What is VB.NET?

The VB.NET stands for Visual Basic. Network Enabled Technologies. It is a simple, high-level, object-oriented programming language developed by Microsoft in 2002. It is a successor of Visual Basic 6.0, that is implemented on the Microsoft .NET framework. Furthermore, it supports the OOPs concept, such as abstraction, encapsulation, inheritance, and polymorphism. Therefore, everything in the VB.NET language is an object, including all primitive data types (Integer, String, char, long, short, Boolean, etc.), user-defined data types, events, and all objects that inherit from its base class. It is not a case sensitive language, whereas, C++, Java, and C# are case sensitive language.

Applications built using the VB.NET language are very reliable and scalable, relying on the .NET Framework to access all libraries that help to execute a VB.NET program. With this language, you can develop a fully object-oriented application that is similar to an application created through another language such as C++, Java, or C#. In addition, applications or programs of VB.NET are not only running on the window operating system but can also run on Linux or Mac OS.

The VB.NET language is designed in such a way that any new beginner or novice and the advanced programmer can quickly develop a simple, secure, robust, high performance of web, windows, console, and mobile application running on .NET Framework.

VB.NET Features

As we know, it is a high-level programming language with many features to develop a secure and robust application. These are the following features that make it the most popular programming language.

- It is an object-oriented programming language that follows various oops concepts such as abstraction, encapsulation, inheritance, and many more. It means that everything in VB.NET programming will be treated as an object.
- This language is used to design user interfaces for window, mobile, and web-based applications.
- It supports a rapid application development tool kit. In which a developer does not need to write all the codes as it can get various code automatically from its libraries. For example, when we create a form in Visual basic.net, it automatically calls events of various form in that class.
- It is not a case sensitive language like other languages such as C++, java, etc.
- It supports Boolean condition for decision making in programming.
- It also supports the multithreading concept, in which you can do multiple tasks at the same time.
- It provides simple events management in .NET application.

- A Window Form enables us to inherit all existing functionality of form that can be used to create a new form. So, in this way, it reduced the code complexity.
- It uses an external object as a **reference** that can be used in a VB.NET application.
- Automatic initialized a garbage collection.
- It follows a structured and extensible programming language for error detection and recovery.
- Conditional compilation and easy to use generic classes.
- It is useful to develop web, window, and mobile applications.
-

Advantages of VB.NET

- The VB.NET executes a program in such a way that runs under CLR (Common Language Runtime), creating a robust, stable, and secure application.
- It is a pure object-oriented programming language based on objects and classes. However, these features are not available in the previous version of Visual Basic 6. That's why Microsoft launched VB.NET language.
- Using the Visual Studio IDE, you can develop a small program that works faster, with a large desktop and web application.
- The .NET Framework is a software framework that has a large collection of libraries, which helps in developing more robust applications.
- It uses drop and drag elements to create web forms in .NET applications.
- However, a Visual Basic .NET allows to connect one application to another application that created in the same language to run on the .NET framework.
- A VB.NET can automatically structure your code.
- The Visual Basic .NET language is also used to transfer data between different layers of the .NET architecture such that data is passed as simple text strings.
- It uses a new concept of error handling in the Visual Basic .NET Framework. The new structure is the try, catch, and finally method used to handle exceptions as a unit. In addition, it allows appropriate action to be taken at the place where it encountered an error. In this way, it discourages the use of the ON ERROR GOTO statement in .NET programming.

Disadvantages of VB.NET

1. The VB.NET programming language is unable to handle pointers directly. Because in this language, it requires a lot of programming, and it is not easy to manage every address by a pointer. Furthermore, additional coding takes extra CPU cycles, that increases the processing time. It shows the slowness of the VB.NET application.
2. The VB.NET programming is easy to learn, that increases a large competition between the programmers to apply the same employment or project in VB.NET. Thus, it reduces a secure job in the programming field as a VB.NET developer.

3. It uses an Intermediate Language (IL) compilation that can be easily decompiled (reverse engineered), but there is nothing that can prevent an application from disintegrating.
4. Just-In-Time (JIT) compiler: It is the process through which a computer can interpret IL (intermediate language) compilation and is also required to run your application. It means that the target computer needs a JIT compiler to interpret a source program in IL, and this interpretation requires an additional CPU cycle that degrades the performance of an application.
5. It contains a large collection of libraries for the JIT compiler that helps to interpret an application. These large libraries hold a vast space in our system that takes more computing time.

Introduction to .NET Framework

The **.NET Framework** is a software development platform that was introduced by Microsoft in the late 1990 under the NGWS. On 13 February 2002, Microsoft launched the first version of the .NET Framework, referred to as the **.NET Framework 1.0**.

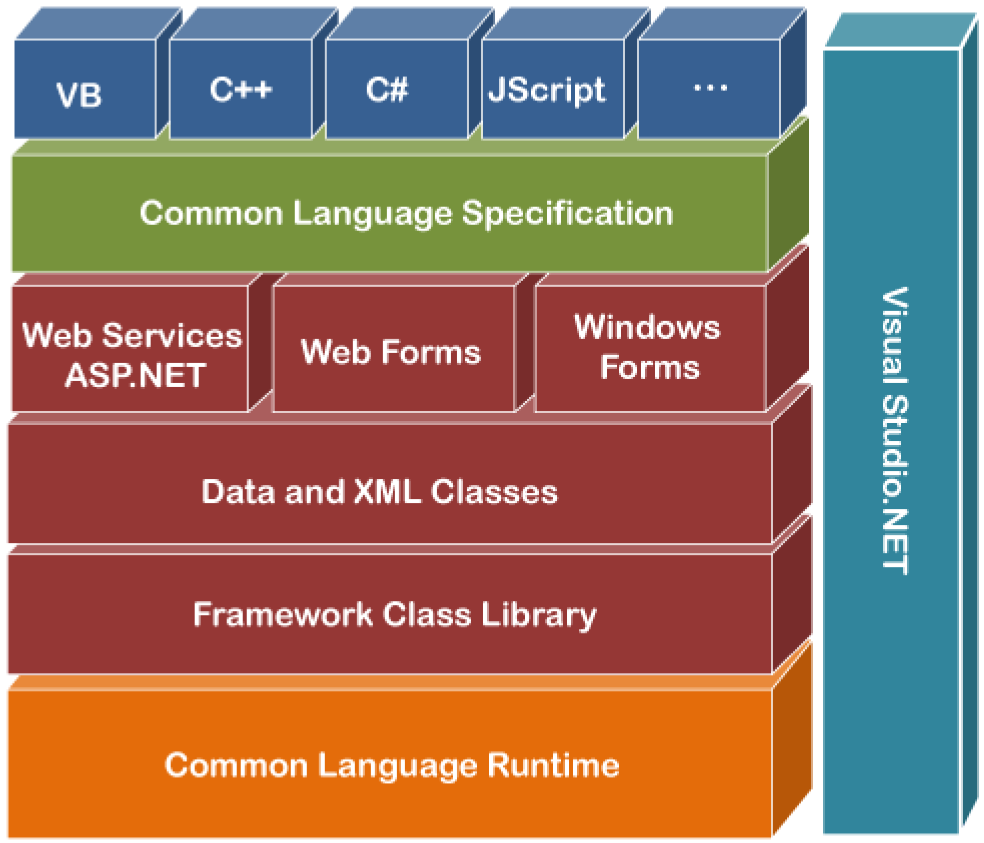
In this section, we will understand the **.NET Framework, characteristics**, **components,** and its **versions**.

What is .NET Framework

It is a virtual machine that provide a common platform to run an application that was built using the different language such as C#, VB.NET, Visual Basic, etc. It is also used to create a form based, console-based, mobile and web-based application or services that are available in Microsoft environment. Furthermore, the .NET framework is a pure object oriented, that similar to the Java language. But it is not a platform independent as the Java. So, its application runs only to the windows platform.

The main objective of this framework is to develop an application that can run on the windows platform. The current version of the .Net framework is 4.8.

Note: The .NET Framework is not only a language, but it is also a software and language neutral platform.



Components of .NET Framework

There are following components of .NET Framework:

1. CLR (Common Language Runtime)
2. CTS (Common Type System)
3. BCL (Base Class Library)
4. CLS (Common Language Specification)
5. FCL (Framework Class Library)
6. .NET Assemblies
7. XML Web Services
8. Window Services

CLR (common language runtime)

It is an important part of a .NET framework that works like a virtual component of the .NET Framework to executes the different languages program like c#, Visual Basic, etc. A CLR also helps to convert a source code into the byte code, and this byte code is known as CIL (Common Intermediate Language) or MSIL (Microsoft Intermediate Language). After converting into a byte code, a CLR uses a JIT compiler at run time that helps to convert a CIL or MSIL code into the machine or native code.

CTS (Common Type System)

It specifies a standard that represent what type of data and value can be defined and managed in computer memory at runtime. A CTS ensures that programming data defined in various languages should beinteract with each other to share information. For example, in C# we define data type as int, while in VB.NET we define integer as a data type.

BCL (Base Class Library)

The base class library has a rich collection of libraries features and functions that help to implement many programming languages in the .NET Framework, such as C #, F #, Visual C ++, and more. Furthermore, BCL divides into two parts:

1. **User defined class library**
    - **Assemblies -** It is the collection of small parts of deployment an application's part. It contains either the DLL (Dynamic Link Library) or exe (Executable) file.
        1. In LL, it uses code reusability, whereas in exe it contains only output file/ or application.
        2. DLL file can't be open, whereas exe file can be open.
        3. DLL file can't be run individually, whereas in exe, it can run individually.
        4. In DLL file, there is no main method, whereas exe file has main method.
2. **Predefined class library**
    - **Namespace -** It is the collection of predefined class and method that present in .Net. In other languages such as, C we used header files, in java we used package similarly we used "using system" in .NET, where using is a keyword and system is a namespace.

CLS (Common language Specification)

It is a subset of common type system (CTS) that defines a set of rules and regulations which should be followed by every language that comes under the .net framework. In other words, a CLS language should be cross-language integration or interoperability. For example, in C# and VB.NET language, the C# language terminate each statement with semicolon, whereas in VB.NET it is not end with semicolon, and when these statements execute in .NET Framework, it provides a common platform to interact and share information with each other.

Microsoft .NET Assemblies

A .NET assembly is the main building block of the .NET Framework. It is a small unit of code that contains a logical compiled code in the Common Language infrastructure (CLI), which is used for deployment, security and versioning. It defines in two parts (process) DLL and library (exe) assemblies. When the .NET program is compiled, it generates a metadata with Microsoft Intermediate Language, which is stored in a file called Assembly.

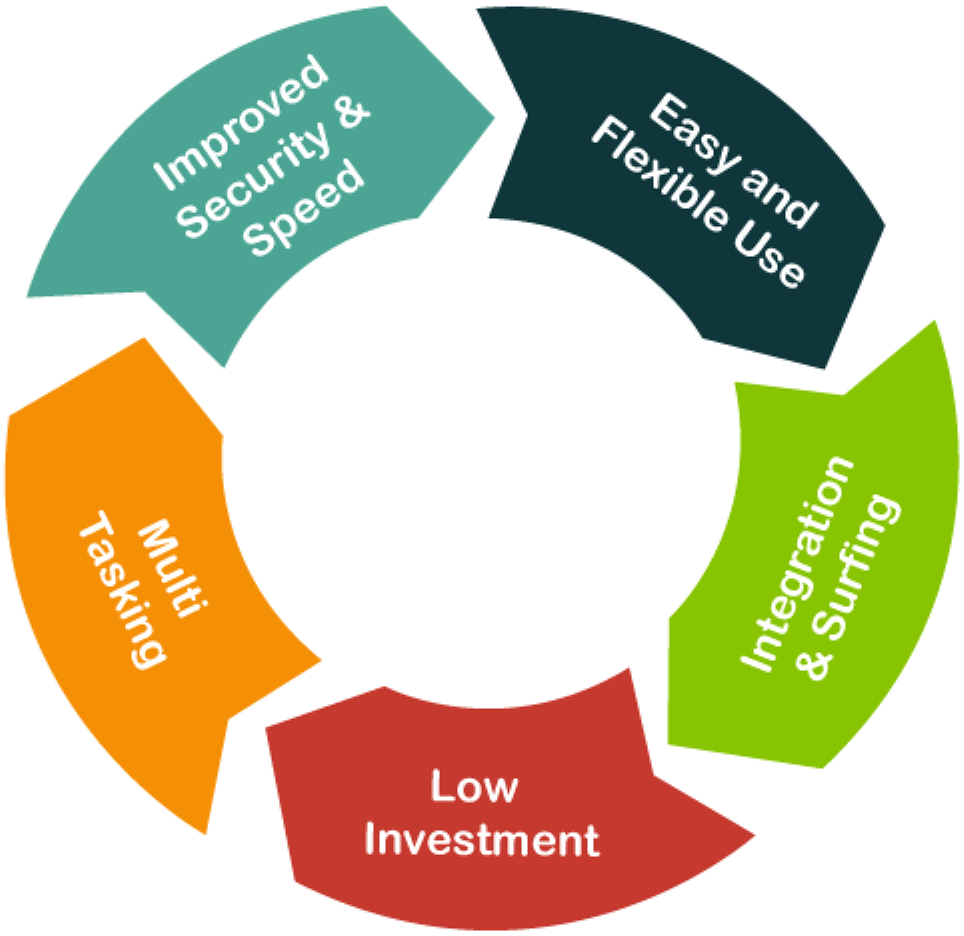
FCL (Framework Class Library)

It provides the various system functionality in the .NET Framework, that includes classes, interfaces and data types, etc. to create multiple functions and different types of application such as desktop, web, mobile application, etc. In other words, it can be defined as, it provides a base on which various applications, controls and components are built in .NET Framework.

Key Components of FCL

1. Object type
2. Implementation of data structure
3. Base data types
4. Garbage collection
5. Security and database connectivity
6. Creating common platform for window and web-based application

Characteristics of .NET Framework



1. CLR (Common Language Runtime)
2. Namespace - Predefined class and function
3. Metadata and Assemblies
4. Application domains
5. It helps to configure and deploy the .net application
6. It provides form and web-based services
7. NET and ASP.NET AJAX

8. LINQ
9. Security and Portability
10. Interoperability
11. It provides multiple environments for developing an application
12.

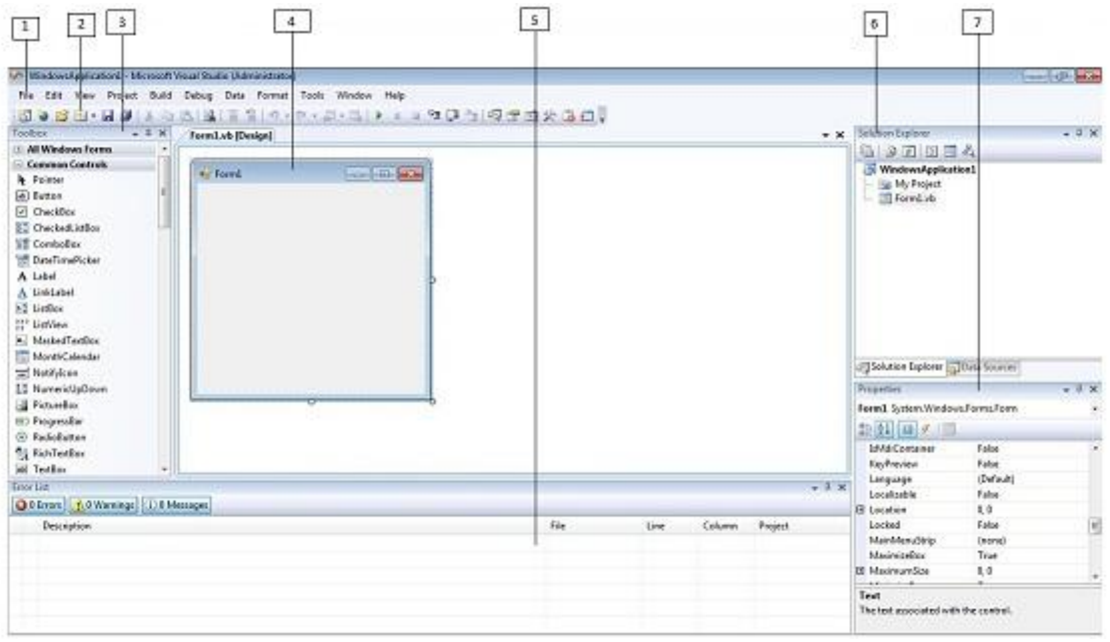
Versions of .NET Framework

1. On 13 February 2002, Microsoft launched first version of .Net framework 1.0.
2. The second version 2.0 of .net framework was launched on 22 January 2006.
3. Third version 3.0 of .Net framework was released on 21 November 2006.
4. A .Net framework version 3.5 was released on 19 November 2007.
5. Version 4.0 of .Net framework was released on 29 September 2008
6. Version 4.5 of .Net framework was released on 15 August 2012.
7. .Net framework 4.5.1 version was announced on 17 October 2013
8. On 5 May 2014, a 4.5.2 version of .Net framework was released.
9. .Net framework 4.6 version was announced on 12 November 2014
10. .Net framework 4.6.1 version was released on 30 October 2015
11. .Net framework 4.6.2 version was announced on March 30, 2016
12. .Net framework 4.7 version was announced on April 5, 2017
13. .Net framework 4.7.1 version was announced on October 17, 2017
14. Version 4.7.2 of .Net framework was released on 30 April 2018.
15. And currently we are using .Net framework version 4.8 that was released on 18 April 2019

Visual Studio

Visual Studio is a powerful and customizable programming environment that contains all the tools you need to build programs quickly and efficiently. It offers a set of tools that help you write and modify the code for your programs, and also detect and correct errors in your programs.

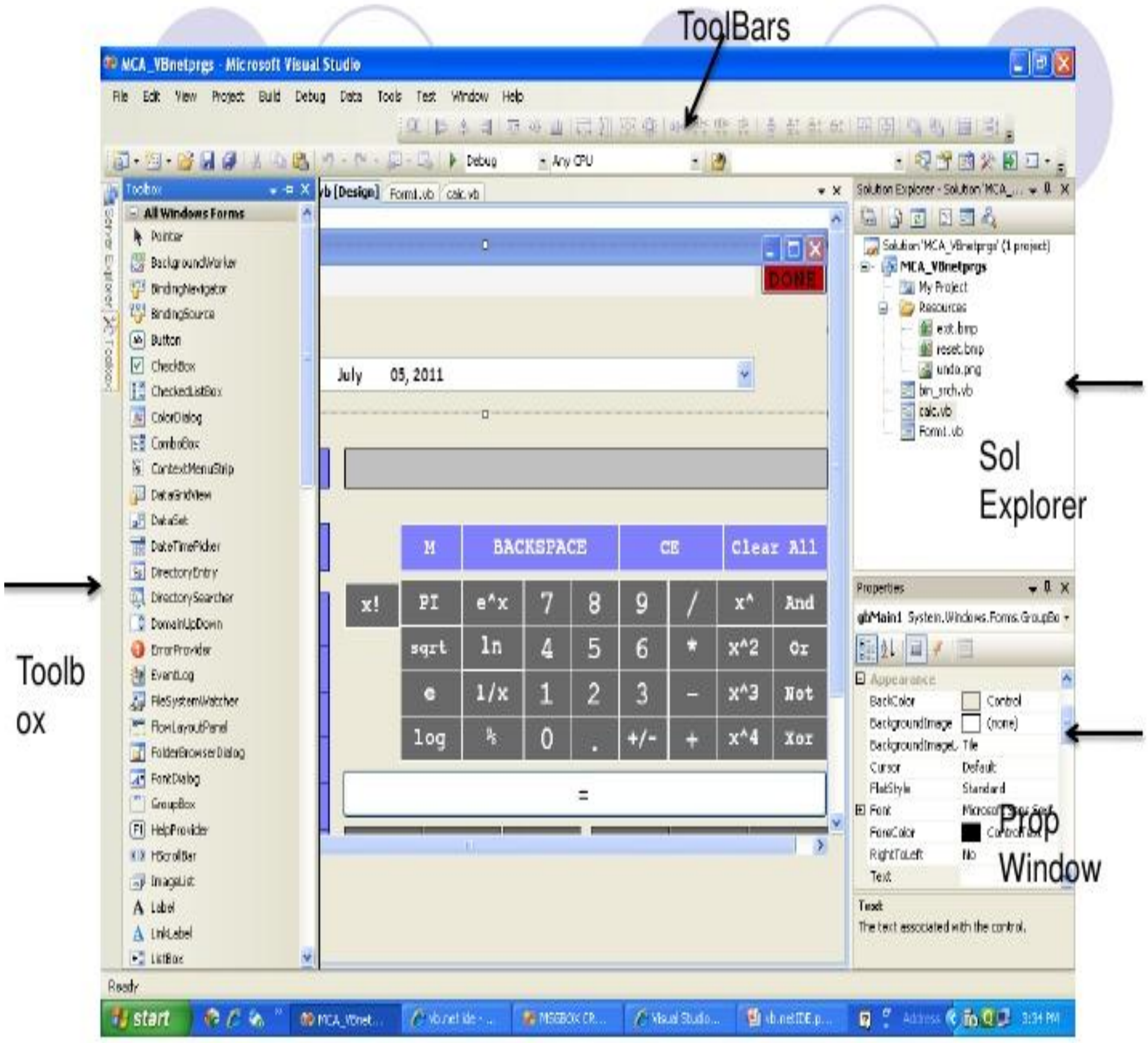
Before you start learning more about VB.NET programming, it is important to understand the development environment and identify some of the frequently using programming tools in Visual Studio IDE.



- 1. Menu Bar
- 2. Standard Toolbar
- 3. ToolBox
- 4. Forms Designer
- 5. Output Window
- 6. Solution Explorer
- 7. Properties Window

Visual Basic.NET IDE is built out of a collection of different windows. Some windows are used for writing code, some for designing interfaces, and others for getting a general overview of files or classes in your application.

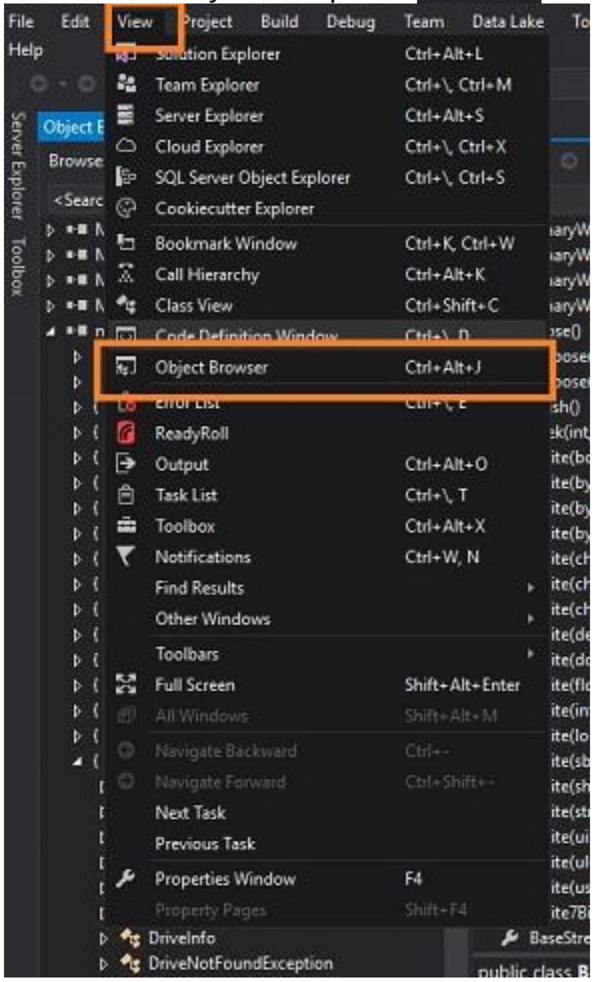
Visual Studio organizes your work in projects and solutions. A solution can contain more than one project, such as a DLL and an executable that references that DLL. From the following chapters you will learn how to use these Visual Studio features for your programming needs.



Object Browser to Explore the Framework Class Library

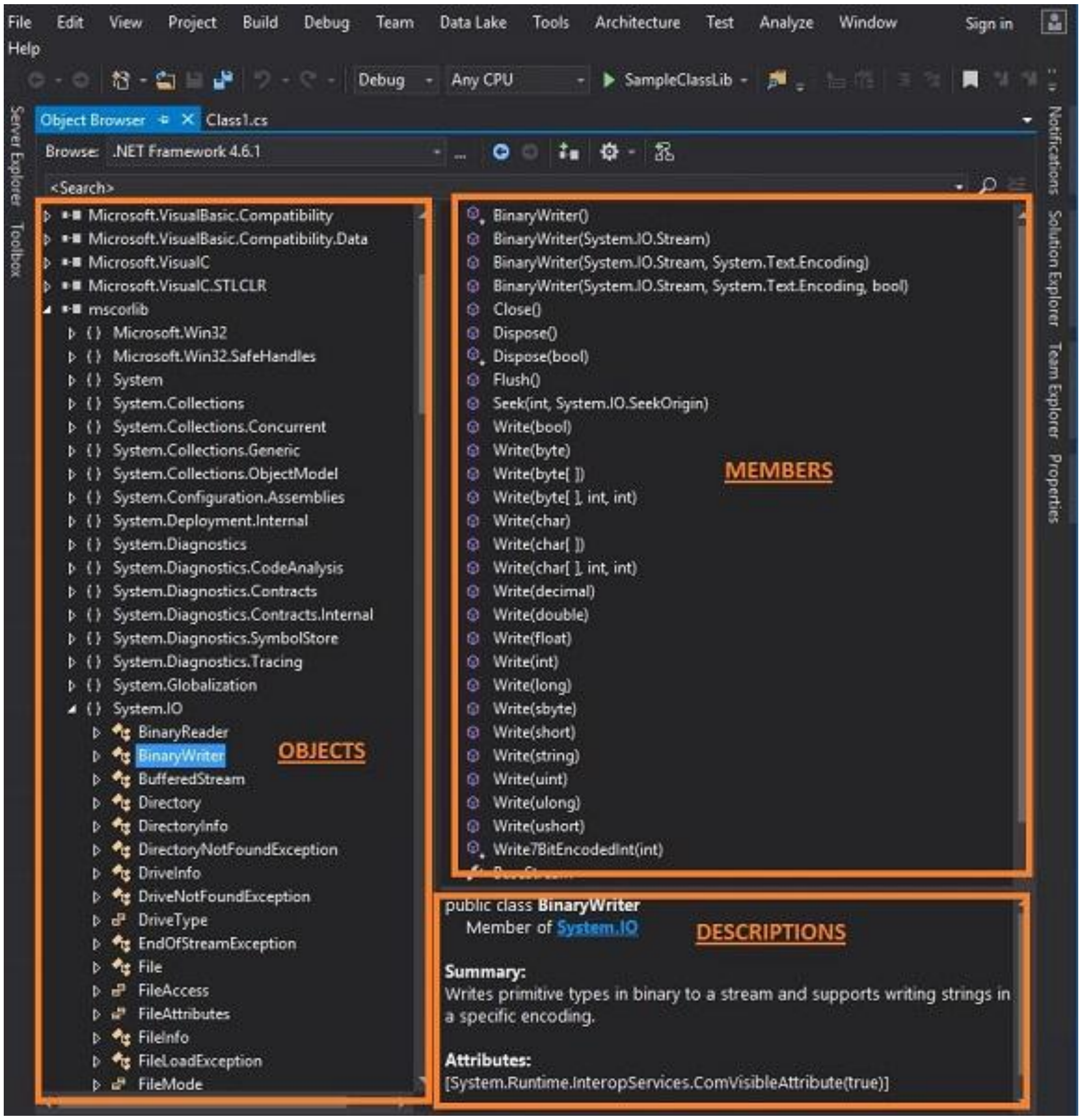
The steps below describe how to use the Visual Studio Object Browser to enumerate and examine classes in the .Net Framework Class Library

1. To open the "Object Browser' in Visual Studio, you can select **View-Object Browser** from the main menu or you can press Ctrl-Alt-J.



File Edit View Project Build Debug Team Data Lake
Help
Solution Explorer Ctrl+Alt+L
Team Explorer Ctrl+\, Ctrl+M
Server Explorer Ctrl+Alt+S
Cloud Explorer Ctrl+\, Ctrl+X
SQL Server Object Explorer Ctrl+\, Ctrl+S
Cookiecutter Explorer
Bookmark Window Ctrl+K, Ctrl+W
Call Hierarchy Ctrl+Alt+K
Class View Ctrl+Shift+C
Code Definition Window Ctrl+\, D
Object Browser Ctrl+Alt+J
Error List Ctrl+\, E
ReadyRoll
Output Ctrl+Alt+O
Task List Ctrl+\, T
Toolbox Ctrl+Alt+X
Notifications Ctrl+W, N
Find Results
Other Windows
Toolbars
Full Screen Shift+Alt+Enter
All Windows Shift+Alt+M
Navigate Backward Ctrl+-
Navigate Forward Ctrl+Shift+-
Next Task
Previous Task
Properties Window F4
Property Pages Shift+F4
DriveInfo
DriveNotFoundException

2. The Object Browser consists of three panes:
    1. Objects
    2. Members
    3. Descriptions



Debug | Any CPU

Browse: .NET Framework 4.6.1

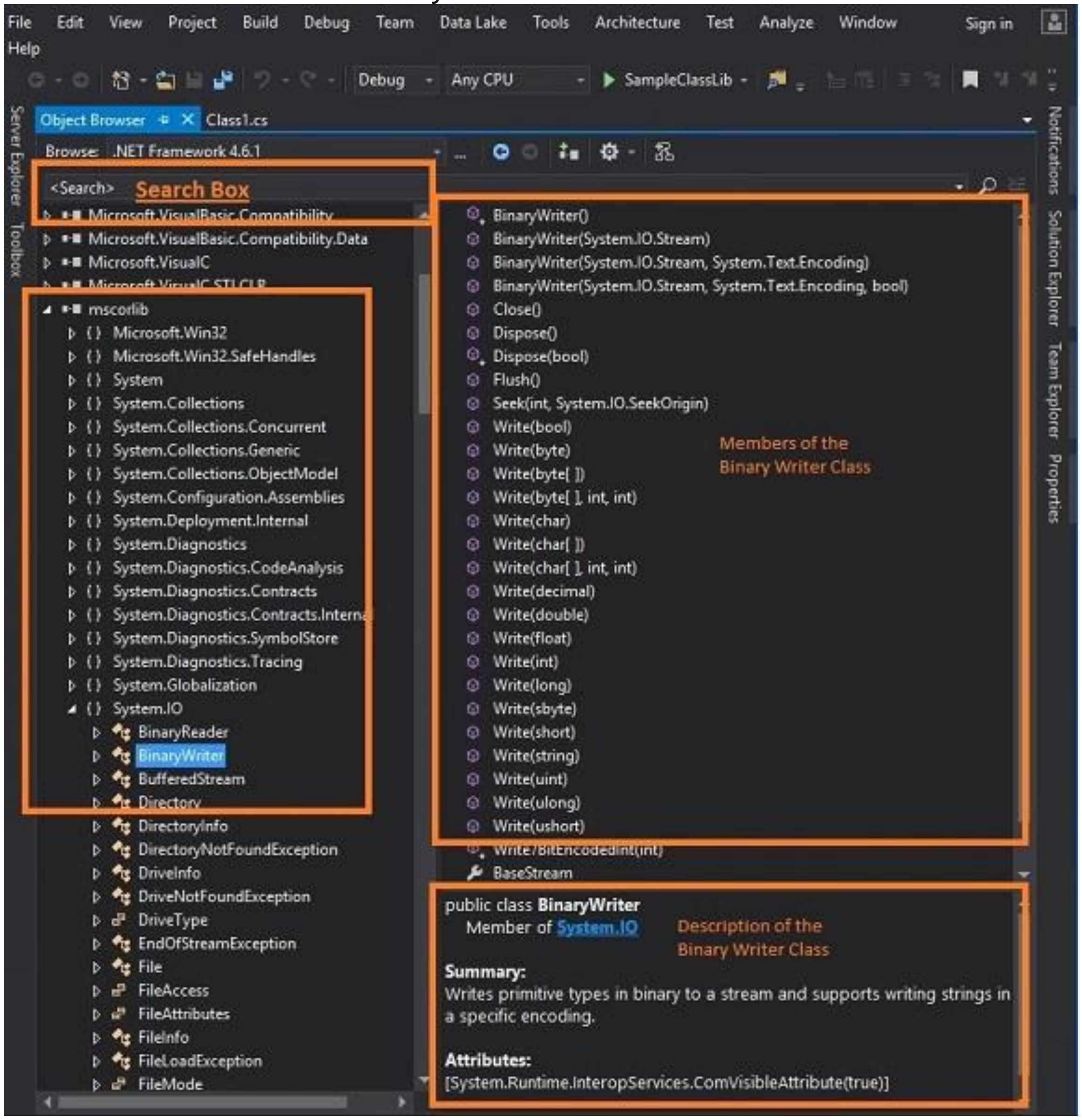
public class **BinaryWriter**
Member of System.IO

Summary:
Writes primitive types in binary to a stream and supports writing strings in a specific encoding.

Attributes:
[System.Runtime.InteropServices.ComVisibleAttribute(true)]

3. To locate a class in the framework, you can use the search box at the top of the Object Browser or you can expand the namespaces in the objects pane.
4. When a class or namespace is selected in the object pane, the members of that class or namespace are displayed in the members pane. As you can see in the image below, I have selected the mscorlib namespace, the subordinate System.IO namespace, and the BinaryWriter class within the IO namespace. With that class selected, the members pane is automatically filled with a list of the members of the BinaryWriter class and the details pane is

filled with details about the BinaryWriter class.


File Edit View Project Build Debug Team Data Lake Tools Architecture Test Analyze Window Sign in
Help
Debug
Any CPU
SampleClassLib
Object Browser
Class1.cs
Browse: .NET Framework 4.6.1
<Search>
Search Box
Members of the Binary Writer Class
Description of the Binary Writer Class
Server Explorer
Toolbox
Notifications
Solution Explorer
Team Explorer
Properties

Microsoft.VisualBasic.Compatibility
Microsoft.VisualBasic.Compatibility.Data
Microsoft.VisualC
Microsoft.VisualC.STLCLR
mscorlib
Microsoft.Win32
Microsoft.Win32.SafeHandles
System
System.Collections
System.Collections.Concurrent
System.Collections.Generic
System.Collections.ObjectModel
System.Configuration.Assemblies
System.Deployment.Internal
System.Diagnostics
System.Diagnostics.CodeAnalysis
System.Diagnostics.Contracts
System.Diagnostics.Contracts.Interna
System.Diagnostics.SymbolStore
System.Diagnostics.Tracing
System.Globalization
System.IO
BinaryReader
BinaryWriter
BufferedStream
Directory
DirectoryInfo
DirectoryNotFoundException
DriveInfo
DriveNotFoundException
DriveType
EndOfStreamException
File
FileAccess
FileAttributes
FileInfo
FileLoadException
FileMode

BinaryWriter()
BinaryWriter(System.IO.Stream)
BinaryWriter(System.IO.Stream, System.Text.Encoding)
BinaryWriter(System.IO.Stream, System.Text.Encoding, bool)
Close()
Dispose()
Dispose(bool)
Flush()
Seek(int, System.IO.SeekOrigin)
Write(bool)
Write(byte)
Write(byte[ ])
Write(byte[ ], int, int)
Write(char)
Write(char[ ])
Write(char[ ], int, int)
Write(decimal)
Write(double)
Write(float)
Write(int)
Write(long)
Write(sbyte)
Write(short)
Write(string)
Write(uint)
Write(ulong)
Write(ushort)
Write7BitEncodedInt(int)
BaseStream

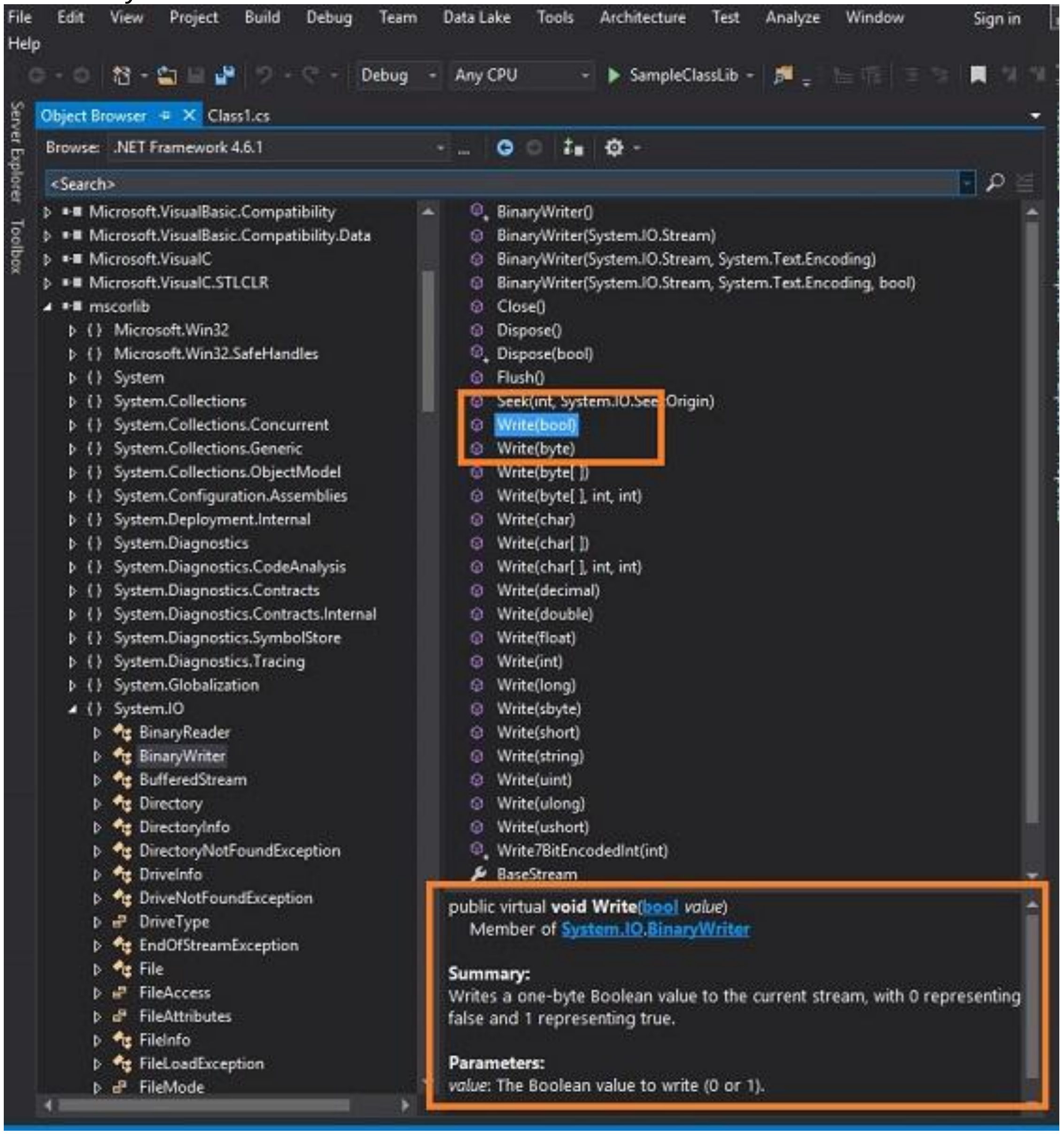
public class **BinaryWriter**
Member of System.IO

Summary:
Writes primitive types in binary to a stream and supports writing strings in a specific encoding.

Attributes:
[System.Runtime.InteropServices.ComVisibleAttribute(true)]

5. When a class member is selected in the **Members** pane, the class member details are then displayed in the descriptions pane. In this example, I have selected the WriteBool method of

the BinaryWriter class.


File Edit View Project Build Debug Team Data Lake Tools Architecture Test Analyze Window Sign in
Help
Debug
Any CPU
SampleClassLib
Server Explorer
Toolbox
Object Browser
Class1.cs
Browse: .NET Framework 4.6.1
<Search>

```
▷ Microsoft.VisualBasic.Compatibility
▷ Microsoft.VisualBasic.Compatibility.Data
▷ Microsoft.VisualC
▷ Microsoft.VisualC.STLCLR
◢ mscorlib
    ▷ { } Microsoft.Win32
    ▷ { } Microsoft.Win32.SafeHandles
    ▷ { } System
    ▷ { } System.Collections
    ▷ { } System.Collections.Concurrent
    ▷ { } System.Collections.Generic
    ▷ { } System.Collections.ObjectModel
    ▷ { } System.Configuration.Assemblies
    ▷ { } System.Deployment.Internal
    ▷ { } System.Diagnostics
    ▷ { } System.Diagnostics.CodeAnalysis
    ▷ { } System.Diagnostics.Contracts
    ▷ { } System.Diagnostics.Contracts.Internal
    ▷ { } System.Diagnostics.SymbolStore
    ▷ { } System.Diagnostics.Tracing
    ▷ { } System.Globalization
    ◢ { } System.IO
        ▷ BinaryReader
        ▷ BinaryWriter
        ▷ BufferedStream
        ▷ Directory
        ▷ DirectoryInfo
        ▷ DirectoryNotFoundException
        ▷ DriveInfo
        ▷ DriveNotFoundException
        ▷ DriveType
        ▷ EndOfStreamException
        ▷ File
        ▷ FileAccess
        ▷ FileAttributes
        ▷ FileInfo
        ▷ FileLoadException
        ▷ FileMode
```

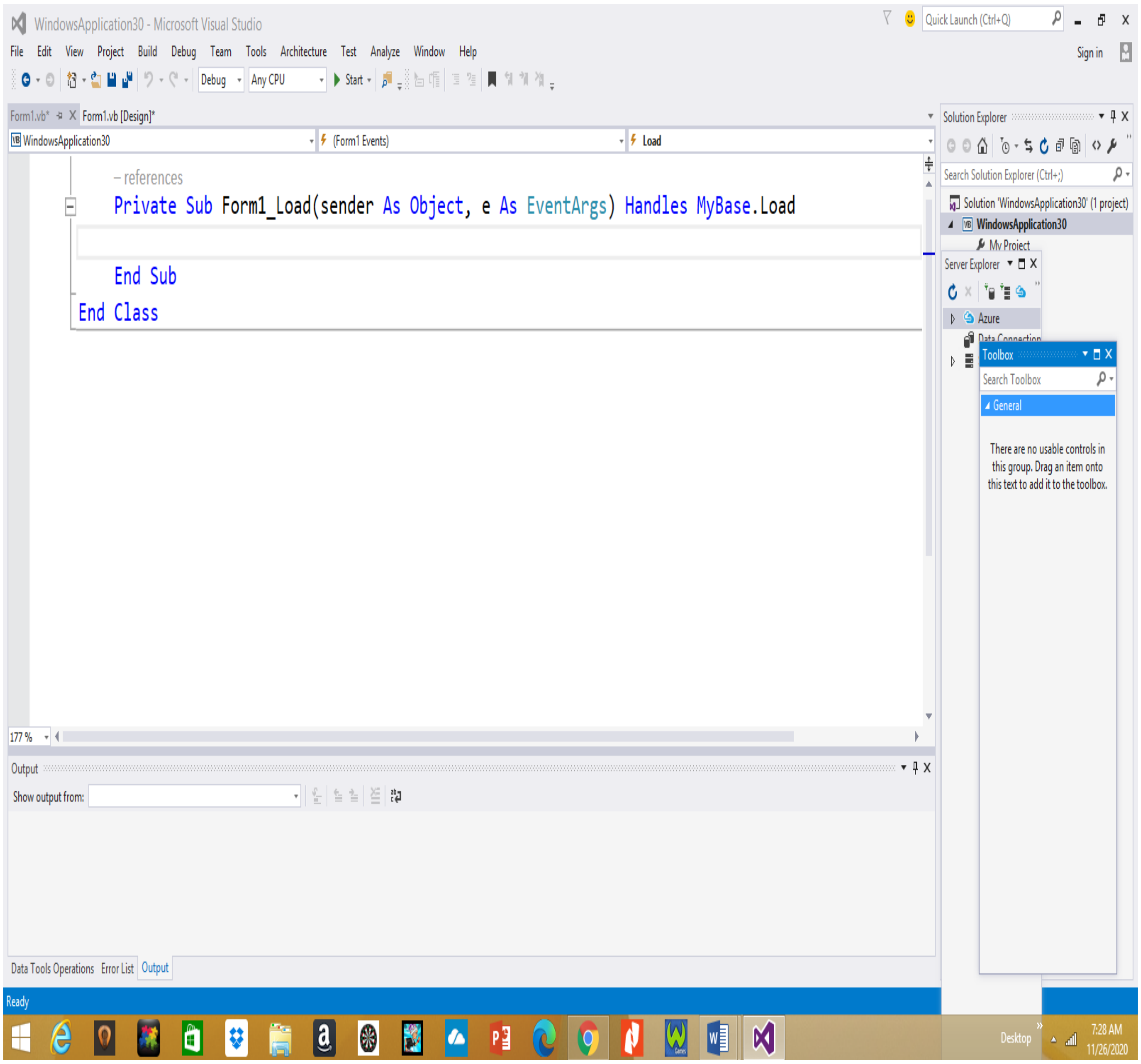
```
BinaryWriter()
BinaryWriter(System.IO.Stream)
BinaryWriter(System.IO.Stream, System.Text.Encoding)
BinaryWriter(System.IO.Stream, System.Text.Encoding, bool)
Close()
Dispose()
Dispose(bool)
Flush()
Seek(int, System.IO.SeekOrigin)
Write(bool)
Write(byte)
Write(byte[ ])
Write(byte[ ], int, int)
Write(char)
Write(char[ ])
Write(char[ ], int, int)
Write(decimal)
Write(double)
Write(float)
Write(int)
Write(long)
Write(sbyte)
Write(short)
Write(string)
Write(uint)
Write(ulong)
Write(ushort)
Write7BitEncodedInt(int)
BaseStream
```

public virtual **void Write**(bool *value*)
Member of System.IO.BinaryWriter

Summary:
Writes a one-byte Boolean value to the current stream, with 0 representing false and 1 representing true.

Parameters:
value: The Boolean value to write (0 or 1).

6. Note that the descriptions pane includes details about the member selected, to include a complete description of the signature, a summary of the member (purpose and usage), a list of parameters if appropriate and a list of all exceptions that the method might raise.



WindowsApplication30 - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Architecture Test Analyze Window Help
Debug
Any CPU
Start
Form1.vb*
Form1.vb [Design]*
WindowsApplication30
(Form1 Events)
Load
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'WindowsApplication30' (1 project)
WindowsApplication30
My Project
Server Explorer
Azure
Data Connection
Toolbox
Search Toolbox
General
There are no usable controls in this group. Drag an item onto this text to add it to the toolbox.
177 %
Output
Show output from:
Data Tools Operations
Error List
Output
Ready
Desktop
7:28 AM
11/26/2020

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

    End Sub
End Class
```

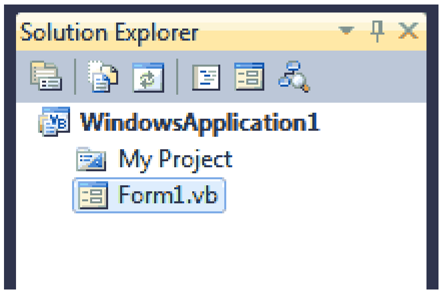
Toolbox Window

The Toolbox window contains all the controls you can use to build your application's interface. This window is usually retracted, and you must move the pointer over it to view the Toolbox. The controls in the Toolbox are organized in various tabs, so take a look at them to become familiar with the controls and their functions. A figure of ToolBox window is illustrated in "Using the Windows Form Designer in Visual Basic 2008".

In the first few chapters, we'll work with the controls in the Common Controls and Menus & Toolbars tabs. The Common Controls tab contains the icons of the most common Windows controls. The Data tab contains the icons of the objects you will use to build data-driven applications (they're explored later in this tutorial). The Dialogs tab contains controls for implementing the common dialog controls, which are so common in Windows interfaces; they're discussed in Chapter, ''Windows Controls.''

Solution Explorer Window

The Solution Explorer window contains a list of the items in the current solution. A solution can contain multiple projects, and each project can contain multiple items. The Solution Explorer displays a hierarchical list of all the components, organized by project. You can right-click any component of the project and choose Properties in the context menu to see the selected component's properties in the Properties window. If you select a project, you will see the Project Properties dialog box. You will find more information on project properties in the following chapter.



If the solution contains multiple projects, you can right-click the project you want to become the startup form and select Set As StartUp Project. You can also add items to a project with the Add Item command of the context menu, or remove a component from the project with the Exclude From Project command. This command removes the selected component from the project, but doesn't affect the component's file on the disk. The Delete command removes the selected component from the project and also deletes the component's file from the disk.

Properties Window

This window (also known as the Properties Browser) displays all the properties of the selected component and its settings. Every time you place a control on a form, you switch to this window to adjust the appearance of the control. You have already seen how to manipulate the properties of a control through the Properties window.

Many properties are set to a single value, such as a number or a string. If the possible settings of a property are relatively few, they're displayed as meaningful constants in a drop-down list. Other properties are set through a more elaborate interface. Color properties, for example, are set from within a Color dialog box that's displayed right in the Properties window. Font properties are set through the usual Font dialog box. Collections are set in a Collection Editor dialog box, in which you can enter one string for each item of the collection, as you did for the items of the ComboBox control earlier in this chapter.

If the Properties window is hidden, or if you have closed it, you can either choose *View > Properties* Window, or right-click a control on the form and choose Properties. Or you can simply press F4 to bring up this window. There will be times when a control might totally overlap another control, and you won't be able to select the hidden control and view its properties. In this case, you can select the desired control in the ComboBox at the top of the Properties window. This box contains the names of all the controls on the form, and you can select a control on the form by selecting its name on this box.

Output Window

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

Command and Immediate Windows

While testing a program, you can interrupt its execution by inserting a so-called breakpoint. When the breakpoint is reached, the program's execution is suspended, and you can execute a statement in the Immediate window. Any statement that can appear in your VB code can also be executed in the Immediate window. To evaluate an expression, enter a question mark followed by the expression you want to evaluate, as in the following samples, where result is a variable in the program you interrupted:

```
? Math.Log(35)

? "The answer is " & result.ToString
```

You can also send output to this window from within your code with the Debug.Write and Debug.WriteLine methods. Actually, this is a widely used debugging technique — to print the values of certain variables before entering a problematic area of the code. There are more elaborate tools to help you debug your application, and you'll find a discussion in the section "Debugging and Error Handling in Visual Basic 2008", but printing a few values to the Immediate window is a time-honored practice in programming with VB.

In many of the examples of this tutorial, especially in the first few chapters, I use the Debug.WriteLine statement to print something to the Immediate window. To demonstrate the use of the DateDiff() function, for example, I'll use a statement like the following:

```
Debug.WriteLine(DateDiff(DateInterval.Day, #3/9/2007#, #5/15/2008#))
```

When this statement is executed, the value 433 will appear in the Immediate window. This statement demonstrates the syntax of the DateDiff() function, which returns the difference between the two dates in days. Sending some output to the Immediate window to test a function or display the results of intermediate calculations is a common practice.

To get an idea of the functionality of the immediate window, switch back to your first sample application and insert the Stop statement after the End If statement in the button's Click event handler. Run the application, select a language, and click the button on the form. After displaying a message box, the application will reach the Stop statement and its execution will be suspended.

You’ll see the immediate window at the bottom of the IDE. If it’s not visible, open the Debug menu and choose Windows > Immediate. In the Immediate window, enter the following statement:

```
? ComboBox1.Items.Count
```

Then press Enter to execute it. Notice that IntelliSense is present while you’re typing in the immediate window. The expression prints the number of items in the Combo Box control. (Don’t worry about the numerous properties of the control and the way I present them here; they’re discussed in detail in Chapter, ‘‘Basic Windows Controls.’’) As soon as you press Enter, the value 5 will be printed on the following line.

You can also manipulate the controls on the form from within the Immediate window. Enter the following statement and press Enter to execute it:

```
ComboBox1.SelectedIndex = 4
```

The fifth item on the control will be selected (the indexing of the items begins with 0). However, you can’t see the effects of your changes, because the application isn’t running. Press F5 to resume the execution of the application and you will see that the item Cobol is now selected in the ComboBox control.

The Immediate window is available only while the application’s execution is suspended. To continue experimenting with it, click the button on the form to evaluate your choice. When the Stop statement is executed again, you’ll be switched to the Immediate window.

Unlike the Immediate window, the Command window is available at design time. The Command window allows you to access all the commands of Visual Studio by typing their names in this window. If you enter the string Edit followed by a period, you will see a list of all commands of the Edit menu, including the ones that are not visible at the time, and you can invoke any of these commands and pass arguments to them. For example, if you enter Edit. Find “Margin” in the Command window and then press Enter, the first instance of the string Margin will be located in the open code window. To start the application, you can type Debug. Start. You can add a new project to the current solution with the AddProj command, and so on. Most developers hardly ever use this window in designing or debugging applications.

Error List Window

This window is populated by the compiler with error messages, if the code can’t be successfully compiled. You can double-click an error message in this window, and the IDE will take you to the line with the statement in error — which you should fix. Change the MsgBox() function name to MsssgBox(). As soon as you leave the line with the error, the name of the function will be underlined with a wiggly red line and the following error description will appear in the Error List window:

```
Name ’MsssgBox’ is not declared
```

The IDE Components in Visual Basic 2008

The IDE of Visual Studio 2008 contains numerous components, and it will take you a while to explore them. It's practically impossible to explain in a single chapter what each tool, window, and menu command does. We'll discuss specific tools as we go along and as the topics get more and more advanced. In this section, I will go through the basic items of the IDE — the ones we'll use in the following few chapters to build simple Windows applications.

The IDE Menu

The IDE menu provides the following commands, which lead to submenus. Notice that most menus can also be displayed as toolbars. Also, not all options are available at all times. The options that cannot possibly apply to the current state of the IDE are either invisible or disabled. The Edit menu is a typical example. It's quite short when you're designing the form and quite lengthy when you edit code. The Data menu disappears altogether when you switch to the code editor — you can't use the options of this menu while editing code. If you open an XML document in the IDE, the XML command will be **added to the main menu of Visual Studio.**

File Menu

The File menu contains commands for opening and saving projects or project items, as well as commands for adding new or existing items to the current project. For the time being, use the New > Project command to create a new project, Open Project/Solution to open an existing project or solution, Save All to save all components of the current project, and the Recent Projects submenu to open one of the recent projects.

Edit Menu

The Edit menu contains the usual editing commands. Among these commands are the Advanced command and the IntelliSense command. Both commands lead to submenus, which are discussed next. Note that these two items are visible only when you're editing your code, and are invisible while you're designing a form.

Edit > Advanced Submenu

The more-interesting options of the Edit > Advanced submenu are the following:

View White Space – Space characters (necessary to indent lines of code and make it easy to read) are replaced by periods.

Word Wrap – When a code line's length exceeds the length of the code window, the line is automatically wrapped.

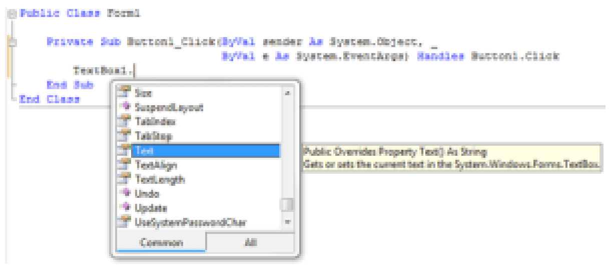
Comment Selection/Uncomment Selection – Comments are lines you insert between your code's statements to document your application. Every line that begins with a single quote is a comment; it is part of the code, but the compiler ignores it. Sometimes, we want to disable a fewlines fromour code but not delete them (because we want to be able to restore them later). A simple technique to disable a line of code is to comment it out (insert the comment symbol in front of the line). This command allows you to comment (or uncomment) large segments of code in a singlemove.

Edit > IntelliSense Submenu

The ***Edit > IntelliSense*** menu item leads to a submenu with five options, which are described next. IntelliSense is a feature of the editor (and of other Microsoft applications) that displays as much information as possible, whenever possible. When you type the name of a control and the following period, IntelliSense displays a list of the control's properties and methods, so that you can select the desired one, rather than guessing its name. When you type the name of a function and the opening parenthesis, IntelliSense will display the syntax of the function — its arguments.

The IntelliSense submenu includes the following options:

List Members – When this option is on, the editor lists all the members (properties, methods, events, and argument list) in a drop-down list. This list will appear when you enter the name of an object or control followed by a period. Then you can select the desired member from the list with the mouse or with the keyboard. Let's say your form contains a control named TextBox1 and you're writing code for this form. When you enter the name of the control followed by a period (TextBox1.), a list with the members of the TextBox control will appear (as seen in Figure 1.12).



```
Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, _
                              ByVal e As System.EventArgs) Handles Button1.Click
        TextBox1.
    End Sub
End Class
```

Public Overrides Property Text() As String
Gets or sets the current text in the System.Windows.Forms.TextBox.

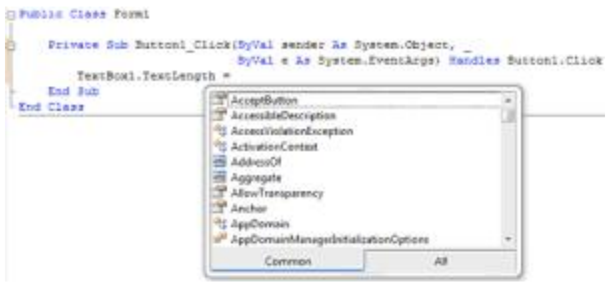
Figure 1.12 – Viewing the members of a control in the IntelliSense drop-down list

In addition, a description of the selected member is displayed in a ToolTip box, as you can see in the same figure. Select the Text property and then enter the equal sign, followed by a string in quotes, as follows:

```
TextBox1.Text = "Your User Name"
```

If you select a property that can accept a limited number of settings, you will see the names of the appropriate constants in a drop-down list. If you enter the following statement, you will see the constants you can assign to the property (see Figure 1.13):

```
TextBox1.TextAlign =
```



```
Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, _
                              ByVal e As System.EventArgs) Handles Button1.Click
        TextBox1.TextLength =
    End Sub
End Class
```

Figure 1.13 – Viewing the possible settings of a property in the IntelliSense drop-down list

Again, you can select the desired value with the mouse. The drop-down list with the members of a control or object (the Members list) remains open until you type a terminator key (the Esc or End key) or select a member by pressing the space bar or the Enter key.

Parameter Info – While editing code, you can move the pointer over a variable, method, or property and see its declaration in a yellow pop-up box. You can also jump to the variable's definition or the body of a procedure by choosing Go To Definition from the context menu that will appear if you right-click the variable or method name in the code window.

Quick Info – This is another IntelliSense feature that displays information about commands and functions. When you type the opening parenthesis following the name of a function, for example, the function's arguments will be displayed in a ToolTip box (a yellow horizontal box). The first argument appears in bold font; after entering a value for this argument, the next one is shown in bold. If an argument accepts a fixed number of settings, these values will appear in a drop-down list, as explained previously.

Complete Word – The Complete Word feature enables you to complete the current word by pressing Ctrl+spacebar. For example, if you type TextB and then press Ctrl+spacebar, you will see a list of words that you're most likely to type (TextBox, TextBox1, and so on).

Insert Snippet – This command opens the Insert Snippet window at the current location in the code editor window. Code snippets, which are an interesting feature of Visual Studio 2008, are discussed in the section ''Using Code Snippets'' later in this chapter.

Edit > Outlining Submenu

A practical application contains a substantial amount of code in a large number of event handlers and custom procedures (subroutines and functions). To simplify the management of the code window, the Outlining submenu contains commands that collapse and expand the various procedures.

Let's say you're finished editing the Click event handlers of several buttons on the form. You can reduce these event handlers to a single line that shows the names of the procedures and a plus sign in front of them. You can expand a procedure's listing at any time by clicking the plus sign in front of its name. When you do so, a minus sign appears in front of the procedure's name, and you can click it to collapse the body of the procedure again. The Outlining submenu contains commands to handle the outlining of the various procedures, or turn off outlining and view the complete listings of all procedures. You will use these commands as you write applications with substantial amounts of code:

Toggle Outlining Expansion – This option lets you change the outline mode of the current procedure. If the procedure's definition is collapsed, the code is expanded, and vice versa.

Toggle All Outlining – This option is similar to the Toggle Outlining Expansion option, but it toggles the outline mode of the current document. A form is reduced to a single statement. A file with multiple classes is reduced to one line per class.

Stop Outlining – This option turns off outlining and adds a new command to the Outlining submenu, Start Automatic Outlining, which you can select to turn on automatic outlining again.

Collapse To Definitions – This option reduces the listing to a list of procedure headers.

View Menu

This menu contains commands to display any toolbar or window of the IDE. You have already seen the Toolbars menu (in the ‘‘Starting a New Project in Visual Basic 2008’’ section). The Other Windows command leads to a submenu with the names of some standard windows, including the Output and Command windows. The Output window is the console of the application. The compiler’s messages, for example, are displayed in the Output window. The Command window allows you to enter and execute statements. When you debug an application, you can stop it and enter VB statements in the Command window.

Project Menu

This menu contains commands for adding items to the current project (an item can be a form, a file, a component, or even another project). The last option in this menu is the Project Properties command, which opens the project’s Properties Pages. The Add Reference and Add Web Reference commands allow you to add references to .NET components and web components, respectively.

Build Menu

The Build menu contains commands for building (compiling) your project. The two basic commands in this menu are Build and Rebuild All. The Build command compiles (builds the executable) of the entire solution, but it doesn’t compile any components of the project that haven’t changed since the last build. The Rebuild All command does the same, but it clears any existing files and builds the solution from scratch.

Debug Menu

This menu contains commands to start or end an application, as well as the basic debugging tools.

Data Menu

This menu contains commands you will use with projects that access data.

Format Menu

The Format menu, which is visible only while you design a Windows or web form, contains commands for aligning the controls on the form. The commands of this menu are discussed in Chapter “GUI Design and Event-Driven Programming in VB”. The Format menu is invisible when you work in the code editor — its commands apply to the visible elements of the interface.

Tools Menu

This menu contains a list of useful tools, such as the Macros command, which leads to a submenu with commands for creating macros. Just as you can create macros in a Microsoft Office application to simplify many tasks, you can create macros to automate many of the repetitive tasks you perform in the IDE. The last command in this menu, the Options command, leads to the Options dialog box, in which you can fully customize the environment. The Choose Toolbox Items command opens a dialog box that enables you to add more controls to the Toolbox. In Chapter, ‘‘Building Custom Windows

Controls in Visual Basic 2008'' you'll learn how to design custom controls and add them to the Toolbox.

Window Menu

This is the typical Window menu of any Windows application. In addition to the list of open windows, it also contains the Hide command, which hides all toolboxes, leaving the entire window of the IDE devoted to the code editor or the Form Designer. The toolboxes don't disappear completely; they're all retracted, and you can see their tabs on the left and right edges of the IDE window. To expand a toolbox, just hover the mouse pointer over the corresponding tab.

Help Menu

This menu contains the various help options. The Dynamic Help command opens the Dynamic Help window, which is populated with topics that apply to the current operation. The Index command opens the Index window, in which you can enter a topic and get help on the specific topic.

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" −

```
ImportsSystem
ModuleModule1
'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 of a class that contains the data and procedures that your program uses.
- Classes or Modules generally would contain more than one procedure. Procedures contain the executable code, or in other words, they define the behavior of the class. A procedure could be any of the following −
    - Function
    - Sub
    - Operator
    - Get
    - Set
    - AddHandler
    - RemoveHandler
    - RaiseEvent
- The next line('This program) will be ignored by the compiler and it has been put to add additional comments in the program.
- The next line defines the Main procedure, which is the entry point for all VB.Net programs. The Main procedure states what the module or class will do when executed.
- The Main procedure specifies its behavior with the statement

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

Compile & Execute VB.Net Program

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

- Start Visual Studio.
- On the menu bar, choose File → New → Project.
- Choose Visual Basic from templates
- Choose Console Application.
- Specify a name and location for your project using the Browse button, and then choose the OK button.
- The new project appears in Solution Explorer.
- Write code in the Code Editor.

- Click the Run button or the F5 key to run the project. A Command Prompt window appears that contains the line Hello World.

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.

VB.Net - Basic Controls

An object is a type of user interface element you create on a Visual Basic form by using a toolbox control. In fact, in Visual Basic, the form itself is 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* −

```
PublicClassForm1
PrivateSubButton1_Click(ByVal sender AsSystem.Object,ByVal e
AsSystem.EventArgs)
HandlesButton1.Click
MessageBox.Show("Hello, World")
EndSub
EndClass
```

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 −

```
PrivateSubForm1_Load(sender AsObject, e AsEventArgs)HandlesMyBase.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 −

Sr.No.	Widget & Description
1	Forms The container for all the controls that make up the user interface.
2	TextBox It represents a Windows text box control.
3	Label It represents a standard Windows label.
4	Button It represents a Windows button control.
5	ListBox It represents a Windows control to display a list of items.
6	ComboBox It represents a Windows combo box control.
7	RadioButton It enables the user to select a single option from a group of choices when paired with other RadioButton controls.
8	CheckBox It represents a Windows CheckBox.
9	PictureBox

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

• DATA TYPES IN VB.NET

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.

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 −

```
ModuleDataTypes
SubMain()
Dim b AsByte
Dim n AsInteger
Dim si AsSingle
Dim d AsDouble
Dim da AsDate
Dim c AsChar
Dim s AsString
Dim bl AsBoolean

      b =1
      n =1234567
      si =0.12345678901234566
      d =0.12345678901234566
      da =Today
      c ="U"c
      s ="Me"

IfScriptEngine="VB"Then
         bl =True
Else
         bl =False
EndIf

If bl Then
'the oath taking
         Console.Write(c & " and," & s & vbCrLf)
         Console.WriteLine("declaring on the day of: {0}", da)
         Console.WriteLine("We will learn VB.Net seriously")
         Console.WriteLine("Lets see what happens to the floating point
variables:")
         Console.WriteLine("The Single: {0}, The Double: {1}", si, d)
      End If
      Console.ReadKey()
   End Sub
End Module
```

When the above code is compiled and executed, it produces 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 −

Sr.No.	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 −

```
ModuleDataTypes
SubMain()
Dim n AsInteger
Dim da AsDate
Dim bl AsBoolean=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()
EndSub
EndModule
```

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

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

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.

VARIABLE

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 −

```
DimStudentIDAsInteger
DimStudentNameAsString
DimSalaryAsDouble
Dim count1, count2 AsInteger
Dim status AsBoolean
Dim exitButton AsNewSystem.Windows.Forms.Button
Dim lastTime, nextTime AsDate
```

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 AsDouble
pi =3.14159
```

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

```
DimStudentIDAsInteger=100
DimStudentNameAsString="Bill Smith"
```

Example

Try the following example which makes use of various types of variables −

```
Module variablesNdataypes
SubMain()
Dim a AsShort
Dim b AsInteger
Dim c AsDouble

      a =10
      b =20
      c = a + b
Console.WriteLine("a = {0}, b = {1}, c = {2}", a, b, c)
Console.ReadLine()
EndSub
EndModule
```

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 AsString
message =Console.ReadLine
```

The following example demonstrates it −

```
Module variablesNdataypes
SubMain()
Dim message AsString
Console.Write("Enter message: ")
      message =Console.ReadLine
Console.WriteLine()
Console.WriteLine("Your Message: {0}", message)
Console.ReadLine()
EndSub
EndModule
```

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
```

CONSTANTS

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 AsLong=4999
PublicConst message AsString="HELLO"
PrivateConst piValue AsDouble=3.1415
```

Example

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

```
Module constantsNenum
SubMain()
Const PI =3.14149
Dim radius, area AsSingle
      radius =7
      area = PI * radius * radius
Console.WriteLine("Area = "&Str(area))
Console.ReadKey()
EndSub
EndModule
```

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 −

Sr.No.	Constant & Description
1	**vbCrLf** Carriage return/linefeed character combination.
2	**vbCr** Carriage return character.
3	**vbLf** Linefeed character.
4	**vbNewLine** Newline character.
5	**vbNullChar** Null character.
6	**vbNullString** Not the same as a zero-length string (""); used for calling external procedures.
7	**vbObjectError** Error number. User-defined error numbers should be greater than this value. For example: Err.Raise(Number) = vbObjectError + 1000
8	**vbTab** Tab character.
9	**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,

```
EnumColors
   red =1
   orange =2
   yellow =3
   green =4
   azure =5
   blue =6
   violet =7
EndEnum
```

Example

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

```
Module constantsNenum
EnumColors
      red =1
      orange =2
      yellow =3
      green =4
      azure =5
      blue =6
      violet =7
EndEnum

SubMain()
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()
EndSub
EndModule
```

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
```

VALUE TYPE

A data type is a value type if it holds the data within its own memory allocation. Value types are stored directly on the stack. Value types can not contain the value null. We assign a value to that variable like this: x=11. When a variable of value type goes out of scope, it is destroyed and it's memory is reclaimed.

Value types include the following:

- All numeric data types
- Boolean, Char, and Date
- All structures, even if their members are reference types
- Enumerations, since their underlying type is always SByte, Short, Integer, Long, Byte, UShort, UInteger, or ULong

For example

The following code defines an int type variable. int type is a value type.

```
Module Module1
    Sub Main()
        Dim m As Integer = 5
        Dim n As Integer = m
        m = 3
        Console.WriteLine("m=" & m)
        Console.WriteLine("n=" & n)
    End Sub
End Module
```

REFERENCE TYPE

A reference type contains a pointer to another memory location that holds the data.while Reference types are stored on the run-time heap. Value types can contain the value null. Creating a variable of reference type is a two-step process, declare and instantiate. The first step is to declare a variable as that type. The second step, instantiation, creates the object.

Reference types include the following:

- String
- All arrays, even if their elements are value types
- Class types, such as Form
- Delegates

For Example

```
Module Module1
    Sub Main()
        Dim objX As New System.Text.StringBuilder(" Rohatash Kumasr")
        Dim objY As System.Text.StringBuilder
        objY = objX

        objX.Replace("World", "Test")

        Console.WriteLine(objY.ToString())
    End Sub
End Module
```

Difference

Value type Data types	**Reference type Data types**
1) Whenever a data type is derived from structure then it is said to be value type data type.	1) Whenever a data type is derived from a class definition then it is said to be value type data type.

2) The variable information and the value will be maintained at the stack memory. Ex: public I as integer = 10I [10] s to 3 stack memory	2) The variable information will be maintained at the stack memory and value will be maintained at heap memory .Ex: public S as string = VB.NetS[H10T] 5105stack memory ----> [vbrowser] H107 heap memory
3) Inheritance is not supported	3) Inheritance is supported.
4) Default values will be assignedEx: integer à 0Boolean à false	4) Default value for any reference type member type member will be null value.
5) value type data types are bytes, s bytes, integer, u integer, short, u short, long, double, decimal, Boolean, char, date time, enum & structure etc.	5) Reference type data types are string, object, interface, delegate and class.

Boxing & Unboxing

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

Boxing

- Boxing is a mechanism in which value type is converted into reference type.
- It is implicit conversion process in which object type (super type) is used.
- In this process type and value both are stored in object type

Unboxing

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

Program to show Boxing and Unboxing:

```
Module Module1
   Sub Main()
      Dim i As Integer = 10
      Dim j As Integer       ' boxing
      Dim o As Object
      o = i       'unboxing
      j = CInt(o)
      Console.WriteLine("value of o object : " & o)
      Console.WriteLine("Value of j : " & j)
      Console.ReadLine()
   End Sub
End Module
```

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 −

Sr.No	Modifier	Description
1	Ansi	Specifies that Visual Basic should marshal all strings to American National Standards Institute (ANSI) values regardless of the name of the external procedure being declared.
2	Assembly	Specifies that an attribute at the beginning of a source file applies to the entire assembly.
3	Async	Indicates that the method or lambda expression that it modifies is asynchronous. Such methods are referred to as async methods. The caller of an async method can resume its work without waiting for the async method to finish.
4	Auto	The *charsetmodifier* part in the Declare statement supplies the character set information for marshaling strings during a call to the external procedure. It also affects how Visual Basic searches the external file for the external procedure name. The Auto modifier specifies that Visual Basic should marshal strings according to .NET Framework rules.
5	ByRef	Specifies that an argument is passed by reference, i.e., the called procedure can change the value of a variable underlying the argument in the calling code. It is used under the contexts of − • Declare Statement • Function Statement • Sub Statement
6	ByVal	Specifies that an argument is passed in such a way that the called procedure or property cannot change the value of a variable

		underlying the argument in the calling code. It is used under the contexts of − • Declare Statement • 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.

STATEMENT

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 −

Sr.No	Statements and Description	Example

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

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

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

4	**Class Statement**
	Declares the name of a class and introduces the definition of the variables, properties, events, and procedures that the class comprises.
		`ClassBox`
		`Public length AsDouble`
		`Public breadth AsDouble`
		`Public height AsDouble`
		`EndClass`

5	**Structure Statement**
	
	Declares the name of a structure and introduces the definition of the variables, properties, events, and procedures that the structure comprises.
		`StructureBox`
		`Public length AsDouble`
		`Public breadth AsDouble`
		`Public height AsDouble`
		`EndStructure`

6	**Module Statement**
	
	Declares the name of a module and introduces the definition of the variables, properties, events, and procedures that the module comprises.
		`PublicModule myModule`
		`SubMain()`
		`Dim user AsString=`
		`InputBox("What is your`
		`name?")`
		`MsgBox("User name is"& user)`
		`EndSub`
		`EndModule`

7	**Interface Statement**
	
	Declares the name of an interface and introduces the definitions of the members that the interface comprises.
		`PublicInterfaceMyInterface`
		`Sub doSomething()`
		`EndInterface`

8	**Function Statement**
	
	Declares the name, parameters, and code that define a Function procedure.
		`Function myFunction`
		`(ByVal n AsInteger)AsDouble`
		`Return5.87* n`
		`EndFunction`

9	**Sub Statement**
	
	Declares the name, parameters, and code that define a Sub procedure.
		`Sub mySub(ByVal s AsString)`
		`Return`
		`EndSub`

10	**Declare Statement**
	
	Declares a reference to a procedure implemented in an external file.
		`DeclareFunction getUserName`
		`Lib"advapi32.dll"`
		`Alias"GetUserNameA"`
		`(`
		`ByVal lpBuffer AsString,`
		`ByRef nSize`
		`AsInteger)AsInteger`

11	**Operator Statement**
	
	Declares the operator symbol, operands, and code that define an operator procedure on a class or structure.
		`PublicSharedOperator+`
		`(ByVal x As obj,ByVal y As`
		`obj)As obj`
		`Dim r AsNew obj`
		`' implemention code for r = x`
		`+ y`
		`   Return r`
		`End Operator`

12	**Property Statement**
	
	Declares the name of a property, and the property procedures used to store and retrieve the value of the property.
		`ReadOnlyProperty`
		`quote()AsString`
		`Get`
		`Return quoteString`
		`EndGet`
		`EndProperty`

13	**Event Statement** Declares a user-defined event.	`PublicEventFinished()`

14	**Delegate Statement**
	
	Used to declare a delegate.
		`DelegateFunctionMathOperator(`
		`ByVal x AsDouble,`
		`ByVal y AsDouble`
		`)AsDouble`

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
SubMain()
'local variable definition '
Dim a AsInteger=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")
EndIf
Console.WriteLine("value of a is : {0}", a)
Console.ReadLine()
EndSub
EndModule
```

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

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

COMBOBOX CONTROL

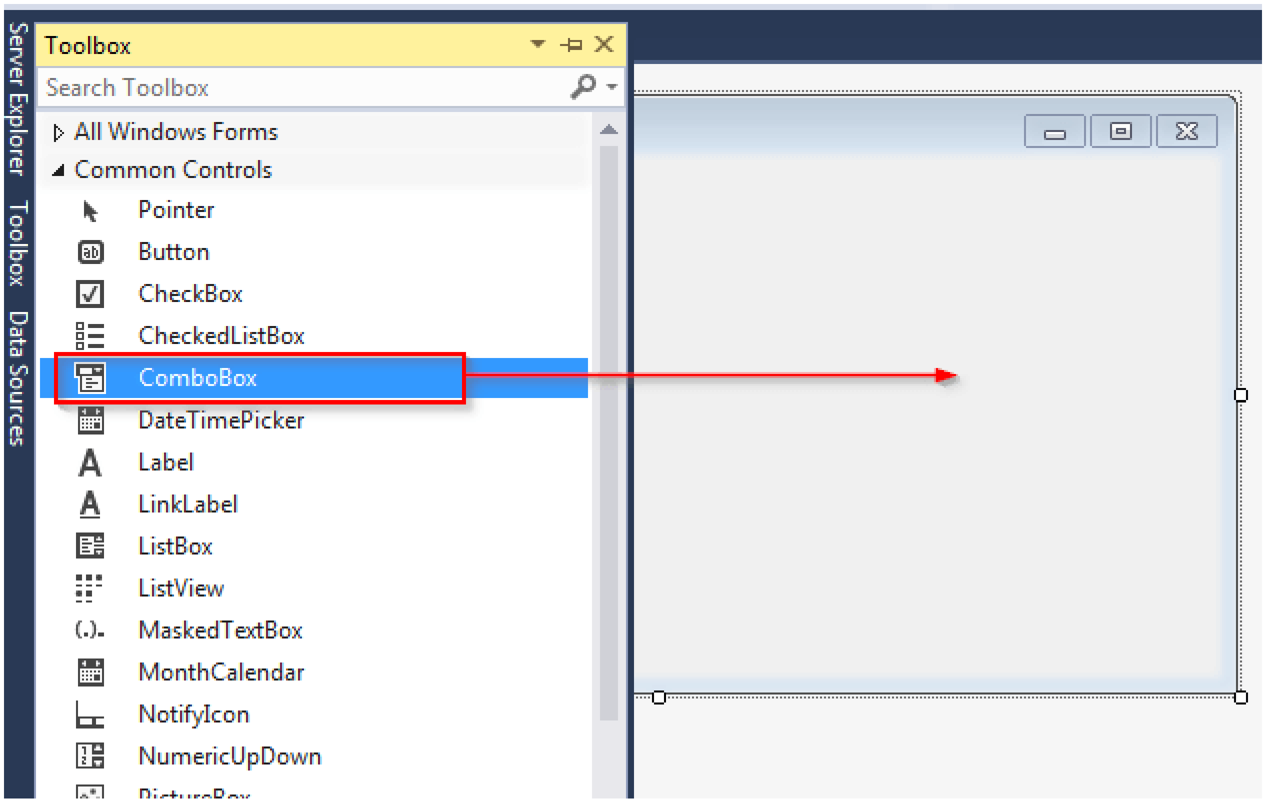
The combobox control helps you to display a drop-down list with many items. See it as a combination of a textbox in which a user enters text and a dropdown list from which a user selects an item. Note that the combobox shows one item at a time.

Creating a Combobox

A ComboBox can be created as follows:

Step 1) Create a new Application.

Step 2) Drag a combobox control from the toolbox to the form.



You will have created a combobox control.

Adding Items to Combobox

Now that we have created a combobox, let us demonstrate how to add items to it.

Double click the combobox control that you have added. You will be moved from the design tab to the tab with code.

To add an item to a combobox control, we use the Items property. Let us demonstrate this by adding two items to the combobox, Male and Female:

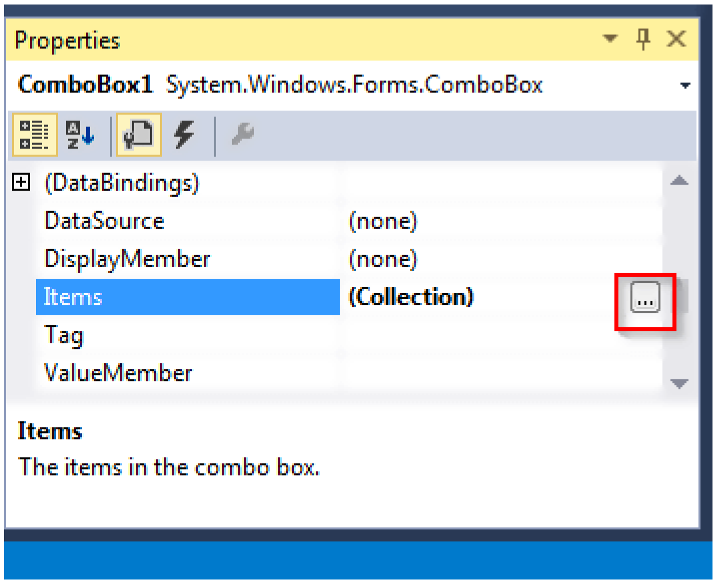
```
ComboBox1.Items.Add("Male")
ComboBox1.Items.Add("Female")
```

We can also choose to add items to the combobox at design time from the Properties window. Here are the steps:

Step 1) Open the design tab and click the combobox control.

Step 2) Move to the Properties window and view the Items option.

Step 3) Click the … located to the right of (Collection).



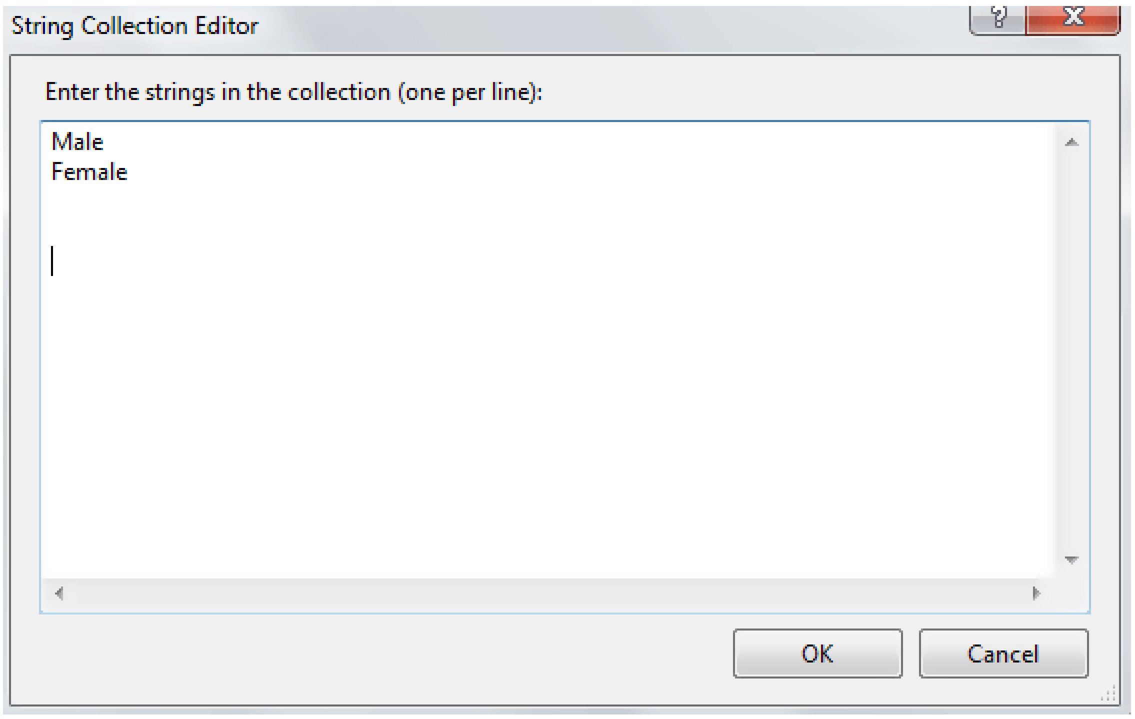
Properties

ComboBox1 System.Windows.Forms.ComboBox

(DataBindings)	
DataSource	(none)
DisplayMember	(none)
Items	(Collection)
Tag	
ValueMember	

Items
The items in the combo box.

Step 4) You will see a new window. This is where you should add items to the combobox, as shown below:



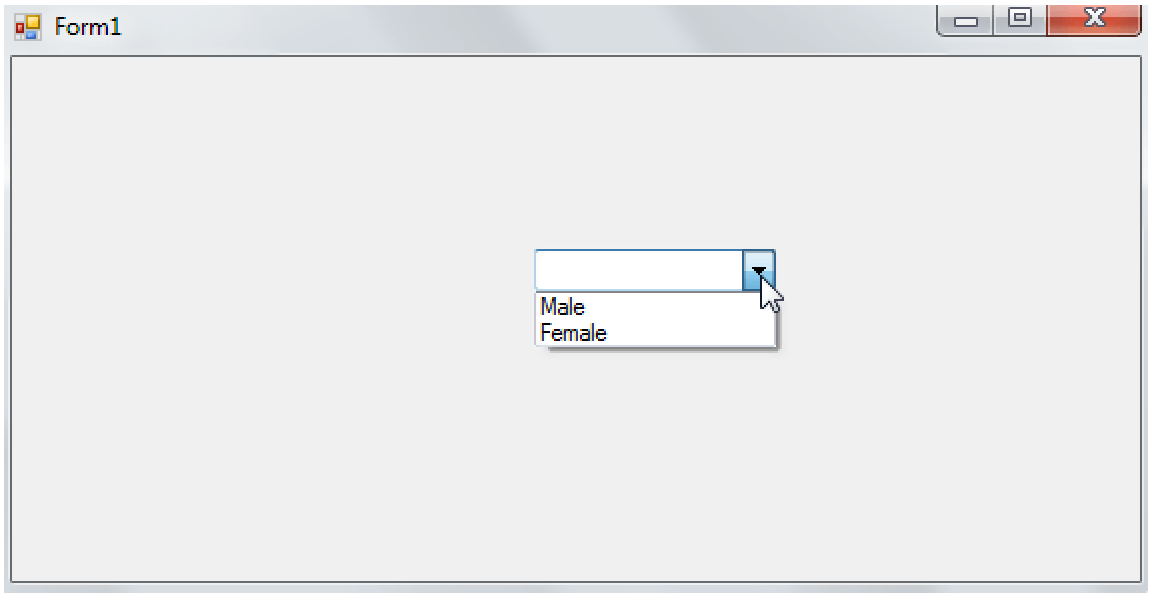
String Collection Editor

Enter the strings in the collection (one per line):

```
Male
Female
```

Step 5) Once done with typing the items, click the OK button.

Step 6) Click the Start button from the top toolbar and click the dropdown icon on the combobox.



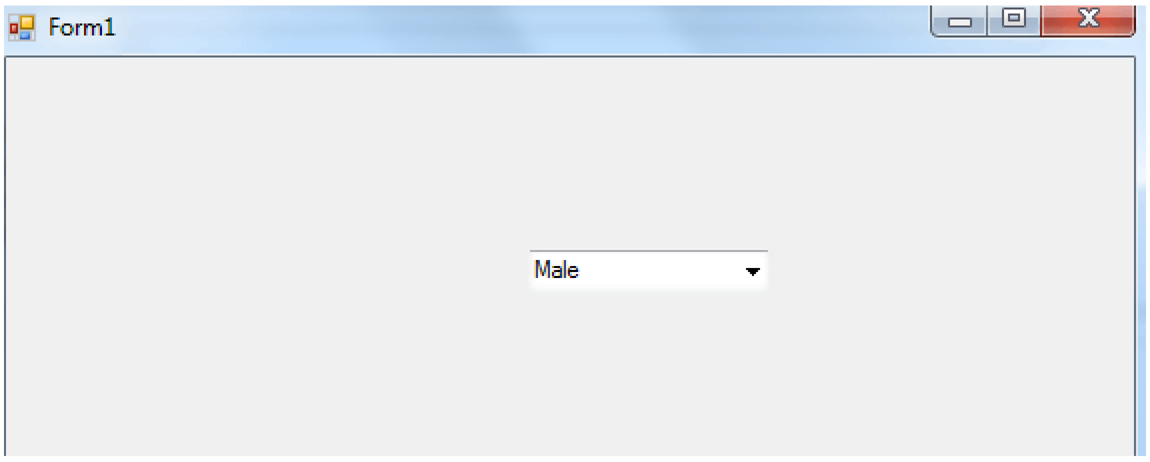
The items were successfully added to the combobox control.

Selecting Combobox Items

You may need to set the default item that will be selected when the form is loaded. You can achieve this via the SelectedItem() method. For example, to set the default selected gender to Male, you can use the following statement:

```
ComboBox1.SelectedItem = "Male"
```

When you run the code, the combobox control should be as shown below:



Retrieving Combobox Values

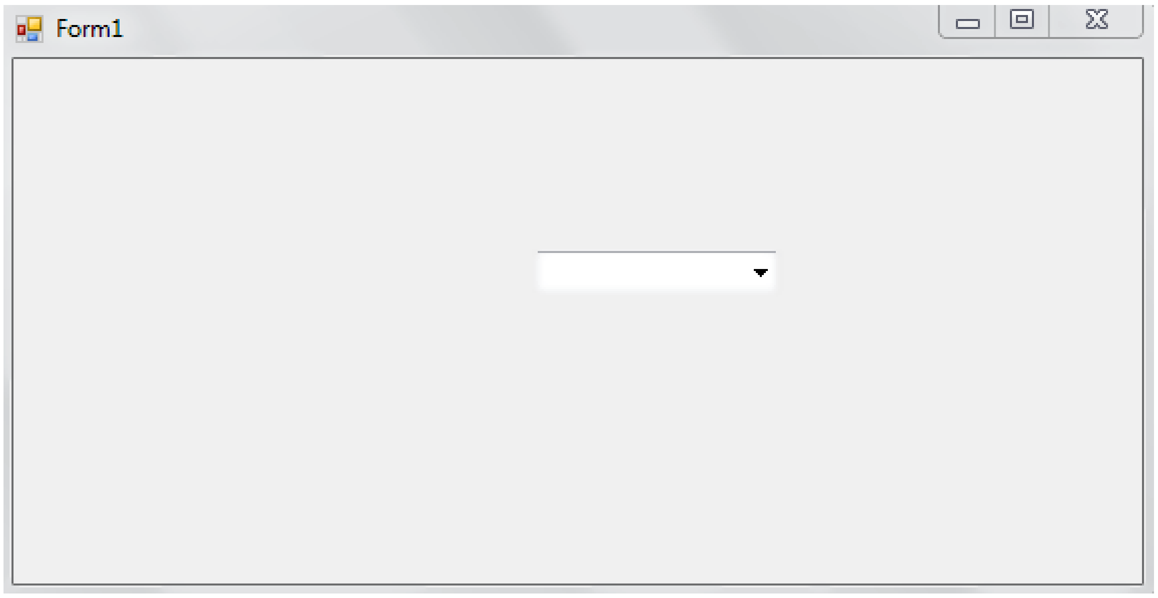
You can get the selected item from your combobox. This can be done using the text property. Let us demonstrate this using our above combobox with two items that is, Male and Female. Follow the steps given below:

Step 1) Double click the combobox to open the tab with VB.NET code.

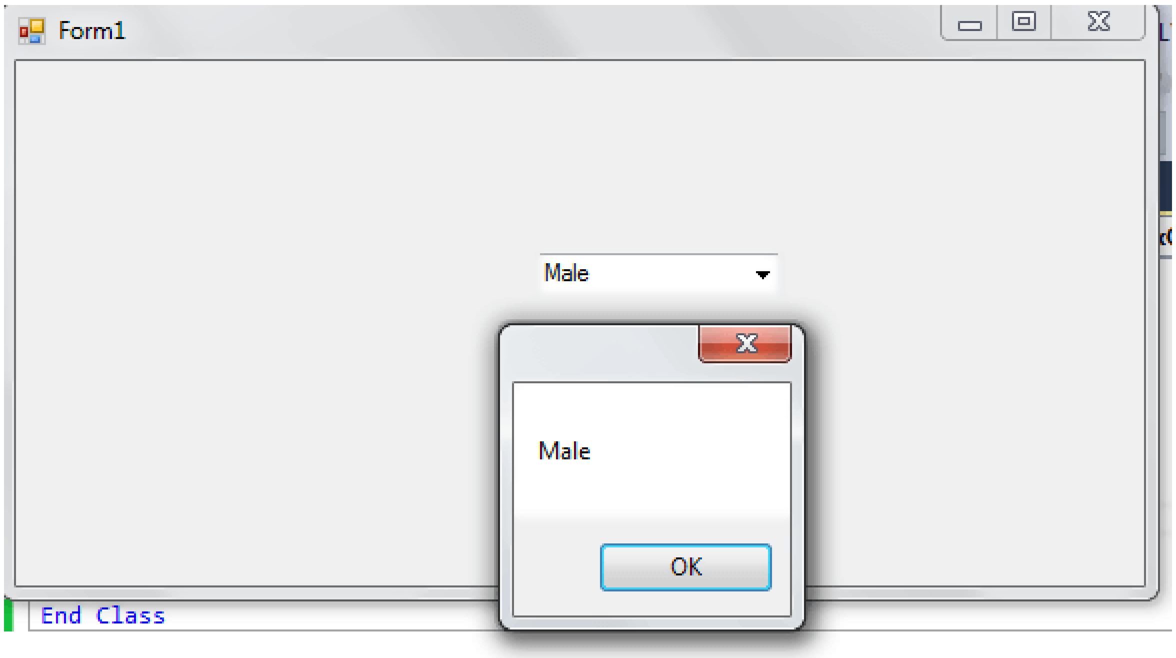
Step 2) Add the following code:

```
Public Class Form1

    Private Sub ComboBox1_SelectedIndexChanged(sd As Object, evnt As EventArgs) Handles
ComboBox1.SelectedIndexChanged

        Dim var_gender As String

        var_gender = ComboBox1.Text

        MessageBox.Show(var_gender)

    End Sub

End Class
```

Step 3) Click the Start button from the toolbar to execute the code. You should get the following form:

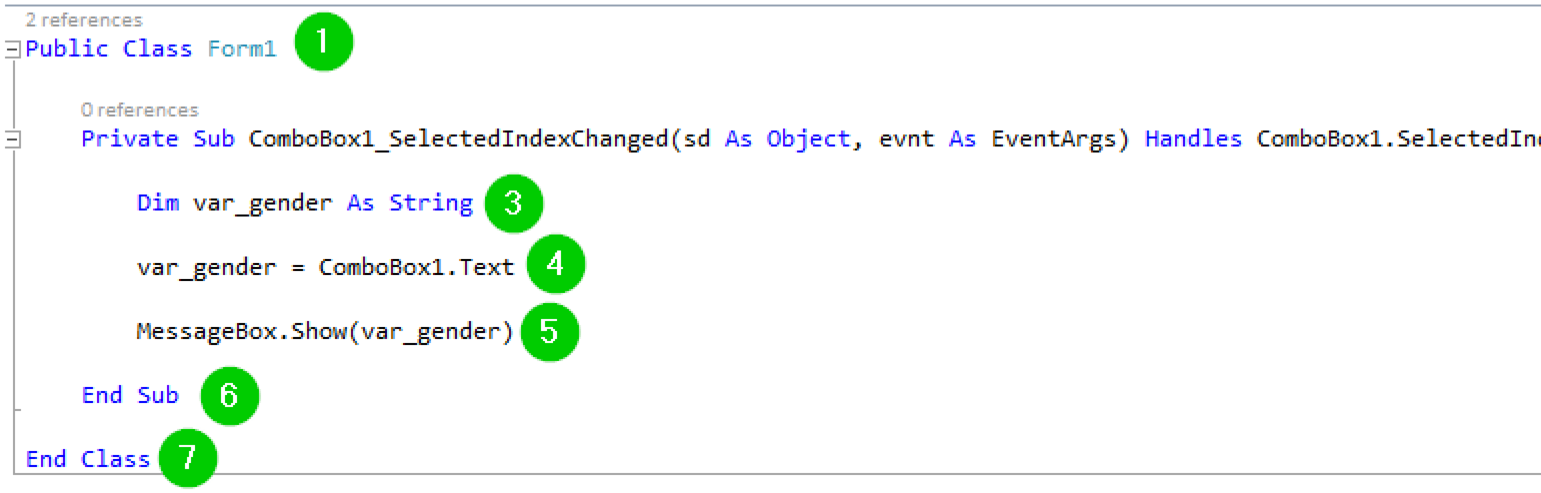


Step 4) Click the dropdown button and choose your gender. In my case. I choose Male, and I get the following:



Male

Here is a screenshot of the code:



```
2 references
Public Class Form1 1

    0 references
    Private Sub ComboBox1_SelectedIndexChanged(sd As Object, evnt As EventArgs) Handles ComboBox1.SelectedInd

        Dim var_gender As String 3

        var_gender = ComboBox1.Text 4

        MessageBox.Show(var_gender) 5

    End Sub 6

End Class 7
```

Explanation of Code:

1. Creating a class named Form1. The class will be publicly accessible since its access modifier has been set to Public.
2. Starting of a sub-procedure named ComboBox1_SelectedIndexChanged. This is generated automatically when you double click the combobox control from the design tab. This sub-procedure will be invoked when you select an item from the combobox. The sd As Object references the object that raised the event while the event As EventArgs has the event data s.
3. Creating a string integer named var_gender.
4. Setting the value of variable var_gender to the item that is selected on the combobox.
5. Printing the value of the variable var_gender on a MesageBox.
6. End of the ComboBox1_SelectedIndexChanged sub-procedure.
7. End of the Form1 class.

Removing Combobox Items

It is possible for you to remove an item from your combobox. There are two ways through which you can accomplish this. You can use either the item index or the name of the item.

When using the item index, you should use the Items.RemoveAt() property as shown below:

```
ComboBox1.Items.RemoveAt(1)
```

In the above example, we are removing the item located at index 1 of the combobox. Note that combobox indexes begin at index 0, meaning that the above command will remove the second item of the combobox.

To remove the item using its name, you should use the Items.Remove() property as shown below:

```
ComboBox1.Items.Remove("Female")
```

The above code should remove the item named Female from the ComboBox1.

Binding DataSource

A ComboBox can be populated from a Dataset. Consider the SQL Query given below:

```
select emp_id, emp_name from employees;
```

You can create a datasource in a program then use the following code to bind it:

```
comboBox1.DataSource = ds.Tables(0)
comboBox1.ValueMember = "emp_id"
comboBox1.DisplayMember = "emp_name"
```

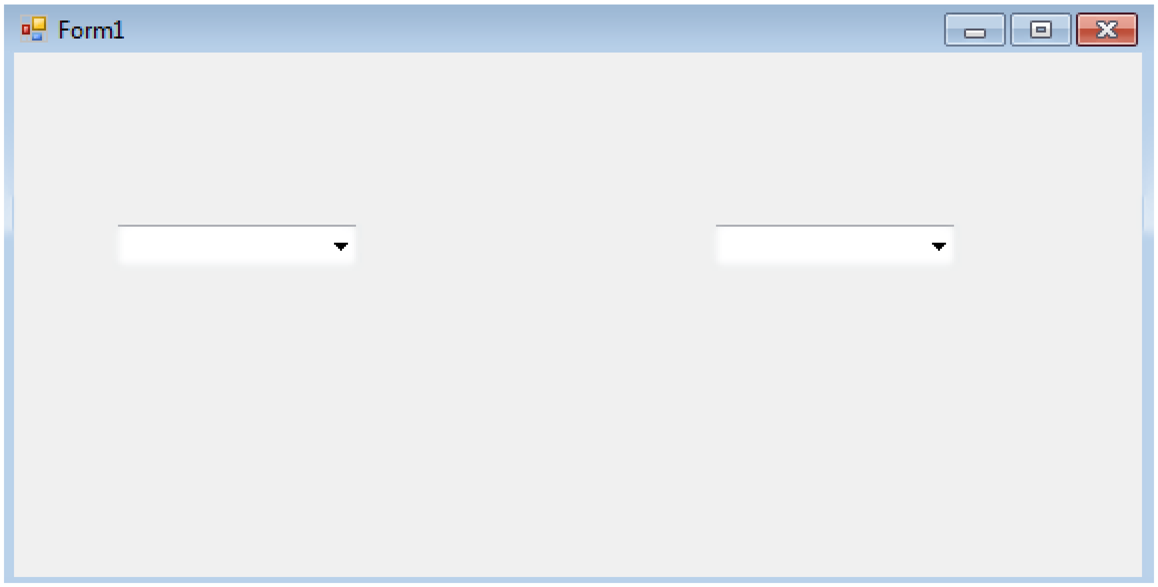
This will provide you with an easy way of populating your combobox control with data without having to type each individual item.

SelectedIndexChanged event

This type of event is invoked when you change the selected item on your combobox. It is the event you should use when you need to implement an action upon a change on the selected item of a combobox. Let us demonstrate this by use of an example:

Step 1) Create a new Window Forms Application.

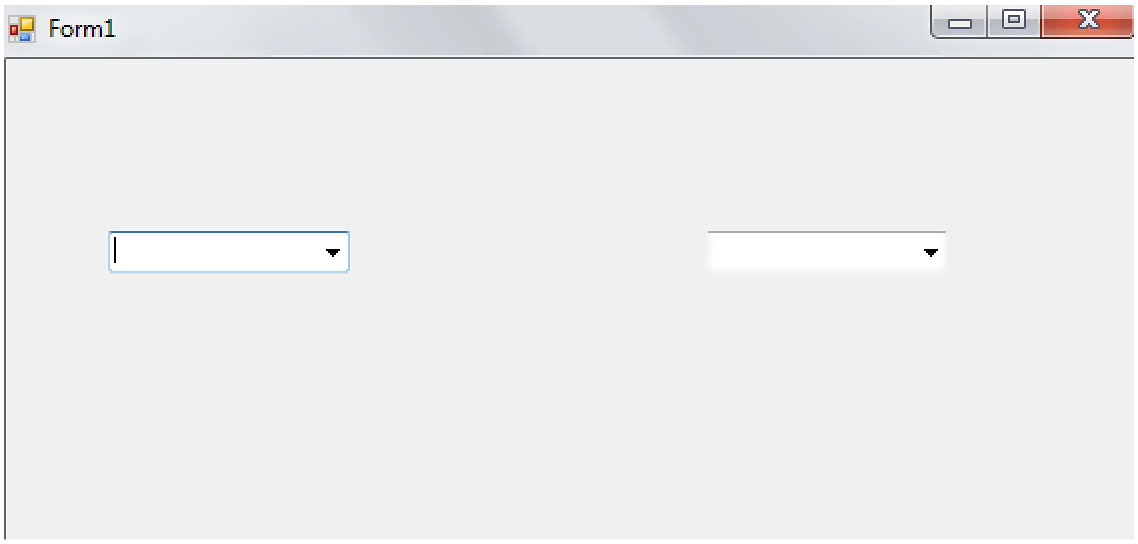
Step 2) After that you need to Drag and drop two combobox controls into the form.



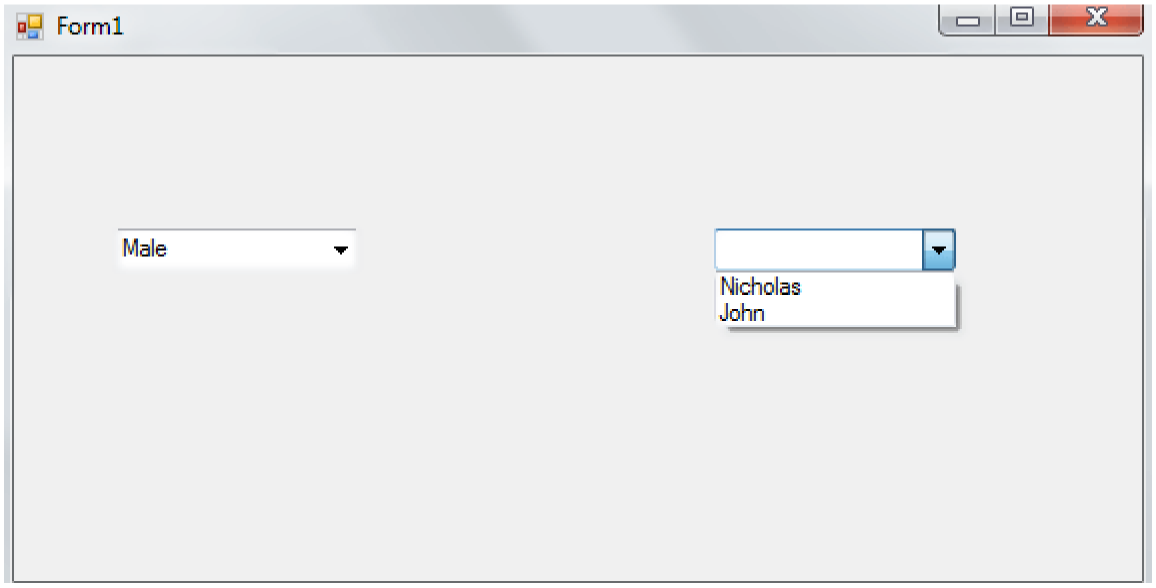
Step 3) Double click inside the form to open the tab for code. Enter the following code:

```
Public Class Form1

    Private Sub Form1_Load(sd As Object, evnt As EventArgs) Handles MyBase.Load

        ComboBox1.Items.Add("Males")

        ComboBox1.Items.Add("Females")

    End Sub

    Private Sub ComboBox1_SelectedIndexChanged(sender As Object, e As EventArgs) Handles
ComboBox1.SelectedIndexChanged

        ComboBox2.Items.Clear()

        If ComboBox1.SelectedItem = "Males" Then

            ComboBox2.Items.Add("Nicholas")

            ComboBox2.Items.Add("John")

        ElseIf ComboBox1.SelectedItem = "Females" Then

            ComboBox2.Items.Add("Alice")

            ComboBox2.Items.Add("Grace")

        End If

    End Sub
End Class
```

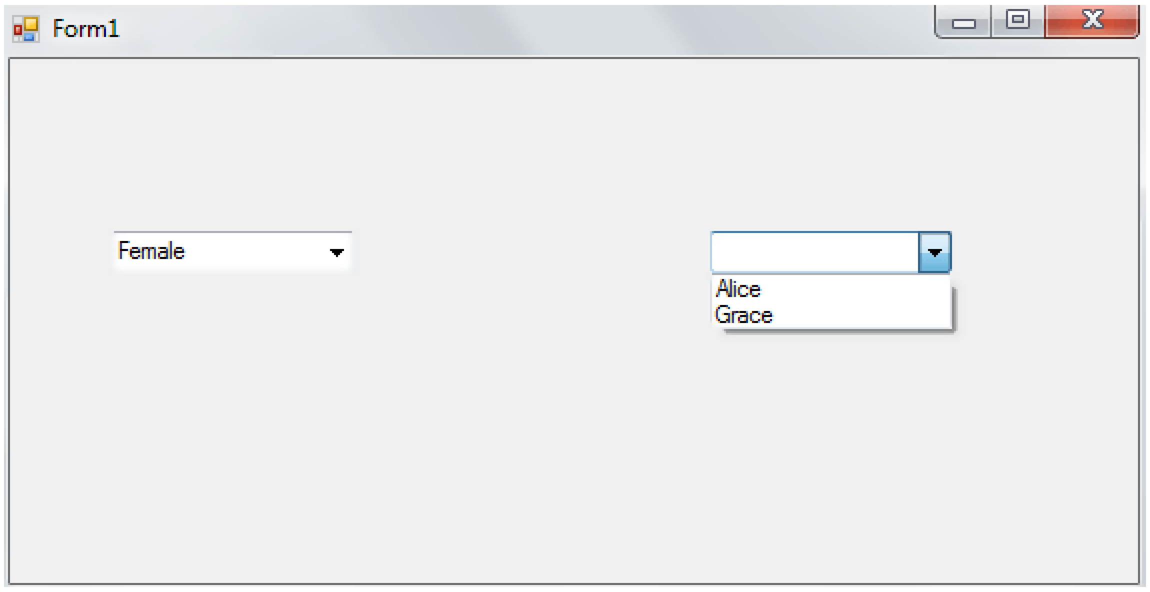
Step 4) Click the Start button from the top bar to run the code. You should get the following output:



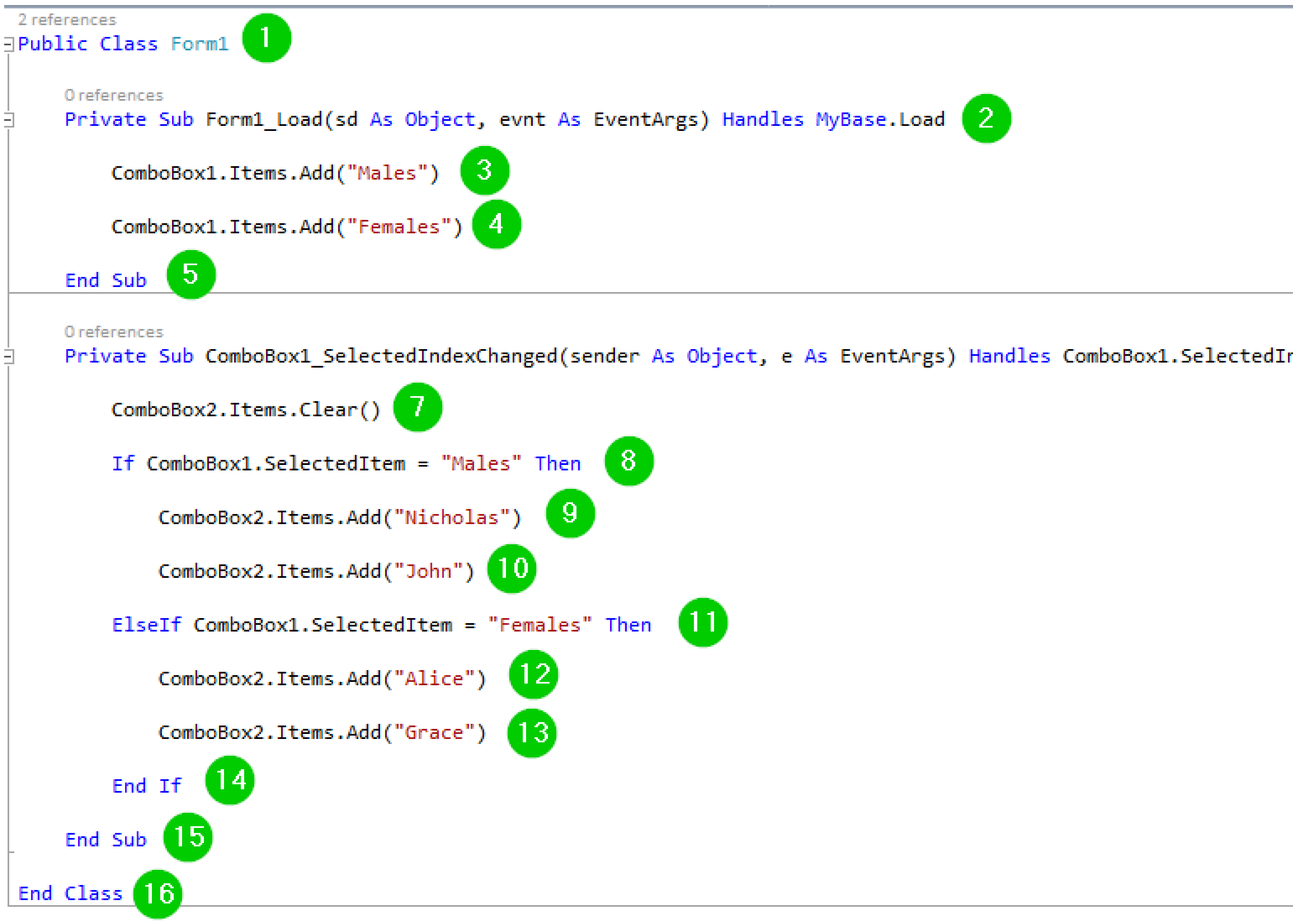
Step 5) Click the dropdown button on the first combobox and choose Male. Move the mouse cursor to the second combobox and click its dropdown button. See the available items:



Step 6) Move to the first combobox and choose Female. Move to the second combobox and see the available items:



Here is a screenshot of the code:



```
2 references
Public Class Form1 1

    0 references
    Private Sub Form1_Load(sd As Object, evnt As EventArgs) Handles MyBase.Load 2

        ComboBox1.Items.Add("Males") 3

        ComboBox1.Items.Add("Females") 4

    End Sub 5

    0 references
    Private Sub ComboBox1_SelectedIndexChanged(sender As Object, e As EventArgs) Handles ComboBox1.SelectedIr

        ComboBox2.Items.Clear() 7

        If ComboBox1.SelectedItem = "Males" Then 8

            ComboBox2.Items.Add("Nicholas") 9

            ComboBox2.Items.Add("John") 10

        ElseIf ComboBox1.SelectedItem = "Females" Then 11

            ComboBox2.Items.Add("Alice") 12

            ComboBox2.Items.Add("Grace") 13

        End If 14

    End Sub 15

End Class 16
```

Explanation of Code:

1. Creating a class named Form1.
2. Start of a sub-procedure named Form1_Load(). This will be triggered once the form is loaded. The sd As Object references the object that raised the event while the system As EventArgs has the event data.
3. Adding the item Males to the ComboBox1.
4. Adding the item Females to the ComboBox1.
5. End of the Form1_Load() sub-procedure.
6. Start of a sub-procedure named ComboBox1_SelectedIndexChanged(). This will be invoked when an item is selected on the first combobox. The sender As Object references the object that raised the event while the e As EventArgs has the event data.
7. Make ComboBox2 empty, clear all items from it.
8. Creating a condition. Checking for whether the selected item on ComboBox1 is Males.
9. Add the item Nicholas to the ComboBox2 when the above condition is true, that is, item selected on ComboBox1 is Male.
10. Add the item John to the ComboBox2 when the above condition is true, that is, item selected on ComboBox1 is Males.
11. Creating a condition. Checking for whether the selected item on ComboBox1 is Females.
12. Add the item Alice to the ComboBox2 when the above condition is true, that is, item selected on ComboBox1 is Females.
13. Add the item Grace to the ComboBox2 when the above condition is true, that is, item selected on ComboBox1 is Females.
14. End of the If block.
15. End of the ComboBox1_SelectedIndexChanged() sub-procedure.
16. End of the class Form1.

Summary

- A ComboBox is created by dragging it from the toolbox and dropping it into the form.
- It provides us with a way of presenting numerous options to the user.
- We can set the default item to be selected on the ComboBox when the form is loaded.
- The SelectedIndexChanged event helps us specify the action to take when a particular item is selected on the combobox.

TEXTBOX CONTROL

The TextBox Control allows you to enter text on your form during runtime. The default setting is that it will accept only one line of text, but you can modify it to accept multiple lines. You can even include scroll bars into your TextBox Control.

In this tutorial, you will learn

- What is TextBox Control?
- TextBox Properties
- Textbox Events
- How to Create a TextBox
- Password character
- Newline in TextBox
- Retrieving Integer Values
- ReadOnly TextBox
- max length

TextBox Properties

The following are the most common properties of the Visual Basic TextBox control:

- **TextAlign**- for setting text alignment
- **ScrollBars**- for adding scrollbars, both vertical and horizontal
- **Multiline**- to set the TextBox Control to allow multiple lines
- **MaxLength**- for specifying the maximum character number the TextBox Control will accept
- **Index**- for specifying the index of control array
- **Enabled**- for enabling the textbox control
- **Readonly**- if set to true, you will be able to use the TextBox Control, if set to false, you won't be able to use the TextBox Control.
- **SelectionStart**- for setting or getting the starting point for the TextBox Control.
- **SelectionLength**- for setting or getting the number of characters that have been selected in the TextBox Control.
- **SelectedText**- returns the TextBox Control that is currently selected.

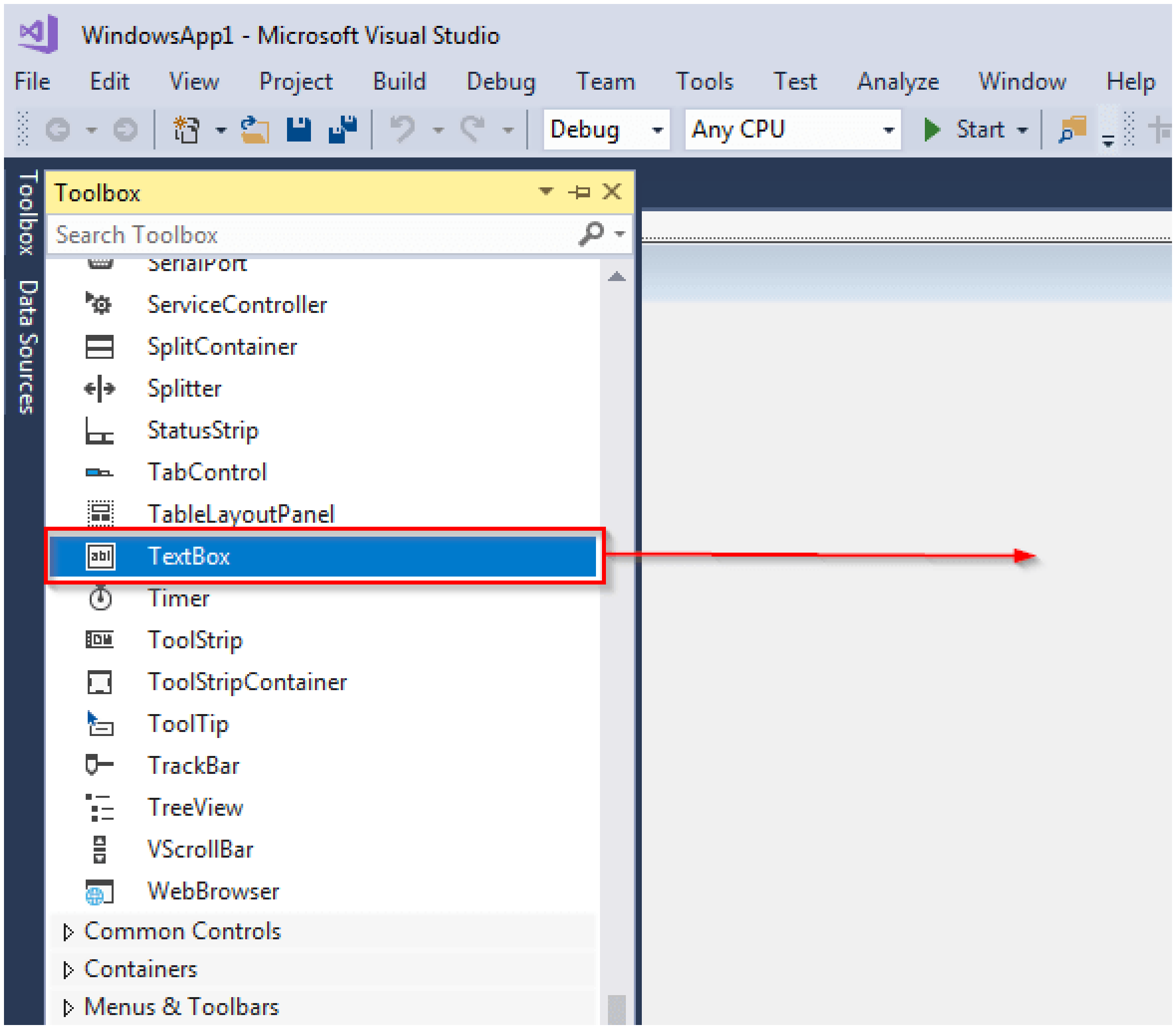
Textbox Events

The purpose of events is to make the TextBox Control respond to user actions such as a click, a double click or change in text alignment. Here are the common events for the TextBox Control:

- **AutoSizeChanged**- Triggered by a change in the AutoSize property.
- **ReadOnlyChanged**- Triggered by a change of the ReadOnly property value.
- **Click**- Triggered by a click on the TextBox Control.

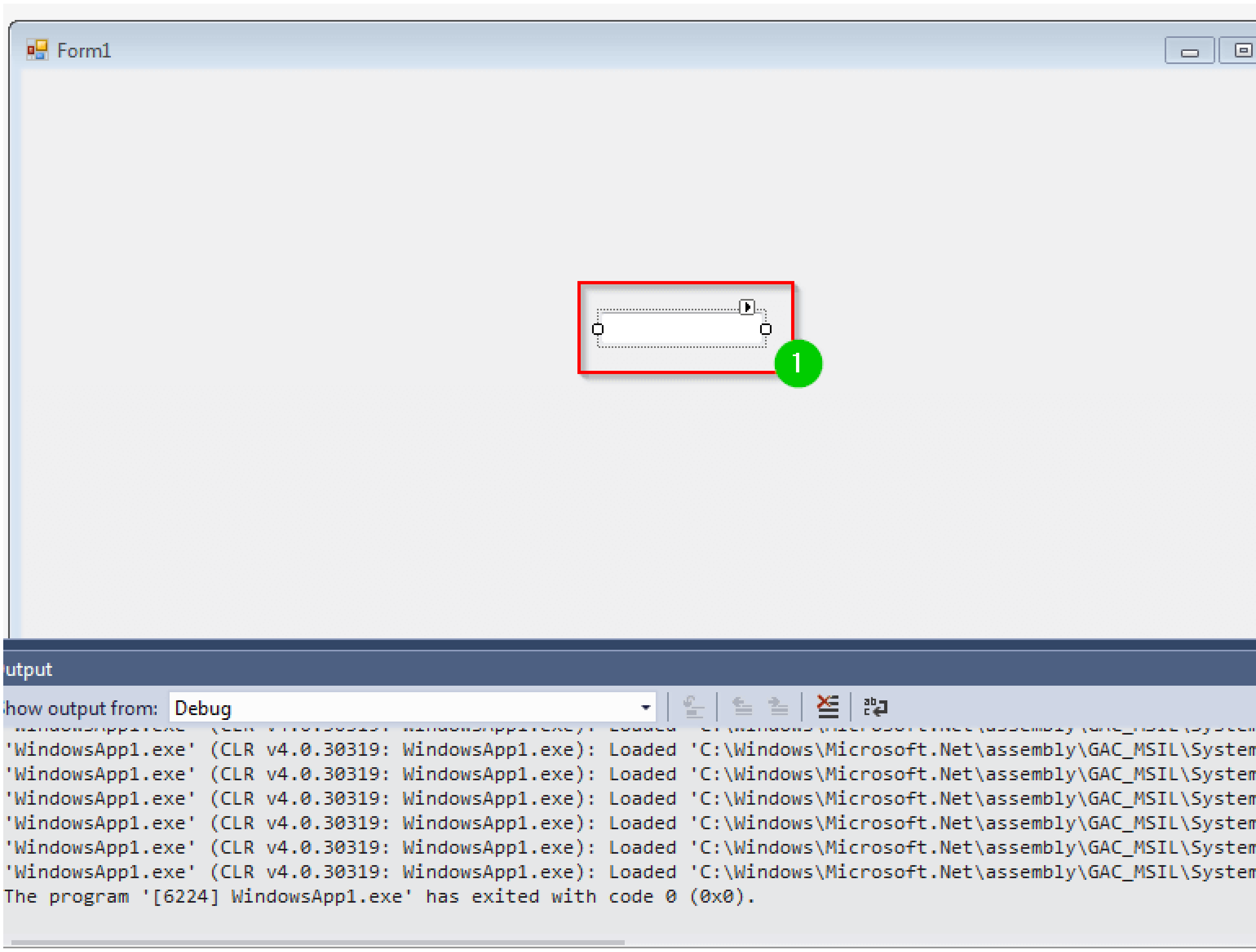
How to Create a TextBox

Step 1) To create a TextBox, drag the TextBox control from the toolbox into the WindowForm:



Step 2)

1. Click the TextBox Control that you have added to the form.
2. Move to the Properties section located on the bottom left of the screen. Change the name of the text box from TextBox1 to HelloTextBox:

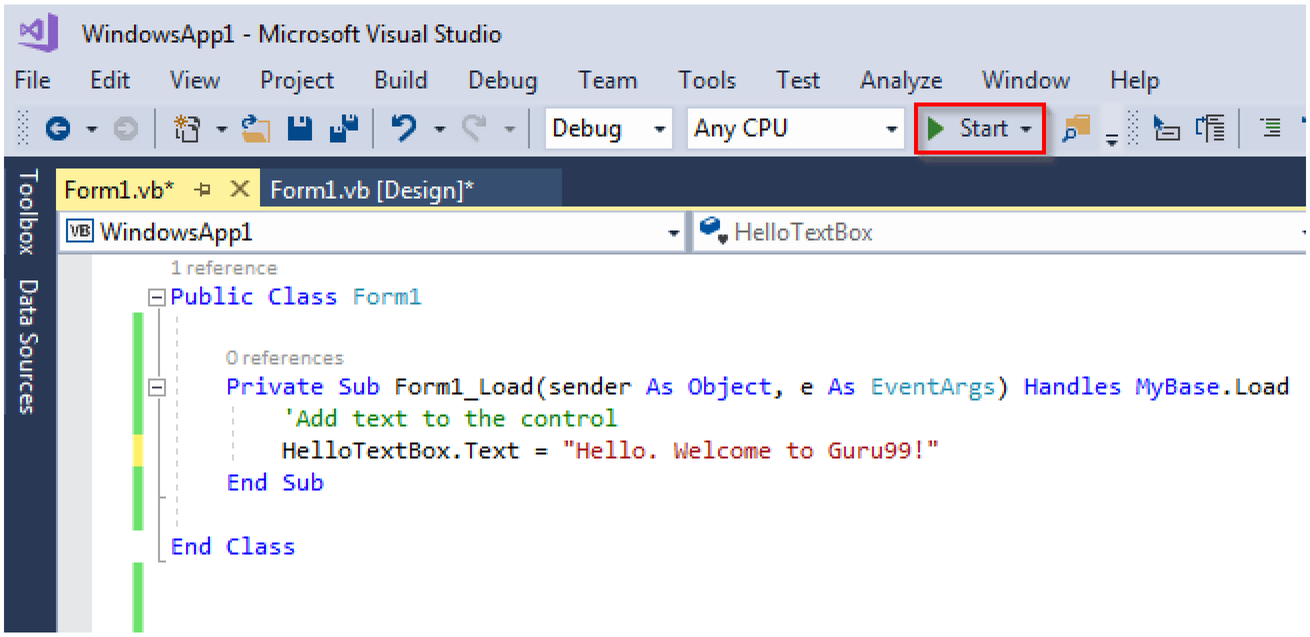


```
'WindowsApp1.exe' (CLR v4.0.30319: WindowsApp1.exe): Loaded 'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System
'WindowsApp1.exe' (CLR v4.0.30319: WindowsApp1.exe): Loaded 'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System
'WindowsApp1.exe' (CLR v4.0.30319: WindowsApp1.exe): Loaded 'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System
'WindowsApp1.exe' (CLR v4.0.30319: WindowsApp1.exe): Loaded 'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System
'WindowsApp1.exe' (CLR v4.0.30319: WindowsApp1.exe): Loaded 'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System
'WindowsApp1.exe' (CLR v4.0.30319: WindowsApp1.exe): Loaded 'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System
The program '[6224] WindowsApp1.exe' has exited with code 0 (0x0).
```

Step 3) Add the following code to add text to the control:

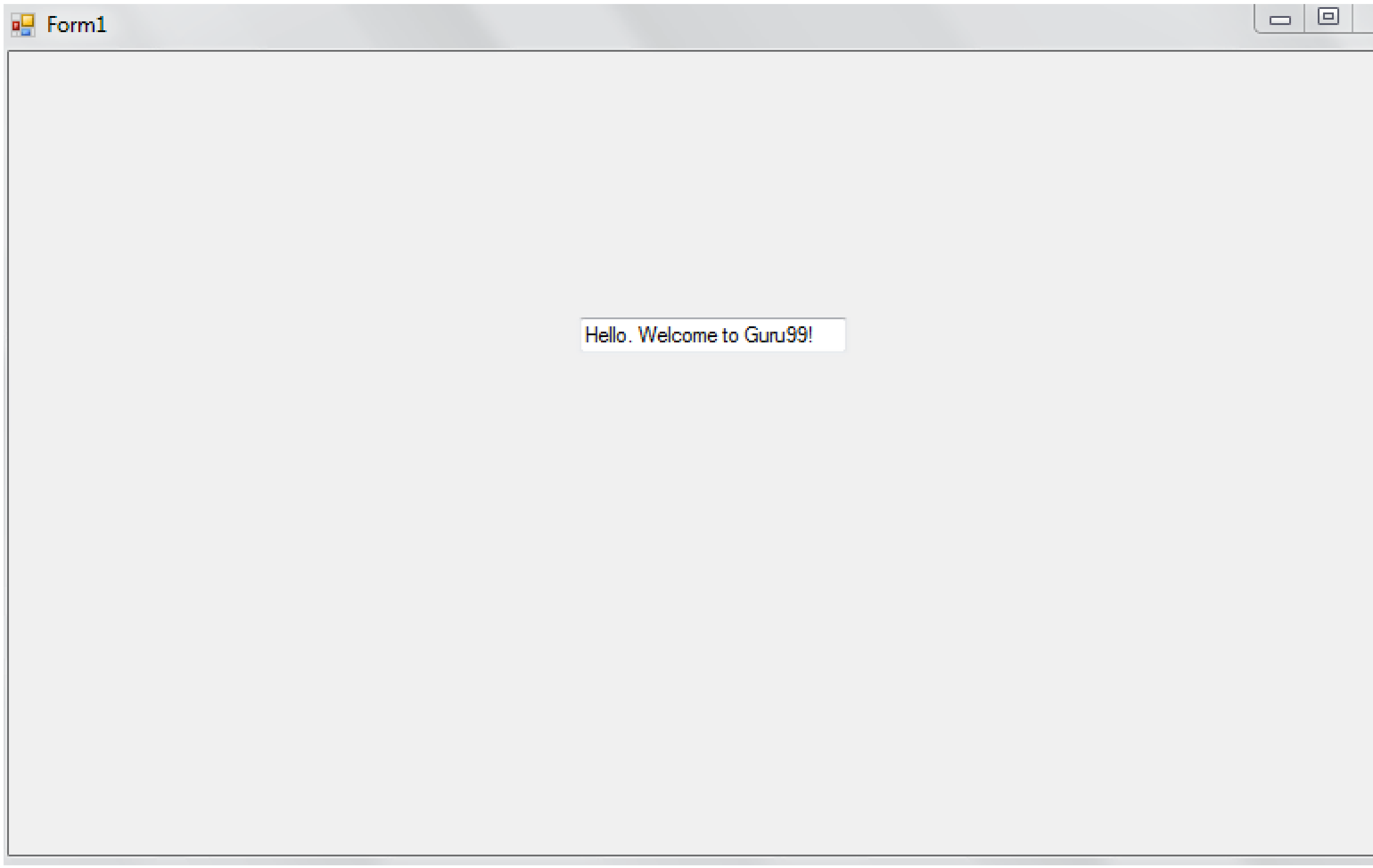
```
Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        'Add text to the control
         HelloTextBox.Text = "Hello. Welcome to Guru99!"
  End Sub
```

Step 4) You can now run the code by clicking the Start button located at the top bar:


WindowsApp1 - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Debug
Any CPU
Start
Toolbox
Data Sources
Form1.vb*
Form1.vb [Design]*
WindowsApp1
HelloTextBox

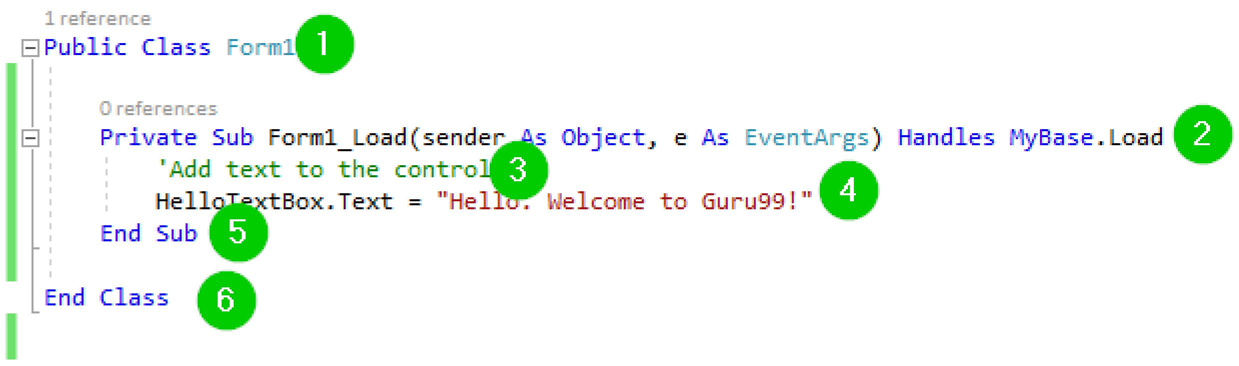
```
1 reference
Public Class Form1

    0 references
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        'Add text to the control
        HelloTextBox.Text = "Hello. Welcome to Guru99!"
    End Sub

End Class
```

Step 5) You should get the following form:



Hello. Welcome to Guru99!

Here is a screenshot of the complete code for the above:



```
1 reference
Public Class Form1

    0 references
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        'Add text to the control
        HelloTextBox.Text = "Hello. Welcome to Guru99!"
    End Sub

End Class
```

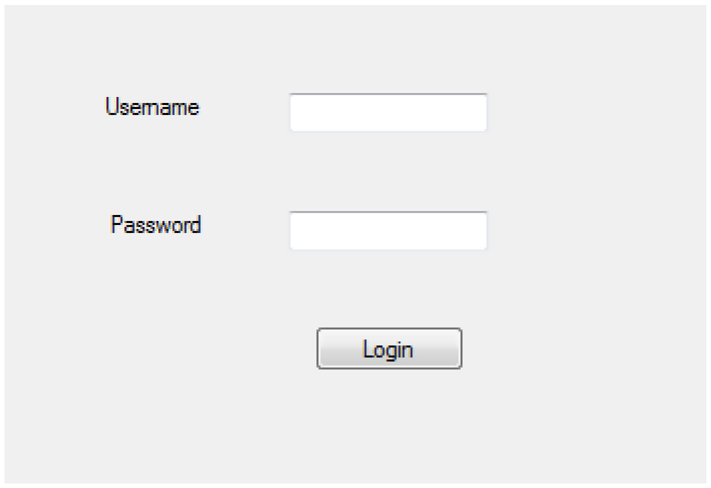
Explanation of code:

1. Creating a public class named Form1
2. Creating a sub procedure named Form1_Load. It will be called when the form is loaded.
3. A comment. The VB.net compiler will skip this.
4. End the subprocedure
5. End the class.

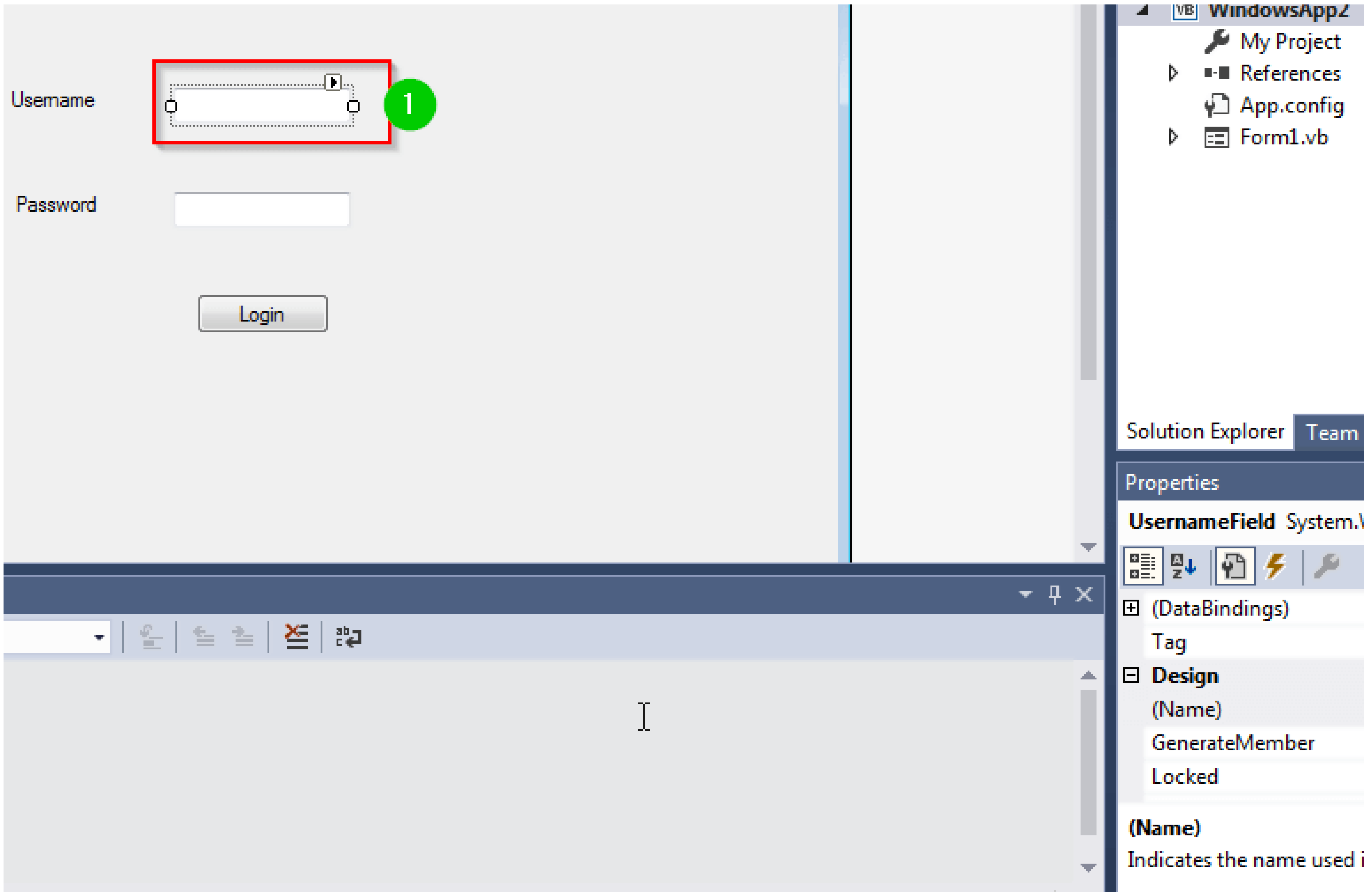
Password character

Sometimes, you want a text box to be used for typing a password. This means that whatever is typed into that text box to remain confidential. This is possible with VB.net. It can be done using the **PasswordChar** property which allows us to use any character that you want. Let us demonstrate this using an example:

Begin by creating a new project. Drag two TextBox Controls, two Labels, and one Button into the form. Change the texts on the two labels and the button to the following:



Click the text box next to Username label, move to the Properties section located at the bottom left of the window. Give it the name UsernameField.

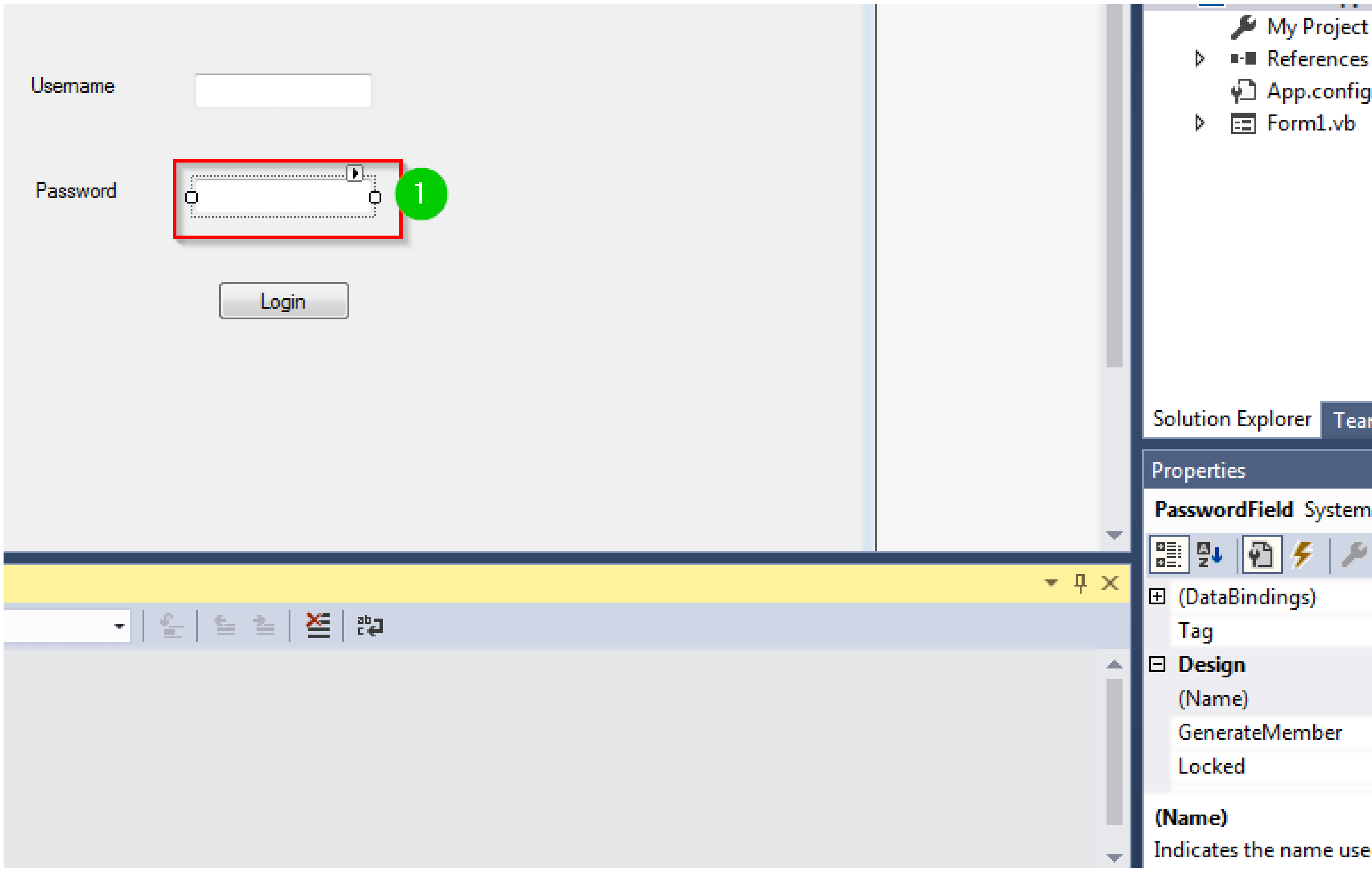


UsernameField System.\

(DataBindings)
Tag
Design
(Name)
GenerateMember
Locked

(Name)
Indicates the name used i

Do the same for the TextBox Control next to Password label, giving it the name PasswordField.

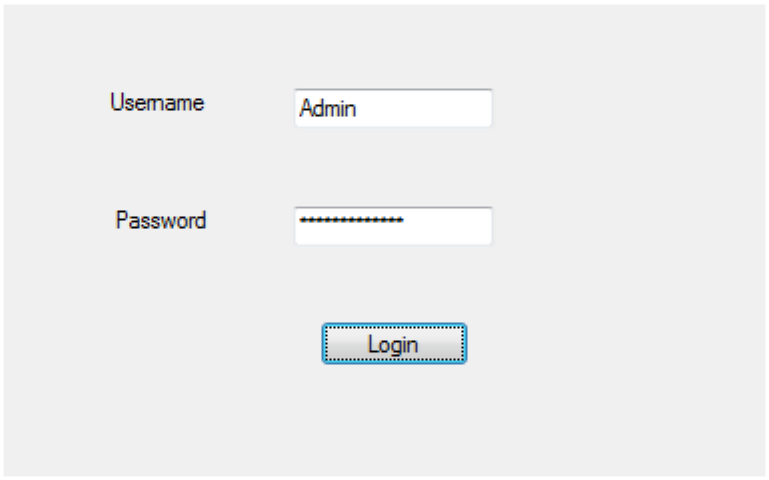


You should now make the PasswordField TextBox Control show * for each character typed in it. Add the following code:

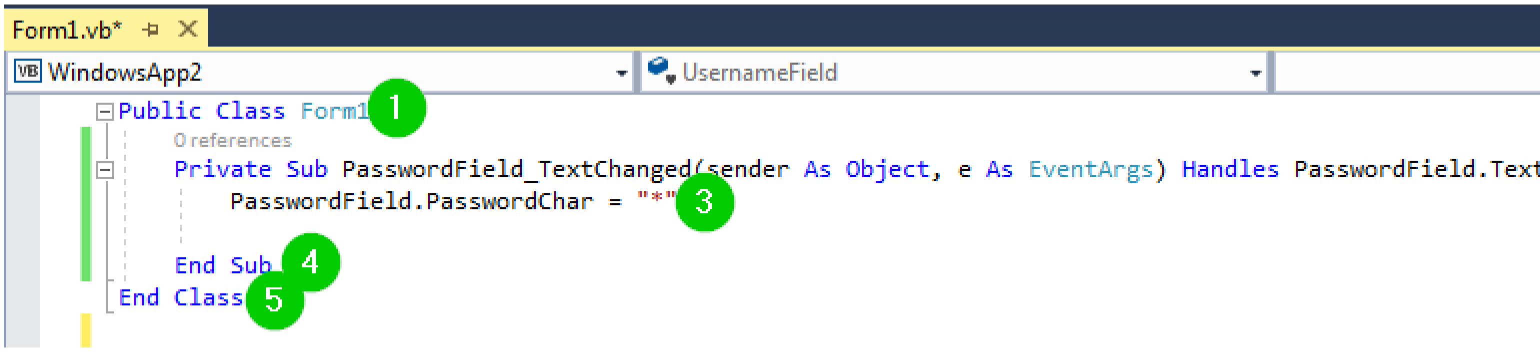
```
Private Sub PasswordField_TextChanged(sender As Object, e As EventArgs) Handles
PasswordField.TextChanged
    PasswordField.PasswordChar = "*"
End Sub
```

Now, run the code by clicking the Start button. A form will popup.

Type the username and the password and observe what happens. You should see the following:



The username is shown, but the password has been hidden. Here is the code for the above:



```
Public Class Form1
    0 references
    Private Sub PasswordField_TextChanged(sender As Object, e As EventArgs) Handles PasswordField.Text
        PasswordField.PasswordChar = "*"

    End Sub
End Class
```

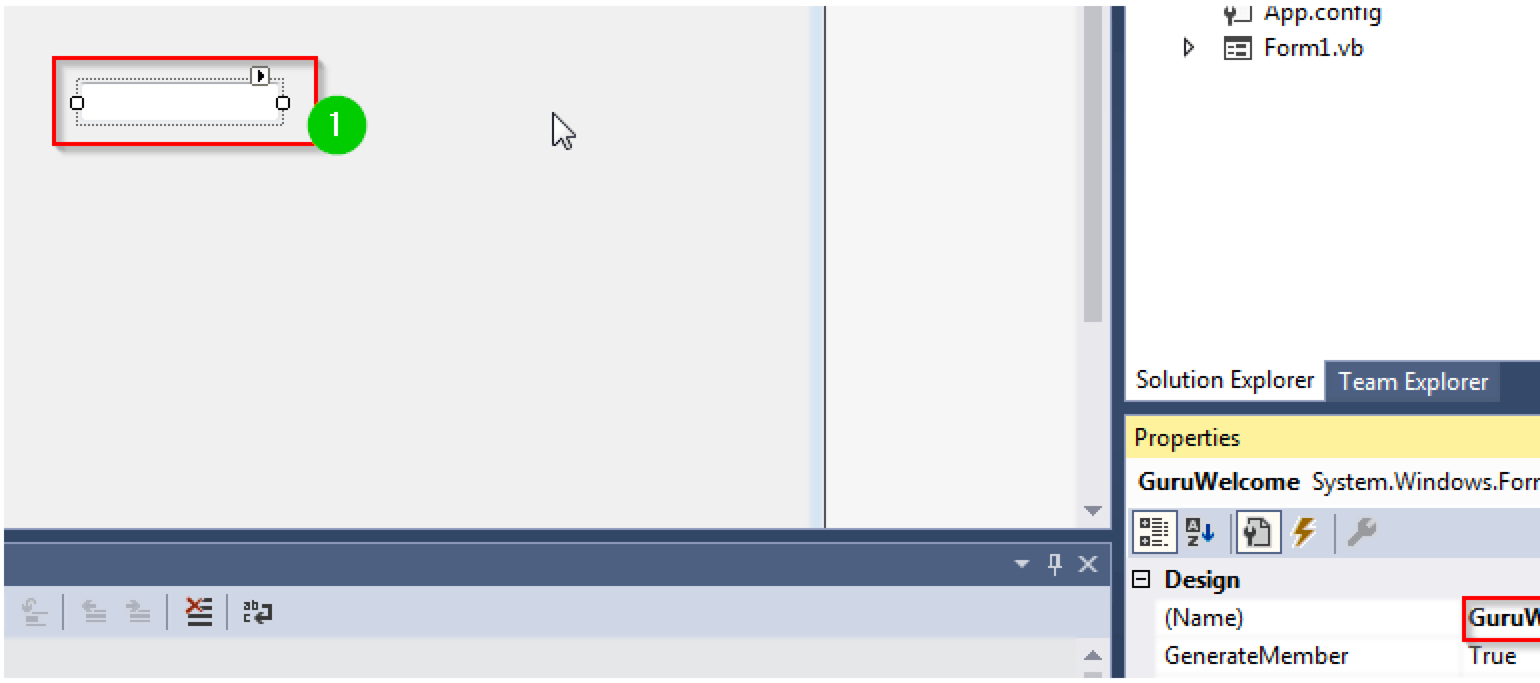
Explanation of code:

1. Creating a class named Form1.
2. Creating a sub-procedure named PasswordField_textchanged(). It will be called when the form is loaded.
3. Using the PasswordChar property to show * as a user types the password.
4. Ending the sub-procedure.
5. Ending the class.

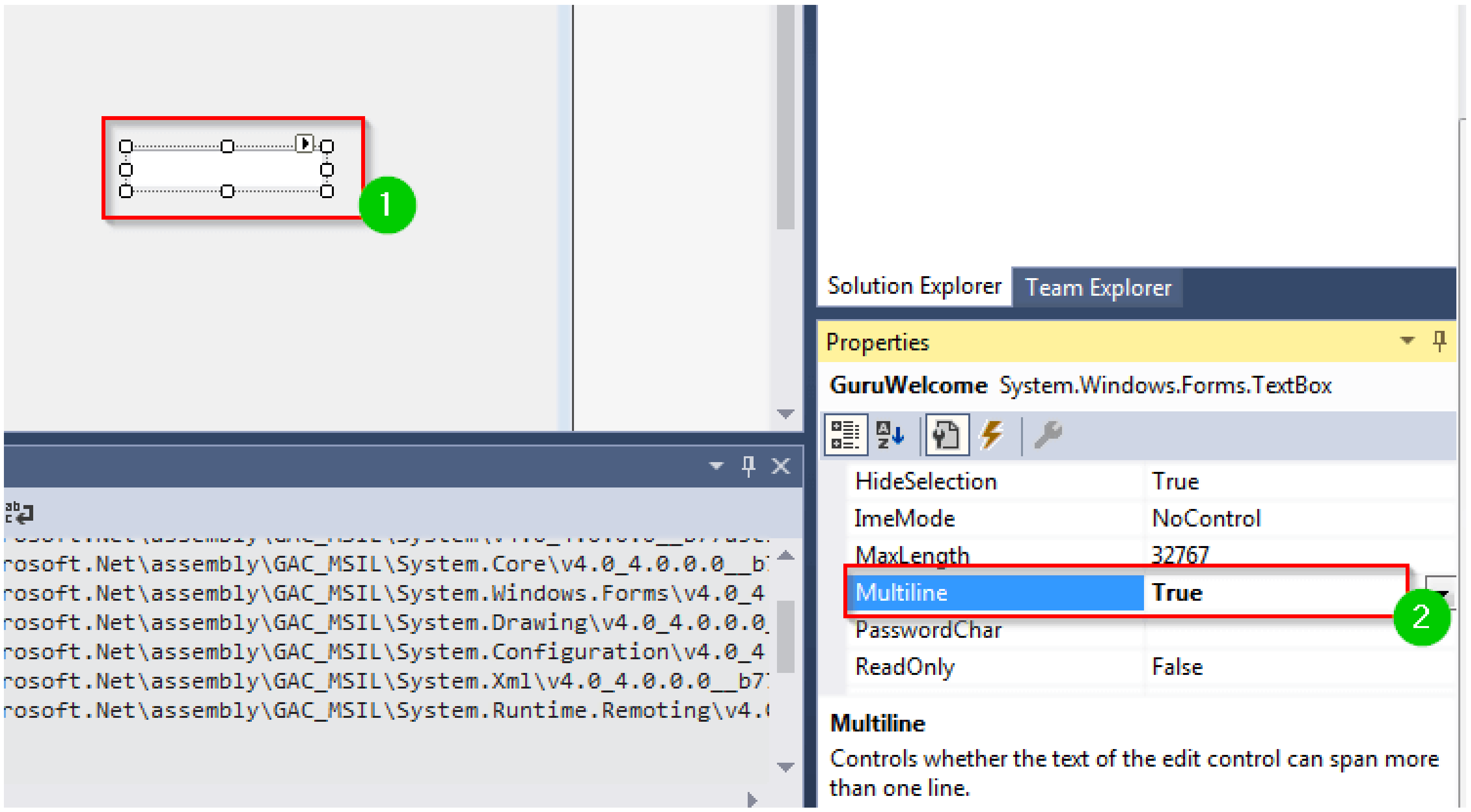
Newline in TextBox

By default, you can only create one line of text in a text box. There are two ways through which we can achieve this. Let us discuss them.

Drag and drop a TextBox Control to your form. Give the control the name GuruWelcome.



Click the text box control again and move the mouse cursor to the Properties section. Change the value of Multiline property to True.



```
rosoft.Net\assembly\GAC_MSIL\System.Core\v4.0_4.0.0.0__b
rosoft.Net\assembly\GAC_MSIL\System.Windows.Forms\v4.0_4
rosoft.Net\assembly\GAC_MSIL\System.Drawing\v4.0_4.0.0.0
rosoft.Net\assembly\GAC_MSIL\System.Configuration\v4.0_4
rosoft.Net\assembly\GAC_MSIL\System.Xml\v4.0_4.0.0.0__b7
rosoft.Net\assembly\GAC_MSIL\System.Runtime.Remoting\v4.
```

HideSelection	True
ImeMode	NoControl
MaxLength	32767
Multiline	True
PasswordChar	
ReadOnly	False

Multiline
Controls whether the text of the edit control can span more than one line.

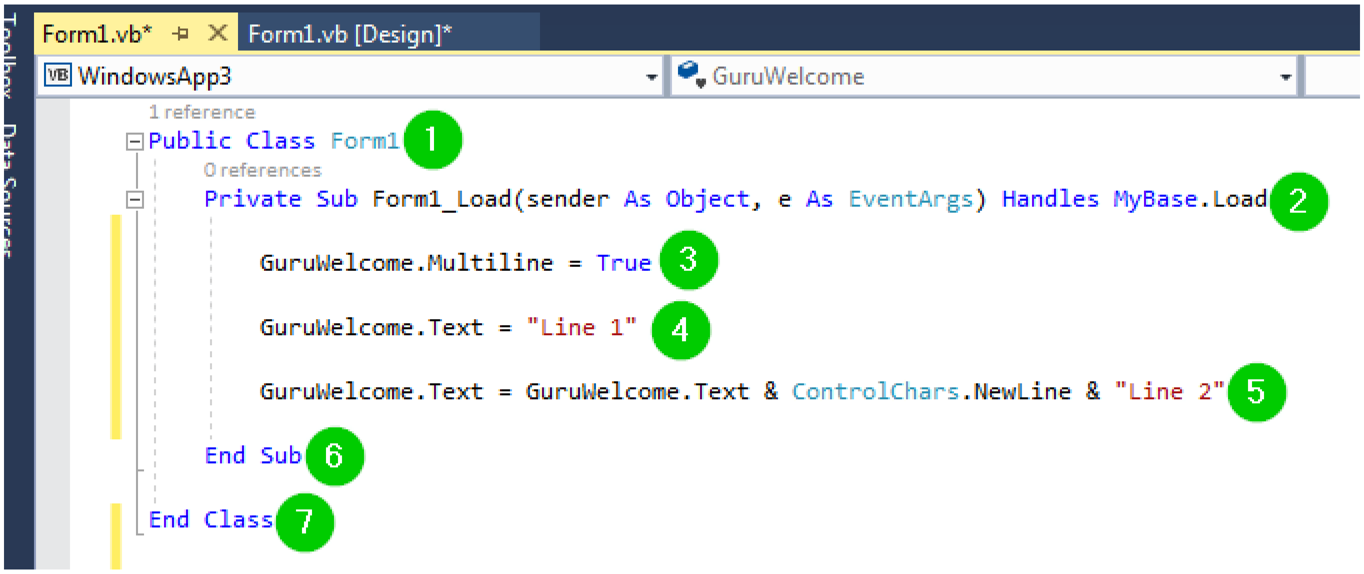
Alternative, you can set the Multiline property to true in your code as follows:

GuruWelcome.Multiline = True

Add the following code:

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

     GuruWelcome.Multiline = True

     GuruWelcome.Text = "Line 1"

     GuruWelcome.Text = GuruWelcome.Text & ControlChars.NewLine & "Line 2"

  End Sub
```

Upon execution, the two lines of text will be separated.



```
Public Class Form1 (1)
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load (2)

        GuruWelcome.Multiline = True (3)

        GuruWelcome.Text = "Line 1" (4)

        GuruWelcome.Text = GuruWelcome.Text & ControlChars.NewLine & "Line 2" (5)

    End Sub (6)

End Class (7)
```

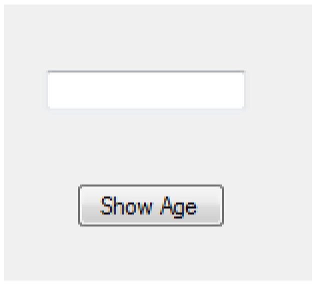
Explanation of Code:

1. Creating a class named Form1
2. Creating a sub-routine named Form1_Load()
3. Setting the Multiline property to True. The textbox will be able to take more than one lines.
4. Adding the first line of text to the text box.
5. Adding the second line of text to the text box. The Controlchars.NewLine property helps us to split the two lines.
6. Ending the sub-routine.
7. Ending the class.
8.

Retrieving Integer Values

VB.net treats everything as a string. This means that you read an integer from the text box as a string, then you convert it into an integer. This is normally done using the **Integer.Parse()** method.

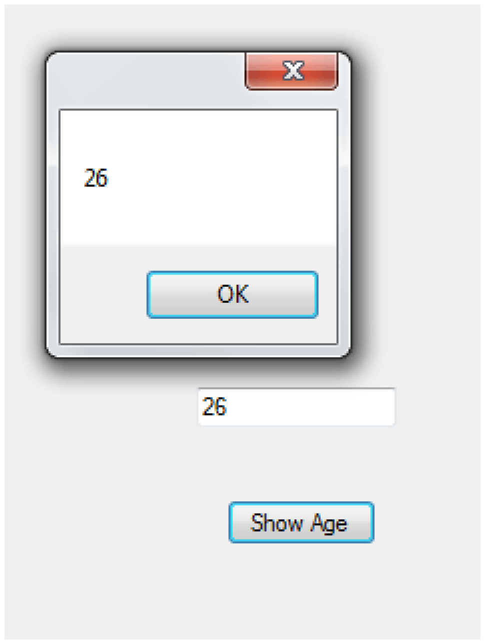
To demonstrate this, create a new text box control plus a button. Give the text box the name age. Give the button the name Button1. You should have the following interface:



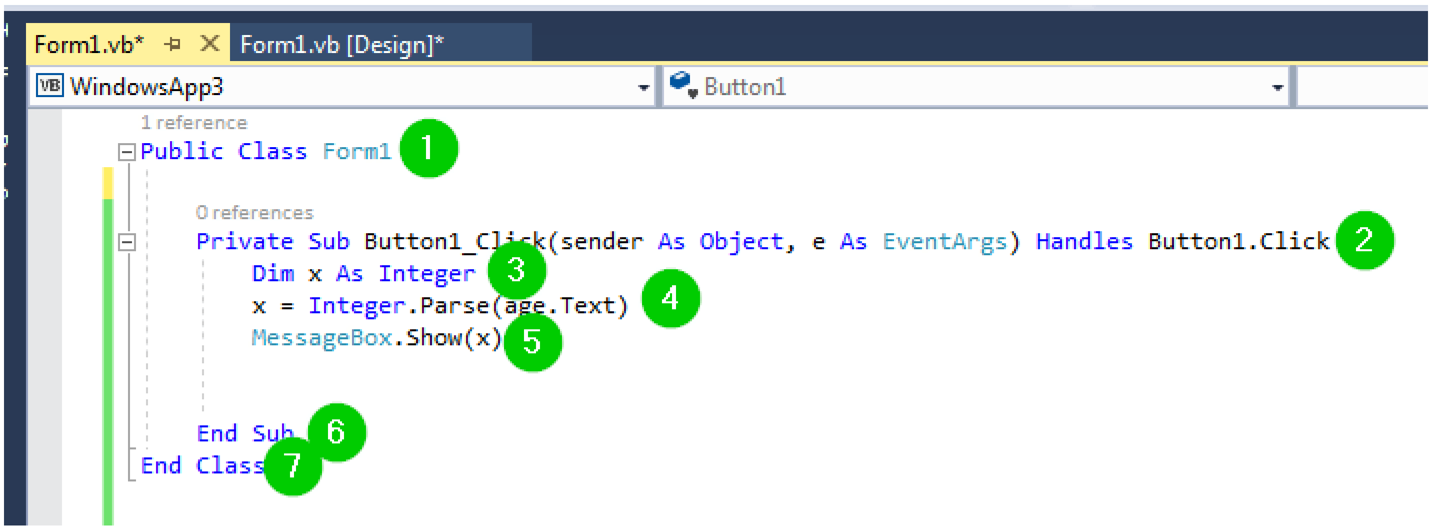
Add the following code:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim x As Integer
        x = Integer.Parse(age.Text)
        MessageBox.Show(x)
```

Run the code, and enter your age into the text box. Click the Show Age button. You should see the following:


26

The value you enter is returned in a MessageBox.



```
Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim x As Integer
        x = Integer.Parse(age.Text)
        MessageBox.Show(x)


    End Sub
End Class
```

Explanation of Code:

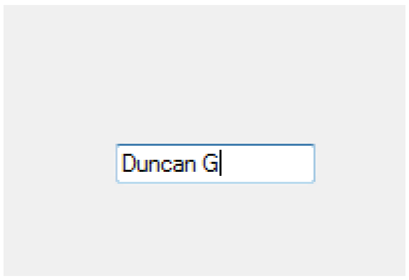
1. Creating a class named Form1.
2. Creating a sub-procedure named Button1_Click. It will be called when the button is clicked.
3. Defining an integer variable named x.
4. Converting the value read from the textbox named age into an integer.
5. Displaying the value converted in the above step in a MessageBox.
6. Ending the sub-procedure.
7. Ending the class.

max length

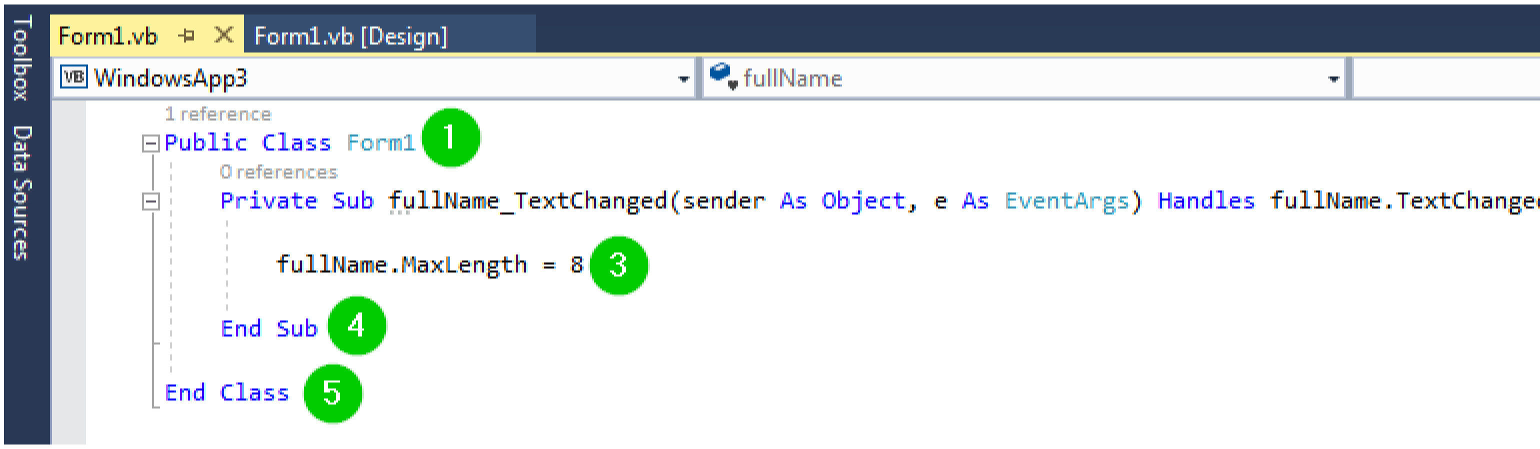
The MaxLength property can help you set the maximum number of words or characters that the textbox will allow. To demonstrate this, create a TextBox control and give it the name fullName. Add the following code:

```
Private Sub fullName_TextChanged(sender As Object, e As EventArgs) Handles
fullName.TextChanged

      fullName.MaxLength = 8

   End Sub
```

Run the code and try to type your full name. You will not be able to type more than 8 characters, with whitespace included:



The code:



```
Public Class Form1
    Private Sub fullName_TextChanged(sender As Object, e As EventArgs) Handles fullName.TextChange

        fullName.MaxLength = 8

    End Sub

End Class
```

Code Explanation:

1. Creating a class named Form1.
2. Creating a sub-procedure named fullName_TextChanged.
3. Making the fullName textbox accept a maximum of only 8 characters.
4. Ending the sub-procedure.
5. Ending the class.

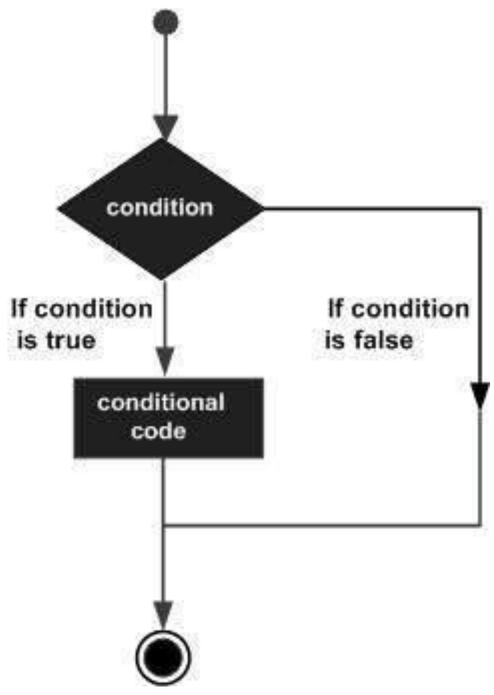
Summary:

- The TextBox Control allows you to enter text into your form during runtime. It is good for getting input from users.
- The default setting is that the TextBox Control will only accept one line of text. However, it is possible for you to change this.
- You can hide what the user types into the TextBox, especially when you need to capture passwords.
- You can also set the maximum number of characters that you need to be entered into the TextBox.
- You can make your TextBox un-editable, meaning that the users won't be able to change the text displayed on it.

DECISION MAKING STRUCTURES

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).

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.

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.

STRING

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

Creating a String Object

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

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

The following example demonstrates this –

```
Module strings
SubMain()
Dim fname, lname, fullname, greetings AsString
      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()AsString={"Hello","From","Tutorials","Point"}
Dim message AsString=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 −

Sr.No	**Property Name & Description**
1	**Chars** Gets the *Char* object at a specified position in the current *String* object.
2	**Length** Gets the number of characters in the current String object.

Methods of the String Class

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

Sr.No	**Method Name & Description**
1	**Public Shared Function Compare (strA As String, strB As String) As Integer**

	Compares two specified string objects and returns an integer that indicates their relative position in the sort order.
2	**Public Shared Function Compare (strA As String, strB As String, ignoreCase As Boolean) As Integer** Compares two specified string objects and returns an integer that indicates their relative position in the sort order. However, it ignores case if the Boolean parameter is true.
3	**Public Shared Function Concat (str0 As String, str1 As String) As String** Concatenates two string objects.
4	**Public Shared Function Concat (str0 As String, str1 As String, str2 As String) As String** Concatenates three string objects.
5	**Public Shared Function Concat (str0 As String, str1 As String, str2 As String, str3 As String) As String** Concatenates four string objects.
6	**Public Function Contains (value As String) As Boolean** Returns a value indicating whether the specified string object occurs within this string.
7	**Public Shared Function Copy (str As String) As String** Creates a new String object with the same value as the specified string.
8	**pPublic Sub CopyTo (sourceIndex As Integer, destination As Char(), destinationIndex As Integer, count As Integer)** Copies a specified number of characters from a specified position of the string object to a specified position in an array of Unicode characters.
9	**Public Function EndsWith (value As String) As Boolean** Determines whether the end of the string object matches the specified string.
10	**Public Function Equals (value As String) As Boolean** Determines whether the current string object and the specified string object have the same value.

11	**Public Shared Function Equals (a As String, b As String) As Boolean** Determines whether two specified string objects have the same value.
12	**Public Shared Function Format (format As String, arg0 As Object) As String** Replaces one or more format items in a specified string with the string representation of a specified object.
13	**Public Function IndexOf (value As Char) As Integer** Returns the zero-based index of the first occurrence of the specified Unicode character in the current string.
14	**Public Function IndexOf (value As String) As Integer** Returns the zero-based index of the first occurrence of the specified string in this instance.
15	**Public Function IndexOf (value As Char, startIndex As Integer) As Integer** Returns the zero-based index of the first occurrence of the specified Unicode character in this string, starting search at the specified character position.
16	**Public Function IndexOf (value As String, startIndex As Integer) As Integer** Returns the zero-based index of the first occurrence of the specified string in this instance, starting search at the specified character position.
17	**Public Function IndexOfAny (anyOf As Char()) As Integer** Returns the zero-based index of the first occurrence in this instance of any character in a specified array of Unicode characters.
18	**Public Function IndexOfAny (anyOf As Char(), startIndex As Integer) As Integer** Returns the zero-based index of the first occurrence in this instance of any character in a specified array of Unicode characters, starting search at the specified character position.
19	**Public Function Insert (startIndex As Integer, value As String) As String** Returns a new string in which a specified string is inserted at a specified index position in the current string object.
20	**Public Shared Function IsNullOrEmpty (value As String) As Boolean** Indicates whether the specified string is null or an Empty string.

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

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

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

Examples

The following example demonstrates some of the methods mentioned above −

Comparing Strings

```
Module strings
SubMain()
Dim str1, str2 AsString
      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.")
EndIf
Console.ReadLine()
EndSub
EndModule
```

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
SubMain()
Dim str1 AsString
      str1 ="This is test"

If(str1.Contains("test"))Then
Console.WriteLine("The sequence 'test' was found.")
EndIf
Console.ReadLine()
EndSub
EndModule
```

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

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

Getting a Substring:

```
Module strings
SubMain()
Dim str AsString
      str ="Last night I dreamt of San Pedro"
Console.WriteLine(str)

Dim substr AsString= str.Substring(23)
Console.WriteLine(substr)
Console.ReadLine()
EndSub
EndModule
```

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
SubMain()
Dim strarray AsString()={
"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 AsString=String.Join(vbCrLf, strarray)
Console.WriteLine(str)
Console.ReadLine()
EndSub
EndModule
```

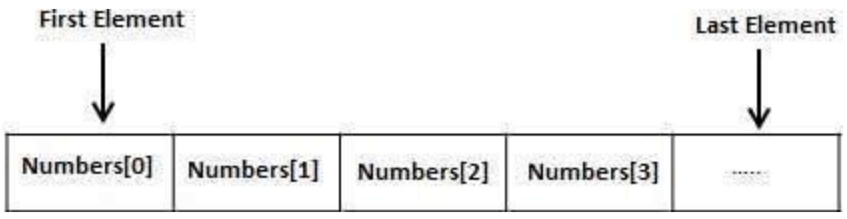
When the above code is compiled and executed, it produces the following result −

```
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
```

ARRAY

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
SubMain()
```

```
Dim n(10)AsInteger
Dim i, j AsInteger

For i =0To10
          n(i)= i +100
      Next i


For j =0To10
Console.WriteLine("Element({0}) = {1}", j, n(j))
Next j
Console.ReadKey()
EndSub
EndModule
```

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

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

Dynamic Arrays

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

Syntax for ReDim statement −

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

Where,

- The **Preserve** keyword helps to preserve the data in an existing array, when you resize it.
- **arrayname** is the name of the array to re-dimension.
- **subscripts** specifies the new dimension.

```
Module arrayApl
SubMain()
Dim marks()AsInteger
ReDim marks(2)
      marks(0)=85
      marks(1)=75
      marks(2)=90

ReDimPreserve marks(10)
      marks(3)=80
      marks(4)=76
      marks(5)=92
      marks(6)=99
      marks(7)=79
      marks(8)=75

For i =0To10
Console.WriteLine(i & vbTab & marks(i))
Next i
Console.ReadKey()
EndSub
EndModule
```

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
SubMain()
' 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 =0To4
For j =0To1
Console.WriteLine("a[{0},{1}] = {2}", i, j, a(i, j))
Next j
Next i
Console.ReadKey()
EndSub
EndModule
```

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
SubMain()
'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 −

Sr.No	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 −

Sr.No	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.

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

Example

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

```
Module arrayApl
SubMain()
Dim list AsInteger()={34,72,13,44,25,30,10}
Dim temp AsInteger()= list
Dim i AsInteger
Console.Write("Original Array: ")

ForEach 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: ")

ForEach i In list
Console.Write("{0} ", i)
Next i
Console.WriteLine()
Console.ReadKey()
EndSub
EndModule
```

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
```

Class

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 instance 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 objects 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.
- **Instance Variables** − Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance 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 −

```
ImportsSystem
PublicClassRectangle
Private length AsDouble
Private width AsDouble

'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 −

```
SharedSubMain()
Dim r AsNewRectangle()
  r.Acceptdetails()
  r.Display()
Console.ReadLine()
EndSub
```

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.

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.

```
FunctionFindMax(ByVal num1 AsInteger,ByVal num2 AsInteger)AsInteger
' 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
FunctionFindMax(ByVal num1 AsInteger,ByVal num2 AsInteger)AsInteger
' 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 AsInteger)AsInteger
' 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()
EndSub
EndModule
```

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
FunctionAddElements(ParamArray arr AsInteger())AsInteger
Dim sum AsInteger=0
Dim i AsInteger=0

ForEach i In arr
         sum += i
Next i
Return sum
EndFunction
SubMain()
Dim sum AsInteger
      sum =AddElements(512,720,250,567,889)
Console.WriteLine("The sum is: {0}", sum)
Console.ReadLine()
EndSub
EndModule
```

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 AsInteger(),ByVal size AsInteger)AsDouble
'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 with5 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()
EndSub
EndModule
```

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

```
Average value is: 214.4
```

Sub Procedures

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
SubCalculatePay(ByRef hours AsDouble,ByRef wage AsDecimal)
'local variable declaration
      Dim pay As Double
      pay = hours * wage
      Console.WriteLine("Total Pay: {0:C}", pay)
   End Sub
   Sub Main()
      'calling the CalculatePaySubProcedure
CalculatePay(25,10)
CalculatePay(40,20)
CalculatePay(30,27.5)
Console.ReadLine()
EndSub
EndModule
```

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 AsInteger,ByVal y AsInteger)
Dim temp AsInteger
      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 AsInteger=100
Dim b AsInteger=200
Console.WriteLine("Before swap, value of a : {0}", a)
Console.WriteLine("Before swap, value of b : {0}", b)
' calling a function to swap the values '
      swap(a, b)
```

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

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 AsInteger,ByRef y AsInteger)
Dim temp AsInteger
      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 AsInteger=100
Dim b AsInteger=200
Console.WriteLine("Before swap, value of a : {0}", a)
Console.WriteLine("Before swap, value of b : {0}", b)
' calling a function to swap the values '
      swap(a, b)
Console.WriteLine("After swap, value of a : {0}", a)
Console.WriteLine("After swap, value of b : {0}", b)
Console.ReadLine()
EndSub
EndModule
```

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
```

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 ]
EndClass
```

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
ClassBox
Public length AsDouble' Length of a box
      Public breadth As Double  'Breadthof a box
Public height AsDouble' Height of a box
   End Class
   Sub Main()
      Dim Box1 As Box = New Box()        'DeclareBox1of type Box
```

```
DimBox2AsBox=NewBox()' 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()
EndSub
EndModule
```

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
ClassBox
Public length AsDouble' Length of a box
      Public breadth As Double   'Breadthof a box
Public height AsDouble' 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()        'DeclareBox1of type Box
DimBox2AsBox=NewBox()' Declare Box2 of type Box
      Dim volume As Double = 0.0         'Store the volume of a box here

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

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

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

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

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 −

```
ClassLine
Private length AsDouble' Length of a line
   Public Sub New()    'constructor
Console.WriteLine("Object is being created")
EndSub

PublicSub setLength(ByVal len AsDouble)
      length = len
```

```
EndSub

PublicFunction getLength()AsDouble
Return length
EndFunction
SharedSubMain()
Dim line AsLine=NewLine()
'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 −

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

PublicFunction getLength()AsDouble
Return length
EndFunction
SharedSubMain()
Dim line AsLine=NewLine(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 explains the concept of destructor −

```
ClassLine
Private length AsDouble
   Public Sub New()
Console.WriteLine("Object is being created")
EndSub

ProtectedOverridesSubFinalize()' 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()
  line.setLength(6.0)
Console.WriteLine("Length of line : {0}", line.getLength())
Console.ReadKey()
EndSub
EndClass
```

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 −

```
ClassStaticVar
PublicShared num AsInteger
PublicSub count()
      num = num +1
EndSub
PublicSharedFunction getNum()AsInteger
Return num
EndFunction
SharedSubMain()
Dim s AsStaticVar=NewStaticVar()
      s.count()
      s.count()
      s.count()
Console.WriteLine("Value of variable num: {0}",StaticVar.getNum())
Console.ReadKey()
EndSub
EndClass
```

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
'Derivedclass
ClassRectangle:InheritsShape
PublicFunction getArea()AsInteger
Return(width * height)
EndFunction
EndClass
ClassRectangleTester
SharedSubMain()
Dim rect AsRectangle=NewRectangle()
      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
   'endclassRectangle
EndClass

'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
    'endclassTabletop
EndClass
ClassRectangleTester
SharedSubMain()
Dim t AsTabletop=NewTabletop(4.5,7.5)
      t.Display()
Console.ReadKey()
EndSub
EndClass
```

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.

NAMESPACE

Software projects consist of several pieces of code such as classes, declarations, procedures and functions etc., known as the component or identifiers of the software project. In large projects the number of these components can be very large. These components can be grouped into smaller subcategories. This logical grouping construct is known as a "Namespace" or we can say that the group of code having a specific name is a "Namespace". In a Namespace the groups of components are somehow related to each other. Namespaces are similar in concept to a folder in a computer file system. Like folders, namespaces enable classes to have a unique name or we can say that it is a logical naming scheme for grouping related types. A Namespace is sometimes also called a name scope. An identifier defined in a Namespace belongs to that Namespace and the same identifier can be independently defined in multiple Namespaces with a different or the same meaning. Every project in C# or VB.NET starts with a Namespace, by default the same name as the name of the project.

Why we need it

We must add a reference of the Namespace object before using that object in a project. Several references are automatically added in the project by default. The VB.Net "Imports" keyword is used to add a reference of a namespace manually.

Example

```
Imports System
```

Note: Imports allow access to classes in the referenced Namespace only not in its internal or child Namespaces. If we want to access internal Namespace we might need to write:

```
Imports System.Collections
```

Namespaces are basically used to avoid naming collisions, when we have multiple classes with the same name, and it is also helpful for organizing classes libraries in a hierarchal structure. Namespaces allow us to organize Classes so that they can be easily accessed in other applications. Namespaces also enable reusability.

A class in .Net Framework cannot belong to multiple Namespaces. One class should belong to only one Namespace. VB.NET does not allow two classes with the same name to be used in a program.

We can define a Namespace using the "Namespace" keyword. The syntax for declaring a Namespace is:

```
Namespace <Namespace_name>

    // Classes and/or structs and/or enums etc.

End Namespace
```

Example

Note: All the classes in the .Net Framework belongs to the System Namespace. The "system" Namespace has built-in VB functionality and all other Namespaces are based on this "system" Namespace.

Accessing Members of a Namespace

We can access a member of a Namespace by using a dot(.) operator, also known as the period operator. The members of a Namespace are the variables, procedures and classes that are defined within a Namespace. To access the member of a namespace in a desired location type the name of the namespace followed by the dot or period operator followed by the desired member of the namespace.

Example

```
MyNamespace.Class1.disp()   'Accessing elements of the MyNamspace
```

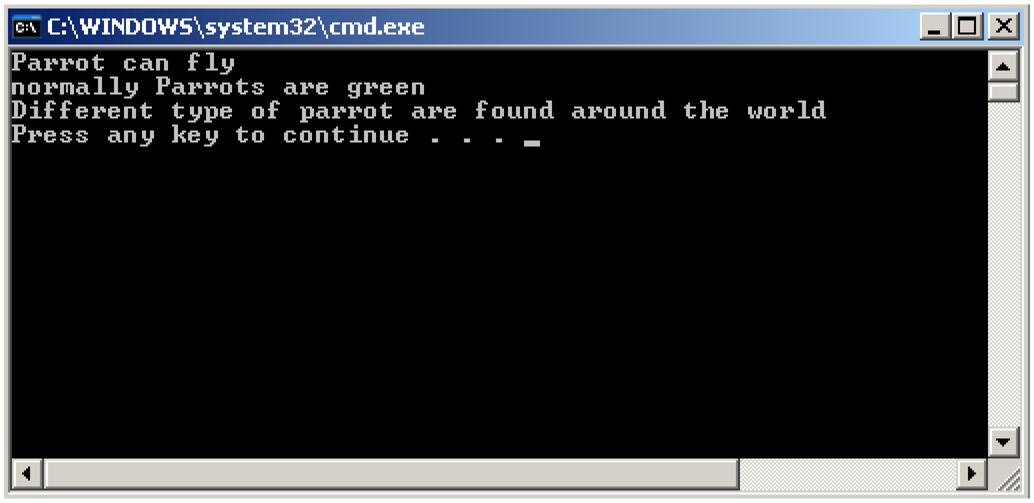
we can access a member of a namespace in various ways. The following program shows accessing the element of a namespace in various ways.

```
Imports System
Namespace Birds    'user defined namespace Bird
    Class Parrot  'Parrot is a class in the namespace Animals
        Public Shared Function fly()  'Fly is a function in this Class
            Console.WriteLine("Parrot can fly")
        End Function
        Public Shared Function color() ' color is another function in parrot class
            Console.WriteLine("normally Parrots are green")
        End Function
        Public Shared Function type()
            Console.WriteLine("Different type of parrot are found around the world")
        End Function
    End Class
End Namespace

Module Module1
    Public Function myfunction()
        Dim P As Birds.Parrot
        P = New Birds.Parrot()
        P.type()     'accessing member of the namespace bird
    End Function
    Sub main()
        Console.Clear()
        Birds.Parrot.fly()       'accessing member of the namespace
        ConsoleApplication5.Birds.Parrot.color() 'another way to access member of the namespace
        myfunction()
    End Sub
End Module
```

Output

```
Namespace MyNamespace                 'class with in a namespace
    Public Class Class1
        Public Shared Function disp()       'function declared within the class
```

```
            Console.Write("hello" & vbCrLf)
        End Function
    End Class
End Namespace
```



```
Parrot can fly
normally Parrots are green
Different type of parrot are found around the world
Press any key to continue . . .
```

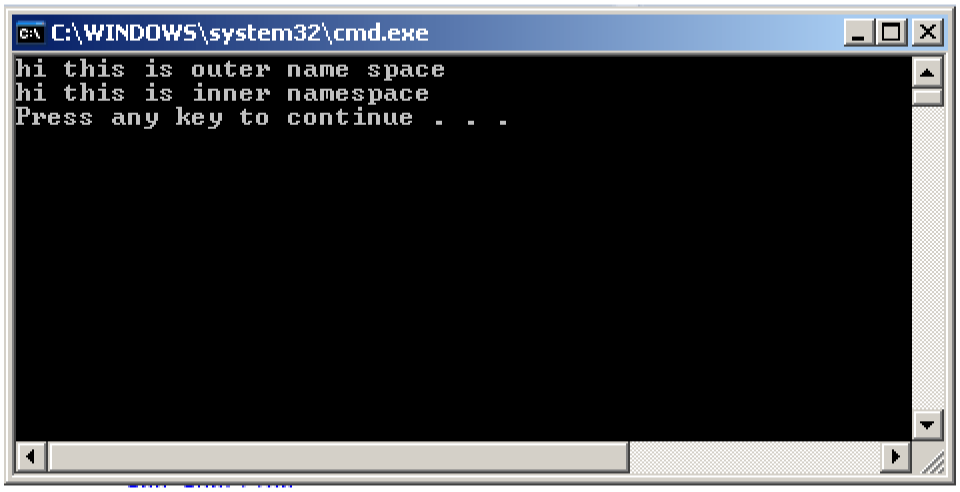
Nesting a Namespace

Nesting a Namespace means create a namespace inside a namespace. A good way to organize namespaces is to put them in a hierarchal order, i.e. general name at the top of the hierarchy and put specific names at the lower level.

Example

```
Imports System
Namespace outer   'declare an outer namespace
    Public Class nameout
        Public Shared Function disp()  'create a function inside a outer namespace
            Console.WriteLine("hi this is outer name space")
        End Function
    End Class

    Namespace inner  'decalre an inner namespace
        Public Class nameinn
            Public Shared Function disp()  'create a function inside the inner namespace
                Console.WriteLine("hi this is inner namespace")
            End Function
        End Class
    End Namespace

End Namespace

Module module1
    Sub main()
        Console.Clear()
        outer.nameout.disp()       'accesing function of the outer namespace
        outer.inner.nameinn.disp()  'accessing fucntion of the inner namespace
    End Sub
End Module

```

Output



```
hi this is outer name space
hi this is inner namespace
Press any key to continue . . .
```

Note: You can not have two classes with the same name in the same scope. In other words, class overloading is not allowed.

Example

```
Namespace MyNamespace
    Public Class one   'sample class with in a namespace
        Public Shared Function disp()  'function declared within the class
            Console.Write("hello" & vbCrLf)
        End Function
    End Class

    Public Class one  'this is not allowed
        Public Shared Function disp1()
            Console.Write("hi")
        End Function
    End Class
End Namespace
```

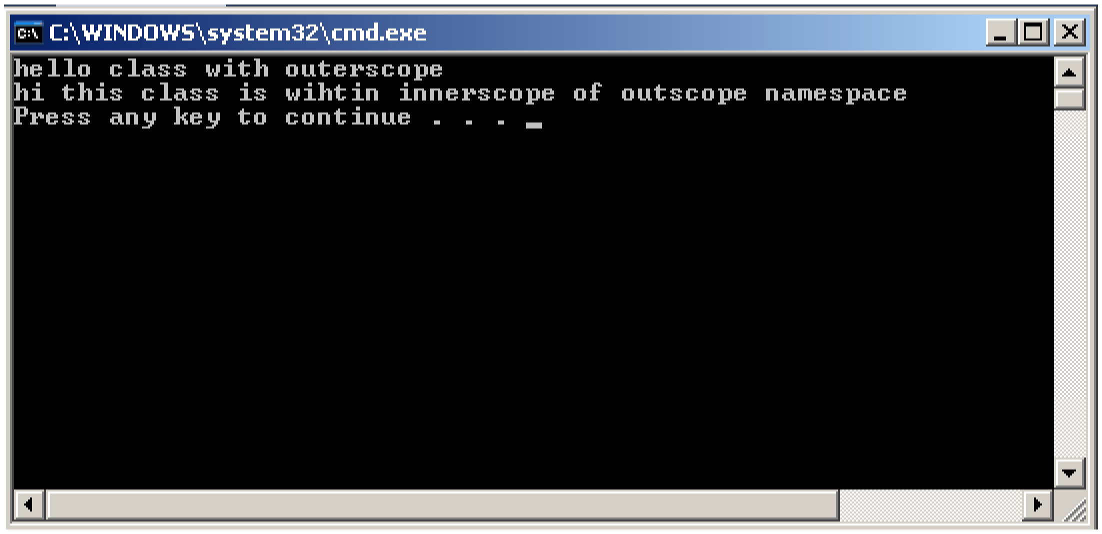
You can avoid this by putting classes with the same name in a different scope.

Example

```
Namespace outerscope          'sample class with in a namespace
    Public Class one
        Public Shared Function disp()  'function declared within the class
            Console.Write("hello class with outerscope" & vbCrLf)
        End Function
    End Class
    Namespace innerscope
```

```
        Public Class one    'same class with different scope
          Public Shared Function disp1()
            Console.Write("hi this class is wihtin innerscope of outscope namespace " & vbCrLf)

          End Function
        End Class
    End Namespace
End Namespace

Module module1
    Sub main()
        Console.Clear()
        outerscope.one.disp()
        outerscope.innerscope.one.disp1()
    End Sub
End Module
```

Output



```
hello class with outerscope
hi this class is wihtin innerscope of outscope namespace
Press any key to continue . . .
```

Delegates and Events

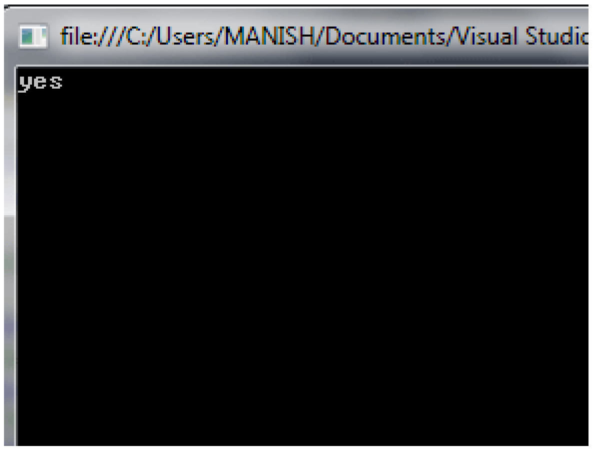
A delegate is a class that can hold a reference to a method. Unlike other classes, a delegate class has a signature, and it can hold references only to methods that match its signature. A delegate is thus equivalent to a type-safe function pointer or a callback. Although delegates have other uses, the discussion here focuses on the event-handling functionality of delegates.

Events in VB.NET are handled by delegates, which serve as a mechanism that defines one or more callback functions to process events. An event is a message sent by an object to signal the occurrence of an action. The action could arise from user interaction, such as a mouse click, or could be triggered by some other program logic. The object that triggers the event is called the event sender. The object that captures the event and responds to it is called the event receiver.

In event communication, the event sender class does not know which object or method will handle the events it raises. It merely functions as an intermediary or pointer-like mechanism between the source and the receiver, as illustrated in Listing 5.47. The .NET framework defines a special type delegate that serves as a function pointer.

Listing 5.47: DelegateEvent.VB, Delegates and Events Example

```
Public Class MyEvt
   Public Delegate Sub t(ByVal sender As [Object], ByVal e As MyArgs)
   ' declare a delegate
   Public Event tEvt As t
   'declares an event for the delegate
   Public Sub mm()
      'function that will raise the callback
      Dim r As New MyArgs()
      RaiseEvent tEvt(Me, r)
      'calling the client code
   End Sub
   Public Sub New()
   End Sub
End Class
'arguments for the callback
Public Class MyArgs
   Inherits EventArgs
   Public Sub New()
   End Sub
End Class
Public Class MyEvtClient
   Private oo As MyEvt
   Public Sub New()
      Me.oo = New MyEvt()
      AddHandler Me.oo.tEvt, New MyEvt.t(AddressOf oo_tt)
   End Sub
   Public Shared Sub Main(ByVal args As [String]())
      Dim cc As New MyEvtClient()
      cc.oo.mm()
   End Sub
   'this code will be called from the server
   Public Sub oo_tt(ByVal sender As Object, ByVal e As MyArgs)
      Console.WriteLine("yes")
      Console.ReadLine()
   End Sub
End Class
```



```
yes
```

EXCEPTIONS HANDLING

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

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

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

Syntax

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

```
Try
   [ tryStatements ]
   [ Exit Try ]
[ Catch [ exception [ As type ] ] [ When expression ]
```

```
   [ catchStatements ]
   [ Exit Try ] ]
[ Catch ... ]
[ Finally
   [ finallyStatements ] ]
End Try
```

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

Exception Classes in .Net Framework

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

The **System.ApplicationException** class supports exceptions generated by application programs. So the exceptions defined by the programmers should derive from this class.

The **System.SystemException** class is the base class for all predefined system exception.

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

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

System.StackOverflowException	Handles errors generated from stack overflow.

Handling Exceptions

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

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

```
Module exceptionProg
Sub division(ByVal num1 AsInteger,ByVal num2 AsInteger)
Dim result AsInteger
Try
         result = num1 \ num2
Catch e AsDivideByZeroException
Console.WriteLine("Exception caught: {0}", e)
Finally
Console.WriteLine("Result: {0}", result)
EndTry
EndSub
SubMain()
      division(25,0)
Console.ReadKey()
EndSub
EndModule
```

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
PublicClassTempIsZeroException:InheritsApplicationException
PublicSubNew(ByVal message AsString)
MyBase.New(message)
EndSub
EndClass
PublicClassTemperature
Dim temperature AsInteger=0
Sub showTemp()
If(temperature =0)Then
Throw(NewTempIsZeroException("Zero Temperature found"))
Else
```

```
Console.WriteLine("Temperature: {0}", temperature)
EndIf
EndSub
EndClass
SubMain()
Dim temp AsTemperature=NewTemperature()
Try
         temp.showTemp()
Catch e AsTempIsZeroException
Console.WriteLine("TempIsZeroException: {0}", e.Message)
EndTry
Console.ReadKey()
EndSub
EndModule
```

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
SubMain()
Try
ThrowNewApplicationException("A custom exception _ is being thrown here...")
Catch e AsException
Console.WriteLine(e.Message)
Finally
Console.WriteLine("Now inside the Finally Block")
EndTry
Console.ReadKey()
EndSub
EndModule
```

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
```

THREAD

When two or more processes execute simultaneously in a program, the process is known as multithreading. And the execution of each process is known as the **thread**. A single thread is used to execute a single logic or task in an application. By default, each application has one or more threads to execute each process, and that thread is known as the **main thread**.

To create and access a new thread in the Thread **class**, we need to import the **System.Threading** namespace. When the execution of a program begins in VB.NET, the Main thread is automatically called to handle the program logic. And if we create another thread to execute the process in Thread class, the new thread will become the **child** thread for the **main** thread.

Create a new Thread

In VB.NET, we can create a thread by extending the Thread class and pass the ThreadStart delegate as an argument to the Thread constructor. A **ThreadStart()** is a method executed by the new thread. We need to call the **Start()** method to start the execution of the new thread because it is initially in the **unstart** state. And the PrintInfo parameter contains **an executable statement that** is executed when creating a new thread.

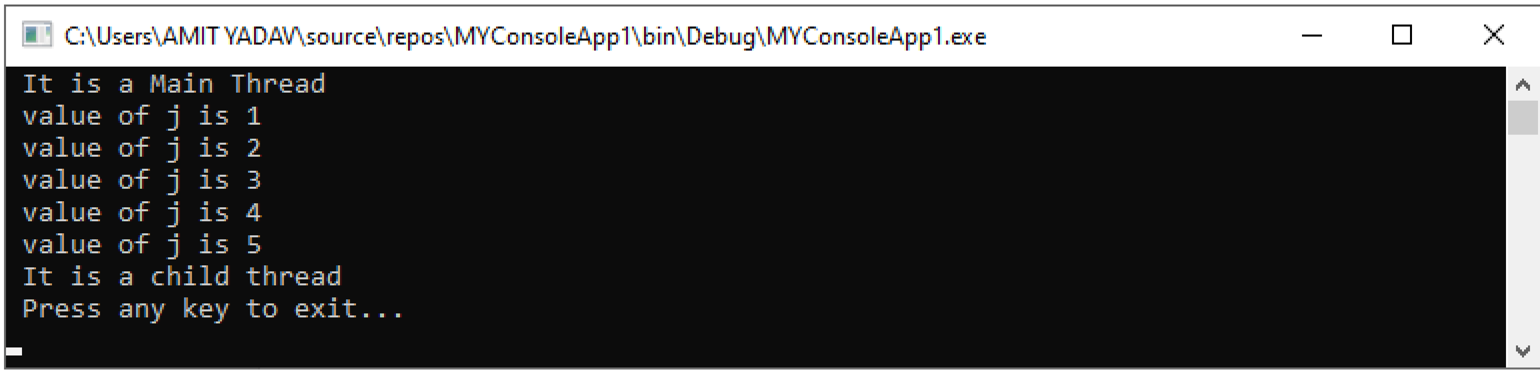
```
' Create a new thread
Dim th As Thread = New Thread( New ThreadStart(PrintInfo) )
' Start the execution of newly thread
th.Start()
```

Let's write a program to create and access the thread in Thread class.

create_Thread.vb

```
Imports System.Threading 'Imports the System.Threading namespace.
Module create_Thread
    Sub Main(ByVal args As String())
        ' create a new thread
        Dim th As Thread = New Thread(New ThreadStart(AddressOf PrintInfo))
        ' start the newly created thread
        th.Start()
        Console.WriteLine(" It is a Main Thread")
    End Sub
    Private Sub PrintInfo()
        For j As Integer = 1 To 5
            Console.WriteLine(" value of j is {0}", j)
        Next j
        Console.WriteLine(" It is a child thread")
        Console.WriteLine(" Press any key to exit...")
        Console.ReadKey()
    End Sub
End Module
```

Output:


C:\Users\AMIT YADAV\source\repos\MYConsoleApp1\bin\Debug\MYConsoleApp1.exe

```
It is a Main Thread
value of j is 1
value of j is 2
value of j is 3
value of j is 4
value of j is 5
It is a child thread
Press any key to exit...
```

In the above program, the main and child threads begin their execution simultaneously. The execution of the main thread is stopped after completing its function, but the child thread will continue to execute until its task is finished.

VB.NET Thread Methods

The following are the most commonly used methods of Thread class.

Method	Description
Abort()	As the name defines, it is used to terminate the execution of a thread.
AllocateDataSlot()	It is used to create a slot for unnamed data on all threads.
AllocateNamedDatsSlot()	It is used to create a slot for defined data on all threads.
Equals()	It is used to check whether the current and defined thread object are equ
Interrupt()	It is used to interrupt a thread from the Wait, sleep, and join thread state
Join()	It is a synchronization method that stops the calling thread until the exec
Resume()	As the name suggests, a Resume() method is used to resume a thread t
Sleep()	It is used to suspend the currently executing thread for a specified time.
Start()	It is used to start the execution of thread or change the state of an ongoi

Suspend()	It is used to stop the currently executing thread.

VB.NET Thread Life Cycle

In VB.NET Multithreading, each thread has a life cycle that starts when a new object is created using the Thread Class. Once the task defined by the thread class is complete, the life cycle of a thread will get the end.

There are some states of thread cycle in VB.NET programming.

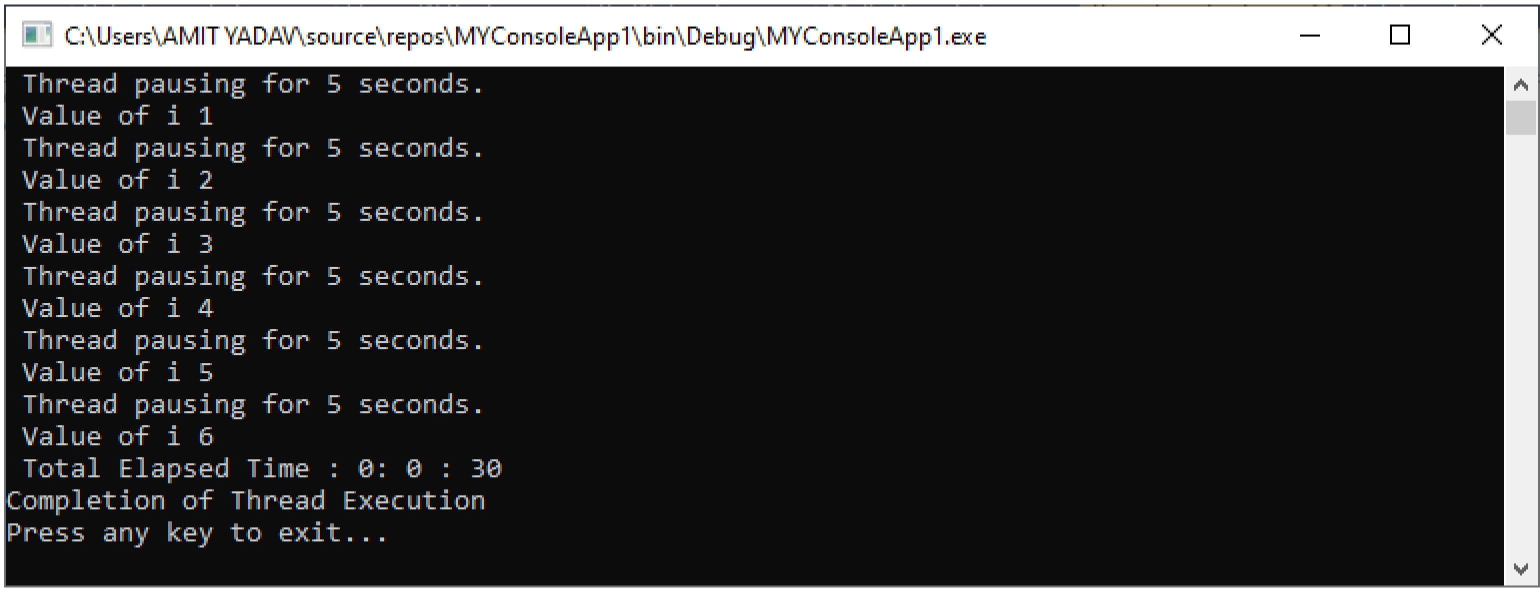
State	Description
Unstarted	When we create a new thread, it is initially in an unstarted state.
Runnable	When we call a Start() method to prepare a thread for running, the runnable situation oc
Running	A Running state represents that the current thread is running.
Not Runnable	It indicates that the thread is not in a runnable state, which means that the thread in slee blocked by the I/O operation.
Dead	If the thread is in a dead state, either the thread has been completed its work or aborted

Let's create a program to manage a thread by using various methods of Thread Class.

Thread_cycle.vb

```
Imports System.Threading
Module Thread_cycle
    Sub Main(ByVal args As String())
        Dim s As Stopwatch = New Stopwatch()
        s.Start()
        Dim t As Thread = New Thread(New ThreadStart(AddressOf PrintInfo))
        t.Start()
        ' Halt another thread execution until the thread execution completed
        t.Join()
        s.[Stop]()
        Dim t1 As TimeSpan = s.Elapsed
        Dim ft As String = String.Format("{0}: {1} : {2}", t1.Hours, t1.Minutes, t1.Seconds)
        Console.WriteLine(" Total Elapsed Time : {0}", ft)
        Console.WriteLine("Completion of Thread Execution ")
        Console.WriteLine("Press any key to exit...")
        Console.ReadKey()
    End Sub
    Private Sub PrintInfo()
        For j As Integer = 1 To 6
            Console.WriteLine(" Halt Thread for {0} Second", 5)
            ' It pause thread for 5 Seconds
            Thread.Sleep(5000)
            Console.WriteLine(" Value of i {0}", j)
        Next
    End Sub
End Module
```

Output:



```
C:\Users\AMIT YADAV\source\repos\MYConsoleApp1\bin\Debug\MYConsoleApp1.exe
 Thread pausing for 5 seconds.
 Value of i 1
 Thread pausing for 5 seconds.
 Value of i 2
 Thread pausing for 5 seconds.
 Value of i 3
 Thread pausing for 5 seconds.
 Value of i 4
 Thread pausing for 5 seconds.
 Value of i 5
 Thread pausing for 5 seconds.
 Value of i 6
 Total Elapsed Time : 0: 0 : 30
Completion of Thread Execution
Press any key to exit...
```

In the above example, we used a different method of the **Thread** class such as the **Start()** method to start execution of the thread, the **Join()** method is used to stop the execution of the thread until the execution of the thread was completed. The **Sleep()** method is used to pause the execution of threads for **5** seconds.

Multithreading

When two or more processes are executed in a program to simultaneously perform multiple tasks, the process is known as **multithreading**.

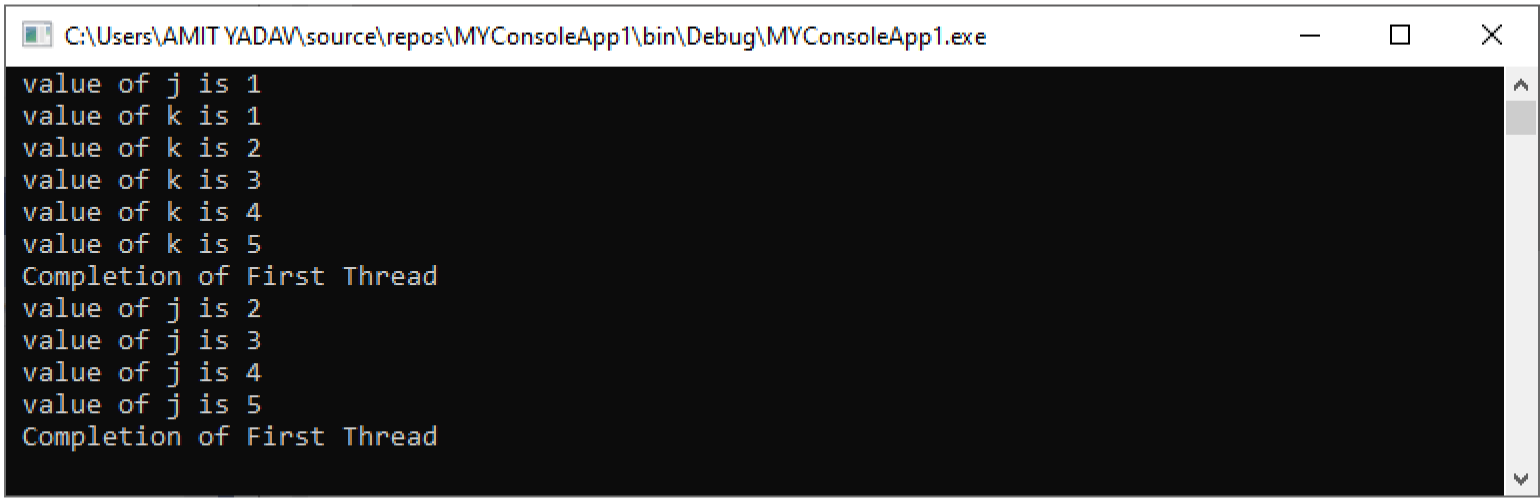
When we execute an application, the Main thread will automatically be called to execute the programming logic synchronously, which means it executes one process after another. In this way, the second process has to wait until the first process is completed, and it takes time. To overcome that situation, VB.NET introduces a new concept Multithreading to execute multiple tasks at the same time by creating multiple threads in a program.

Let's write a program of multiple threads to execute the multiple tasks at the same time in the VB.NET application.

Multi_thread.vb

```
Imports System.Threading
Module Multi_thread
    Sub Main(ByVal arg As String())
        Dim th As Thread = New Thread(New ThreadStart(AddressOf PrintInfo))
        Dim th2 As Thread = New Thread(New ThreadStart(AddressOf PrintInfo2))
        th.Start()
        th2.Start()
        Console.ReadKey()
    End Sub
    Private Sub PrintInfo()
        For j As Integer = 1 To 5
            Console.WriteLine(" value of j is {0}", j)
            Thread.Sleep(1000)
        Next
        Console.WriteLine(" Completion of First Thread")
    End Sub
    Private Sub PrintInfo2()
        For k As Integer = 1 To 5
            Console.WriteLine(" value of k is {0}", k)
        Next
        Console.WriteLine(" Completion of second thread")
    End Sub
End Module
```

Output:


C:\Users\AMIT YADAV\source\repos\MYConsoleApp1\bin\Debug\MYConsoleApp1.exe

```
value of j is 1
value of k is 1
value of k is 2
value of k is 3
value of k is 4
value of k is 5
Completion of First Thread
value of j is 2
value of j is 3
value of j is 4
value of j is 5
Completion of First Thread
```

In the above example, we have created two threads (**th, th2**) to execute the **PrintInfo** and **PrintInfo2** method at the same time. And when execution starts, both threads execute simultaneously. But the first statement of the PrintInfo method is executed, and then it waits for the next statement until the PrintInfo2 method is completed in the program.

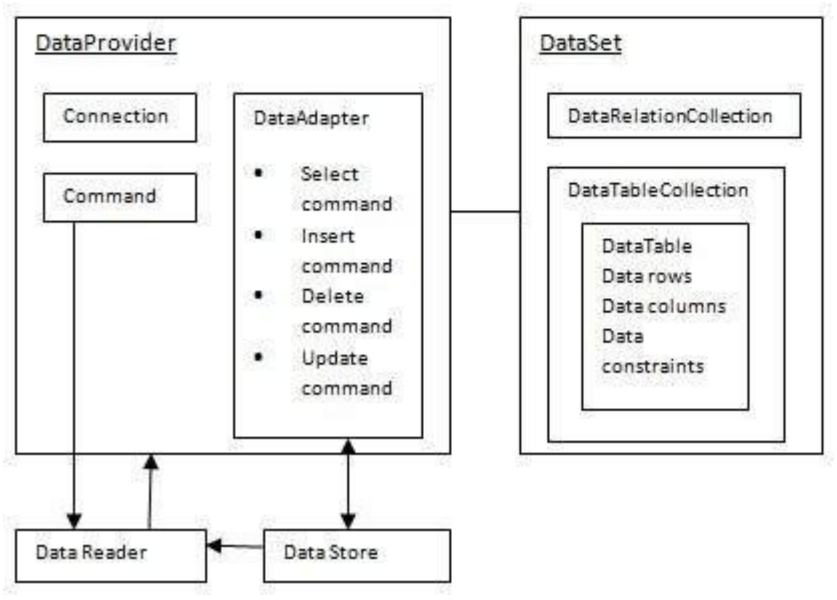
VB.Net - Database Access

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

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

ADO.Net Object Model

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



The data residing in a data store or database is retrieved through the **data provider**. Various components of the data provider 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 −

Sr.No.	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.

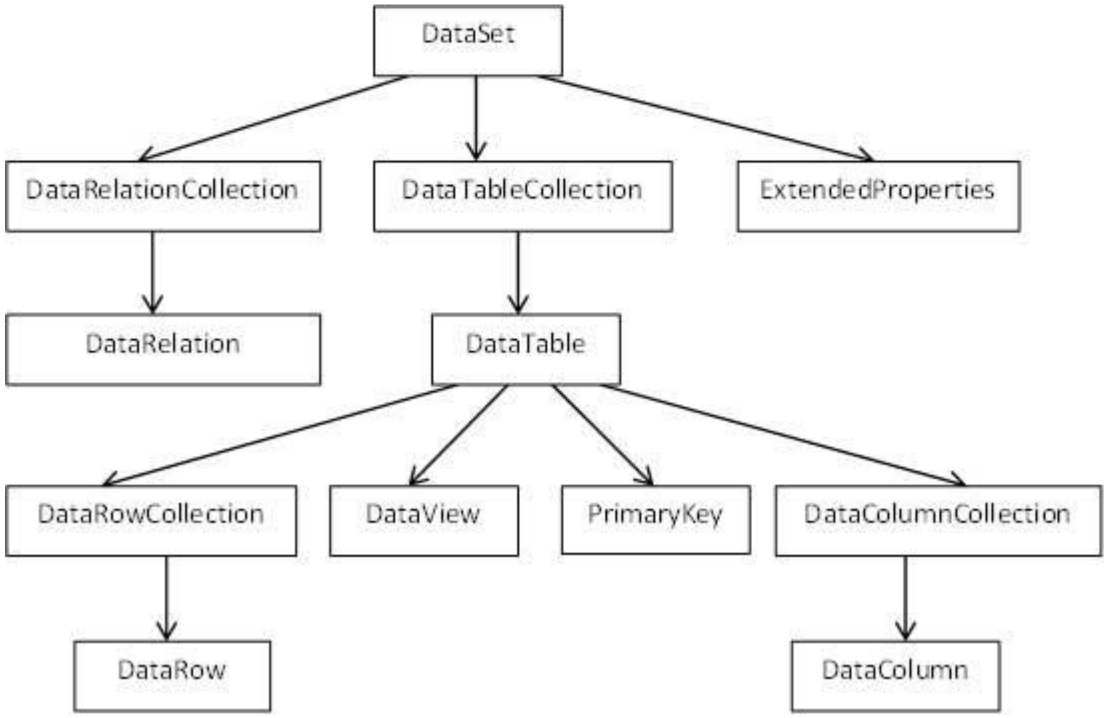
There are following different types of data providers included in ADO.Net

- 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 −

Sr.No.	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 −

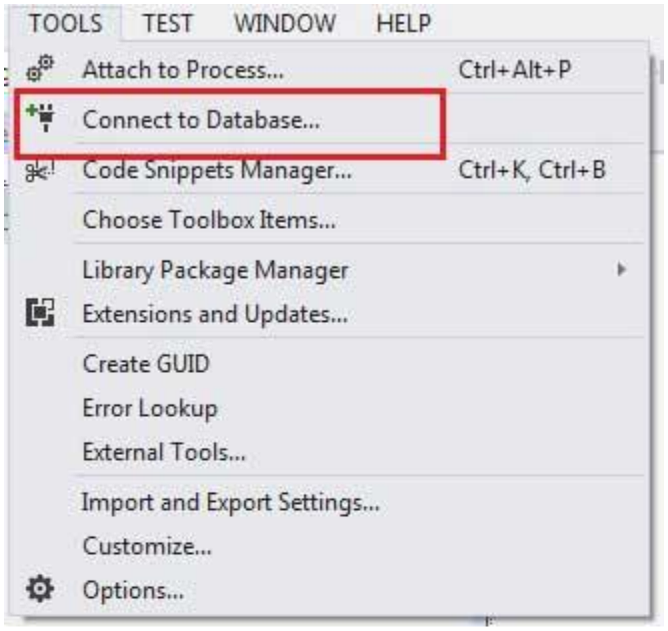
- **SqlConnection** − designed for connecting to Microsoft SQL Server.
- **OleDbConnection** − designed for connecting to a wide range of databases, like Microsoft Access and Oracle.

Example 1

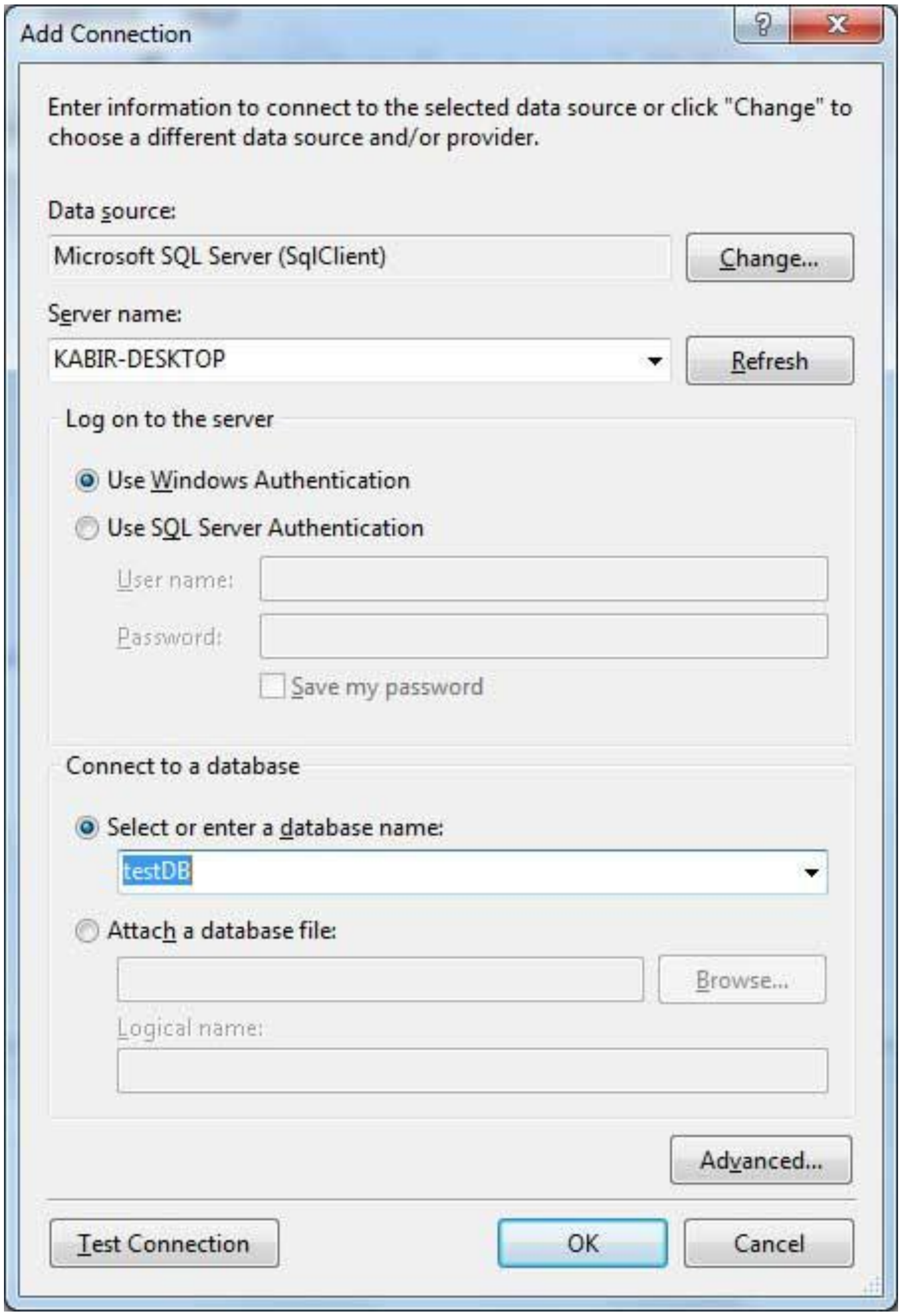
We have a table stored in Microsoft SQL Server, named Customers, in a database named testDB. Please consult 'SQL Server' tutorial for creating databases and database tables in SQL Server.

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.


Add Connection

Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.

Data source: Microsoft SQL Server (SqlClient)

Server name: KABIR-DESKTOP

Log on to the server

(●) Use Windows Authentication

( ) Use SQL Server Authentication

User name:

Password:

[ ] Save my password

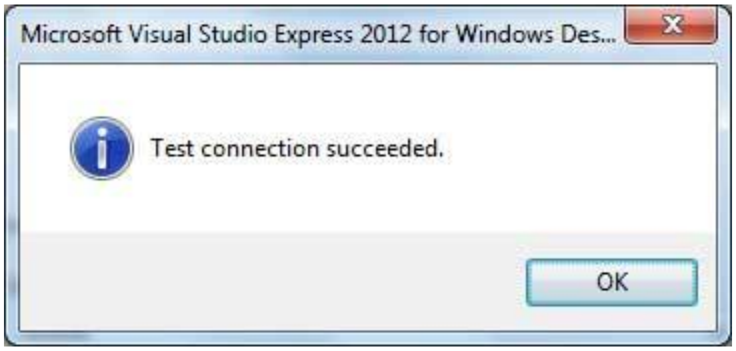
Connect to a database

(●) Select or enter a database name: testDB

( ) Attach a database file:

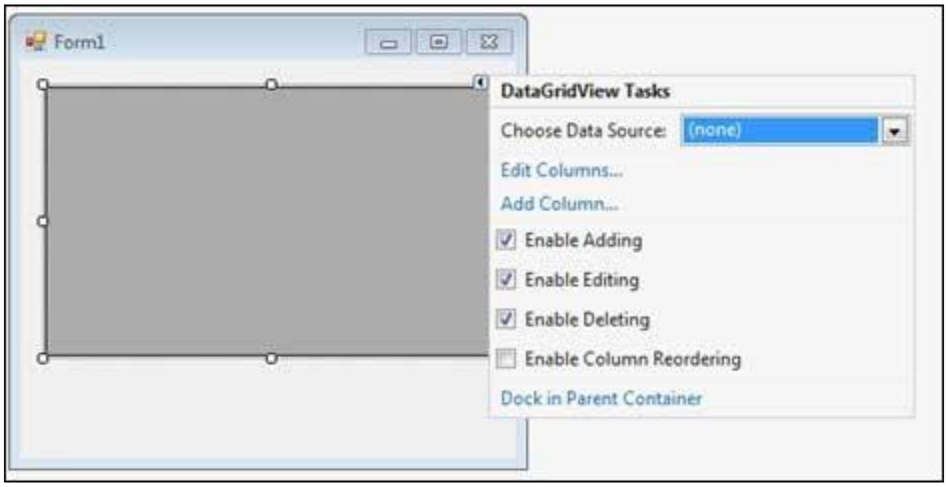
Logical name:

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

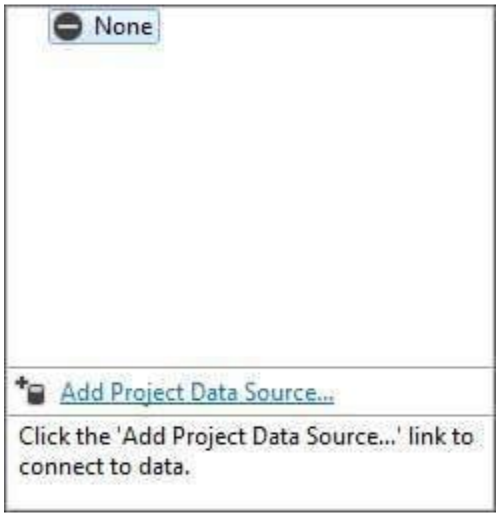

Microsoft Visual Studio Express 2012 for Windows Des...

Test connection succeeded.

- Add a DataGridView on the form.

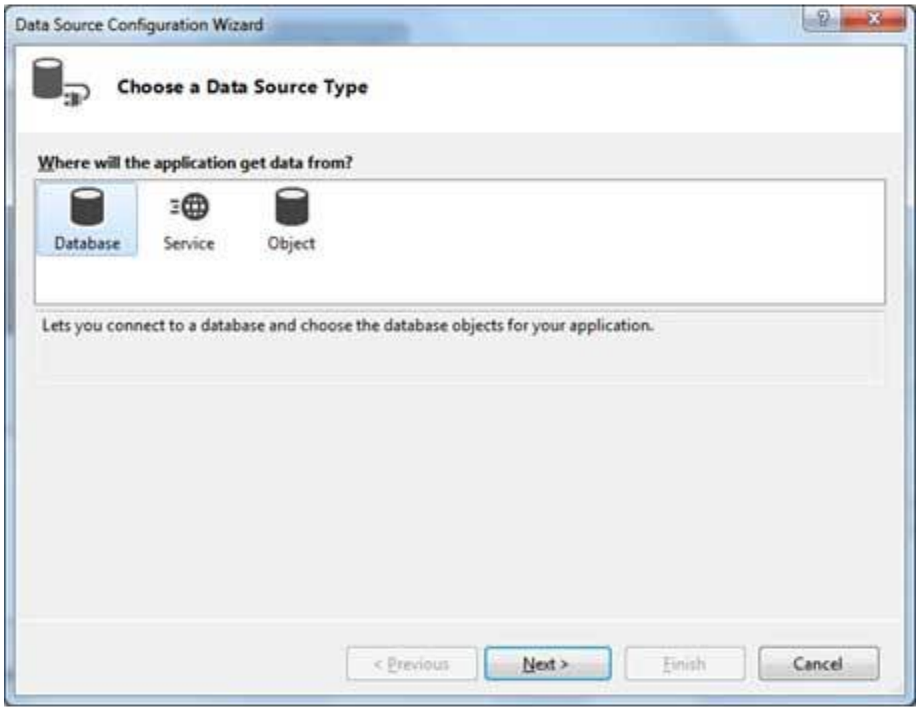


- Click on the Choose Data Source combo box.
- Click on the Add Project Data Source link.



Click the 'Add Project Data Source...' link to connect to data.

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

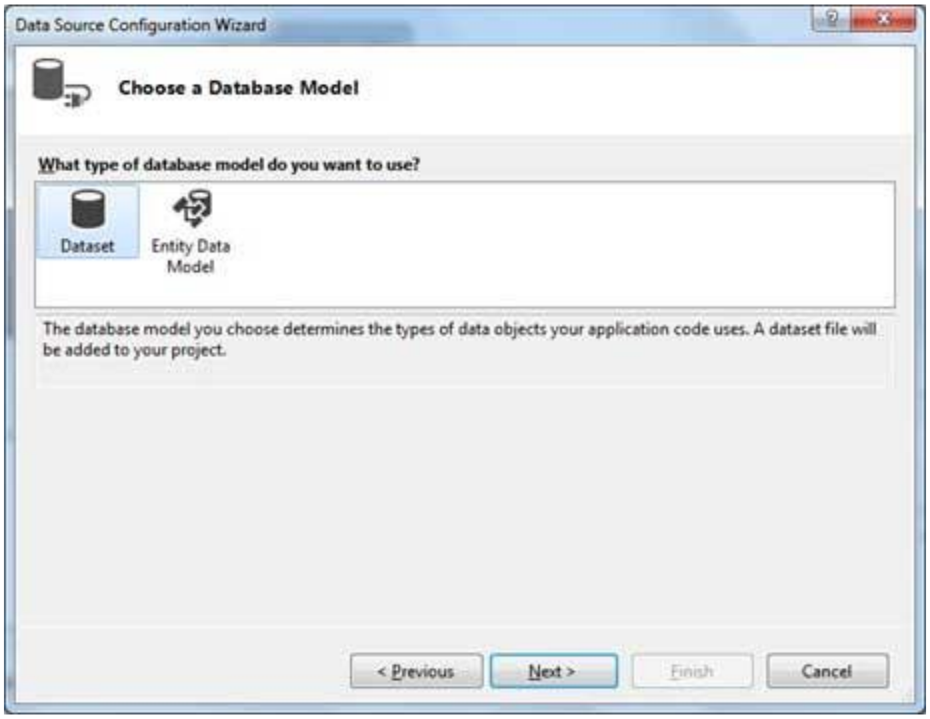

Data Source Configuration Wizard

Choose a Data Source Type

Where will the application get data from?

Lets you connect to a database and choose the database objects for your application.

- Choose DataSet as the database model.

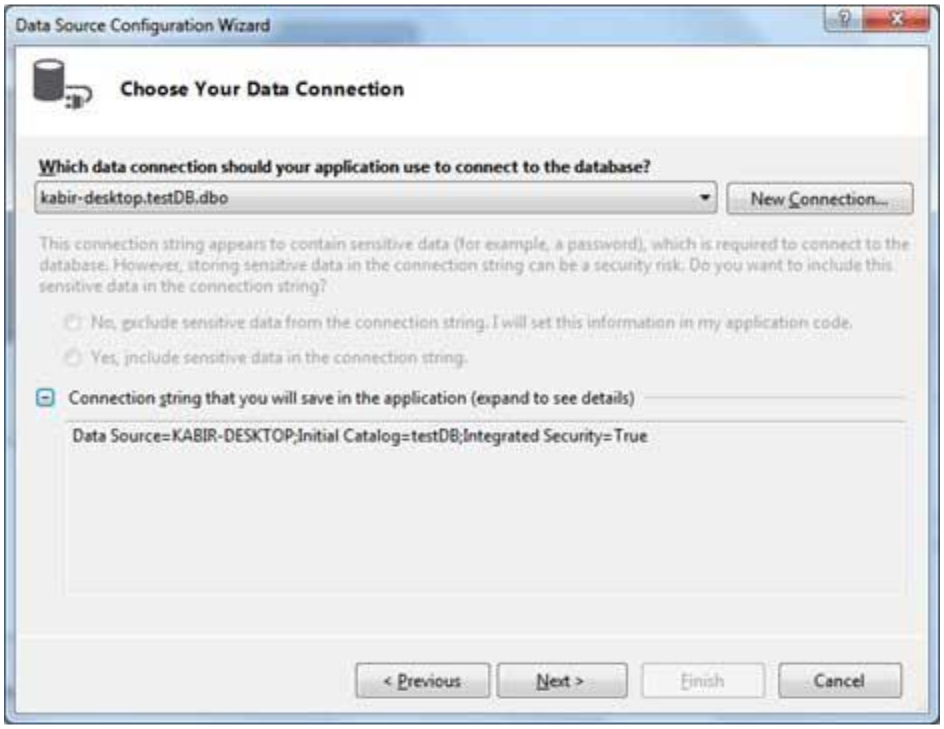

Data Source Configuration Wizard

Choose a Database Model

What type of database model do you want to use?

The database model you choose determines the types of data objects your application code uses. A dataset file will be added to your project.

- Choose the connection already set up.



Which data connection should your application use to connect to the database?

kabir-desktop.testDB.dbo

This connection string appears to contain sensitive data (for example, a password), which is required to connect to the database. However, storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

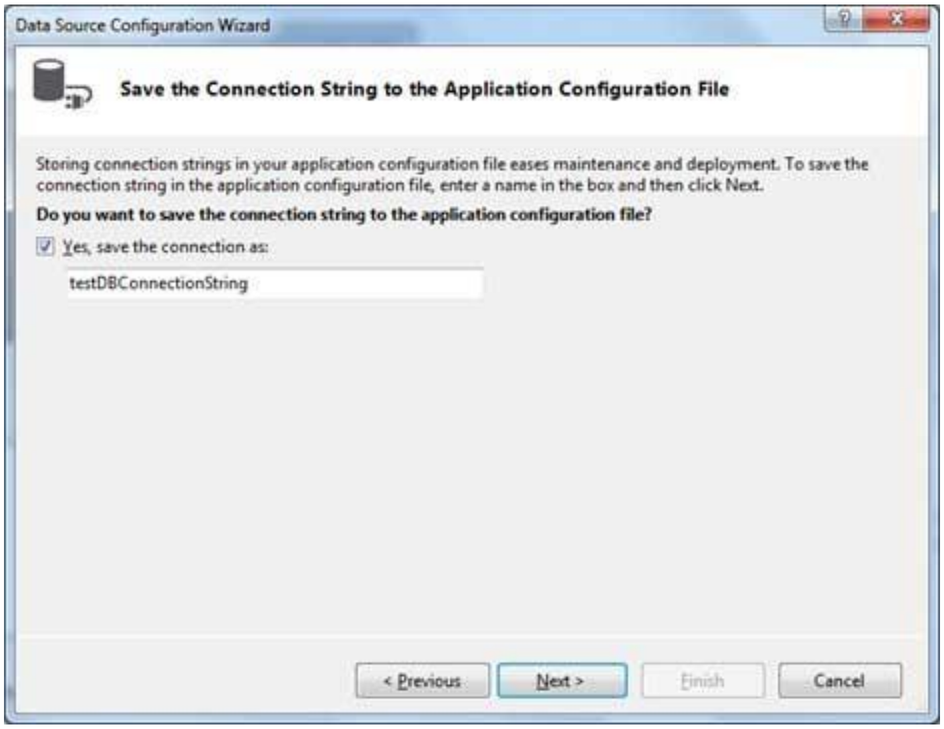
No, exclude sensitive data from the connection string. I will set this information in my application code.

Yes, include sensitive data in the connection string.

Connection string that you will save in the application (expand to see details)

```
Data Source=KABIR-DESKTOP;Initial Catalog=testDB;Integrated Security=True
```

- Save the connection string.


Data Source Configuration Wizard
Save the Connection String to the Application Configuration File

Storing connection strings in your application configuration file eases maintenance and deployment. To save the connection string in the application configuration file, enter a name in the box and then click Next.

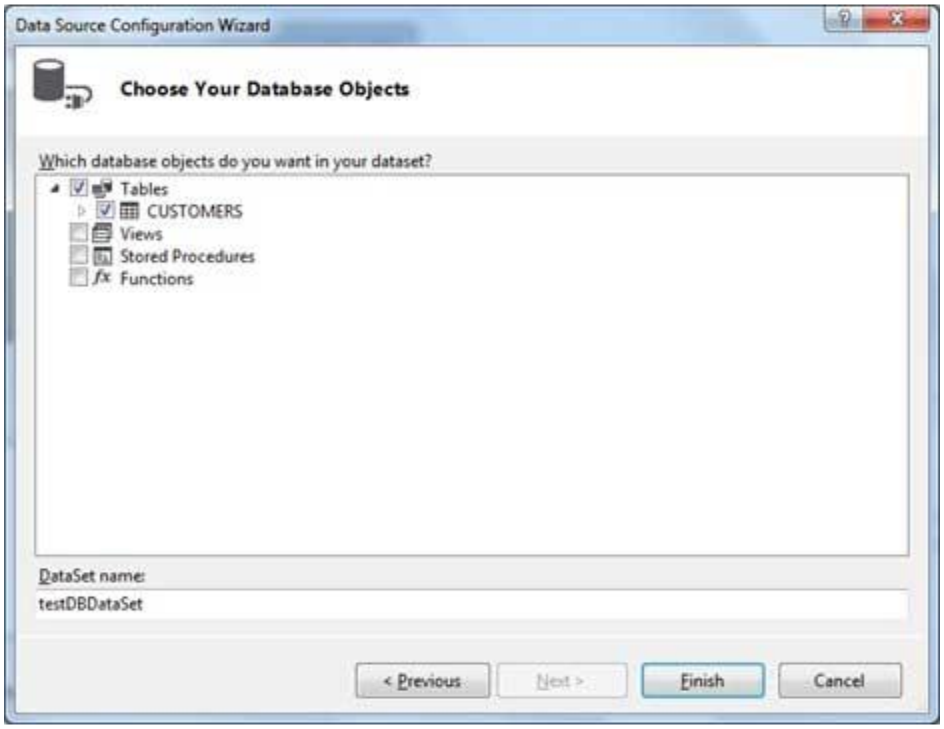
Do you want to save the connection string to the application configuration file?

☑ Yes, save the connection as:

testDBConnectionString

< Previous
Next >
Finish
Cancel

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


Data Source Configuration Wizard
Choose Your Database Objects

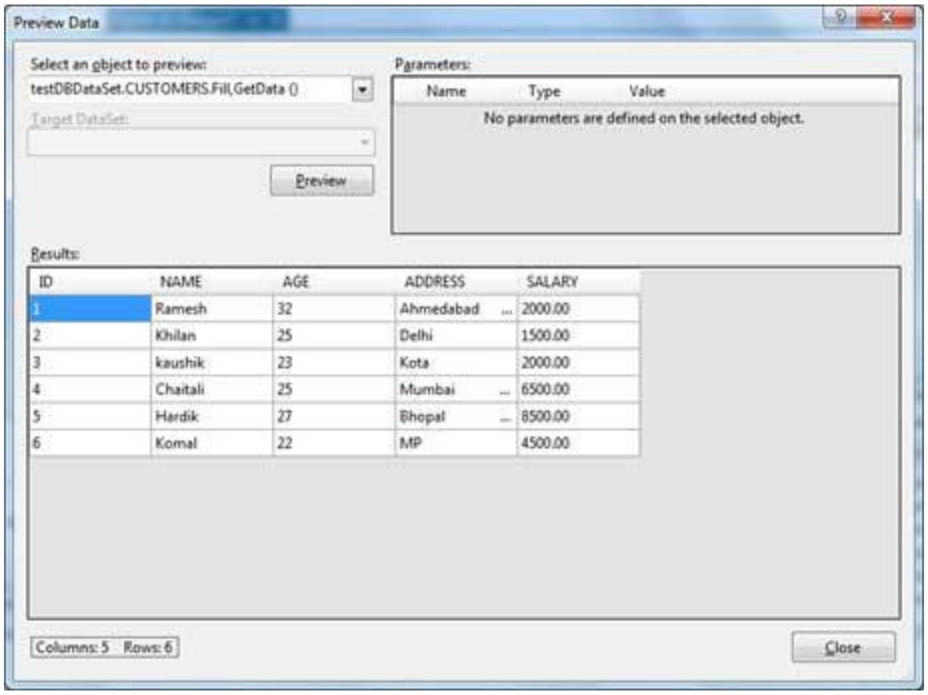
Which database objects do you want in your dataset?

- [✓] Tables
  - [✓] CUSTOMERS
- [ ] Views
- [ ] Stored Procedures
- [ ] Functions

DataSet name: testDBDataSet

< Previous
Next >
Finish
Cancel

- Select the Preview Data link to see the data in the Results grid −



Select an object to preview: testDBDataSet.CUSTOMERS.Fill,GetData ()

Target DataSet:

Parameters:

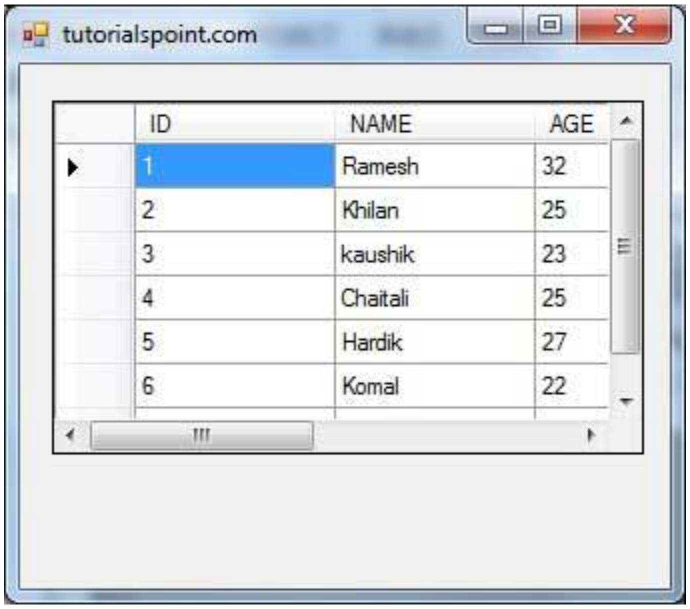
Name	Type	Value
No parameters are defined on the selected object.		

Results:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad ...	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai ...	6500.00
5	Hardik	27	Bhopal ...	8500.00
6	Komal	22	MP	4500.00

Columns: 5 Rows: 6

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



	ID	NAME	AGE
▶	1	Ramesh	32
	2	Khilan	25
	3	kaushik	23
	4	Chaitali	25
	5	Hardik	27
	6	Komal	22

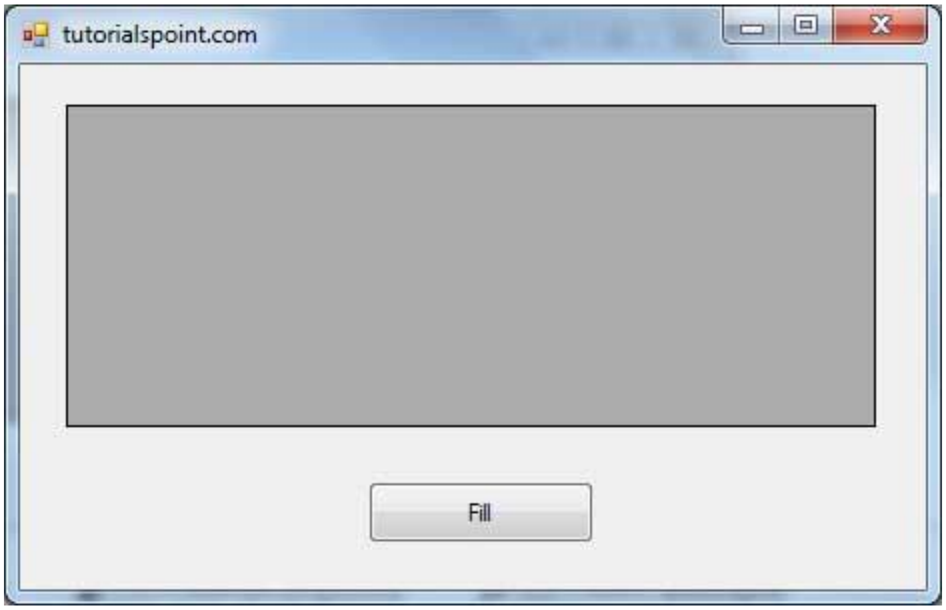
Example 2

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

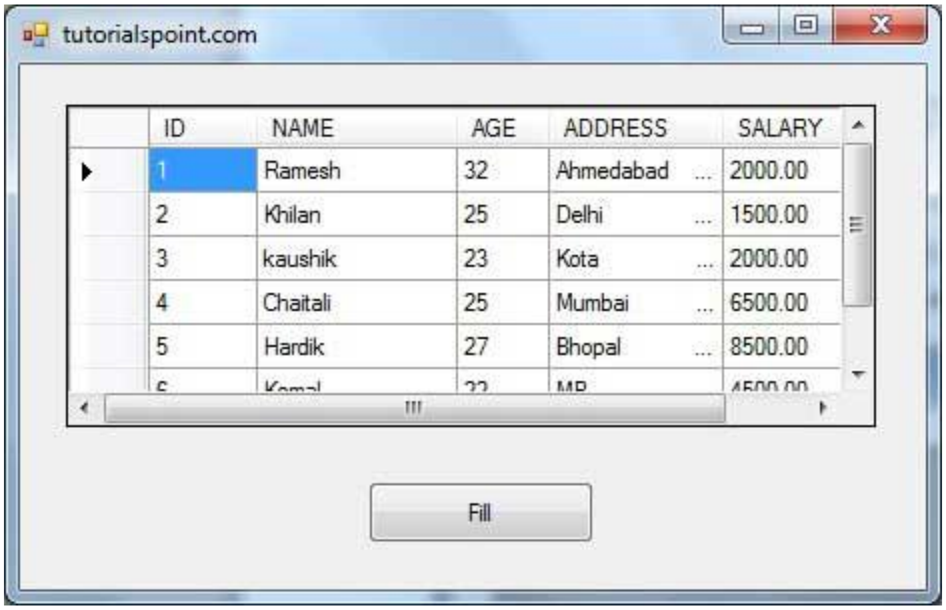
- Add a DataGridView control and a button in the form.
- Change the text of the button control to 'Fill'.
- Double click the button control to add the required code for the Click event of the button, as shown below −

```
ImportsSystem.Data.SqlClient
PublicClassForm1
PrivateSubForm1_Load(sender AsObject, e AsEventArgs) _
HandlesMyBase.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"
EndSub

PrivateSubButton1_Click(sender AsObject, e AsEventArgs)HandlesButton1.Click
Dim connection AsSqlConnection=New sqlconnection()
      connection.ConnectionString="Data Source=KABIR-DESKTOP; _
         Initial Catalog=testDB;Integrated Security=True"
      connection.Open()
Dim adp AsSqlDataAdapter=NewSqlDataAdapter _
("select * from Customers", connection)
Dim ds AsDataSet=NewDataSet()
      adp.Fill(ds)
DataGridView1.DataSource= ds.Tables(0)
EndSub
EndClass
```

- 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 −



ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad ...	2000.00
2	Khilan	25	Delhi ...	1500.00
3	kaushik	23	Kota ...	2000.00
4	Chaitali	25	Mumbai ...	6500.00
5	Hardik	27	Bhopal ...	8500.00

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

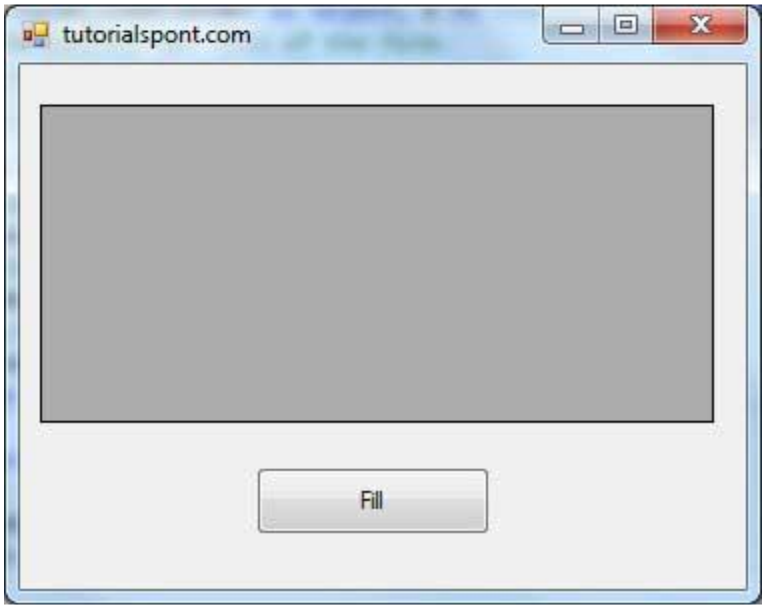
So far, we have used tables and databases already existing in our computer. In this example, we will create a table, add columns, rows and data into it and display the table using a DataGridView object.

Take the following steps −

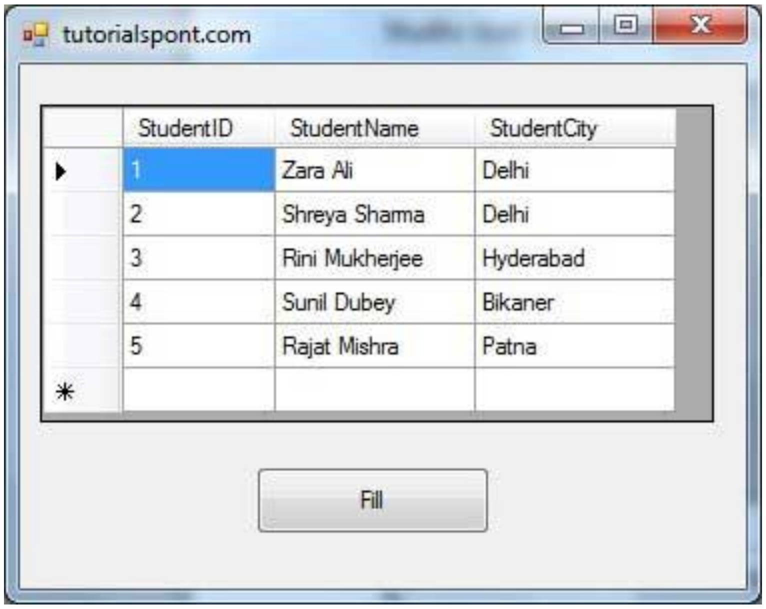
- Add a DataGridView control and a button in the form.
- Change the text of the button control to 'Fill'.
- Add the following code in the code editor.

```
PublicClassForm1
PrivateSubForm1_Load(sender AsObject, e AsEventArgs)HandlesMyBase.Load
' Set the caption bar text of the form.
      Me.Text = "tutorialspont.com"
   End Sub

   Private Function CreateDataSet() As DataSet
      'creating a DataSetobjectfor tables
Dim dataset AsDataSet=NewDataSet()
' 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
PrivateSubAddNewRow(ByRef table AsDataTable,ByRef id AsInteger,_
ByRef name AsString,ByRef city AsString)
Dim newrow AsDataRow= table.NewRow()
      newrow("StudentID")= id
      newrow("StudentName")= name
      newrow("StudentCity")= city
      table.Rows.Add(newrow)
EndSub

PrivateSubButton1_Click(sender AsObject, e AsEventArgs)HandlesButton1.Click
Dim ds AsNewDataSet
      ds =CreateDataSet()
DataGridView1.DataSource= ds.Tables("Student")
EndSub
EndClass
```

- 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 −



	StudentID	StudentName	StudentCity
▶	1	Zara Ali	Delhi
	2	Shreya Sharma	Delhi
	3	Rini Mukherjee	Hyderabad
	4	Sunil Dubey	Bikaner
	5	Rajat Mishra	Patna
*			