CHRISTIAN EMINENT COLLEGE, INDORE
(Academy of Management, Professional Education and Research)
An Autonomous Institution Accredited with 'A' Grade by NAAC

E-Content

On

"Dot Net Programming using C#"

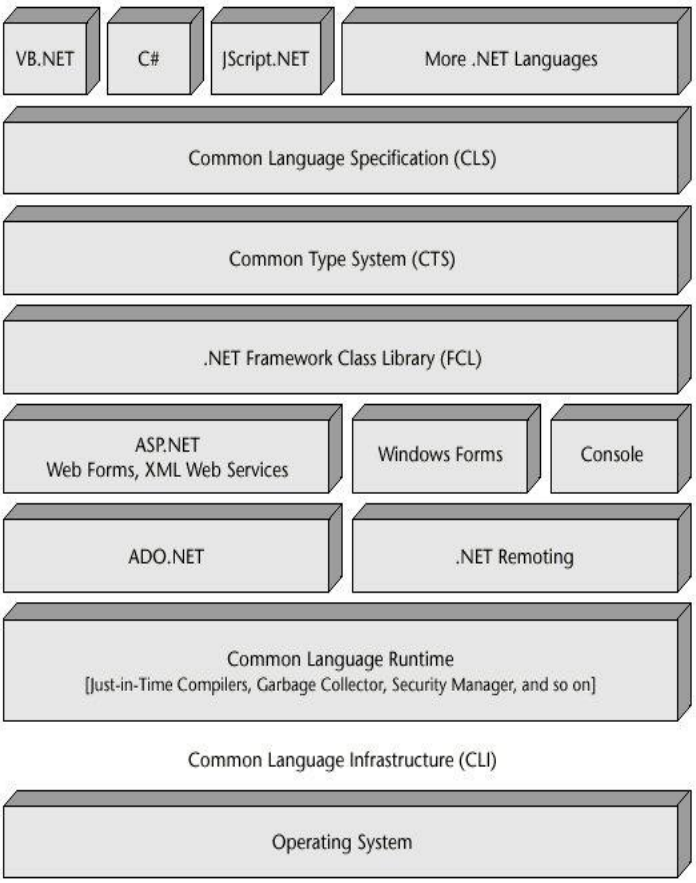
Prepared By: Prof. Himanshu Dehariya

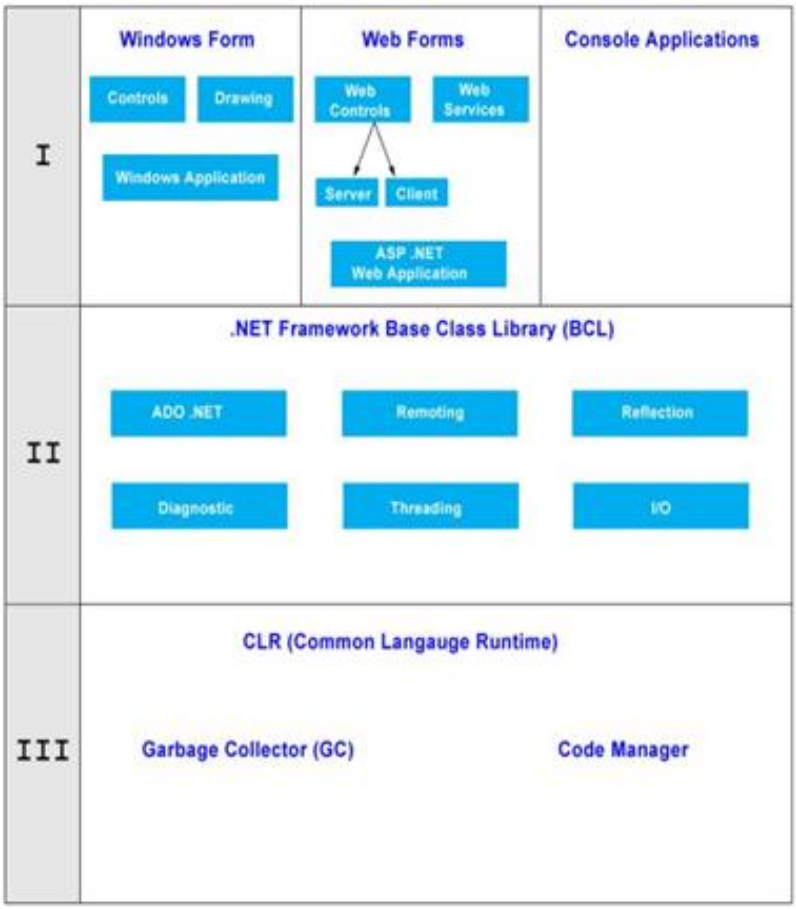
Department of Computer Science & Electronics

.NET Framework Architecture

.NET is tiered, modular, and hierarchal. Each tier of the .NET Framework is a layer of abstraction. .NET languages are the top tier and the most abstracted level. The common language runtime is the bottom tier, this is important since the common language runtime works closely with the operating environment to manage .NET applications. The .NET Framework is partitioned into modules, each with its own distinct responsibility. Finally, since higher tiers request services only from the lower tiers, .NET is hierarchal.

Definition: A programming infrastructure created by Microsoft for building, deploying, and running applications and services that use .NET technologies, such as desktop applications, Web application , console application,Web services etc.





.NET Framewotk consists of mainly two parts-

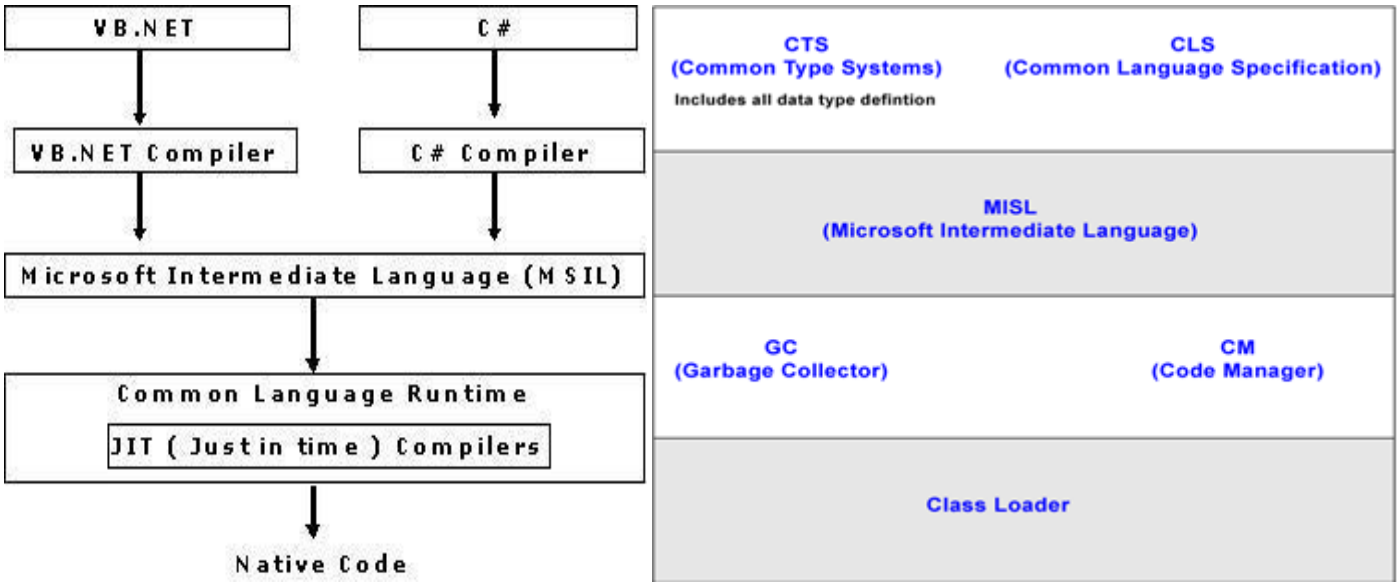
Common Language Runtime (CLR): The execution environment for .NET applications.
Framework Class Library (FCL): The base classes for .Net framework.

Common Language Runtime (CLR)

.Net Framework provides runtime environment called Common Language Runtime (CLR). The code which runs under the CLR is called as **Managed Code**. Programmers need not to worry on managing the memory if the programs are running under the CLR as it provides memory management and thread management. When our program needs memory, CLR allocates the memory for scope and de-allocates the memory if the scope is completed.Language Compilers (e.g. C#, VB.Net, J#) will convert the Code/Program to **Microsoft Intermediate Language** (MSIL) intern this will be converted to **Native Code** by CLR.

Managed code is managed by the common language runtime. CLR manages security, code verification, type verification, exception handling, garbage collection, a common runtime, and other important elements of program execution. When an assembly is executed, **mscoree.dll** is loaded into the memory of the running process. mscoree.dll contains the common language runtime, which then manages the executing application

The CLR interfaces between the application & the operating system. When a .Net application is run, CLR is responsible for loading the code and setting up the environment with the required resources. It also provides number of other services to .Net applications.



Common Type System (CTS)

It describes set of data types that can be used in different .Net languages in common. (i.e), CTS ensures that objects written in different .Net languages can interact with each other.
The Common Type System (CTS) is a catalog of .NET types.System.Int32, System.Decimal, System.Boolean, and so on. Developers are not required to use these types directly. These types are the underlying objects of the specific data types provided in each managed language. For Example C# has **int** Data Type and VB.Net has **Integer** Data Type. Hence a variable declared as int in C# or Integer in vb.net, finally after compilation, uses the same structure **Int32** from CTS.

C Sharp Integer	Vb.Net Integer
int a;	Dim a as integer

Preferably, you should use the syntax of the language and not the underlying object type, leaving .NET the flexibility to select the most appropriate type and size for the operating environment. The common type system supports two general categories of types:

Value types: Value types directly contain their data, and instances of value types are either allocated on the stack Value types can be built-in (implemented by the runtime), user-defined, or enumerations.

Reference types: Reference types store a reference to the value's memory address, and are allocated on the heap. Reference types can be self-describing types, pointer types, or interface types.

Common Language Specification (CLS)

It is a sub set of CTS and it specifies a set of rules that needs to be adhered or satisfied by all language compilers targeting CLR. It helps in cross language inheritance and cross language debugging.

CLS stands for Common Language Specification and it is a subset of CTS. It defines a set of rules and restrictions that every language must follow which runs under .NET framework. The languages which follow these set of rules are said to be CLS Compliant. In simple words, CLS enables cross-language integration.

For example, one rule is that you cannot use multiple inheritance within .NET Framework. As you know C++ supports multiple inheritance but; when you will try to use that C++ code within C#, it is not possible because C# doesn't support multiple inheritance.

One another rule is that you cannot have members with same name with case difference only i.e. you cannot have add() and Add() methods. This easily works in C# because it is case-sensitive but when you will try to use that C# code in VB.NET, it is not possible because VB.NET is not case-sensitive.

Cross Language integration

You can use a utility of a language in another language (It uses Class Language Integration). It includes no restriction on the type of applications that are possible. The .NET Framework allows the creation of Windows applications, Web applications, Web services, and lot more. The .NET Framework has been designed so that it can be used from any language, including C#, C++, Visual Basic, JScript, and even older languages such as COBOL.

Framework Class Liabrary (FCL)

This is also called as **Base Class Library (BCL)** and it is common for all types of applications i.e. the way you access the Library Classes and Methods in VB.NET will be the same in C#, and it is common for all other languages in .NET. The following are different types of applications that can make use of .net class library.

1. Windows Application- It is desktop application.
2. Console Application- an application that takes input and displays output at a command line console.
3. Web Application- All websites are example of web application. They use a web server.
4. Web Services- It doesn't use web-based server. Internet payment systems are example of web services.

In short, developers just need to import the BCL in their language code and use its predefined methods and properties to implement common and complex functions like reading and writing to file, graphic rendering, database interaction, and XML document manipulation.

FCL includes some 600 managed classes. Aflat hierarchy consisting of hundreds of classes would be difficult to navigate. Microsoft partitioned the managed classes of FCL into separate namespaces based on functionality. For example, classes pertaining to local input/output can be found in the namespace **System.IO**. To further refine the hierarchy, FCL namespaces are often nested; the tiers of namespaces are delimited with dots. **System.Runtime.InteropServices**, **System.Security.Permissions**, and **System.Windows.Forms** are examples of nested namespaces. The root namespace is System, which provides classes for console input/output, management of application domains, delegates, garbage collection, and more.

DLL Hell

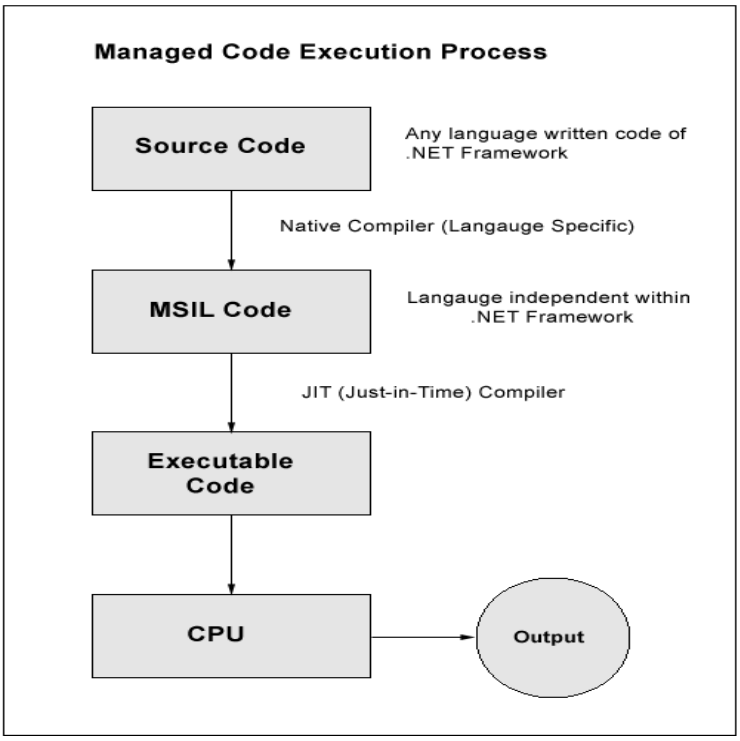
It refers to the set of problems caused when multiple applications attempt to share a common component like a dynamic link library (DLL) or a Component Object Model (COM) class. The reason for this issue was that the version information about the different components of an application was not recorded by the system. (Windows Registry cannot support the multiple versions of same COM component this is called the dll hell problem.)

.NET supports two kind of code

1) Managed Code 2) Unmanaged Code

Managed Code

The resource, which is with in your application domain is, managed code. The resources that are within domain are faster. The code, which is developed in .NET framework, is known as managed code. This code is directly executed by CLR with help of managed code execution. Any language that is written in .NET Framework is managed code. Managed code uses CLR which in turns looks after your applications by managing memory, handling security, allowing cross - language debugging, and so on.

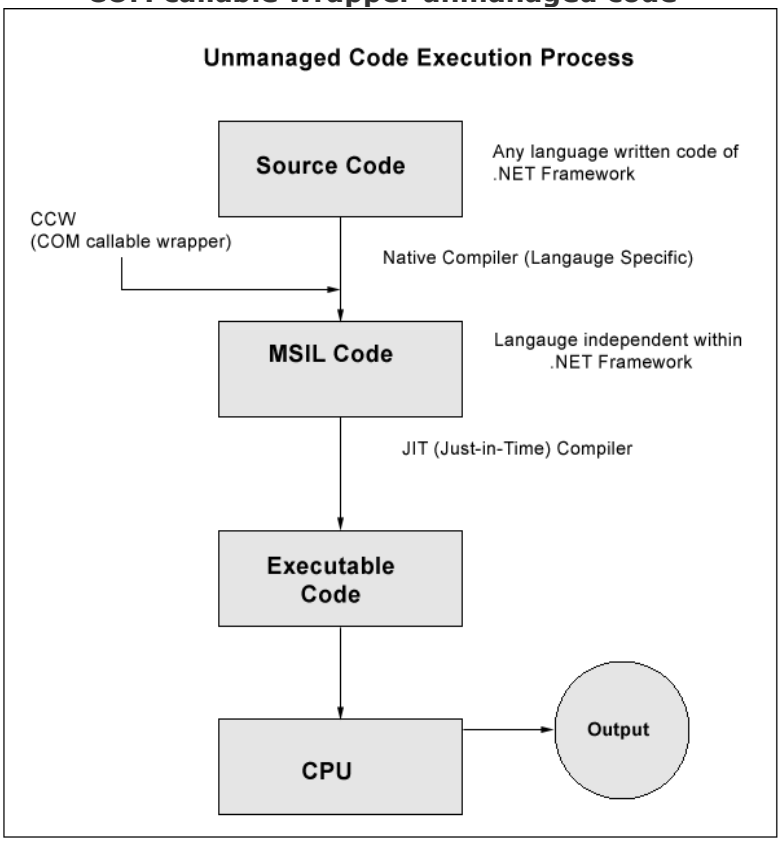


Unmanaged Code

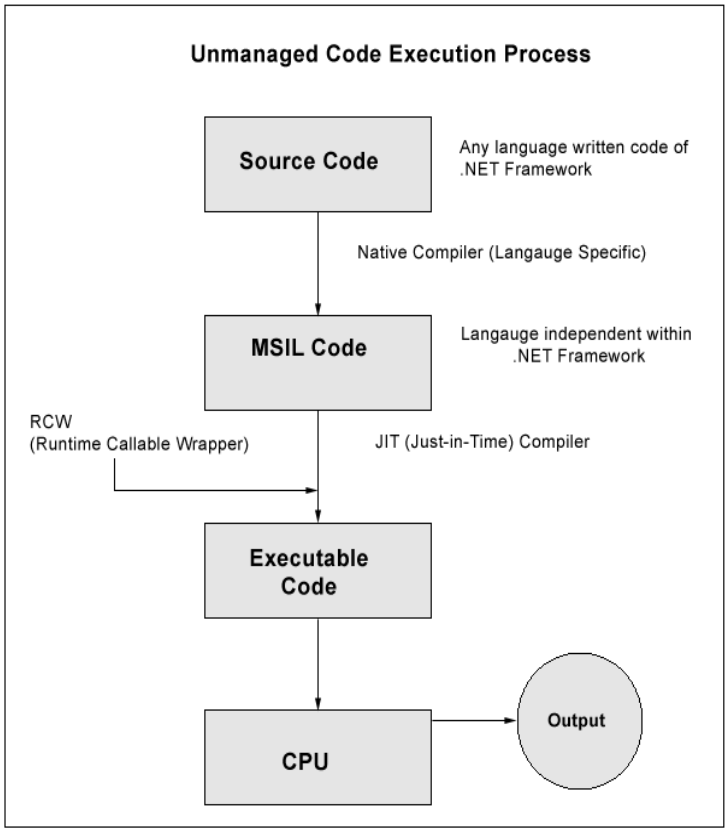
The code, which is developed outside .NET, Framework is known as unmanaged code. Applications that do not run under the control of the CLR are said to be unmanaged, and certain languages such as C++ can be used to write such applications, which, for example, access low - level functions of the operating system. Background compatibility with code of VB, ASP and COM are examples of unmanaged code.

Unmanaged code can be unmanaged source code and unmanaged compile code. Unmanaged code is executed with help of wrapper classes. Wrapper classes are of two types: CCW (COM callable wrapper) and RCW (Runtime Callable Wrapper). Wrapper is used to cover difference with the help of CCW and RCW.

COM callable wrapper unmanaged code



Runtime Callable Wrapper unmanaged code



Native Code

The code to be executed must be converted into a language that the target operating system understands, known as native code. This conversion is called **compiling** code, an act that is performed by a compiler. Under the .NET Framework, however, this is a two - stage process. With help of **MSIL** and **JIT**.

MSIL (Microsoft Intermediate Language)

It is language independent code. When you compile code that uses the .NET Framework library, you don't immediately create operating system - specific native code. Instead, you compile your code into Microsoft Intermediate Language (MSIL) code. The MSIL code is not specific to any operating system or to any language.

JIT (Just-in-Time)

Just - in - Time (JIT) compiler, this compiles MSIL into native code that is specific to the OS and machine architecture being targeted. Only at this point can the OS execute the application. The just - in - time part of the name reflects the fact that MSIL code is only compiled as, and when, it is needed.

In the past, it was often necessary to compile your code into several applications, each of which targeted a specific operating system and CPU architecture. Often, this was a form of optimization. This is now unnecessary, because a JIT compiler uses MSIL code, which is independent of the machine, operating system, and CPU. Several JIT compilers exist, each targeting a different architecture, and the appropriate one will be used to create the native code required. JIT are of three types:

1. **Pre JIT-** It converts all the code in executable code and it is slow
2. **Econo JIT-** It will convert the called executable code only, but when a code is called again, evey time.

3. **Normal JIT**- It will only convert the called code and will store in cache so that it will not require converting code again. Normal JIT is fast.

Assemblies

When you compile an application, the MSIL code created is stored in an assembly. Assemblies include both executable application files that you can run directly from Windows without the need for any other programs (these have an .exe file extension), and libraries (which have a .dll extension) for use by other applications. In addition to containing MSIL, assemblies also include meta information (information about the information contained in the assembly, also known as metadata) and optional resources (additional data used by the MSIL, such as sound files and pictures).

The meta information enables assemblies to be fully self - descriptive. You need no other information to use an assembly, meaning you avoid situations such as failing to add required data to the system registry and so on, which was often a problem when developing with other platforms. This means that deploying applications is often as simple as copying the files into a directory on a remote computer. Because no additional information is required on the target systems, you can just run an executable file from this directory and (assuming the .NET CLR is installed) you're good to go.

Of course, you won't necessarily want to include everything required to run an application in one place. You might write some code that performs tasks required by multiple applications. In situations like that, it is often useful to place the reusable code in a place accessible to all applications. In the .NET Framework, this is the Global Assembly Cache (GAC). Placing code in the GAC is simple - you just place the assembly containing the code in the directory containing this cache. We can create two types of Assembly:

- ✓ **Private Assembly:** A private Assembly is used only by a single application, and usually it is stored in that application's install directory.
- ✓ **Shared Assembly:** A shared Assembly is one that can be referenced by more than one application. If multiple applications need to access an Assembly, we should add the Assembly to the Global Assembly Cache (GAC).

Garbage Collection (GC)

One of the most important features of managed code is the concept of garbage collection. This is the .NET method of making sure that the memory used by an application is freed up completely when the application is no longer in use. Prior to .NET this was mostly the responsibility of programmers, and a few simple errors in code could result in large blocks of memory mysteriously disappearing as a result of being allocated to the wrong place in memory. That usually meant a progressive slowdown of your computer followed by a system crash.

.NET garbage collection works by inspecting the memory of your computer every so often and removing anything from it that is no longer needed. There is no set time frame for this; it might happen thousands of times a second, once every few seconds, or whenever, but you can rest assured that it will happen.

Namespace

Namespaces are C# program elements designed to help you organize your programs. They also provide assistance in avoiding name clashes between two sets of code. A namespace is designed for providing a way to keep one set of names separate from another. The class names declared in one namespace does not conflict with the same class names declared in another. Namespaces are the way to organize .NET Framework Class Library into a logical grouping according to their functionality, usability as well as category they should belong to, or we can say Namespaces are logical grouping of types for the purpose of identification.

The .NET Framework Class Library (FCL) is a large collection of thousands of Classes. These Classes are organized in a hierarchical tree. The System Namespaces is the root for types in the .NET Framework. We can uniquely identify any Class in the .NET Framework Class Library (FCL) by using the full Namespaces of the class .In .Net languages every program is created with a default Namespaces. Programmers can also create their own Namespaces in .Net languages.

Advantages:

- We can establish security, version, reference, and deployment boundaries by using namespaces
- Because of grouping of namespaces we can create hierarchy which is easy to identify classes by fully qualified names.
- Namespace is logical division of class, structure and interface
- Namespaces are a way of grouping type names and reducing the chance of name collisions.
- The namespace with all the built-in functionality comes under **System** namespace. All other namespaces comes under this **System** namespace.

Difference between Namespace and Assembly

A namespace is a logical grouping of types. An assembly can contain types in multiple namespaces and a single namespace can be spread across assemblies.

Event Driven Programming

In computer programming, event-driven programming is a programming paradigm in which the flow of the program is determined by events such as user actions (mouse clicks, key presses), sensor outputs, or messages from other programs/threads. Event-driven programming is the dominant paradigm used in graphical user interfaces and other applications (e.g. JavaScript web applications) that are centered on performing certain actions in response to user input. The common events are Click, DblClick, Load, MouseMove, MouseDown, MouseUp, KeyPress, KeyUp, KeyDown, GotFocus, LostFocus, etc.

When you fire an event, the code in the event procedure is executed, and then visual basic performs its operations as per the instructions written in the event procedure code. For example, in the first sample program, when you click the 'Print' button, the click event is fired, and then the code in the click event procedure gets executed. The code tells Visual Basic to print a text on the form. So as a result, you see a text printed on the form.

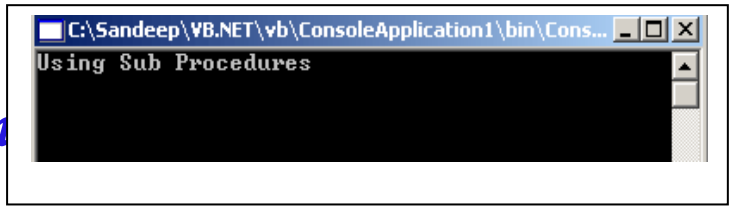
Methods

A Method is a procedure built into the class. They are a series of statements that are executed when called. Methods allow us to handle code in a simple and organized fashion. There are two types of methods in VB .NET: those that return a value (Functions) and those that do not return a value (Sub Procedures).

 # Sub Procedures

Sub procedures are methods which do not return a value. Each time when the Sub procedure is called the statements within it are executed until the matching End Sub is encountered. Sub Main(), the starting point of the program itself is a sub procedure. When the application starts execution, control is transferred to Main Sub procedure automatically which is called by default.

```
Module Module1

Sub Display()
  System.Console.WriteLine("Using Sub Procedures")
End Sub

Sub Main()
```

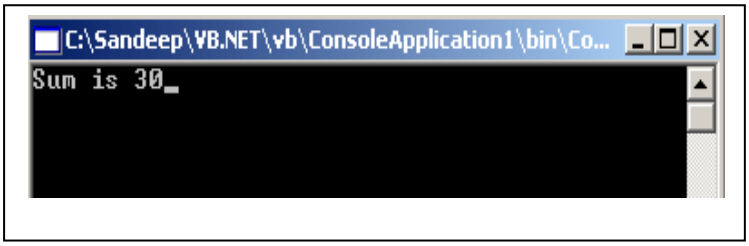


```
    Display()
End Sub

End Module
```

 # Functions

Function is a method which returns a value. Functions are used to evaluate data, make calculations or to transform data. Declaring a Function is similar to declaring a Sub procedure. Functions are declared with the Function keyword. The following code is an example on Functions:

```
Module Module1

Public Function Add() As Integer
 Dim i, j As Integer
 i = 10
 j = 20
 Return (i + j)
End Function

Sub Main()
 Write("Sum is" & " " & Add())
End Sub

End Module
```



 # Calling Methods

A method is not executed until it is called. A method is called by referencing it's name along with any required parameters. For example, the above code called the Add method in Sub main like this:

```
Write("Sum is" & " " & Add()).
```

 # Method Variables

Variables declared within methods are called method variables. They have method scope which means that once the method is executed they are destroyed and their memory is reclaimed. For example, from the above code (Functions) the Add method declared two integer variables i, j. Those two variables are accessible only within the method and not from outside the method.

#Parameters

A parameter is an argument that is passed to the method by the method that calls it. Parameters are enclosed in parentheses after the method name in the method declaration. You must specify types for these parameters. The general form of a method with parameters looks like this:

```
Public Function Add (ByVal x1 as Integer, ByVal y1 as Integer)
------------
      <Function Body>
------------
End Function
```

Events

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

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

Event handlers are functions that tell how to respond to an event. VB.Net is an event-driven language. There are mainly two types of events:

- Mouse events
- Keyboard events

Handling Mouse Events

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

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

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

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

Handling Keyboard Events

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

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

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

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

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

- **Handled** - indicates if the KeyPress event is handled

- **KeyChar** - stores the character corresponding to the key pressed

---

Dot Net as Better

1. In .NET you have a choice of languages to code with (C#, VB.NET, Java, Boo, Python e.t.c), producing the same type of compiled code but in Java one is limited to the java language.

2. NET prgrams run at native speed while java is interpreted which makes java slower.Although java has Just In Time compilation but it stills run slower.
3. Calling native code in java is not a very clean process. You need to generate some stub files which makes the whole process cumbersome and dirty. In .NET you just declare that you are calling a native function from a specified library and just start calling it.
4. Dot NET languages are richer than Java. They have object oriented feature that are absent in java e.g properties, delegates, generics.

5. Java GUI programs look alien on the host operating system. Even if you use the OS's theme you still notice that the java widgets look out of place.
6. Dot NET in the form of Mono has brought a whole revolution on the linux desktop in form of great applications like beagle, tomboy, diva, iFolder, banshee e.t.c. This is something that java has failed to do despite the fact that it's been there long before .NET
7. Many programs that would have been difficult to develop with java have been developed with .NET things like compilers (Mono's C# and VB.NET) 3D game engines (unity game engine) e.t.c
8. The CLI is an open standard maintained by an independent standards organization (E.C.M.A) while java is still governed by SUN microsystems.Even though java has recently been open-sourced, it's future will still be highly influenced by SUN.
9. You can code on the .NET platform using Java but you cannot code on Java platform using any of the .NET languages.

 # Declaring Variables

Variables are named storage areas inside computer memory where a program places data during processing. A program sets aside these storage areas by declaring variables, that is, by assigning them a name and indicating what types of data will occupy these areas during processing. Within a Visual Basic program, variables are declared using the Dim statement as shown below.

Dim <*variable*> as <*type*>

Where variable is a programmer-supplied name for the storage area and type is one of the data types in VB.

 # Rules for Naming Variables

Programmer-supplied variable names are also referred to as identifiers, and they must conform to Visual Basic naming conventions. A variable name must

- Begin with an alphabetic or numeric character, or the underscore (_) character.
- Be composed only of alphabetic, numeric, and underscore characters.
- Not contain embedded blank spaces.
- Not be the same as a Visual Basic keyword, words the comprise the Visual Basic language itself.

Ex: X, My_Variable, MyVariable, My_Data_Variable,

#Data Types

When declaring a variable an indication can be given about what type of data will be stored. There are many data types permissible under Visual Basic. The following are common types of data involved in computer processing.

Short: A nondecimal number with a value in the range of -32,768 to +32,767.

Integer: A nondecimal number with a value in the range of -2,147,483,648 to +2,147,483,647.

Long: A nondecimal number with value in range of -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807.

Single: A floating-point (decimal) positive or negative number with a value in the range of up to 3.402823E38

Double: A floating-point positive or negative number with a value in the range of up to 1.79769313486232E308.

Decimal: A decimal number with a value in the range of up to +79,228,162,514,264,337,593,950,335. without decimal precision and up to +7.9228162514264337593543950335 with 28 places of decimal precision.

String: Any number of alphabetic, numerical, and special characters.

Date: Dates ranges from January 1, 0001 to December 31, 9999 & times ranges from 0:00:00 to 23:59:59.

Boolean: The value True or False.

Ex: Dim My_Counter as Integer, Dim My_Salary as Decimal, Dim My_Name as String, Dim DOB As Date

 # Assigning Values to Variables

Variable declarations simply set aside memory storage areas as temporary containers of data, temporary because they are no longer accessible once the program ends. During processing, of course, these variables are occupied by data values. In any case, variables take on values by assigning values to them. Variables are assigned values with the Visual Basic assignment statement whose general format is shown below.

variable = value

The data value on the right of the equal sign is assigned to the variable identified on the left of the equal sign.

 # Assigning Constants to Variables

A literal data value a particular number, string, date, or Boolean value can be assigned to the variable. A number is assigned by specifying its value to the right of the equal sign:

Ex: My_Integer = 10, My_FloatingPoint = 12.34567, My_Decimal = 123.45

 # The Variant Data Type

If variable types are not explicitly declared - using As Integer, As Decimal, As String, etc. then values assigned to the variable are stored as a Variant data type. Any type of data can be stored in the variable and any type of operation can be performed on it. It is a general-purpose data type that provides flexibility in storing different types of data in a single variable. It can be assigned a numeric value that is immediately replaced by a string value.

Although it does offer some programming flexibility, there are minor inefficiencies associated with converting different data types for representation in the same storage area. For this reason it is best always to declare specific data types for specific variables. This practice of using strong typing of variables leads to more efficient

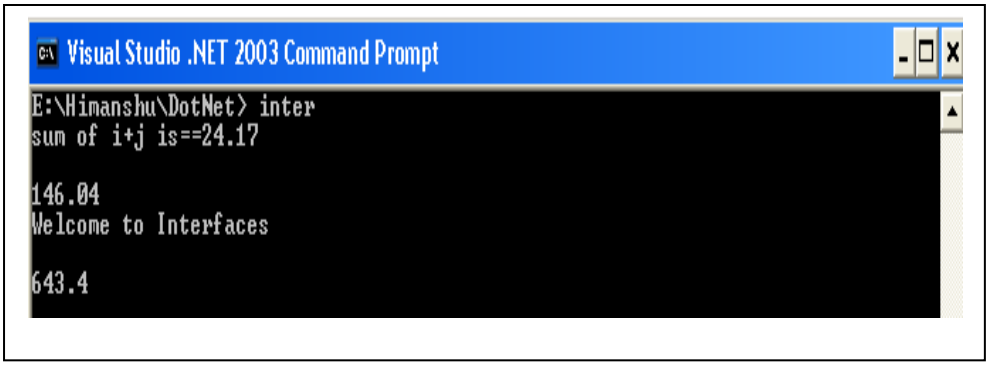
programming and reduction in errors from inadvertently storing the wrong kinds of data in the wrong variables.

Interface

Interfaces allow us to create definitions for component interaction. They also provide another way of implementing polymorphism. Through interfaces, we specify methods that a component must implement without actually specifying how the method is implemented. We just specify the methods in an interface and leave it to the class to implement those methods. Visual Basic .NET does not support multiple inheritance directly but using interfaces we can achieve multiple inheritance. We use the Interface keyword to create an interface and implements keyword to implement the interface. Once you create an interface you need to implement all the methods specified in that interface. The following code demonstrates the use of interface.

Example:

```
Imports System.Console
Module Module1

Public Interface Test
  Sub disp()
  Function Multiply() As Double
End Interface


Public Class One
  Implements Test

    Public i As Double = 12
    Public j As Double = 12.17

    Sub disp() Implements Test.disp
      WriteLine("sum of i+j is==" & i + j)
      Read()
    End Sub


   Public Function multiply() As Double Implements Test.Multiply
      WriteLine(i * j)
      Read()
   End Function
End Class

Public Class Two
  Implements Test

   Public a As Double = 20
   Public b As Double = 32.17

   Sub disp() Implements Test.disp
     WriteLine("Welcome to Interfaces")
```

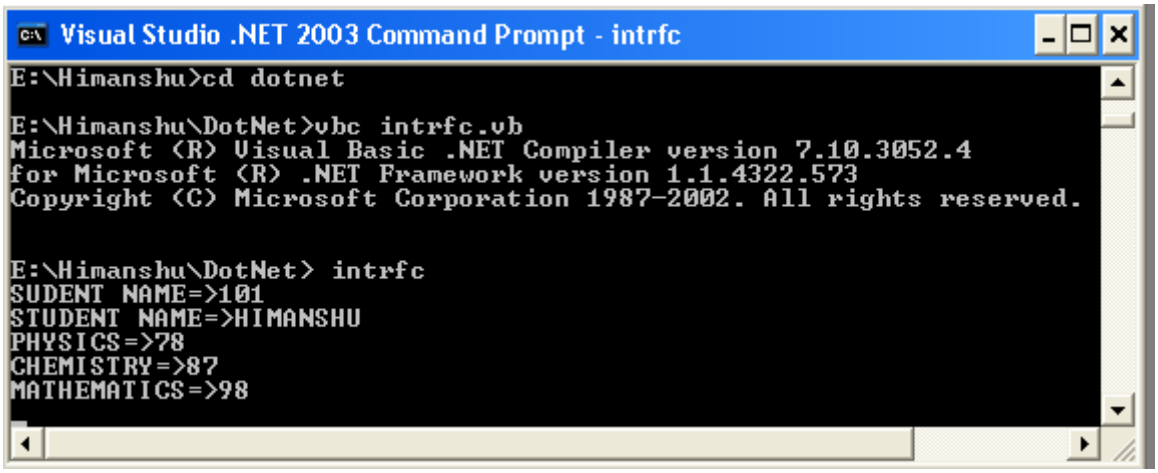


```
      Read()
    End Sub

    Public Function multiply() As Double Implements Test.Multiply
      WriteLine(a * b)
      Read()
  End Function
End Class

Sub Main()
    Dim OneObj As New One()
    Dim TwoObj As New Two()
    OneObj.disp()
    OneObj.multiply()
    TwoObj.disp()
    TwoObj.multiply()
End Sub

End Module
```

Example:

```
Imports System
Module Module1
Public Interface inter
        Sub GetData(ByVal rno As Integer, ByVal sname As String)
        Sub Show()
End Interface

Class Record
   Dim p, c, m As Integer
   Public Sub Getdata1(ByVal phy As Integer, ByVal chm As Integer, ByVal Mat As Integer)
      p = phy
      c = chm
      m = Mat
   End Sub
   Public Sub show1()
      Console.WriteLine("PHYSICS=>" & p)
      Console.WriteLine("CHEMISTRY=>" & c)
      Console.WriteLine("MATHEMATICS=>" & m)

   End Sub
End Class

Class Print
     Inherits Record
     Implements inter

    Public r As Integer
    Public nm As String
    Public Sub GetData(ByVal rno As Integer, ByVal sname As String) Implements inter.GetData
       r = rno
       nm = sname
    End Sub
    Public Sub Show() Implements inter.Show
```

```
        Console.WriteLine("SUDENT NAME=>" & r)
        Console.WriteLine("STUDENT NAME=>" & nm)
    End Sub
End Class

Sub main()
     Dim p As New Print
     p.GetData(101, "HIMANSHU")
     p.Getdata1(78, 87, 98)
     p.Show()
     p.show1()
     Console.ReadLine()
End Sub

End Module
```



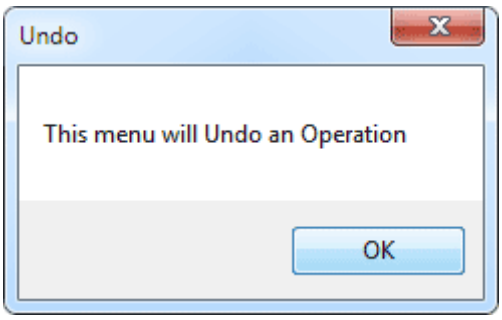
Message Box

Message Box is one of the built-in dialog boxes that help you to provide a rich user interface in your front-end applications. MessageBox.Show displays a dialog box. It interrupts the user. It immediately blocks further interaction. We specify buttons, default buttons, an icon, and some other properties of the dialog.

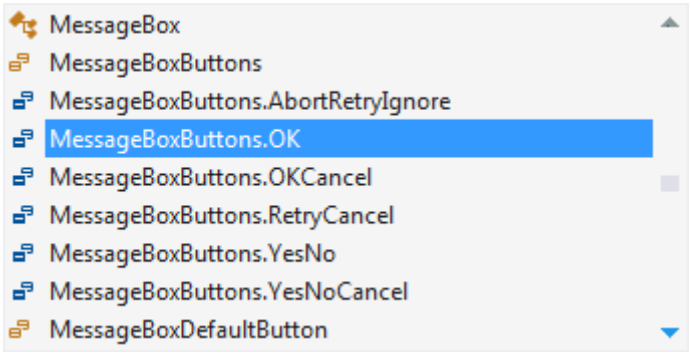
So if your message box function was this:

MessageBox.Show ("This menu will Undo an Operation", "Undo")

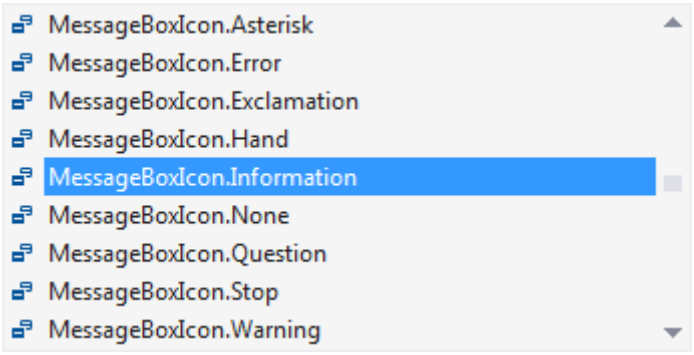
You would get this message box popping up:



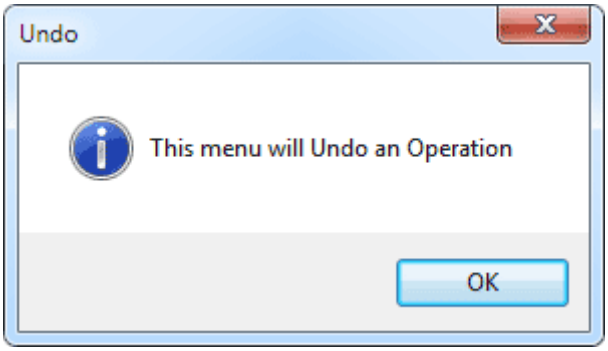
Each option for your message box is separated by a comma. If you type a comma after the "Undo", and then a space, you'll get another pop-up menu. On this menu, you can specify which buttons you want on your message box:



You only need the OK button on your message boxes. Double click this item, then type another comma, and hit the spacebar. Yet another pop-up menu will appear. On this menu, you can specify the symbol that appears in the message box:



It's up to you which symbol you choose. Experiment with all of them and see what they look like. In the image below, we've gone for the Information symbol:



Input Box

When you need to prompt the user to enter an expected value you use Input Box. It displays a prompt in a dialog box, waits for the user to enter value and finally returns a string containing the entry.

An input box is a specially designed dialog box that allows the programmer to request a value from the user and use that value as necessary. An input box displays a title, a message to indicate the requested value, a text box for the user, and two buttons: OK and Cancel. Here is an example:

When an input box displays, it presents a request to the user who can then provide a value. After using the input box, the user can change his or her mind and press Esc or click Cancel. If the user provided a value and want to acknowledge it, he or she can click OK or press Enter. This would transfer the contents of the text box to the application that displayed the input box.

```
Dim strUserName as String
strUserName = InputBox("Enter your name:", "InputBox Test", "Type your name here.")
```

Syntax:

String variable = InputBox(*prompt*[, *title*] [, *default*] [, *xpos*] [, *ypos*])

The arguments to the InputBox function are described below:

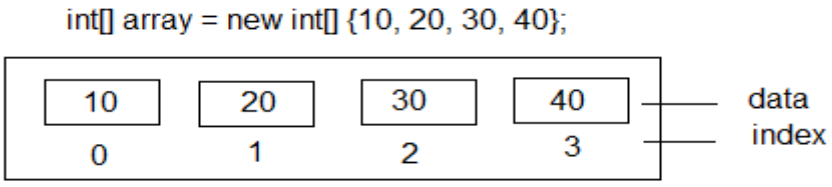
Argument	**Description**
prompt	Required. String expression displayed as the message in the dialog box. The maximum length of ***prompt*** is approximately 1024 characters, depending on the width of the characters used.
title	Optional. String expression displayed in the title bar of the dialog box. If you omit ***title***, the application name is placed in the title bar.

default	Optional. String expression displayed in the text box as the default response if no other input is provided. If you omit ***default***, the text box is displayed empty.
xpos and ***ypos***	Both optional. Numeric expressions that specify custom positioning of the box on screen (by default, the box is displayed in the center of the screen, which is usually desired).

If the user clicks OK or presses ENTER, the InputBox function returns whatever is in the text box. If the user clicks Cancel, the function returns a zero-length string ("").

Array

Arrays are using for store similar data types grouping as a single unit. It is a fixed collection of same data type that are stored contiguously and that are accessible by an index. We specify their length and we can initialize arrays with data. We can access Array elements by its numeric index.



Integer Array: Declaring and Initializing an Integer Array

Dim ar (10) As Integer
array (0) = 10
array (1) = 20
array (2) = 30

In the above code we declare an Integer Array of four elements and assign the value to array index. That means we assign values to array index 0-4.We can retrieve these values from array by using a for loop.

Example

```
Imports System
Module module1
  Sub Main()

    Dim ar (10) As Integer
    Dim i as Integer
    For i = 1 To 10
      Console.WriteLine ("Value=" & i)
```

```
    Next
    Console.Read ()

  End Sub
End Module
```

String Array: Declaring and Initializing a String Array

Dim week (6) As String
week (0) = "Sunday"
week (1) = "Monday"

Example

```
Imports System
Module module1

  Sub Main()
    Dim ar () As String = {"dog", "cat", "fish"}

    For Each value As String In ar
      Console.WriteLine (value)
    Next
    Console.Read()
  End Sub
End Module
```

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.

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.
- **subscript** specifies the new dimension.

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 MyArray (10, 20) As String

Or, a 3-dimensional array of Integer variables:

Dim MDArray (10, 10, 10) As Integer

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

```
Module Module1

 Sub Main()
    Dim MyArray(3, 3) As Integer
    Dim i, j As Integer
    Console.WriteLine("Enter Value")
    For i = 0 To 2
       For j = 0 To 2
         MyArray (i, j) = Int32.Parse(Console.ReadLine)
       Next j
    Next i

    For i = 0 To 2
       For j = 0 To 2
          Console.Write(MyArray (i, j))
       Next j
       Console.Write(Environment.NewLine)
    Next i
    Console.ReadKey()
 End Sub

End Module
```

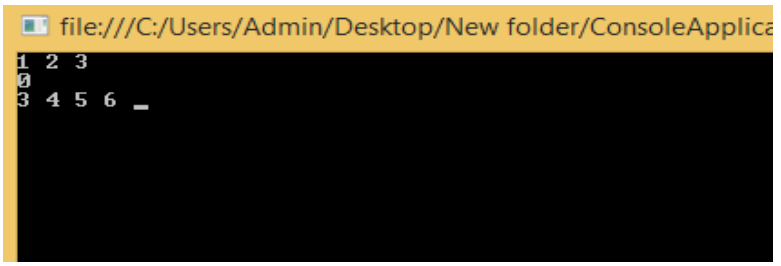
Jagged Array

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)(){}
```

This kind of array is uneven in shape. If the sub arrays you need vary in length, a jagged array becomes extremely efficient. Jagged arrays can use less memory and be faster than two-dimensional arrays. If the shape of your data is uneven, they can save a lot of memory.

Following Program demonstrate the use of jagged array. This program uses the ()() syntax after the jagged array identifier to signify that the array contains other arrays. Temporary arrays are created in the standard way in VB.NET and then assigned to elements in the jagged array. Two nested For-loops are used to loop through the top-level array and all the sub arrays.




```
Module Module1
   Sub Main()
```

```
    ' Create jagged array with maximum index of 2.
    Dim jagged()() As Integer = New Integer(2)() {}
    ' Create temporary array and place in index 0.
    Dim temp(2) As Integer
    temp(0) = 1
    temp(1) = 2
    temp(2) = 3
    jagged(0) = temp
    ' Create small temporary array and place in index 1.
    Dim temp2(0) As Integer
    jagged(1) = temp2
    ' Use array constructor and place result in index 2.
    jagged(2) = New Integer() {3, 4, 5, 6}
    ' Loop through top-level arrays.
    For i As Integer = 0 To jagged.Length - 1
      ' Loop through elements in subarrays.
      Dim inner As Integer() = jagged(i)
      For a As Integer = 0 To inner.Length - 1
        Console.Write(inner(a))
        Console.Write(" "c)
      Next
      Console.ReadLine()
    Next
  End Sub
End Module
```

The Array Class

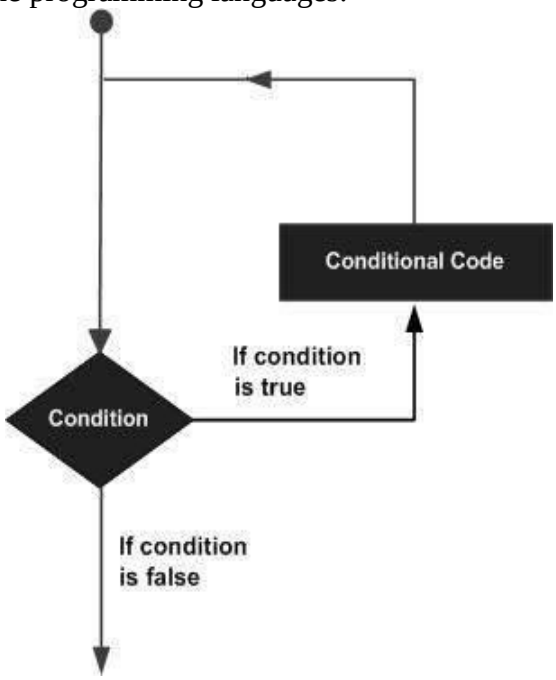
The Array class is the base class for all the arrays in VB.Net. It is defined in the System namespace. The Array class provides various properties and methods to work with arrays.

```
Module Module1

  Sub Main()
    Dim list As Integer() = {34, 72, 13, 44, 25, 30, 10}
    Dim temp As Integer() = list
    Dim i As Integer
    Console.Write("Original Array: ")
    For Each i In list
      Console.Write("{0} ", i)
    Next i
    Console.WriteLine()
    Array.Reverse(temp)
    Console.Write("Reversed Array: ")
    For Each i In temp
      Console.Write("{0} ", i)
    Next i
```

```
    Console.WriteLine()
    Array.Sort(list)
    Console.Write("Sorted Array: ")
    For Each i In list
      Console.Write("{0} ", i)
    Next i
    Console.WriteLine()
    Console.ReadKey()
  End Sub
End Module
```

Loop Statements

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.
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.

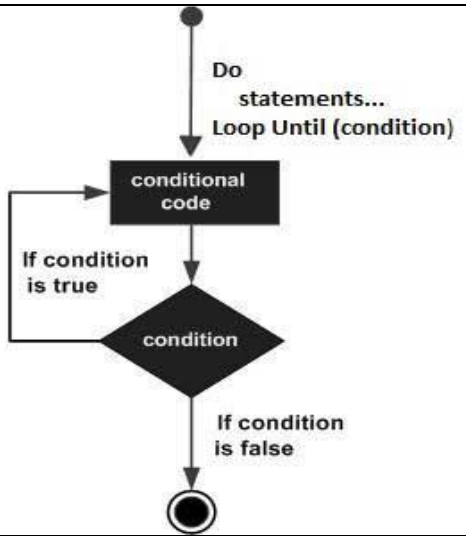
Loop Control Statements:

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

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.

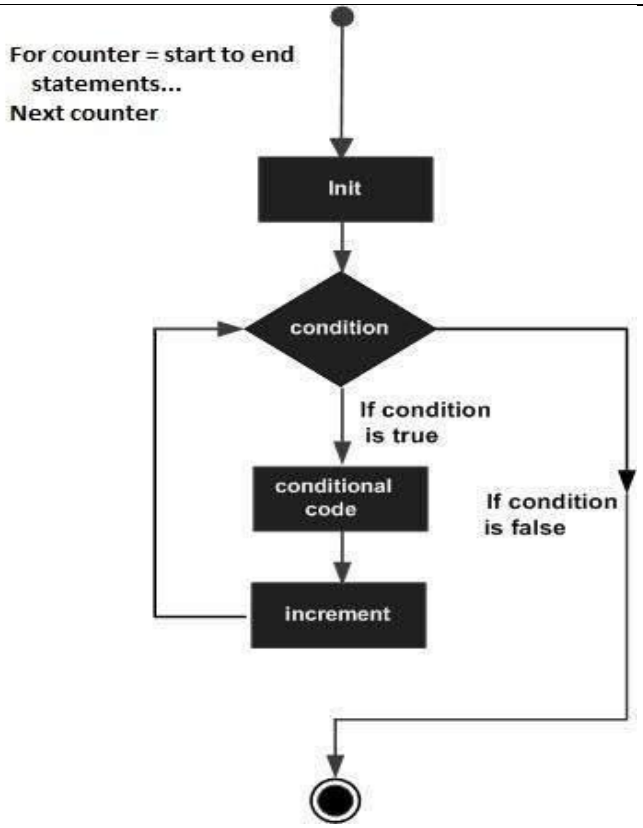
Do Loop Example	

```
Module Module1
   Sub Main()

      Dim a As Integer = 1
      Do
         Console.WriteLine("value of a: {0}", a)
         a = a + 1
      Loop While (a <= 10)
      Console.ReadLine()

   End Sub
End Module
```



For Loop Example

```
Module Module1
   Sub Main()
      Dim a As Integer
      For a = 1 To 10
      'For a = 10 To 20 Step 2
         Console.WriteLine(a)
      Next
         Console.ReadLine()

        End Sub
End Module
```

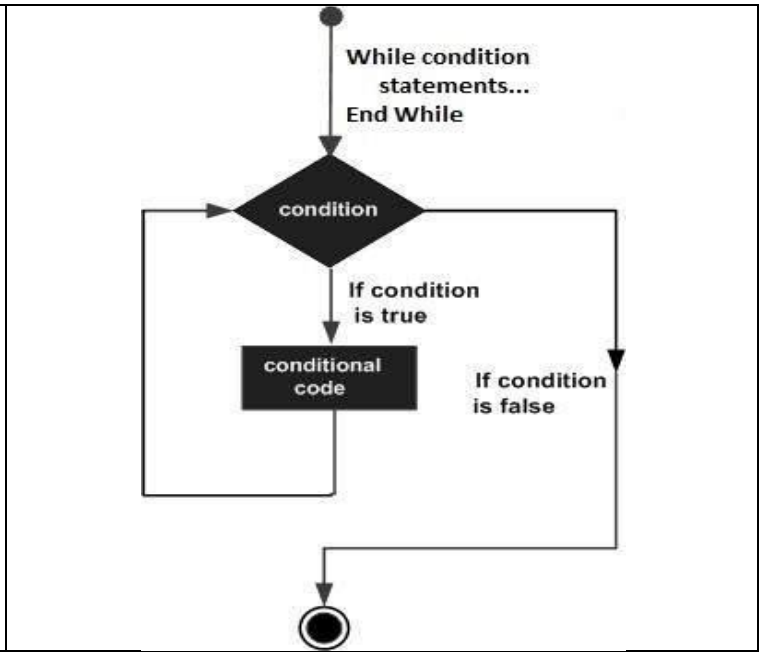


For Each Loop Example

```
Module module1
   Sub Main()
     Dim array() As String = {"dog", "cat", "fish"}

      For Each value As String In array
         Console.WriteLine(value)
      Next
      Console.Read()

   End Sub
End Module
```

While Loop Example

```
Module Module1
   Sub Main()

      Dim a As Integer = 10
      While a < 20
         Console.WriteLine("value of a: {0}", a)
         a = a + 1
      End While
      Console.ReadLine()

   End Sub
End Module
```



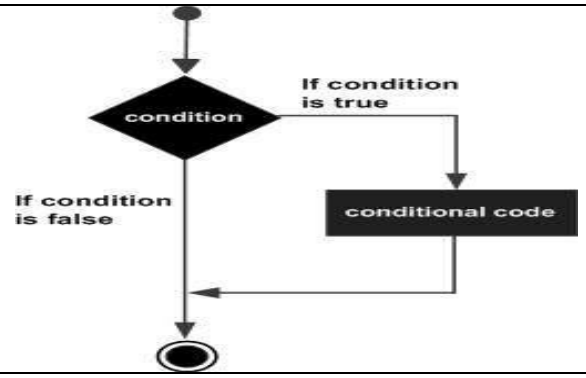
Conditional Statements

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 true, and optionally, other statements to be executed if the condition is false. VB.Net provides the following types of decision making statements.

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

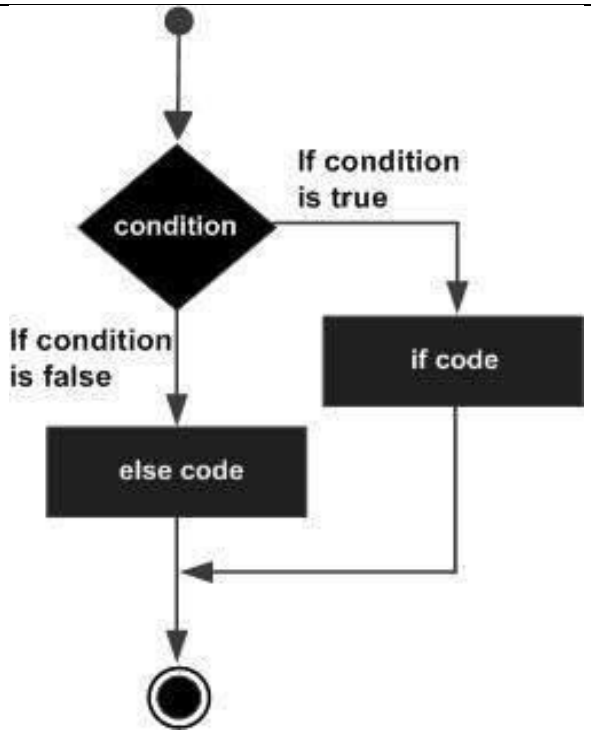
If ... Then Statement Example	

```
Module Module1
   Sub Main()
      Dim a As Integer = 10
      If (a < 20) Then
         Console.WriteLine("a is less than 20")
      End If
      Console.WriteLine("value of a is : {0}", a)
      Console.ReadLine()
   End Sub
End Module
```



If ... Then...Else Statement Example

```
Module Module1
   Sub Main()
      Dim a As Integer = 100
      If (a < 20) Then
         Console.WriteLine("a is less than 20")
      Else
         Console.WriteLine("a is not less than 20")
      End If
      Console.WriteLine("value of a is : {0}", a)
      Console.ReadLine()
   End Sub
End Module
```

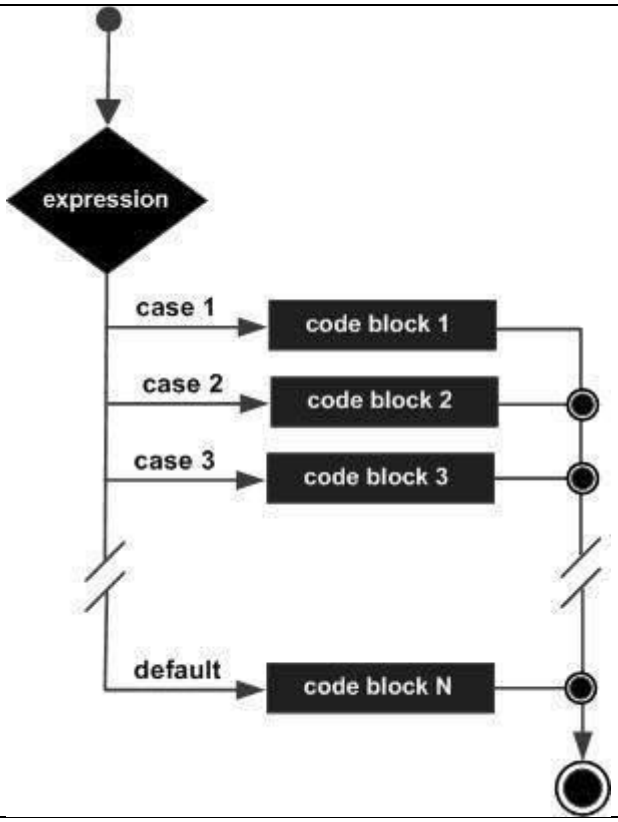


Nested If Statement Example

```
Module Module1
   Sub Main()
      Dim a As Integer = 100
      Dim b As Integer = 200
      If (a = 100) Then
         If (b = 200) Then
            Console.WriteLine("Value of a is 100 and b is 200")
         End If
      End If
      Console.WriteLine("Exact value of a is : {0}", a)
      Console.WriteLine("Exact value of b is : {0}", b)
      Console.ReadLine()
   End Sub
End Module
```

Select ...Case Statement Example

```
Module Module1
   Sub Main()
      Dim grade As Char
      Console.WriteLine("Enter grade A,B,C,D,E,F")
      grade = Console.ReadLine()

      Select Case grade
         Case "A"
            Console.WriteLine("Excellent!")
         Case "B", "C"
            Console.WriteLine("Well done")
         Case "D"
            Console.WriteLine("You passed")
         Case "F"
            Console.WriteLine("Better try again")
         Case Else
            Console.WriteLine("Invalid grade")
      End Select
      Console.WriteLine("Your grade is  {0}", grade)
      Console.ReadLine()
   End Sub
End Module
```



Nested Select ...Case Statement Example

```
Module Module1
   Sub Main()
      Dim a As Integer = 100
      Dim b As Integer = 200
      Select Case a
         Case 100
            Console.WriteLine("This is part of outer case ")
            Select Case b
               Case 200
                  Console.WriteLine("This is part of inner case ")
            End Select
      End Select
      Console.WriteLine("Exact value of a is : {0}", a)
      Console.WriteLine("Exact value of b is : {0}", b)
      Console.ReadLine()
   End Sub
End Module
```

VB.Net Controls

Form: Visual Basic Form is the container for all the controls that make up the user interface. Every window you see in a running visual basic application is a form, thus the terms form and window describe the same entity. Visual Studio creates a default form for you when you create a **Windows Forms Application**. Every form will have title bar on which the form's caption is displayed and there will be buttons to close, maximize and minimize the form shown below:

Properties	*Methods*	*Events*
BackColor: Sets the form background color.	**Close:** Closes the form.	**Click:** Occurs when the form is clicked.
Name: This is the actual name of the form.	**Focus:** Sets input focus to the control.	**GotFocus:** Occurs when the form control receives focus.
StartPosition: This property determines the initial position of the form when it's first displayed.	**Hide:** Conceals the control from the user.	**Load:** Occurs before a form is displayed for the first time.
Text: The text, which will appear at the title bar of the form.	**Refresh:** Forces the control to invalidate its client area & immediately redraw itself and any child controls.	**LostFocus:** Occurs when the form loses focus.
Font: This property specify font type, style, size	**Show :** Displays the control to the user.	**VisibleChanged:** Occurs when the Visible property value changes.

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

Properties	*Methods*	*Events*
Multiline: Gets or sets a value indicating whether this is a multiline Textbox control.	**Clear:** Clears all text from the text box control.	**Click:** Occurs when the control is clicked.
PasswordChar: Gets or sets the character used to mask characters of a password in a single-line Textbox control.	**ToString:** Returns a string that represents the TextBoxBase control.	**DoubleClick:** Occurs when the control is double-clicked.
ReadOnly: Gets or sets a value indicating whether text in the text box is read-only.	**Undo:** Undoes the last edit operation in the text box.	**TextAlignChanged:** Occurs when the TextAlign property value changes.
Text: Gets or sets the current text in the Textbox.	**ResetText:** Resets the Text property to its default value.	
TextLength: Gets the length of text in the control.	**AppendText:** Appends text to the current text of a text box.	

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

Properties	*Methods*	*Events*
ForeColor: Gets or sets the foreground color of the control.	**Refresh:** Forces the control to invalidate its client area and immediately redraw itself and any child	**GotFocus:** Occurs when the control receives focus.

	controls.	
Text: Gets or sets the text associated with this control.	**Select:** Activates the control.	**LostFocus:** Occurs when the control loses focus.
TextAlign: Gets or sets the alignment of text in the label.	**Show:** Displays the control to the user.	**TextChanged:** Occurs when the Text property value changes.
FlatStyle: Gets or sets the flat style appearance of the Label control	**ToString:** Returns a String that contains the name of the control.	**Click:** Occurs when the control is clicked.
BorderStyle: Gets or sets the border style for the control.		**TabIndexChanged:** Occurs when the TabIndex property value changes.

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

Properties	*Methods*	*Events*
BackColor: Gets or sets the background color of the control.	**Select:** Activates the control.	**Click:** Occurs when the control is clicked.
ForeColor: Gets or sets the foreground color of the control.	**ToString:** Returns a String containing the name of the Component, if any. This method should not be overridden.	**DoubleClick:** Occurs when the user double-clicks the Button control.
Text: Gets or sets the text associated with this control.	**GetPreferredSize:** Retrieves the size of a rectangular area into which a control can be fitted.	**GotFocus:** Occurs when the control receives focus.
TabIndex: Gets or sets the tab order of the control within its container.		**TextChanged:** Occurs when the Text property value changes.
Image: Gets or sets the image that is displayed on a button control.		**TabIndexChanged:** Occurs when the TabIndex property value changes.

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

Properties	*Methods*	*Events*
Appearance: Gets or sets a value determining the appearance of the radio button	**PerformClick:** Generates a Click event for the control, simulating a click by a user.	**AppearanceChanged:** Occurs when the value of the Appearance property of the Radio Button control is changed.
Checked: Gets or sets a value indicating whether the control is checked.		**CheckedChanged:** Occurs when the value of the Checked property of the Radio Button control is changed.
Text: Gets or sets the caption for a radio button.		

Check Box: The Check Box control allows the user to set true/false or yes/no type options. The user can select or deselect it. When a check box is selected it has the value True, and when it is cleared, it holds the value False. The CheckBox control has three states, **checked**, **unchecked** and **indeterminate**. In the indeterminate state, the check box is grayed out. To enable the indeterminate state, the *ThreeState* property of the check box is set to be **True**.

Properties	*Methods*	*Events*
ThreeState: Gets or sets a value indicating whether or not a check box should allow three check states rather than two.	**OnCheckedChanged:** Raises the CheckedChanged event.	**AppearanceChanged:** Occurs when the value of the Appearance property of the check box is changed.
Checked: Gets or sets a value indicating whether the check box is selected.	**OnCheckStateChanged:** Raises the CheckStateChanged event.	**CheckedChanged:** Occurs when the value of the Checked property of the CheckBox control is changed.
Appearance: Gets or sets a value determining the appearance of the check box.	**OnClick:** Raises the OnClick event.	**CheckStateChanged:** Occurs when the value of the CheckState property of the CheckBox control is changed.

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

Properties	*Methods*	*Events*
Image: Gets or sets the image that is displayed in the control.	**Load:** Displays an image in the picture box	**Click:** Occurs when the control is clicked.
ImageLocation: Gets or sets the path or the URL for the image displayed in the control.		**Resize:** Occurs when the control is resized.

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

Properties	*Methods*	*Events*
DataBindings: Gets the data bindings for the control.	**FindString:** Finds the first item in the combo box that starts with the string specified as an argument.	**SelectedIndexChanged :** Occurs when the SelectedIndex property of a Combo Box control has changed.
DataSource: Gets or sets the data source for this Combo Box.	**FindStringExact:** Finds the first item in the combo box that exactly matches the specified string.	**DropDownStyleChanged:** Occurs when the DropDownStyle property of the Combo Box has changed.
SelectedIndex: Gets or sets the index specifying the currently selected item.	**SelectAll:** Selects all the text in the editable area of the combo box.	**DropDown:** Occurs when the drop-down portion of a combo box is displayed.
Items: Gets an object representing the collection of the items contained in this Combo Box.		
SelectedItem: Gets or sets currently selected item in the Combo Box.		
SelectedText: Gets or sets the text that is selected in the editable portion of a Combo Box.		
SelectedValue: Gets or sets the value of the member property specified by the Value Member property.		

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

Properties	*Methods*	*Events*
Items: Gets the items of the list box.		**Click:** Occurs when a list box is selected.
SelectedIndex: Gets or sets the zero-based index of the currently selected item in a list box.	**OnSelectedIndexChanged:** Raises the SelectedIndexChanged event.	**SelectedIndexChanged:** Occurs when the SelectedIndex property of a list box is changed.
SelectedIndices: Gets a collection that contains the zero-based indexes of all currently selected items in the list box.	**OnSelectedValueChanged:** Raises the SelectedValueChanged event.	
SelectedItem: Gets or sets the currently selected item in the list box.	**SetSelected:** Selects or clears the selection for the specified item in a List Box.	
SelectedItems: Gets a collection containing the currently selected items in the list box.	**GetSelected:** Returns a value indicating whether the specified item is selected.	
SelectedValue: Gets or sets the value of the member property specified by the ValueMember property.	**ClearSelected :** Unselects all items in the List Box.	
SelectionMode: Gets or sets the method in which items are selected in the list box. This property has values: None, One, MultiSimple, MultiExtended		

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

Properties	*Methods*	*Events*
Maximum: Gets or sets the upper limit of values of the scrollable range.	**OnClick:** Generates the Click event.	**Scroll:** Occurs when the control is moved.
Minimum: Gets or sets the lower limit of values of the scrollable range.	**Select:** Activates the control.	**ValueChanged:** Occurs when the Value property changes, either by handling the Scroll event or programmatically.

Tree View: The Tree View control is used to display hierarchical representations of items similar to the ways the files and folders are displayed in the left pane of the Windows Explorer. Each node may contain one or more child nodes.

Properties	*Methods*	*Events*
DataBindings: Gets the data bindings for the control.	**CollapseAll:** Collapses all the nodes including all child nodes in the tree view control.	**TextChanged:** Occurs when the Text property changes.
Nodes: Gets the collection of tree nodes that are assigned to the tree view control.	**ExpandAll:** Expands all the nodes.	**AfterSelect:** Occurs after the tree node is selected.
PathSeparator: Gets or sets the delimiter string that the tree node path uses.	**GetNodeAt:** Gets the node at the specified location.	**AfterCollapse:** Occurs after the tree node is collapsed.

SelectedNode: Gets or sets the tree node that is currently selected in the tree view control.	**GetNodeCount:** Gets the number of tree nodes.	**AfterExpand:** Occurs after the tree node is expanded.
ShowLines: Gets or sets a value indicating whether lines are drawn between tree nodes in the tree view control.	**Sort :** Sorts all the items in the tree view control.	
ShowPlusMinus: Gets or sets a value indicating whether plus-sign (+) and minus-sign (-) buttons are displayed next to tree nodes that contain child tree nodes.	**ToString :** Returns a string containing the name of the control.	
ShowRootLines: Gets or sets a value indicating whether lines are drawn between the tree nodes that are at the root of the tree view.		

List View: The ListView control is used to display a list of items. Along with the TreeView control, it allows you to create a Windows Explorer like interface. he *ListView* control displays a list of items along with icons. The Item property of the ListView control allows you to add and remove items from it. The *SelectedItem* property contains a collection of the selected items. The *MultiSelect* property allows you to set select more than one item in the list view. The *CheckBoxes* property allows you to set check boxes next to the items.

Properties	***Methods***	***Events***
Items: Gets a collection containing all items in the control.	**Clear:** Removes all items from the ListView control.	**SelectedIndexChanged:** Occurs when the selected index is changed.
MultiSelect: Gets or sets a value indicating whether multiple items can be selected.	**ToString:** Returns a string containing the string representation of the control.	**TextChanged:** Occurs when the Text property is changed.
SelectedIndices: Gets the indexes of the selected items in the control.		**ItemCheck:** Occurs when an item in the control is checked or unchecked.
SelectedItems: Gets the items that are selected in the control.		
CheckBoxes: Gets or sets a value indicating whether a check box appears next to each item in the control.		

Image List: The main advantage of using the **ImageList** is that you can treat the images as a collection. ***ImageList Control*** is used to store images that can be used with other controls. This control works fine with the controls that have the **ImageList** and **ImageIndex** property.

Properties	***Methods***	***Events***
Images: Property gets an ImageCollection object for this image list.	**Draw:** Method used to draw the given image.	

Timer: ***Timer Control*** is used to set time intervals; this control is visible only at design time and not in the runtime.

Properties	*Methods*	*Events*
Enabled: Property used to Get or set whether the timer is running.	**Start:** Method used to start timer.	**Tick:** Triggered when the time interval has elapsed.
Interval: Property used to set or get the time in millisecond between the timer clicks.	**Stop:** Method used to stop timer.	

Splitters: Permits the user to resize docked controls at runtime, With additional Splitter controls You can divide the form into smaller and smaller resizable areas Make sure to include a container control to hold the Splitter and the two docked controls SplitContainer control provides functionality of a splitter to divide and resize two controls.

Properties	*Methods*	*Events*
SplitterDistance: this property gets or sets the location of the splitter, in pixels, from the left or top edge of the SplitContainer.		
SplitterIncrement: this property gets or sets a value representing the increment of splitter movement in pixels.		
SplitterRectangle: this property gets the size and location of the splitter relative to the SplitContainer.		
Orientation: Panels on a SplitContainer can be placed horizontally or vertically.		

Progress Bar: It represents a Windows progress bar control. It is used to provide visual feedback to your users about the status of some task. It shows a bar that fills in from left to right as the operation progresses. The ProgressBar control is typically used when an application performs tasks such as copying files or printing documents. To a user the application might look unresponsive if there is no visual cue. In such cases, using the ProgressBar allows the programmer to provide a visual status of progress.

Properties	*Methods*	*Events*
Maximum: Gets or sets the maximum value of the range of the control.	**Increment:** Increments the current position of the ProgressBar control by specified amount.	**Click:** Occurs when the control is clicked.
Minimum: Gets or sets the minimum value of the range of the control.	**PerformStep:** Increments the value by the specified step.	**TextChanged:** Occurs when the Text property changes.
Value: Gets or sets the current position of the progress bar.	**ResetText:** Resets the Text property to its default value.	

Menu Strip: The **MenuStrip** control represents the container for the menu structure. The MenuStrip control works as the top-level container for the menu structure. The ToolStripMenuItem class and the ToolStripDropDownMenu class provide the functionalities to create menu items, sub menus and drop-down menus.

Properties	*Methods*	*Events*
GripStyle: Gets or sets the visibility of the grip used to reposition the control.		**MenuActivate:** Occurs when the user accesses the menu with the keyboard or mouse.
Stretch: Gets or sets a value indicating whether the MenuStrip stretches from end to end in its container.		**MenuDeactivate:** Occurs when the MenuStrip is deactivated.

ShowItemToolTips: Gets or sets a value indicating whether ToolTips are shown for the MenuStrip.		
MdiWindowListItem: Gets or sets the ToolStripMenuItem that is used to display a list of Multiple-document interface (MDI) child forms.		

Status Bar: The Status Bar presents a single panel at the bottom of form

Tool Bar: A toolbar is a bar that displays in the top section under the main menu. A toolbar is primarily a container, which itself means nothing and doesn't even do anything. The controls you position on it give it meaning. Still, because of its position, it enjoys some visual characteristics but also imposes some restrictions to its objects. When you add a toolbar to a form, it automatically positions itself in the top section of the form and uses the same width as the form. This means that the default **Dock** value of a toolbar is **Top**. The **ToolStripMenuItem** class supports the menus and menu items in a menu system. You handle these menu items through the click events in a menu system.

Properties	*Methods*	*Events*
Enabled: Gets or sets a value indicating whether the control is enabled		**CheckedChanged:** Occurs when the value of the Checked property changes.
ShortcutKeys: Gets or sets the shortcut keys associated with the ToolStripMenuItem.		**CheckStateChanged:** Occurs when the value of the CheckState property changes.
ShowShortcutKeys: Gets or sets a value indicating whether the shortcut keys that are associated with the ToolStripMenuItem are displayed next to the ToolStripMenuItem.		
Checked: Gets or sets a value indicating whether the ToolStripMenuItem is checked.		
CheckOnClick: Gets or sets a value indicating whether the ToolStripMenuItem should automatically appear checked and unchecked when clicked.		
CheckState: Gets or sets a value indicating whether a ToolStripMenuItem is in the checked, unchecked, or indeterminate state.		

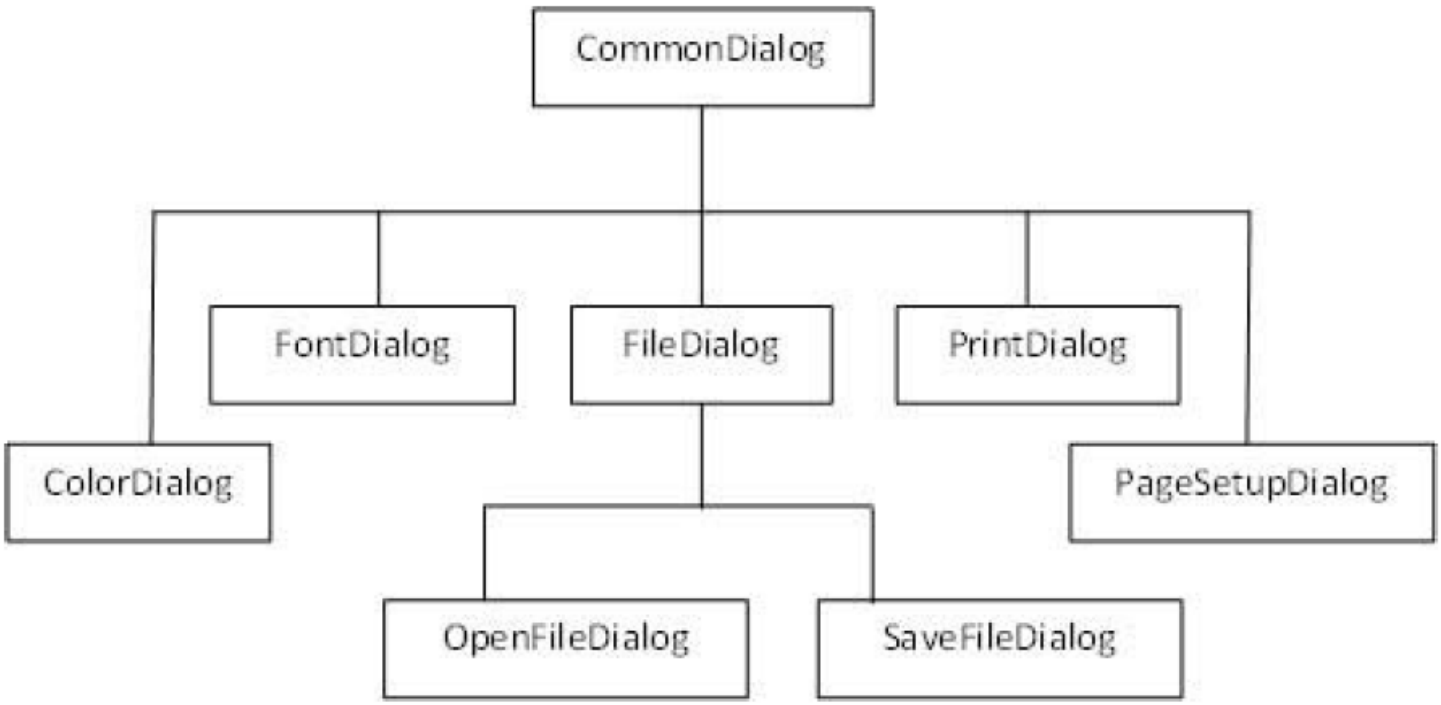
VB.NET - DIALOG BOXES

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

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

The RunDialog() function is automatically invoked when a user of a dialog box calls its ShowDialog() function.
The ShowDialog method is used to display all the dialog box controls at run time.

The following diagram shows the common dialog class inheritance:



All these above mentioned classes have corresponding controls that could be added from the Toolbox during design time. You can include relevant functionality of these classes to your application, either by instantiating the class programmatically or by using relevant controls. When you double click any of the dialog controls in the toolbox or drag the control onto the form, it appears in the Component tray at the bottom of the Windows Forms Designer, they do not directly show up on the form.

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

ColorDialog: It represents a common dialog box that displays available colors along with controls that enable the user to define custom colors.

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    If ColorDialog1.ShowDialog <> Windows.Forms.DialogResult.Cancel Then
        Label1.ForeColor = ColorDialog1.Color
    End If
End Sub
```

FontDialog: It prompts the user to choose a font from among those installed on the local computer and lets the user select the font, font size, and color.

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    If FontDialog1.ShowDialog <> Windows.Forms.DialogResult.Cancel Then
        RichTextBox1.ForeColor = FontDialog1.Color
        RichTextBox1.Font = FontDialog1.Font
    End If
End Sub
```

SaveFileDialog: It prompts the user to select a location for saving a file and allows the user to specify the name of the file to save data.

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    SaveFileDialog1.Filter = "TXT Files (*.txt*)|*.txt"
        If SaveFileDialog1.ShowDialog = Windows.Forms.DialogResult.OK  Then
            My.Computer.FileSystem.WriteAllText (SaveFileDialog1.FileName, RichTextBox1.Text, True)
        End If
End Sub
```

PrintDialog: It lets the user to print documents by selecting a printer and choosing which sections of the document to print from a Windows Forms application.

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    PrintDialog1.Document = PrintDocument1
    PrintDialog1.PrinterSettings = PrintDocument1.PrinterSettings
    PrintDialog1.AllowSomePages = True
        If PrintDialog1.ShowDialog = DialogResult.OK Then
            PrintDocument1.PrinterSettings = PrintDialog1.PrinterSettings
            PrintDocument1.Print()
        End If
End Sub
```

OpenFileDialog: It prompts the user to open a file and allows the user to select a file to open.

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    If OpenFileDialog1.ShowDialog <> Windows.Forms.DialogResult.Cancel Then
        PictureBox1.Image = Image.FromFile(OpenFileDialog1.FileName)
    End If
End Sub
```

Delegate:

A delegate is very much like a C/C++ **'function pointer'** or a **'typedef** that represents the address of a function'. In C/C++, the address of a function is just a memory address. The C/C++ function pointer holds only the address of a function. This address doesn't carry any additional information, such as the number of parameters expected by the function, types of parameters it takes etc. In fact, this all makes a function pointer type unsafe. Traditionally, calling a function by its address depends on the language supporting the function pointers. And function pointers are inherently dangerous. Delegates add a safety to the idea of traditional function pointers.

The .NET framework has added the bonus of providing a type-safe mechanism called **Delegate**s. They follow a Trust-But-Verify model, with automatic verification of the signature by the compiler. Unlike function pointers however, delegates are object-oriented, type-safe, and secured. In short, a delegate is a data structure that refers to a static method or to an object instance, and an instance method of that object. When the delegate references an instance method, it stores not only a reference to the method entry point, but also a reference to the object instance for which to invoke the method. In VB.NET, we define a delegate using the '**`Delegate`**' keyword.

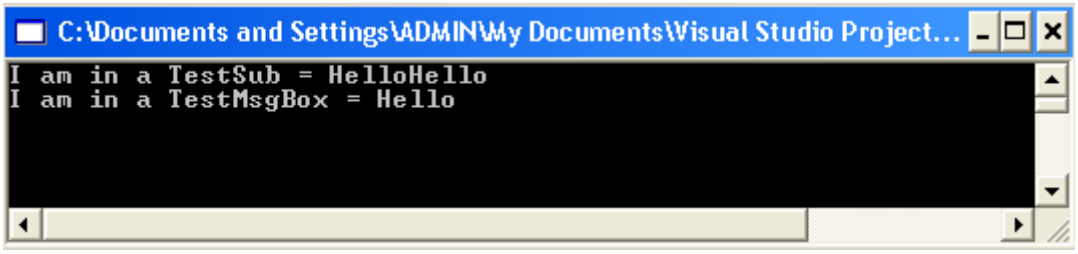
E.g.: `Public Delegate Sub MyDelegate(ByVal MsgString As String)`

"Delegate" is a name used to describe procedures -- that is, functions and subroutines -- in VB.NET. Just like the type Integer refers to whole numbers and the type Boolean refers to true or false, the type Delegate refers to procedures. The syntax of declaring a Delegate is almost the same. You don't use the Dim keyword but you can use a keyword like Public or Private just like you would with another declaration. A delegate is declared by coding the name and signature of a procedure with the keyword Delegate.

Example

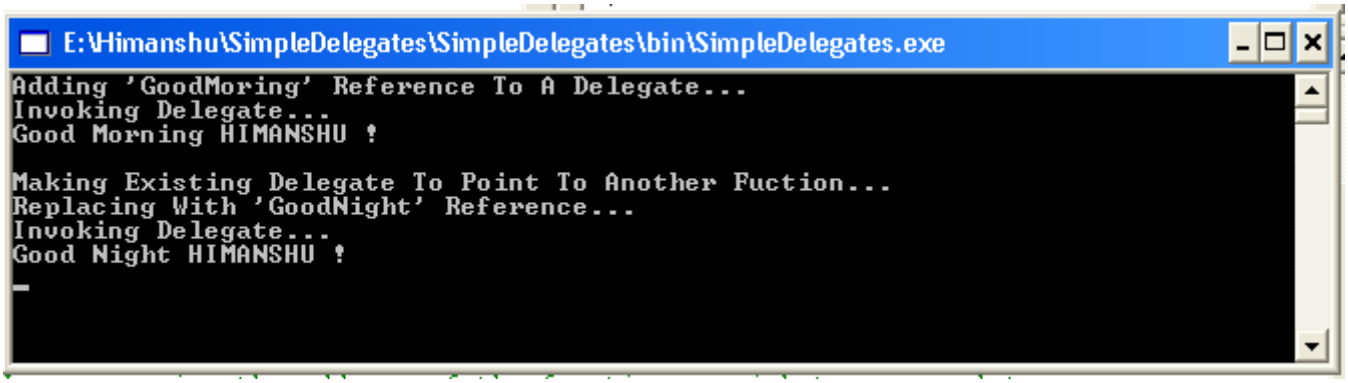
```
Imports System
Module Module1

    Public Delegate Sub MyDel(ByVal Msg As String)

    Public Class Demo

        Public Sub TestSub(ByVal Msg As String)
            Console.WriteLine("I am in a TestSub=" & Msg & Msg)
        End Sub

        Public Sub TestMsgBox(ByVal Msg As String)
            Console.WriteLine("I am in a TestMsgBox=" & Msg)
        End Sub
    End Class

    Sub Main()

        Dim T As New Demo
        Dim D As MyDel

        D = AddressOf T.TestSub
        D("Hello")

        D = AddressOf T.TestMsgBox
        D("Hello")

        Console.Read()

    End Sub
```

```
End Module
```



 # Example Program for Delegate

```
Imports System
Module Module1

'Defining the delegate
    Public Delegate Sub GreetingDelegate(ByVal MsgString As String)
'Two different functions
    Public Sub GoodMoring(ByVal YourName As String)
        Console.WriteLine("Good Morning " + YourName + " !")
    End Sub

    Public Sub GoodNight(ByVal YourName As String)
        Console.WriteLine("Good Night " + YourName + " !")
    End Sub

'Main Method
    Sub Main()
'Instantiating the delegate
        Dim MyGreeting As GreetingDelegate
'Here we assign the address of the function we wish to encapsulate to the delegate
        Console.WriteLine("Adding 'GoodMoring' Reference To A Delegate...")
        MyGreeting = AddressOf GoodMoring

'Invoking the delegate
        Console.WriteLine("Invoking Delegate...")
        MyGreeting.Invoke("HIMANSHU")

'Assiging the addressof the another function to the same delegate
        Console.WriteLine()
        Console.WriteLine("Making Existing Delegate To Point To Another Fuction.")
        Console.WriteLine("Replacing With 'GoodNight' Reference...")
        MyGreeting = New GreetingDelegate(AddressOf GoodNight)

'Another  way of invoking the delegate.
        Console.WriteLine("Invoking Delegate...")
        MyGreeting("HIMANSHU")
    End Sub

End Module
```



Class Library Overview

The .NET Framework class library is a library of classes, interfaces, and value types that provide access to system functionality. It is the foundation on which .NET Framework applications, components, and controls are built.

This is also called as **Base Class Library (BCL)** and it is common for all types of applications i.e. the way you access the Library Classes and Methods in VB.NET will be the same in C#, and it is common for all other languages in .NET. The following are different types of applications that can make use of .net class library.

5. Windows Application- It is desktop application.
6. Console Application- an application that takes input and displays output at a command line console.
7. Web Application- All websites are example of web application. They use a web server.
8. Web Services- It doesn't use web-based server. Internet payment systems are example of web services.

In short, developers just need to import the BCL in their language code and use its predefined methods and properties to implement common and complex functions like reading and writing to file, graphic rendering, database interaction, and XML document manipulation.

FCL includes some 600 managed classes. A flat hierarchy consisting of hundreds of classes would be difficult to navigate. Microsoft partitioned the managed classes of FCL into separate namespaces based on functionality. For example, classes pertaining to local input/output can be found in the namespace **System.IO**. To further refine the hierarchy, FCL namespaces are often nested; the tiers of namespaces are delimited with dots. **System.Runtime.InteropServices**, **System.Security.Permissions**, and **System.Windows.Forms** are examples of nested namespaces. The root namespace is System, which provides classes for console input/output, management of application domains, delegates, garbage collection, and more. Some examples are as follows-

System: Contains fundamental classes and base classes that define commonly used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions. Other classes provide services supporting data type conversion, method parameter manipulation, mathematics, remote and local program invocation, application environment management, and supervision of managed and unmanaged applications.

System.Data: Contains classes that constitute most of the ADO.NET architecture. The ADO.NET architecture enables you to build components that efficiently manage data from multiple data sources.

System.Data.Design: Contains classes that can be used to generate a custom typed dataset.

System.Data.SqlClient: Contains classes that encapsulate the .NET Framework Data Provider for SQL Server. The .NET Framework Data Provider for SQL Server describes a collection of classes used to access a SQL Server database in the managed space.

System.Drawing: Provides access to GDI+ basic graphics functionality. More advanced functionality is provided in the System.Drawing.Drawing2D, System.Drawing.Imaging, and System.Drawing.Text namespaces.

System.IO: Contains types that enable synchronous and asynchronous reading and writing on data streams and files.

System.Net: Provides a simple programming interface for many of the protocols used on networks today. The WebRequest and WebResponse classes form the basis of what are called pluggable protocols, an implementation of network services that enables you to develop applications that use Internet resources without worrying about the specific details of the individual protocols.

System.Web: Provides classes and interfaces that enable browser-server communication. This namespace includes the HttpRequest class, which provides extensive information about the current HTTP request, the HttpResponse class, which manages HTTP output to the client, and the HttpServerUtility class, which provides access to server-side utilities and processes. System.Web also includes classes for cookie manipulation, file transfer, exception information, and output cache control.

Creating User Defined Class

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.

Objects (Instance Variable) and Classes

Each object in Visual Basic is defined by a *class*. A class describes the variables, properties, procedures, and events of an object. Objects are instances of classes; you can create as many objects you need once you have defined a class.

Dim <object name> as New <Class Name>()

Dim obj As New Test()

Member Functions and Data Member

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.

Differences between Classes and Modules

The main difference between classes and modules is that classes can be instantiated as objects while standard modules cannot. Because there is only one copy of a standard module's data, when one part of your program changes a public variable in a standard module, any other part of the program gets the same value if it then reads that variable. In contrast, object data exists separately for each instantiated object.

Classes and modules also use different scopes for their members. Members defined within a class are scoped within a specific instance of the class and exist only for the lifetime of the object. To access class members from outside a class, you must use fully qualified names in the format of *Object.Member*.

On the other hand, members declared within a module are publicly accessible by default, and can be accessed by any code that can access the module. This means that variables in a standard module are effectively global variables because they are visible from anywhere in your project, and they exist for the life of the program.

Example

```
Imports System
Module Module1

    Public Class Test
        Sub Display ()
            Console.Write ("VB Dot Net")
        End Sub
    End Class

Sub Main ()
     Dim obj As New Test ()        ---------------------------- Creating Object of the class
     obj.Display ()
     Console.Read ()
End Sub

End Module
```

Constructor and Instance Variable (Object)

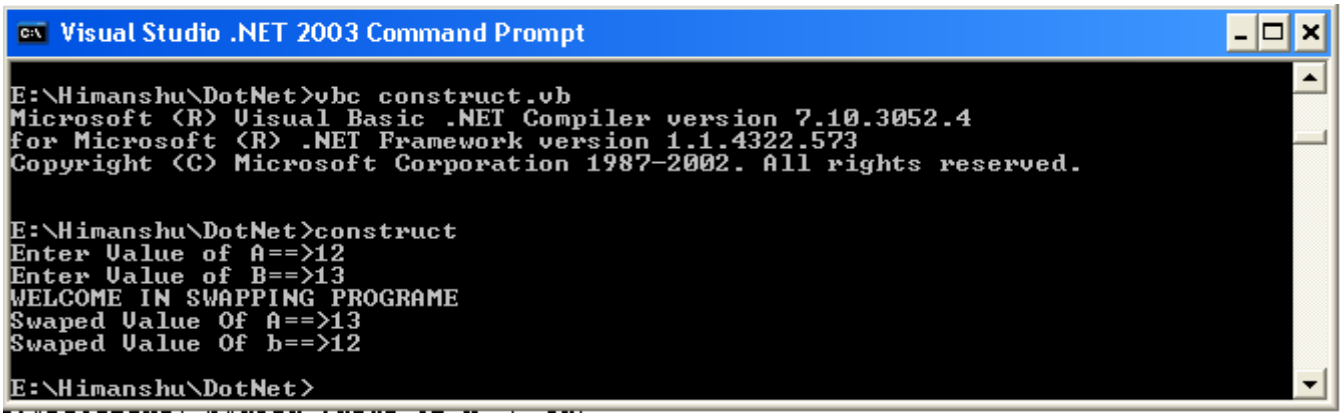
 # Constructor

1) A constructor is a special member function whose task is to initialize the objects of it's class. This is the first method that is run when an instance of a type is created. A class can have multiple constructors.
2) A constructor is invoked whenever an object of it's associated class is created. If a class contains a constructor, then an object created by that class will be initialized automatically.
3) We pass data to constructor by enclosing it in the parentheses when creating an object.
4) Constructors can never return a value, and can be overridden to provide custom initialization functionality.
5) In .Net we create constructors by adding a Sub procedure named **New** to a class, regardless the name of class
6) It can have any access modifier (Public, Protected, Friend, Private, or Default). In most cases, a constructor has a **Public** access modifier.

Example:

```
Imports System
Module Module1

Public Class Cons
    Public a,b,temp as Integer

    Public sub New()
       Console.WriteLine("WELCOME IN SWAPPING PROGRAME")
    End Sub

    Public Sub New(ByVal x as Integer,ByVal y as Integer)
      a=x
      b=y
```

```
      temp=a
      a=b
      b=temp
    End Sub

    Public Function Ans() as integer
       Console.WriteLine("Swaped Value Of A==>"&a)
       Console.WriteLine("Swaped Value Of b==>"&b)
    End Function
End Class

Sub Main()
  Dim p,q as integer
  Console.Write("Enter Value of A==>")
  p= Console.ReadLine()
  Console.Write("Enter Value of B==>")
  q=Console.ReadLine()
  Dim obj1 as New Cons()
  Dim obj As New Cons(p,q)
  obj.Ans()
End Sub

End Module
```

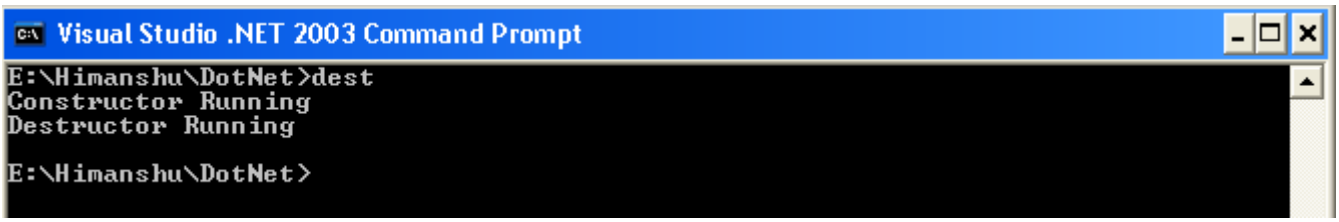


 # Destructor

A destructor, also known as **finalizer,** is the last method run by a class. Within a destructor we can place code to clean up the object after it is used, which might include decrementing counters or releasing resources. We use **Finalize** method in Visual Basic for this and the Finalize method is called automatically when the .NET runtime determines that the object is no longer required. When working with destructors we need to use the **overrides** keyword with Finalize method as we will override the Finalize method built into the Object class. We normally use Finalize method to deallocate resources and inform other objects that the current object is going to be destroyed. Because of the nondeterministic nature of garbage collection, it is very hard to determine when a class's destructor will be called.

```
Imports System
Module Module1

Public Class Destructor
    Public Sub New()
      Console.Writeline("Constructor Running")
    End Sub
    Protected Overrides Sub Finalize()
        Console.Writeline("Destructor Running")
```

```
        Console.ReadLine()
    End Sub
End Class

Sub Main()
  Dim obj As New Destructor()
End Sub

End Module
```



 # Instance Variable (Fields/Objects)

Fields are variable that declared so that they are available to all code within a class. Typically fields are private in scope, available only to the code in the class itself. They are also sometimes referred to as instance variable, member variable or data member.

Property and fields are both different terminology. A property is a type of method that is geared to retrieving and setting values, while a field is variable within the class that may hold the value exposed by the property.

```
Public Class Student
    Private Sname as string
    Public Age as integer

    Public Sub getValue(Bval sn as integer,Byval ag as integer)
      Sname=sn
      Age=ag
    End Sub

    Public Sub display()
        Console.Writeline("Student Name="&Sname)
        Console.Writeline("Student Age="&Age)
    End Sub
End Class

Sub Main()
     Dim obj As New Student()
     obj.getvalue("John",28)
     obj.display()
     Console.Readline
End Sub
```

Error Handling (Exception Handling) in Dot Net

An error *is a term used to describe any issue that arises unexpectedly that causes a computer to not function properly. Computers can encounter either software errors or hardware errors. Following types of error generally comes to existence while programming-*

1) Syntax Errors
These errors are the easiest to find because they are highlighted by the compiler. This type of error is caused by the failure of the programmer to use the correct grammatical rules of the language. Syntax errors are detected, and displayed, by the compiler as it attempts to translate your program, i.e. the Source code into the Object code. If a program has a syntax error it cannot be translated, and the program will not be executed.
The compiler tries to highlight syntax errors where there seems to be a problem, however, it is not perfect and sometimes the compiler will indicate the next line of code as having the problem rather than the line of code where the problem actually exists.

2) Run-Time Errors
Run-time errors are detected by the computer and displayed during execution of a program. They will halt the program when they occur but they often do not show up for some time. A run-time error occurs when the user directs the computer to perform an illegal operation, eg: Dividing a number by zero, Assigning a variable to the wrong type of variable, Using a variable in a program before assigning a value to it.
When a run-time error occurs, the computer stops executing your program, and displays a diagnostic message that indicates the line where the error occurred.

3) Logic Errors
These are the hardest errors to find as they do not halt the program. They arise from faulty thinking on behalf of the programmer. They can be very troublesome. These are mistakes in a program's logic. Programs with logic errors will often compile, execute, and output results. However, at least some of the time the output will be incorrect. Error messages will generally not appear if a logic error occurs, this makes logic errors very difficult to locate and correct.

Exception Handling/Error 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.

```
Public Class Form1
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        Try
            Dim i As Integer
            Dim resultValue As Integer
```

```
            i = 100
            resultValue = i / 0
            MsgBox("The result is " & resultValue)
        Catch ex As Exception
            MsgBox("Exception catch here ..")
        Finally
            MsgBox("Finally block executed ")
        End Try
    End Sub
End Class
```

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.

System.Exception

System.Exception represents errors that occur during application execution. This is the base class for all exceptions in VB.net, so any other type of exception must be derived from it. The Exception class is present in the System namespace and in the assembly mscorlib.dll. Some Important Properties of the Exception class:

- **StackTrace**: This StackTrace property can be used to determine where the error occurred.
- **Message**: The Message property returns a description to the error's cause.
- **InnerException**: The InnerException property can be used to create and preserve a series of exceptions during exception handling.
- **HelpLink**: The HelpLink property can hold a URL to a help file that provides extensive information about the cause of the exception.

The following are 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 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 differencing 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.

The *Err* Object:

Err is a special Visual Basic object that's assigned detailed error handling information each time a runtime error occurs. The most useful *Err* properties for identifying runtime errors are *Err.Number* and *Err.Description*.

Err.Number: contains the number of the most recent runtime error.

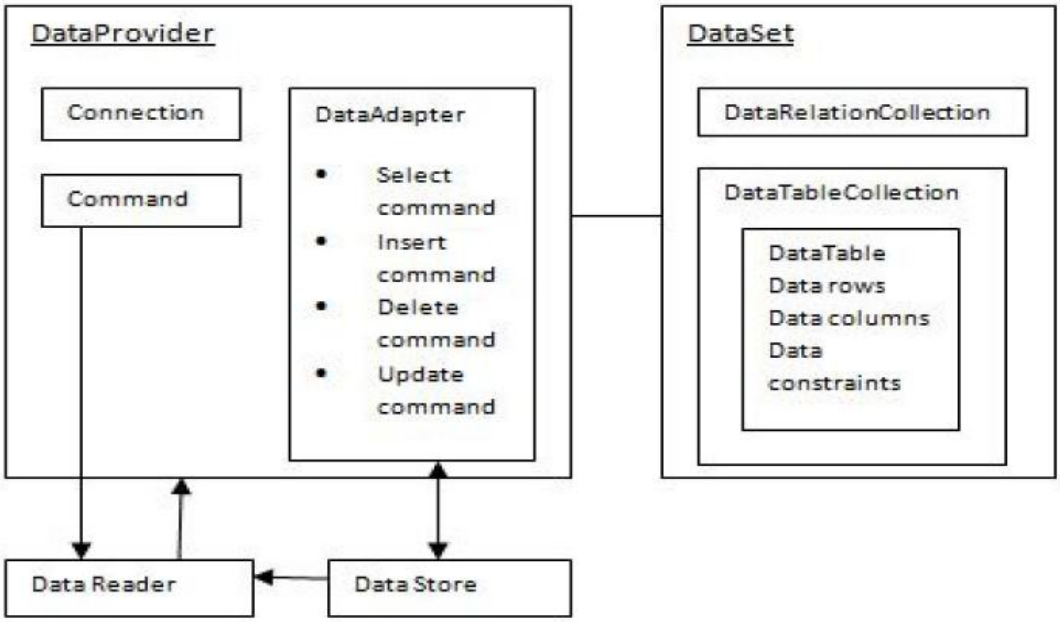
Err.Description: a short error message that matches the runtime error numbers.

Error Number	Default Error Message
5	Procedure call or argument is not valid
6	Overflow
7	Out of Memory
11	Division by Zero
51	Internal Error
52	Bad file name or number
53	File not found
55	File already open
76	Path not found
482	Printer error

VB.Net Database Access:

Applications communicate with a 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 database is retrieved through the data provider. Various components of the data provider, retrieves data and update data. An application accesses data either through a dataset or a data reader.

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

Core ADO.NET Namespaces

- ✓ **System.Data:** It makes up the core objects such as DataTable, DataColumn, DataView, and Constraints, to name a few. This namespace forms the basis for the others.
- ✓ **System.Data.OleDb:** Defines objects that we use to connect to and modify data in various data sources. It contained drivers for Microsoft SQL Server, the Microsoft OLE DB Provider for Oracle, and Microsoft Provider for Jet 4.0. This class is useful if your project connects to many different data sources, but you want more performance than the ODBC provider.
- ✓ **System.Data.SqlClient:** A data provider namespace created specifically for Microsoft SQL Server version 7.0 and later. When using Microsoft SQL Server, this namespace is written to take advantage of the Microsoft SQL Server API directly
- ✓ **System.Data.SQLTypes:** Provides classes for data types specific to Microsoft SQL Server. These classes are designed specifically for SQL Server and provide better performance.
- ✓ **System.Data.ODBC:** This namespace is intended to work with all compliant ODBC drivers, and is available as a separate download from Microsoft

----------***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:

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:** It 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.

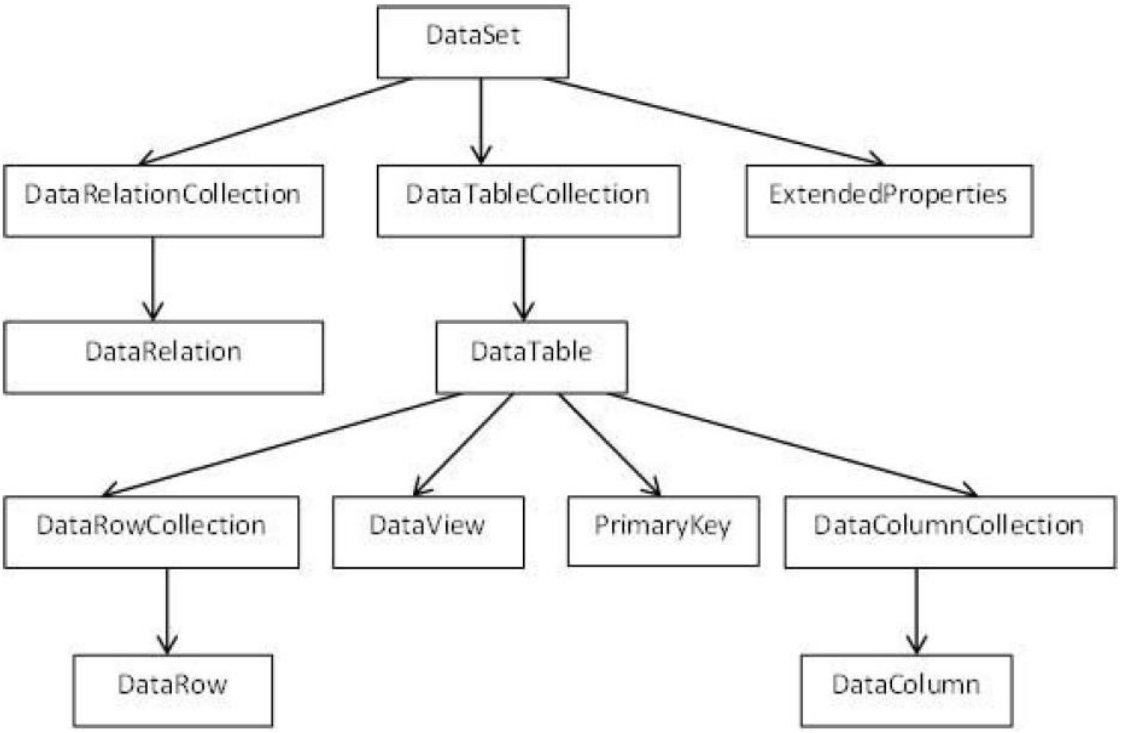
The Command, Connection, DataReader, and DataAdapter are the core objects in ADO.NET. They form the basis for all operations regarding data in .NET. These objects are created from the ***System.Data.OleDb***, ***System.Data.SqlClient***, and the ***System.Data*.ODBC** namespaces.

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 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. The DataSet class is present in the **System.Data** namespace.

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:



- ✓ **DataTableCollection:** It contains all the tables retrieved from the data source.
- ✓ **DataRelationCollection:** It contains relationships and the links between tables in a data set.
- ✓ **ExtendedProperties:** It contains additional information, like the SQL statement for retrieving data, time of retrieval etc.
- ✓ **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.
- ✓ **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.
- ✓ **DataRowCollection:** It contains all the rows in a DataTable.
- ✓ **DataView:** It represents a fixed customized view of a DataTable for sorting, filtering, searching, editing and navigation.
- ✓ **PrimaryKey:** It represents the column that uniquely identifies a row in a DataTable.
- ✓ **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.
- ✓ **DataColumnCollection:** It represents all the columns in a DataTable.
- ✓ **DataColumn:** It consists of the number columns that comprise a DataTable.

Command Object

The Command objects **OleDbCommand** and **SqlCommand** allow us to execute statements directly against the database. They provide for simple and direct route to our data, regardless of where the data resides. Command objects are particularly useful for executing INSERT, UPDATE, and DELETE statements, but can also generate DataReader and XMLDataReader

Command is used to execute almost any SQL command from within the .net application. The SQL command like insert, update, delete, select, create, alter, drop can be executed with command object and you can also call stored procedures with the command object.

Command object has the following important properties.

- **Connection:** used to specify the connection to be used by the command object.
- **CommandType:** Used to specify the type of SQL command you want to execute. To assign a value to this property, use the enumeration CommandType that has the members Text, StoredProcedure and TableDirect. Text is the default and is set when you want to execute ant SQL command with command object. StoredProcedure is set when you want to call a stored procedure or function and **TableDirect** is set when you want to retrieve data from the table directly by specifying the table name without writing a select statement.
- **CommandText:** Used to specify the SQL statement you want to execute.
- **Transaction:** Used to associate a transaction object to the command object so that the changes made to the database with command object can be committed or rollback.

Command object has the following important methods.

- **ExecuteNonQuery():** Used to execute an SQL statement that doesn't return any value like insert, update and delete. Return type of this method is **int** and it returns the no. of rows affected by the given statement.
- **ExecuteScalar():** Used to execute an SQL statement and return a single value. When the select statement executed by executescalar() method returns a row and multiple rows, then the method will return the value of first column of first row returned by the query. Return type of this method is object.
- **ExecuteReader():** Used to execute a select a statement and return the rows returned by the select statement as a DataReader. Return type of this method is DataReader.

Connection Object

The Connection Object is a part of ADO.NET Data Provider and it is a unique session with the Data Source. In .Net Framework the Connection Object is handling the part of physical communication between the application and the Data Source. Depends on the parameter specified in the Connection String , ADO.NET Connection Object connect to the specified Database and open a connection between the application and the Database . When the connection is established, SQL Commands may be executed, with the help of the Connection Object, to retrieve or manipulate data in the Database. Once the Database activity is over, Connection should be closed and releases the resources. In ADO.NET the type of the Connection is depend on what Database system you are working with. The following are the commonly using the connections in the ADO.NET

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

```
Imports System.Data.SqlClient
Module Module1
   Sub Main()
       Dim str As String = "Data Source=.;uid=sa;pwd=123;database=master"
       Dim con As New SqlConnection(str)
         Try
              con.Open()
             Console.WriteLine("connection open")
             con.Close()
          Catch ex As Exception
              Console.WriteLine("can not open connection")
          End Try
   End Sub
End Module
```

Data Reader

DataReader is a read-only, forward only and connected record set from the database. In DataReader, database connection is opened until the object is closed unlike DataSet. Using DataReader we can able to access one row at a time so there it is not required storing it in memory. In one DataReader we can get more than one result set and access it one by one

SqlDataReader Object provides a connection oriented data access to the SQL Server data sources. The method *ExecuteReader()* in the *SqlCommand Object* send the SQL statements to *SqlConnection Object* and populate a *SqlDataReader Object* based on the SQL statement.

Dim sqlReader As SqlDataReader = sqlCmd.ExecuteReader ()

When the ExecuteReader method in SqlCommand Object execute, it instantiate a SqlClient.SqlDataReader Object. When started to read from a DataReader it should always be open and positioned prior to the first record. The Read() method in the DataReader is used to read the rows from DataReader and it always moves forward to a new valid row, if any row exist . ***SqlDataReader.Read()***

There are two way two ways to read the data which is stored in database. One way is *DataSet* and other is *DataReader*. Dataset works with disconnected mode and DataReader works with connected architecture. DataReader is used only for read only and forward only so we cannot do any transaction on them. Using DataReader we can able to access one row at a time so there is no need to storing it in memory.

ExecuteReader and ExecuteNonQuery method

ExecuteReader: ExecuteReader used for getting the query results as a DataReader object. It is read-only forward only retrieval of records and it uses select command to read through the table from the first to the last.

```
Dim reader As SqlDataReader
    reader = Command. ExecuteReader ()
        While reader. Read ()
            MsgBox (reader.Item (0))
        End While
    reader.Close ()
```

ExecuteNonQuery: ExecuteNonQuery used for executing queries that does not return any data. It is used to execute the sql statements like update, insert, delete etc. ExecuteNonQuery executes the command and returns the number of rows affected.

```
Dim Value as Integer
    Command = New SqlCommand (Sql, Connection)
    Value = Command. ExecuteNonQuery ()
```

Example

```
Dim str As String = "Data Source=SqlServer; Initial catalog=MYD; Integrated Security=True"
Dim con As New SqlConnection (str)
  Try
      con.Open ()
        Dim cmd As SqlCommand = con.CreateCommand
         cmd.CommandText = "SELECT Sname FROM Student"
        Dim dr As SqlDataReader = command. ExecuteReader
            If dr.HasRows Then
                While dr.Read
                    Console.WriteLine (reader(0).ToString)
                End While
            End If
          dr.Close()
        cmd.Dispose()
 Catch ex As Exception
      Console.Write(ex.Message)
 Finally
   con.Close()
End Try
```

Data Adapter

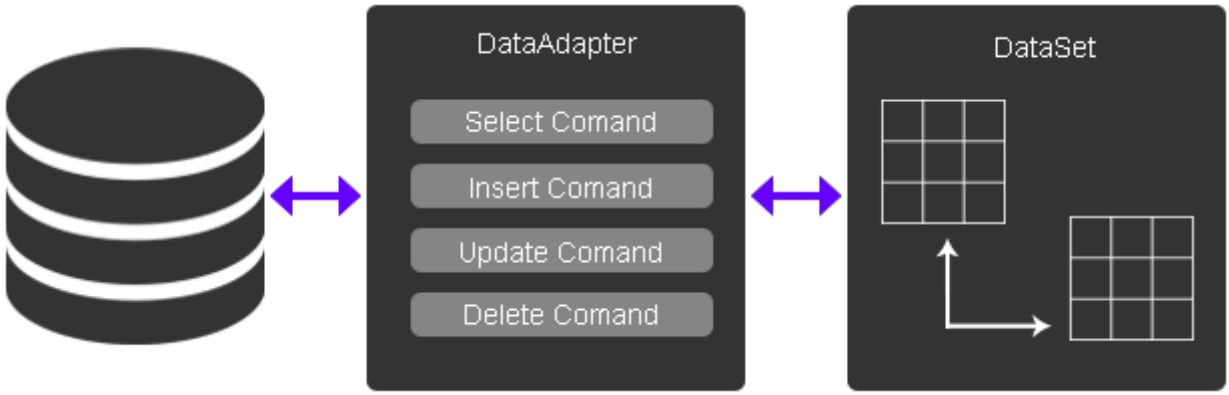
SqlDataAdapter is a part of the ADO.NET Data Provider and it resides in the System.Data.SqlClient namespace. It provides the communication between the Dataset and the SQL database. We can use SqlDataAdapter Object in combination with Dataset Object.

SqlDataAdapter Object and DataSet objects are combining to perform both data access and data manipulation operations in the SQL Server Database. When the user perform the SQL operations like Select , Insert etc. in the data containing in the Dataset Object , it won't directly affect the Database, until the user invoke the Update method in the SqlDataAdapter.

The DataAdapter serves as a bridge between a DataSet and a data source for retrieving and saving data. The DataAdapter provides this bridge by mapping Fill, which changes the data in the DataSet to match the data in the data source, and Update, which changes the data in the data source to match the data in the DataSet.

DataAdapter.Fill (DataSet)
DataAdapter.Fill (DataTable)

The Fill method retrieves rows from the data source using the SELECT statement specified by an associated SelectCommand property. If the data adapter encounters duplicate columns while populating a DataTable, it generates names for the subsequent columns.
It **differs from the DataReader**, in that the DataReader uses the Connection object to access the data directly, without having to use a DataAdapter. The DataAdapter essentially **decouples the DataSet** object from the actual source of the data, whereas the **DataReader** is tightly bound to the data in a read-only fashion.



Example

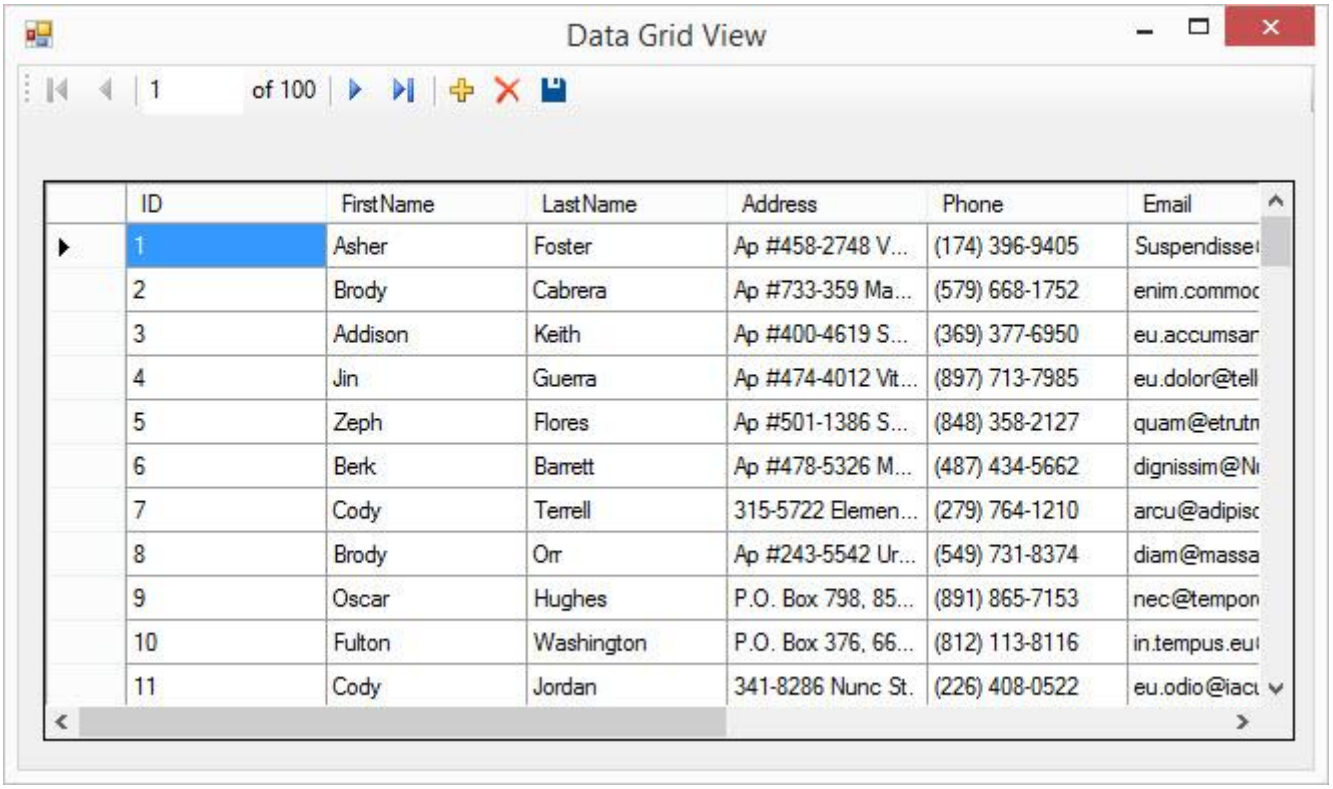
```
Private Sub BindGrid ()
    Dim constring As String = "Data Source=SqlServer; Initial catalog=MYD; Integrated Security=True"
    Dim con As New SqlConnection (constring)
    Dim cmd As New SqlCommand ("SELECT * FROM STUDENT", con)
        cmd. CommandType = CommandType. Text
    Dim da As New SqlDataAdapter (cmd)
    Dim ds As New DataSet ()
        da.Fill (ds)
        dataGridView1.DataSource = ds.Tables (0)
  End Sub
```

DataGridView Control

It is a control which provides an interactive user-interface to show information graphically from database. The end-user can add, edit and remove columns. And also, they can edit the content if the permission is given. The DataGridView control looks like the Excel spreadsheet, which can be added onto the form so that the end-user can use it for specific purposes.

The DataGridView control is basically made of rows and columns. Additionally, every column has its header. Vertical and horizontal scrollbars appear automatically whenever they are needed. The scrollbars are used to scroll through the pages to see more content. So a DataGridView control may contain information of multiple pages.

DataGridView provides a visual interface to data. It is an excellent way to display and allow editing for your data. It is accessed with VB.NET code. Data edited in the DataGridView can then be persisted in the database.



Important DataGridView Properties

DataMember	Gets or sets the name of the list or table in the data source for which the DataGridView is displaying data.
DataSource	Gets or sets the data source that the DataGridView is displaying data for.
RowCount	Gets or sets the number of rows displayed in the DataGridView.
Rows	Gets a collection that contains all the rows in the DataGridView control.
EditMode	Gets or sets a value indicating how to begin editing a cell.

Example

```
Dim str As String = "Data Source=SqlServer; Initial catalog=MYDB; Integrated Security=True"
Dim sql As String = "SELECT * FROM Authors"
Dim con As New SqlConnection(str)
Dim da As New SqlDataAdapter (sql, con)
Dim ds As New DataSet ()
con.Open ()
da.Fill (ds, "Authors_table")
con.Close ()
DataGridView1.DataSource = ds
DataGridView1.DataMember = "Authors_table"
```

Validation Controls

One of the most common requests for any Web application is the ability to perform client-side validation of input.
There are a number of reasons for performing validation on the client. First, you give the user a better experience. If you can immediately notify the user that he did not fill in a required field, you just saved him the time it would have taken to submit the form, have the server generate a message to inform him of the problem, and return the error message to him. In addition, using client-side validation lessens network traffic and server load by never sending invalid data to the server. Microsoft provides a series of validation controls in ASP.NET that automate client side form validation. The validation controls in ASP.NET are smart; they will perform the validation at the client if possible.

Types of Validators

• **RequiredFieldValidator:** This validator requires that its *ControlToValidate property* have a value. In other words, the control to which this validator is tied cannot be left blank.

• **CompareValidator:** This validator compares the value the user entered with a value you specify. Your specified value could be a constant, a calculated value, or a value from a database.

Properties	Description
Type	It specifies the data type.
ControlToCompare	It specifies the value of the input control to compare with.
ValueToCompare	It specifies the constant value to compare with.
Operator	It specifies the comparison operator, the available values are: Equal, NotEqual, GreaterThan, GreaterThanEqual, LessThan, LessThanEqual, and DataTypeCheck.

• **RangeValidator:** This validator requires that entered data be within a particular range. The range can be numeric, dates, currency, or alphabetical.

Properties	Description
Type	It defines the type of the data. The available values are: Currency, Date, Double, Integer, and String.
MinimumValue	It specifies the minimum value of the range.
MaximumValue	It specifies the maximum value of the range.

• **RegularExpressionValidator:** Regular expressions are also known as *masks*. This validator can make sure that entered data matches a particular format, such as the format of phone numbers and Social Security numbers.

commonly used syntax constructs for regular expressions:	
Character Escapes	**Description**
\b	Matches a backspace.
\t	Matches a tab.
\r	Matches a carriage return.
\v	Matches a vertical tab.
\f	Matches a form feed.
\n	Matches a new line.
****	Escape character.

Quantifiers could be added to specify number of times a character could appear.	
Quantifier	**Description**
*****	Zero or more matches.
+	One or more matches.
?	Zero or one matches.
{N}	N matches.
{N,}	N or more matches.
{N,M}	Between N and M matches.

Apart from single character match, a class of characters could be specified that can be matched, called the metacharacters.	
Metacharacters	**Description**
.	Matches any character except \n.
[abcd]	Matches any character in the set.
[^abcd]	Excludes any character in the set.
[2-7a-mA-M]	Matches any character specified in the range.
\w	Matches any alphanumeric character and underscore.
\W	Matches any non-word character.
\s	Matches whitespace characters like, space, tab, new line etc.
\S	Matches any non-whitespace character.
\d	Matches any decimal character.
\D	Matches any non-decimal character.

• **CustomValidator:** This validator uses code you write yourself to validate the data. The CustomValidator control allows writing application specific custom validation routines for both the client side and the server side validation.

• **ValidationSummary:** The ValidationSummary control does not perform any validation but shows a summary of all errors in the page. The summary displays the values of the ErrorMessage property of all validation controls that failed validation. The following two mutually inclusive properties list out the error message:

- ShowSummary: shows the error messages in specified format.
- ShowMessageBox: shows the error messages in a separate window.

Data Binding Controls

Every ASP.NET web form control inherits the DataBind method from its parent Control class, which gives it an inherent capability to bind data to at least one of its properties. This is known as **simple data binding** or **inline data binding**. Simple data binding involves attaching any collection (item collection) which implements the *IEnumerable* interface, or the DataSet and DataTable classes to the DataSource property of the control.

On the other hand, some controls can bind records, lists, or columns of data into their structure through a DataSource control. These controls derive from the BaseDataBoundControl class. This is called **declarative data binding**.

The data source controls help the data-bound controls implement functionalities such as, sorting, paging, and editing data collections.

The BaseDataBoundControl is an abstract class, which is inherited by two more abstract classes:

- DataBoundControl
- HierarchicalDataBoundControl

The abstract class DataBoundControl is again inherited by two more abstract classes:

- ListControl
- CompositeDataBoundControl

The controls capable of simple data binding are derived from the ListControl abstract class and these controls are:

- BulletedList
- CheckBoxList
- DropDownList
- ListBox
- RadioButtonList

The controls capable of declarative data binding (a more complex data binding) are derived from the abstract class CompositeDataBoundControl. These controls are:

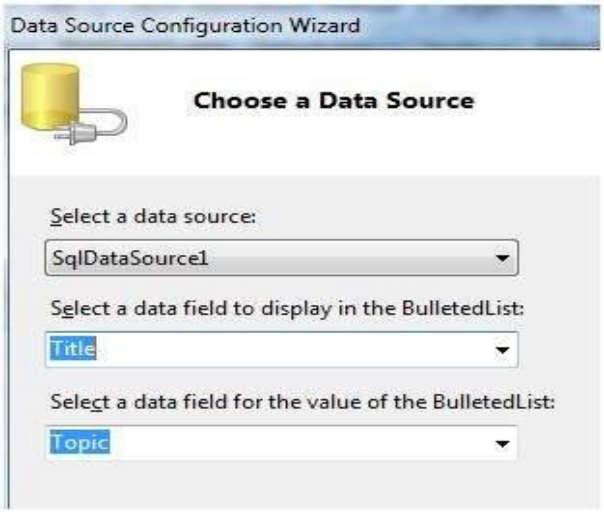
- DetailsView
- FormView
- GridView
- RecordList

Simple Data Binding

Simple data binding involves the read-only selection lists. These controls can bind to an array list or fields from a database. Selection lists takes two values from the database or the data source; one value is displayed by the list and the other is considered as the value corresponding to the display.

Let us take up a small example to understand the concept. Create a web site with a bulleted list and a SqlDataSource control on it. Configure the data source control to retrieve two values from your database
Choosing a data source for the bulleted list control involves:

- Selecting the data source control
- Selecting a field to display, which is called the data field
- Selecting a field for the value



When the application is executed, check that the entire title column is bound to the bulleted list and displayed.

Declarative Data Binding

The data binding involves the following objects:

- A dataset that stores the data retrieved from the database.
- The data provider, which retrieves data from the database by using a command over a connection.
- The data adapter that issues the select statement stored in the command object; it is also capable of update the data in a database by issuing Insert, Delete, and Update statements.

Relation between the data binding objects: