

VISUAL BASIC.NET PROGRAMMING

LECTURE NOTES (Semester-IV)

for

Bachelor of Computer Applications



Department of Computer Science and Applications

Vinayaka Mission's Research Foundation

**School of Arts And Science, Av Campus
Chennai-603104**

*Lecture Note Prepared By
S.MAHALAKSHMI, Asst.Professor*

SYLLUBUS VISUAL BASIC.NET PROGRAMMING

OBJECTIVES:

At the end of this course the learner is expected:

1. To gain in-depth knowledge on .NET frame work
2. To develop business applications using VB .net
3. To understand ADO .Net for database programming.

UNIT - I

(12 Hours)

.NET FRAMEWORK AND VB.NET: Evolution of the .NET Framework – Overview of the .Net Framework – VB.NET – Simple VB.Net Program. **VARIABLES, CONSTANTS AND EXPRESSIONS:** Value Types and Reference Types – Variable Declarations and Initializations – Value Data Types – Reference Data Types – Boxing and Unboxing – Arithmetic Operators – Textbox Control – Label Control – Button Control.

UNIT – II

(12 Hours)

CONTROL STATEMENTS: If Statements – Radio Button Control – Check Box Control – Group Box Control – Listbox Control – Checked List Box Control – Combo box Control – Select Case Statement – While Statement – Do Statement – For Statement. **METHODS AND ARRAYS:** Types of Methods – One Dimensional Array – Multi Dimensional Arrays – Jagged Arrays. **CLASSES:** Definition And Usage of a Class – Constructor Overloading – Copy Constructor – Instance and Shared Class Members – Shared Constructors.

UNIT – III

(12 Hours)

INHERITANCE AND POLYMORPHISM: Virtual Methods – Abstract Class and Abstract Methods – Sealed Classes. **INTERFACES, NAMESPACES AND COMPONENTS:** Definition of Interfaces – Multiple Implementations of Interfaces – Interface Inheritance – Namespaces – Components – Access Modifiers. **DELEGATES, EVENTS AND ATTRIBUTES:** Delegates – Events – Attributes – Reflection.

UNIT - IV

(12 Hours)

EXCEPTION HANDLING: Default Exception Handling Mechanism – User Defined Exception Handling Mechanism – Throw Statement – Custom Exception. **MULTITHREADING:** Usage Of Threads – Thread Class – Start(), Abort(), Join(), and Sleep() Methods – Suspend() And Resume() Methods – Thread Priority – Synchronization **I/O STREAMS:** Binary Data Files – Text Files - Data Files – FileInfo and DirectoryInfo Classes.

UNIT - V

(12 Hours)

ADDITIONAL CONTROLS: Timer – ProgressBar – LinkLabel – Panel – TreeView – Splitter – Menu – SDI & MDI – Dialog Boxes – Toolbar – StatusBar. **DATABASE CONNECTIVITY:** Advantages Of ADO.NET – Managed Data Providers – Developing a Simple ADO.NET Based Application – Creation of Data Table – Retrieving Data From Tables – Table Updating – Disconnected Data Access Through Dataset Objects.

Total Hours: 60

TEXT BOOK

1. Muthu C. (2008), "Visual Basic.NET", 2nd Ed., Vijay Nicole Imprints Pvt.Ltd.,.

REFERENCES

1. Jeffrey R.Shaprio (2002), "Visual Basic .NET The Complete Reference", Mac Graw Hill
2. Michael Halvorson (2010), "Visual Basic 2010 Step by Step", Microsoft Press.
3. Harold Davis (2002), "Visual Basic.NET Programming", Sybex.

VISUAL BASIC.NET PROGRAMMING

UNIT-I

.NET FRAMEWORK AND VB.NET

Evolution of the .net Framework:

The .Net framework is a software development platform developed by Microsoft. The framework was meant to create applications, which would run on the Windows Platform. The first version of the .Net framework was released in the year 2002.

The version was called .Net framework 1.0. The .Net framework has come a long way since then, and the current version is 4.7.1.

The .Net framework can be used to create both - **Form-based** and **Web-based** applications. Web services can also be developed using the .Net framework.

The framework also supports various programming languages such as Visual Basic and C#. So developers can choose and select the language to develop the required application. In this chapter, you will learn some basics of the .Net framework.

In this tutorial, you will learn-

- .Net Framework Architecture
- .NET Components
- .Net Framework Design Principle

Overview of the .net framework:

.NET Framework is a technology that supports building and running Windows apps and web services. .NET Framework is designed to fulfill the following objectives:

- To provide a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but web-distributed, or executed remotely.
- To provide a code-execution environment that minimizes software deployment and versioning conflicts.
- To provide a code-execution environment that promotes safe execution of code, including code created by an unknown or semi-trusted third party.
- To provide a code-execution environment that eliminates the performance problems of scripted or interpreted environments.
- To make the developer experience consistent across widely varying types of apps, such as Windows-based apps and Web-based apps.

Visual Basic.Net Programming

- To build all communication on industry standards to ensure that code based on .NET Framework integrates with any other code.

VB.Net:

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

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

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

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

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

Strong Programming Features VB.Net

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

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

Visual Basic.Net Programming

- Indexers
- Conditional Compilation
- Simple Multithreading

Simple VB.Net Program:

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

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

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

Imports System

```
Module Module1
```

```
"This program will display Hello World
```

```
Sub Main()
```

```
Console.WriteLine("Hello World")
```

```
Console.ReadKey()
```

```
End Sub
```

```
End Module
```

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

Hello, World!

Let us look various parts of the above program –

- The first line of the program **Imports System** is used to include the System namespace in the program.
- The next line has a **Module** declaration, the module *Module1*. VB.Net is completely object oriented, so every program must contain a module of a class that contains the data and procedures that your program uses.
- Classes or Modules generally would contain more than one procedure. Procedures contain the executable code, or in other words, they define the behavior of the class. A procedure could be any of the following –
 - Function

- Sub
 - Operator
 - Get
 - Set
 - AddHandler
 - RemoveHandler
 - RaiseEvent
- The next line('This program) will be ignored by the compiler and it has been put to add additional comments in the program.
 - The next line defines the Main procedure, which is the entry point for all VB.Net programs. The Main procedure states what the module or class will do when executed.
 - The Main procedure specifies its behavior with the statement
Console.WriteLine("Hello World") *WriteLine* is a method of the *Console* class defined in the *System* namespace. This statement causes the message "Hello, World!" to be displayed on the screen.
 - The last line **Console.ReadKey()** is for the VS.NET Users. This will prevent the screen from running and closing quickly when the program is launched from Visual Studio .NET.

Compile & Execute VB.Net Program

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

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

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

- Open a text editor and add the above mentioned code.
- Save the file as **helloworld.vb**

Visual Basic.Net Programming

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

VARIABLES, CONSTANTS AND EXPRESSIONS

Value Types and Reference Types:

There are two kinds of types in Visual Basic: reference types and value types. Variables of reference types store references to their data (objects), while variables of value types directly contain their data. With reference types, two variables can reference the same object; therefore, operations on one variable can affect the object referenced by the other variable. With value types, each variable has its own copy of the data, and it is not possible for operations on one variable to affect the other

Value Types

A data type is a *value type* if it holds the data within its own memory allocation. Value types include the following:

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

Every structure is a value type, even if it contains reference type members. For this reason, value types such as Char and Integer are implemented by .NET Framework structures.

You can declare a value type by using the reserved keyword, for example, Decimal. You can also use the New keyword to initialize a value type. This is especially useful if the type has a constructor that takes parameters. An example of this is the Decimal(Int32, Int32, Int32, Boolean, Byte) constructor, which builds a new Decimal value from the supplied parts.

Reference Types

A *reference type* stores a reference to its data. Reference types include the following:

- String
- All arrays, even if their elements are value types

Visual Basic.Net Programming

- Class types, such as Form
- Delegates

A class is a *reference type*. Note that every array is a reference type, even if its members are value types.

Since every reference type represents an underlying .NET Framework class, you must use the New Operator keyword when you initialize it. The following statement initializes an array.

```
Dim totals() As Single = New Single(8) { }
```

Variable Declaration and Initialization:

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

The basic value types provided in VB.Net can be categorized as

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

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

Variable Declaration in VB.Net

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

Syntax for variable declaration in VB.Net is –

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

Where,

- **attributelist** is a list of attributes that apply to the variable. Optional.
- **accessmodifier** defines the access levels of the variables, it has values as - Public, Protected, Friend, Protected Friend and Private. Optional.
- **Shared** declares a shared variable, which is not associated with any specific instance of a class or structure, rather available to all the instances of the class or structure. Optional.

Visual Basic.Net Programming

- **Shadows** indicate that the variable re-declares and hides an identically named element, or set of overloaded elements, in a base class. Optional.
- **Static** indicates that the variable will retain its value, even when the after termination of the procedure in which it is declared. Optional.
- **ReadOnly** means the variable can be read, but not written. Optional.
- **WithEvents** specifies that the variable is used to respond to events raised by the instance assigned to the variable. Optional.
- **Variablelist** provides the list of variables declared.

Each variable in the variable list has the following syntax and parts –
variablename[([boundslist])] [As [New] datatype] [= initializer]

Where,

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

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

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

Variable Initialization in VB.Net

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

```
variable_name = value;
```

for example,

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

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

Visual Basic.Net Programming

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

Example

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

Live Demo

```
Module variablesNdatatypes
Sub Main()
Dim a As Short
Dim b As Integer
Dim c As Double

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

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

a = 10, b = 20, c = 30

Value Data Types:

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

Value types include the following:

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

For example

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

```
ModuleModule1
Sub Main()
Dim m AsInteger = 5
```

Visual Basic.Net Programming

```
Dim n As Integer = m
m = 3
Console.WriteLine("m=" & m)
Console.WriteLine("n=" & n)
EndSub
EndModule
```

OUTPUT

w1.gif

Reference Data Type:

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

String

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

For Example

```
Module Module1
Sub Main()
Dim objX As New System.Text.StringBuilder(" Rohatash Kumar")
Dim objY As System.Text.StringBuilder
objY = objX
objX.Replace("World", "Test")
Console.WriteLine(objY.ToString())
EndSub
EndModule
```

OUTPUT

w2.gif

Boxing and UnBoxing:

VB provides us with Value types and Reference Types. Value Types are stored on the stack and Reference types are stored on the heap. The conversion of value type to reference type is known as Boxing and converting reference type back to the value type is known as Unboxing.

Boxing:

Convert ValueTypes to Reference Types also known as boxing.

```
Dimx AsInt32 = 10
Dimo AsObject= x ' Implicit boxing
Console.WriteLine("The Object o = ", &o) ' prints out 10
```

```
Dimx AsInt32 = 10
Dimo AsObject= CObj(x) ' Explicit Boxing
Console.WriteLine("The object o = ", &o) ' prints out 10
```

Unboxing:

UnBoxing an object type back to value type.

```
Dimx AsInt32 = 5
Dimo As Object= x ' Implicit Boxing
x = o ' Implicit UnBoxing
```

```
Dimx AsInt32 = 5
Dimo As Object= x ' Implicit Boxing
x = CInt(Fix(o)) ' Explicit UnBoxing
```

Arithmetic Operators in VB.Net:

You can use arithmetic operators to perform various mathematical operations in VB.NET. They include:

Symbol	Description
^	for raising an operand to the power of another operand
+	for adding two operands.
-	for subtracting the second operand from the first operand.
*	for multiplying both operands.
/	for dividing an operand against another. It returns a floating point result.
\	for dividing an operand against another. It returns an integer result.
MOD	known as the modulus operator. It returns the remainder after division.

Let us demonstrate how to use these using an example:

Visual Basic.Net Programming

Step 1) Create a new console application. To know this, visit our previous tutorial on Data Types and Variables.

13

Step 2) Add the following code:

Module Module1

```
Sub Main()
Dim var_w As Integer = 11
Dim var_x As Integer = 5
Dim var_q As Integer = 2
Dim var_y As Integer
Dim var_z As Single

var_y = var_w + var_z
Console.WriteLine(" Result of 11 + 5 is {0} ", var_y)

var_y = var_w - var_x
Console.WriteLine(" Result of 11 - 5 is {0} ", var_y)

var_y = var_w * var_x
Console.WriteLine(" Result of 11 * 5 is {0} ", var_y)

var_z = var_w / var_x
Console.WriteLine(" Result of 11 / 5 is {0}", var_z)

var_y = var_w \ var_x
Console.WriteLine(" Result of 11 \ 5 is {0}", var_y)

var_y = var_w Mod var_x
Console.WriteLine(" Result of 11 MOD 5 is {0}", var_y)

var_y = var_x ^ var_x
Console.WriteLine(" Result of 5 ^ 5 is {0}", var_y)
Console.ReadLine()

End Sub

End Module
```

Step 3) Click the Start button to execute the code.

Textbox Control:

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

TextBox Properties

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

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

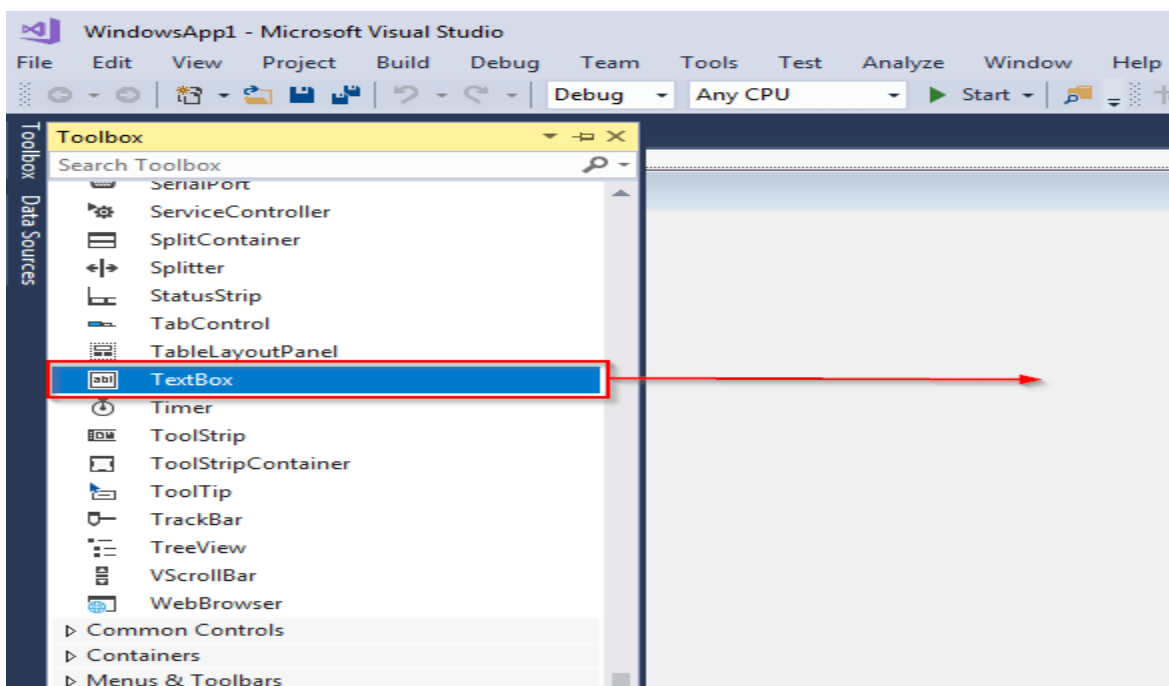
Textbox Events

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

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

How to Create a TextBox

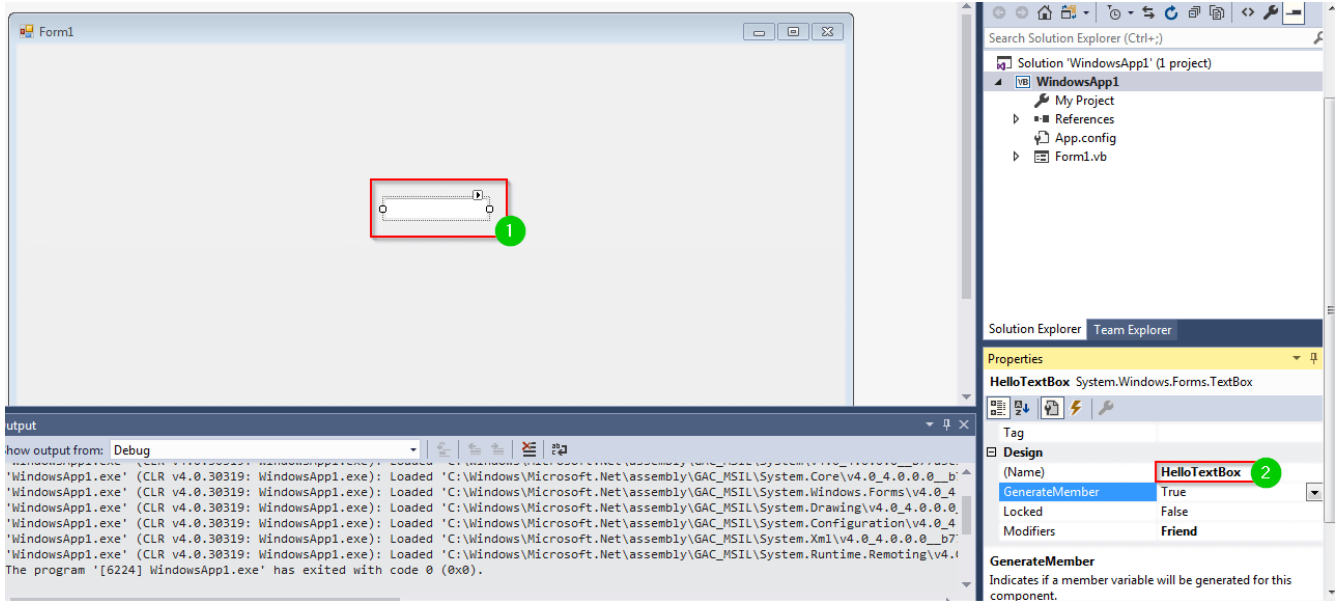
Step 1) To create a TextBox, drag the TextBox control from the toolbox into the WindowForm:



Visual Basic.Net Programming

Step 2)

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

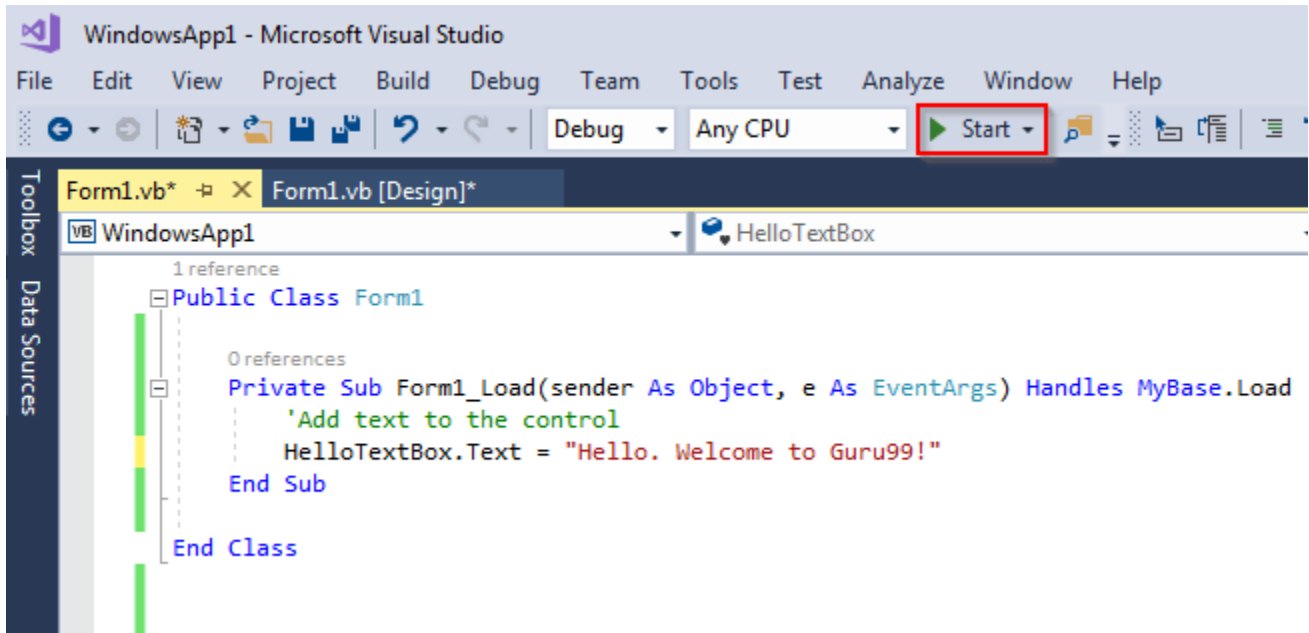


Step 3) Add the following code to add text to the control:

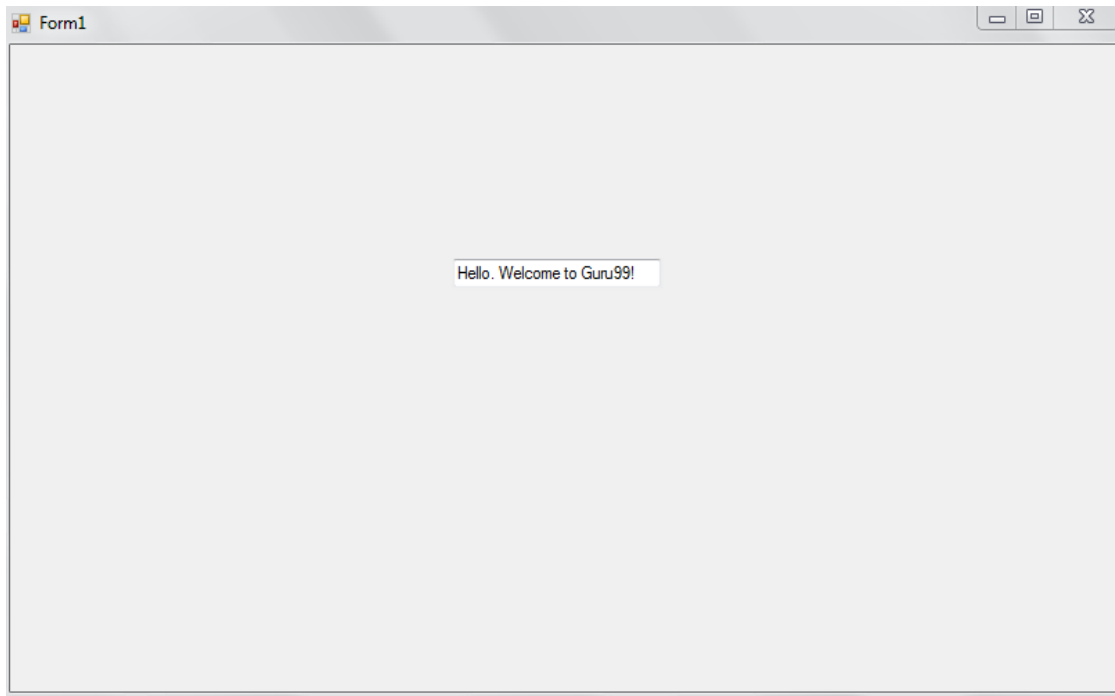
```
Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
'Add text to the control
HelloTextBox.Text = "Hello. Welcome to Guru99!"
End Sub
```

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

Visual Basic.Net Programming



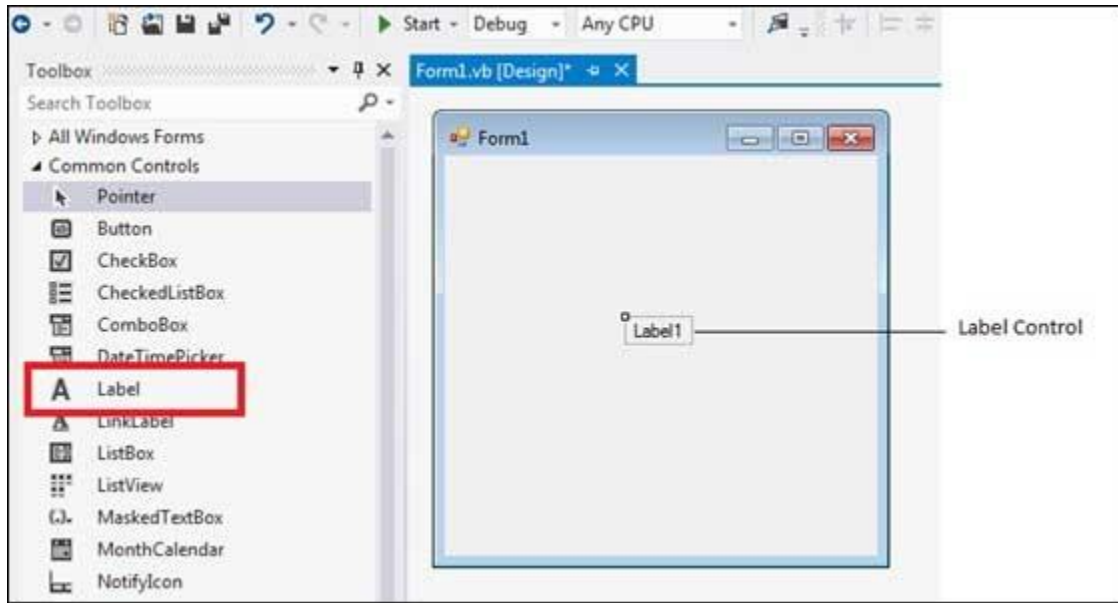
Step 5) You should get the following form:



Label Control

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

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



Properties of the Label Control

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

Sr.No. Property & Description

Autosize

- 1 Gets or sets a value specifying if the control should be automatically resized to display all its contents.

BorderStyle

- 2 Gets or sets the border style for the control.

FlatStyle

- 3 Gets or sets the flat style appearance of the Label control

Font

- 4 Gets or sets the font of the text displayed by the control.

FontHeight

5 Gets or sets the height of the font of the control.

ForeColor

6 Gets or sets the foreground color of the control.

PreferredHeight

7 Gets the preferred height of the control.

PreferredWidth

8 Gets the preferred width of the control.

TabStop

9 Gets or sets a value indicating whether the user can tab to the Label. This property is not used by this class.

Text

10 Gets or sets the text associated with this control.

TextAlign

11 Gets or sets the alignment of text in the label.

Methods of the Label Control

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

Sr.No. Method Name & Description

GetPreferredSize

1 Retrieves the size of a rectangular area into which a control can be fitted.

Refresh

2 Forces the control to invalidate its client area and immediately redraw itself and any child controls.

Select

3 Activates the control.

Show

4 Displays the control to the user.

ToString

5 Returns a String that contains the name of the control.

Events of the Label Control

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

Sr.No. Event & Description

AutoSizeChanged

1 Occurs when the value of the AutoSize property changes.

Click

2 Occurs when the control is clicked.

DoubleClick

3 Occurs when the control is double-clicked.

GotFocus

4 Occurs when the control receives focus.

Leave

5 Occurs when the input focus leaves the control.

LostFocus

6 Occurs when the control loses focus.

TabIndexChanged

7

Occurs when the TabIndex property value changes.

TabStopChanged

8 Occurs when the TabStop property changes.

TextChanged

9 Occurs when the Text property value changes.

Example

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

' Create two buttons to use as the accept and cancel buttons.
' Set window width and height
Me.Height = 300
Me.Width = 560

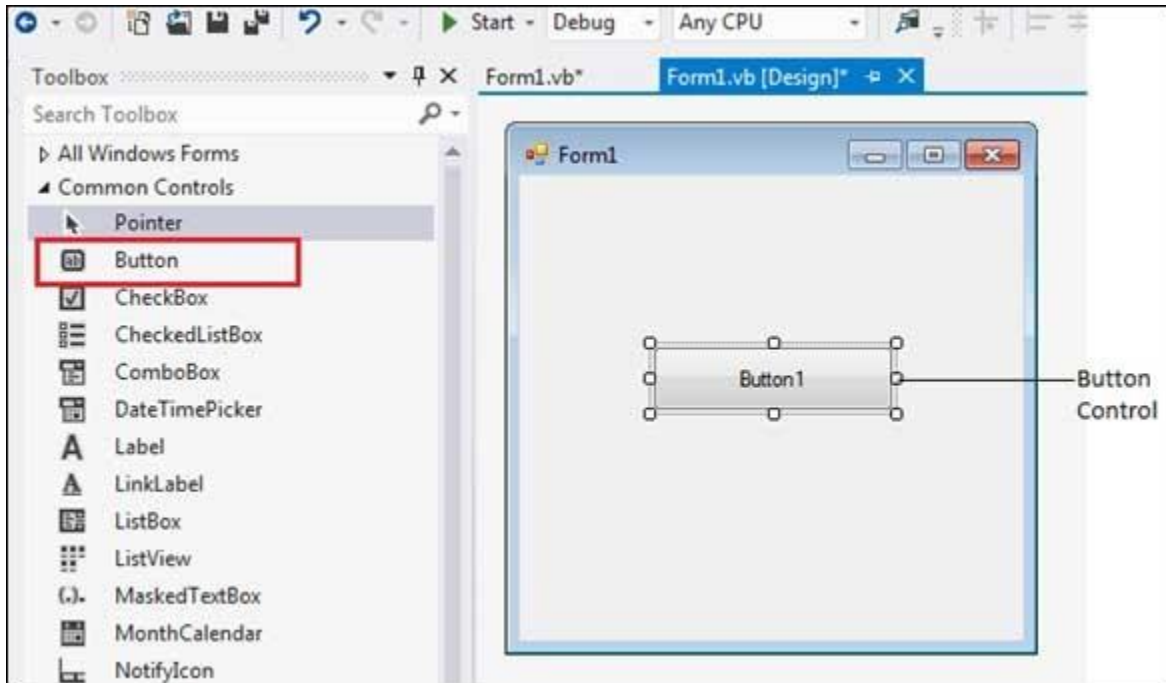
' Set the caption bar text of the form.
Me.Text = "tutorialspoint.com"
' Display a help button on the form.
Me.HelpButton = True
End Sub

Private Sub Label1_Click(sender As Object, e As EventArgs) _
Handles Label1.Click
Label1.Location = New Point(50, 50)
Label1.Text = "You have just moved the label"
End Sub
Private Sub Label1_DoubleClick(sender As Object, e As EventArgs)
Handles Label1.DoubleClick
Dim Label2 As New Label
Label2.Text = "New Label"
Label2.Location = New Point(Label1.Left, Label1.Height + _
Label1.Top + 25)
Me.Controls.Add(Label2)
End Sub
End Class
```

Button Control:

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

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



Properties of the Button Control

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

Sr.No. Property & Description

AutoSizeMode

- 1 Gets or sets the mode by which the Button automatically resizes itself.

BackColor

- 2 Gets or sets the background color of the control.

BackgroundImage

- 3 Gets or sets the background image displayed in the control.

DialogResult

- 4 Gets or sets a value that is returned to the parent form when the button is clicked. This is used

while creating dialog boxes.

ForeColor

5 Gets or sets the foreground color of the control.

Image

6 Gets or sets the image that is displayed on a button control.

Location

7 Gets or sets the coordinates of the upper-left corner of the control relative to the upper-left corner of its container.

TabIndex

8 Gets or sets the tab order of the control within its container.

Text

9 Gets or sets the text associated with this control.

Methods of the Button Control

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

Sr.No. Method Name & Description

GetPreferredSize

1 Retrieves the size of a rectangular area into which a control can be fitted.

NotifyDefault

2 Notifies the Button whether it is the default button so that it can adjust its appearance accordingly.

Select

3 Activates the control.

ToString

4

Returns a String containing the name of the Component, if any. This method should not be overridden.

Events of the Button Control

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

Sr.No. Event & Description

Click

1 Occurs when the control is clicked.

DoubleClick

2 Occurs when the user double-clicks the Button control.

GotFocus

3 Occurs when the control receives focus.

TabIndexChanged

4 Occurs when the TabIndex property value changes.

TextChanged

5 Occurs when the Text property value changes.

Validated

6 Occurs when the control is finished validating.

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

Example

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

Private Sub btnMoto_Click(sender As Object, e As EventArgs) Handles btnMoto.Click
```

Visual Basic.Net Programming

```
btnImage.Visible = False  
Label1.Text = "Simple Easy Learning"  
End Sub
```

