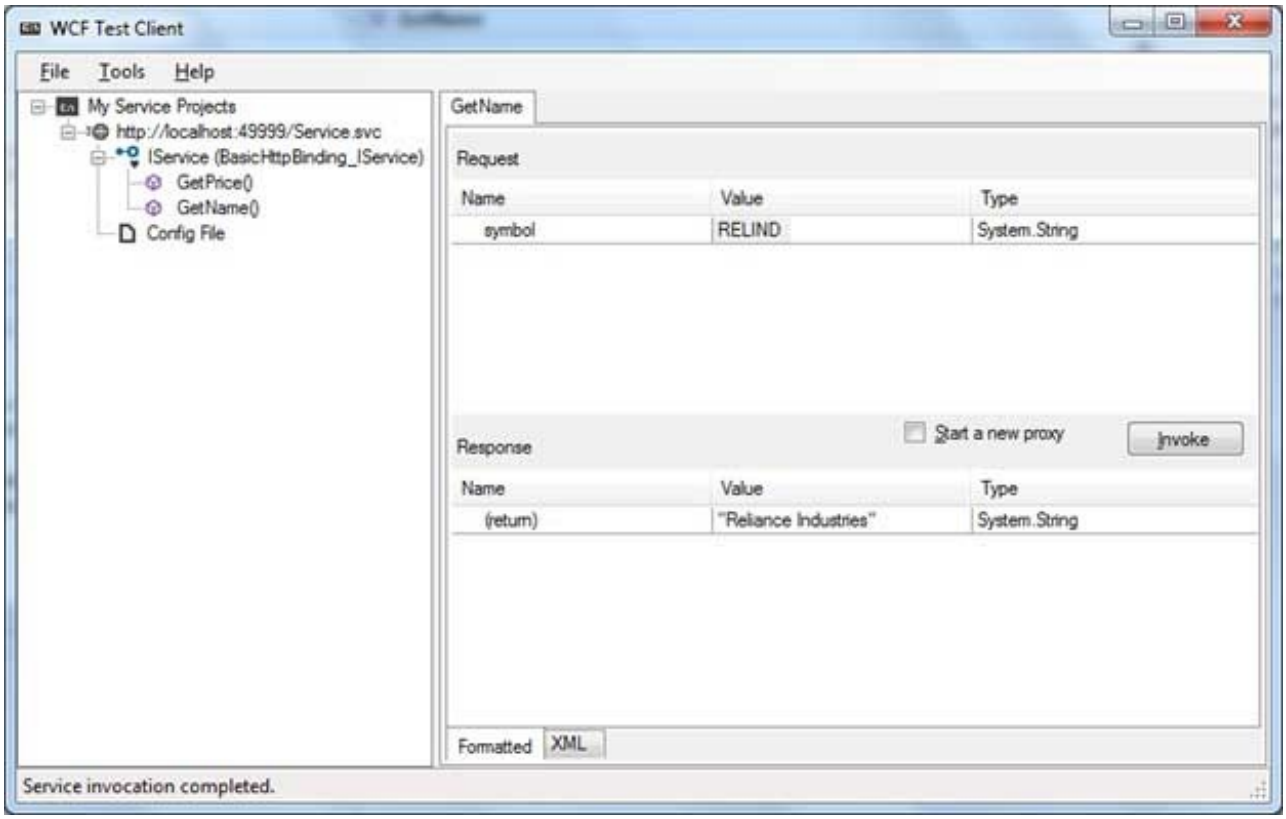
For testing the service operations, double click the name of the operation from the tree on the left pane. A new tab will appear on the right pane.

Enter the value of parameters in the Request area of the right pane and click the 'Invoke' button.

The following diagram displays the result of testing the **GetPrice** operation:



The following diagram displays the result of testing the **GetName** operation:



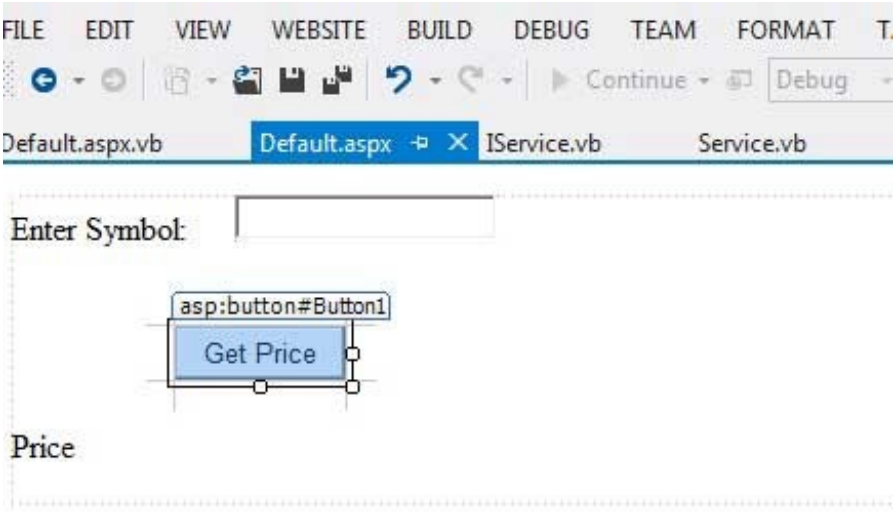
Utilizing the Service

Let us add a default page, a ASP.NET web form in the same solution from which we will be using the WCF Service we have just created.

Take the following steps:

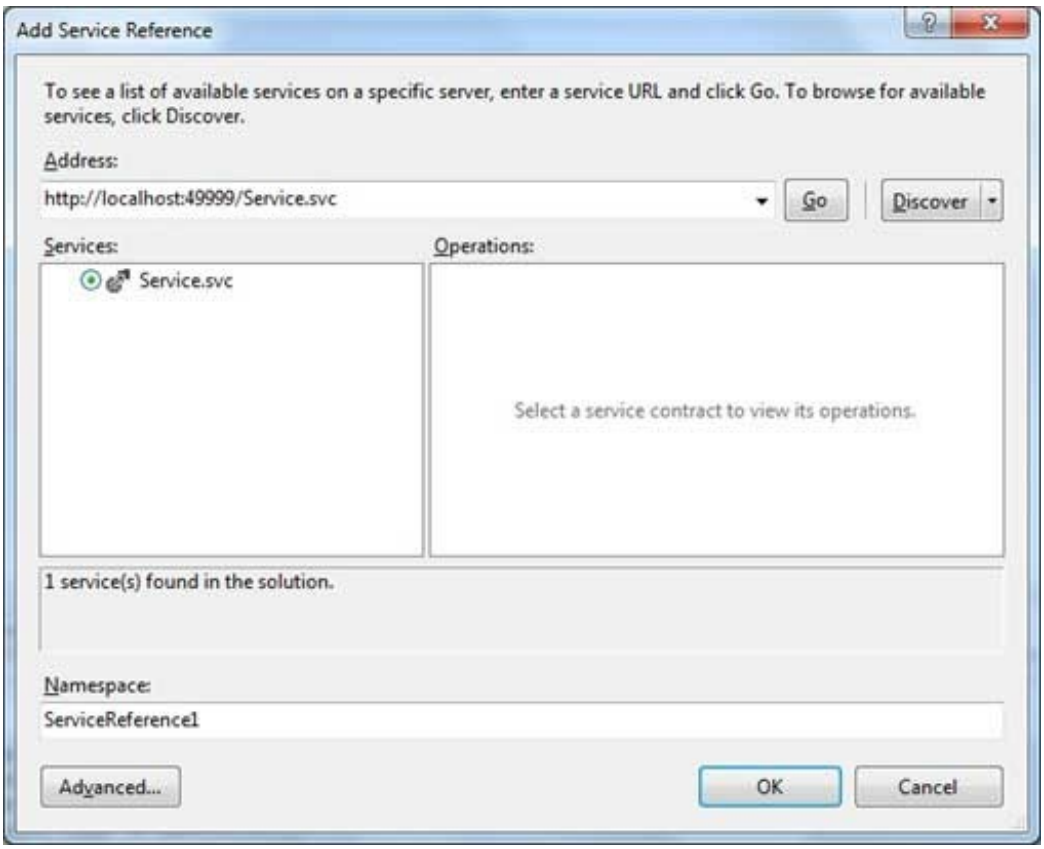
1. Right click on the solution name in the Solution Explorer and add a new web form to the solution. It will be named Default.aspx.

2. Add two labels, a text box and a button on the form.



3. We need to add a service reference to the WCF service we just created. Right click the website in the Solution Explorer and select Add Service Reference option. This opens the Add Service Reference Dialog box.

4. Enter the URL(location) of the Service in the Address text box and click the Go button. It creates a service reference with the default name **ServiceReference1**. Click the OK button.



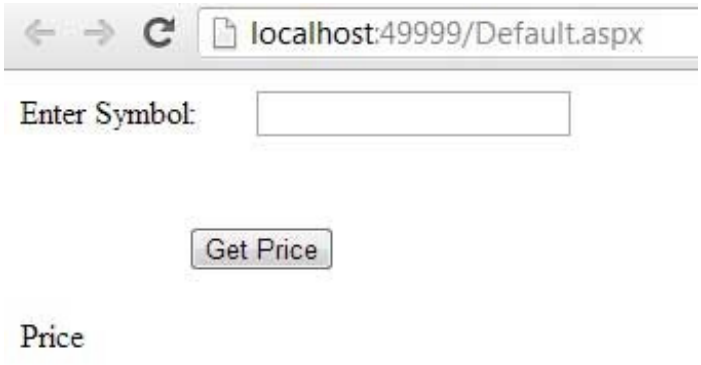
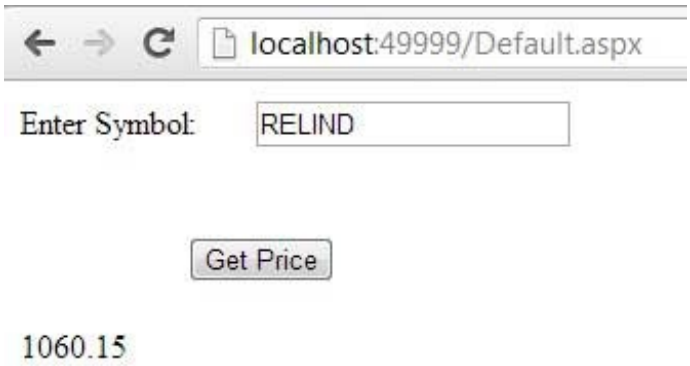
Adding the reference does two jobs for your project:

- Creates the Address and Binding for the service in the web.config file.
- Creates a proxy class to access the service.

5. Double click the Get Price button in the form, to enter the following code snippet on its Click event:

```
Partial Class _Default
   Inherits System.Web.UI.Page

   Protected Sub Button1_Click(sender As Object, e As EventArgs) _
   Handles Button1.Click
      Dim ser As ServiceReference1.ServiceClient = _
      New ServiceReference1.ServiceClient
      Label2.Text = ser.GetPrice(TextBox1.Text).ToString()
   End Sub
End Class
```

When the above code is executed and run using Start button available at the Microsoft Visual Studio tool bar, the following page opens in the browser:



Enter a symbol and click the Get Price button to get the hard-coded price: