



VB.NET

programming language reference

tutorialspoint
SIMPLY EASY LEARNING

www.tutorialspoint.com



<https://www.facebook.com/tutorialspointindia>



<https://twitter.com/tutorialspoint>

About the Tutorial

VB.Net is a simple, modern, object-oriented computer programming language developed by Microsoft to combine the power of .NET Framework and the common language runtime with the productivity benefits that are the hallmark of Visual Basic.

This tutorial will teach you basic VB.Net programming and will also take you through various advanced concepts related to VB.Net programming language.

Audience

This tutorial has been prepared for the beginners to help them understand basic VB.Net programming. After completing this tutorial, you will find yourself at a moderate level of expertise in VB.Net programming from where you can take yourself to next levels.

Prerequisites

VB.Net programming is very much based on BASIC and Visual Basic programming languages, so if you have basic understanding on these programming languages, then it will be a fun for you to learn VB.Net programming language.

Copyright & Disclaimer

© Copyright 2015 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book can retain a copy for future reference but commercial use of this data is not allowed. Distribution or republishing any content or a part of the content of this e-book in any manner is also not allowed without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com

Table of Contents

About the Tutorial.....	1
Audience	1
Prerequisites	1
Copyright & Disclaimer.....	1
Table of Contents.....	2
 1. OVERVIEW.....	8
Strong Programming Features VB.Net.....	8
 2. ENVIRONMENT SETUP.....	10
The .Net Framework	10
Integrated Development Environment (IDE) For VB.Net	11
Writing VB.Net Programs on Linux or Mac OS	11
 3. PROGRAM STRUCTURE.....	12
VB.Net Hello World Example.....	12
Compile & Execute VB.Net Program	13
 4. BASIC SYNTAX.....	15
A Rectangle Class in VB.Net.....	15
Identifiers.....	17
VB.Net Keywords	17
 5. DATA TYPES.....	19
Data Types Available in VB.Net	19
Example	21
The Type Conversion Functions in VB.Net	22
Example	24

6.	VARIABLES.....	25
	Variable Declaration in VB.Net.....	25
	Variable Initialization in VB.Net	27
	Example	27
	Accepting Values from User	28
	Lvalues and Rvalues	28
7.	CONSTANTS AND ENUMERATIONS.....	30
	Declaring Constants	30
	Example	31
	Print and Display Constants in VB.Net.....	31
	Declaring Enumerations	32
	Example	33
8.	MODIFIERS	35
	List of Available Modifiers in VB.Net	35
9.	STATEMENTS.....	40
	Declaration Statements.....	40
	Executable Statements.....	44
10.	DIRECTIVES.....	45
	Compiler Directives in VB.Net	45
11.	OPERATORS.....	50
	Arithmetic Operators	50
	Example	51
	Comparison Operators	52
	Logical/Bitwise Operators	54
	Example	55
	Bit Shift Operators	57

Example	59
Assignment Operators.....	60
Example	61
Miscellaneous Operators	62
Example	63
Operators Precedence in VB.Net	64
Example	65
12. DECISION MAKING.....	67
If...Then Statement	68
If...Then...Else Statement	70
The If...Else If...Else Statement.....	71
Nested If Statements.....	73
Select Case Statement.....	74
Nested Select Case Statement.....	76
13. LOOPS	78
Do Loop.....	79
For...Next Loop.....	82
Each...Next Loop	84
While... End While Loop	85
With... End With Statement	88
Nested Loops	89
Loop Control Statements.....	91
Exit Statement	92
Continue Statement.....	94
GoTo Statement.....	95

14. STRINGS.....	98
Creating a String Objec.....	98
Properties of the String Class	99
Methods of the String Class.....	99
Examples	105
15. DATE & TIME	108
Properties and Methods of the DateTime Structure.....	109
Creating a DateTime Object	112
Getting the Current Date and Time	113
Formatting Date.....	114
Predefined Date/Time Formats	115
Properties and Methods of the DateAndTime Class	117
16. ARRAYS.....	121
Creating Arrays in VB.Net.....	121
Dynamic Arrays	122
Multi-Dimensional Arrays	124
Jagged Array.....	125
The Array Class.....	126
17. COLLECTIONS	131
Various Collection Classes and Their Usage	131
ArrayList.....	132
Hashtable.....	136
SortedList.....	138
Stack	142
Queue	144
BitArray.....	146

18. FUNCTIONS	151
Defining a Function	151
Example	151
Function Returning a Value	152
Recursive Function	153
Param Arrays	154
Passing Arrays as Function Arguments	154
19. SUB PROCEDURES	156
Defining Sub Procedures	156
Example	156
Passing Parameters by Value	157
Passing Parameters by Reference.....	158
20. CLASSES & OBJECTS.....	160
Class Definition	160
Member Functions and Encapsulation	162
Constructors and Destructors.....	163
Shared Members of a VB.Net Class	166
Inheritance.....	167
Base & Derived Classes.....	167
Base Class Initialization	169
21. EXCEPTION HANDLING	171
Syntax	171
Exception Classes in .Net Framework.....	172
Handling Exceptions	173
Creating User-Defined Exceptions	174
Throwing Objects	175

22. FILE HANDLING.....	176
Binary Files.....	183
23. BASIC CONTROLS.....	193
24. DIALOG BOXES.....	286
25. ADVANCED FORM	308
26. EVENT HANDLING.....	331
27. REGULAR EXPRESSIONS.....	337
28. DATABASE ACCESS.....	351
29. EXCEL SHEET.....	366
30. SEND EMAIL	371
31. XML PROCESSING	377
32. WEB PROGRAMMING.....	392

1. 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 inherit from the base class Object.

VB.NET is implemented by Microsoft's .NET framework. Therefore, it has full access to 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

2. Environment Setup

In this chapter, we will discuss the tools available for creating VB.Net applications.

We have already mentioned that VB.Net is part of .Net framework and used for writing .Net applications. Therefore before discussing the available tools for running a VB.Net program, let us understand how VB.Net relates to the .Net framework.

The .Net Framework

The .Net framework is a revolutionary platform that helps you to write the following types of applications:

- Windows applications
- Web applications
- Web services

The .Net framework applications are multi-platform applications. The framework has been designed in such a way that it can be used from any of the following languages: Visual Basic, C#, C++, Jscript, and COBOL, etc.

All these languages can access the framework as well as communicate with each other.

The .Net framework consists of an enormous library of codes used by the client languages like VB.Net. These languages use object-oriented methodology.

Following are some of the components of the .Net framework:

- Common Language Runtime (CLR)
- The .Net Framework Class Library
- Common Language Specification
- Common Type System
- Metadata and Assemblies
- Windows Forms
- ASP.Net and ASP.Net AJAX
- ADO.Net

- Windows Workflow Foundation (WF)
- Windows Presentation Foundation
- Windows Communication Foundation (WCF)
- LINQ

For the jobs each of these components perform, please see [ASP.Net - Introduction](#), and for details of each component, please consult Microsoft's documentation.

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.

3. Program Structure

Before we study basic building blocks of the VB.Net programming language, let us look at 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 the 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 or 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.

You can compile a VB.Net program by using the command line instead of the Visual Studio IDE:

- Open a text editor and add the above mentioned code.
- Save the file as **helloworld.vb**
- Open the command prompt tool and go to the directory where you saved the file.
- Type **vbc helloworld.vb** and press enter to compile your code.
- If there are no errors in your code the command prompt will take you to the next line and would generate **helloworld.exe** executable file.
- Next, type **helloworld** to execute your program.
- You will be able to see "Hello World" printed on the screen.

4. 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/blueprint that describes 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 displaying 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 the 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		

5. 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.WriteLine(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

```

```
End Sub
```

```
End Module
```

When the above code is compiled and executed, it produces the 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 in-line type conversion functions:

S.N	Functions & 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 the following result:

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

6. 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 the following example which makes use of various types of variables:

```

Module variablesNdatatype
    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 the 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 variablesNdatatypes
    Sub Main()
        Dim message As String
        Console.WriteLine("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 the following result (assume the user inputs Hello World):

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

Lvalues and Rvalues

There are two kinds of expressions:

- **lvalue** : An expression that is an lvalue may appear as either the left-hand or right-hand side of an assignment.
- **rvalue** : An expression that is an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned and can not appear on the left-hand side. Following is a valid statement:

```
Dim g As Integer = 20
```

But following is not a valid statement and would generate compile-time error:

```
20 = g
```

7. 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 the 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.
- **accessmodifier**: 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 the following result:

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

8. 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., indicate 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 <i>charsetmodifier</i> 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 Function 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.

9. 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.	<pre>Dim number As Integer Dim quantity As Integer = 100 Dim message As String = "Hello!"</pre>
2	Const Statement Declares and defines one or more constants.	<pre>Const maximum As Long = 1000</pre>

		<pre>Const naturalLogBase As Object = CDec(2.7182818284)</pre>
3	<p>Enum Statement</p> <p>Declares an enumeration and defines the values of its members.</p>	<pre>Enum CoffeeMugSize Jumbo ExtraLarge Large Medium Small End Enum</pre>
4	<p>Class Statement</p> <p>Declares the name of a class and introduces the definition of the variables, properties, events, and procedures that the class comprises.</p>	<pre>Class Box Public length As Double Public breadth As Double Public height As Double End Class</pre>
5	<p>Structure Statement</p> <p>Declares the name of a structure and introduces the definition of the variables, properties, events, and procedures that the structure comprises.</p>	<pre>Structure Box Public length As Double Public breadth As Double Public height As Double End Structure</pre>

6	<p>Module Statement</p> <p>Declares the name of a module and introduces the definition of the variables, properties, events, and procedures that the module comprises.</p>	<pre>Public Module myModule Sub Main() Dim user As String = InputBox("What is your name?") MsgBox("User name is" & user) End Sub End Module</pre>
7	<p>Interface Statement</p> <p>Declares the name of an interface and introduces the definitions of the members that the interface comprises.</p>	<pre>Public Interface MyInterface Sub doSomething() End Interface</pre>
8	<p>Function Statement</p> <p>Declares the name, parameters, and code that define a Function procedure.</p>	<pre>Function myFunction (ByVal n As Integer) As Double Return 5.87 * n End Function</pre>
9	<p>Sub Statement</p> <p>Declares the name, parameters, and code that define a Sub procedure.</p>	<pre>Sub mySub(ByVal s As String) Return End Sub</pre>
10	<p>Declare Statement</p> <p>Declares a reference to a procedure implemented in an external file.</p>	<pre>Declare Function getUserName Lib "advapi32.dll" Alias "GetUserNameA" (</pre>

		<pre>ByVal lpBuffer As String, ByRef nSize As Integer) As Integer</pre>
11	<p>Operator Statement</p> <p>Declares the operator symbol, operands, and code that define an operator procedure on a class or structure.</p>	<pre>Public Shared Operator + (ByVal x As obj, ByVal y As obj) As obj Dim r As New obj ' implementation code for r = x + y Return r End Operator</pre>
12	<p>Property Statement</p> <p>Declares the name of a property, and the property procedures used to store and retrieve the value of the property.</p>	<pre>ReadOnly Property quote() As String Get Return quoteString End Get End Property</pre>
13	<p>Event Statement</p> <p>Declares a user-defined event.</p>	<pre>Public Event Finished()</pre>
14	<p>Delegate Statement</p> <p>Used to declare a delegate.</p>	<pre>Delegate Function MathOperator(ByVal x As Double, ByVal y As Double) As Double</pre>

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 the following result:

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

10. Directives

The VB.Net compiler directives give instructions 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 the 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 the 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 the 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
```

11. 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 types 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:

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
$*$	Multiplies both operands	$A * B$ will give 14
$/$	Divides one operand by another and returns a floating point result	B / A will give 3.5
\backslash	Divides one operand by another and returns an integer result	$B \backslash A$ will give 3
MOD	Modulus Operator and remainder of after an integer division	$B \text{ MOD } A$ will give 1

Example

Try the following example to understand all the arithmetic operators available in VB.Net:

```
Module operators
Sub Main()
    Dim a As Integer = 21
    Dim b As Integer = 10
    Dim p As Integer = 2
    Dim c As Integer
    Dim d As Single
    c = a + b
    Console.WriteLine("Line 1 - Value of c is {0}", c)
    c = a - b
    Console.WriteLine("Line 2 - Value of c is {0}", c)
    c = a * b
    Console.WriteLine("Line 3 - Value of c is {0}", c)
    d = a / b
    Console.WriteLine("Line 4 - Value of d is {0}", d)
    c = a \ b
    Console.WriteLine("Line 5 - Value of c is {0}", c)
    c = a Mod b
    Console.WriteLine("Line 6 - Value of c is {0}", c)
    c = b ^ p
    Console.WriteLine("Line 7 - Value of c is {0}", c)
    Console.ReadLine()
End Sub
End Module
```

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

```
Line 1 - Value of c is 31
Line 2 - Value of c is 11
Line 3 - Value of c is 210
Line 4 - Value of d is 2.1
Line 5 - Value of c is 2
```

Line 6 - Value of c is 1
 Line 7 - Value of c is 100

Comparison Operators

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

Operator	Description	Example
=	Checks if the values of two operands are equal or not; if yes, then condition becomes true.	(A = B) is not true.
<>	Checks if the values of two operands are 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.

Try the following example to understand all the relational operators available in VB.Net:

```
Module operators
Sub Main()
    Dim a As Integer = 21
    Dim b As Integer = 10
```

```

If (a = b) Then
    Console.WriteLine("Line 1 - a is equal to b")
Else
    Console.WriteLine("Line 1 - a is not equal to b")
End If
If (a < b) Then
    Console.WriteLine("Line 2 - a is less than b")
Else
    Console.WriteLine("Line 2 - a is not less than b")
End If
If (a > b) Then
    Console.WriteLine("Line 3 - a is greater than b")
Else
    Console.WriteLine("Line 3 - a is not greater than b")
End If
' Lets change value of a and b
a = 5
b = 20
If (a <= b) Then
    Console.WriteLine("Line 4 - a is either less than or equal to
b")
End If
If (b >= a) Then
    Console.WriteLine("Line 5 - b is either greater than or equal
to b")
End If
Console.ReadLine()
End Sub
End Module

```

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

```

Line 1 - a is not equal to b
Line 2 - a is not less than b
Line 3 - a is greater than b
Line 4 - a is either less than or equal to b

```

Line 5 - b is either greater than or equal to b

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.

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:

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.	

Example

Try the following example to understand all the logical/bitwise operators available in VB.Net:

```
Module logicalOp

Sub Main()
    Dim a As Boolean = True
    Dim b As Boolean = True
    Dim c As Integer = 5
```

```

Dim d As Integer = 20
'logical And, Or and Xor Checking

If (a And b) Then
    Console.WriteLine("Line 1 - Condition is true")
End If

If (a Or b) Then
    Console.WriteLine("Line 2 - Condition is true")
End If

If (a Xor b) Then
    Console.WriteLine("Line 3 - Condition is true")
End If

'bitwise And, Or and Xor Checking

If (c And d) Then
    Console.WriteLine("Line 4 - Condition is true")
End If

If (c Or d) Then
    Console.WriteLine("Line 5 - Condition is true")
End If

If (c Or d) Then
    Console.WriteLine("Line 6 - Condition is true")
End If

'Only logical operators

If (a AndAlso b) Then
    Console.WriteLine("Line 7 - Condition is true")
End If

If (a OrElse b) Then
    Console.WriteLine("Line 8 - Condition is true")
End If

' lets change the value of a and b
a = False
b = True
If (a And b) Then
    Console.WriteLine("Line 9 - Condition is true")

```

```

Else
    Console.WriteLine("Line 9 - Condition is not true")
End If
If (Not (a And b)) Then
    Console.WriteLine("Line 10 - Condition is true")
End If
Console.ReadLine()
End Sub
End Module

```

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

```

Line 1 - Condition is true
Line 2 - Condition is true
Line 3 - Condition is true
Line 4 - Condition is true
Line 5 - Condition is true
Line 6 - Condition is true
Line 7 - Condition is true
Line 8 - Condition is true
Line 9 - Condition is not true
Line 10 - Condition 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 operations. 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:

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 -61, which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the	A << 2 will give 240, which is 1111 0000

	number of bits specified by the right operand.	
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15, which is 0000 1111

Example

Try the following example to understand all the bitwise operators available in VB.Net:

```
Module BitwiseOp
    Sub Main()
        Dim a As Integer = 60          ' 60 = 0011 1100
        Dim b As Integer = 13          ' 13 = 0000 1101
        Dim c As Integer = 0
        c = a And b      ' 12 = 0000 1100
        Console.WriteLine("Line 1 - Value of c is {0}", c)
        c = a Or b       ' 61 = 0011 1101
        Console.WriteLine("Line 2 - Value of c is {0}", c)
        c = a Xor b      ' 49 = 0011 0001
        Console.WriteLine("Line 3 - Value of c is {0}", c)
        c = Not a         ' -61 = 1100 0011
        Console.WriteLine("Line 4 - Value of c is {0}", c)
        c = a << 2       ' 240 = 1111 0000
        Console.WriteLine("Line 5 - Value of c is {0}", c)
        c = a >> 2       ' 15 = 0000 1111
        Console.WriteLine("Line 6 - Value of c is {0}", c)
        Console.ReadLine()
    End Sub
End Module
```

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

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
```

```

Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15

```

Assignment Operators

There are following assignment operators supported by VB.Net:

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 assigns 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 assigns 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 assigns 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 assigns 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 assigns 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

Example

Try the following example to understand all the assignment operators available in VB.Net:

```
Module assignment
Sub Main()
    Dim a As Integer = 21
    Dim pow As Integer = 2
    Dim str1 As String = "Hello! "
    Dim str2 As String = "VB Programmers"
    Dim c As Integer
    c = a
    Console.WriteLine("Line 1 - = Operator Example, _
        Value of c = {0}", c)
    c += a
    Console.WriteLine("Line 2 - += Operator Example, _
        Value of c = {0}", c)
    c -= a
    Console.WriteLine("Line 3 - -= Operator Example, _
        Value of c = {0}", c)
    c *= a
    Console.WriteLine("Line 4 - *= Operator Example, _
        Value of c = {0}", c)
    c /= a
    Console.WriteLine("Line 5 - /= Operator Example, _")
```

```

Value of c = {0}", c)
c = 20
c ^= pow
Console.WriteLine("Line 6 - ^= Operator Example, _
Value of c = {0}", c)
c <= 2
Console.WriteLine("Line 7 - <= Operator Example,_
Value of c = {0}", c)
c >= 2
Console.WriteLine("Line 8 - >= Operator Example,_
Value of c = {0}", c)
str1 &= str2
Console.WriteLine("Line 9 - &= Operator Example,_
Value of str1 = {0}", str1)
Console.ReadLine()
End Sub
End Module

```

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

```

Line 1 - = Operator Example, Value of c = 21
Line 2 - += Operator Example, Value of c = 42
Line 3 - -= Operator Example, Value of c = 21
Line 4 - *= Operator Example, Value of c = 441
Line 5 - /= Operator Example, Value of c = 21
Line 6 - ^= Operator Example, Value of c = 400
Line 7 - <= Operator Example, Value of c = 1600
Line 8 - >= Operator Example, Value of c = 400
Line 9 - &= Operator Example, Value of str1 = Hello! VB Programmers

```

Miscellaneous Operators

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

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"))

Example

The following example demonstrates some of these operators:

```
Module assignment
```

```
Sub Main()
```

```

Dim a As Integer = 21
Console.WriteLine(GetType(Integer).ToString())
Console.WriteLine(GetType(Double).ToString())
Console.WriteLine(GetType(String).ToString())
Dim multiplywith5 = Function(num As Integer) num * 5
Console.WriteLine(multiplywith5(5))
Console.WriteLine(If(a >= 0, "Positive", "Negative"))
Console.ReadLine()
End Sub
End Module

```

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

```

System.Int32
System.Double
System.String
25
Positive

```

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 gets 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.

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

Example

The following example demonstrates operator precedence in a simple way:

```
Module assignment

Sub Main()
    Dim a As Integer = 20
    Dim b As Integer = 10
    Dim c As Integer = 15
    Dim d As Integer = 5
    Dim e As Integer
    e = (a + b) * c / d      ' ( 30 * 15 ) / 5
    Console.WriteLine("Value of (a + b) * c / d is : {0}", e)
    e = ((a + b) * c) / d    ' (30 * 15 ) / 5
```

```
Console.WriteLine("Value of ((a + b) * c) / d is : {0}", e)
e = (a + b) * (c / d)    ' (30) * (15/5)
Console.WriteLine("Value of (a + b) * (c / d) is : {0}", e)
e = a + (b * c) / d      ' 20 + (150/5)
Console.WriteLine("Value of a + (b * c) / d is : {0}", e)
Console.ReadLine()

End Sub

End Module
```

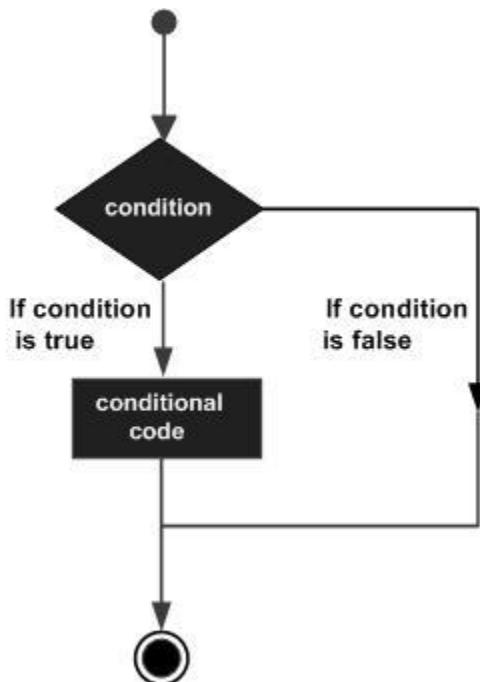
When the above code is compiled and executed, it produces the following result:

```
Value of (a + b) * c / d is : 90
Value of ((a + b) * c) / d is : 90
Value of (a + b) * (c / d) is : 90
Value of a + (b * c) / d is : 50
```

12. 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 form of a typical decision making structure found in most of the programming languages:



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

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).

If...Then Statement

It is the simplest form of control statement, frequently used in decision making and changing the control flow of the program execution. Syntax for if-then statement is:

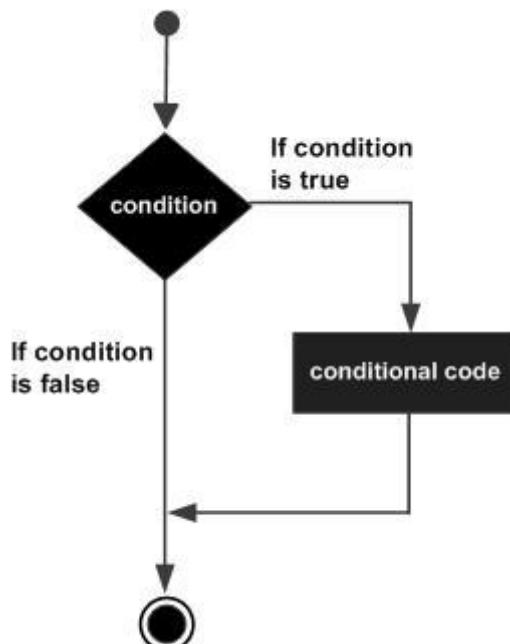
```
If condition Then
[Statement(s)]
End If
```

Where, *condition* is a Boolean or relational condition and *Statement(s)* is a simple or compound statement. Example of an If-Then statement is:

```
If (a <= 20) Then
    c= c+1
End If
```

If the condition evaluates to true, then the block of code inside the If statement will be executed. If condition evaluates to false, then the first set of code after the end of the If statement (after the closing End If) will be executed.

Flow Diagram



Example

```

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 the following result:

```

a is less than 20
value of a is : 10

```

If...Then...Else Statement

An **If** statement can be followed by an optional **Else** statement, which executes when the Boolean expression is false.

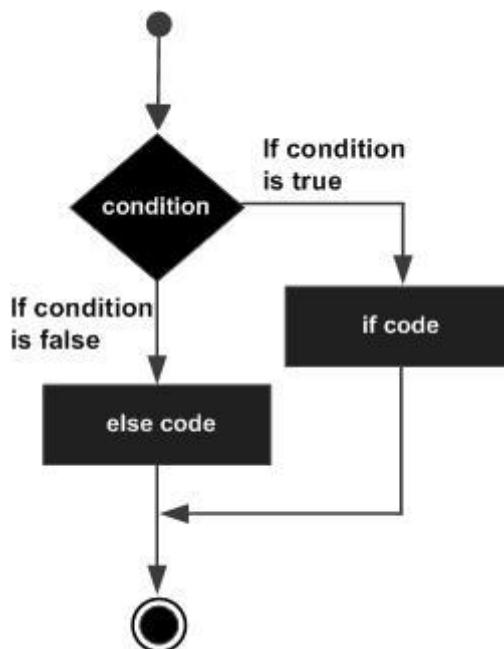
Syntax

The syntax of an If...Then... Else statement in VB.Net is as follows:

```
If(boolean_expression)Then
    'statement(s) will execute if the Boolean expression is true
Else
    'statement(s) will execute if the Boolean expression is false
End If
```

If the Boolean expression evaluates to **true**, then the if block of code will be executed, otherwise else block of code will be executed.

Flow Diagram



Example

```
Module decisions
Sub Main()
    'local variable definition '
    Dim a As Integer = 100
```

```

' 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")
Else
    ' if condition is false then print the following
    Console.WriteLine("a is not 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 the following result:

```

a is not less than 20
value of a is : 100

```

The If...Else If...Else Statement

An **If** statement can be followed by an optional **Else if...Else** statement, which is very useful to test various conditions using single If...Else If statement.

When using If... Else If... Else statements, there are few points to keep in mind.

- An If can have zero or one Else's and it must come after an Else If's.
- An If can have zero to many Else If's and they must come before the Else.
- Once an Else if succeeds, none of the remaining Else If's or Else's will be tested.

Syntax

The syntax of an if...else if...else statement in VB.Net is as follows:

```

If(boolean_expression 1)Then
    ' Executes when the boolean expression 1 is true
ElseIf( boolean_expression 2)Then
    ' Executes when the boolean expression 2 is true
ElseIf( boolean_expression 3)Then

```

```

' Executes when the boolean expression 3 is true
Else
    ' executes when the none of the above condition is true
End If

```

Example

```

Module decisions
Sub Main()
    'local variable definition '
    Dim a As Integer = 100
    ' check the boolean condition '
    If (a = 10) Then
        ' if condition is true then print the following '
        Console.WriteLine("Value of a is 10") '
    ElseIf (a = 20) Then
        'if else if condition is true '
        Console.WriteLine("Value of a is 20") '
    ElseIf (a = 30) Then
        'if else if condition is true
        Console.WriteLine("Value of a is 30")
    Else
        'if none of the conditions is true
        Console.WriteLine("None of the values is matching")
    End If
    Console.WriteLine("Exact value of a is: {0}", a)
    Console.ReadLine()
End Sub
End Module

```

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

```

None of the values is matching
Exact value of a is: 100

```

Nested If Statements

It is always legal in VB.Net to nest If-Then-Else statements, which means you can use one If or ElseIf statement inside another If ElseIf statement(s).

Syntax

The syntax for a nested If statement is as follows:

```
If( boolean_expression 1)Then
    'Executes when the boolean expression 1 is true
    If(boolean_expression 2)Then
        'Executes when the boolean expression 2 is true
    End If
End If
```

You can nest ElseIf...Else in the similar way as you have nested If statement.

Example

```
Module decisions
Sub Main()
    'local variable definition
    Dim a As Integer = 100
    Dim b As Integer = 200
    ' check the boolean condition
    If (a = 100) Then
        ' if condition is true then check the following
        If (b = 200) Then
            ' if condition is true then print the following
            Console.WriteLine("Value of a is 100 and b is 200")
        End If
    End If
    Console.WriteLine("Exact value of a is : {0}", a)
    Console.WriteLine("Exact value of b is : {0}", b)
    Console.ReadLine()
End Sub
End Module
```

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

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

Select Case Statement

A **Select Case** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each select case.

Syntax

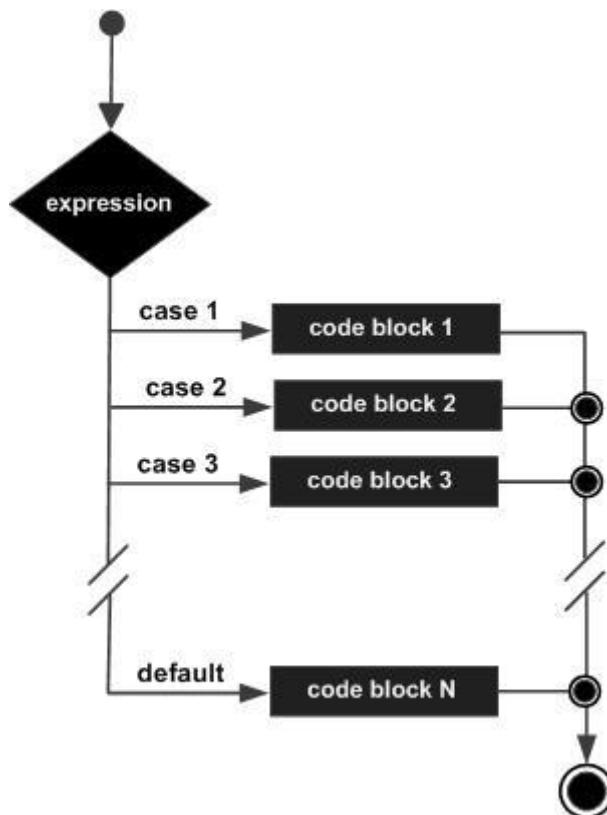
The syntax for a Select Case statement in VB.Net is as follows:

```
Select [ Case ] expression
    [ Case expressionlist
        [ statements ] ]
    [ Case Else
        [ elsestatements ] ]
End Select
```

Where,

- **expression**: is an expression that must evaluate to any of the elementary data type in VB.Net, i.e., Boolean, Byte, Char, Date, Double, Decimal, Integer, Long, Object, SByte, Short, Single, String, UInteger, ULong, and UShort.
- **expressionlist**: List of expression clauses representing match values for *expression*. Multiple expression clauses are separated by commas.
- **statements**: statements following Case that run if the select expression matches any clause in *expressionlist*.
- **elsestatements**: statements following Case Else that run if the select expression does not match any clause in the *expressionlist* of any of the Case statements.

Flow Diagram



Example

```

Module decisions

Sub Main()
    'local variable definition
    Dim grade As Char
    grade = "B"
    Select grade
        Case "A"
            Console.WriteLine("Excellent!")
        Case "B", "C"
            Console.WriteLine("Well done")
        Case "D"
            Console.WriteLine("You passed")
        Case "F"
            Console.WriteLine("Better try again")
        Case Else
    End Select
End Sub
  
```

```

        Console.WriteLine("Invalid grade")
End Select

Console.WriteLine("Your grade is {0}", grade)
Console.ReadLine()
End Sub
End Module

```

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

```

Well done
Your grade is B

```

Nested Select Case Statement

It is possible to have a select statement as part of the statement sequence of an outer select statement. Even if the case constants of the inner and outer select contain common values, no conflicts will arise.

Example

```

Module decisions
Sub Main()
    'local variable definition
    Dim a As Integer = 100
    Dim b As Integer = 200
    Select a
        Case 100
            Console.WriteLine("This is part of outer case ")
            Select Case b
                Case 200
                    Console.WriteLine("This is part of inner case ")
            End Select
        End Select
        Console.WriteLine("Exact value of a is : {0}", a)
        Console.WriteLine("Exact value of b is : {0}", b)
        Console.ReadLine()
    End Sub

```

```
End Module
```

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

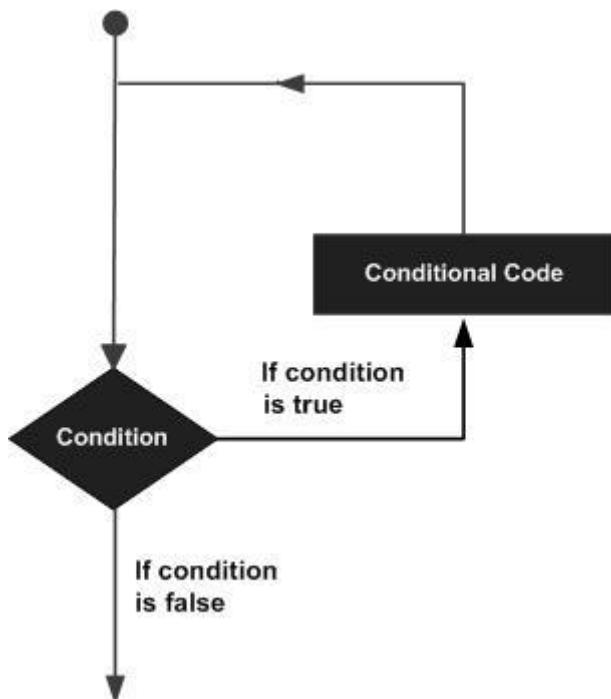
```
This is part of outer case  
This is part of inner case  
Exact value of a is : 100  
Exact value of b is : 200
```

13. 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 form of a loop statement in most of the programming languages:



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

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 refer to a single object or structure.
Nested loops	You can use one or more loops inside any another While, For or Do loop.

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.

The syntax for this loop construct is:

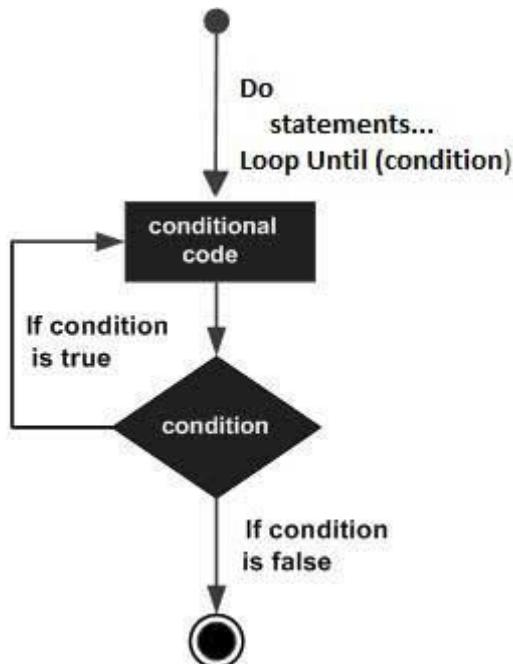
```

Do { While | Until } condition
    [ statements ]
    [ Continue Do ]
    [ statements ]
    [ Exit Do ]
    [ statements ]
Loop
-or-
Do
    [ statements ]
    [ Continue Do ]
    [ statements ]

```

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

Flow Diagram



Example

```
Module loops

Sub Main()
    ' local variable definition
    Dim a As Integer = 10
    'do loop execution
    Do
        Console.WriteLine("value of a: {0}", a)
        a = a + 1
    Loop While (a < 20)
    Console.ReadLine()
End Sub
End Module
```

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

```

value of a: 10
value of a: 11

value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

```

The program would behave in same way, if you use an Until statement, instead of While:

```

Module loops

Sub Main()
    ' local variable definition
    Dim a As Integer = 10
    'do loop execution
    Do
        Console.WriteLine("value of a: {0}", a)
        a = a + 1
    Loop Until (a = 20)
    Console.ReadLine()
End Sub
End Module

```

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

```

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16

```

```
value of a: 17  
value of a: 18  
value of a: 19
```

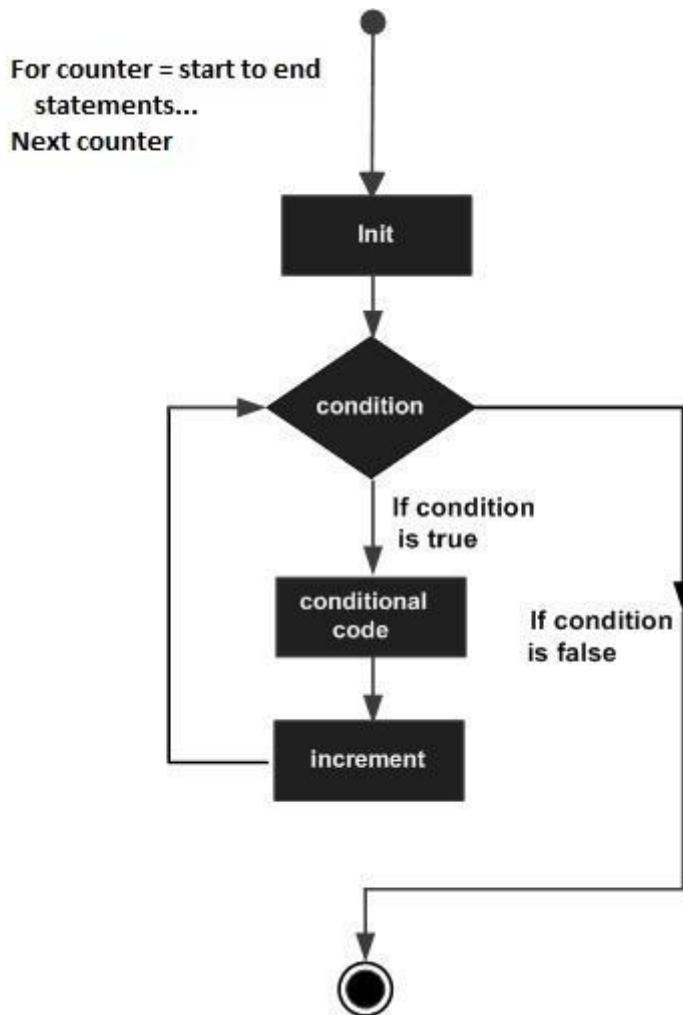
For...Next Loop

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.

The syntax for this loop construct is:

```
For counter [ As datatype ] = start To end [ Step step ]  
    [ statements ]  
    [ Continue For ]  
    [ statements ]  
    [ Exit For ]  
    [ statements ]  
Next [ counter ]
```

Flow Diagram



Example

```

Module loops
Sub Main()
    Dim a As Byte
    ' for loop execution
    For a = 10 To 20
        Console.WriteLine("value of a: {0}", a)
    Next
    Console.ReadLine()
End Sub
End Module

```

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

```
value of a: 10
```

```

value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
value of a: 20

```

If you want to use a step size of 2, for example, you need to display only even numbers, between 10 and 20:

```

Module loops
Sub Main()
    Dim a As Byte
    ' for loop execution
    For a = 10 To 20 Step 2
        Console.WriteLine("value of a: {0}", a)
    Next
    Console.ReadLine()
End Sub
End Module

```

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

```

value of a: 10
value of a: 12
value of a: 14
value of a: 16
value of a: 18
value of a: 20

```

Each...Next Loop

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.

The syntax for this loop construct is:

```
For Each element [ As datatype ] In group
    [ statements ]
    [ Continue For ]
    [ statements ]
    [ Exit For ]
    [ statements ]
Next [ element ]
```

Example

```
Module loops
Sub Main()
    Dim anArray() As Integer = {1, 3, 5, 7, 9}
    Dim arrayItem As Integer
    'displaying the values
    For Each arrayItem In anArray
        Console.WriteLine(arrayItem)
    Next
    Console.ReadLine()
End Sub
End Module
```

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

```
1
3
5
7
9
```

While... End While Loop

It executes a series of statements as long as a given condition is True.

The syntax for this loop construct is:

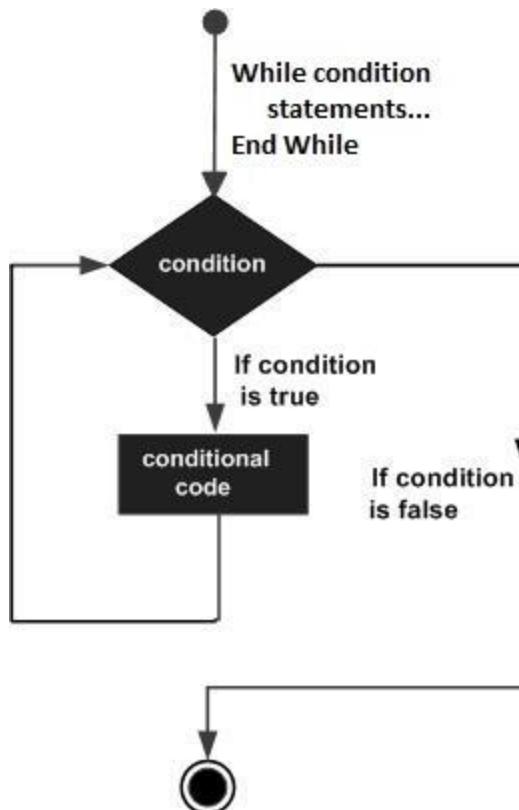
```
While condition
```

```
[ statements ]  
[ Continue While ]  
[ statements ]  
[ Exit While ]  
[ statements ]  
End While
```

Here, statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is logical true. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

Flow Diagram



Here, key point of the *While* loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```

Module loops

Sub Main()
    Dim a As Integer = 10
    ' while loop execution '
    While a < 20
        Console.WriteLine("value of a: {0}", a)
        a = a + 1
    End While
    Console.ReadLine()
End Sub
End Module
  
```

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

```

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

```

With... End With Statement

It is not exactly a looping construct. It executes a series of statements that repeatedly refers to a single object or structure.

The syntax for this loop construct is:

```

With object
    [ statements ]
End With

```

Example

```

Module loops

Public Class Book

    Public Property Name As String
    Public Property Author As String
    Public Property Subject As String

End Class

Sub Main()
    Dim aBook As New Book
    With aBook
        .Name = "VB.Net Programming"
        .Author = "Zara Ali"
        .Subject = "Information Technology"
    End With
End Sub

```

```

With aBook
    Console.WriteLine(.Name)
    Console.WriteLine(.Author)
    Console.WriteLine(.Subject)
End With
Console.ReadLine()
End Sub
End Module

```

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

```

VB.Net Programming
Zara Ali
Information Technology

```

Nested Loops

VB.Net allows using one loop inside another loop. Following section shows few examples to illustrate the concept.

Syntax

The syntax for a **nested For loop** statement in VB.Net is as follows:

```

For counter1 [ As datatype1 ] = start1 To end1 [ Step step1 ]
    For counter2 [ As datatype2 ] = start2 To end2 [ Step step2 ]
        ...
    Next [ counter2 ]
Next [ counter 1]

```

The syntax for a **nested While loop** statement in VB.Net is as follows:

```

While condition1
    While condition2
        ...
    End While
End While

```

The syntax for a **nested Do...While loop** statement in VB.Net is as follows:

```

Do { While | Until } condition1
  Do { While | Until } condition2
  ...
Loop
Loop

```

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example, a for loop can be inside a while loop or vice versa.

Example

The following program uses a nested for loop to find the prime numbers from 2 to 100:

```

Module loops
Sub Main()
    ' local variable definition
    Dim i, j As Integer
    For i = 2 To 100
        For j = 2 To i
            ' if factor found, not prime
            If ((i Mod j) = 0) Then
                Exit For
            End If
        Next j
        If (j > (i \ j)) Then
            Console.WriteLine("{0} is prime", i)
        End If
    Next i
    Console.ReadLine()
End Sub
End Module

```

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

```

2 is prime
3 is prime
5 is prime

```

```

7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime

```

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 details.

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.

Exit Statement

The Exit statement transfers the control from a procedure or block immediately to the statement following the procedure call or the block definition. It terminates the loop, procedure, try block or the select block from where it is called.

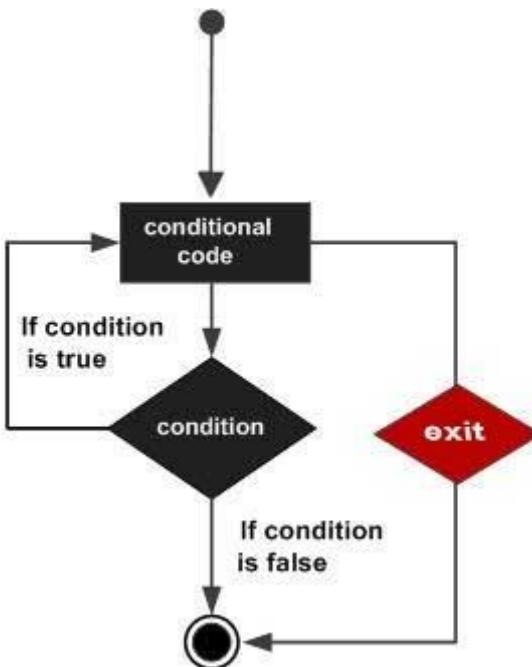
If you are using nested loops (i.e., one loop inside another loop), the Exit statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax

The syntax for the Exit statement is:

```
Exit { Do | For | Function | Property | Select | Sub | Try | While }
```

Flow Diagram



Example

Module loops

```

Sub Main()
    ' local variable definition
    Dim a As Integer = 10
    ' while loop execution '
    While (a < 20)
        Console.WriteLine("value of a: {0}", a)
        a = a + 1
        If (a > 15) Then
            'terminate the loop using exit statement
            Exit While
        End If
    End While
    Console.ReadLine()
End Sub
End Module
  
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
```

Continue Statement

The Continue statement causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. It works somewhat like the Exit statement. Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code in between.

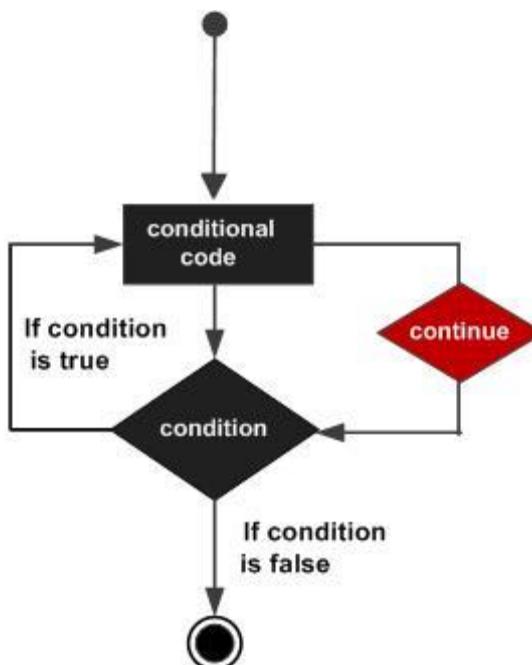
For the For...Next loop, Continue statement causes the conditional test and increment portions of the loop to execute. For the While and Do...While loops, continue statement causes the program control to pass to the conditional tests.

Syntax

The syntax for a Continue statement is as follows:

```
Continue { Do | For | While }
```

Flow Diagram



Example

```

Module loops

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

    Do
        If (a = 15) Then
            ' skip the iteration '
            a = a + 1
            Continue Do
        End If
        Console.WriteLine("value of a: {0}", a)
        a = a + 1
    Loop While (a < 20)
    Console.ReadLine()
End Sub
End Module

```

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

```

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19

```

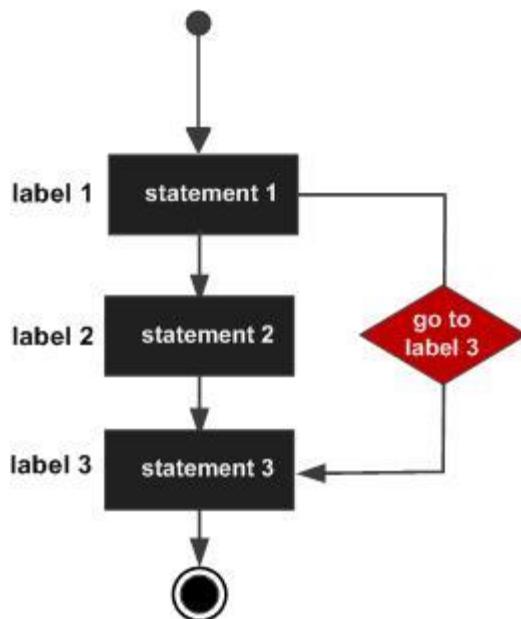
GoTo Statement

The GoTo statement transfers control unconditionally to a specified line in a procedure.

The syntax for the GoTo statement is:

```
GoTo label
```

Flow Diagram



Example

```

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

    Line1:
    Do
        If (a = 15) Then
            ' skip the iteration '
            a = a + 1
            GoTo Line1
        End If
        Console.WriteLine("value of a: {0}", a)
        a = a + 1
    Loop While (a < 20)
    Console.ReadLine()
End Sub
End Module
  
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

14. 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 Objec

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

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

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

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

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

Examples

The following example demonstrates some of the methods mentioned above:

Comparing Strings

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

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

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

String Contains String

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

```
End Module
```

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

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

Getting a Substring

```
Module strings
```

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

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

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

Joining Strings

```
Module strings
```

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

```
End Sub  
End Module
```

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

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

15. Date & Time

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

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

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

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

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

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

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

Note:

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

Properties and Methods of the DateTime Structure

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

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

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

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

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

4	<p>Public Function AddMinutes (value As Double) As DateTime</p> <p>Returns a new DateTime that adds the specified number of minutes to the value of this instance.</p>
5	<p>Public Function AddMonths (months As Integer) As DateTime</p> <p>Returns a new DateTime that adds the specified number of months to the value of this instance.</p>
6	<p>Public Function AddSeconds (value As Double) As DateTime</p> <p>Returns a new DateTime that adds the specified number of seconds to the value of this instance.</p>
7	<p>Public Function AddYears (value As Integer) As DateTime</p> <p>Returns a new DateTime that adds the specified number of years to the value of this instance.</p>
8	<p>Public Shared Function Compare (t1 As DateTime,t2 As DateTime) As Integer</p> <p>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.</p>
9	<p>Public Function CompareTo (value As DateTime) As Integer</p> <p>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.</p>
10	<p>Public Function Equals (value As DateTime) As Boolean</p> <p>Returns a value indicating whether the value of this instance is equal to the value of the specified DateTime instance.</p>
11	<p>Public Shared Function Equals (t1 As DateTime, t2 As DateTime) As Boolean</p>

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

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

Creating a DateTime Object

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

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

The following example demonstrates this:

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

```

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

```

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

```

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

```

Getting the Current Date and Time

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

Current Time

```

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

```

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

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

Current Date

```

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

```

```

Console.ReadKey()
End Sub
End Module

```

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

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

Formatting Date

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

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

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

The following example demonstrates this.

```

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

```

```

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

```

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

```

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

```

Predefined Date/Time Formats

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

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

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

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

For other formats like user-defined formats, please consult [Microsoft Documentation](#).

Properties and Methods of the DateAndTime Class

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

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

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

S.N	Method Name & Description

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

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

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

The following program demonstrates some of these and methods:

```

Module Module1
Sub Main()

    Dim birthday As Date
    Dim bday As Integer
    Dim month As Integer
    Dim monthname As String

    ' Assign a date using standard short format.
    birthday = #7/27/1998#
    bday = Microsoft.VisualBasic.DateAndTime.Day(birthday)
    month = Microsoft.VisualBasic.DateAndTime.Month(birthday)
    monthname = Microsoft.VisualBasic.DateAndTime.MonthName(month)
    Console.WriteLine(birthday)
    Console.WriteLine(bday)
    Console.WriteLine(month)
    Console.WriteLine(monthname)
    Console.ReadKey()

End Sub
End Module

```

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

```

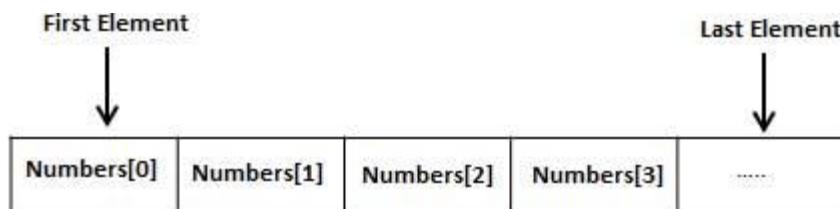
7/27/1998 12:00:00 AM
27
7
July

```

16. Arrays

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

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



Creating Arrays in VB.Net

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

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

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

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

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

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

```

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

```

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

```

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

```

Dynamic Arrays

Dynamic arrays are arrays that can be dimensioned and re-dimensioned as per 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 arrayAp1
Sub Main()
    Dim marks() As Integer
    ReDim marks(2)
    marks(0) = 85
    marks(1) = 75
    marks(2) = 90
    ReDim Preserve marks(10)
    marks(3) = 80
    marks(4) = 76
    marks(5) = 92
    marks(6) = 99
    marks(7) = 79
    marks(8) = 75
    For i = 0 To 10
        Console.WriteLine(i & vbTab & marks(i))
    Next i
    Console.ReadKey()
End Sub
End Module
```

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

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

Multi-Dimensional Arrays

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

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

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

or, a 3-dimensional array of Integer variables:

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

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

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

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

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

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

Jagged Array

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

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

The following example illustrates using a jagged array:

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

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

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

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

The Array Class

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

Properties of the Array Class

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

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

Methods of the Array Class

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

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

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

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

Example

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

```
Module arrayAp1
    Sub Main()
        Dim list As Integer() = {34, 72, 13, 44, 25, 30, 10}
        Dim temp As Integer() = list
        Dim i As Integer
        Console.Write("Original Array: ")
        For Each i In list
            Console.Write("{0} ", i)
        Next i
        Console.WriteLine()
        ' reverse the array
        Array.Reverse(temp)
        Console.Write("Reversed Array: ")
        For Each i In temp
            Console.Write("{0} ", i)
        Next i
        Console.WriteLine()
        'sort the array
        Array.Sort(list)
        Console.Write("Sorted Array: ")
        For Each i In list
            Console.Write("{0} ", i)
        Next i
        Console.WriteLine()
        Console.ReadKey()
    End Sub
End Module
```

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

Original Array: 34 72 13 44 25 30 10

Reversed Array: 10 30 25 44 13 72 34

Sorted Array: 10 13 25 30 34 44 72

17. Collections

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

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

Various Collection Classes and Their Usage

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

Class	Description and Usage
ArrayList	<p>It represents ordered collection of an object that can be indexed individually.</p> <p>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.</p>
Hashtable	<p>It uses a key to access the elements in the collection.</p> <p>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.</p>
SortedList	<p>It uses a key as well as an index to access the items in a list.</p>

	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	<p>It represents a last-in, first out collection of object.</p> <p>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.</p>
Queue	<p>It represents a first-in, first out collection of object.</p> <p>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 dequeue.</p>
BitArray	<p>It represents an array of the binary representation using the values 1 and 0.</p> <p>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.</p>

ArrayList

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

Properties and Methods of the ArrayList Class

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

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

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

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

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

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

Example

The following example demonstrates the concept:

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

```
End Sub
End Module
```

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

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

Hashtable

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

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

Properties and Methods of the Hashtable Class

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

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

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

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

Example

The following example demonstrates the concept:

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

```

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

```

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

```

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

```

SortedList

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

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

Properties and Methods of the SortedList Class

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

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

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

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

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

Example

The following example demonstrates the concept:

```

Module collections

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

```

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

```

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

```

004: Mausam Banazir Nur

005: M. Amlan

006: M. Arif

007: Ritesh Saikia

008: Nuha Ali

Stack

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

Properties and Methods of the Stack Class

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

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

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

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

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

Example

The following example demonstrates use of Stack:

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

```

st.Pop()
Console.WriteLine("Current stack: ")

For Each c In st
    Console.Write(c + " ")
Next c
Console.ReadKey()
End Sub
End Module

```

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

```

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

```

Queue

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

Properties and Methods of the Queue Class

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

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

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

S.N	Method Name & Purpose

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

Example

The following example demonstrates use of Queue:

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

```

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

```

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

```

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

```

BitArray

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

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

Properties and Methods of the BitArray Class

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

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

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

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

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

Example

The following example demonstrates the use of BitArray class:

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

```

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

```

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

```

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

```


18. Functions

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

- Functions
- Sub procedures or Subs

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

Defining a Function

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

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

Where,

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

Example

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

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

```

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

```

Function Returning a Value

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

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

The following example demonstrates using the *FindMax* function:

```

Module myfunctions

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

    Sub Main()
        Dim a As Integer = 100
        Dim b As Integer = 200
        Dim res As Integer
        res = FindMax(a, b)
        Console.WriteLine("Max value is : {0}", res)
        Console.ReadLine()
    End Sub
End Module

```

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

```
Max value is : 200
```

Recursive Function

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

```
Module myfunctions

    Function factorial(ByVal num As Integer) As Integer
        ' local variable declaration */
        Dim result As Integer
        If (num = 1) Then
            Return 1
        Else
            result = factorial(num - 1) * num
            Return result
        End If
    End Function

    Sub Main()
        'calling the factorial method
        Console.WriteLine("Factorial of 6 is : {0}", factorial(6))
        Console.WriteLine("Factorial of 7 is : {0}", factorial(7))
        Console.WriteLine("Factorial of 8 is : {0}", factorial(8))
        Console.ReadLine()
    End Sub
End Module
```

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

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

Param Arrays

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

The following example demonstrates this:

```
Module myparamfunc

    Function AddElements(ParamArray arr As Integer()) As Integer
        Dim sum As Integer = 0
        Dim i As Integer = 0
        For Each i In arr
            sum += i
        Next i
        Return sum
    End Function

    Sub Main()
        Dim sum As Integer
        sum = AddElements(512, 720, 250, 567, 889)
        Console.WriteLine("The sum is: {0}", sum)
        Console.ReadLine()
    End Sub
End Module
```

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

```
The sum is: 2938
```

Passing Arrays as Function Arguments

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

```
Module arrayParameter

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

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

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

```
Average value is: 214.4
```

19. Sub Procedures

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

Defining Sub Procedures

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

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

Where,

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

Example

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

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

```

'calling the CalculatePay Sub Procedure
CalculatePay(25, 10)

CalculatePay(40, 20)
CalculatePay(30, 27.5)
Console.ReadLine()

End Sub
End Module

```

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

```

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

```

Passing Parameters by Value

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

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

```

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

```

```

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

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

End Sub

End Module

```

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

```

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

```

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

Passing Parameters by Reference

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

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

```

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

```

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

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

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

20. Classes & Objects

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

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

Class Definition

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

```
[ <attributelist> ] [ accessmodifier ] [ Shadows ] [ MustInherit |  
NotInheritable ] [ Partial ] _  
Class name [ ( Of typelist ) ]  
    [ Inherits classname ]  
    [ Implements interfacenames ]  
    [ statements ]  
End Class
```

Where,

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

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

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

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

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

Volume of Box1 : 210

Volume of Box2 : 1560

Member Functions and Encapsulation

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

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

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

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

```

' box 1 specification

Box1.setLength(6.0)
Box1.setBreadth(7.0)
Box1.setHeight(5.0)

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

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

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

End Sub
End Module

```

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

```

Volume of Box1 : 210
Volume of Box2 : 1560

```

Constructors and Destructors

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

Following program explains the concept of constructor:

```

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

```

```

    Console.WriteLine("Object is being created")
End Sub

Public Sub setLength(ByVal len As Double)
    length = len
End Sub

Public Function getLength() As Double
    Return length
End Function

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

End Class

```

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

```

Object is being created
Length of line : 6

```

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

```

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

```

```

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

```

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

```

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

```

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

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

Following example demonstrates the concept of destructor:

```

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

```

```

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

```

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

```

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

```

Shared Members of a VB.Net Class

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

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

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

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

The following example demonstrates the use of shared members:

```

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

```

```

    num = num + 1
End Sub

Public Shared Function getNum() As Integer
    Return num
End Function

Shared Sub Main()
    Dim s As StaticVar = New StaticVar()
    s.count()
    s.count()
    s.count()

    Console.WriteLine("Value of variable num: {0}",
StaticVar.getNum())
    Console.ReadKey()
End Sub

End Class

```

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

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

Inheritance

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

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

Base & Derived Classes

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

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

```

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

```

```

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

```

Consider a base class Shape and its derived class Rectangle:

```

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

```

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

Total area: 35

Base Class Initialization

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

The following program demonstrates this:

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

'Derived class
Class Tabletop : Inherits Rectangle
    Private cost As Double
    Public Sub New(ByVal l As Double, ByVal w As Double)
        MyBase.New(l, w)
    End Sub
    Public Function GetCost() As Double
        Dim cost As Double
```

```

cost = GetArea() * 70
Return cost

End Function

Public Overrides Sub Display()
    MyBase.Display()
    Console.WriteLine("Cost: {0}", GetCost())
End Sub

'end class Tabletop

End Class

Class RectangleTester
    Shared Sub Main()
        Dim t As Tabletop = New Tabletop(4.5, 7.5)
        t.Display()
        Console.ReadKey()
    End Sub
End Class

```

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

```

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

```

VB.Net supports multiple inheritance.

21. Exception Handling

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

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

- **Try:** A Try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more Catch blocks.
- **Catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The Catch keyword indicates the catching of an exception.
- **Finally:** The Finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
- **Throw:** A program throws an exception when a problem shows up. This is done using a Throw keyword.

Syntax

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

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

End Try

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

Exception Classes in .Net Framework

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

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

The following table provides some of the predefined exception classes derived from the System.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 referencing a null object.
System.DivideByZeroException	Handles errors generated from dividing a dividend with zero.
System.InvalidCastException	Handles errors generated during typecasting.
System.OutOfMemoryException	Handles errors generated from insufficient free memory.

System.StackOverflowException	Handles errors generated from stack overflow.
-------------------------------	---

Handling Exceptions

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

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

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

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

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

Creating User-Defined Exceptions

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

```

Module exceptionProg

    Public Class TempIsZeroException : Inherits ApplicationException
        Public Sub New(ByVal message As String)
            MyBase.New(message)
        End Sub
    End Class

    Public Class Temperature
        Dim temperature As Integer = 0
        Sub showTemp()
            If (temperature = 0) Then
                Throw (New TempIsZeroException("Zero Temperature found"))
            Else
                Console.WriteLine("Temperature: {0}", temperature)
            End If
        End Sub
    End Class

    Sub Main()
        Dim temp As Temperature = New Temperature()
        Try
            temp.showTemp()
        Catch e As TempIsZeroException
            Console.WriteLine("TempIsZeroException: {0}", e.Message)
        End Try
        Console.ReadKey()
    End Sub
End Module

```

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

```
TempIsZeroException: Zero Temperature found
```

Throwing Objects

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

```
Throw [ expression ]
```

The following program demonstrates this:

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

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

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

22. File Handling

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

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

VB.Net I/O Classes

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

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

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

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

The FileStream Class

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

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

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

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

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

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

	<ul style="list-style-type: none"> • 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	<p>FileShare enumerators have the following members:</p> <ul style="list-style-type: none"> • Inheritable: It allows a file handle to pass inheritance to the child processes • None: It declines sharing of the current file • Read: It allows opening the file for reading • ReadWrite: It allows opening the file for reading and writing • Write: It allows opening the file for writing

Example

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

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

```

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

```

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

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

Advanced File Operations in VB.Net

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

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

Topic and Description

[Reading from and Writing into Text files](#)

It involves reading from and writing into text files.

The **StreamReader** and **StreamWriter** classes help to accomplish it.

[Reading from and Writing into Binary files](#)

It involves reading from and writing into binary files. The **BinaryReader** and **BinaryWriter** classes help to accomplish this.

[Manipulating the Windows file system](#)

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

Reading from and Writing to Text Files

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

The StreamReader Class

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

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

Example

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

```

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

```

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

```

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

The StreamWriter Class

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

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

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

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

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

Example

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

```
Imports System.IO
Module fileProg
    Sub Main()
        Dim names As String() = New String() {"Zara Ali", _
```

```

    "Nuha Ali", "Amir Sohel", "M Amlan"}

Dim s As String

Using sw As StreamWriter = New StreamWriter("names.txt")
    For Each s In names
        sw.WriteLine(s)
    Next s
End Using

' Read and show each line from the file.

Dim line As String

Using sr As StreamReader = New StreamReader("names.txt")
    line = sr.ReadLine()
    While (line <> Nothing)
        Console.WriteLine(line)
        line = sr.ReadLine()
    End While
End Using

Console.ReadKey()

End Sub

End Module

```

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

```

Zara Ali
Nuha Ali
Amir Sohel
M Amlan

```

Binary Files

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

The BinaryReader Class

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

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

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

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

The BinaryWriter Class

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

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

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

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

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

Example

The following example demonstrates reading and writing binary data:

```

Imports System.IO

Module fileProg

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

```

```

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

```

```

    Console.WriteLine(e.Message + "\n Cannot read from file.")

    Return

End Try

br.Close()

Console.ReadKey()

End Sub

End Module

```

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

```

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

```

Windows File System

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

The DirectoryInfo Class

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

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

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

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

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

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

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

The FileInfo Class

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

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

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

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

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

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

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

Example

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

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

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

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

23. Basic Controls

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

Properties which describe the object,

Methods cause an object to do something and

Events are what happens when an object does something.

Control Properties

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

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

```
Object. Property = Value
```

Where,

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

Property is the characteristic you want to change.

Value is the new property setting.

For example,

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

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

Control Methods

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

- If you are using a control such as one of those provided by the Toolbox, you can call any of its public methods. The requirements of such a method depend on the class being used.
- If none of the existing methods can perform your desired task, you can add a method to a class.

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

```
Public Class Form1
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
        Handles Button1.Click
        MessageBox.Show("Hello, World")
    End Sub
End Class
```

Control Events

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

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

```
Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
    'event handler code goes here
End Sub
```

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

Basic Controls

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

The following table lists some of the commonly used controls:

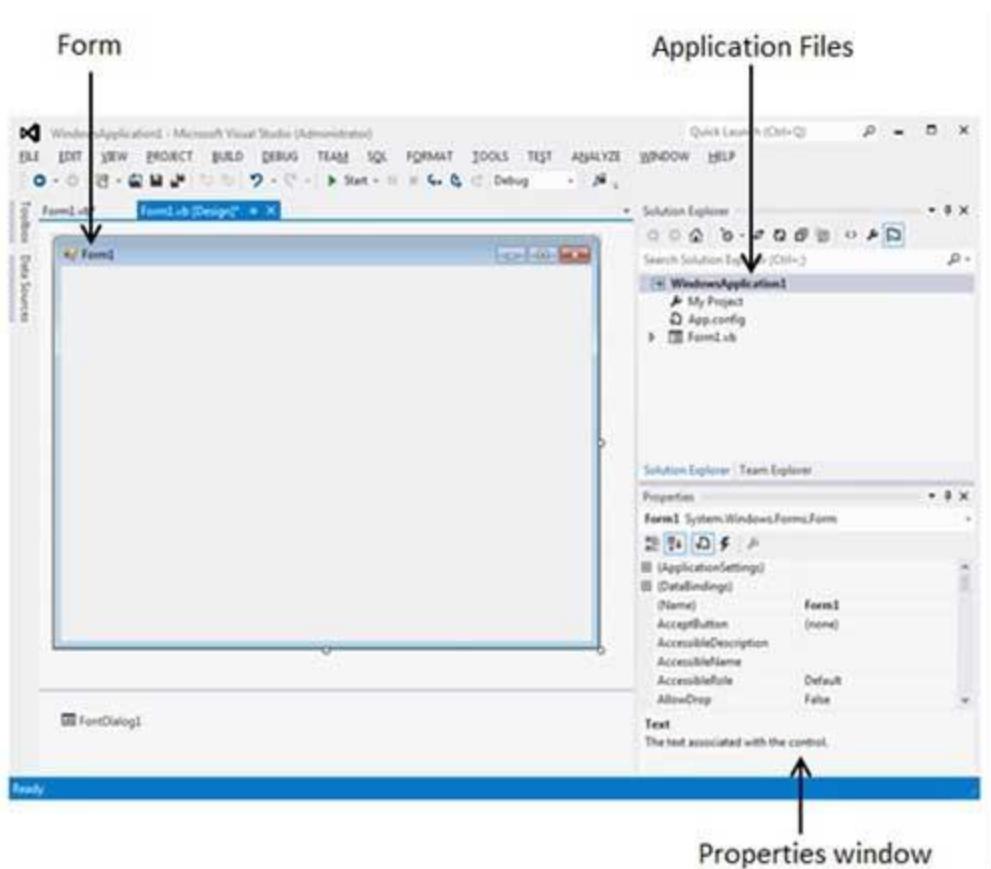
S.N.	Widget & Description
1	Forms The container for all the controls that make up the user interface.
2	TextBox It represents a Windows text box control.
3	Label It represents a standard Windows label.
4	Button It represents a Windows button control.
5	ListBox It represents a Windows control to display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

11	<u>ScrollBar</u> It Implements the basic functionality of a scroll bar control.
12	<u>DateTimePicker</u> 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	<u>TreeView</u> It displays a hierarchical collection of labeled items, each represented by a TreeNode.
14	<u>ListView</u> It represents a Windows list view control, which displays a collection of items that can be displayed using one of four different views.

Forms

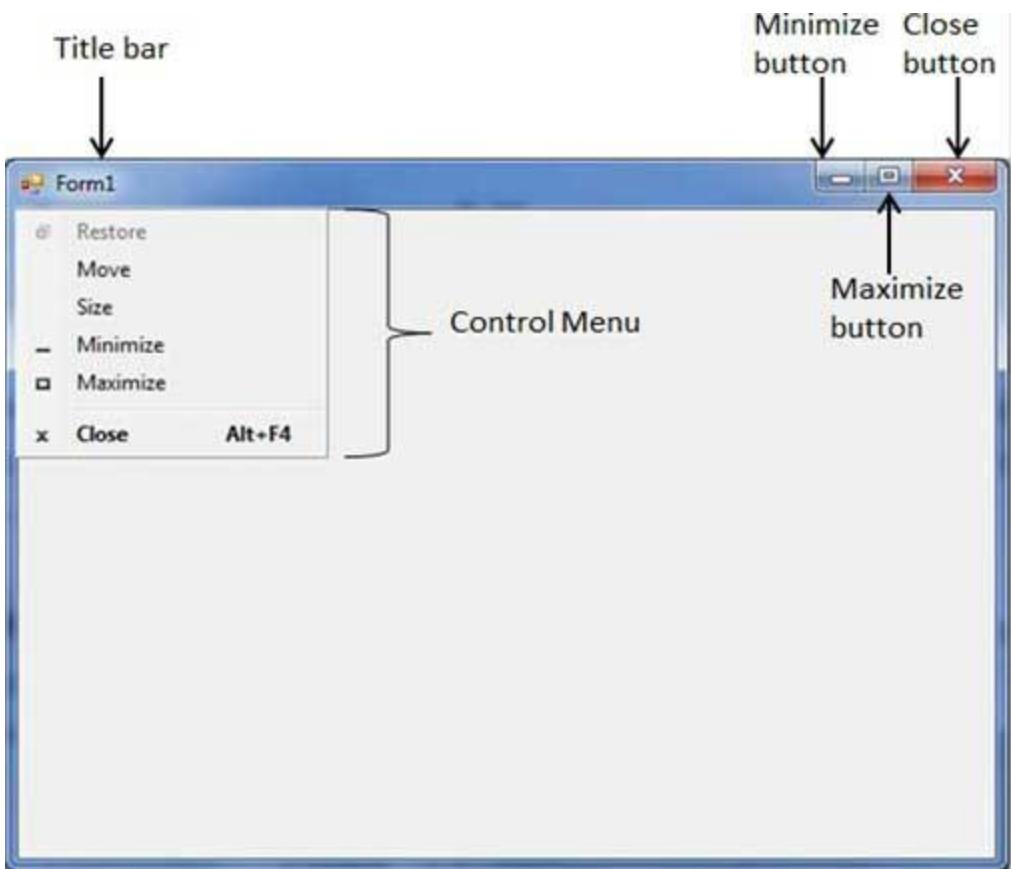
Let's start with creating a Window Forms Application by following the following steps in Microsoft Visual Studio: **File -> New Project -> Windows Forms Applications**

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



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

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



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

Form Properties

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

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

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

		<ul style="list-style-type: none"> • FixedToolWindow: A fixed window with a Close button only. It looks like the toolbar displayed by the drawing and imaging applications. • SizableToolWindow: Same as the FixedToolWindow but resizable. In addition, its caption font is smaller than the usual.
9	ControlBox	By default, this property is True and you can set it to False to hide the icon and disable the Control menu.
10	Enabled	If True, allows the form to respond to mouse and keyboard events; if False, disables form.
11	Font	This property specify font type, style, size
12	HelpButton	Determines whether a Help button should be displayed in the caption box of the form.
13	Height	This is the height of the Form in pixels.
14	MinimizeBox	By default, this property is True and you can set it to False to hide the Minimize button on the title bar.
15	MaximizeBox	By default, this property is True and you can set it to False to hide the Maximize button on the title bar.
16	MinimumSize	This specifies the minimum height and width of the window you can minimize.
17	MaximumSize	This specifies the maximum height and width of the window you maximize.
18	Name	This is the actual name of the form.

19	StartPosition	<p>This property determines the initial position of the form when it's first displayed. It will have any of the following values:</p> <ul style="list-style-type: none"> • CenterParent: The form is centered in the area of its parent form. • CenterScreen: The form is centered on the monitor. • Manual: The location and size of the form will determine its starting position. • WindowsDefaultBounds: The form is positioned at the default location and size determined by Windows. • WindowsDefaultLocation: The form is positioned at the Windows default location and has the dimensions you've set at design time.
20	Text	The text, which will appear at the title bar of the form.
21	Top, Left	These two properties set or return the coordinates of the form's top-left corner in pixels.
22	TopMost	This property is a True/False value that lets you specify whether the form will remain on top of all other forms in your application. Its default property is False.
23	Width	This is the width of the form in pixel.

Form Methods

The following are some of the commonly used methods of the Form class. You can refer to Microsoft documentation for a complete list of methods associated with forms control:

S.N.	Method Name & Description
1	Activate Activates the form and gives it focus.
2	ActivateMdiChild Activates the MDI child of a form.
3	AddOwnedForm Adds an owned form to this form.
4	BringToFront Brings the control to the front of the z-order.
5	CenterToParent Centers the position of the form within the bounds of the parent form.
6	CenterToScreen Centers the form on the current screen.
7	Close Closes the form.
8	Contains Retrieves a value indicating whether the specified control is a child of the control.
9	Focus Sets input focus to the control.
10	Hide Conceals the control from the user.
11	Refresh Forces the control to invalidate its client area and immediately redraw itself and any child controls.

12	Scale(SizeF) Scales the control and all child controls by the specified scaling factor.
13	ScaleControl Scales the location, size, padding, and margin of a control.
14	ScaleCore Performs scaling of the form.
15	Select Activates the control.
16	SendToBack Sends the control to the back of the z-order.
17	SetAutoScrollMargin Sets the size of the auto-scroll margins.
18	SetDesktopBounds Sets the bounds of the form in desktop coordinates.
19	SetDesktopLocation Sets the location of the form in desktop coordinates.
20	SetDisplayRectLocation Positions the display window to the specified value.
21	Show Displays the control to the user.
22	ShowDialog Shows the form as a modal dialog box.

Form Events

Following table lists down various important events related to a form. You can refer to Microsoft documentation for a complete list of events associated with forms control:

S.N	Event	Description

1	Activated	Occurs when the form is activated in code or by the user.
2	Click	Occurs when the form is clicked.
3	Closed	Occurs before the form is closed.
4	Closing	Occurs when the form is closing.
5	DoubleClick	Occurs when the form control is double-clicked.
6	DragDrop	Occurs when a drag-and-drop operation is completed.
7	Enter	Occurs when the form is entered.
8	GotFocus	Occurs when the form control receives focus.
9	HelpButtonClicked	Occurs when the Help button is clicked.
10	KeyDown	Occurs when a key is pressed while the form has focus.
11	KeyPress	Occurs when a key is pressed while the form has focus.
12	KeyUp	Occurs when a key is released while the form has focus.
13	Load	Occurs before a form is displayed for the first time.
14	LostFocus	Occurs when the form loses focus.

15	MouseDown	Occurs when the mouse pointer is over the form and a mouse button is pressed.
16	MouseEnter	Occurs when the mouse pointer enters the form.
17	MouseHover	Occurs when the mouse pointer rests on the form.
18	MouseLeave	Occurs when the mouse pointer leaves the form.
19	MouseMove	Occurs when the mouse pointer is moved over the form.
20	MouseUp	Occurs when the mouse pointer is over the form and a mouse button is released.
21	MouseWheel	Occurs when the mouse wheel moves while the control has focus.
22	Move	Occurs when the form is moved.
23	Resize	Occurs when the control is resized.
24	Scroll	Occurs when the user or code scrolls through the client area.
25	Shown	Occurs whenever the form is first displayed.
26	VisibleChanged	Occurs when the Visible property value changes.

Example

Following is an example, which shows how we create two buttons at the time of form load event and different properties are being set at the same time.

Because **Form1** is being referenced within its own event handler, so it will be written as **Me** instead of using its name, but if we access the same form inside any other control's event handler, then it will be accessed using its name **Form1**.

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
        ' Create two buttons to use as the accept and cancel buttons.
        Dim button1 As New Button()
        Dim button2 As New Button()
        ' Set the text of button1 to "OK".
        button1.Text = "OK"
        ' Set the position of the button on the form.
        button1.Location = New Point(10, 10)
        ' Set the text of button2 to "Cancel".
        button2.Text = "Cancel"
        ' Set the position of the button based on the location of button1.
        button2.Location = _
            New Point(button1.Left, button1.Height + button1.Top + 10)
        ' Set the caption bar text of the form.
        Me.Text = "tutorialspoint.com"
        ' Display a help button on the form.
        Me.HelpButton = True
        ' Define the border style of the form to a dialog box.
        Me.FormBorderStyle = FormBorderStyle.FixedDialog
        ' Set the MaximizeBox to false to remove the maximize box.
        Me.MaximizeBox = False
        ' Set the MinimizeBox to false to remove the minimize box.
        Me.MinimizeBox = False
        ' Set the accept button of the form to button1.
        Me.AcceptButton = button1
        ' Set the cancel button of the form to button2.
        Me.CancelButton = button2
    End Sub
End Class

```

```

' Set the start position of the form to the center of the screen.
Me.StartPosition = FormStartPosition.CenterScreen

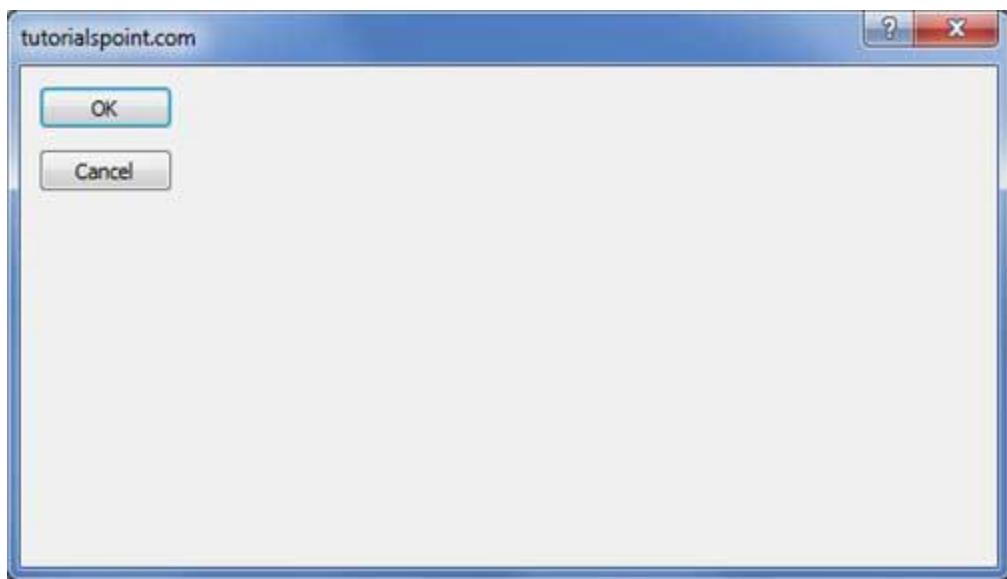
' Set window width and height
Me.Height = 300
Me.Width = 560
' Add button1 to the form.
Me.Controls.Add(button1)
' Add button2 to the form.
Me.Controls.Add(button2)

End Sub

End Class

```

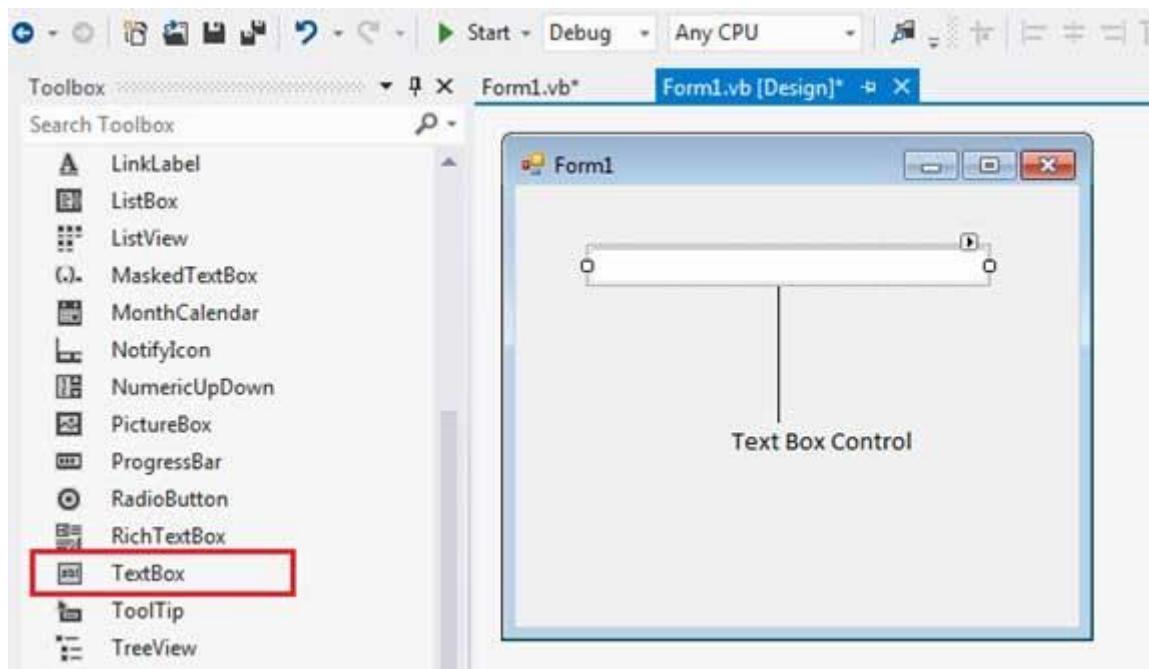
When the above code is executed and run using **Start** button available at the Microsoft Visual Studio tool bar, it will show the following window:



TextBox Control

Text box controls allow entering text on a form at runtime. By default, it takes a single line of text, however, you can make it accept multiple texts and even add scroll bars to it.

Let's create a text box by dragging a Text Box control from the Toolbox and dropping it on the form.



The Properties of the TextBox Control

The following are some of the commonly used properties of the TextBox control:

S.N	Property	Description
1	AcceptsReturn	Gets or sets a value indicating whether pressing ENTER in a multiline TextBox control creates a new line of text in the control or activates the default button for the form.
2	AutoCompleteCustomSource	Gets or sets a custom System.Collections.Specialized.StringCollection to use when the AutoCompleteSourceproperty is set to CustomSource.
3	AutoCompleteMode	Gets or sets an option that controls how automatic completion works for the TextBox.

4	AutoCompleteSource	Gets or sets a value specifying the source of complete strings used for automatic completion.
5	CharacterCasing	Gets or sets whether the TextBox control modifies the case of characters as they are typed.
6	Font	Gets or sets the font of the text displayed by the control.
7	FontHeight	Gets or sets the height of the font of the control.
8	ForeColor	Gets or sets the foreground color of the control.
9	Lines	Gets or sets the lines of text in a text box control.
10	Multiline	Gets or sets a value indicating whether this is a multiline TextBox control.
11	PasswordChar	Gets or sets the character used to mask characters of a password in a single-line TextBox control.
12	ReadOnly	Gets or sets a value indicating whether text in the text box is read-only.
13	ScrollBars	Gets or sets which scroll bars should appear in a multiline TextBox control. This property has values: <ul style="list-style-type: none"> • None • Horizontal • Vertical

		<ul style="list-style-type: none"> Both
14	TabIndex	Gets or sets the tab order of the control within its container.
15	Text	Gets or sets the current text in the TextBox.
16	 TextAlign	Gets or sets how text is aligned in a TextBox control. This property has values: <ul style="list-style-type: none"> Left Right Center
17	TextLength	Gets the length of text in the control.
18	WordWrap	Indicates whether a multiline text box control automatically wraps words to the beginning of the next line when necessary.

The Methods of the TextBox Control

The following are some of the commonly used methods of the TextBox control:

S.N	Method Name & Description
1	AppendText Appends text to the current text of a text box.
2	Clear Clears all text from the text box control.
3	Copy Copies the current selection in the text box to the Clipboard .
4	Cut Moves the current selection in the text box to the Clipboard .
5	Paste

	Replaces the current selection in the text box with the contents of the Clipboard .
6	Paste(String) Sets the selected text to the specified text without clearing the undo buffer.
7	ResetText Resets the Text property to its default value.
8	ToString Returns a string that represents the TextBoxBase control.
9	Undo Undoes the last edit operation in the text box.

Events of the TextBox Control

The following are some of the commonly used events of the Text control:

S.N	Event	Description
1	Click	Occurs when the control is clicked.
2	DoubleClick	Occurs when the control is double-clicked.
3	 TextAlignChanged	Occurs when the TextAlign property value changes.

Example

In this example, we create three text boxes and use the Click event of a button to display the entered text using a message box. Take the following steps:

- Drag and drop three Label controls and three TextBox controls on the form.
- Change the texts on the labels to: Name, Organization and Comments, respectively.
- Change the names of the text boxes to txtName, txtOrg and txtComment, respectively.

- Drag and drop a button control on the form. Set its name to btnMessage and its text property to 'Send Message'.
- Click the button to add the Click event in the code window and add the following code.

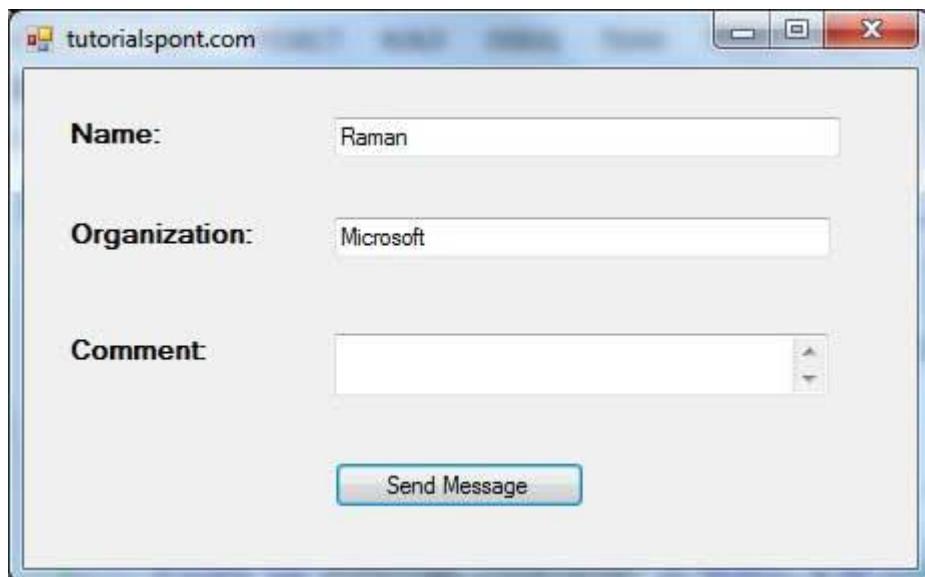
```

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 = "tutorialspong.com"
    End Sub

    Private Sub btnMessage_Click(sender As Object, e As EventArgs) _
        Handles btnMessage.Click
        MessageBox.Show("Thank you " + txtName.Text + " from " +
        txtOrg.Text)
    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 the following window:



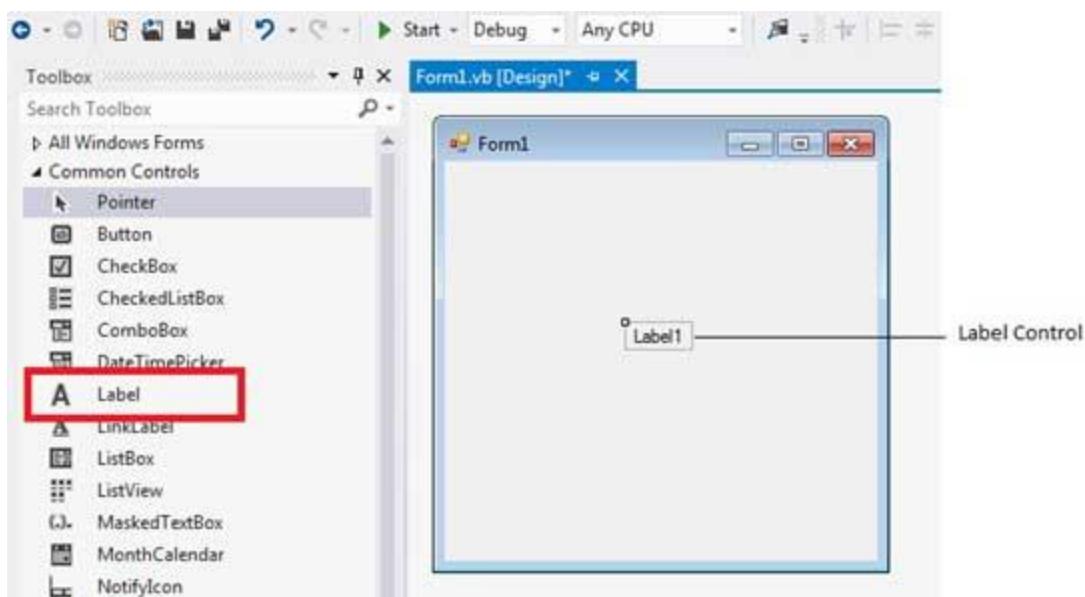
Clicking the Send Message button would show the following message box:



Label Control

The Label control represents a standard Windows label. It is generally used to display some informative text on the GUI which is not changed during runtime.

Let's create a label by dragging a Label control from the Toolbox and dropping it on the form.



Properties of the Label Control

The following are some of the commonly used properties of the Label control:

S.N	Property	Description
1	Autosize	Gets or sets a value specifying if the control should be automatically resized to display all its contents.

2	BorderStyle	Gets or sets the border style for the control.
3	FlatStyle	Gets or sets the flat style appearance of the Label control
4	Font	Gets or sets the font of the text displayed by the control.
5	FontHeight	Gets or sets the height of the font of the control.
6	ForeColor	Gets or sets the foreground color of the control.
7	PreferredSize	Gets the preferred height of the control.
8	PreferredWidth	Gets the preferred width of the control.
9	TabStop	Gets or sets a value indicating whether the user can tab to the Label. This property is not used by this class.
10	Text	Gets or sets the text associated with this control.
11	.TextAlign	Gets or sets the alignment of text in the label.

Methods of the Label Control

The following are some of the commonly used methods of the Label control:

S.N	Method Name & Description
1	GetPreferredSize Retrieves the size of a rectangular area into which a control can be fitted.
2	Refresh

	Forces the control to invalidate its client area and immediately redraw itself and any child controls.
3	Select Activates the control.
4	Show Displays the control to the user.
5	ToString Returns a String that contains the name of the control.

Events of the Label Control

The following are some of the commonly used events of the Label control:

S.N	Event	Description
1	AutoSizeChanged	Occurs when the value of the AutoSize property changes.
2	Click	Occurs when the control is clicked.
3	DoubleClick	Occurs when the control is double-clicked.
4	GotFocus	Occurs when the control receives focus.
5	Leave	Occurs when the input focus leaves the control.
6	LostFocus	Occurs when the control loses focus.
7	TabIndexChanged	Occurs when the TabIndex property value changes.
8	TabStopChanged	Occurs when the TabStop property changes.

9	TextChanged	Occurs when the Text property value changes.
---	--------------------	--

Refer the Microsoft documentation for a detailed list of properties, methods and events of the Label control.

Example

Following is an example, which shows how we can create two labels. Let us create the first label from the designer view tab and set its properties from the properties window. We will use the Click and the DoubleClick events of the label to move the first label and change its text and create the second label and add it to the form, respectively.

Take the following steps:

1. Drag and drop a Label control on the form.
2. Set the Text property to provide the caption "This is a Label Control".
3. Set the Font property from the properties window.
4. Click the label to add the Click event in the code window and add the following codes.

```
Public Class Form1

    Private Sub Form1_Load(sender As Object, e As EventArgs) _
        Handles MyBase.Load
        ' Create two buttons to use as the accept and cancel buttons.
        ' Set window width and height
        Me.Height = 300
        Me.Width = 560

        ' Set the caption bar text of the form.
        Me.Text = "tutorialspoint.com"
        ' Display a help button on the form.
        Me.HelpButton = True
    End Sub

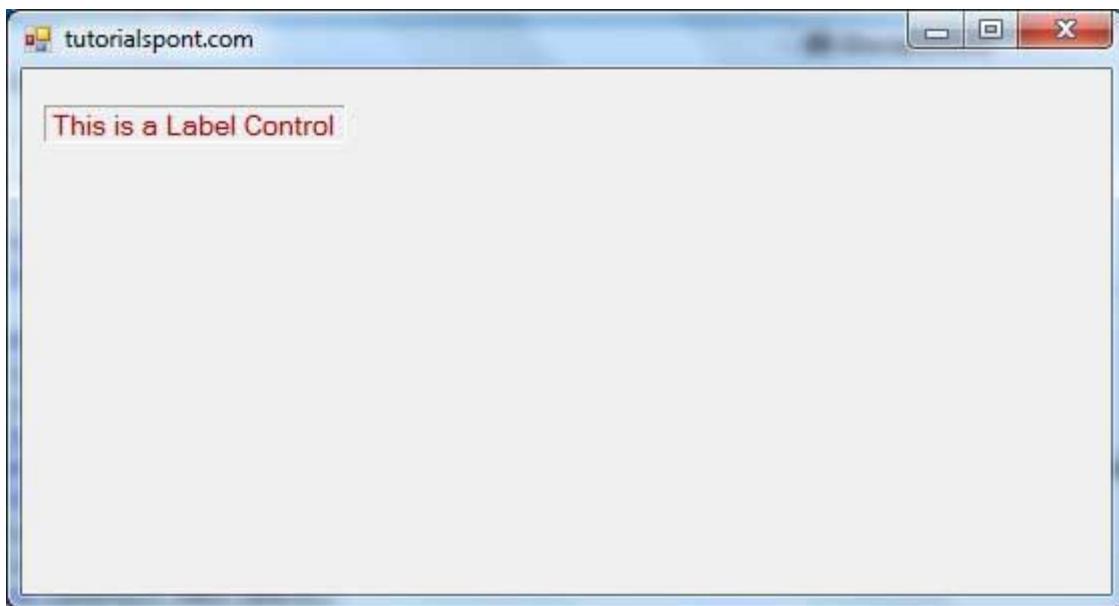
    Private Sub Label1_Click(sender As Object, e As EventArgs) _
        Handles Label1.Click
```

```
Label1.Location = New Point(50, 50)
Label1.Text = "You have just moved the label"
End Sub

Private Sub Label1_DoubleClick(sender As Object, e As EventArgs)
    Handles Label1.DoubleClick
    Dim Label2 As New Label
    Label2.Text = "New Label"
    Label2.Location = New Point(Label1.Left, Label1.Height + _
    Label1.Top + 25)
    Me.Controls.Add(Label2)
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 the following window:



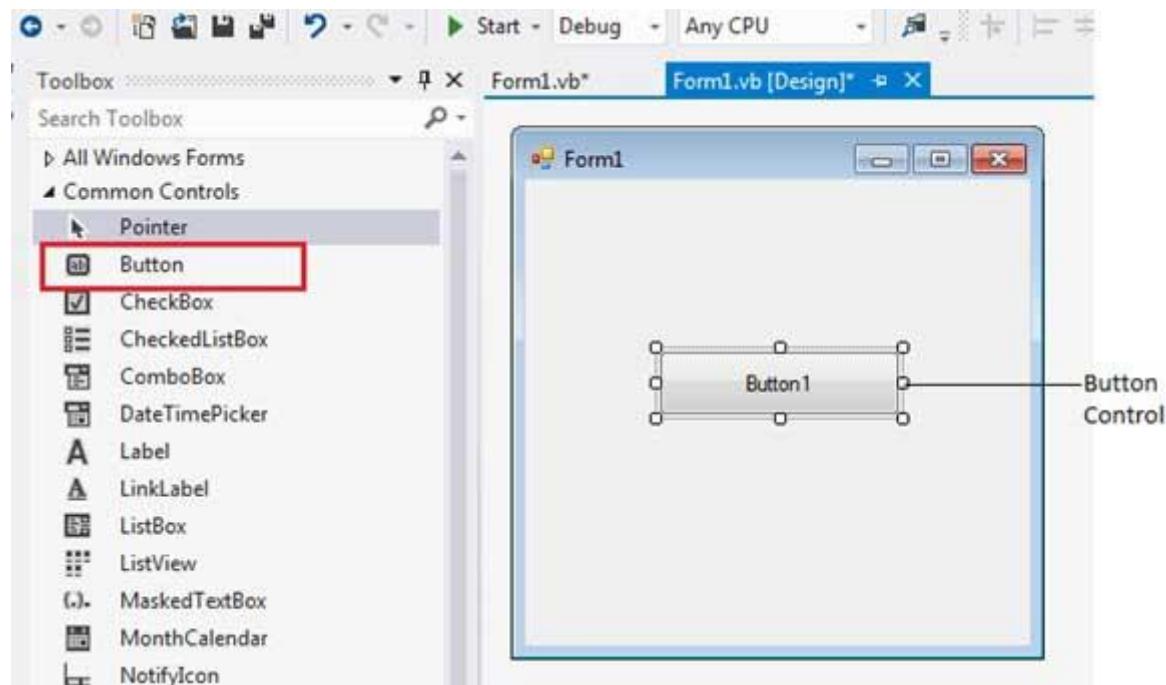
Clicking and double clicking the label would produce the following effect:



Button Control

The Button control represents a standard Windows button. It is generally used to generate a Click event by providing a handler for the Click event.

Let's create a label by dragging a Button control from the Toolbox and dropping it on the form.



Properties of the Button Control

The following are some of the commonly used properties of the Button control:

S.N	Property	Description
1	AutoSizeMode	Gets or sets the mode by which the Button automatically resizes itself.
2	BackColor	Gets or sets the background color of the control.
3	BackgroundImage	Gets or sets the background image displayed in the control.
4	DialogResult	Gets or sets a value that is returned to the parent form when the button is clicked. This is used while creating dialog boxes.
5	ForeColor	Gets or sets the foreground color of the control.
6	Image	Gets or sets the image that is displayed on a button control.
7	Location	Gets or sets the coordinates of the upper-left corner of the control relative to the upper-left corner of its container.
8	TabIndex	Gets or sets the tab order of the control within its container.
9	Text	Gets or sets the text associated with this control.

Methods of the Button Control

The following are some of the commonly used methods of the Button control:

S.N	Method Name & Description

1	GetPreferredSize Retrieves the size of a rectangular area into which a control can be fitted.
2	NotifyDefault Notifies the Button whether it is the default button so that it can adjust its appearance accordingly.
3	Select Activates the control.
4	ToString Returns a String containing the name of the Component, if any. This method should not be overridden.

Events of the Button Control

The following are some of the commonly used events of the Button control:

S.N	Event	Description
1	Click	Occurs when the control is clicked.
2	DoubleClick	Occurs when the user double-clicks the Button control.
3	GotFocus	Occurs when the control receives focus.
4	TabIndexChanged	Occurs when the TabIndex property value changes.
5	TextChanged	Occurs when the Text property value changes.
6	Validated	Occurs when the control is finished validating.

Consult Microsoft documentation for detailed list of properties, methods and events of the Button control.

Example

In the following example, we create three buttons. In this example, let us:

220

- Set captions for the buttons
- Set some image for the button
- Handle the click events of each buttons

Take following steps:

- Drag and drop a Label control on the form.
- Set the Text property to provide the caption "Tutorials Point".
- Drag and drop three buttons on the form.
- Using the properties window, change the Name properties of the buttons to btnMoto, btnLogo and btnExit respectively.
- Using the properties window, change the Text properties of the buttons to Show Moto, Show Logo and Exit respectively.
- Drag and Drop another button, using the properties window, set its Image property and name it btnImage.

At this stage, the form looks like:

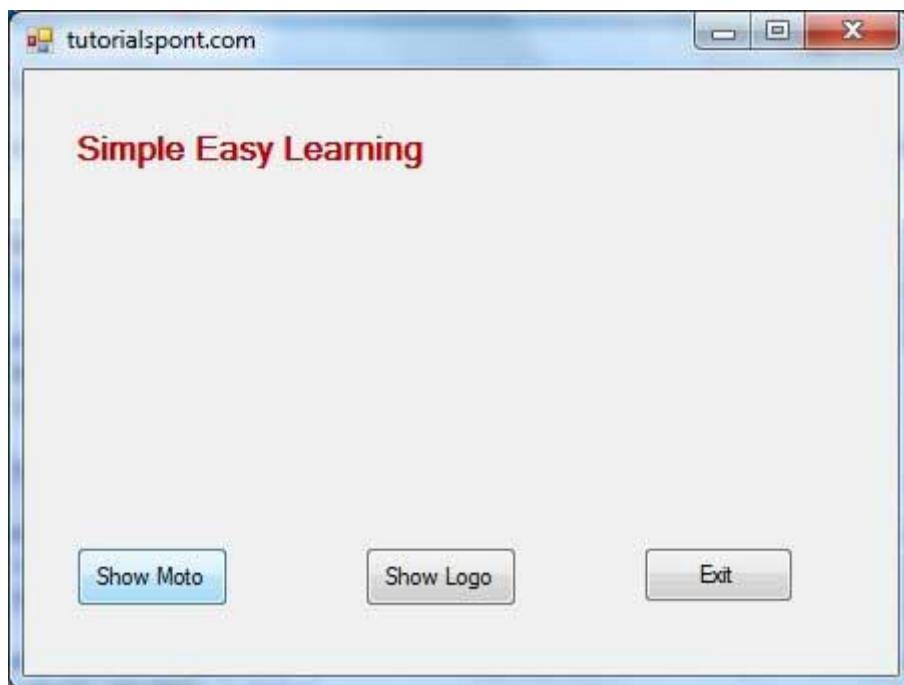


Click the form and add following code in the code editor:

```
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"  
btnImage.Visible = False  
End Sub  
  
Private Sub btnMoto_Click(sender As Object, e As EventArgs) Handles  
btnMoto.Click  
    btnImage.Visible = False  
    Label1.Text = "Simple Easy Learning"  
End Sub  
  
Private Sub btnExit_Click(sender As Object, e As EventArgs) Handles  
btnExit.Click  
    Application.Exit()  
End Sub  
  
Private Sub btnLogo_Click(sender As Object, e As EventArgs) Handles  
btnLogo.Click  
    Label1.Visible = False  
    btnImage.Visible = True  
End Sub  
End Class
```

Clicking the first button, displays:



Clicking the second button displays:

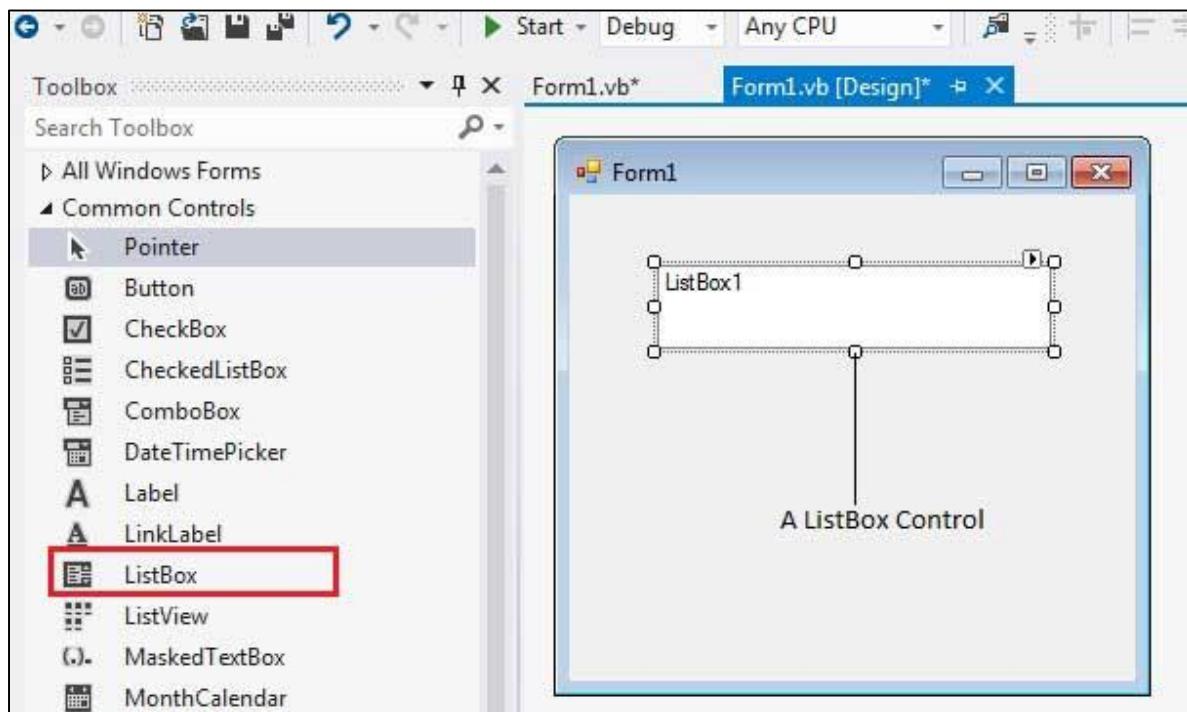


Clicking the third button, exits the application.

ListBox Control

The **ListBox** represents a Windows control to display a list of items to a user. A user can select an item from the list. It allows the programmer to add items at design time by using the properties window or at the runtime.

Let's create a list box by dragging a **ListBox** control from the Toolbox and dropping it on the form.



You can populate the list box items either from the properties window or at runtime. To add items to a ListBox, select the ListBox control and get to the properties window, for the properties of this control. Click the ellipses (...) button next to the Items property. This opens the String Collection Editor dialog box, where you can enter the values one at a line.

Properties of the ListBox Control

The following are some of the commonly used properties of the ListBox control:

S.N	Property	Description
1	AllowSelection	Gets a value indicating whether the ListBox currently enables selection of list items.
2	BorderStyle	Gets or sets the type of border drawn around the list box.
3	ColumnWidth	Gets or sets the width of columns in a multicolumn list box.
4	HorizontalExtent	Gets or sets the horizontal scrolling area of a list box.
5	HorizontalScrollBar	Gets or sets the value indicating whether a horizontal scrollbar is displayed in the list box.
6	ItemHeight	Gets or sets the height of an item in the list box.
7	Items	Gets the items of the list box.
8	MultiColumn	Gets or sets a value indicating whether the list box supports multiple columns.
9	ScrollAlwaysVisible	Gets or sets a value indicating whether the vertical scroll bar is shown at all times.

10	SelectedIndex	Gets or sets the zero-based index of the currently selected item in a list box.
11	SelectedIndices	Gets a collection that contains the zero-based indexes of all currently selected items in the list box.
12	SelectedItem	Gets or sets the currently selected item in the list box.
13	SelectedItems	Gets a collection containing the currently selected items in the list box.
14	SelectedValue	Gets or sets the value of the member property specified by the ValueMember property.
15	SelectionMode	Gets or sets the method in which items are selected in the list box. This property has values: <ul style="list-style-type: none"> • None • One • MultiSimple • MultiExtended
16	Sorted	Gets or sets a value indicating whether the items in the list box are sorted alphabetically.
17	Text	Gets or searches for the text of the currently selected item in the list box.
18	TopIndex	Gets or sets the index of the first visible item of a list box.

Methods of the ListBox Control

The following are some of the commonly used methods of the ListBox control:

S.N	Method Name & Description
1	BeginUpdate Prevents the control from drawing until the EndUpdate method is called, while items are added to the ListBox one at a time.
2	ClearSelected Unselects all items in the ListBox.
3	EndUpdate Resumes drawing of a list box after it was turned off by the BeginUpdate method.
4	FindString Finds the first item in the ListBox that starts with the string specified as an argument.
5	FindStringExact Finds the first item in the ListBox that exactly matches the specified string.
6	GetSelected Returns a value indicating whether the specified item is selected.
7	SetSelected Selects or clears the selection for the specified item in a ListBox.
8	OnSelectedIndexChanged Raises the SelectedIndexChanged event.
8	OnSelectedValueChanged Raises the SelectedValueChanged event.

Events of the ListBox Control

The following are some of the commonly used events of the ListBox control:

S.N	Event	Description
1	Click	Occurs when a list box is selected.

2	SelectedIndexChanged	Occurs when the SelectedIndex property of a list box is changed.
---	----------------------	--

Consult Microsoft documentation for detailed list of properties, methods, and events of the ListBox control.

Example 1

In the following example, let us add a list box at design time and add items on it at runtime.

Take the following steps:

Drag and drop two labels, a button and a ListBox control on the form.

Set the Text property of the first label to provide the caption "Choose your favourite destination for higher studies".

Set the Text property of the second label to provide the caption "Destination".

The text on this label will change at runtime when the user selects an item on the list.

Click the listbox and the button controls to add the following codes 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"
        ListBox1.Items.Add("Canada")
        ListBox1.Items.Add("USA")
        ListBox1.Items.Add("UK")
        ListBox1.Items.Add("Japan")
        ListBox1.Items.Add("Russia")
        ListBox1.Items.Add("China")
        ListBox1.Items.Add("India")
    End Sub

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        MsgBox("You have selected " + ListBox1.SelectedItem.ToString())
    End Sub

```

```

End Sub

Private Sub ListBox1_SelectedIndexChanged(sender As Object, e As
EventArgs)

    Handles ListBox1.SelectedIndexChanged
    Label2.Text = ListBox1.SelectedItem.ToString()
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 the following window:



When the user chooses a destination, the text in the second label changes:



Clicking the Select button displays a message box with the user's choice:



Example 2

In this example, we will fill up a list box with items, retrieve the total number of items in the list box, sort the list box, remove some items and clear the entire list box.

Design the Form:



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 = "tutorialspoint.com"
        ' creating multi-column and multiselect list box
        ListBox1.MultiColumn = True
        ListBox1.SelectionMode = SelectionMode.MultiExtended
    End Sub
    'populates the list
    Private Sub Button1_Click_1(sender As Object, e As EventArgs) _
        Handles Button1.Click
        ListBox1.Items.Add("Safety")
        ListBox1.Items.Add("Security")
        ListBox1.Items.Add("Governance")
        ListBox1.Items.Add("Good Music")
        ListBox1.Items.Add("Good Movies")
        ListBox1.Items.Add("Good Books")
        ListBox1.Items.Add("Education")
        ListBox1.Items.Add("Roads")
        ListBox1.Items.Add("Health")
    End Sub

```

```

    ListBox1.Items.Add("Food for all")
    ListBox1.Items.Add("Shelter for all")
    ListBox1.Items.Add("Industrialisation")
    ListBox1.Items.Add("Peace")
    ListBox1.Items.Add("Liberty")
    ListBox1.Items.Add("Freedom of Speech")

End Sub

'sorting the list

Private Sub Button2_Click(sender As Object, e As EventArgs) _
    Handles Button2.Click

    ListBox1.Sorted = True

End Sub

'clears the list

Private Sub Button3_Click(sender As Object, e As EventArgs) _
    Handles Button3.Click

    ListBox1.Items.Clear()

End Sub

'removing the selected item

Private Sub Button4_Click(sender As Object, e As EventArgs) _
    Handles Button4.Click

    ListBox1.Items.Remove(ListBox1.SelectedItem.ToString())

End Sub

'counting the numer of items

Private Sub Button5_Click(sender As Object, e As EventArgs) _
    Handles Button5.Click

    Label1.Text = ListBox1.Items.Count

End Sub

'displaying the selected item on the third label

Private Sub ListBox1_SelectedIndexChanged(sender As Object, e As EventArgs) _
    Handles ListBox1.SelectedIndexChanged

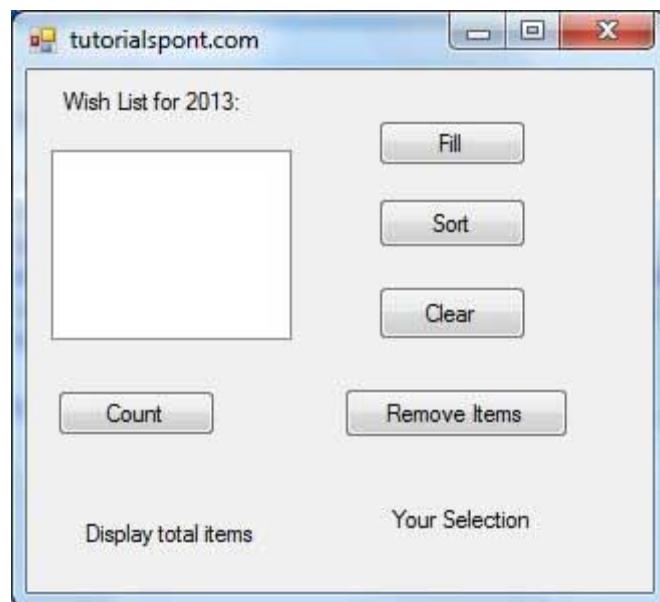
    Label3.Text = ListBox1.SelectedItem.ToString()

End Sub

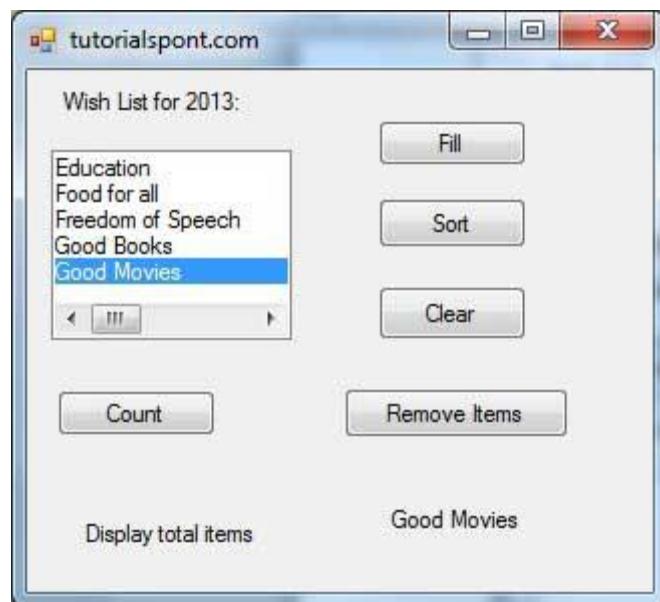
End Class

```

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



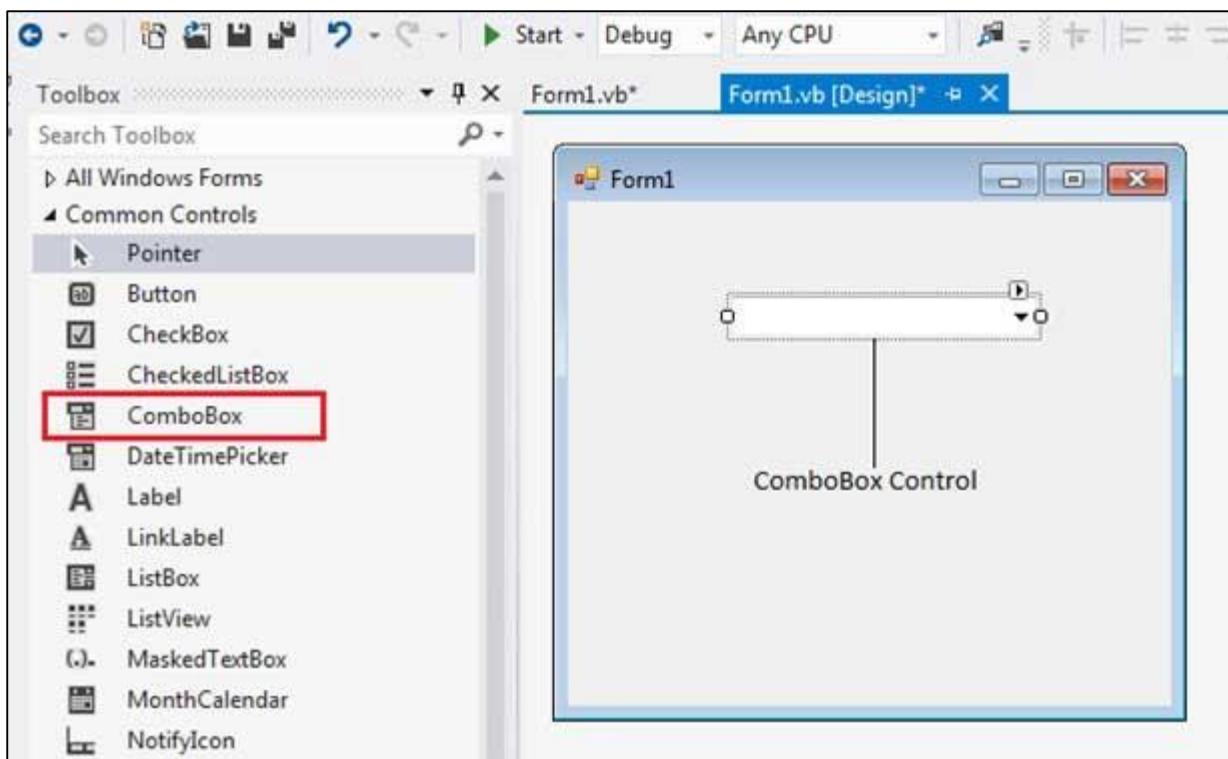
Fill the list and check workings of other buttons:



ComboBox Control

The ComboBox control is used to display a drop-down list of various items. It is a combination of a text box in which the user enters an item and a drop-down list from which the user selects an item.

Let's create a combo box by dragging a ComboBox control from the Toolbox and dropping it on the form.



You can populate the list box items either from the properties window or at runtime. To add items to a ListBox, select the ListBox control and go to the properties window for the properties of this control. Click the ellipses (...) button next to the Items property. This opens the String Collection Editor dialog box, where you can enter the values one at a line.

Properties of the ComboBox Control

The following are some of the commonly used properties of the ComboBox control:

S.N	Property	Description
1	AllowSelection	Gets a value indicating whether the list enables selection of list items.
2	AutoCompleteCustomSource	Gets or sets a custom System.Collections.Specialized.StringCollection to use when the AutoCompleteSourceproperty is set to CustomSource.

3	AutoCompleteMode	Gets or sets an option that controls how automatic completion works for the ComboBox.
4	AutoCompleteSource	Gets or sets a value specifying the source of complete strings used for automatic completion.
5	DataBindings	Gets the data bindings for the control.
6	DataManager	Gets the CurrencyManager associated with this control.
7	DataSource	Gets or sets the data source for this ComboBox.
8	DropDownHeight	Gets or sets the height in pixels of the drop-down portion of the ComboBox.
9	DropDownStyle	Gets or sets a value specifying the style of the combo box.
10	DropDownWidth	Gets or sets the width of the of the drop-down portion of a combo box.
11	DroppedDown	Gets or sets a value indicating whether the combo box is displaying its drop-down portion.
12	FlatStyle	Gets or sets the appearance of the ComboBox.
13	ItemHeight	Gets or sets the height of an item in the combo box.

14	Items	Gets an object representing the collection of the items contained in this ComboBox.
15	MaxDropDownItems	Gets or sets the maximum number of items to be displayed in the drop-down part of the combo box.
16	MaxLength	Gets or sets the maximum number of characters a user can enter in the editable area of the combo box.
17	SelectedIndex	Gets or sets the index specifying the currently selected item.
18	SelectedItem	Gets or sets currently selected item in the ComboBox.
19	SelectedText	Gets or sets the text that is selected in the editable portion of a ComboBox.
20	SelectedValue	Gets or sets the value of the member property specified by the ValueMember property.
21	SelectionLength	Gets or sets the number of characters selected in the editable portion of the combo box.
22	SelectionStart	Gets or sets the starting index of text selected in the combo box.
23	Sorted	Gets or sets a value indicating whether the items in the combo box are sorted.

24	Text	Gets or sets the text associated with this control.
----	-------------	---

Methods of the ComboBox Control

The following are some of the commonly used methods of the ComboBox control:

S.N	Method Name & Description
1	BeginUpdate Prevents the control from drawing until the EndUpdate method is called, while items are added to the combo box one at a time.
2	EndUpdate Resumes drawing of a combo box, after it was turned off by the BeginUpdate method.
3	FindString Finds the first item in the combo box that starts with the string specified as an argument.
4	FindStringExact Finds the first item in the combo box that exactly matches the specified string.
5	SelectAll Selects all the text in the editable area of the combo box.

Events of the ComboBox Control

The following are some of the commonly used events of the ComboBox control:

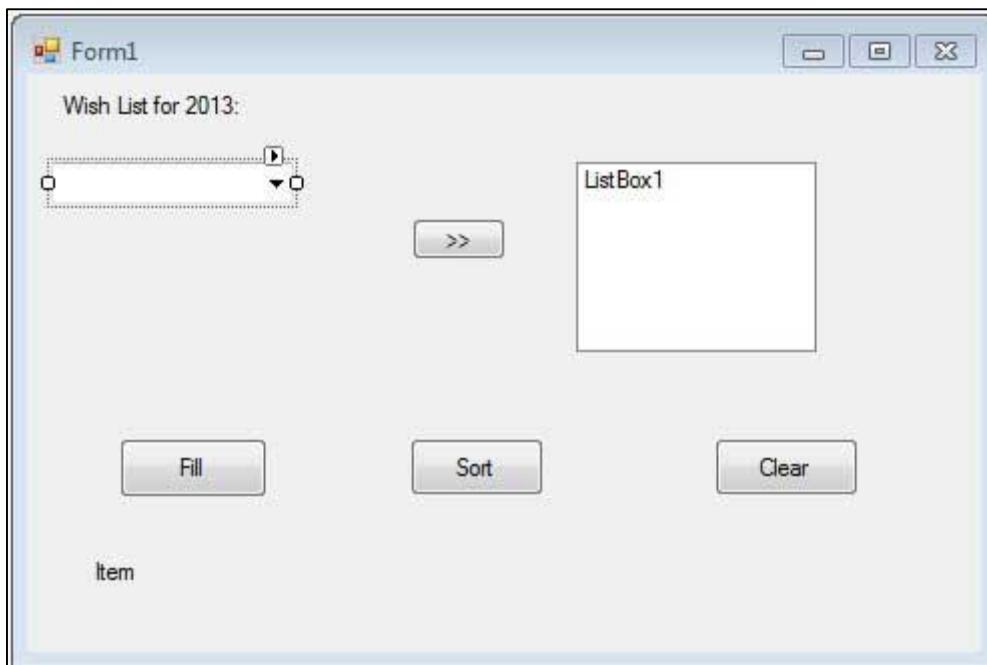
S.N	Event	Description
1	DropDown	Occurs when the drop-down portion of a combo box is displayed.
2	DropDownClosed	Occurs when the drop-down portion of a combo box is no longer visible.
3	DropDownStyleChanged	Occurs when the DropDownStyle property of the ComboBox has changed.
4	SelectedIndexChanged	Occurs when the SelectedIndex property of a ComboBox control has changed.
5	SelectionChangeCommitted	Occurs when the selected item has changed and the change appears in the combo box.

Example

In this example, let us fill a combo box with various items, get the selected items in the combo box and show them in a list box and sort the items.

Drag and drop a combo box to store the items, a list box to display the selected items, four button controls to add to the list box with selected items, to fill the combo box, to sort the items and to clear the combo box list, respectively.

Add a label control that would display the selected item.



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 = "tutorialspoint.com"
    End Sub
    'sends the selected items to the list box
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        If ComboBox1.SelectedIndex > -1 Then
            Dim sindex As Integer
            sindex = ComboBox1.SelectedIndex
            Dim sitem As Object
            sitem = ComboBox1.SelectedItem
            ListBox1.Items.Add(sitem)
        End If
    End Sub
    'populates the list
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click

```

```

ComboBox1.Items.Clear()
ComboBox1.Items.Add("Safety")

ComboBox1.Items.Add("Security")
ComboBox1.Items.Add("Governance")
ComboBox1.Items.Add("Good Music")
ComboBox1.Items.Add("Good Movies")
ComboBox1.Items.Add("Good Books")
ComboBox1.Items.Add("Education")
ComboBox1.Items.Add("Roads")
ComboBox1.Items.Add("Health")
ComboBox1.Items.Add("Food for all")
ComboBox1.Items.Add("Shelter for all")
ComboBox1.Items.Add("Industrialisation")
ComboBox1.Items.Add("Peace")
ComboBox1.Items.Add("Liberty")
ComboBox1.Items.Add("Freedom of Speech")
ComboBox1.Text = "Select from..."

End Sub

'sorting the list
Private Sub Button3_Click(sender As Object, e As EventArgs)
    ComboBox1.Sorted = True
End Sub

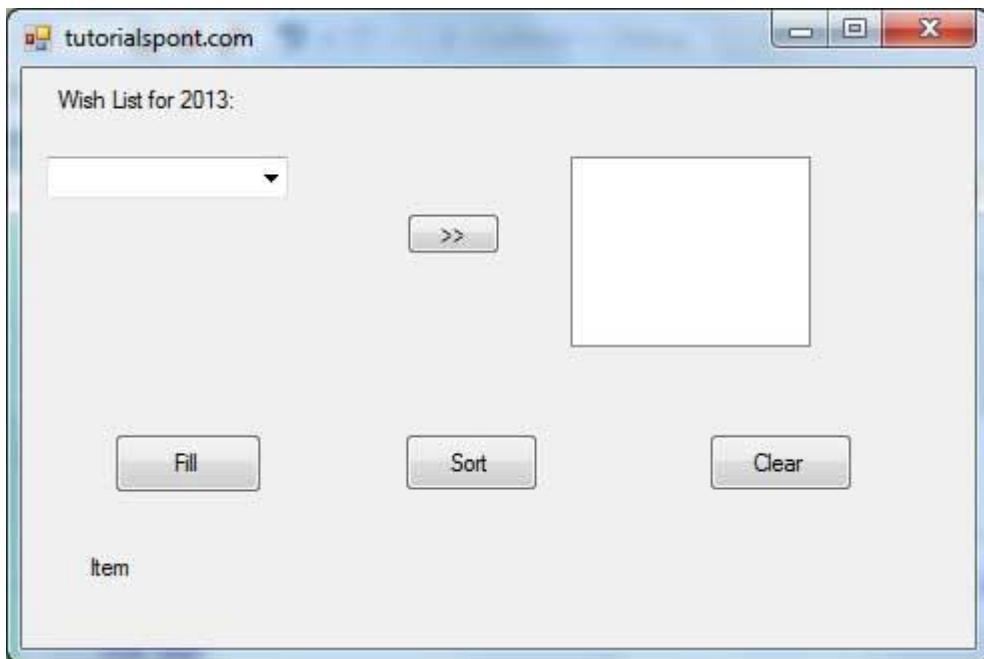
'clears the list
Private Sub Button4_Click(sender As Object, e As EventArgs)
    ComboBox1.Items.Clear()
End Sub

'displaying the selected item on the label
Private Sub ComboBox1_SelectedIndexChanged(sender As Object, e As EventArgs) _
    Handles ListBox1.SelectedIndexChanged
    Label1.Text = ComboBox1.SelectedItem.ToString()
End Sub

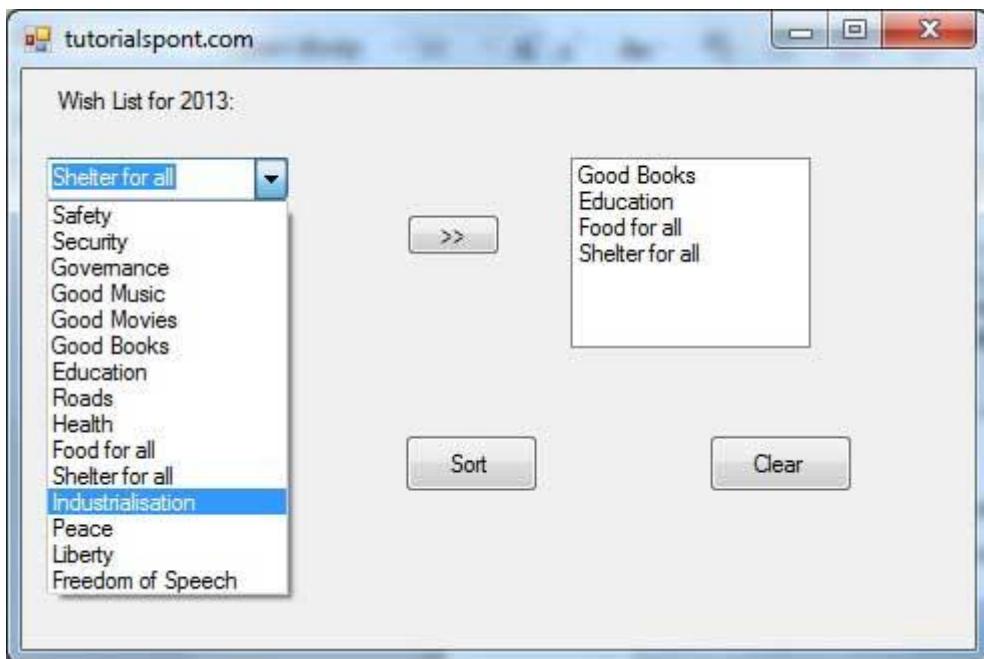
End Class

```

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



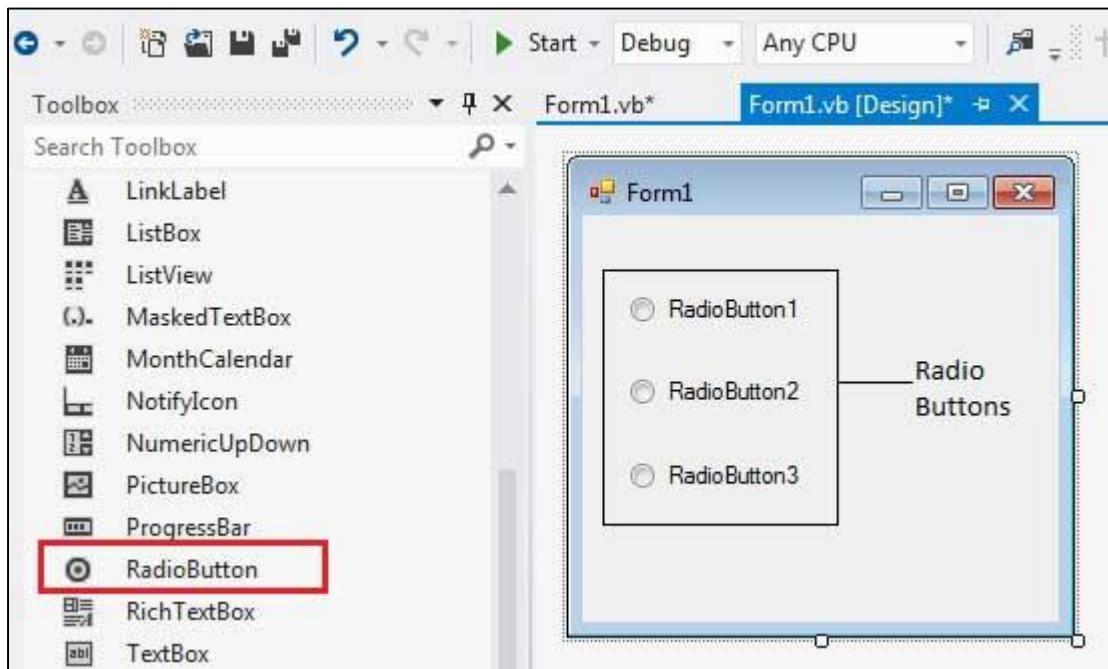
Click on various buttons to check the actions performed by each:



RadioButton Control

The RadioButton control is used to provide a set of mutually exclusive options. The user can select one radio button in a group. If you need to place more than one group of radio buttons in the same form, you should place them in different container controls like a GroupBox control.

Let's create three radio buttons by dragging RadioButton controls from the Toolbox and dropping on the form.



The *Checked* property of the radio button is used to set the state of a radio button. You can display text, image or both on radio button control. You can also change the appearance of the radio button control by using the *Appearance* property.

Properties of the RadioButton Control

The following are some of the commonly used properties of the RadioButton control:

S.N	Property	Description
1	Appearance	Gets or sets a value determining the appearance of the radio button.
2	AutoCheck	Gets or sets a value indicating whether the Checked value and the appearance of the control automatically change when the control is clicked.
3	CheckAlign	Gets or sets the location of the check box portion of the radio button.
4	Checked	Gets or sets a value indicating whether the control is checked.

5	Text	Gets or sets the caption for a radio button.
6	TabStop	Gets or sets a value indicating whether a user can give focus to the RadioButton control using the TAB key.

Methods of the RadioButton Control

The following are some of the commonly used methods of the RadioButton control:

S.N	Method Name & Description
1	PerformClick Generates a Click event for the control, simulating a click by a user.

Events of the RadioButton Control

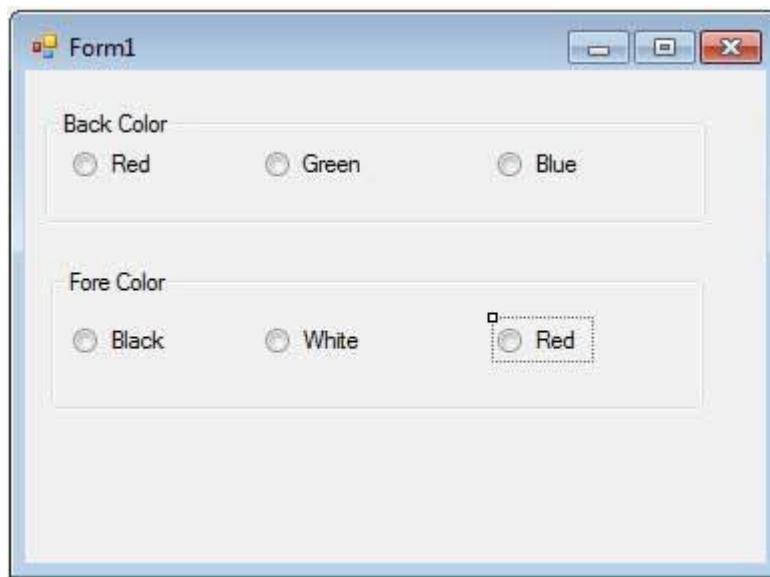
The following are some of the commonly used events of the RadioButton control:

S.N	Event	Description
1	AppearanceChanged	Occurs when the value of the Appearance property of the RadioButton control is changed.
2	CheckedChanged	Occurs when the value of the Checked property of the RadioButton control is changed.

Consult Microsoft documentation for detailed list of properties, methods and events of the RadioButton control.

Example

In the following example, let us create two groups of radio buttons and use their CheckedChanged events for changing the BackColor and ForeColor property of the form.



Let's double click on the radio buttons 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
        ' Set the caption bar text of the form.
        Me.Text = "tutorialspoint.com"
    End Sub

    Private Sub RadioButton1_CheckedChanged(sender As Object, _
        e As EventArgs) Handles RadioButton1.CheckedChanged
        Me.BackColor = Color.Red
    End Sub

    Private Sub RadioButton2_CheckedChanged(sender As Object, _
        e As EventArgs) Handles RadioButton2.CheckedChanged
        Me.BackColor = Color.Green
    End Sub

    Private Sub RadioButton3_CheckedChanged(sender As Object, _
        e As EventArgs) Handles RadioButton3.CheckedChanged
        Me.BackColor = Color.Blue
    End Sub

    Private Sub RadioButton4_CheckedChanged(sender As Object, _

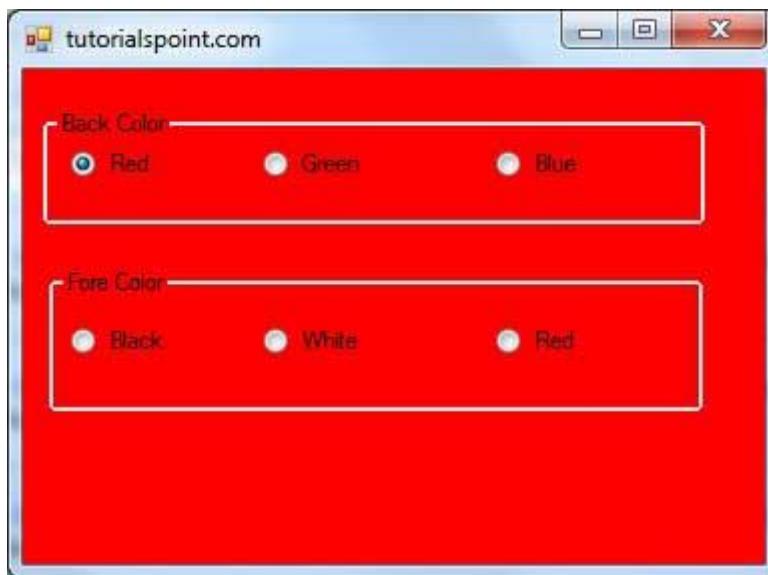
```

```

e As EventArgs) Handles RadioButton4.CheckedChanged
Me.ForeColor = Color.Black
End Sub
Private Sub RadioButton5_CheckedChanged(sender As Object,
e As EventArgs) Handles RadioButton5.CheckedChanged
Me.ForeColor = Color.White
End Sub
Private Sub RadioButton6_CheckedChanged(sender As Object,
e As EventArgs) Handles RadioButton6.CheckedChanged
Me.ForeColor = Color.Red
End Sub
End Class

```

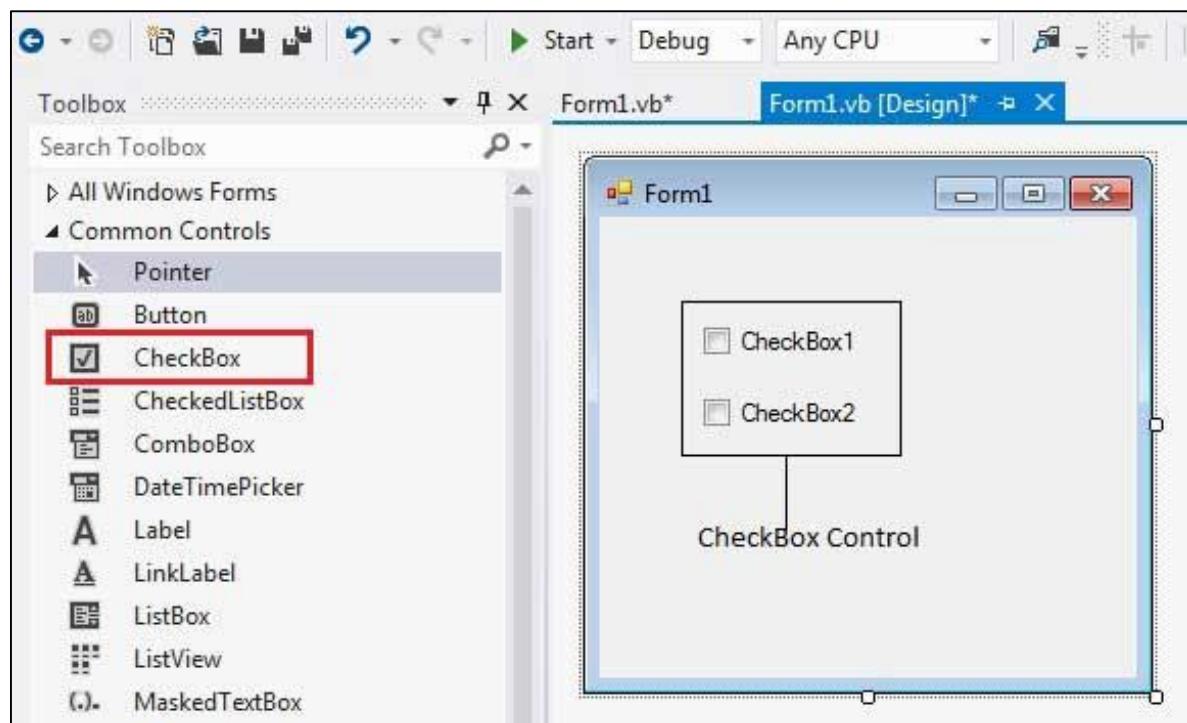
When the above code is executed and run using Start button available at the Microsoft Visual Studio tool bar, it will show the following window:



CheckBox Control

The CheckBox control allows the user to set true/false or yes/no type options. The user can select or deselect it. When a check box is selected it has the value True, and when it is cleared, it holds the value False.

Let's create two check boxes by dragging CheckBox controls from the Toolbox and dropping on the form.



The CheckBox control has three states, **checked**, **unchecked** and **indeterminate**. In the indeterminate state, the check box is grayed out. To enable the indeterminate state, the *ThreeState* property of the check box is set to be **True**.

Properties of the CheckBox Control

The following are some of the commonly used properties of the CheckBox control:

S.N	Property	Description
1	Appearance	Gets or sets a value determining the appearance of the check box.
2	AutoCheck	Gets or sets a value indicating whether the Checked or CheckedState value and the appearance of the control automatically change when the check box is selected.
3	CheckAlign	Gets or sets the horizontal and vertical alignment of the check mark on the check box.
4	Checked	Gets or sets a value indicating whether the check box is selected.

5	CheckState	Gets or sets the state of a check box.
6	Text	Gets or sets the caption of a check box.
7	ThreeState	Gets or sets a value indicating whether or not a check box should allow three check states rather than two.

Methods of the CheckBox Control

The following are some of the commonly used methods of the CheckBox control:

S.N	Method Name & Description
1	OnCheckedChanged Raises the CheckedChanged event.
2	OnCheckStateChanged Raises the CheckStateChanged event.
3	OnClick Raises the OnClick event.

Events of the CheckBox Control

The following are some of the commonly used events of the CheckBox control:

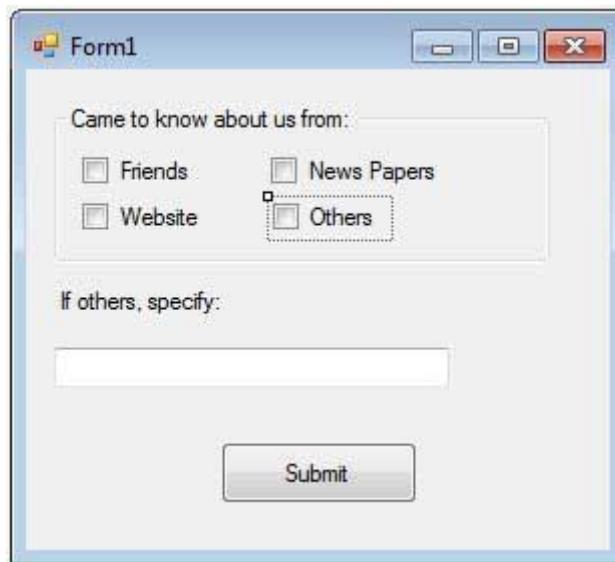
S.N	Event	Description
1	AppearanceChanged	Occurs when the value of the Appearance property of the check box is changed.
2	CheckedChanged	Occurs when the value of the Checked property of the CheckBox control is changed.
3	CheckStateChanged	Occurs when the value of the CheckState property of the CheckBox control is changed.

Consult Microsoft documentation for detailed list of properties, methods and events of the CheckBox control.

Example

In this example, let us add four check boxes in a group box. The check boxes will allow the users to choose the source from which they came to know about the organization. If the user chooses the check box with text "others", then the user is asked to specify and a text box is provided to give input. When the user clicks the Submit button, he/she gets an appropriate message.

The form in design view:



Let's put 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 = "tutorialspoint.com"
        Label1.Visible = False
        TextBox1.Visible = False
        TextBox1.Multiline = True
    End Sub
    Private Sub Button1_Click(sender As Object, e As EventArgs) _
        Handles Button1.Click
        Dim str As String
        str = " "
        If CheckBox1.Checked = True Then
            str &= CheckBox1.Text
        End If
    End Sub

```

```

        str &= " "
End If

If CheckBox2.Checked = True Then
    str &= CheckBox2.Text
    str &= " "
End If

If CheckBox3.Checked = True Then
    str &= CheckBox3.Text
    str &= " "
End If

If CheckBox4.Checked = True Then
    str &= TextBox1.Text
    str &= " "
End If

If str <> Nothing Then
    MsgBox(str + vbCrLf + "Thank you")
End If

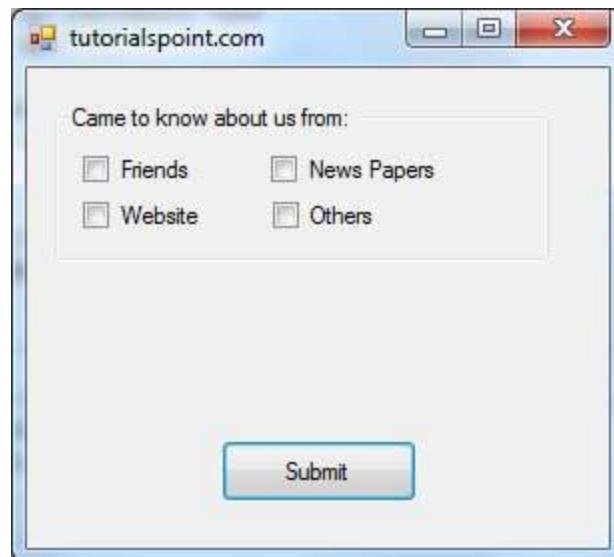
End Sub

Private Sub CheckBox4_CheckedChanged(sender As Object, _
e As EventArgs) Handles CheckBox4.CheckedChanged
    Label1.Visible = True
    TextBox1.Visible = True
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 the following window:



Checking all the boxes:



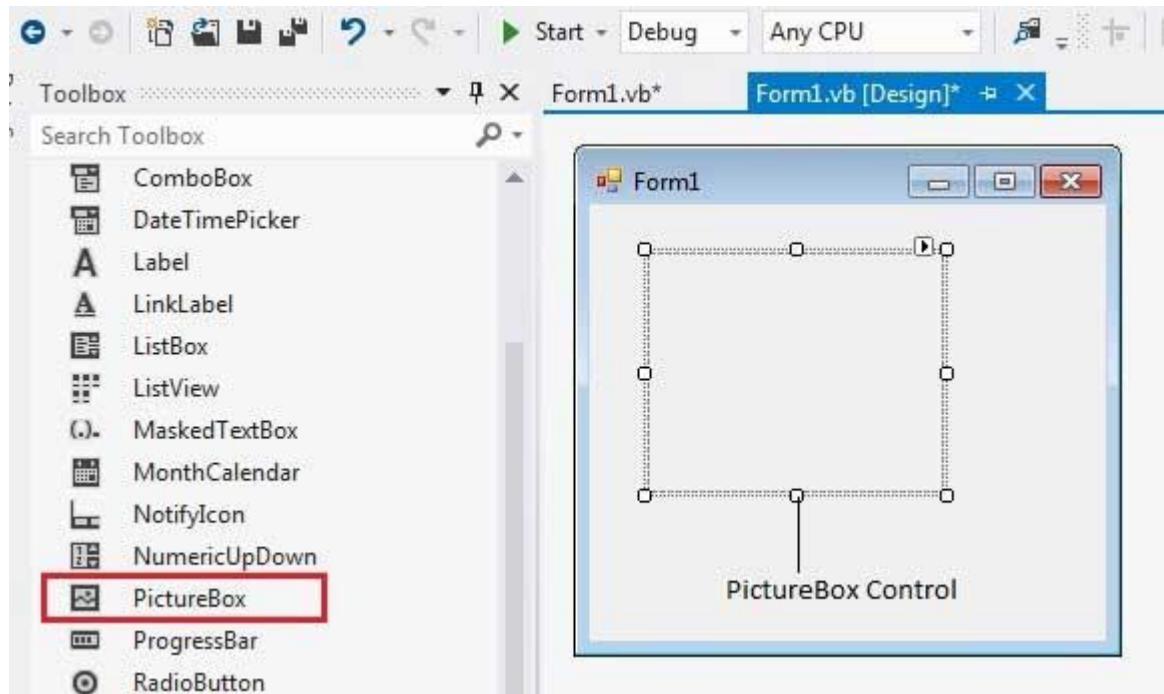
Clicking the Submit button:



PictureBox Control

The PictureBox control is used for displaying images on the form. The Image property of the control allows you to set an image both at design time or at run time.

Let's create a picture box by dragging a PictureBox control from the Toolbox and dropping it on the form.



Properties of the PictureBox Control

The following are some of the commonly used properties of the PictureBox control:

S.N	Property	Description
1	AllowDrop	Specifies whether the picture box accepts data that a user drags on it.
2	ErrorImage	Gets or specifies an image to be displayed when an error occurs during the image-loading process or if the image load is cancelled.
3	Image	Gets or sets the image that is displayed in the control.

4	ImageLocation	Gets or sets the path or the URL for the image displayed in the control.
5	InitialImage	Gets or sets the image displayed in the control when the main image is loaded
6	SizeMode	<p>Determines the size of the image to be displayed in the control. This property takes its value from the PictureBoxSizeMode enumeration, which has values:</p> <ul style="list-style-type: none"> • Normal - the upper left corner of the image is placed at upper left part of the picture box • StretchImage - allows stretching of the image • AutoSize - allows resizing the picture box to the size of the image • CenterImage - allows centering the image in the picture box • Zoom - allows increasing or decreasing the image size to maintain the size ratio.
7	TabIndex	Gets or sets the tab index value.
8	TabStop	Specifies whether the user will be able to focus on the picture box by using the TAB key.
9	Text	Gets or sets the text for the picture box.
10	WaitOnLoad	Specifies whether or not an image is loaded synchronously.

Methods of the PictureBox Control

The following are some of the commonly used methods of the PictureBox control:

S.N	Method Name & Description
1	CancelAsync Cancels an asynchronous image load.
2	Load Displays an image in the picture box
3	LoadAsync Loads image asynchronously.
4	ToString Returns the string that represents the current picture box.

Events of the PictureBox Control

The following are some of the commonly used events of the PictureBox control:

S.N	Event	Description
1	CausesValidationChanged	Overrides the Control.CausesValidationChanged property.
2	Click	Occurs when the control is clicked.
3	Enter	Overrides the Control.Enter property.
4	FontChanged	Occurs when the value of the Font property changes.
5	ForeColorChanged	Occurs when the value of the ForeColor property changes.
6	KeyDown	Occurs when a key is pressed when the control has focus.

7	KeyPress	Occurs when a key is pressed when the control has focus.
8	KeyUp	Occurs when a key is released when the control has focus.
9	Leave	Occurs when input focus leaves the PictureBox.
10	LoadCompleted	Occurs when the asynchronous image-load operation is completed, been canceled, or raised an exception.
11	LoadProgressChanged	Occurs when the progress of an asynchronous image-loading operation has changed.
12	Resize	Occurs when the control is resized.
13	RightToLeftChanged	Occurs when the value of the RightToLeft property changes.
14	SizeChanged	Occurs when the Size property value changes.
15	SizeModeChanged	Occurs whenSizeMode changes.
16	TabIndexChanged	Occurs when the value of the TabIndex property changes.
17	TabStopChanged	Occurs when the value of the TabStop property changes.
18	TextChanged	Occurs when the value of the Text property changes.

Example

In this example, let us put a picture box and a button control on the form. We set the image property of the picture box to logo.png, as we used before. The Click

event of the button named Button1 is coded to stretch the image to a specified size:

```
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  
        PictureBox1.ClientSize = New Size(300, 300)  
        PictureBox1.SizeMode = PictureBoxSizeMode.StretchImage  
    End Sub  
End Class
```

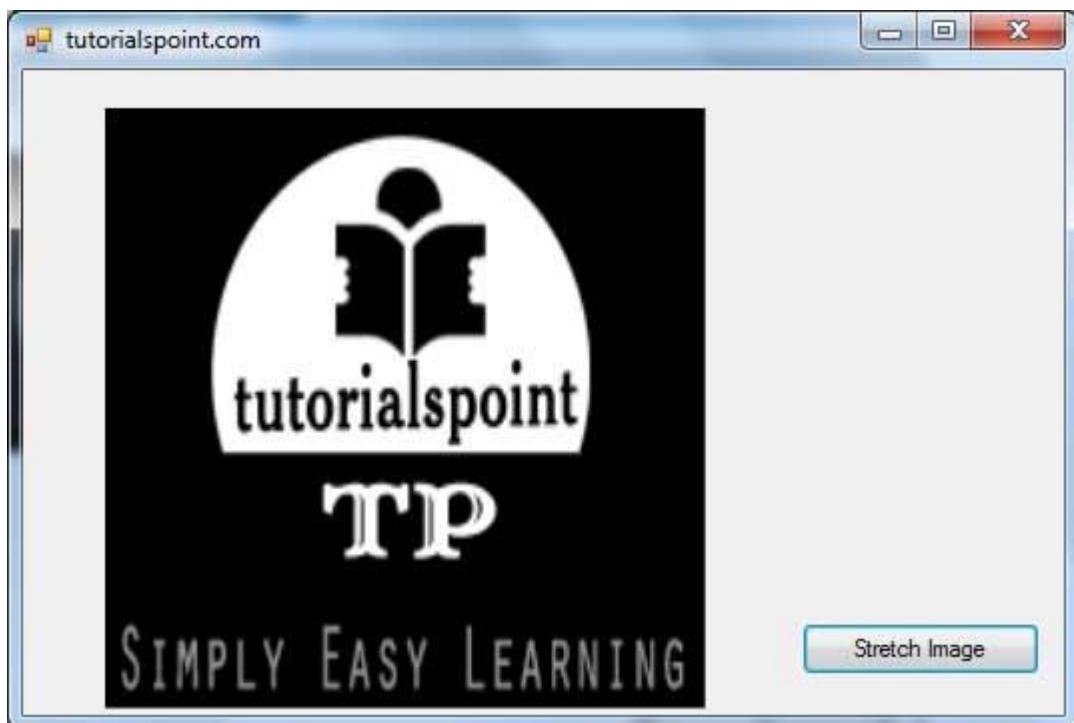
Design View:



When the application is executed, it displays:



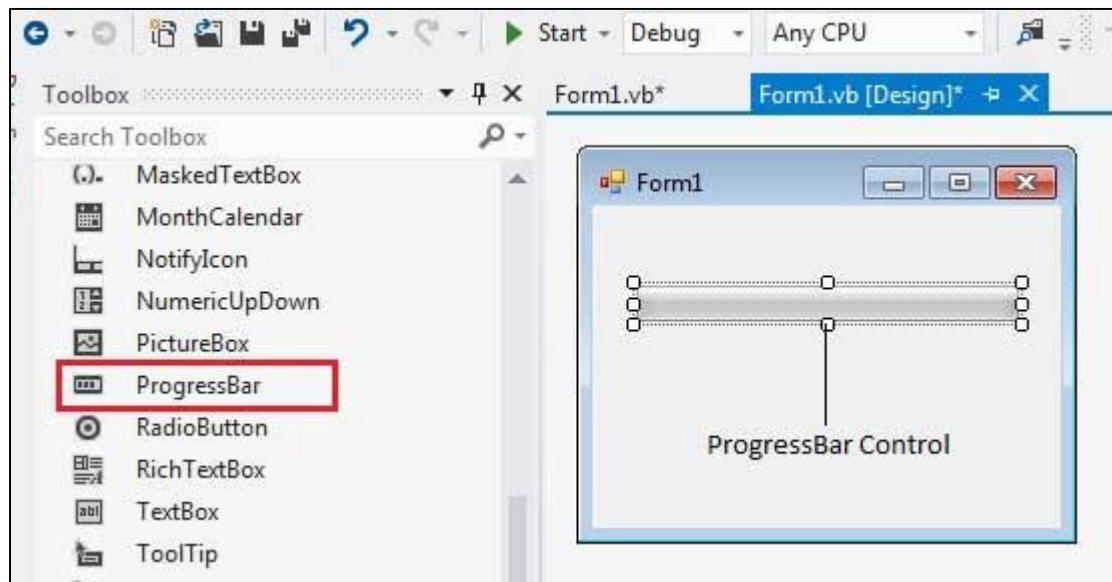
Clicking on the button results in:



ProgressBar Control

It represents a Windows progress bar control. It is used to provide visual feedback to your users about the status of some task. It shows a bar that fills in from left to right as the operation progresses.

Let's click on a ProgressBar control from the Toolbox and place it on the form.



The main properties of a progress bar are *Value*, *Maximum* and *Minimum*. The *Minimum* and *Maximum* properties are used to set the minimum and maximum values that the progress bar can display. The *Value* property specifies the current position of the progress bar.

The *ProgressBar* control is typically used when an application performs tasks such as copying files or printing documents. To a user the application might look unresponsive if there is no visual cue. In such cases, using the *ProgressBar* allows the programmer to provide a visual status of progress.

Properties of the ProgressBar Control

The following are some of the commonly used properties of the *ProgressBar* control:

S.N	Property	Description
1	AllowDrop	Overrides Control.AllowDrop.
2	BackgroundImage	Gets or sets the background image for the <i>ProgressBar</i> control.

3	BackgroundImageLayout	Gets or sets the layout of the background image of the progress bar.
4	CausesValidation	Gets or sets a value indicating whether the control, when it receives focus, causes validation to be performed on any controls that require validation.
5	Font	Gets or sets the font of text in the ProgressBar.
6	ImeMode	Gets or sets the input method editor (IME) for the ProgressBar.
7	ImeModeBase	Gets or sets the IME mode of a control.
8	MarqueeAnimationSpeed	Gets or sets the time period, in milliseconds, that it takes the progress block to scroll across the progress bar.
9	Maximum	Gets or sets the maximum value of the range of the control.
10	Minimum	Gets or sets the minimum value of the range of the control.
11	Padding	Gets or sets the space between the edges of a ProgressBar control and its contents.
12	RightToLeftLayout	Gets or sets a value indicating whether the ProgressBar and any text it contains is displayed from right to left.
13	Step	Gets or sets the amount by which a call to the PerformStep method increases the current position of the progress bar.
14	Style	Gets or sets the manner in which progress should be indicated on the progress bar.

15	Value	Gets or sets the current position of the progress bar.
----	--------------	--

Methods of the ProgressBar Control

The following are some of the commonly used methods of the ProgressBar control:

S.N	Method Name & Description
1	Increment Increments the current position of the ProgressBar control by specified amount.
2	PerformStep Increments the value by the specified step.
3	ResetText Resets the Text property to its default value.
4	ToString Returns a string that represents the progress bar control.

Events of the ProgressBar Control

The following are some of the commonly used events of the ProgressBar control:

S.N	Event	Description
1	BackgroundImageChanged	Occurs when the value of the BackgroundImage property changes.
2	BackgroundImageLayoutChanged	Occurs when the value of the BackgroundImageLayout property changes.
3	CausesValidationChanged	Occurs when the value of the CausesValidation property changes.

4	Click	Occurs when the control is clicked.
5	DoubleClick	Occurs when the user double-clicks the control.
6	Enter	Occurs when focus enters the control.
7	FontChanged	Occurs when the value of the Font property changes.
8	ImeModeChanged	Occurs when the value of the ImeMode property changes.
9	KeyDown	Occurs when the user presses a key while the control has focus.
10	KeyPress	Occurs when the user presses a key while the control has focus.
11	KeyUp	Occurs when the user releases a key while the control has focus.
12	Leave	Occurs when focus leaves the ProgressBar control.
13	MouseClick	Occurs when the control is clicked by the mouse.
14	MouseDoubleClick	Occurs when the user double-clicks the control.
15	PaddingChanged	Occurs when the value of the Padding property changes.
16	Paint	Occurs when the ProgressBar is drawn.
17	RightToLeftLayoutChanged	Occurs when the RightToLeftLayout property changes.

18	TabStopChanged	Occurs when the TabStop property changes.
18	TextChanged	Occurs when the Text property changes.

Example

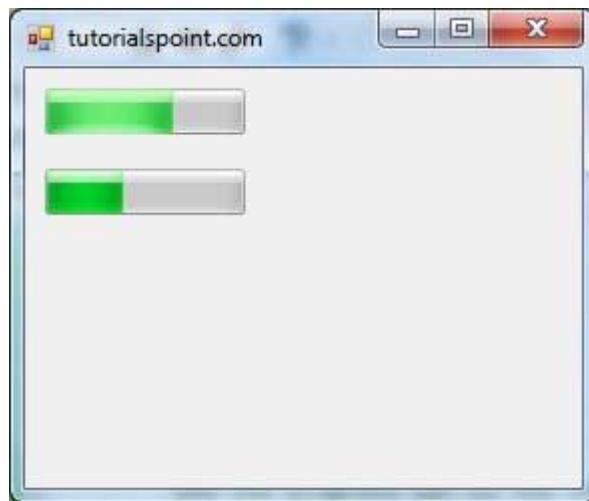
In this example, let us create a progress bar at runtime. Let's double click on the Form and put the follow code in the opened window.

```

Public Class Form1
    Private Sub Form1_Load(sender As Object, e As EventArgs) _
        Handles MyBase.Load
        'create two progress bars
        Dim ProgressBar1 As ProgressBar
        Dim ProgressBar2 As ProgressBar
        ProgressBar1 = New ProgressBar()
        ProgressBar2 = New ProgressBar()
        'set position
        ProgressBar1.Location = New Point(10, 10)
        ProgressBar2.Location = New Point(10, 50)
        'set values
        ProgressBar1.Minimum = 0
        ProgressBar1.Maximum = 200
        ProgressBar1.Value = 130
        ProgressBar2.Minimum = 0
        ProgressBar2.Maximum = 100
        ProgressBar2.Value = 40
        'add the progress bar to the form
        Me.Controls.Add(ProgressBar1)
        Me.Controls.Add(ProgressBar2)
        ' Set the caption bar text of the form.
        Me.Text = "tutorialspoint.com"
    End Sub
End Class

```

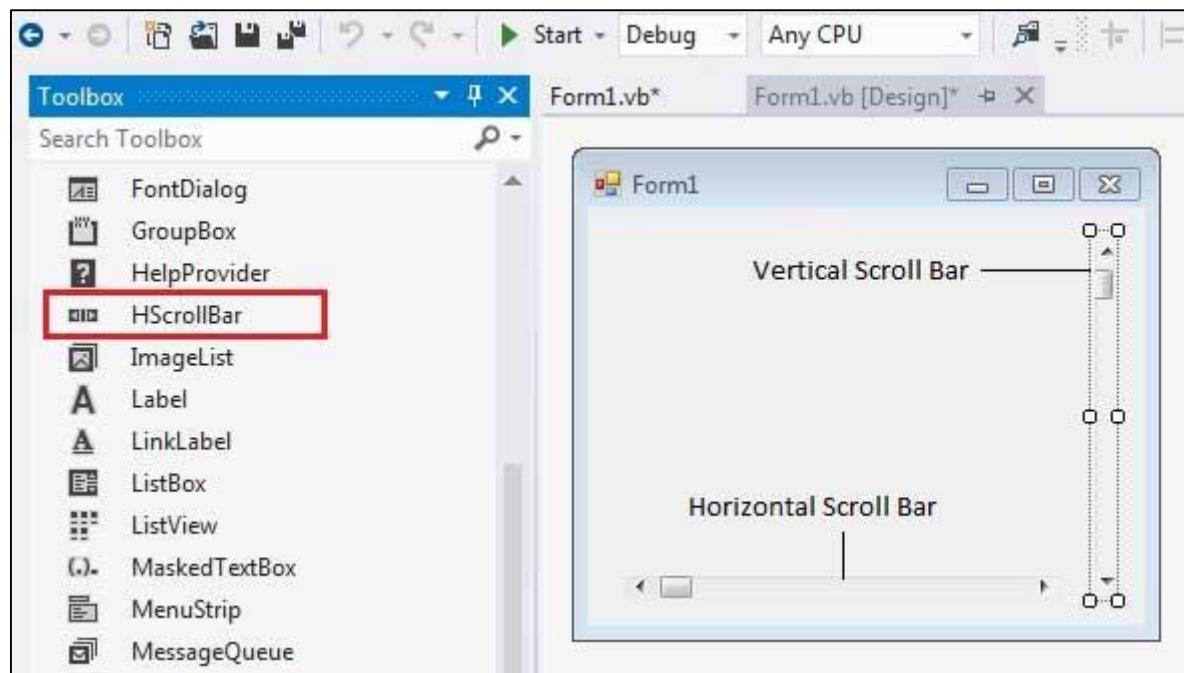
When the above code is executed and run using **Start** button available at the Microsoft Visual Studio tool bar, it will show the following window:



ScrollBar Control

The ScrollBar controls display vertical and horizontal scroll bars on the form. This is used for navigating through large amount of information. There are two types of scroll bar controls: **HScrollBar** for horizontal scroll bars and **VScrollBar** for vertical scroll bars. These are used independently from each other.

Let's click on HScrollBar control and VScrollBar control from the Toolbox and place them on the form.



Properties of the ScrollBar Control

The following are some of the commonly used properties of the ScrollBar control:

S.N	Property	Description
1	AutoSize	Gets or sets a value indicating whether the ScrollBar is automatically resized to fit its contents.
2	BackColor	Gets or sets the background color for the control.
3	ForeColor	Gets or sets the foreground color of the scroll bar control.
4	ImeMode	Gets or sets the Input Method Editor (IME) mode supported by this control.
5	LargeChange	Gets or sets a value to be added to or subtracted from the Value property when the scroll box is moved a large distance.
6	Maximum	Gets or sets the upper limit of values of the scrollable range.
7	Minimum	Gets or sets the lower limit of values of the scrollable range.
8	SmallChange	Gets or sets the value to be added to or subtracted from the Value property when the scroll box is moved a small distance.
9	Value	Gets or sets a numeric value that represents the current position of the scroll box on the scroll bar control.

Methods of the ScrollBar Control

The following are some of the commonly used methods of the ScrollBar control:

S.N	Method Name & Description
1	OnClick Generates the Click event.
2	Select Activates the control.

Events of the ScrollBar Control

The following are some of the commonly used events of the ScrollBar control:

S.N	Event	Description
1	Click	Occurs when the control is clicked.
2	DoubleClick	Occurs when the user double-clicks the control.
3	Scroll	Occurs when the control is moved.
4	ValueChanged	Occurs when the Value property changes, either by handling the Scroll event or programmatically.

Example

In this example, let us create two scroll bars at runtime. Let's double click on the Form and put the follow code in the opened window.

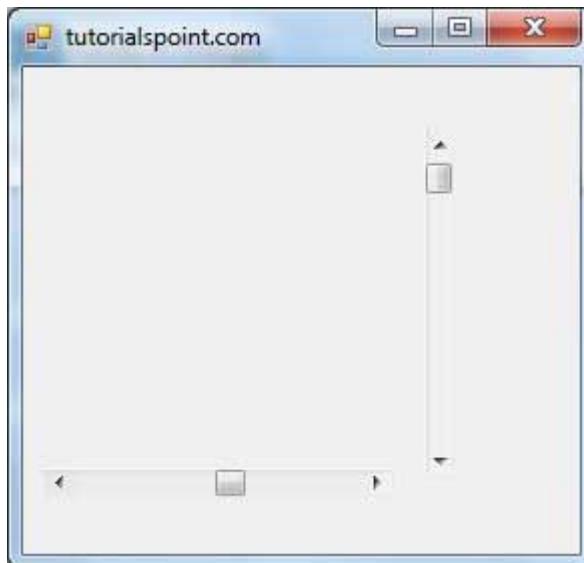
```
Public Class Form1
    Private Sub Form1_Load(sender As Object, e As EventArgs) _
        Handles MyBase.Load
        'create two scroll bars
        Dim hs As HScrollBar
        Dim vs As VScrollBar
        hs = New HScrollBar()
```

```

vs = New VScrollBar()
'set properties
hs.Location = New Point(10, 200)
hs.Size = New Size(175, 15)
hs.Value = 50
vs.Location = New Point(200, 30)
vs.Size = New Size(15, 175)
hs.Value = 50
'adding the scroll bars to the form
Me.Controls.Add(hs)
Me.Controls.Add(vs)
' Set the caption bar text of the form.
Me.Text = "tutorialspoint.com"
End Sub
End Class

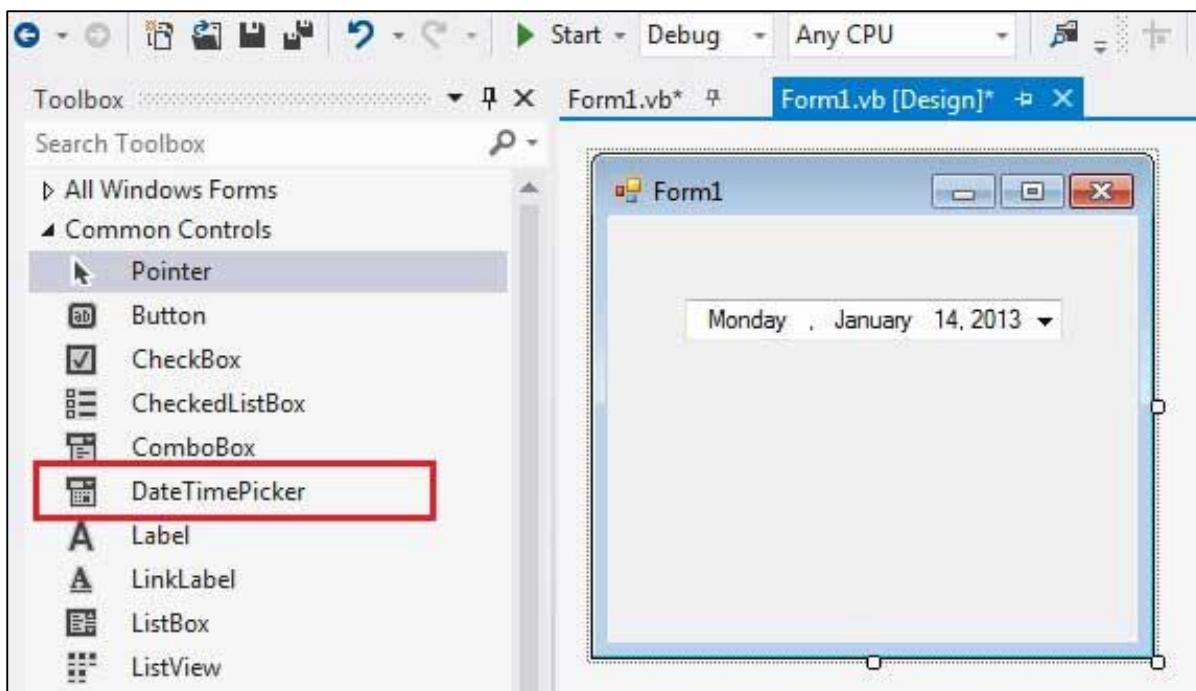
```

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



DateTimePicker Control

The DateTimePicker control allows selecting a date and time by editing the displayed values in the control. If you click the arrow in the DateTimePicker control, it displays a month calendar, like a combo box control. The user can make selection by clicking the required date. The new selected value appears in the text box part of the control.



The **MinDate** and the **MaxDate** properties allow you to put limits on the date range.

Properties of the DateTimePicker Control

The following are some of the commonly used properties of the DateTimePicker control:

S.N	Property	Description
1	BackColor	Gets or sets a value indicating the background color of the DateTimePicker control.
2	BackgroundImage	Gets or sets the background image for the control.
3	BackgroundImageLayout	Gets or sets the layout of the background image of the DateTimePicker control.
4	CalendarFont	Gets or sets the font style applied to the calendar.
5	CalendarForeColor	Gets or sets the foreground color of the calendar.

6	CalendarMonthBackground	Gets or sets the background color of the calendar month.
7	CalendarTitleBackColor	Gets or sets the background color of the calendar title.
8	CalendarTitleForeColor	Gets or sets the foreground color of the calendar title.
9	CalendarTrailingForeColor	Gets or sets the foreground color of the calendar trailing dates.
10	Checked	Gets or sets a value indicating whether the Value property has been set with a valid date/time value and the displayed value is able to be updated.
11	CustomFormat	Gets or sets the custom date/time format string.
12	DropDownAlign	Gets or sets the alignment of the drop-down calendar on the DateTimePicker control.
13	ForeColor	Gets or sets the foreground color of the DateTimePicker control.
14	Format	Gets or sets the format of the date and time displayed in the control.
15	MaxDate	Gets or sets the maximum date and time that can be selected in the control.
16	MaximumDateTime	Gets the maximum date value allowed for the DateTimePicker control.
17	MinDate	Gets or sets the minimum date and time that can be selected in the control.
18	MinimumDateTime	Gets the minimum date value allowed for the DateTimePicker control.

19	PreferredHeight	Gets the preferred height of the DateTimePicker control.
20	RightToLeftLayout	Gets or sets whether the contents of the DateTimePicker are laid out from right to left.
21	ShowCheckBox	Gets or sets a value indicating whether a check box is displayed to the left of the selected date.
22	ShowUpDown	Gets or sets a value indicating whether a spin button control (also known as an up-down control) is used to adjust the date/time value.
23	Text	Gets or sets the text associated with this control.
24	Value	Gets or sets the date/time value assigned to the control.

Methods of the DateTimePicker Control

The following are some of the commonly used methods of the DateTimePicker control:

S.N	Method Name & Description
1	ToString Returns the string representing the control.

Events of the DateTimePicker Control

The following are some of the commonly used events of the DateTimePicker control:

S.N	Event	Description
1	BackColorChanged	Occurs when the value of the BackColor property changes.

2	BackgroundImageChanged	Occurs when the value of the BackgroundImage property changes.
3	BackgroundImageLayoutChanged	Occurs when the value of the BackgroundImageLayout property changes.
4	Click	Occurs when the control is clicked.
5	CloseUp	Occurs when the drop-down calendar is dismissed and disappears.
6	DoubleClick	Occurs when the control is double-clicked.
7	DragDrop	Occurs when a drag-and-drop operation is completed.
8	ForeColorChanged	Occurs when the value of the ForeColor property changes.
9	FormatChanged	Occurs when the Format property value has changed.
10	MouseClick	Occurs when the control is clicked with the mouse.
11	MouseDoubleClick	Occurs when the control is double-clicked with the mouse.
12	PaddingChanged	Occurs when the value of the Padding property changes.
13	Paint	Occurs when the control is redrawn.

14	RightToLeftLayoutChanged	Occurs when the RightToLeftLayout property changes.
15	TextChanged	Occurs when the value of the Text property changes.
16	ValueChanged	Occurs when the Value property changes.

Example

In this example, let us create a small application for calculating days of leave. Let us add two DateTimePicker controls on the form, where the user will enter the date of going on leave and the date of joining. Let us keep a button control for performing the calculation and appropriate label controls for displaying information.

The form in design view:



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 = "tutorialspoint.com"
    End Sub
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click

```

```

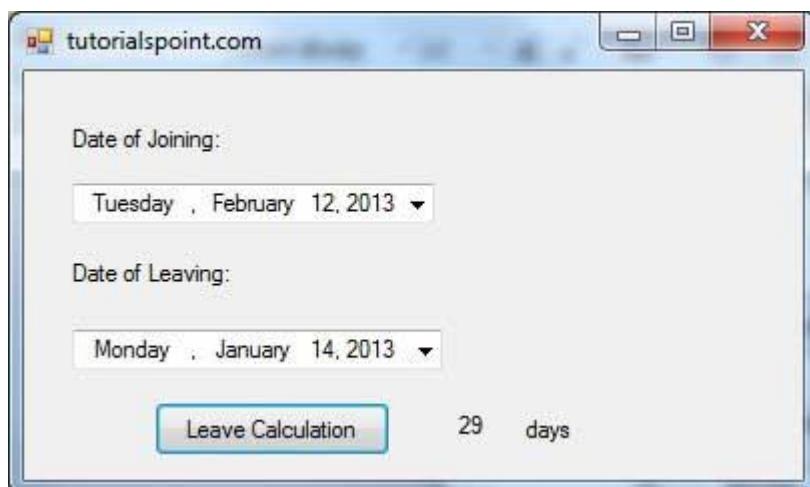
Dim d1 As DateTime = DateTimePicker1.Value
Dim d2 As DateTime = DateTimePicker2.Value
Dim result As TimeSpan = d1.Subtract(d2)
Dim days As Integer = result.TotalDays
Label3.Text = days
End Sub
End Class

```

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



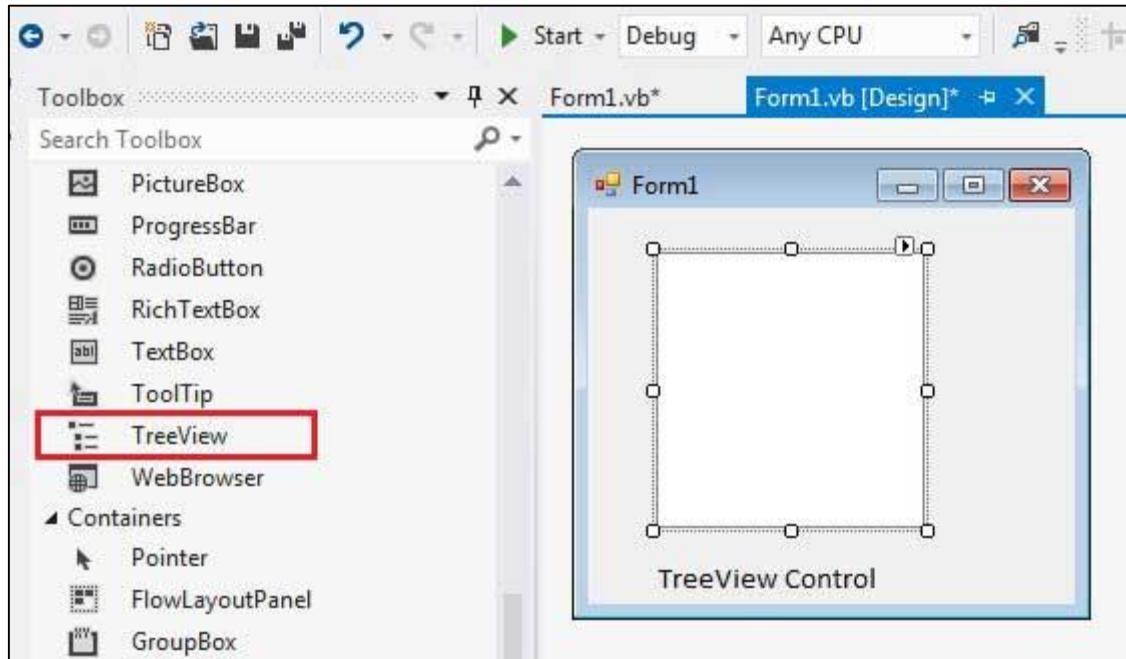
Select two dates and click on the button for leave calculation:



TreeView Control

The TreeView control is used to display hierarchical representations of items similar to the ways the files and folders are displayed in the left pane of the Windows Explorer. Each node may contain one or more child nodes.

Let's click on a TreeView control from the Toolbox and place it on the form.



Properties of the TreeView Control

The following are some of the commonly used properties of the TreeView control:

S.N	Property	Description
1	BackColor	Gets or sets the background color for the control.
2	BackgroundImage	Gets or set the background image for the TreeView control.
3	BackgroundImageLayout	Gets or sets the layout of the background image for the TreeView control.
4	BorderStyle	Gets or sets the border style of the tree view control.
5	Checkboxes	Gets or sets a value indicating whether check boxes are displayed next to the tree nodes in the tree view control.

6	DataBindings	Gets the data bindings for the control.
7	Font	Gets or sets the font of the text displayed by the control.
8	FontHeight	Gets or sets the height of the font of the control.
9	ForeColor	The current foreground color for this control, which is the color the control uses to draw its text.
10	ItemHeight	Gets or sets the height of each tree node in the tree view control.
11	Nodes	Gets the collection of tree nodes that are assigned to the tree view control.
12	PathSeparator	Gets or sets the delimiter string that the tree node path uses.
13	RightToLeftLayout	Gets or sets a value that indicates whether the TreeView should be laid out from right-to-left.
14	Scrollable	Gets or sets a value indicating whether the tree view control displays scroll bars when they are needed.
15	SelectedImageIndex	Gets or sets the image list index value of the image that is displayed when a tree node is selected.
16	SelectedImageKey	Gets or sets the key of the default image shown when a TreeNode is in a selected state.
17	SelectedNode	Gets or sets the tree node that is currently selected in the tree view control.

18	ShowLines	Gets or sets a value indicating whether lines are drawn between tree nodes in the tree view control.
19	ShowNodeToolTips	Gets or sets a value indicating ToolTips are shown when the mouse pointer hovers over a TreeNode.
20	ShowPlusMinus	Gets or sets a value indicating whether plus-sign (+) and minus-sign (-) buttons are displayed next to tree nodes that contain child tree nodes.
21	ShowRootLines	Gets or sets a value indicating whether lines are drawn between the tree nodes that are at the root of the tree view.
22	Sorted	Gets or sets a value indicating whether the tree nodes in the tree view are sorted.
23	StateImageList	Gets or sets the image list that is used to indicate the state of the TreeView and its nodes.
24	Text	Gets or sets the text of the TreeView.
25	TopNode	Gets or sets the first fully-visible tree node in the tree view control.
26	TreeViewNodeSorter	Gets or sets the implementation of IComparer to perform a custom sort of the TreeView nodes.
27	VisibleCount	Gets the number of tree nodes that can be fully visible in the tree view control.

Methods of the TreeView Control

The following are some of the commonly used methods of the TreeView control:

S.N	Method Name & Description
1	CollapseAll Collapses all the nodes including all child nodes in the tree view control.
2	ExpandAll Expands all the nodes.
3	GetNodeAt Gets the node at the specified location.
4	GetNodeCount Gets the number of tree nodes.
5	Sort Sorts all the items in the tree view control.
6	ToString Returns a string containing the name of the control.

Events of the TreeView Control

The following are some of the commonly used events of the TreeView control:

S.N	Event	Description
1	AfterCheck	Occurs after the tree node check box is checked.
2	AfterCollapse	Occurs after the tree node is collapsed.
3	AfterExpand	Occurs after the tree node is expanded.
4	AfterSelect	Occurs after the tree node is selected.
5	BeforeCheck	Occurs before the tree node check box is checked.
6	BeforeCollapse	Occurs before the tree node is collapsed.
7	BeforeExpand	Occurs before the tree node is expanded.

8	BeforeLabelEdit	Occurs before the tree node label text is edited.
9	BeforeSelect	Occurs before the tree node is selected.
10	ItemDrag	Occurs when the user begins dragging a node.
11	NodeMouseClick	Occurs when the user clicks a TreeNode with the mouse.
12	NodeMouseDoubleClick	Occurs when the user double-clicks a TreeNode with the mouse.
13	NodeMouseHover	Occurs when the mouse hovers over a TreeNode.
14	PaddingChanged	Occurs when the value of the Padding property changes.
15	Paint	Occurs when the TreeView is drawn.
16	RightToLeftLayoutChanged	Occurs when the value of the RightToLeftLayout property changes.
17	TextChanged	Occurs when the Text property changes.

The TreeNode Class

The *TreeNode* class represents a **node** of a *TreeView*. Each node in a *TreeView* control is an object of the *TreeNode* class. To be able to use a *TreeView* control we need to have a look at some commonly used properties and methods of the *TreeNode* class.

Properties of the TreeNode Class

The following are some of the commonly used properties of the *TreeNode* class:

S.N	Property	Description
1	BackColor	Gets or sets the background color of the tree node.

2	Checked	Gets or sets a value indicating whether the tree node is in a checked state.
3	ContextMenu	Gets the shortcut menu that is associated with this tree node.
4	ContextMenuStrip	Gets or sets the shortcut menu associated with this tree node.
5	FirstNode	Gets the first child tree node in the tree node collection.
6	FullPath	Gets the path from the root tree node to the current tree node.
7	Index	Gets the position of the tree node in the tree node collection.
8	IsEditing	Gets a value indicating whether the tree node is in an editable state.
9	IsExpanded	Gets a value indicating whether the tree node is in the expanded state.
10	isSelected	Gets a value indicating whether the tree node is in the selected state.
11	IsVisible	Gets a value indicating whether the tree node is visible or partially visible.
12	LastNode	Gets the last child tree node.
13	Level	Gets the zero-based depth of the tree node in the TreeView control.
14	Name	Gets or sets the name of the tree node.
15	NextNode	Gets the next sibling tree node.

16	Nodes	Gets the collection of TreeNode objects assigned to the current tree node.
17	Parent	Gets the parent tree node of the current tree node.
18	PrevNode	Gets the previous sibling tree node.
19	PrevVisibleNode	Gets the previous visible tree node.
20	Tag	Gets or sets the object that contains data about the tree node.
21	Text	Gets or sets the text displayed in the label of the tree node.
22	ToolTipText	Gets or sets the text that appears when the mouse pointer hovers over a TreeNode.
23	TreeView	Gets the parent tree view that the tree node is assigned to.

Methods of the TreeNode Class

The following are some of the commonly used methods of the TreeNode class:

S.N	Method Name & Description
1	Collapse Collapses the tree node.
2	Expand Expands the tree node.
3	ExpandAll Expands all the child tree nodes.
4	GetNodeCount Returns the number of child tree nodes.

5	Remove Removes the current tree node from the tree view control.
6	Toggle Toggles the tree node to either the expanded or collapsed state.
7	ToString Returns a string that represents the current object.

Example

In this example, let us create a tree view at runtime. Let's double click on the Form and put the follow code in the opened window.

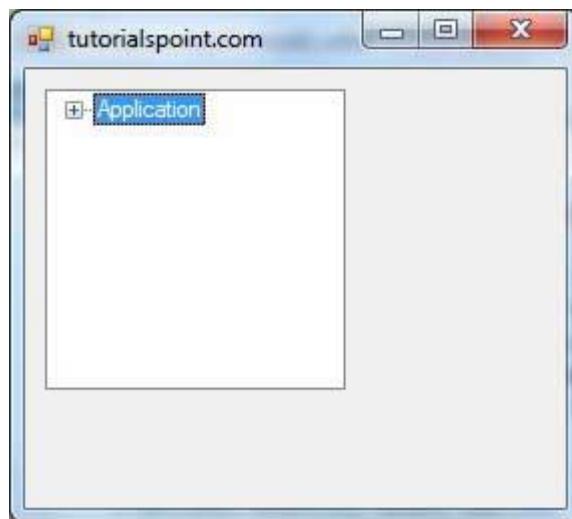
```
Public Class Form1
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        'create a new TreeView
        Dim TreeView1 As TreeView
        TreeView1 = New TreeView()
        TreeView1.Location = New Point(10, 10)
        TreeView1.Size = New Size(150, 150)
        Me.Controls.Add(TreeView1)
        TreeView1.Nodes.Clear()
        'Creating the root node
        Dim root = New TreeNode("Application")
        TreeView1.Nodes.Add(root)
        TreeView1.Nodes(0).Nodes.Add(New TreeNode("Project 1"))
        'Creating child nodes under the first child
        For loopindex As Integer = 1 To 4
            TreeView1.Nodes(0).Nodes(0).Nodes.Add(New _
                TreeNode("Sub Project" & Str(loopindex)))
        Next loopindex
        ' creating child nodes under the root
        TreeView1.Nodes(0).Nodes.Add(New TreeNode("Project 6"))
        'creating child nodes under the created child node
        For loopindex As Integer = 1 To 3
```

```

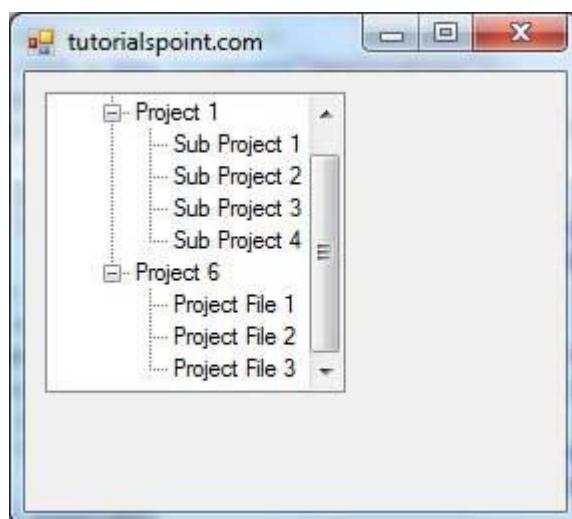
TreeView1.Nodes(0).Nodes(1).Nodes.Add(New _
    TreeNode("Project File" & Str(loopindex)))
Next loopindex
' Set the caption bar text of the form.
Me.Text = "tutorialspoint.com"
End Sub
End Class

```

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

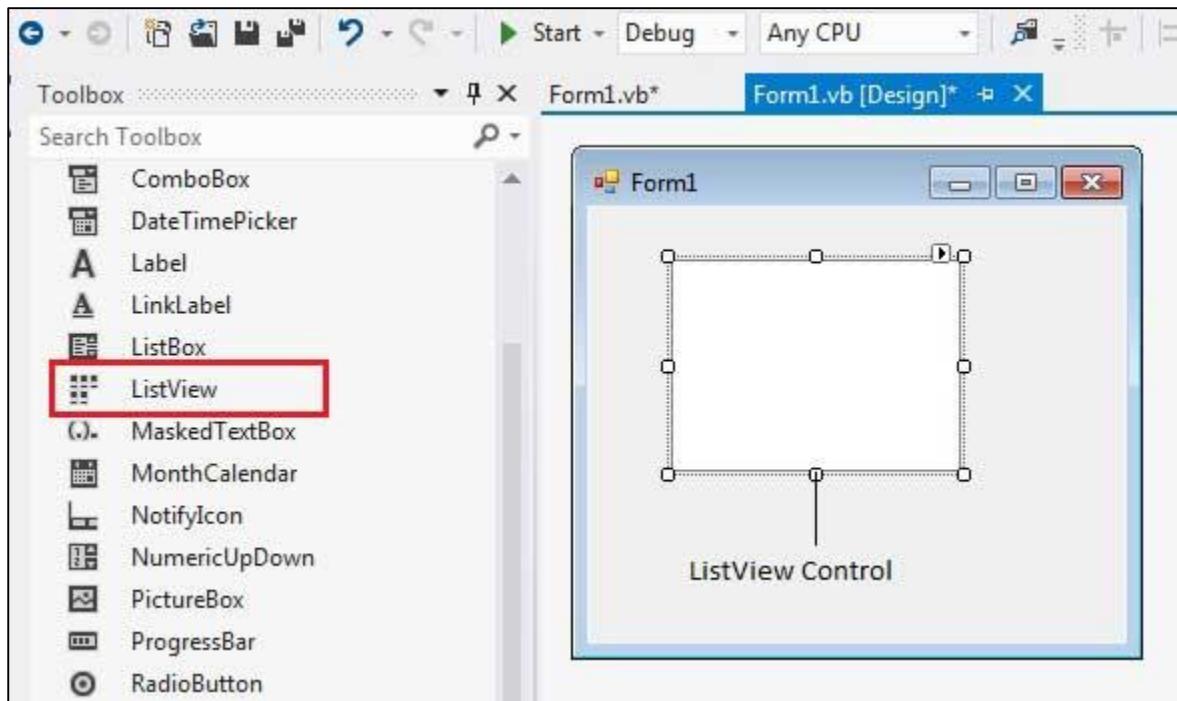


You can expand the nodes to see the child nodes:



ListView Control

The ListView control is used to display a list of items. Along with the TreeView control, it allows you to create a Windows Explorer like interface. Let's click on a ListView control from the Toolbox and place it on the form.



The *ListView* control displays a list of items along with icons. The *Item* property of the *ListView* control allows you to add and remove items from it. The *SelectedItem* property contains a collection of the selected items. The *MultiSelect* property allows you to set select more than one item in the list view. The *Checkboxes* property allows you to set check boxes next to the items.

Properties of the ListView Control

The following are some of the commonly used properties of the ListView control:

S.N	Property	Description
1	Alignment	Gets or sets the alignment of items in the control.
2	AutoArrange	Gets or sets whether icons are automatically kept arranged.
3	BackColor	Gets or sets the background color.

4	Checkboxes	Gets or sets a value indicating whether a check box appears next to each item in the control.
5	CheckedIndices	Gets the indexes of the currently checked items in the control.
6	CheckedItems	Gets the currently checked items in the control.
7	Columns	Gets the collection of all column headers that appear in the control.
8	GridLines	Gets or sets a value indicating whether grid lines appear between the rows and columns containing the items and subitems in the control.
9	HeaderStyle	Gets or sets the column header style.
10	HideSelection	Gets or sets a value indicating whether the selected item in the control remains highlighted when the control loses focus.
11	HotTracking	Gets or sets a value indicating whether the text of an item or subitem has the appearance of a hyperlink when the mouse pointer passes over it.
12	HoverSelection	Gets or sets a value indicating whether an item is automatically selected when the mouse pointer remains over the item for a few seconds.
13	InsertionMark	Gets an object used to indicate the expected drop location when an item is dragged within a ListView control.
14	Items	Gets a collection containing all items in the control.

15	LabelWrap	Gets or sets a value indicating whether item labels wrap when items are displayed in the control as icons.
16	LargeImageList	Gets or sets the ImageList to use when displaying items as large icons in the control.
17	MultiSelect	Gets or sets a value indicating whether multiple items can be selected.
18	RightToLeftLayout	Gets or sets a value indicating whether the control is laid out from right to left.
19	Scrollable	Gets or sets a value indicating whether a scroll bar is added to the control when there is not enough room to display all items.
20	SelectedIndices	Gets the indexes of the selected items in the control.
21	SelectedItems	Gets the items that are selected in the control.
22	ShowGroups	Gets or sets a value indicating whether items are displayed in groups.
23	ShowItemToolTips	Gets or sets a value indicating whether ToolTips are shown for the ListViewItem objects contained in the ListView.
24	SmallImageList	Gets or sets the ImageList to use when displaying items as small icons in the control.
25	Sorting	Gets or sets the sort order for items in the control.

26	StateImageList	Gets or sets the ImageList associated with application-defined states in the control.
27	TopItem	Gets or sets the first visible item in the control.
28	View	<p>Gets or sets how items are displayed in the control. This property has the following values:</p> <ul style="list-style-type: none"> • LargeIcon - displays large items with a large 32 x 32 pixels icon. • SmallIcon - displays items with a small 16 x 16 pixels icon • List - displays small icons always in one column • Details - displays items in multiple columns with column headers and fields • Tile - displays items as full-size icons with item labels and sub-item information.
29	VirtualListSize	Gets or sets the number of ListViewItem objects contained in the list when in virtual mode.
30	VirtualMode	Gets or sets a value indicating whether you have provided your own data-management operations for the ListView control.

Methods of the ListView Control

The following are some of the commonly used methods of the ListView control:

S.N	Method Name & Description
1	Clear Removes all items from the ListView control.
1	ToString Returns a string containing the string representation of the control.

Events of the ListView Control

The following are some of the commonly used events of the ListView control:

S.N	Event	Description
1	ColumnClick	Occurs when a column header is clicked.
2	ItemCheck	Occurs when an item in the control is checked or unchecked.
3	SelectedIndexChanged	Occurs when the selected index is changed.
4	TextChanged	Occurs when the Text property is changed.

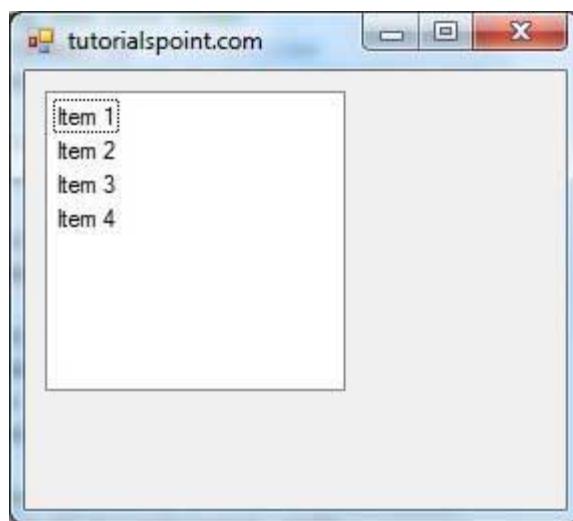
Example

In this example, let us create a list view at runtime. 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
        'create a new ListView
        Dim ListView1 As ListView
        ListView1 = New ListView()
        ListView1.Location = New Point(10, 10)
        ListView1.Size = New Size(150, 150)
        Me.Controls.Add(ListView1)
        'Creating the list items
        Dim ListItem1 As ListViewItem
        ListItem1 = ListView1.Items.Add("Item 1")
        Dim ListItem2 As ListViewItem
    End Sub
End Class
```

```
ListItem2 = ListView1.Items.Add("Item 2")
Dim ListItem3 As ListViewItem
ListItem3 = ListView1.Items.Add("Item 3")
Dim ListItem4 As ListViewItem
ListItem4 = ListView1.Items.Add("Item 4")
'set the view property
ListView1.View = View.SmallIcon
' 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 the following window:



24. 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 setup, etc., to the user of an application. These built-in dialog boxes reduce the developer's time and workload.

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.

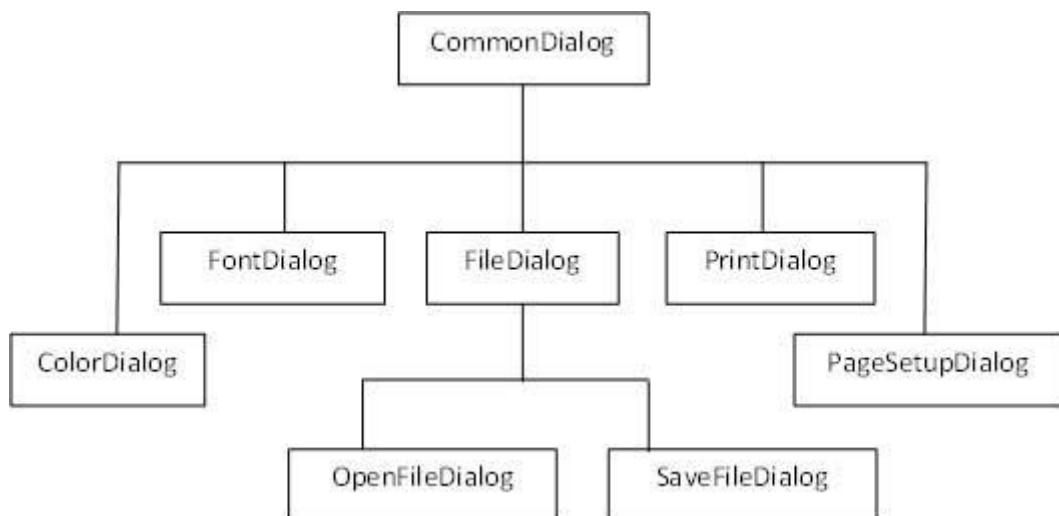
None - returns nothing and the dialog box continues running.

OK - returns **DialogResult.OK**, when user clicks an OK button

Retry - returns **DialogResult.Retry** , when user clicks a 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](#)

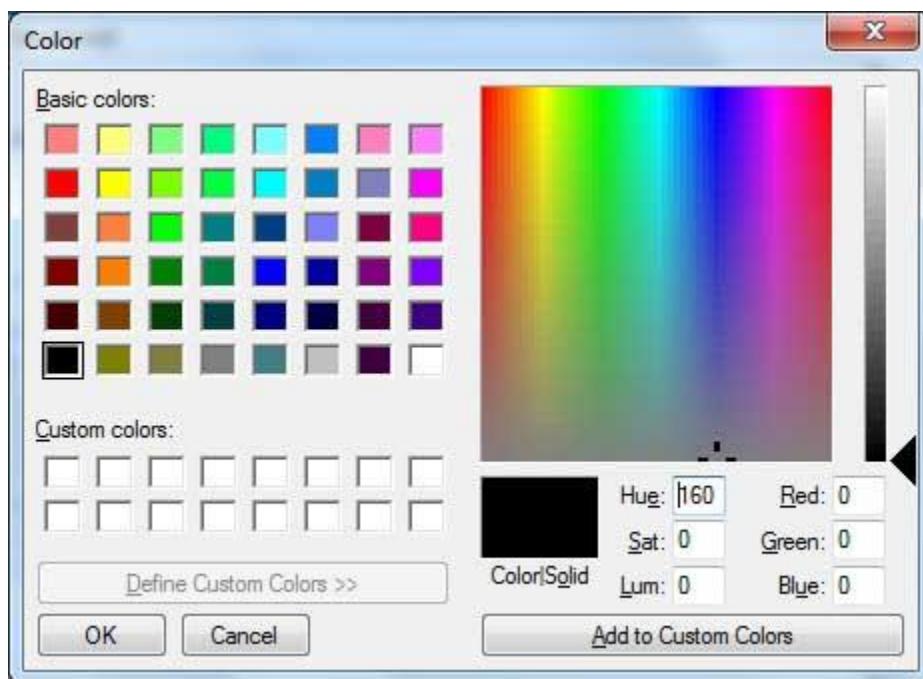
It lets the user to print documents by selecting a printer and choosing which sections of the document to print from a Windows Forms application.

ColorDialog Control

The **ColorDialog** control class represents a common dialog box that displays available colors along with controls that enable the user to define custom colors. It lets the user select a color.

The main property of the **ColorDialog** control is **Color**, which returns a **Color** object.

Following is the Color dialog box:



Properties of the ColorDialog Control

The following are some of the commonly used properties of the **ColorDialog** control:

S.N	Property	Description
1	AllowFullOpen	Gets or sets a value indicating whether the user can use the dialog box to define custom colors.

2	AnyColor	Gets or sets a value indicating whether the dialog box displays all available colors in the set of basic colors.
3	CanRaiseEvents	Gets a value indicating whether the component can raise an event.
4	Color	Gets or sets the color selected by the user.
5	CustomColors	Gets or sets the set of custom colors shown in the dialog box.
6	FullOpen	Gets or sets a value indicating whether the controls used to create custom colors are visible when the dialog box is opened
7	ShowHelp	Gets or sets a value indicating whether a Help button appears in the color dialog box.
8	SolidColorOnly	Gets or sets a value indicating whether the dialog box will restrict users to selecting solid colors only.

Methods of the ColorDialog Control

The following are some of the commonly used methods of the ColorDialog control:

S.N	Method Name & Description
1	Reset Resets all options to their default values, the last selected color to black, and the custom colors to their default values.
2	RunDialog When overridden in a derived class, specifies a common dialog box.
3	ShowDialog Runs a common dialog box with a default owner.

Events of the ColorDialog Control

The following are some of the commonly used events of the ColorDialog control:

S.N	Event	Description
1	HelpRequest	Occurs when the user clicks the Help button on a common dialog box.

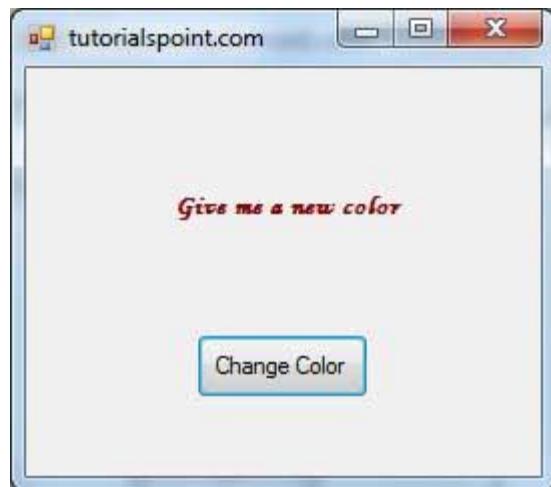
Example

In this example, let's change the forecolor of a label control using the color dialog box. Take the following steps:

- Drag and drop a label control, a button control and a ColorDialog control on the form.
- Set the Text property of the label and the button control to 'Give me a new Color' and 'Change Color', respectively.
- Change the font of the label as per your likings.
- Double-click the Change Color button and modify the code of the Click event.

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles
    Button1.Click
    If ColorDialog1.ShowDialog <> Windows.Forms.DialogResult.Cancel Then
        Label1.ForeColor = ColorDialog1.Color
    End If
End Sub
```

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

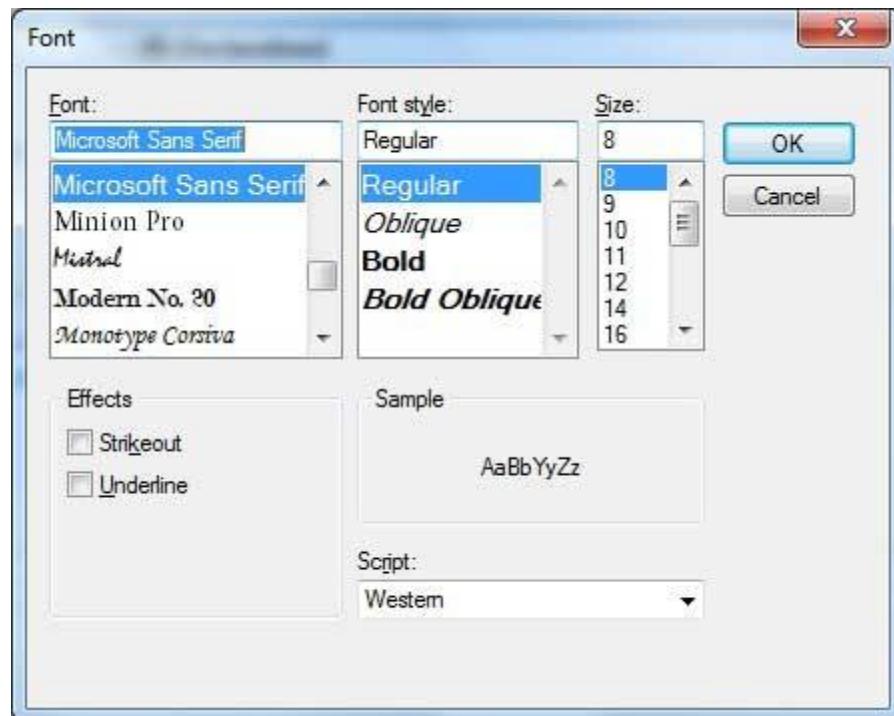


Clicking on the Change Color button, the color dialog appears, select a color and click the OK button. The selected color will be applied as the forecolor of the text of the label.

FontDialog Control

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. It returns the Font and Color objects.

Following is the Font dialog box:



By default, the Color ComboBox is not shown on the Font dialog box. You should set the **ShowColor** property of the FontDialog control to be **True**.

Properties of the FontDialog Control

The following are some of the commonly used properties of the FontDialog control:

S.N	Property	Description
1	AllowSimulations	Gets or sets a value indicating whether the dialog box allows graphics device interface (GDI) font simulations.
2	AllowVectorFonts	Gets or sets a value indicating whether the dialog box allows vector font selections.
3	AllowVerticalFonts	Gets or sets a value indicating whether the dialog box displays both vertical and horizontal fonts, or only horizontal fonts.
4	Color	Gets or sets the selected font color.
5	FixedPitchOnly	Gets or sets a value indicating whether the dialog box allows only the selection of fixed-pitch fonts.
6	Font	Gets or sets the selected font.
7	FontMustExist	Gets or sets a value indicating whether the dialog box specifies an error condition if the user attempts to select a font or style that does not exist.
8	MaxSize	Gets or sets the maximum point size a user can select.
9	MinSize	Gets or sets the minimum point size a user can select.
10	ScriptsOnly	Gets or sets a value indicating whether the dialog box allows selection of fonts for all non-OEM and Symbol character sets, as well as the ANSI character set.
11	ShowApply	Gets or sets a value indicating whether the dialog box contains an Apply button.

12	ShowColor	Gets or sets a value indicating whether the dialog box displays the color choice.
13	ShowEffects	Gets or sets a value indicating whether the dialog box contains controls that allow the user to specify strikethrough, underline, and text color options.
14	ShowHelp	Gets or sets a value indicating whether the dialog box displays a Help button.

Methods of the FontDialog Control

The following are some of the commonly used methods of the FontDialog control:

S.N	Method Name & Description
1	Reset Resets all options to their default values.
2	RunDialog When overridden in a derived class, specifies a common dialog box.
3	ShowDialog Runs a common dialog box with a default owner.

Events of the FontDialog Control

The following are some of the commonly used events of the FontDialog control:

S.N	Event	Description
1	Apply	Occurs when the Apply button on the font dialog box is clicked.

Example

In this example, let's change the font and color of the text from a rich text control using the Font dialog box. Take the following steps:

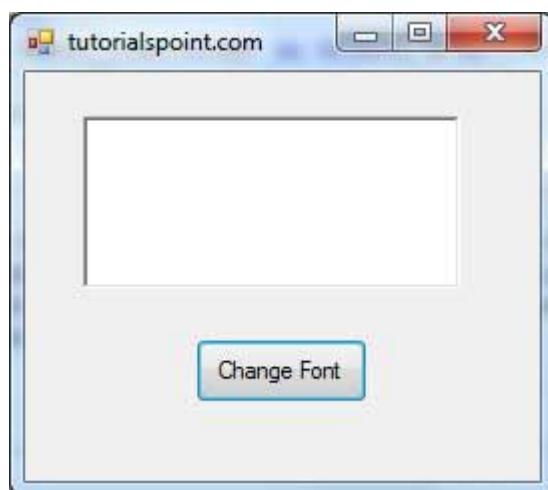
- Drag and drop a RichTextBox control, a Button control and a FontDialog control on the form.

- Set the Text property of the button control to 'Change Font'.
- Set the ShowColor property of the FontDialog control to True.

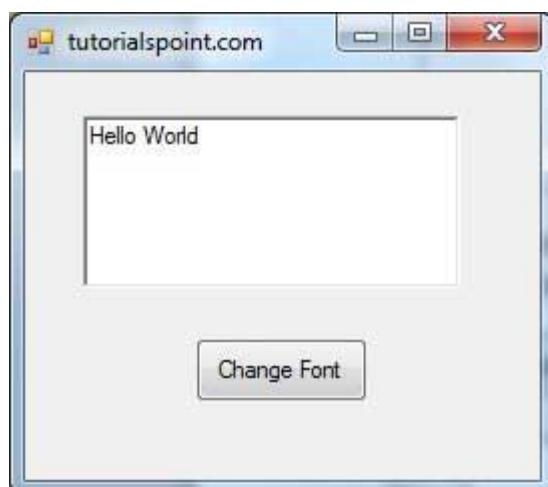
Double-click the Change Color button and modify the code of the Click event:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles
Button1.Click
    If FontDialog1.ShowDialog <> Windows.Forms.DialogResult.Cancel Then
        RichTextBox1.ForeColor = FontDialog1.Color
        RichTextBox1.Font = FontDialog1.Font
    End If
End Sub
```

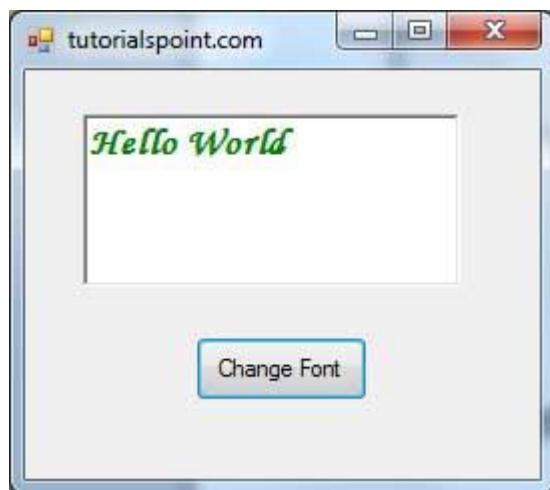
When the application is compiled and run using **Start** button available at the Microsoft Visual Studio tool bar, it will show the following window:



Enter some text and Click on the Change Font button.



The Font dialog appears, select a font and a color and click the OK button. The selected font and color will be applied as the font and fore color of the text of the rich text box.

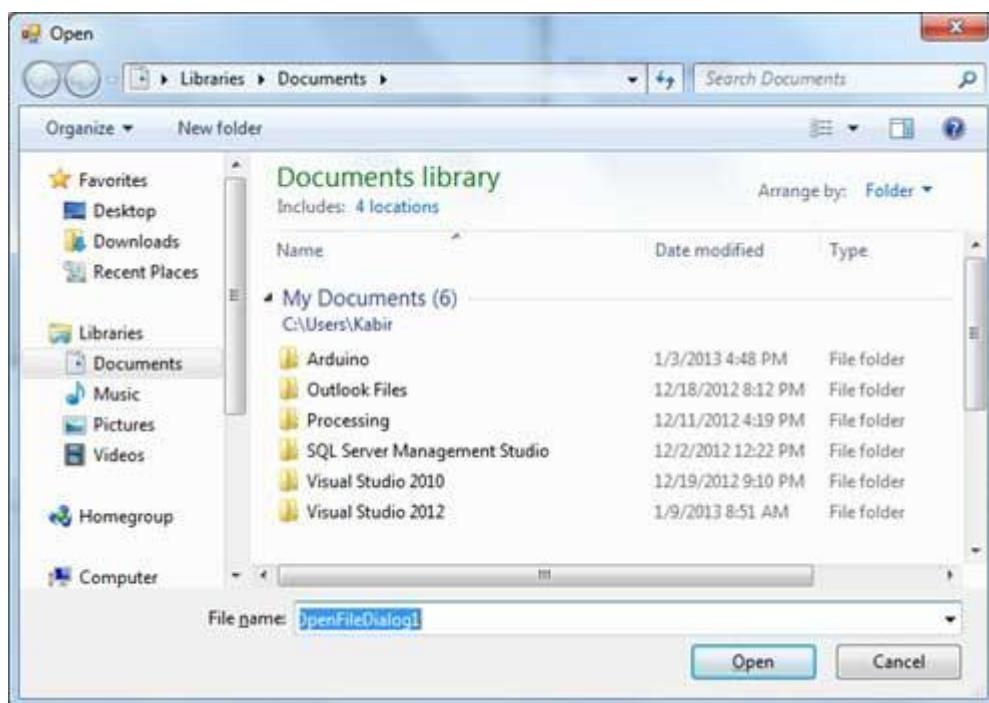


OpenFileDialog Control

The **OpenFileDialog** control prompts the user to open a file and allows the user to select a file to open. The user can check if the file exists and then open it. The OpenFileDialog control class inherits from the abstract class **FileDialog**.

If the ShowReadOnly property is set to True, then a read-only check box appears in the dialog box. You can also set the ReadOnlyChecked property to True, so that the read-only check box appears checked.

Following is the Open File dialog box:



Properties of the OpenFileDialog Control

The following are some of the commonly used properties of the OpenFileDialog control:

S.N	Property	Description
1	AddExtension	Gets or sets a value indicating whether the dialog box automatically adds an extension to a file name if the user omits the extension.
2	AutoUpgradeEnabled	Gets or sets a value indicating whether this FileDialog instance should automatically upgrade appearance and behavior when running on Windows Vista.
3	CheckFileExists	Gets or sets a value indicating whether the dialog box displays a warning if the user specifies a file name that does not exist.
4	CheckPathExists	Gets or sets a value indicating whether the dialog box displays a warning if the user specifies a path that does not exist.
5	CustomPlaces	Gets the custom places collection for this FileDialog instance.
6	DefaultExt	Gets or sets the default file name extension.
7	DereferenceLinks	Gets or sets a value indicating whether the dialog box returns the location of the file referenced by the shortcut or whether it returns the location of the shortcut (.lnk).
8	FileName	Gets or sets a string containing the file name selected in the file dialog box.

9	FileNames	Gets the file names of all selected files in the dialog box.
10	Filter	Gets or sets the current file name filter string, which determines the choices that appear in the "Save as file type" or "Files of type" box in the dialog box.
11	FilterIndex	Gets or sets the index of the filter currently selected in the file dialog box.
12	InitialDirectory	Gets or sets the initial directory displayed by the file dialog box.
13	Multiselect	Gets or sets a value indicating whether the dialog box allows multiple files to be selected.
14	ReadOnlyChecked	Gets or sets a value indicating whether the read-only check box is selected.
15	RestoreDirectory	Gets or sets a value indicating whether the dialog box restores the current directory before closing.
16	SafeFileName	Gets the file name and extension for the file selected in the dialog box. The file name does not include the path.
17	SafeFileNames	Gets an array of file names and extensions for all the selected files in the dialog box. The file names do not include the path.
18	ShowHelp	Gets or sets a value indicating whether the Help button is displayed in the file dialog box.

19	ShowReadOnly	Gets or sets a value indicating whether the dialog box contains a read-only check box.
20	SupportMultiDottedExtensions	Gets or sets whether the dialog box supports displaying and saving files that have multiple file name extensions.
21	Title	Gets or sets the file dialog box title.
22	ValidateNames	Gets or sets a value indicating whether the dialog box accepts only valid Win32 file names.

Methods of the OpenFileDialog Control

The following are some of the commonly used methods of the OpenFileDialog control:

S.N	Method Name & Description
1	OpenFile Opens the file selected by the user, with read-only permission. The file is specified by the FileName property.
2	Reset Resets all options to their default value.

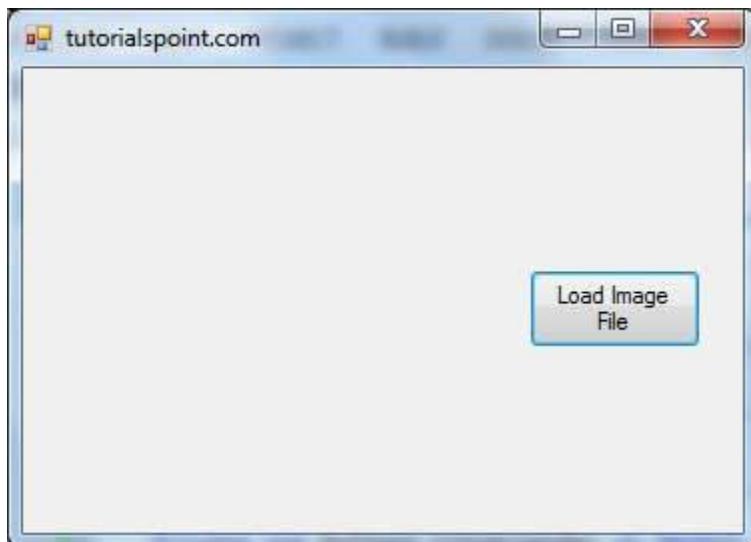
Example

In this example, let's load an image file in a picture box, using the open file dialog box. Take the following steps:

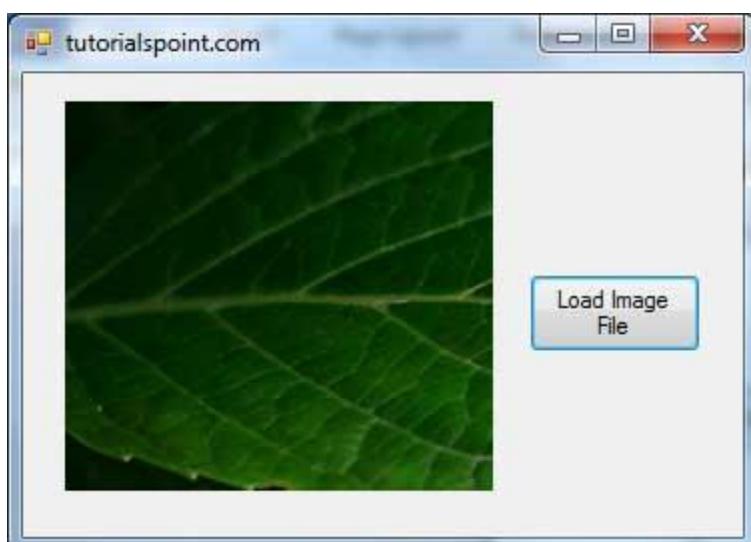
- Drag and drop a PictureBox control, a Button control and a OpenFileDialog control on the form.
- Set the Text property of the button control to 'Load Image File'.
- Double-click the Load Image File button and modify the code of the Click event.

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles
Button1.Click
    If OpenFileDialog1.ShowDialog <> Windows.Forms.DialogResult.Cancel
    Then
        PictureBox1.Image = Image.FromFile(OpenFileDialog1.FileName)
    End If
End Sub
```

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



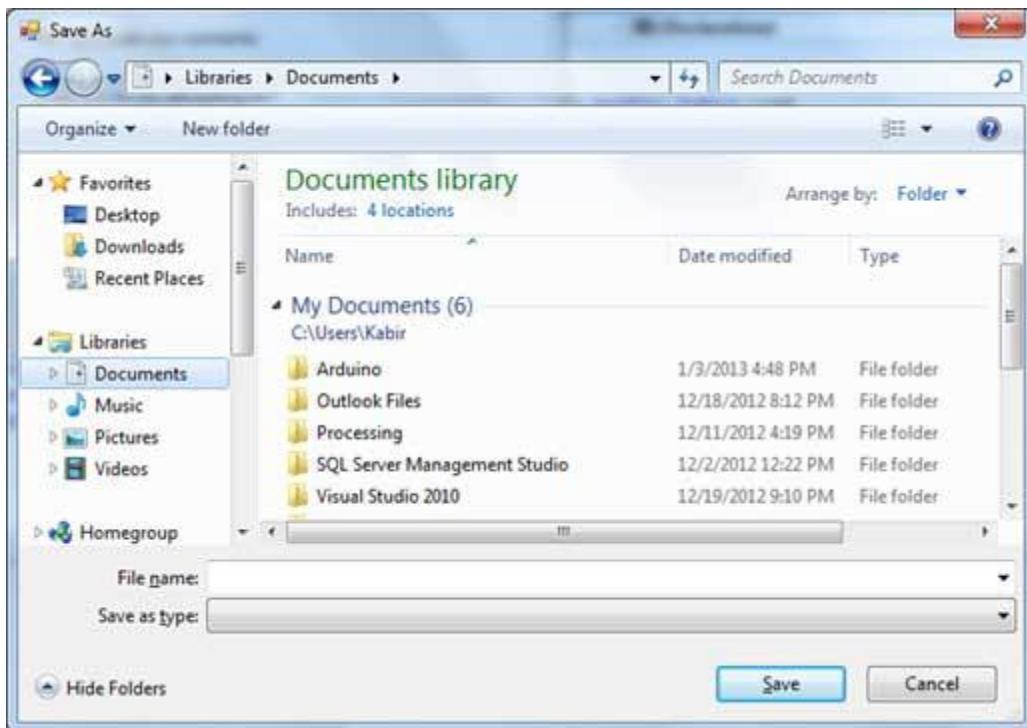
Click on the Load Image File button to load an image stored in your computer.



SaveFileDialog Control

The **SaveFileDialog** control 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. The SaveFileDialog control class inherits from the abstract class **FileDialog**.

Following is the Save File dialog box:



Properties of the SaveFileDialog Control

The following are some of the commonly used properties of the SaveFileDialog control:

S.N	Property	Description
1	AddExtension	Gets or sets a value indicating whether the dialog box automatically adds an extension to a file name if the user omits the extension.
2	CheckFileExists	Gets or sets a value indicating whether the dialog box displays a warning if the user specifies a file name that does not exist.

3	CheckPathExists	Gets or sets a value indicating whether the dialog box displays a warning if the user specifies a path that does not exist.
4	CreatePrompt	Gets or sets a value indicating whether the dialog box prompts the user for permission to create a file if the user specifies a file that does not exist.
5	DefaultExt	Gets or sets the default file name extension.
6	DereferenceLinks	Gets or sets a value indicating whether the dialog box returns the location of the file referenced by the shortcut or whether it returns the location of the shortcut (.lnk).
7	FileName	Gets or sets a string containing the file name selected in the file dialog box.
8	Filenames	Gets the file names of all selected files in the dialog box.
9	Filter	Gets or sets the current file name filter string, which determines the choices that appear in the "Save as file type" or "Files of type" box in the dialog box.
10	FilterIndex	Gets or sets the index of the filter currently selected in the file dialog box.
11	InitialDirectory	Gets or sets the initial directory displayed by the file dialog box.
12	OverwritePrompt	Gets or sets a value indicating whether the Save As dialog box

		displays a warning if the user specifies a file name that already exists.
13	RestoreDirectory	Gets or sets a value indicating whether the dialog box restores the current directory before closing.
14	ShowHelp	Gets or sets a value indicating whether the Help button is displayed in the file dialog box.
15	SupportMultiDottedExtensions	Gets or sets whether the dialog box supports displaying and saving files that have multiple file name extensions.
16	Title	Gets or sets the file dialog box title.
17	ValidateNames	Gets or sets a value indicating whether the dialog box accepts only valid Win32 file names.

Methods of the SaveFileDialog Control

The following are some of the commonly used methods of the SaveFileDialog control:

S.N	Method Name & Description
1	OpenFile Opens the file with read/write permission.
2	Reset Resets all dialog box options to their default values.

Example

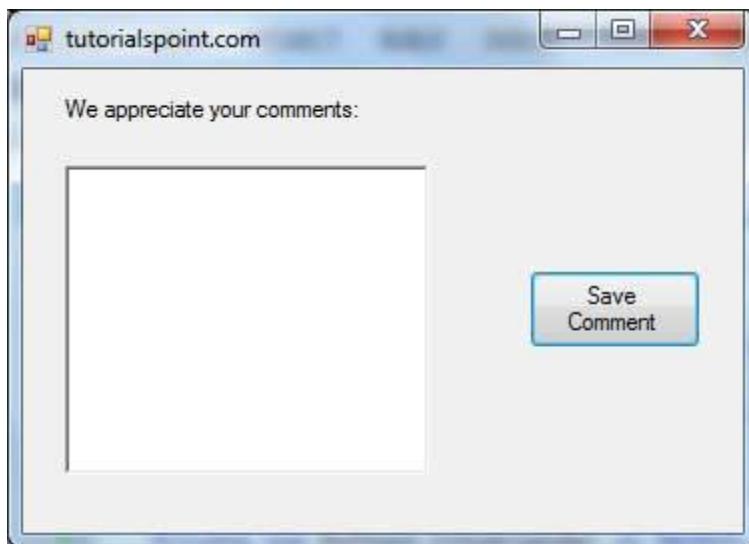
In this example, let's save the text entered into a rich text box by the user using the save file dialog box. Take the following steps:

- Drag and drop a Label control, a RichTextBox control, a Button control and a SaveFileDialog control on the form.

- Set the Text property of the label and the button control to 'We appreciate your comments' and 'Save Comments', respectively.
- Double-click the Save Comments button and modify the code of the Click event as shown:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles
Button1.Click
    SaveFileDialog1.Filter = "TXT Files (*.txt)|*.txt"
    If SaveFileDialog1.ShowDialog = Windows.Forms.DialogResult.OK _
    Then
        My.Computer.FileSystem.WriteAllText _
        (SaveFileDialog1.FileName, RichTextBox1.Text, True)
    End If
End Sub
```

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



We have set the Filter property of the SaveFileDialog control to display text file types with .txt extensions only.

Write some text in the text box and click on the Save Comment button to save the text as a text file in your computer.

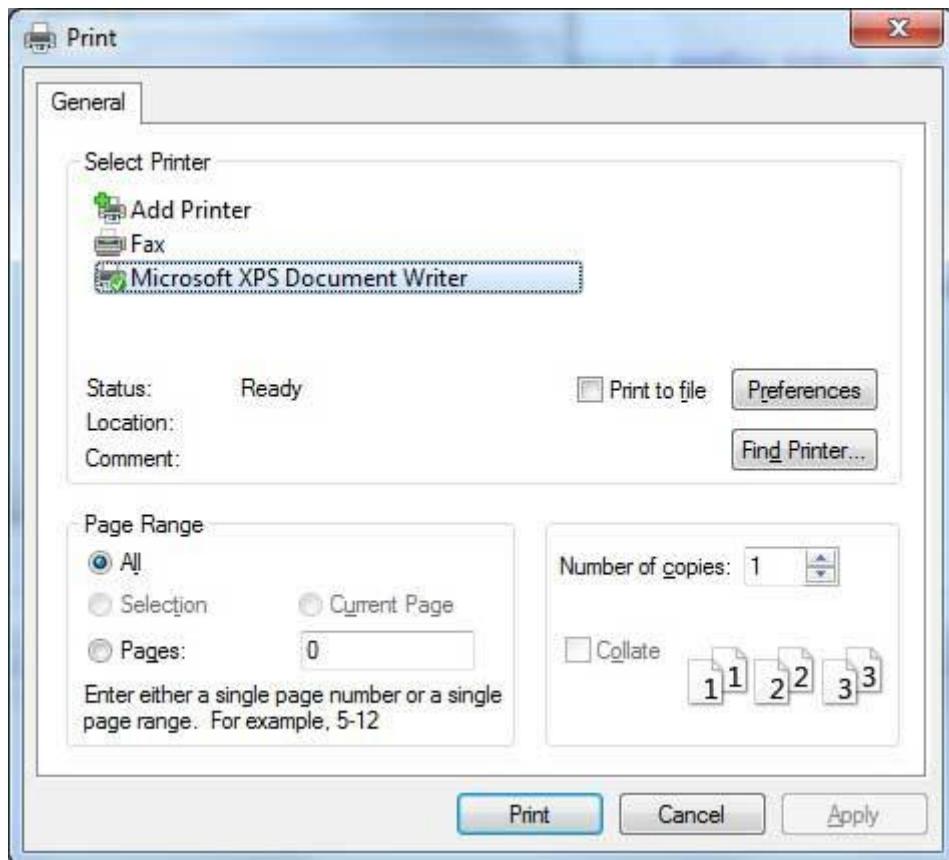
PrintDialog Control

The PrintDialog control lets the user to print documents by selecting a printer and choosing which sections of the document to print from a Windows Forms application.

There are various other controls related to printing of documents. Let us have a brief look at these controls and their purpose. These other controls are:

- The **PrintDocument** control - it provides support for actual events and operations of printing in Visual Basic and sets the properties for printing.
- The **PrinterSettings** control - it is used to configure how a document is printed by specifying the printer.
- The **PageSetUpDialog** control - it allows the user to specify page-related print settings including page orientation, paper size and margin size.
- The **PrintPreviewControl** control - it represents the raw preview part of print previewing from a Windows Forms application, without any dialog boxes or buttons.
- The **PrintPreviewDialog** control - it represents a dialog box form that contains a PrintPreviewControl for printing from a Windows Forms application.

Following is the Print dialog box:



Properties of the PrintDialog Control

The following are some of the commonly used properties of the PrintDialog control:

S.N	Property	Description
1	AllowCurrentPage	Gets or sets a value indicating whether the Current Pageoption button is displayed.
2	AllowPrintToFile	Gets or sets a value indicating whether the Print to filecheckbox is enabled.
3	AllowSelection	Gets or sets a value indicating whether the Selection option button is enabled.
4	AllowSomePages	Gets or sets a value indicating whether the Pages option button is enabled.
5	Document	Gets or sets a value indicating the PrintDocument used to obtain PrinterSettings.
6	PrinterSettings	Gets or sets the printer settings the dialog box modifies.
7	PrintToFile	Gets or sets a value indicating whether the Print to filecheckbox is selected.
8	ShowHelp	Gets or sets a value indicating whether the Help button is displayed.
9	ShowNetwork	Gets or sets a value indicating whether the Network button is displayed.

Methods of the PrintDialog Control

The following are some of the commonly used methods of the PrintDialog control:

S.N	Method Name & Description
1	Reset Resets all options to their default values.

2	RunDialog When overridden in a derived class, specifies a common dialog box.
3	ShowDialog Runs a common dialog box with a default owner.

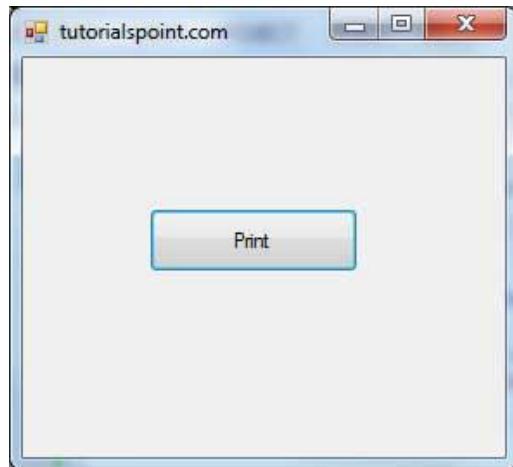
Example

In this example, let us see how to show a Print dialog box in a form. Take the following steps:

- Add a PrintDocument control, a PrintDialog control and a Button control on the form. The PrintDocument and the PrintDialog controls are found on the Print category of the controls toolbox.
- Change the text of the button to 'Print'.
- Double-click the Print button and modify the code of the Click event as shown:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles
Button1.Click
    PrintDialog1.Document = PrintDocument1
    PrintDialog1.PrinterSettings = PrintDocument1.PrinterSettings
    PrintDialog1.AllowSomePages = True
    If PrintDialog1.ShowDialog = DialogResult.OK Then
        PrintDocument1.PrinterSettings = PrintDialog1.PrinterSettings
        PrintDocument1.Print()
    End If
End Sub
```

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



- Click the Print button to make the Print dialog box appear.

25. Advanced Form

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 version 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 following 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")

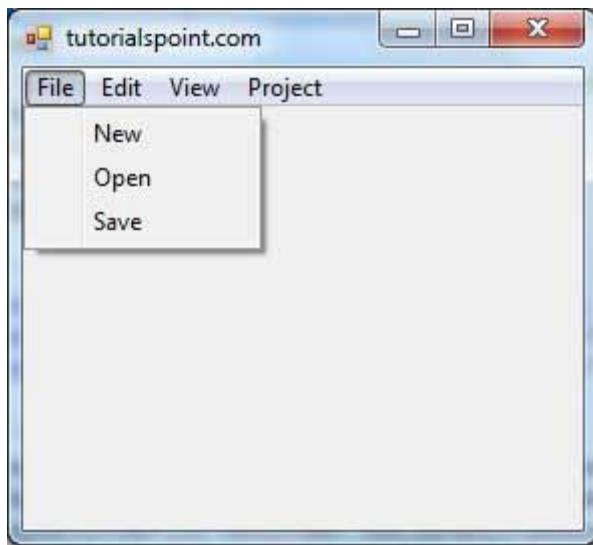
' add sub menus to the File menu
myMenuItemFile.MenuItems.Add(myMenuItemNew)
myMenuItemFile.MenuItems.Add(myMenuItemOpen)
myMenuItemFile.MenuItems.Add(myMenuItemSave)

' 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 the following window:



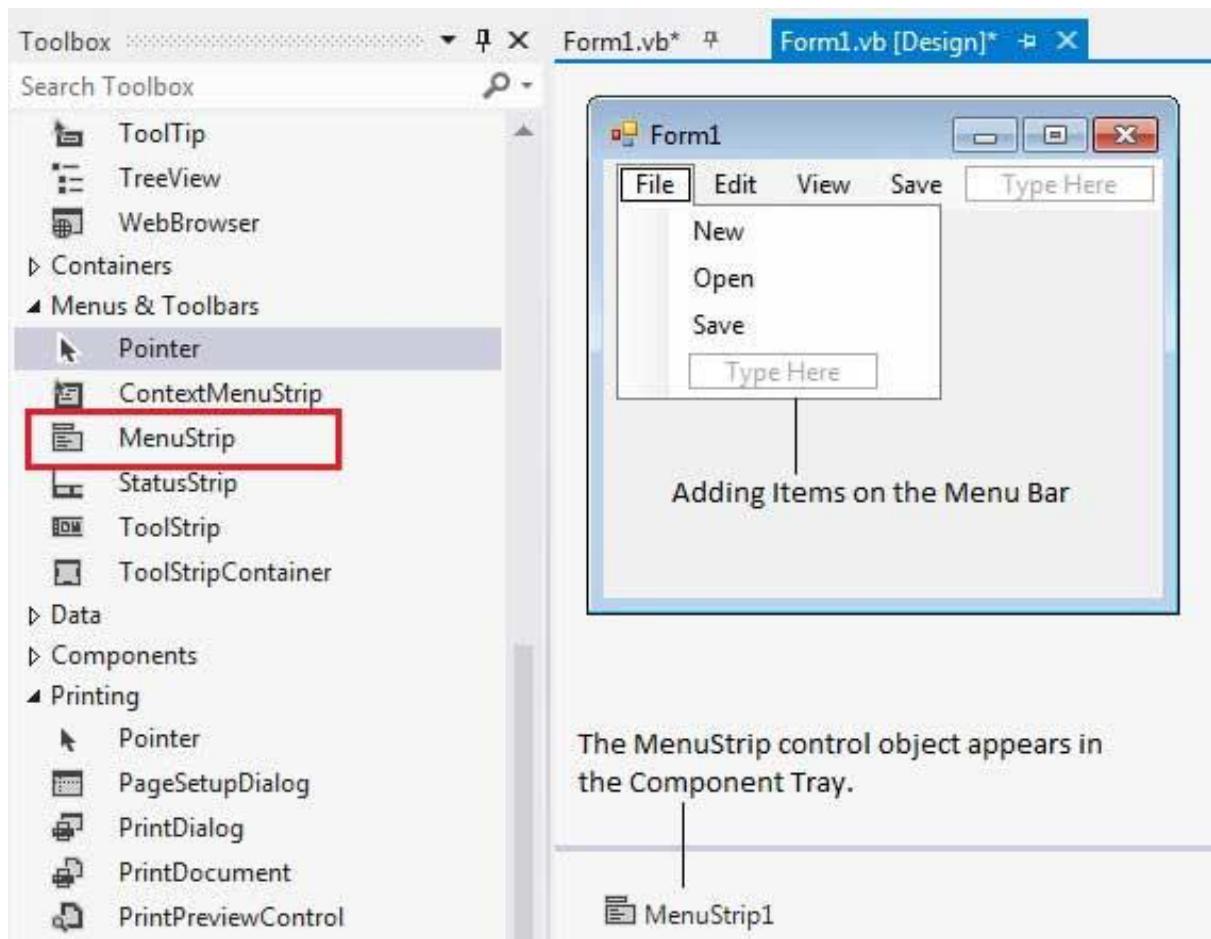
Windows Forms contain a rich set of classes for creating your own custom menus with modern appearance, look and feel. The **MenuStrip**, **ToolStripMenuItem**, **ContextMenuStrip** controls are used to create menu bars and context menus efficiently.

Click the following links to check their details:

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.
3	ContextMenuStrip It represents a shortcut menu.

MenuStrip Control

The **MenuStrip** control represents the container for the menu structure. The MenuStrip control works as the top-level container for the menu structure. The ToolStripMenuItem class and the ToolStripDropDownMenu class provide the functionalities to create menu items, sub menus and drop-down menus. The following diagram shows adding a MenuStrip control on the form:



Properties of the ToolStrip Control

The following are some of the commonly used properties of the ToolStrip control:

S.N	Property	Description
1	CanOverflow	Gets or sets a value indicating whether the ToolStrip supports overflow functionality.
2	GripStyle	Gets or sets the visibility of the grip used to reposition the control.
3	MdiWindowListItem	Gets or sets the ToolStripMenuItem that is used to display a list of Multiple-document interface (MDI) child forms.
4	ShowItemToolTips	Gets or sets a value indicating whether ToolTips are shown for the ToolStrip.

5	Stretch	Gets or sets a value indicating whether the ToolStrip stretches from end to end in its container.
---	----------------	---

Events of the ToolStrip Control

The following are some of the commonly used events of the ToolStrip control:

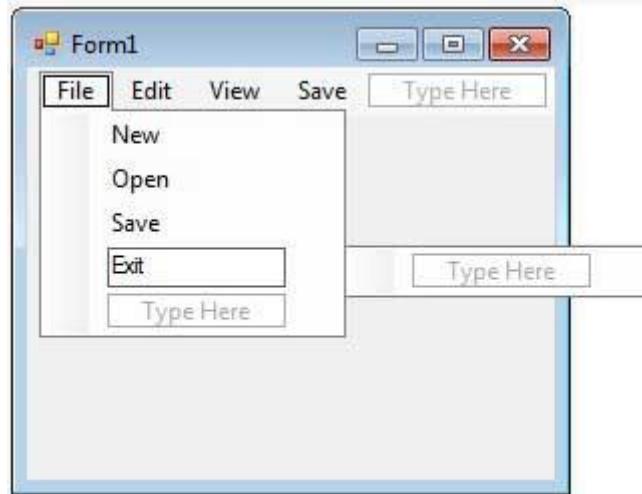
S.N	Event	Description
1	MenuActivate	Occurs when the user accesses the menu with the keyboard or mouse.
2	MenuDeactivate	Occurs when the ToolStrip is deactivated.

Example

In this example, let us add menu and sub-menu items.

Take the following steps:

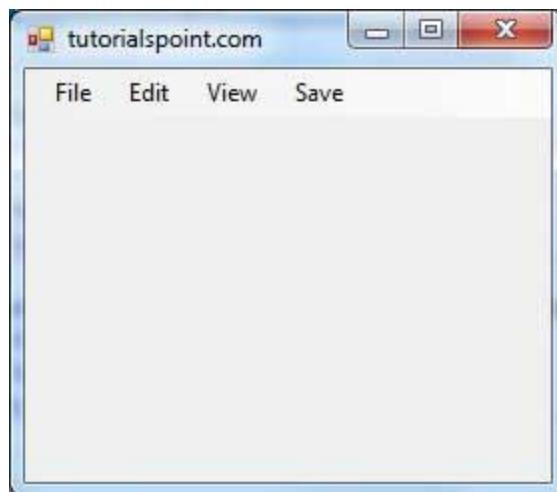
- Drag and drop or double click on a ToolStrip control, to add it to the form.
- Click the Type Here text to open a text box and enter the names of the menu items or sub-menu items you want. When you add a sub-menu, another text box with 'Type Here' text opens below it.
- Complete the menu structure shown in the diagram above.
- Add a sub menu **Exit** under the **File** menu.



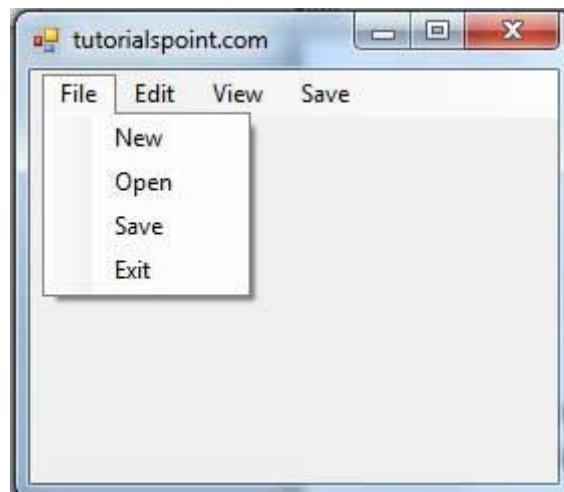
- Double-Click the Exit menu created and add the following code to the **Click** event of **ExitToolStripMenuItem**:

```
Private Sub ExitToolStripMenuItem_Click(sender As Object, e As EventArgs) _
Handles ExitToolStripMenuItem.Click
End
End Sub
```

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



Click on the File -> Exit to exit from the application:



StripMenuItem Control

The **ToolStripMenuItem** class supports the menus and menu items in a menu system. You handle these menu items through the click events in a menu system.

Properties of the ToolStripMenuItem Control

The following are some of the commonly used properties of the ToolStripMenuItem control:

S.N	Property	Description
1	Checked	Gets or sets a value indicating whether the ToolStripMenuItem is checked.
2	CheckOnClick	Gets or sets a value indicating whether the ToolStripMenuItem should automatically appear checked and unchecked when clicked.
3	CheckState	Gets or sets a value indicating whether a ToolStripMenuItem is in the checked, unchecked, or indeterminate state.
4	Enabled	Gets or sets a value indicating whether the control is enabled.
5	IsMdiWindowListEntry	Gets a value indicating whether the ToolStripMenuItem appears on a multiple document interface (MDI) window list.

6	ShortcutKeyDisplayString	Gets or sets the shortcut key text.
7	ShortcutKeys	Gets or sets the shortcut keys associated with the ToolStripMenuItem.
8	ShowShortcutKeys	Gets or sets a value indicating whether the shortcut keys that are associated with the ToolStripMenuItem are displayed next to the ToolStripMenuItem.

Events of the ToolStripMenuItem Control

The following are some of the commonly used events of the ToolStripMenuItem control:

S.N	Event	Description
1	CheckedChanged	Occurs when the value of the Checked property changes.
2	CheckStateChanged	Occurs when the value of the CheckState property changes.

Example

In this example, let us continue with the example from the chapter 'VB.Net - MenuStrip control'. Let us:

- Hide and display menu items.
- Disable and enable menu items.
- Set access keys for menu items
- Set shortcut keys for menu items.

Hide and Display Menu Items

The **Visible** property of the **ToolStripMenuItem** class allows you to hide or show a menu item. Let us hide the Project Menu on the menu bar.

Add the following code snippet to the Form1_Load event:

```
Private Sub Form1_Load(sender As Object, e As EventArgs) _
```

```

Handles MyBase.Load
    ' Hide the project menu
    ProjectToolStripMenuItem1.Visible = False
    ' Set the caption bar text of the form.
    Me.Text = "tutorialspoint.com"
End Sub

```

Add a button control on the form with text 'Show Project'.

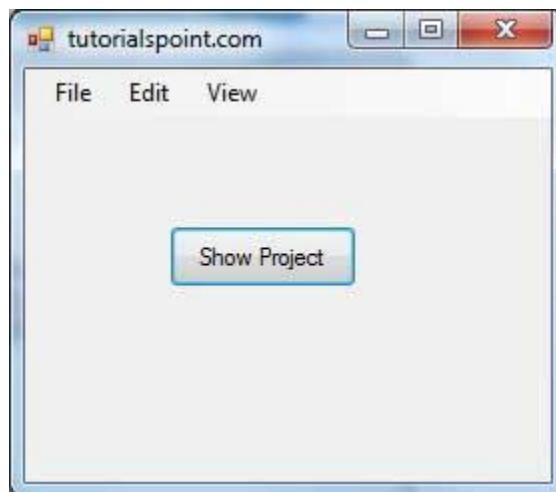
Add the following code snippet to the Button1_Click event:

```

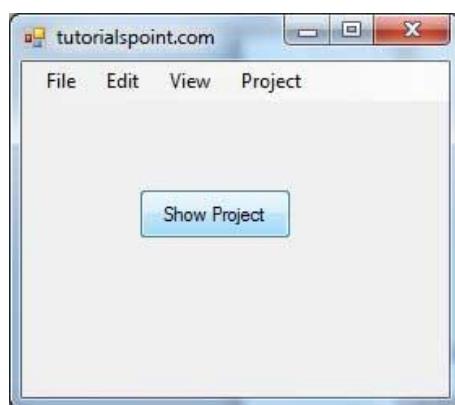
Private Sub Button1_Click(sender As Object, e As EventArgs) _
Handles Button1.Click
    ProjectToolStripMenuItem1.Visible = True
End Sub

```

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



Clicking on the Show Project button displays the project menu:



Disable and Enable Menu Items

The **Enabled** property allows you to disable or gray out a menu item. Let us disable the Project Menu on the menu bar.

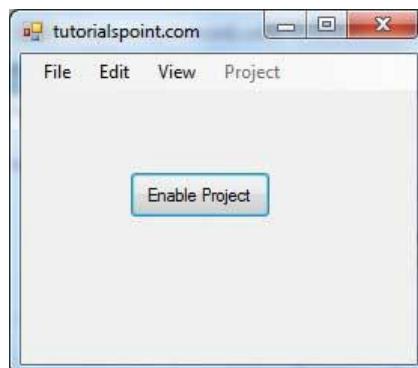
Add the following code snippet to the Form1_Load event:

```
Private Sub Form1_Load(sender As Object, e As EventArgs) _
Handles MyBase.Load
    ' Disable the project menu
    ProjectToolStripMenuItem1.Enabled = False
    ' Set the caption bar text of the form.
    Me.Text = "tutorialspoint.com"
End Sub
```

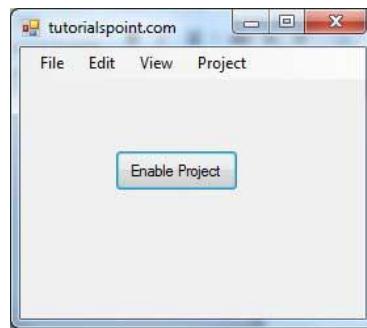
Add a button control on the form with text 'Enable Project'. Add the following code snippet to the Button1_Click event:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) _
Handles Button1.Click
    ProjectToolStripMenuItem1.Enabled = True
End Sub
```

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



Clicking on the Enable Project button enables the project menu:



Set Access Keys for Menu Items

Setting access keys for a menu allows a user to select it from the keyboard by using the ALT key.

For example, if you want to set an access key ALT + F for the file menu, change its **Text** with an added & (ampersand) preceding the access key letter. In other words, you change the text property of the file menu to &File.

Properties	
FileToolStripMenuItem	System.Windows.Forms.
<input type="checkbox"/> Padding	4, 0, 4, 0
RightToLeft	No
RightToLeftAutoMirrorImage	False
ShortcutKeyDisplayString	
ShortcutKeys	None
ShowShortcutKeys	True
<input type="checkbox"/> Size	37, 20
Tag	
Text	&File
Text The text to display on the item.	

Set Shortcut Keys for Menu Items

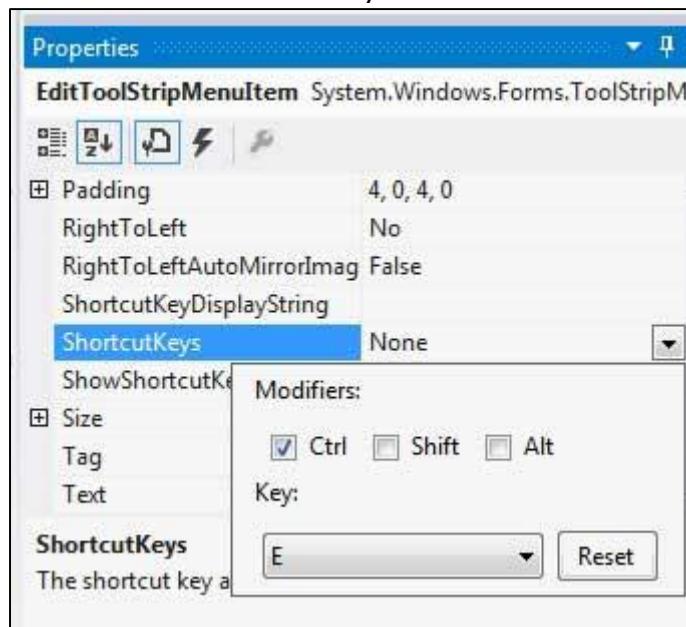
When you set a shortcut key for a menu item, user can press the shortcut from the keyboard and it would result in occurrence of the Click event of the menu.

A shortcut key is set for a menu item using the **ShortcutKeys** property. For example, to set a shortcut key CTRL + E, for the Edit menu:

Select the Edit menu item and select its ShortcutKeys property in the properties window.

Click the drop down button next to it.

Select Ctrl as Modifier and E as the key.



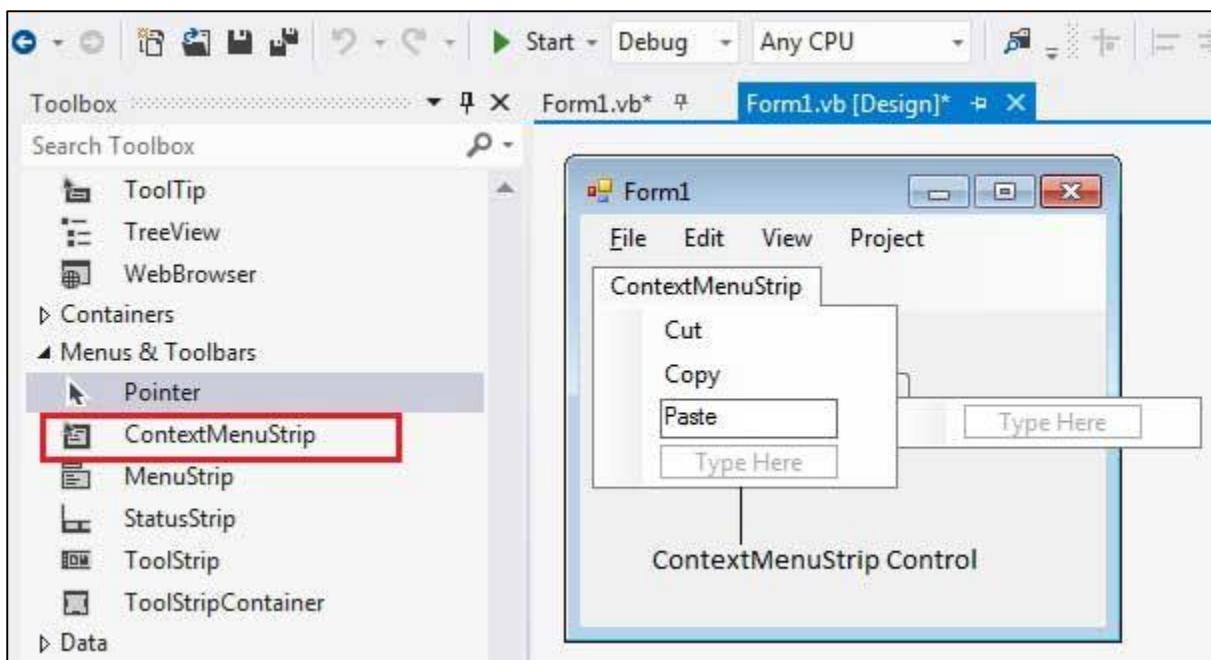
ContextMenuStrip Control

The **ContextMenuStrip** control represents a shortcut menu that pops up over controls, usually when you right click them. They appear in context of some specific controls, so are called context menus. For example, Cut, Copy or Paste options.

This control associates the context menu with other menu items by setting that menu item's ContextMenuStrip property to the ContextMenuStrip control you designed.

Context menu items can also be disabled, hidden, or deleted. You can also show a context menu with the help of the Show method of the ContextMenuStrip control.

The following diagram shows adding a ContextMenuStrip control on the form:



Properties of the ContextMenuStrip Control

The following are some of the commonly used properties of the ContextMenuStrip control:

S.N	Property	Description
1	SourceControl	Gets the last control that displayed the ContextMenuStrip control.

Example

In this example, let us add a content menu with the menu items Cut, Copy and Paste.

Take the following steps:

- Drag and drop or double click on a ContextMenuStrip control to add it to the form.
- Add the menu items, Cut, Copy and Paste to it.
- Add a RichTextBox control on the form.
- Set the ContextMenuStrip property of the rich text box to ContextMenuStrip1 using the properties window.

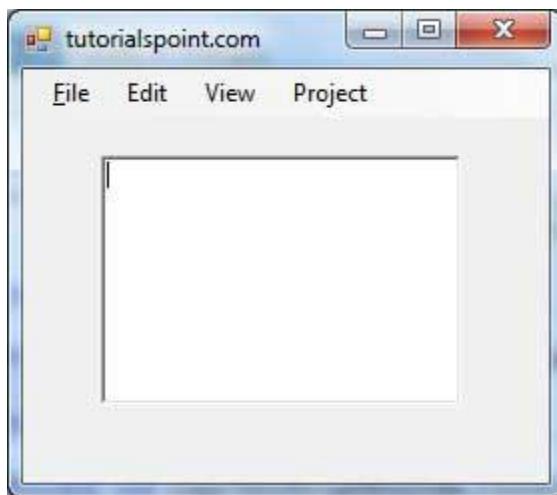
- Double the menu items and add following codes in the Click event of these menus:

```
Private Sub CutToolStripMenuItem_Click(sender As Object, e As EventArgs)
    Handles CutToolStripMenuItem.Click
    RichTextBox1.Cut()
End Sub

Private Sub CopyToolStripMenuItem_Click(sender As Object, e As EventArgs) _
    Handles CopyToolStripMenuItem.Click
    RichTextBox1.Copy()
End Sub

Private Sub PasteToolStripMenuItem_Click(sender As Object, e As EventArgs) _
    Handles PasteToolStripMenuItem.Click
    RichTextBox1.Paste()
End Sub
```

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



Enter some text in the rich text box, select it and right-click to get the context menu appear:



Now, you can select any menu items and perform cut, copy or paste on the text box.

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:

- Add a rich text box control and three button controls on the form.
- Change the text property of the buttons to Cut, Copy and Paste, respectively.
- 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.
    End Sub
End Class
```

```

Me.Text = "tutorialspoint.com"
End Sub

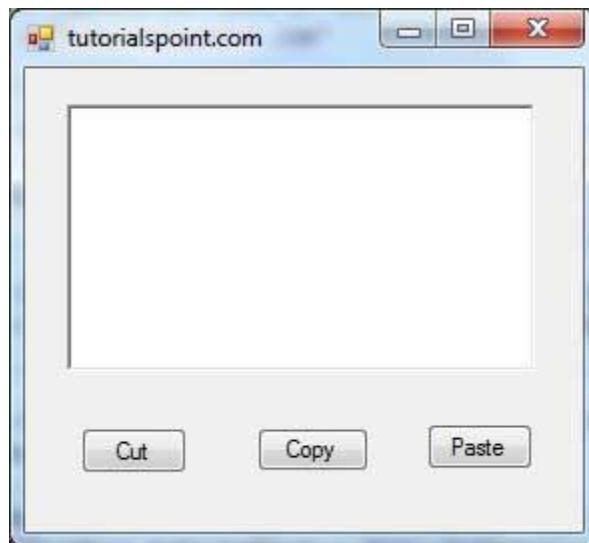
Private Sub Button1_Click(sender As Object, e As EventArgs) _
Handles Button1.Click
Clipboard.SetDataObject(RichTextBox1.SelectedText)
RichTextBox1.SelectedText = ""
End Sub

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 the following window:



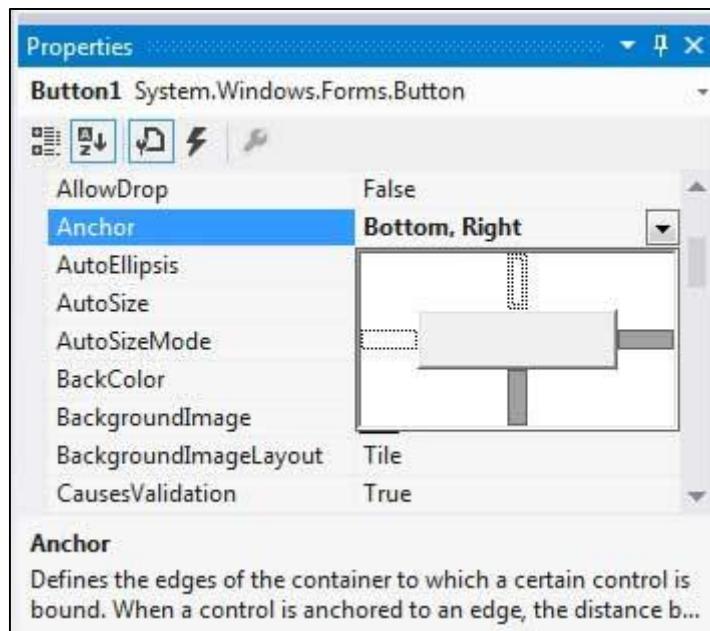
- Enter some text and check how the buttons work.

Anchoring and Docking Controls in a Form

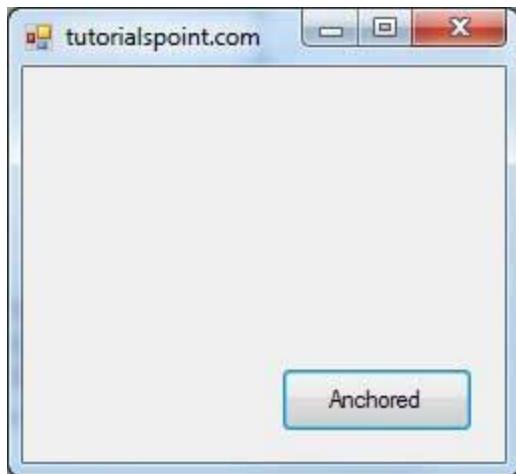
Anchoring allows you to set an anchor position for a control to the edges of its container control, for example, the form. The **Anchor** property of the Control class allows you to set values of this property. The Anchor property gets or sets the edges of the container to which a control is bound and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

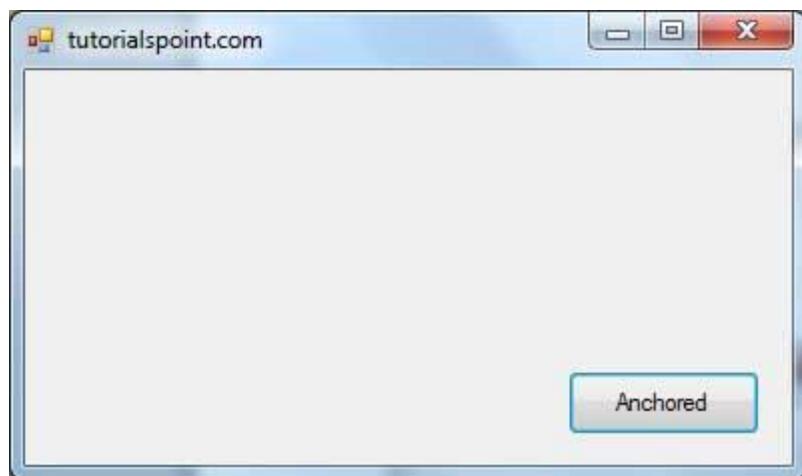
You can set the Anchor property values of a control from the Properties window:



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.



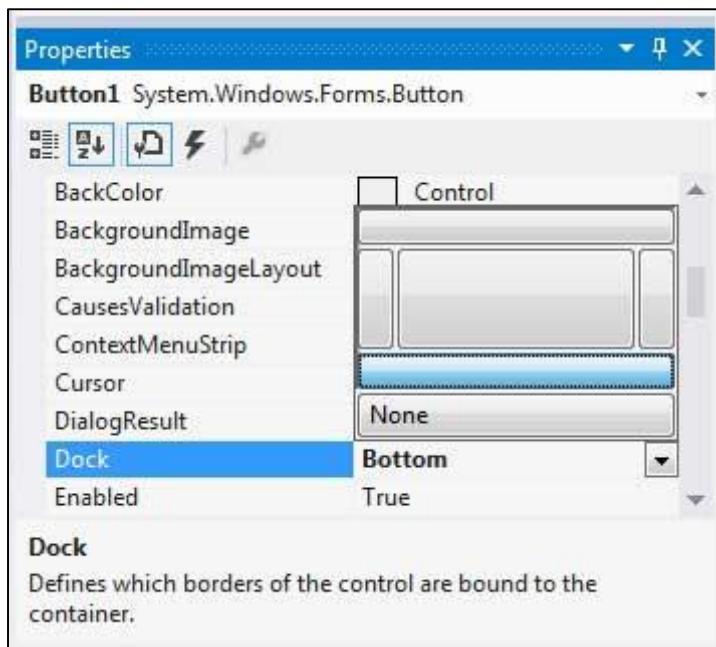
Now, when you stretch the form, the distance between the Button and the bottom right corner of the form remains same.



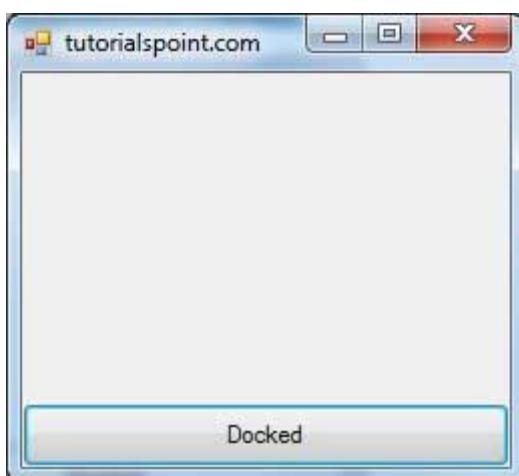
Docking of a control means docking it to one of the edges of its container. In docking, the control fills certain area of the container completely.

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.

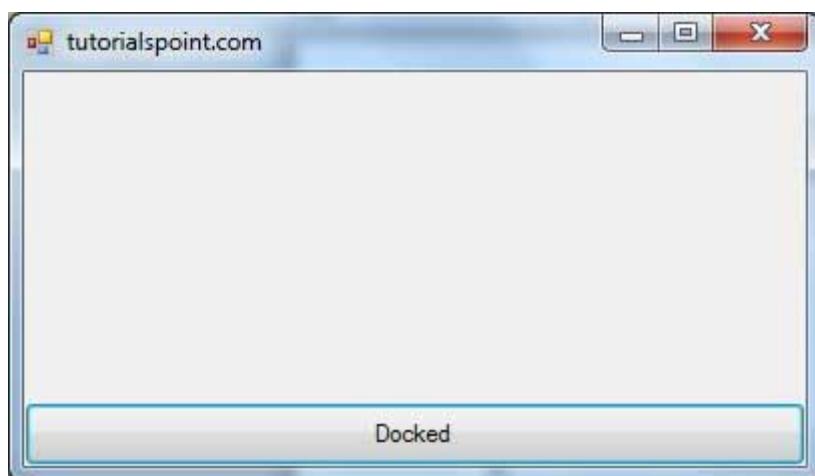
You can set the Dock property values of a control from the Properties window:



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

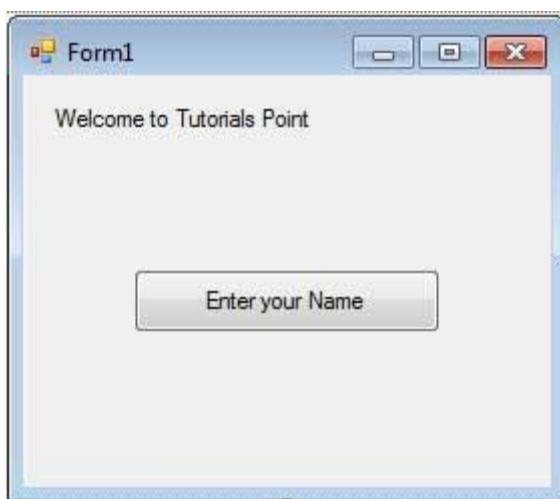
Modal Forms are those forms that need to be closed or hidden before you can continue working with the rest of the application. All dialog boxes are modal forms. A MessageBox is also a modal form.

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:

- Add a form, Form1 to your application, and add two labels and a button control to Form1
- 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.



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

- 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 the following window:



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



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



26. 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:

- Add three labels, three text boxes and a button control in the form.
- Change the text properties of the labels to - Customer ID, Name and Address, respectively.
- Change the name properties of the text boxes to txtID, txtName and txtAddress, respectively.
- Change the text property of the button to 'Submit'.
- 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 = "tutorialspoint.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

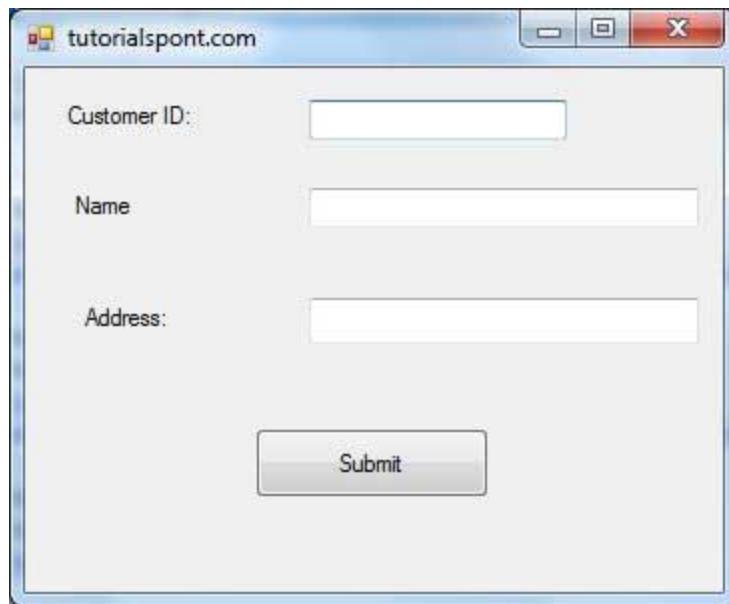
    MsgBox("Thank you " & txtName.Text & ", for your kind
cooperation")

End Sub

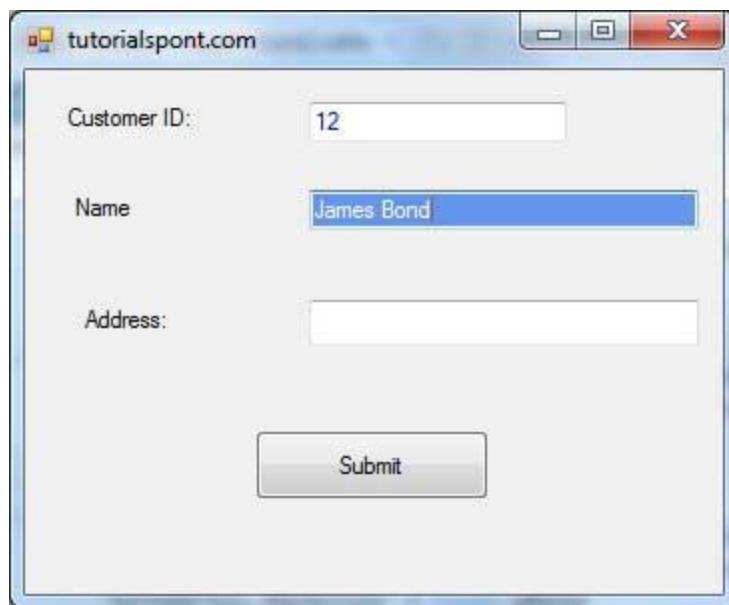
End Class

```

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

- Add a label with text Property as 'Age' and add a corresponding text box named txtAge.
- Add the following codes for handling the KeyUP events 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
```

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

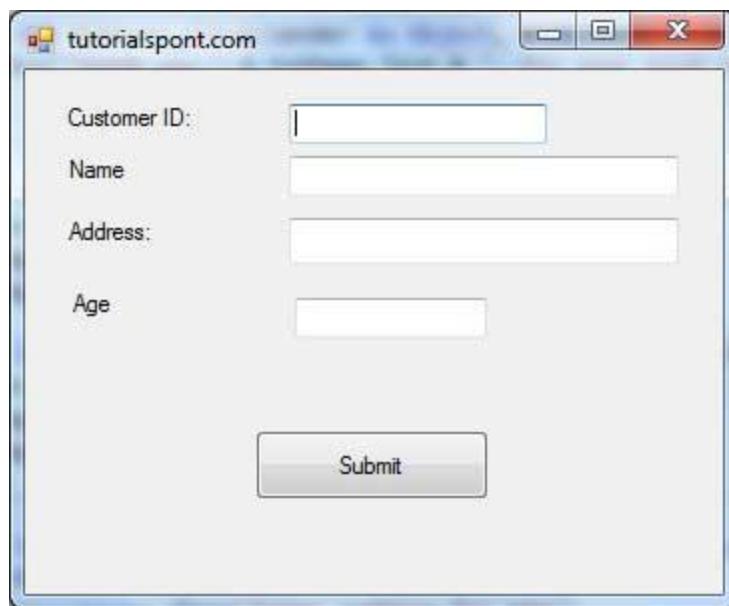
```
Private Sub txtAge_KeyUp(sender As Object, e As KeyEventArgs) _
```

```

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 the 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:



27. 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 following links to find these constructs.

[Character escapes](#)

[Character classes](#)

[Anchors](#)

[Grouping constructs](#)

[Quantifiers](#)

[Backreference constructs](#)

[Alternation constructs](#)

[Substitutions](#)

[Miscellaneous constructs](#)

Character Escapes

These are basically the special characters or escape characters. The backslash character (\) in a regular expression indicates that the character that follows it either is a special character or should be interpreted literally. The following table lists the escape characters:

Escaped character	Description	Pattern	Matches
\a	Matches a bell character, \u0007.	\a	"\u0007" in "Warning!" + '\u0007'

\b	In a character class, matches a backspace, \u0008.	[\b]{3,}	"\b\b\b\b" in "\b\b\b\b"
\t	Matches a tab, \u0009.	(\w+)\t	"Name\t", "Addr\t" in "Name\tAddr\t"
\r	Matches a carriage return, \u000D. (\r is not equivalent to the newline character, \n.)	\r\n(\w+)	"\r\nHello" in "\rHello\nWorld."
\v	Matches a vertical tab, \u000B.	[\v]{2,}	"\v\v\v" in "\v\v\v"
\f	Matches a form feed, \u000C.	[\f]{2,}	"\f\f\f" in "\f\f\f"
\n	Matches a new line, \u000A.	\r\n(\w+)	"\r\nHello" in "\rHello\nWorld."
\e	Matches an escape, \u001B.	\e	"\x001B" in "\x001B"
\nnn	Uses octal representation to specify a character (nnn consists of up to three digits).	\w\040\w	"a b", "c d" in "a bc d"
\x nn	Uses hexadecimal representation to specify a character (nn consists of exactly two digits).	\w\x20\w	"a b", "c d" in "a bc d"
\c X\c x	Matches the ASCII control character that is specified by X or x, where X or x is the letter of the control character.	\cC	"\x0003" in "\x0003" (Ctrl-C)

\u nnnn	Matches a Unicode character by using hexadecimal representation (exactly four digits, as represented by nnnn).	\w\u0020\w	"a b", "c d" in "a bc d"
\	When followed by a character that is not recognized as an escaped character, matches that character.	\d+[\+-x*]\d+\d+[\+-x*]\d+	"2+2" and "3*9" in "(2+2) * 3*9"

Character Classes

A character class matches any one of a set of characters. The following table describes the character classes:

Character class	Description	Pattern	Matches
[character_group]	Matches any single character in character_group. By default, the match is case-sensitive.	[mn]	"m" in "mat" "m", "n" in "moon"
[^character_group]	Negation: Matches any single character that is not in character_group. By default, characters in character_group are case-sensitive.	[^aei]	"v", "l" in "avail"
[first - last]	Character range: Matches any single character in the range from first to last.	(\w+)\t	"Name\t", "Addr\t" in "Name\tAddr\t"
.	Wildcard: Matches any single character except \n.	a.e	"ave" in "have" "ate" in "mate"

\p{ name }	Matches any single character in the Unicode general category or named block specified by <i>name</i> .	\p{Lu}	"C", "L" in "City Lights"
\P{ name }	Matches any single character that is not in the Unicode general category or named block specified by <i>name</i> .	\P{Lu}	"i", "t", "y" in "City"
\w	Matches any word character.	\w	"R", "o", "m" and "1" in "Room#1"
\W	Matches any non-word character.	\W	"#" in "Room#1"
\s	Matches any white-space character.	\w\s	"D " in "ID A1.3"
\S	Matches any non-white-space character.	\s\S	"_" in "int __ctr"
\d	Matches any decimal digit.	\d	"4" in "4 = IV"
\D	Matches any character other than a decimal digit.	\D	" ", "=", " ", "I", "V" in "4 = IV"

Anchors

Anchors allow a match to succeed or fail depending on the current position in the string. The following table lists the anchors:

Assertion	Description	Pattern	Matches
^	The match must start at the beginning of the string or line.	<code>^\d{3}</code>	"567" in "567-777-"

\$	The match must occur at the end of the string or before \n at the end of the line or string.	-\d{4}\$	"-2012" in "8-12-2012"
\A	The match must occur at the start of the string.	\A\w{3}	"Code" in "Code-007-"
\Z	The match must occur at the end of the string or before \n at the end of the string.	-\d{3}\Z	"-007" in "Bond-901-007"
\z	The match must occur at the end of the string.	-\d{3}\z	"-333" in "-901-333"
\G	The match must occur at the point, where the previous match ended.	\G\(\d\)	"(1)", "(3)", "(5)" in "(1)(3)(5)[7](9)"
\b	The match must occur on a boundary between a \w(alphanumeric) and a \W (non-alphanumeric) character.	\w	"R", "o", "m" and "1" in "Room#1"
\B	The match must not occur on a\b boundary.	\Bend\w*\b	"ends", "ender" in "end sends endure lender"

Grouping Constructs

Grouping constructs delineate sub-expressions of a regular expression and capture substrings of an input string. The following table lists the grouping constructs:

Grouping construct	Description	Pattern	Matches
(subexpression)	Captures the matched subexpression	(\w)\1	"ee" in "deep"

	and assigns it a zero-based ordinal number.		
(?< name >subexpression)	Captures the matched subexpression into a named group.	(?< double>\w)\k< double>	"ee" in "deep"
(?< name1 - name2 >subexpression)	Defines a balancing group definition.	((('Open'\()[^(\())*)+((?'Close-Open'\())[^\(\)]*)+)*?(?'Open')(?!)\$	"((1-3)*(3-1))" in "3+2^((1-3)*(3-1))"
(?: subexpression)	Defines a noncapturing group.	Write(?:Line)?	"WriteLine" in "Console.WriteLine()"
(?imnsx-imnsx:subexpression)	Applies or disables the specified options within <i>subexpression</i> .	A\d{2}(?'i:\w+)\b	"A12xl", "A12XL" in "A12xl A12XL a12xl"
(?= subexpression)	Zero-width positive lookahead assertion.	\w+(?=.)	"is", "ran", and "out" in "He is. The dog ran. The sun is out."
(?! subexpression)	Zero-width negative lookahead assertion.	\b(?!un)\w+\b	"sure", "used" in "unsure sure unity used"
(?< =subexpression)	Zero-width positive lookbehind assertion.	(?< =19)\d{2}\b	"51", "03" in "1851 1999 1950 1905 2003"

(?< subexpression)	!	Zero-width negative lookbehind assertion.	(?< !19)\d{2}\b	"ends", "ender" in "end sends endure lender"
(?> subexpression)		Nonbacktracking (or "greedy") subexpression.	[13579](?>A+B+)	"1ABB", "3ABB", and "5AB" in "1ABB 3ABBC 5AB 5AC"

Quantifiers

Quantifiers specify how many instances of the previous element (which can be a character, a group, or a character class) must be present in the input string for a match to occur.

Quantifier	Description	Pattern	Matches
*	Matches the previous element zero or more times.	\d*\.\d	".0", "19.9", "219.9"
+	Matches the previous element one or more times.	"be+"	"bee" in "been", "be" in "bent"
?	Matches the previous element zero or one time.	"rai?n"	"ran", "rain"
{ n }	Matches the previous element exactly n times.	",\d{3}"	",043" in "1,043.6", ",876", ",543", and ",210" in "9,876,543,210"
{ n ,}	Matches the previous element at least n times.	"166", "29", "1930"	
{ n , m }	Matches the previous element at least n times,	"166", "17668" "19302" in "193024"	

	but no more than m times.		
?	Matches the previous element zero or more times, but as few times as possible.	\d?\.\d	".0", "19.9", "219.9"
+?	Matches the previous element one or more times, but as few times as possible.	"be+?"	"be" in "been", "be" in "bent"
??	Matches the previous element zero or one time, but as few times as possible.	"rai??n"	"ran", "rain"
{ n }?	Matches the preceding element exactly n times.	",\d{3}?"	",043" in "1,043.6", ",876", ",543", and ",210" in "9,876,543,210"
{ n ,}?	Matches the previous element at least n times, but as few times as possible.	"166", "29", "1930"	
{ n , m }?	Matches the previous element between n and m times, but as few times as possible.	"166", "17668" "193", "024" in "193024"	

Backreference Constructs

Backreference constructs allow a previously matched sub-expression to be identified subsequently in the same regular expression. The following table lists these constructs:

Backreference construct	Description	Pattern	Matches

\ number	Backreference. Matches the value of a numbered subexpression.	(\w)\1	"ee" in "seek"
\k< name >	Named backreference. Matches the value of a named expression.	(?<char>\w)\k<char>	"ee" in "seek"

Alternation Constructs

Alternation constructs modify a regular expression to enable either/or matching. The following table lists the alternation constructs:

Alternation construct	Description	Pattern	Matches
	Matches any one element separated by the vertical bar () character.	th(e is at)	"the", "this" in "this is the day. "
(?(expression)yes no)	Matches yes if expression matches; otherwise, matches the optional no part. Expression is interpreted as a zero-width assertion.	(?(A)A\d{2}\b \b\d{3}\b)	"A10", "910" in "A10 C103 910"
(?(name)yes no)	Matches yes if the named capture name has a match; otherwise, matches the optional no.	(?< quoted>")?(?("ed).+?" \S+\s)	Dogs.jpg, "Yiska playing.jpg" in "Dogs.jpg "Yiska playing.jpg""

Substitutions

Substitutions are used in replacement patterns. The following table lists the substitutions:

Character	Description	Pattern	Replacement pattern	Input string	Result string
\$number	Substitutes the substring matched by group number.	\b(\w+)(\s)(\w+)\b	\$3\$2\$1	"one two"	"two one"
\${name}	Substitutes the substring matched by the namedgroup name.	\b(?<word1>\w+)(\s)(?<word2>\w+)\b	\${word2} \${word1}	"one two"	"two one"
\$\$	Substitutes a literal "\$".	\b(\d+)\s?USD	\$\$1	"103 USD"	"\$103"
\$&	Substitutes a copy of the whole match.	(\\$*(\d*(.+)\d+)?){1})	**\$&	"\$1.30"	"**\$1.30**"
\$`	Substitutes all the text of the input string before the match.	B+	\$`	"AABBCC"	"AAAACC"
\$'	Substitutes all the text of the input string after the match.	B+	\$'	"AABBCC"	"AACCCC"
\$+	Substitutes the last group	B+(C+)	\$+	"AABBCCDD"	AACCDD

	that was captured.				
\$_	Substitutes the entire input string.	B+	\$_	"AABBCC"	"AAAABBCC CC"

Miscellaneous Constructs

Following are various miscellaneous constructs:

Construct	Definition	Example
(?imnsx-imnsx)	Sets or disables options such as case insensitivity in the middle of a pattern.	\bA(?i)b\w+\b matches "ABA", "Able" in "ABA Able Act"
(?#comment)	In-line comment. The comment ends at the first closing parenthesis.	\bA(?#Matches words starting with A)\w+\b
# [to end of line]	X-mode comment. The comment starts at an unescaped # and continues to the end of the line.	(?x)\bA\w+\b#Matches words starting with A

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 the 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 the 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 the following result:

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

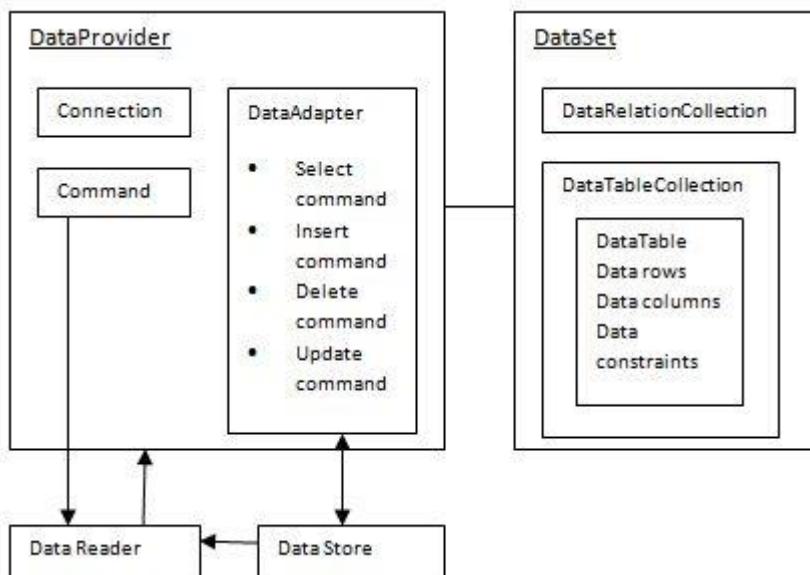
28. Database Access

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 retrieve data for the application and update data.

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

- **Datasets** store data in a disconnected cache and the application retrieves 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

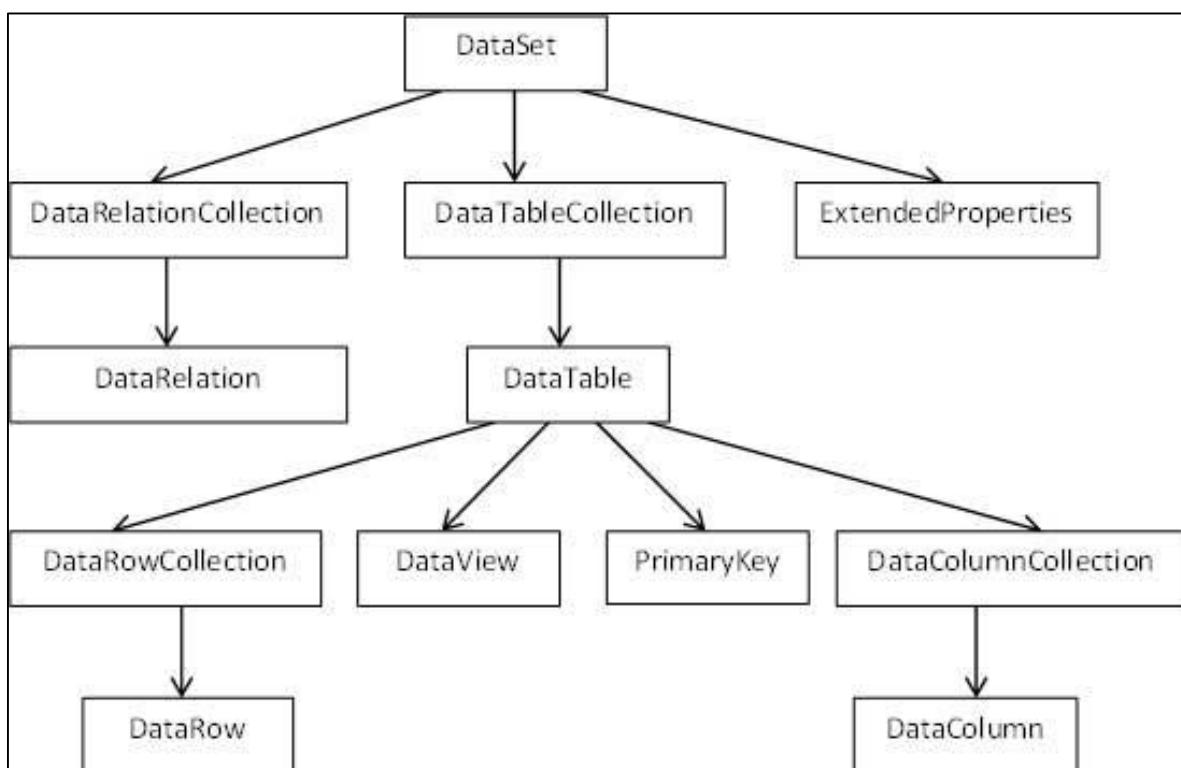
- The .Net Framework data provider for SQL Server - provides access to Microsoft SQL Server.
- The .Net Framework data provider for OLE DB - provides access to data sources exposed by using OLE DB.
- The .Net Framework data provider for ODBC - provides access to data sources exposed by ODBC.
- The .Net 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 of 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.

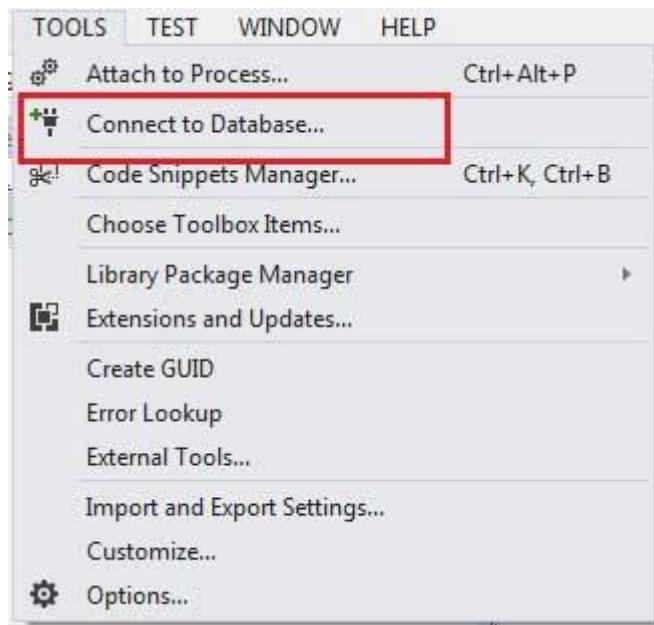
OleDbConnection - designed for connecting to a wide range of databases, like Microsoft Access and Oracle.

Example 1

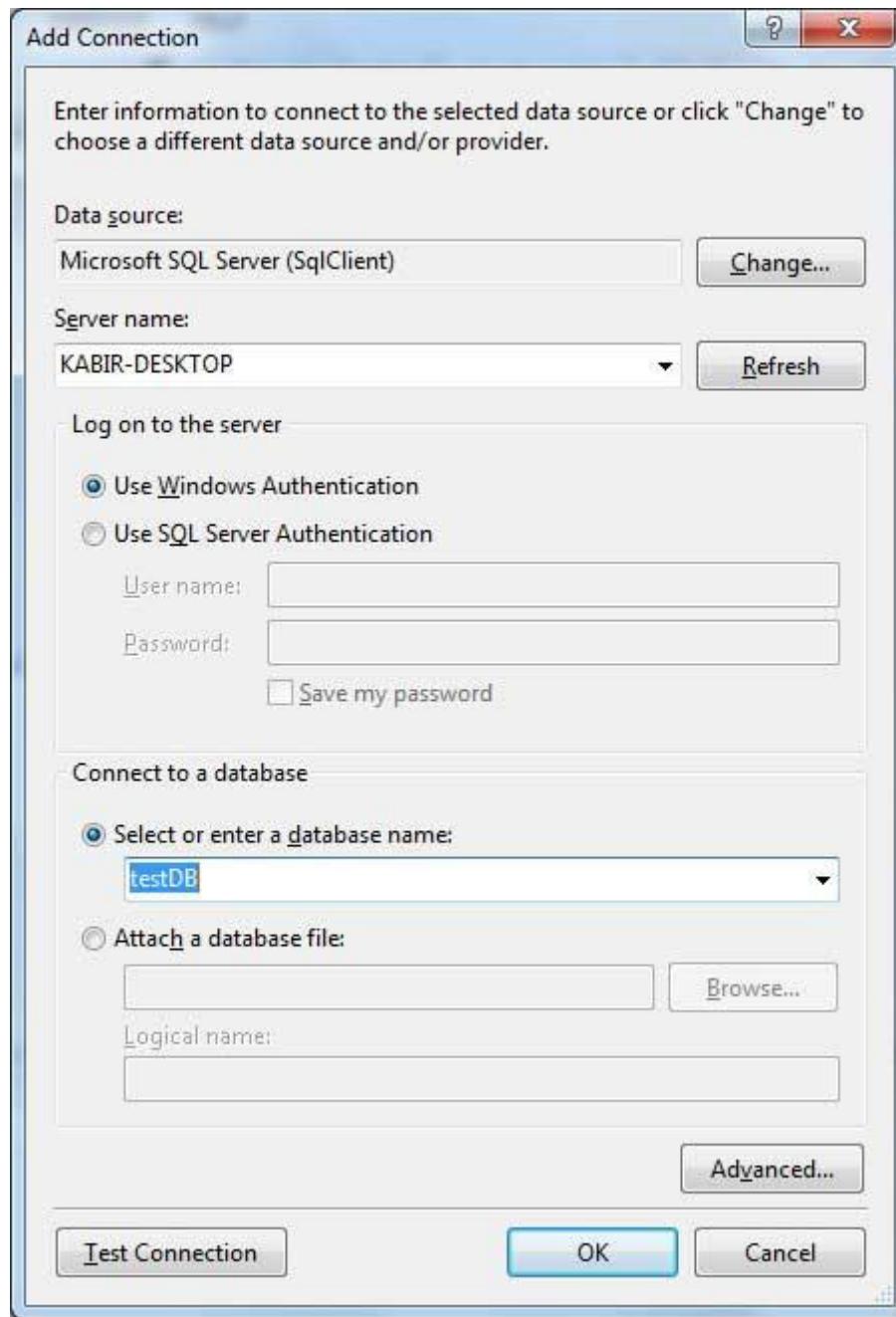
We have a table stored in Microsoft SQL Server, named Customers, in a database named testDB. Please consult 'SQL Server' tutorial for creating databases and database tables in SQL Server.

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

- Select TOOLS -> Connect to Database



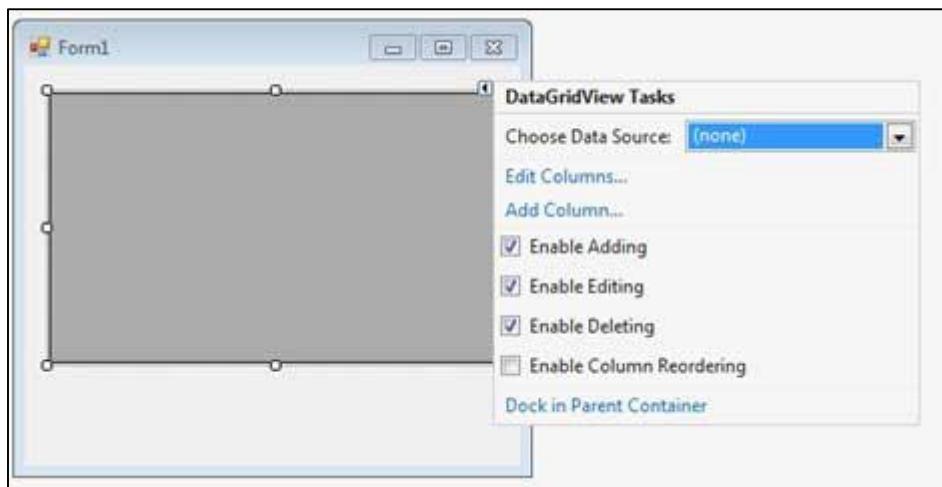
- Select a server name and the database name in the Add Connection dialog box.



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



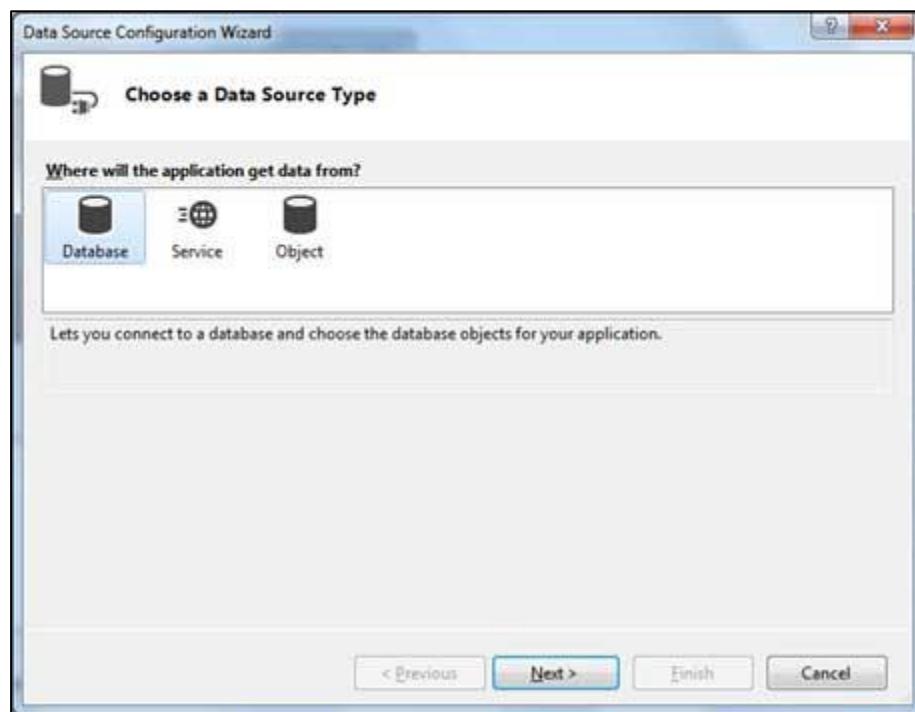
- Add a DataGridView on the form.



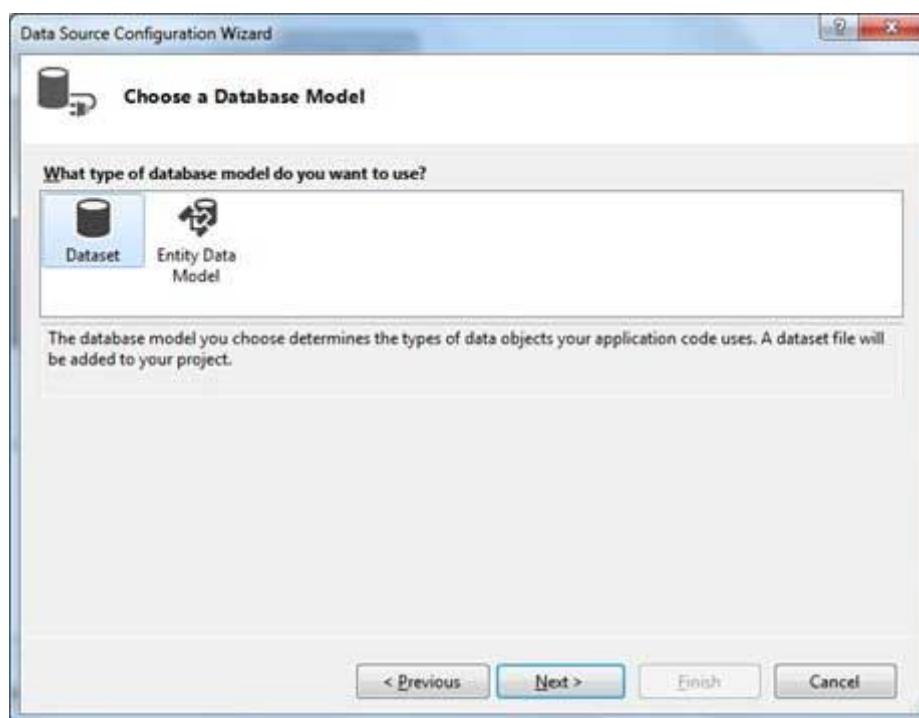
- Click on the Choose Data Source combo box.
- Click on the Add Project Data Source link.



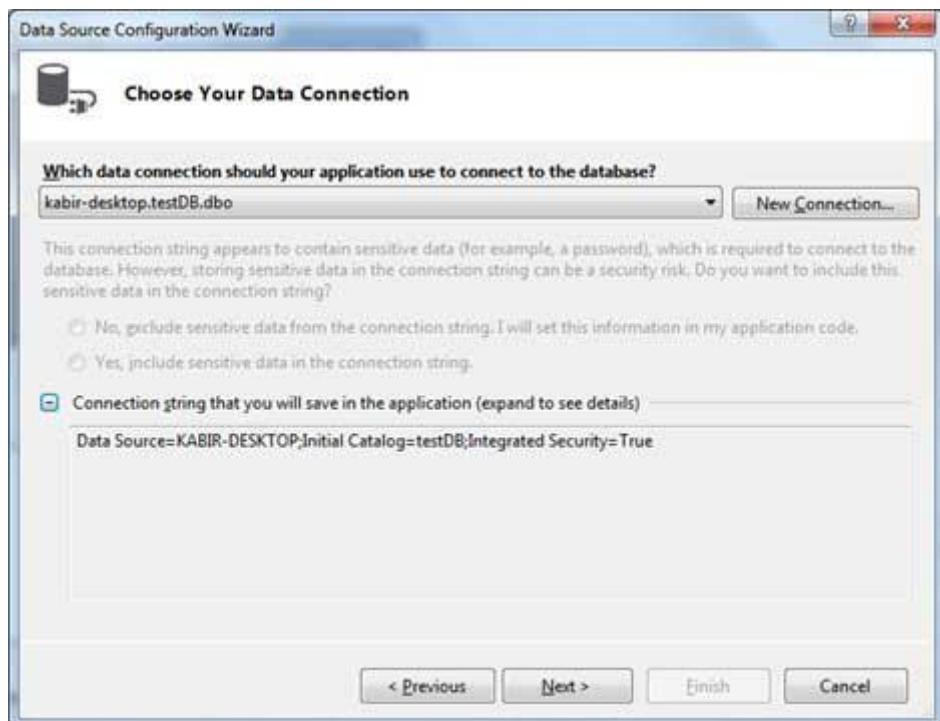
- This opens the Data Source Configuration Wizard.
- Select Database as the data source type



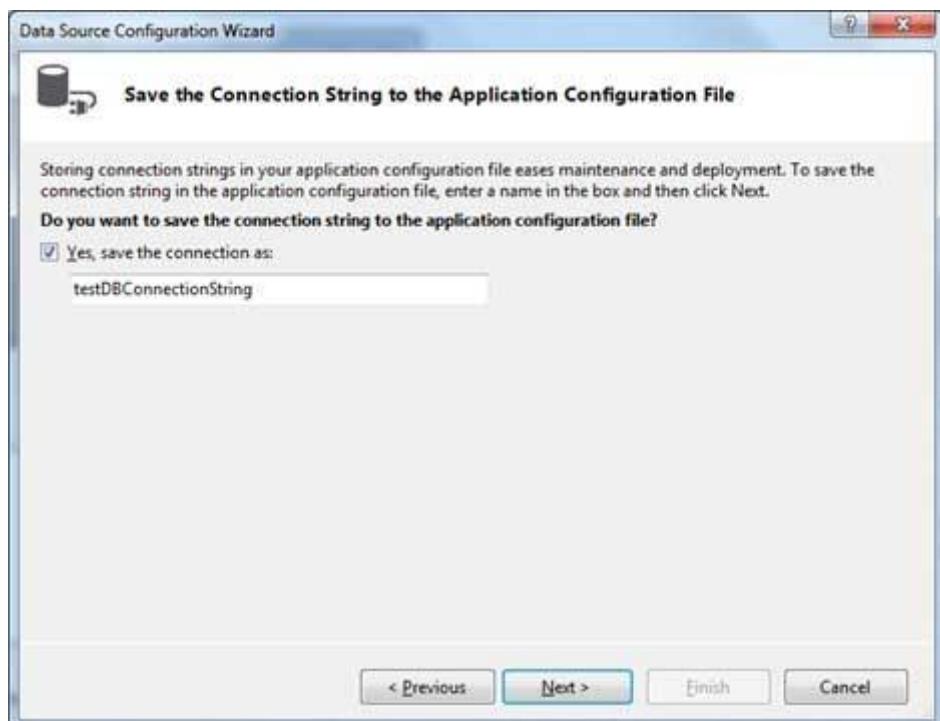
- Choose DataSet as the database model.



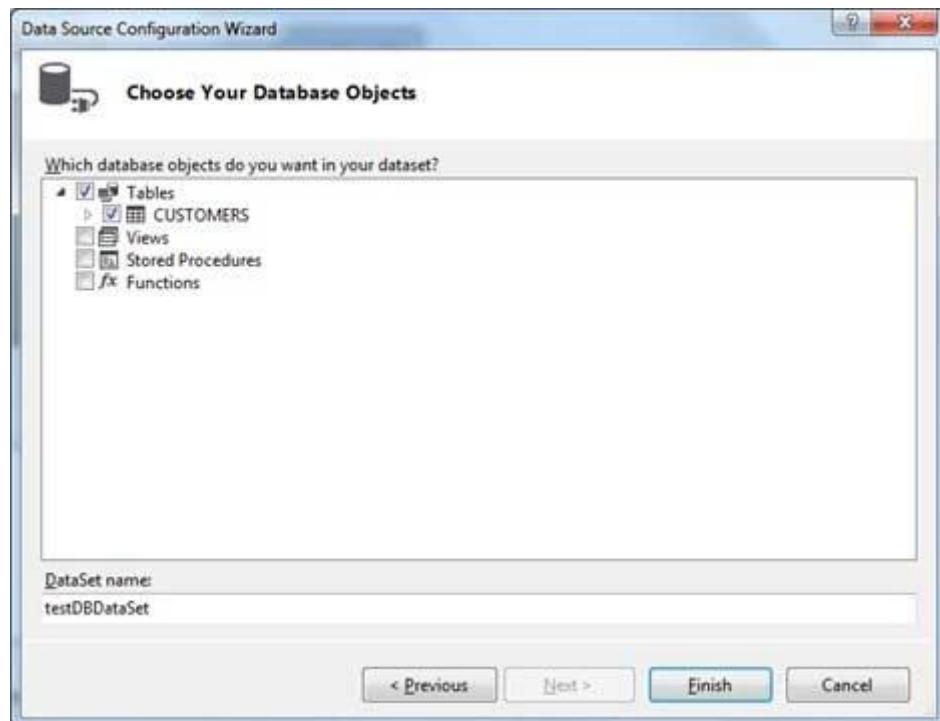
- Choose the connection already set up.



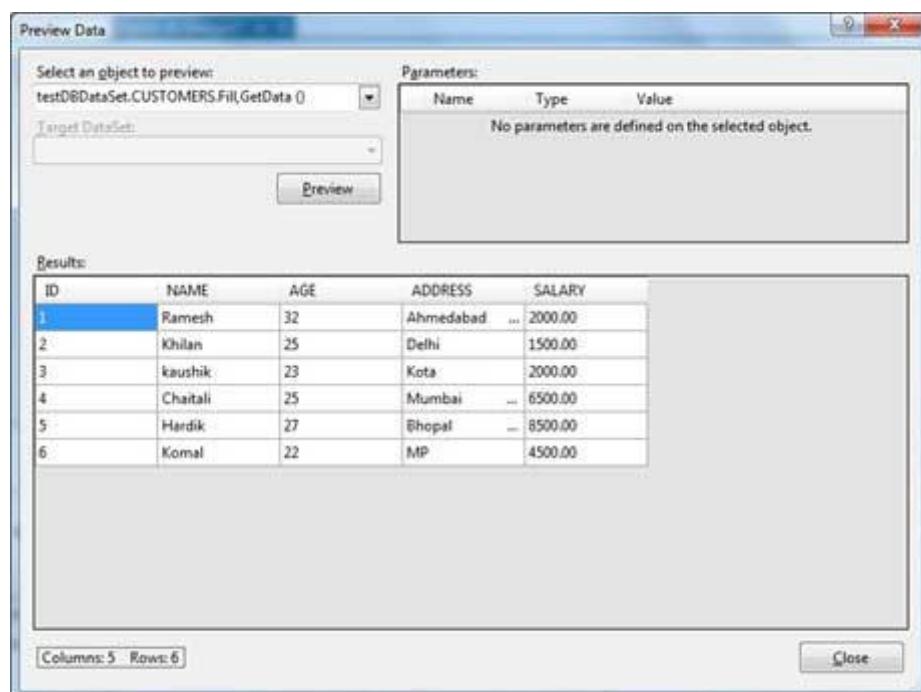
- Save the connection string.



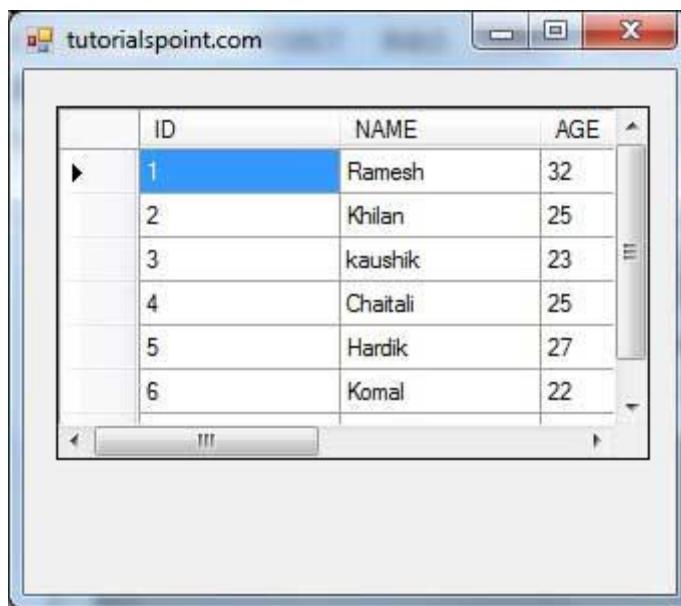
- Choose the database object, Customers table in our example, and click the Finish button.



- 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 the following window:



Example 2

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

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

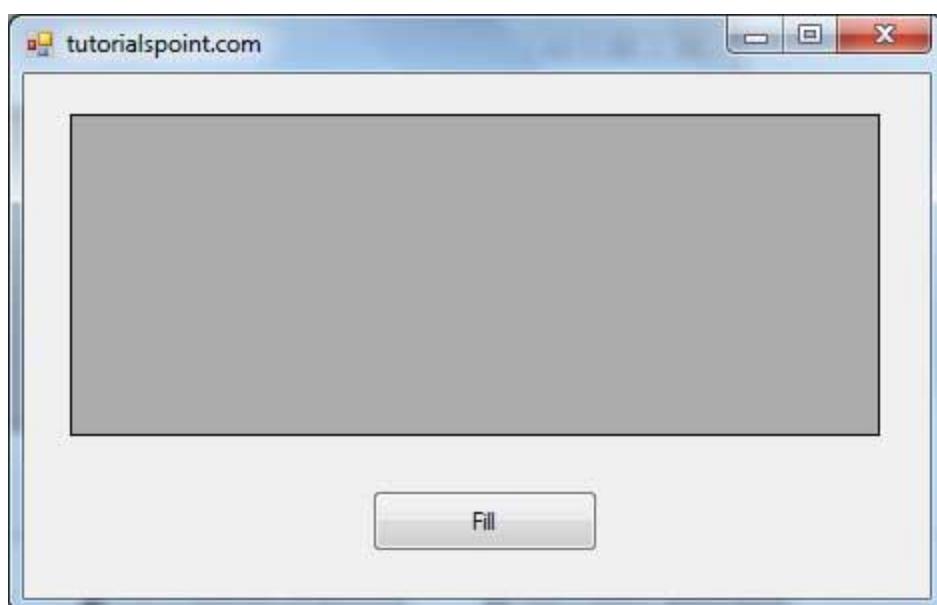
    DataGridView1.DataSource = ds.Tables(0)

End Sub

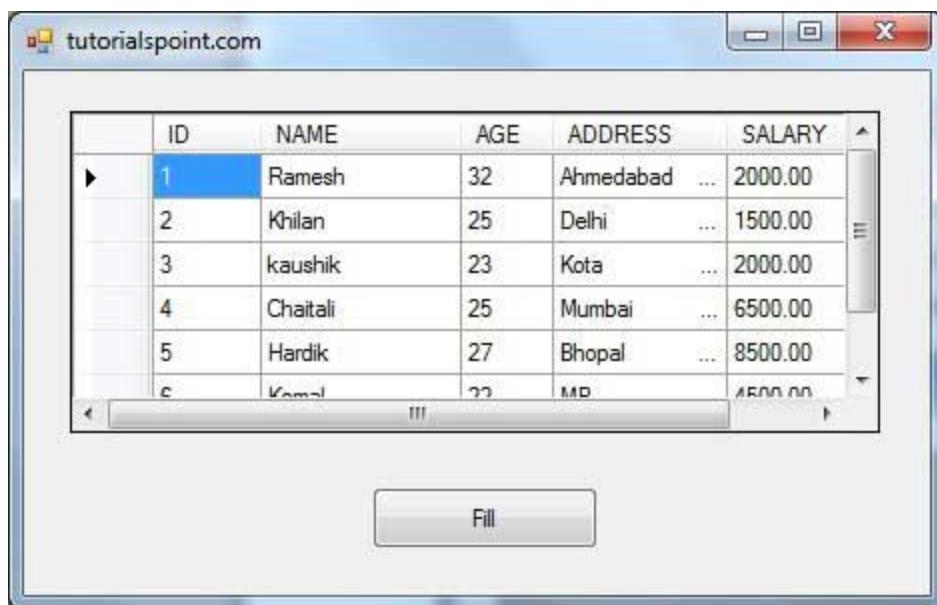
End Class

```

- When the above code is executed and run using **Start** button available at the Microsoft Visual Studio tool bar, it will show the 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 allow 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 columns, 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 = "tutorialspt.com"
    End Sub
    Private Function CreateDataSet() As DataSet
        'creating a DataSet object for tables
    End Function

```

```

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

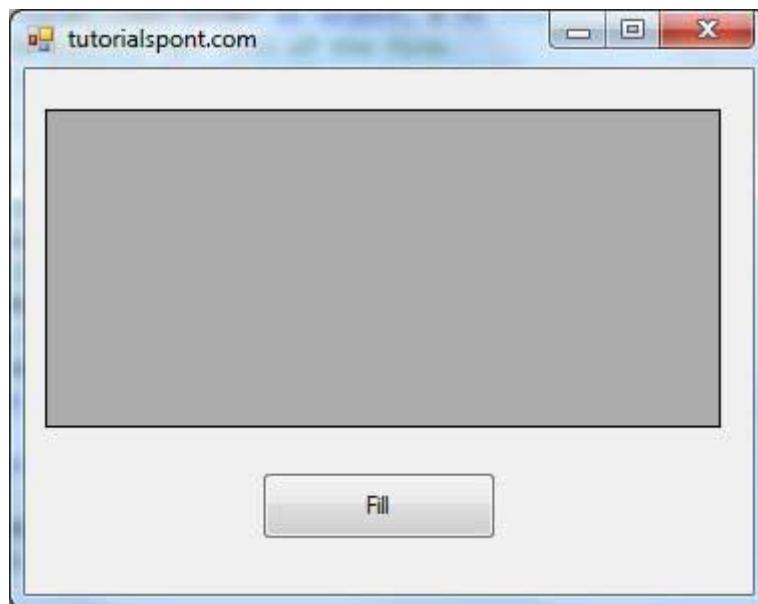
```

```
    table.Rows.Add(newrow)
End Sub

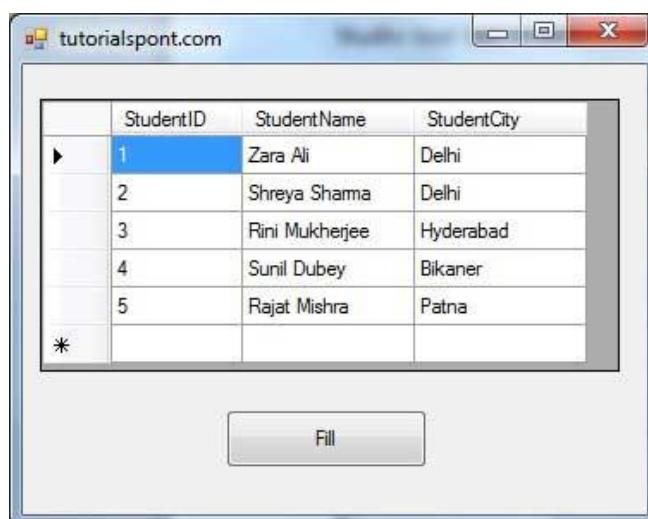
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles
Button1.Click
    Dim ds As New DataSet
    ds = CreateDataSet()
    DataGridView1.DataSource = ds.Tables("Student")
End Sub

End Class
```

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



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



29. 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

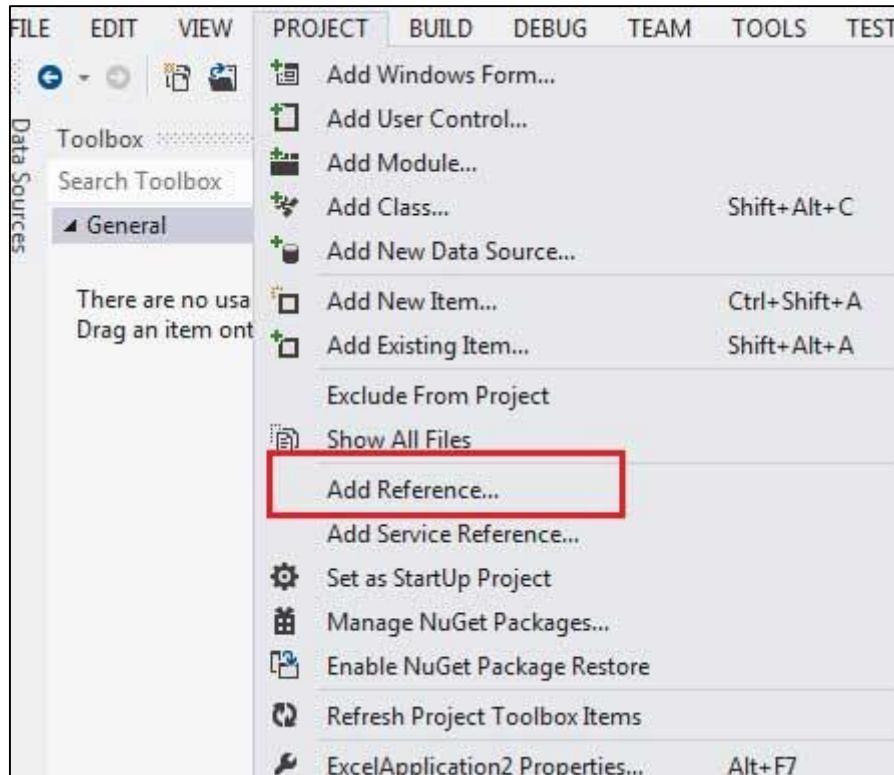
Let's start with creating a Window Forms Application by following the following steps in Microsoft Visual Studio: **File -> New Project -> Windows Forms Applications**

Finally, select OK, Microsoft Visual Studio creates your project and displays following **Form1**.

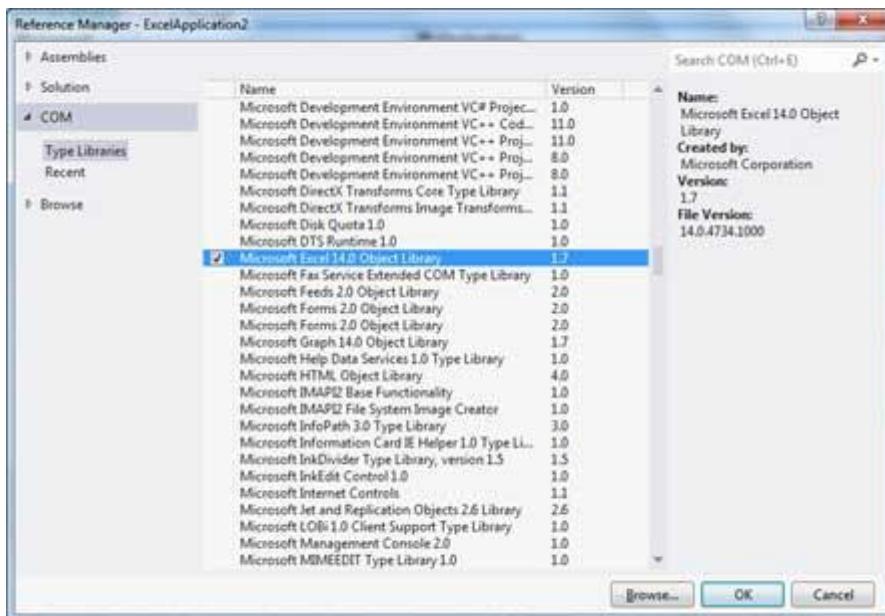
Insert a Button control Button1 in the form.

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

Select Add Reference from the Project Menu.



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



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"
    End Sub
End Class
```

```

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.

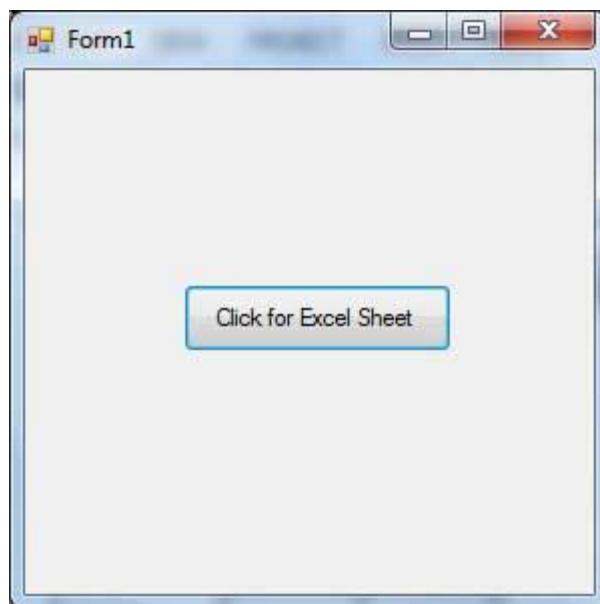
```

```

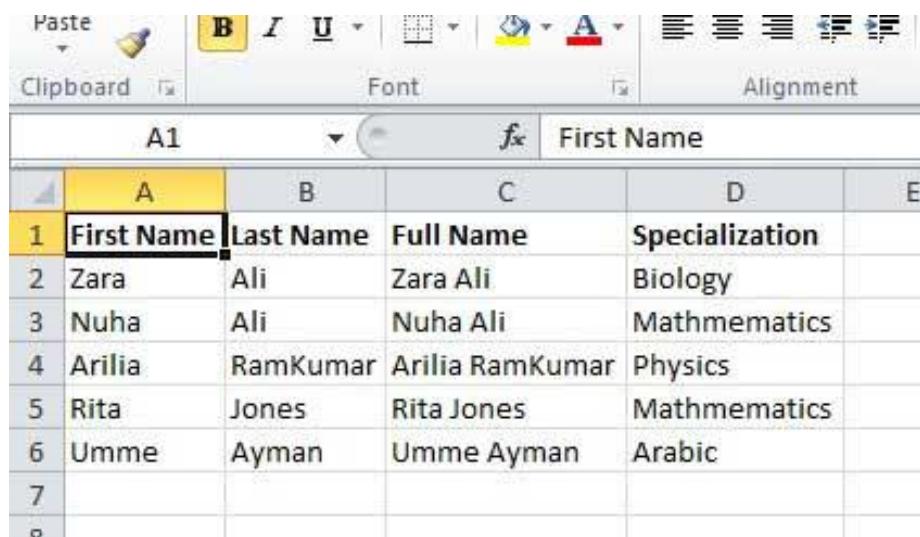
raXL = shXL.Range("A1", "D1")
raXL.EntireColumn.AutoFit()
    ' Make sure Excel is visible and give the user control
    ' of Excel's lifetime.
appXL.Visible = True
appXL.UserControl = True
    ' Release object references.
raXL = Nothing
shXL = Nothing
wbXL = Nothing
appXL.Quit()
appXL = Nothing
Exit 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 the following window:



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



A screenshot of Microsoft Excel showing a table of student data. The table has columns labeled 'First Name', 'Last Name', 'Full Name', and 'Specialization'. The data includes rows for students Zara Ali, Nuha Ali, Arilia RamKumar, Rita Jones, and Umme Ayman.

	A1	Font	First Name	Alignment	
	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					
8					

30. Send Email

VB.Net allows sending e-mails from your application. The **System.Net.Mail** namespace contains classes used for sending e-mails 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	Stores 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	Specifies 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	Specifies 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.ToProperties, respectively.
- You should also specify the message content using the MailMessage.Bodyproperty.

Example

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

- Add three labels, three text boxes and a button control in the form.
- Change the text properties of the labels to - 'From', 'To:' and 'Message:' respectively.
- Change the name properties of the texts to txtFrom, txtTo and txtMessage respectively.
- Change the text property of the button control to 'Send'
- 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"

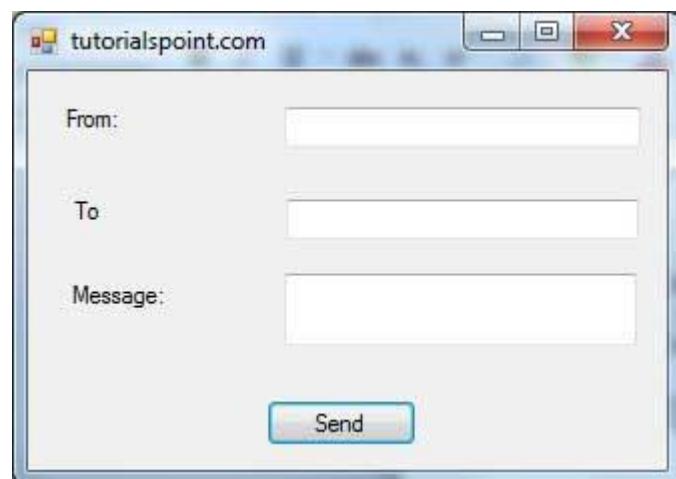
e_mail = New MailMessage()
e_mail.From = New MailAddress(txtFrom.Text)
e_mail.To.Add(txtTo.Text)
e_mail.Subject = "Email Sending"
e_mail.IsBodyHtml = False
e_mail.Body = txtMessage.Text
Smtp_Server.Send(e_mail)
MsgBox("Mail Sent")

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 the following window, which you will use to send your e-mails, try it yourself.



31. 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	XmDictionary	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	XmISignificantWhitespace	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:

- Add the movies.xml file in the bin\Debug folder of your application.
- Import the System.Xml namespace in Form1.vb file.
- Add a label in the form and change its text to 'Movies Galore'.
- Add three list boxes and three buttons to show the title, type and description of a movie from the xml file.
- 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

```

```

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:

- Add a WebBrowser control and a button control in the form.
- Change the Text property of the button to Show Authors File.
- 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()

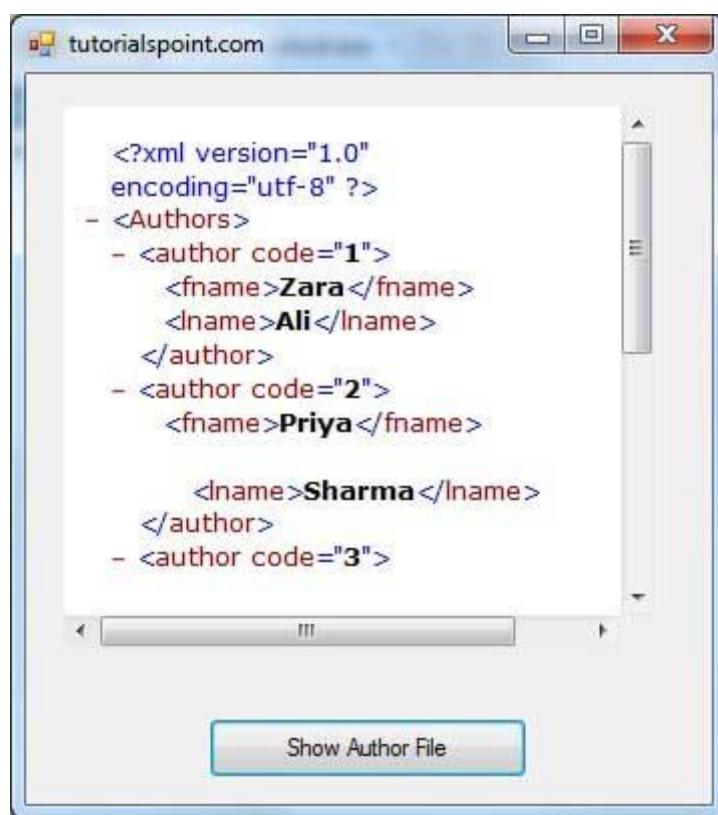
WebBrowser1.Url = New Uri(AppDomain.CurrentDomain.BaseDirectory +
"authors.xml")

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 **XmIDocument** 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 **XmIDocument** 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 an 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
```

```

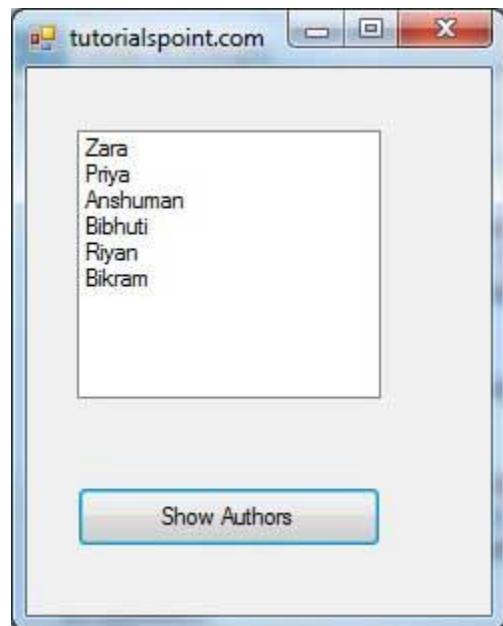
Public Class Form1

    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.



32. 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.

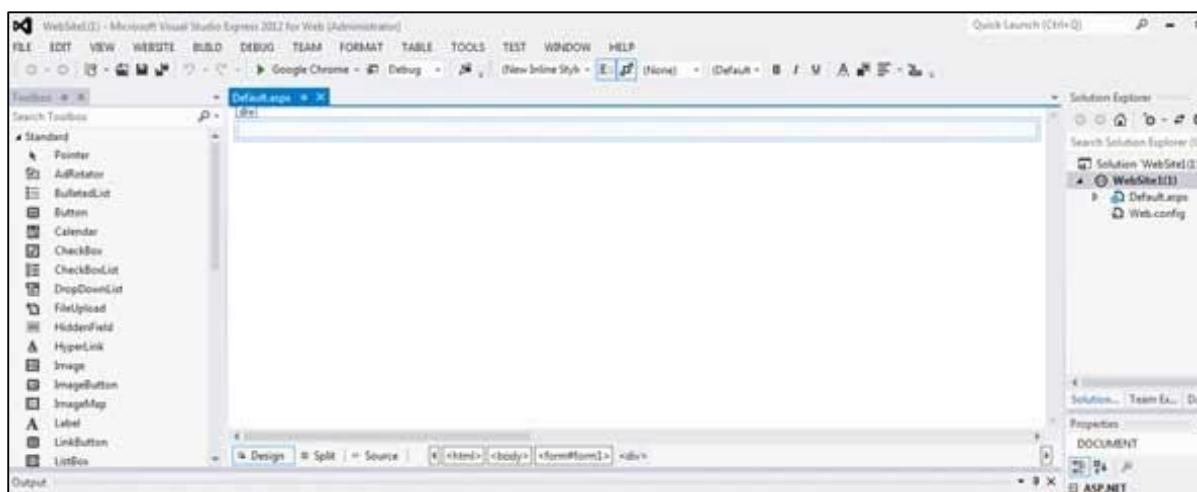
	You use this object to store and retrieve information to be shared among all users of an application. For example, you can use an Application object to create an e-commerce page.
Request	<p>Describes the methods, properties, and collections of the object that stores information related to the HTTP request. This includes forms, cookies, server variables, and certificate data.</p> <p>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.</p>
Response	<p>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.</p> <p>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.</p>
Server	<p>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.</p> <p>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.</p>
Session	<p>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.</p> <p>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.</p>

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.



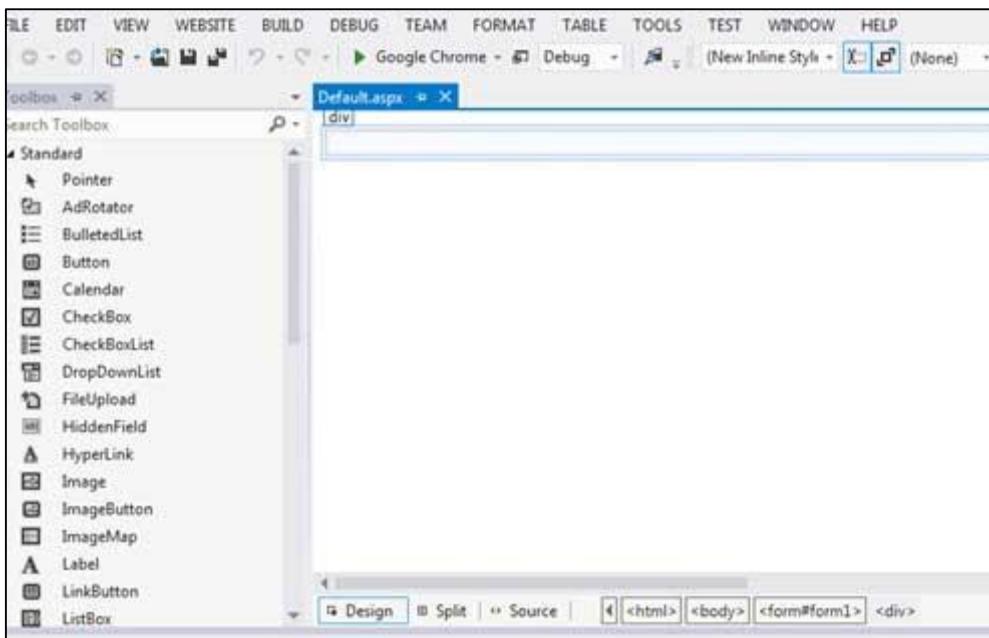
Web Forms

Web forms consist 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.

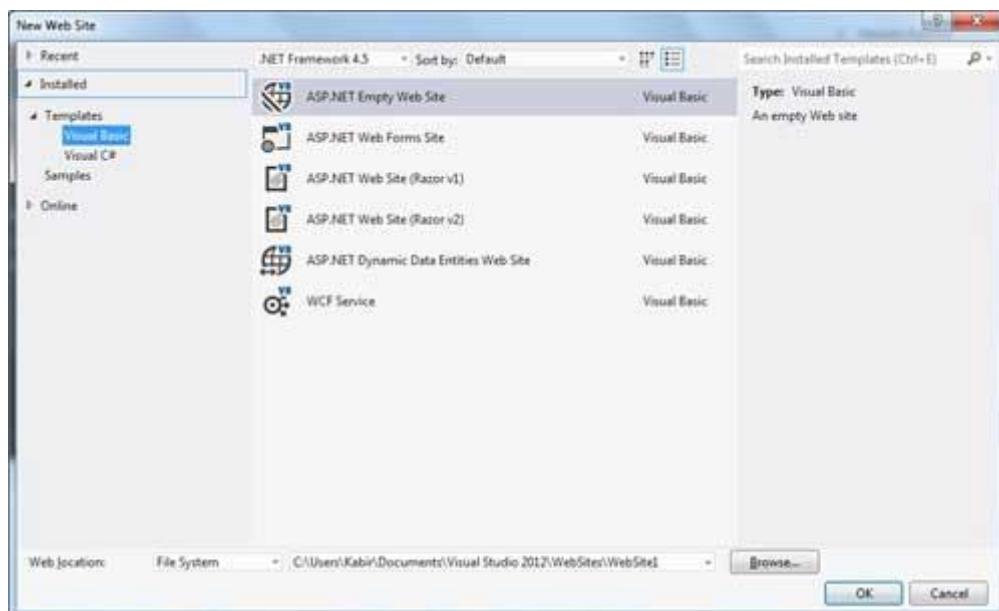
The application logic consists of code applied to the user interface elements in the page. You write this code in any of .Net language like, VB.Net, or C#. The following figure shows a Web Form in Design view:



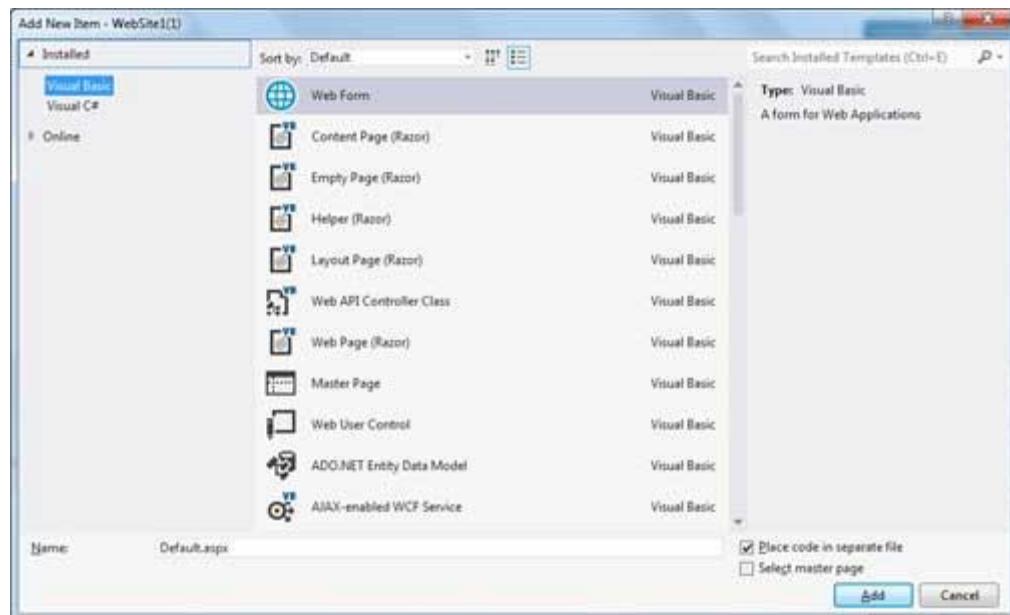
Example

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:

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



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



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

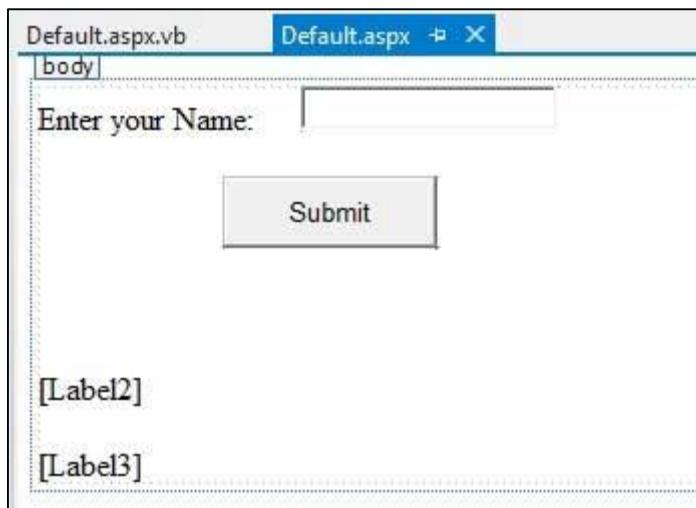
The Default page is shown in Source view

```

FILE EDIT VIEW WEBSITE BUILD DEBUG TEAM TOOLS TEST WINDOW HELP
Default.aspx * X
<%@ Page Language="VB" AutoEventWireup="false" CodeFile="Default.aspx.vb" Inherits="_Default" %>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
        </div>
    </form>
</body>
</html>

```

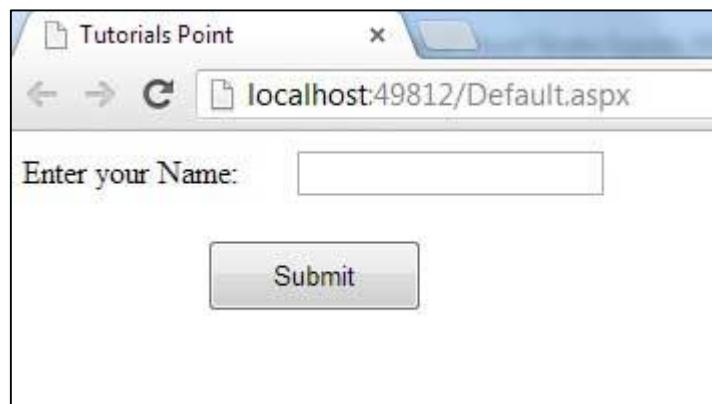
- Set the title for the Default web page by adding a value to the `<title>` tag.
- To add controls on the web page, go to the design view. Add three labels, a text box and a button on the form.



- 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:

The screenshot shows a web browser window titled 'Tutorials Point' with the URL 'localhost:49812/Default.aspx'. The page contains a form with a text input field labeled 'Enter your Name:' containing 'James Bond' and a 'Submit' button. Below the form, the text 'Welcome to Tutorials Point: James Bond' is displayed, followed by 'You visited us at: 2/6/2013 9:03:30 PM'.

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 a 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

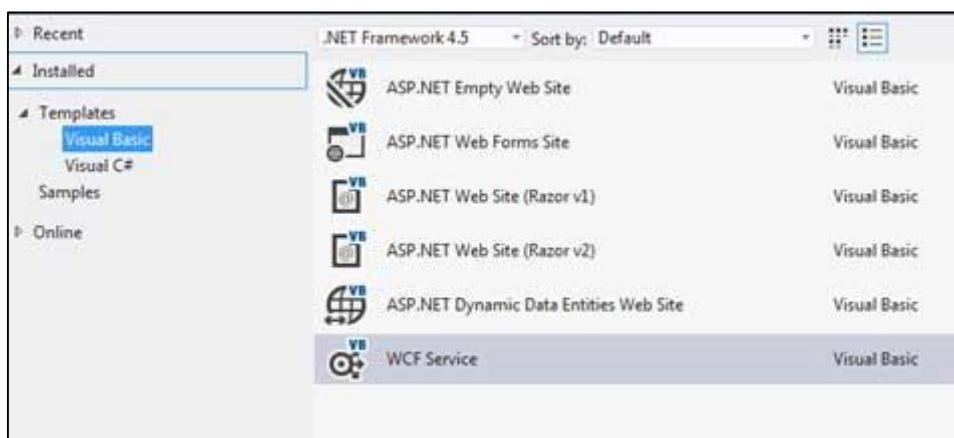
To understand the concept let us create a simplistic service that will provide stock price information. The clients can query about the name and price of a stock based on the stock symbol. To keep this example simple, the values are hardcoded in a two-dimensional array. This service will have two methods:

- 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:

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



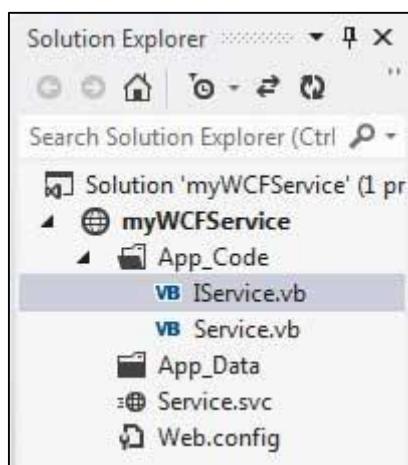
- Select File System from the Web location drop-down list.

- 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
    End Function

```

```

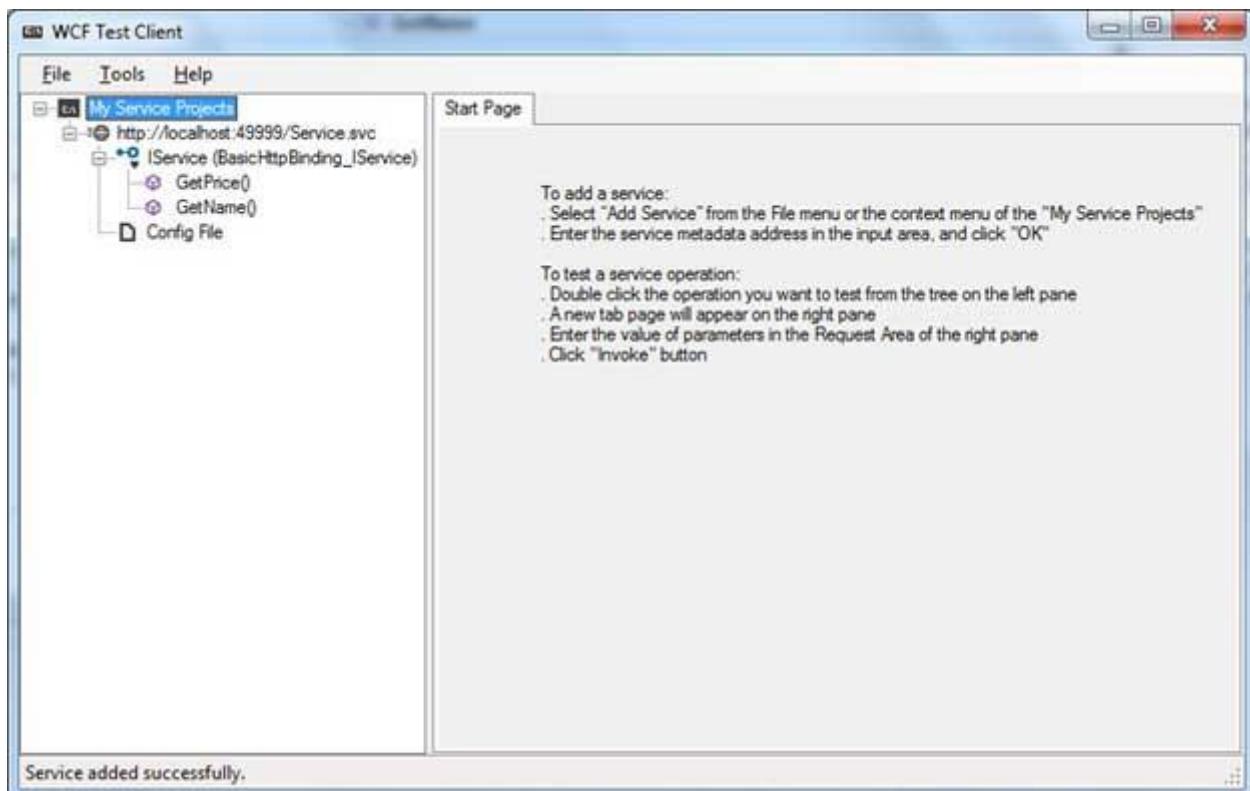
Dim i As Integer
For i = 0 To i = stocks.GetLength(0) - 1

    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

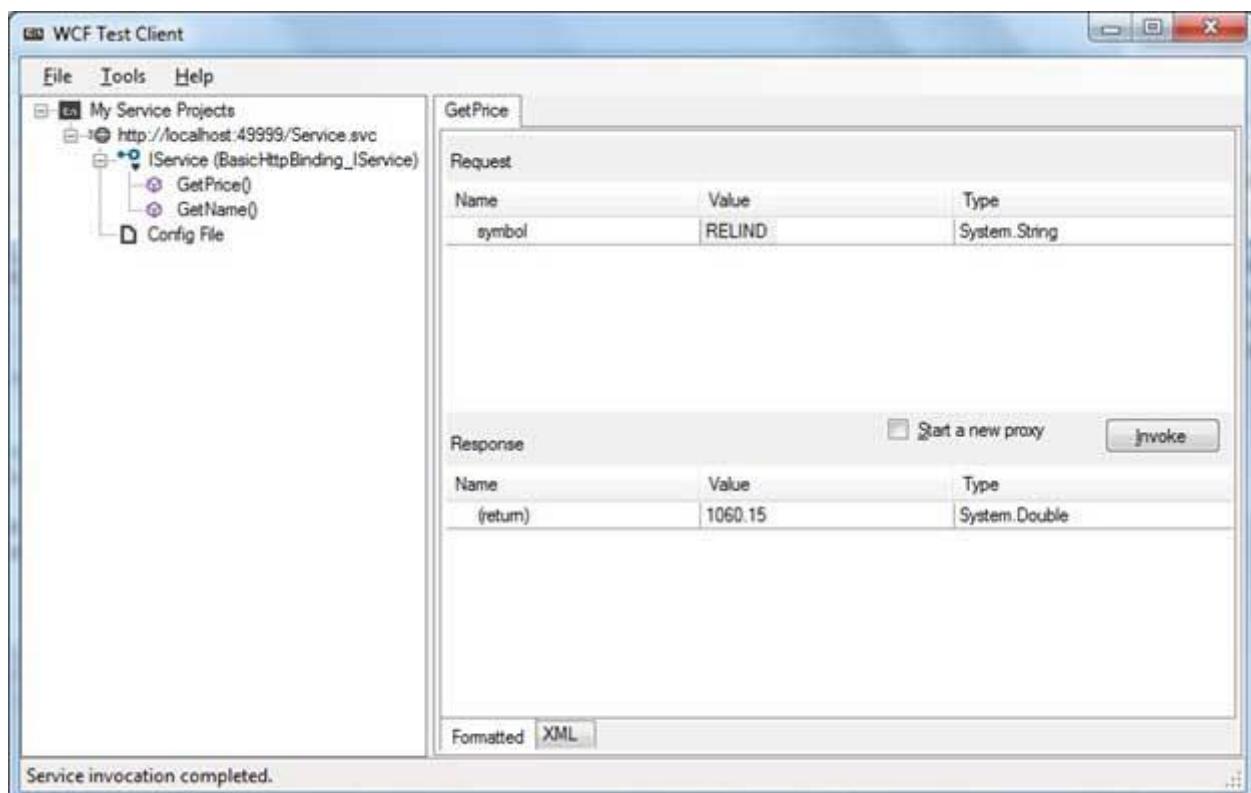
To run the WCF Service, so created, select the Debug->Start Debugging option from the menu bar. The output would be:



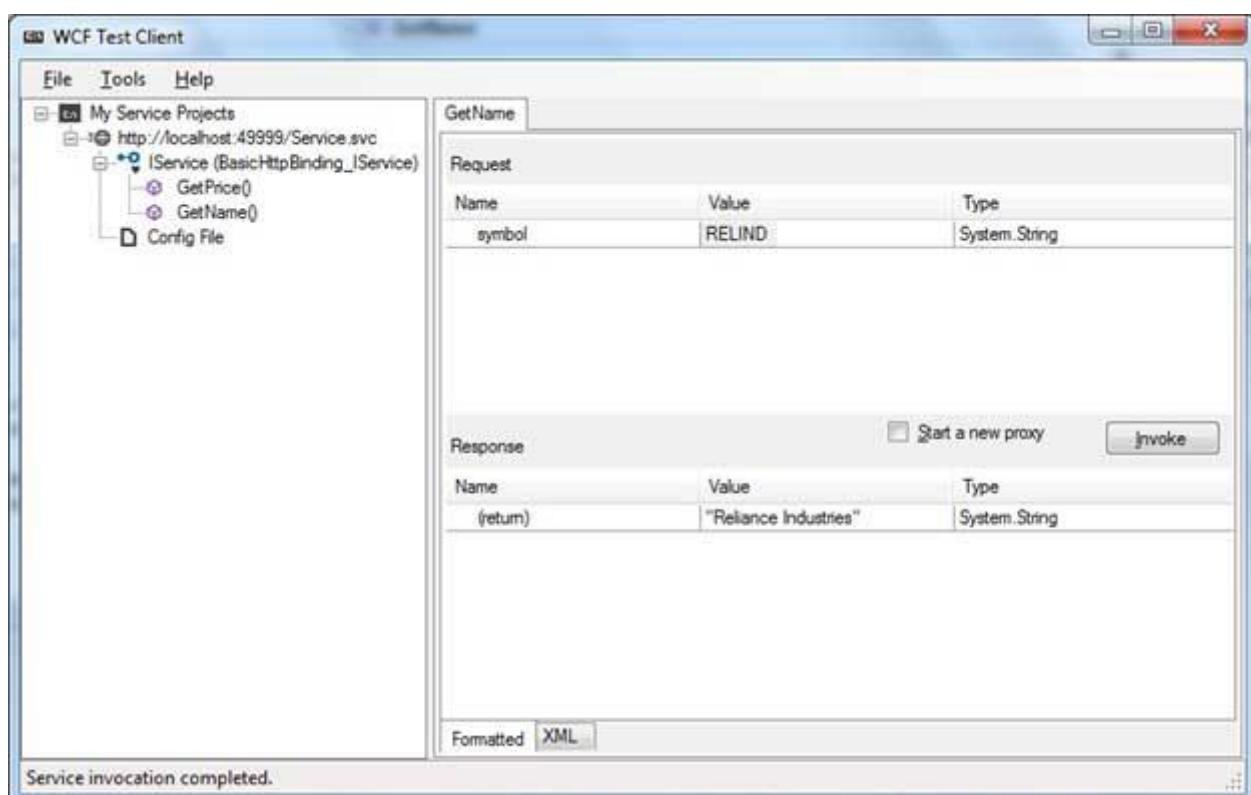
For testing the service operations, double click the name of the operation from the tree on the left pane. A new tab will appear on the right pane.

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:

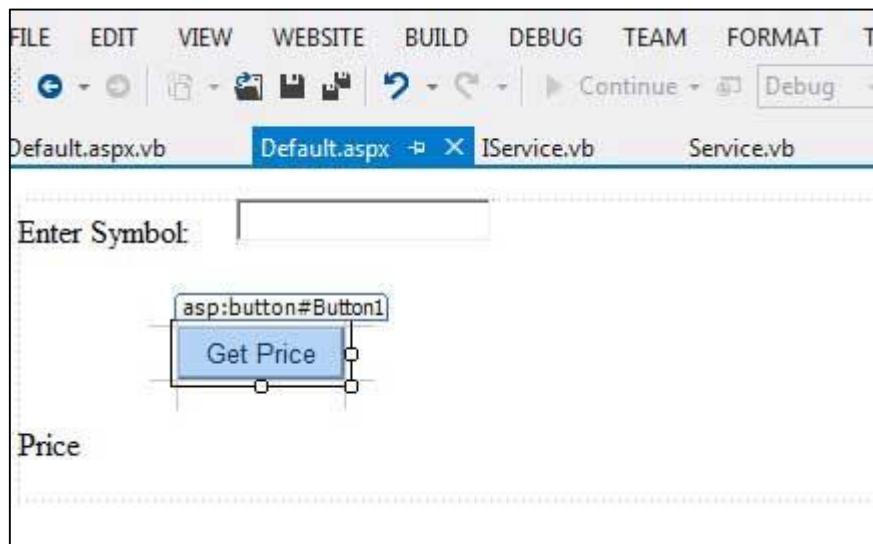


Utilizing the Service

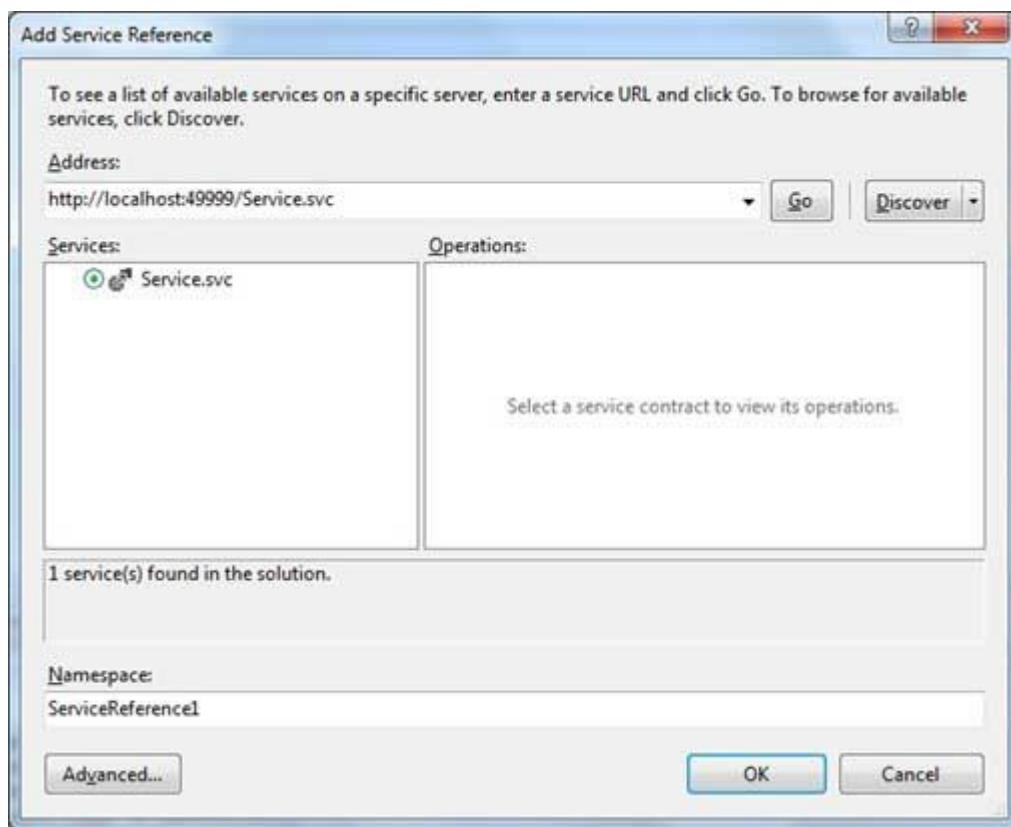
Let us add a default page, an ASP.NET web form in the same solution from which we will be using the WCF Service we have just created.

Take the following steps:

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



- 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.
- 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.
- 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:

localhost:49999/Default.aspx

Enter Symbol:

Get Price

Price

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

localhost:49999/Default.aspx

Enter Symbol:

Get Price

Price
1060.15