```
Private Sub btnExit_Click(sender As Object, e As EventArgs) Handles btnExit.Click  
Application.Exit()  
End Sub
```

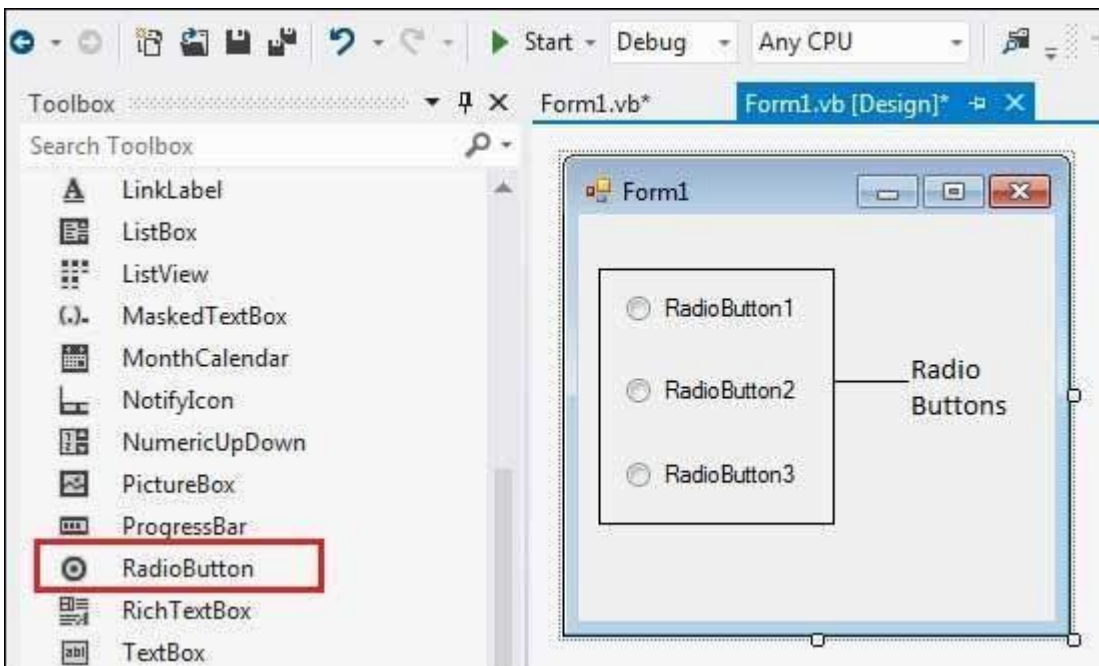
```
Private Sub btnLogo_Click(sender As Object, e As EventArgs) Handles btnLogo.Click  
Label1.Visible = False  
btnImage.Visible = True  
End Sub  
End Class
```



Unit – II
Radio Button Control

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

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



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

Properties of the RadioButton Control

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

Sr.No.	Property & Description
1	<p>Appearance</p> <p>Gets or sets a value determining the appearance of the radio button.</p>

2	<p>AutoCheck</p> <p>Gets or sets a value indicating whether the Checked value and the appearance of the control automatically change when the control is clicked.</p>
3	<p>CheckAlign</p> <p>Gets or sets the location of the check box portion of the radio button.</p>
4	<p>Checked</p> <p>Gets or sets a value indicating whether the control is checked.</p>
5	<p>Text</p> <p>Gets or sets the caption for a radio button.</p>
6	<p>TabStop</p> <p>Gets or sets a value indicating whether a user can give focus to the RadioButton control using the TAB key.</p>

Methods of the RadioButton Control

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

Sr.No.	Method Name & Description
1	<p>PerformClick</p> <p>Generates a Click event for the control, simulating a click by a user.</p>

Events of the RadioButton Control

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

Sr.No	Event & Description
1	<p>AppearanceChanged</p>

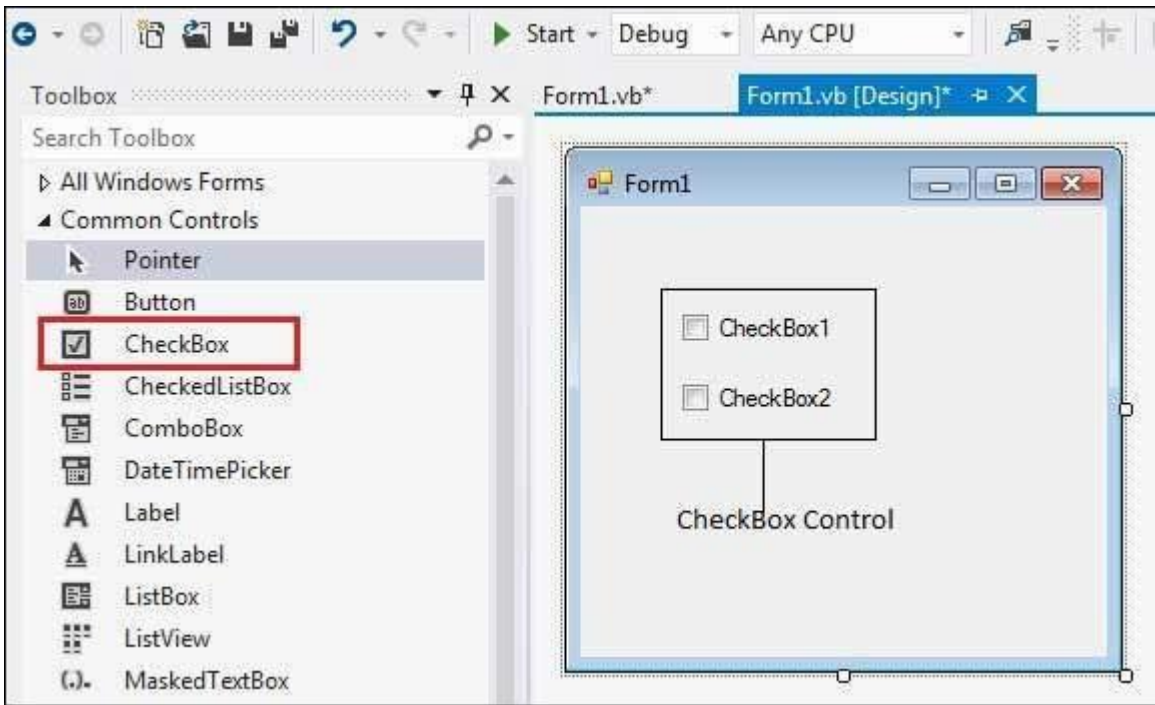
	Occurs when the value of the Appearance property of the RadioButton control is changed.
2	<p>CheckedChanged</p> <p>Occurs when the value of the Checked property of the RadioButton control is changed.</p>

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

CheckBox Control

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

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



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

Properties of the CheckBox Control

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

Sr.No.	Property & Description
1	<p>Appearance Gets or sets a value determining the appearance of the check box.</p>
2	<p>AutoCheck Gets or sets a value indicating whether the Checked or CheckState value and the appearance of the control automatically change when the check box is selected.</p>
3	<p>CheckAlign Gets or sets the horizontal and vertical alignment of the check mark on the check box.</p>
4	<p>Checked Gets or sets a value indicating whether the check box is selected.</p>
5	<p>CheckState Gets or sets the state of a check box.</p>
6	<p>Text Gets or sets the caption of a check box.</p>
7	<p>ThreeState Gets or sets a value indicating whether or not a check box should allow three check states rather than two.</p>

Methods of the CheckBox Control

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

Sr.No.	Method Name & Description
1	<p>OnCheckedChanged</p>

	Raises the CheckedChanged event.
2	OnCheckStateChanged Raises the CheckStateChanged event.
3	OnClick Raises the OnClick event.

Events of the CheckBox Control

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

Sr.No.	Event & Description
1	AppearanceChanged Occurs when the value of the Appearance property of the check box is changed.
2	CheckedChanged Occurs when the value of the Checked property of the CheckBox control is changed.
3	CheckStateChanged Occurs when the value of the CheckState property of the CheckBox control is changed.

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

GroupBox Control

GroupBox control is used to group other controls of VB.NET. GroupBox control having a frame to indicate boundary and a text to indicate header or title. Generally GroupBox control is used as a container for Radio Button. When Radio Buttons are grouped using GroupBox, user can select one RadioButton from each GroupBox.



Properties of GroupBox Control in VB.NET

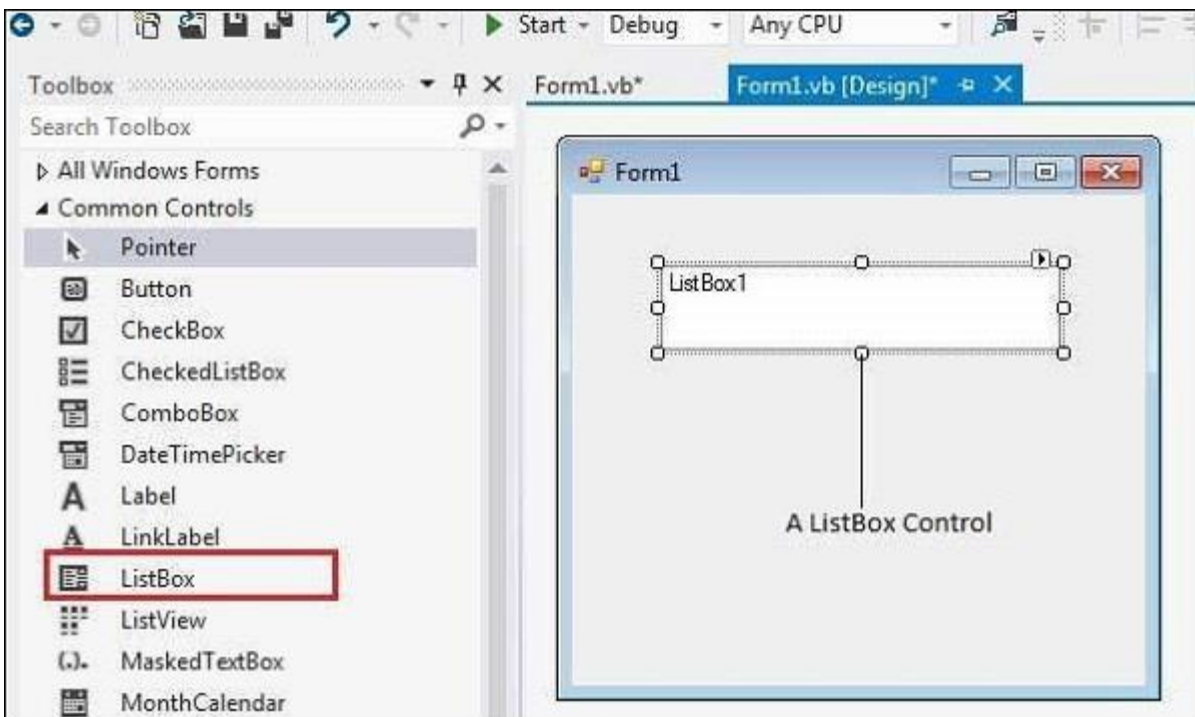
Property	Purpose
BackColor	It is used to get or set background color of the GroupBox.
BackgroundImage	It is used to get or set background Image of the GroupBox.
BackgroundImageLayout	It is used to get or set background Image layout of the GroupBox. It has one of the following values: None, Tile, Centre, Stretch, Zoom
Font	It is used to get or set font Style, Font Size, Font Face of the text contained in GroupBox Control.
ForeColor	It is used to get or set color of the text contained in GroupBox Control.
Enabled	It is used to specify weather GroupBox Control is enabled or not. It has Boolean value. Default value is true.
Visible	It is used to specify weather GroupBox Control is visible or not at run time. It has Boolean value. Default value is true.
Text	It is used to get or set Title or Header Text of the GroupBox Control.

Method	Purpose
Show	It is used to show GroupBox Control.
Hide	It is used to Hide GroupBox Control at run time.
Focus	It is used to set cursor or focus on GroupBox Control.

Listbox Control

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

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



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

Visual Basic.Net Programming

Properties of the ListBox Control

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

Sr.No.	Property & Description
1	AllowSelection Gets a value indicating whether the ListBox currently enables selection of list items.
2	BorderStyle Gets or sets the type of border drawn around the list box.
3	ColumnWidth Gets or sets the width of columns in a multicolumn list box.
4	HorizontalExtent Gets or sets the horizontal scrolling area of a list box.
5	HorizontalScrollBar Gets or sets the value indicating whether a horizontal scrollbar is displayed in the list box.
6	ItemHeight Gets or sets the height of an item in the list box.
7	Items Gets the items of the list box.
8	MultiColumn Gets or sets a value indicating whether the list box supports multiple columns.
9	ScrollAlwaysVisible Gets or sets a value indicating whether the vertical scroll bar is shown at all times.

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

	Gets or sets the index of the first visible item of a list box.
--	---

Methods of the ListBox Control

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

Sr.No.	Method Name & Description
1	<p>BeginUpdate</p> <p>Prevents the control from drawing until the EndUpdate method is called, while items are added to the ListBox one at a time.</p>
2	<p>ClearSelected</p> <p>Unselects all items in the ListBox.</p>
3	<p>EndUpdate</p> <p>Resumes drawing of a list box after it was turned off by the BeginUpdate method.</p>
4	<p>FindString</p> <p>Finds the first item in the ListBox that starts with the string specified as an argument.</p>
5	<p>FindStringExact</p> <p>Finds the first item in the ListBox that exactly matches the specified string.</p>
6	<p>GetSelected</p> <p>Returns a value indicating whether the specified item is selected.</p>
7	<p>SetSelected</p> <p>Selects or clears the selection for the specified item in a ListBox.</p>
8	<p>OnSelectedIndexChanged</p> <p>Raises the SelectedIndexChanged event.</p>

8	<p>OnSelectedValueChanged</p> <p>Raises the SelectedValueChanged event.</p>
---	--

Events of the ListBox Control

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

Sr.No.	Event & Description
1	<p>Click</p> <p>Occurs when a list box is selected.</p>
2	<p>SelectedIndexChanged</p> <p>Occurs when the SelectedIndex property of a list box is changed.</p>

CheckedListBox Control

CheckedListBox is a ListBox with Checkbox to the left side of each item in the list. It is derived from ListBox so it provides all the functionality of ListBox Control.

Properties of CheckedListBox Control in VB.NET

Property	Purpose
CheckOnClick	It is used to specify weather CheckBox should be toggled (change state) or not when an item is selected in the CheckedListBox. It has Boolean value. Default value is False.
MultiColumn	It is used to specify weather CheckedListBox supports multiple columns or not. It has Boolean value. Default value is false.
ColumnWidth	It is used to specify width of each column in MultiColumn CheckedListBox.
Items	It represents collection of items contained in CheckedListBox control.
Sorted	It is used to specify weather items of CheckedListBox are sorted in alphabetical order

	or not. It has Boolean value. Default value is false.
SelectionMode	It is used to get or set SelectionMode of CheckedListBox. It determines how user can select the Items of CheckedListBox. It can have one of the following four options: (1) None: No Selection is allowed (2) One: User can select only one item at a time. (3) MultiSimple: User can select or deselect item just by mouse click or pressing spacebar. (4) MultiExtended: User can select or deselect items by holding Ctrl key and mouse click. User can also select or deselect items by pressing Shift key and mouse click or arrow key. Default value is One.

Methods of CheckedListBox Control in VB.NET

Method	Purpose
ClearSelected	It is used to unselect all the items that are currently selected in ListBox.
FindString	It is used to find first occurrences of an item in the ListBox that partially match with string specified as an argument. If an item is found then it returns zero based index of that item, otherwise it returns -1. The search performed by this method is case insensitive.
FindStringExact	It is used to find first occurrences of an item in the ListBox that exactly match with string specified as an argument. If an item is found then it returns zero based index of that item, otherwise it returns -1. The search performed by this method is case insensitive.
GetSelected	It is used to determine whether an item whose index is passed as an argument is selected or not. It returns Boolean value.
SetSelected	It is used to select or deselect an item whose index is passed as an argument. Example: ListBox1.SetSelected (1, true) will select second item of ListBox.
ClearSelected	It is used to unselect all items in CheckedListBox.
GetItemChecked	It is used to check whether an item whose index is passed as an argument is checked or not. It returns Boolean value.

Visual Basic.Net Programming

GetItemCheckState	It is used to get check state of an item whose index is passed as an argument. It returns 1 if item is checked otherwise false.	37
SetItemCheckState	It is used to set the check state of an item whose index is passed as an argument. Example: CheckedListBox1.SetItemChecked (1, CheckState.Checked)) will check the second item of CheckedListBox.	

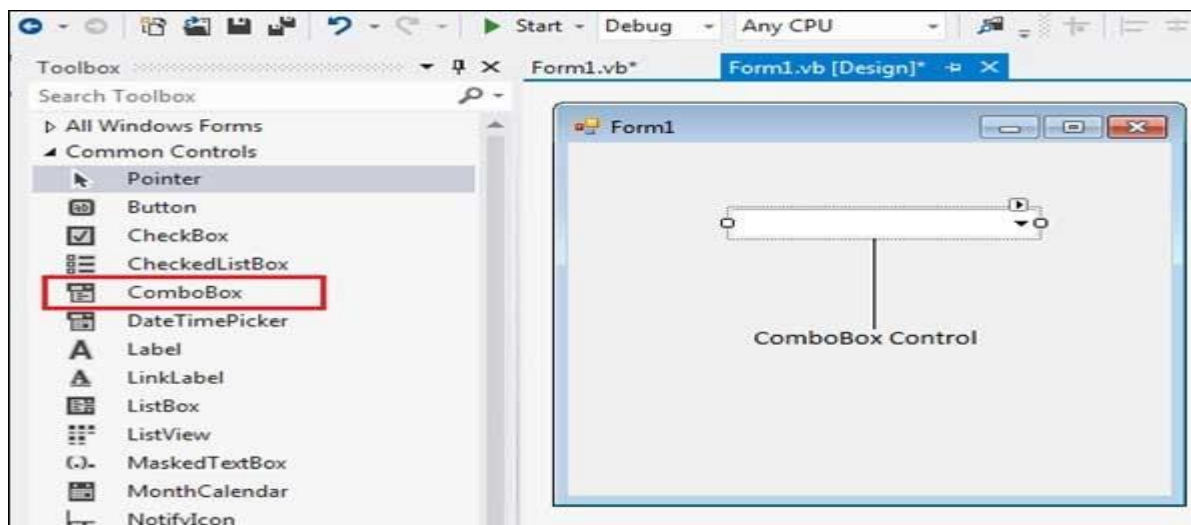
Events of CheckedListBox Control in VB.NET

Event	Purpose
SelectedIndexChanged	It is the default event of ListBox Control. It fires each time a selected Item in the ListBox is changed.
ItemCheck	It fires each time an item is checked or unchecked.

ComboBox Control

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

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



Visual Basic.Net Programming

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

Properties of the ComboBox Control

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

Sr.No.	Property & Description
1	AllowSelection Gets a value indicating whether the list enables selection of list items.
2	AutoCompleteCustomSource Gets or sets a custom System.Collections.Specialized.StringCollection to use when the AutoCompleteSourceproperty is set to CustomSource.
3	AutoCompleteMode Gets or sets an option that controls how automatic completion works for the ComboBox.
4	AutoCompleteSource Gets or sets a value specifying the source of complete strings used for automatic completion.
5	DataBindings Gets the data bindings for the control.
6	DataManager Gets the CurrencyManager associated with this control.
7	DataSource Gets or sets the data source for this ComboBox.

8	DropDownHeight Gets or sets the height in pixels of the drop-down portion of the ComboBox.
9	DropDownStyle Gets or sets a value specifying the style of the combo box.
10	DropDownWidth Gets or sets the width of the of the drop-down portion of a combo box.

Methods of the ComboBox Control

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

Sr.No.	Method Name & Description
1	BeginUpdate Prevents the control from drawing until the EndUpdate method is called, while items are added to the combo box one at a time.
2	EndUpdate Resumes drawing of a combo box, after it was turned off by the BeginUpdate method.
3	FindString Finds the first item in the combo box that starts with the string specified as an argument.
4	FindStringExact Finds the first item in the combo box that exactly matches the specified string.
5	SelectAll Selects all the text in the editable area of the combo box.

Events of the ComboBox Control

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

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

If...Then Statement

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

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

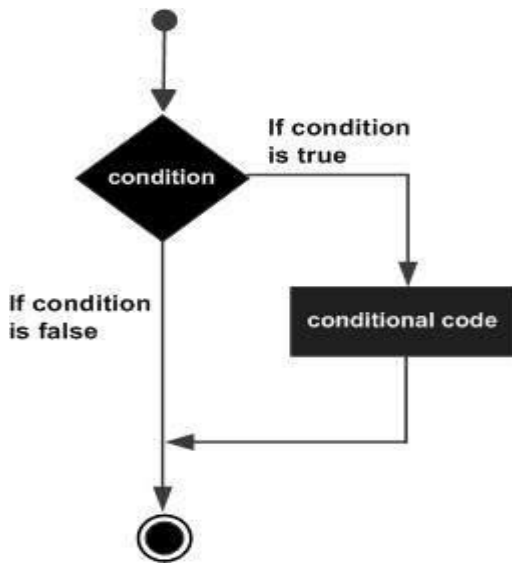
Where, *condition* is a Boolean or relational condition and Statement(s) is a simple or compound statement. Example of an If-Then statement is –

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


Visual Basic.Net Programming

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

Flow Diagram



Example

Module decisions

```
Sub Main()  
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
```

Output:

a is less than 20
value of a is : 10

Select Case Statement

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

Visual Basic.Net Programming

Syntax

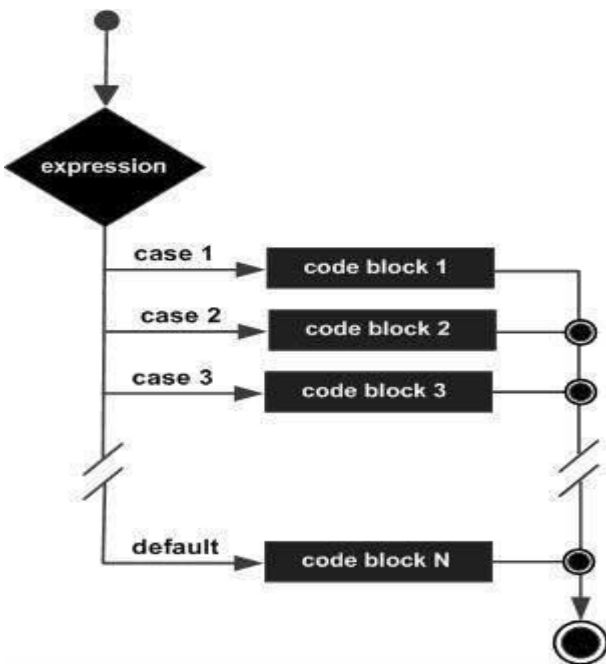
The syntax for a Select Case statement in VB.Net is as follows

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

Where,

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

Flow Diagram



Example:

```
Module decisions
Sub Main()
'local variable definition
Dim grade As Char
grade = "B"
Select grade
Case "A"
Console.WriteLine("Excellent!")
Case "B", "C"
Console.WriteLine("Well done")
Case "D"
Console.WriteLine("You passed")
Case "F"
Console.WriteLine("Better try again")
Case Else
Console.WriteLine("Invalid grade")
End Select
Console.WriteLine("Your grade is {0}", grade)
Console.ReadLine()
End Sub
End Module
```

Output:

Well done
Your grade is B

While... End While Loop

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

The syntax for this loop construct is –

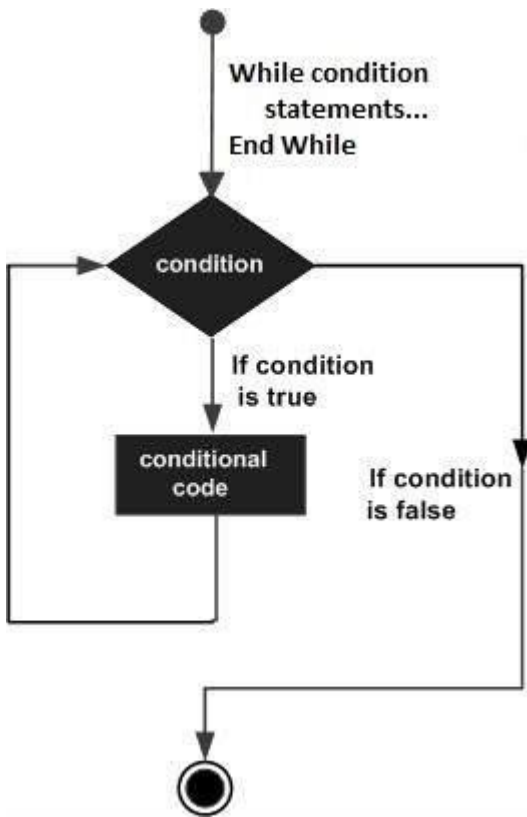
```
While condition
[ statements ]
[ Continue While ]
[ statements ]
[ Exit While ]
[ statements ]
End While
```

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

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

Visual Basic.Net Programming

Flow Diagram



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

Example

Module loops

```
Sub Main()
```

```
Dim a As Integer = 10
```

```
' while loop execution '
```

```
While a < 20
```

```
Console.WriteLine("value of a: {0}", a)
```

```
a = a + 1
```

```
End While
```

```
Console.ReadLine()
```

```
End Sub
```

Output:

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

Do Loop

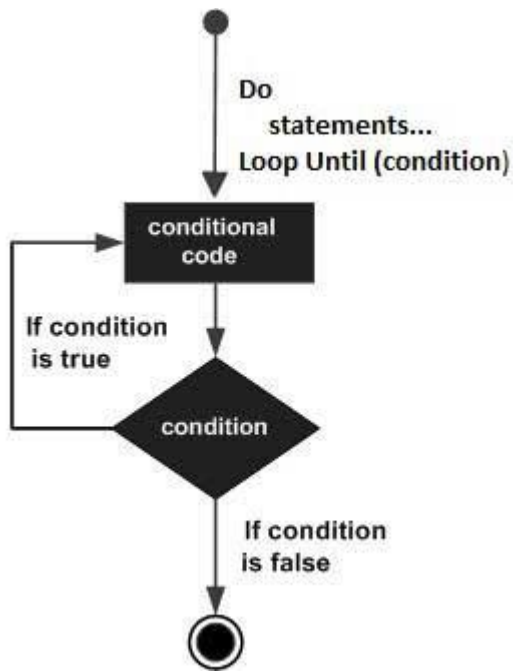
It repeats the enclosed block of statements while a Boolean condition is True or until the condition becomes True. It could be terminated at any time with the Exit Do statement.

The syntax for this loop construct is –

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

Visual Basic.Net Programming

Flow Diagram



Example

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

Output:

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

value of a: 19

For...Next Loop

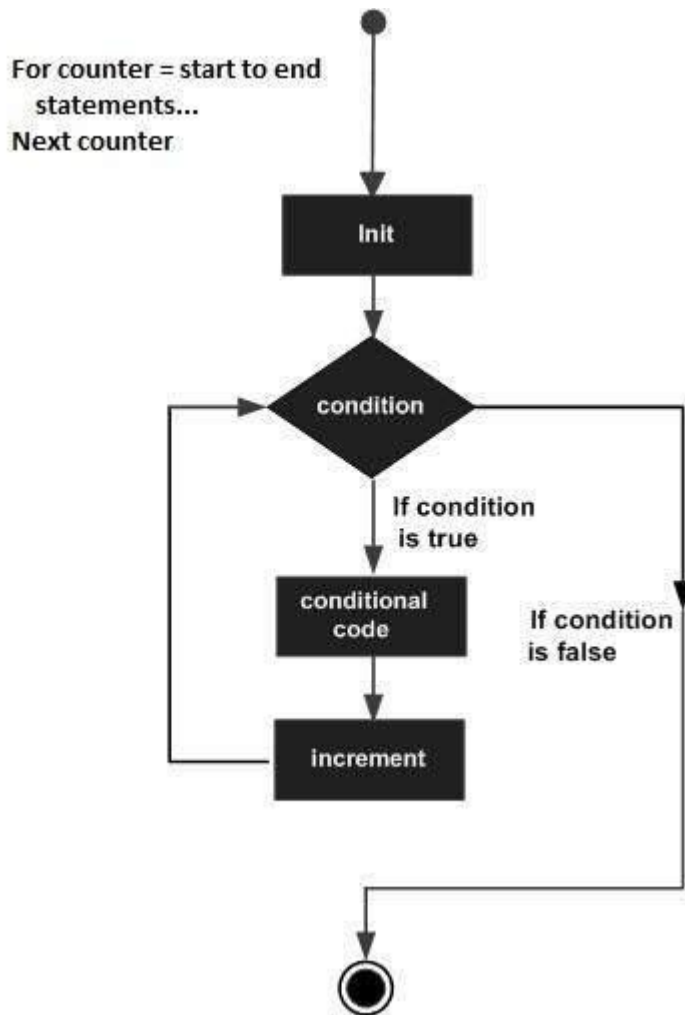
It repeats a group of statements a specified number of times and a loop index counts the number of loop iterations as the loop executes.

The syntax for this loop construct is –

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

Visual Basic.Net Programming

Flow Diagram



Example

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


Output:

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
value of a: 20

Types of Methods

Methods

Methods are simply member procedures built into the class. In Visual Basic .NET there are two types of methods Functions and Sub Procedures. Methods help us to handle code in a simple and organized fashion. Functions return a value, but Sub Procedures does not return any value. Methods are basically a series of statements that are executed when called. Detail explanation of Sub Procedures and Functions are given below:

Sub Procedures

In Visual Basic .NET Sub Procedures are the statements enclosed by the Sub and End Sub statements. Statements are executed when we call the Sub procedure. The statements within it are executed until the matching End Sub is not found. A **Sub** procedure performs actions but does not return a value. The starting point of the program Sub Main(), it is also a sub procedure. The control is transferred to Sub Main() Sub procedure automatically when application start execution.

Example:

```
Imports System.Console  
Module Module1
```

```
    Sub Main()  
        'sub procedure Main() is called by default  
        Show()  
        'sub procedure Show() which we are creating  
        Read()  
    End Sub
```

```
Sub Show()  
    Write("This is Sub Procedures")  
    'executing sub procedure Show()  
End Sub  
End Module
```

The output of above code is given below:

This is Sub Procedures

Functions

Functions are just like Sub procedures except that they can return a value. When we perform some action on data like evaluate data, calculations or to transform data then we use function. We can declare functions like Sub Procedures except that we have to use the Function keyword instead of Sub.

Example:

```
Imports System.Console  
Module Module1  
  
    Sub Main()  
        Write("Sum of two integer is" & " " & Sum())  
        'calling the function  
        Read()  
    End Sub  
  
    Public Function Sum() As Integer  
        'declaring a function Sum  
        Dim A, B As Integer  
        'declaring two integers and assigning values to them  
        A = 20  
        B = 40  
        Return (A + B)  
        'performing the addition of two integers and returning it's value  
    End Function  
  
End Module
```

End Module

The output of above code is given below:

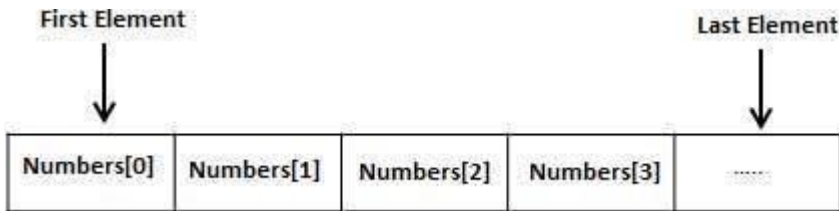
Sum of two integer is 60

One Dimensional Array

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

Visual Basic.Net Programming

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



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

```
Dim intData(30) ' an array of 31 elements
Dim strData(20)
```

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

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

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

Module arrayApl

Sub Main()

```
Dim n(10) As Integer ' n is an array of 11 integers '
```

```
Dim i, j As Integer
```

```
' initialize elements of array n '
```

```
For i = 0 To 10
```

```
n(i) = i + 100 ' set element at location i to i + 100
```

```
Next i
```

```
' output each array element's value '
```

```
For j = 0 To 10
```

```
Console.WriteLine("Element({0}) = {1}", j, n(j))
```

```
Next j
```

```
Console.ReadKey()
```

End Sub

End Module

Output:

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

Multi-Dimensional Arrays

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

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

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

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

```
Module arrayApl
```

```
Sub Main()
```

```
' an array with 5 rows and 2 columns
```

```
Dim a(.) As Integer = {{0, 0}, {1, 2}, {2, 4}, {3, 6}, {4, 8}}
```

```
Dim i, j As Integer
```

```
' output each array element's value '
```

```
For i = 0 To 4
```

```
For j = 0 To 1
```

```
Console.WriteLine("a[{0},{1}] = {2}", i, j, a(i, j))
```

```
Next j
```

```
Next i
```

```
Console.ReadKey()
```

```
End Sub
```

```
End Module
```

Output:

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

Jagged Array

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

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

The following example illustrates using a jagged array –

```
Module arrayApl
```

```
Sub Main()
```

```
'a jagged array of 5 array of integers
```

```
Dim a As Integer()() = New Integer(4)() { }
```

```
a(0) = New Integer() {0, 0}
```

```
a(1) = New Integer() {1, 2}
```

```
a(2) = New Integer() {2, 4}
```

```
a(3) = New Integer() {3, 6}
```

Visual Basic.Net Programming

```
a(4) = New Integer() {4, 8}
```

```
Dim i, j As Integer
```

```
' output each array element's value
```

```
For i = 0 To 4
```

```
For j = 0 To 1
```

```
Console.WriteLine("a[{0},{1}] = {2}", i, j, a(i)(j))
```

```
Next j
```

```
Next i
```

```
Console.ReadKey()
```

```
End Sub
```

```
End Module
```

Output:

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

Class Definition and Usage

A class definition starts with the keyword `Class` followed by the class name; and the class body, ended by the `End Class` statement.

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

Following is the general form of a class definition

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

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

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

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

```
'volume of box 1
volume = Box1.height * Box1.length * Box1.breadth
Console.WriteLine("Volume of Box1 : {0}", volume)

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

Output:

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

Instance and Shared Members of a Class

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

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

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

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

The following example demonstrates the use of shared members –

Class StaticVar

```
Public Shared num As Integer
```

```
Public Sub count()
```

```
num = num + 1
```

```
End Sub
```

```
Public Shared Function getNum() As Integer
```

```
Return num
```


End Function

Shared Sub Main()

Dim s As StaticVar = New StaticVar()

s.count()

s.count()

s.count()

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

Console.ReadKey()

End Sub

End Class

Output:

Value of variable num: 3

Constructor overloading

When the same method name is used for more than one method, with different types of parameters and returned types, then the method is said to be overloaded. Constructor is a special method called 'New()' in vb.net and is defined as a Sub.

Overloading feature is used most frequently to overload the constructor. We overload the constructor by defining more than one 'Sub New()' procedure. By overloading a constructor, we make available more than one constructor. So, while creating an object we can choose which constructor we want to use to instantiate the object.

Example:

```
Public Class Account
Private mCode As String
Private mName As String
Private mdescription As String
Protected mBalance As Double
```

Constructor1: Constructor to initialize all the member variables.

Visual Basic.Net Programming

```
Public Sub New(ByVal code, ByVal name,
ByVal description, ByVal balance)
mCode = code
mName = name
mdescription = description
mBalance = balance
End Sub
```

```
Constructor2:
Public Sub New()
End Sub
```

```
Public Class AccountForm
Private Sub OkButton_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles OkButton.Click
'Instantiate the Account Class Object by using the Constructor1
Dim myPartyAcc As Account = New
Account(txtCode.Text, txtName.Text,
txtdescription.Text, txtBalance.Text)
MessageBox.Show(myPartyAcc.Code)
MessageBox.Show(myPartyAcc.Name)
MessageBox.Show(myPartyAcc.description)
End Sub
End Class
```

Copy Constructors

A *copy constructor* creates a new object by copying variables from an existing object of the same type. For example, you might want to pass a Time object to a Time constructor so that the new Time object has the same values as the old one.

VB.NET does not provide a copy constructor, so if you want one you must provide it yourself. Such a constructor copies the elements from the original object into the new one:

Example:

```
Public Sub New(ByVal existingObject As Time)
year = existingObject.Year
month = existingObject.Month
date = existingObject.Date
hour = existingObject.Hour
minute = existingObject.Minute
second = existingObject.Second
End Sub
```

A copy constructor is invoked by instantiating an object of type Time and passing it the name of the Time object to be copied

Dim t2 As New Time(existingObject)

Here an existing Time object (existingObject) is passed as a parameter to the copy constructor that will create a new Time object ()

Shared constructor

The class could also have a Shared constructor. A Shared constructor is called when the application starts and the class is being registered for use. It can be used for initialization, calculations and so on. Similarly to instance constructors, we are able to create instances of classes and store them into Shared fields in Shared constructors.

Example:

```
Public Class YourClass
Private Shared ID as Integer = 10

Public Shared ReadOnly Property CurrentID as Integer
Get
Return ID
End Get
End Property

Public Shared Function GetID() as Integer
ID += 1
Return ID
End Function

Shared Sub New()
Console.WriteLine("Before init: " & ID)
ID = 100
Console.WriteLine("After init: " & ID)
End Sub
End Class

Module Test
Sub Main()
Dim CountValue As Integer
For CountValue = 1 to 10
```

Visual Basic.Net Programming

```
Console.WriteLine(YourClass.GetID())  
Next  
End Sub  
End Module
```

60

Output:

```
Before init: 10  
After init: 100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110
```

Unit-III

Inheritance and Polymorphism

Virtual Methods

A virtual method is a declared class method that allows overriding by a method with the same derived class signature. Virtual methods are tools used to implement the polymorphism feature of an object-oriented language, such as C#. When a virtual object instance method is invoked, the method to be called is determined based on the object's runtime type, which is usually that of the most derived class. A virtual method is used to override specified base class implementation when a runtime object is of the derived type. Thus, virtual methods facilitate the consistent functionality of a related object set. Virtual method implementation differs in programming languages like C++, Java, C# and Visual Basic .NET. In Java, all non-static methods are virtual by default, with the exception of methods that are private or marked with the keyword final. C# requires the keyword virtual for virtual methods, with the exception of private, static and abstract methods, and the keyword override for overriding the derived class method.

A pure virtual method is a virtual method that mandates a derived class to implement a method and does not allow instantiation of the base class, or abstract class.

Abstract Class and methods

Abstract class is a special kind of class that cannot be instantiated. It only allows other classes to inherit from it but cannot be instantiated. These are the important point which are related to the abstract class.

1. abstract class may contain concrete methods.
2. class may contain non-public members.
3. abstract class can be used as a single inheritance.
4. abstract class can be invoked if a main() exists.

It only allows other classes to inherit from it but cannot be instantiated. The advantage is that it enforces certain hierarchies for all the subclasses. In other word, it is a kind of contract that forces all the subclasses to carry on the same hierarchies or standards.

Creating a abstract class

we use **MustInherit** keyword to create abstract class. Abstract classes can also specify abstract members. Like abstract classes, abstract members also provide no details regarding their implementation. Only the member type, access level, required parameters and return type are specified. and to declare the abstract member we use the **MustOverride** keyword.

Example

```
Imports System.Console
Imports System.Math
Module Module1
Public MustInherit Class Abstractclass
Public MustOverride Function square() As Integer
Public MustOverride Function cube() As Integer
End Class
Public Class AbstractFirst
Inherits AbstractClass
Dim A As Integer = 4
Dim B As Integer = 5
Public Overrides Function square() As Integer
Return A * A
```

Visual Basic.Net Programming

```
End Function

Public Overrides Function cube() As Integer

Return B * B * B

End Function

End Class

Sub Main()

Dim abs1 As New AbstractFirst()

WriteLine("square of A is :" & " " & abs1.square())
WriteLine("Cube of B is :" & " " & abs1.cube())
Read()
End Sub
End Module
```

Output:

square of A is :16

Cube of B is :125

Sealed Class

A sealed class is a class the does not allow inheritance . Means you cannot inherit the sealed class .
In VB.NET sealed class is represented as Non Inheritable class.

Definition of Interfaces

The *interface* is a set of definitions of properties, methods and events. Unlike classes [...], interfaces do not contain the implementation. Interfaces are implemented [...] by classes, but are defined as separate entities.

A class that implement an interface must implement all elements defined in that interface.

Defining a VB.NET interface is achieved using the specification **Interface** and interface implementation using specification **Implements**.

Visual Basic.Net Programming

The definition of an interface can retrieve specifications defined in other interfaces. Inheritance of elements defined in other components are implemented in VB.NET language mentioning specification **Inherits**. 63

In a *namespace* [...], interfaces have associated the modifier **Friend** implicitly, and interfaces defined in classes, modules, interfaces and structures have associated modifier **Public**. Interface is a powerful tool of programming, because the objects definition and the implementation are separated. Cases in which it is recommended to define interfaces:

- Classes with high orthogonality: small-scale implementation of inheritance to define classes;
- High flexibility: a class can implement multiple interfaces;
- Implementation inheritance is not desired from a base class;
- Inheritance cannot be used: structures [...] cannot inherit classes, but can implement interfaces.

Example

```
Interface IOperatii
Event Calcul(ByVal x As Integer, ByVal y As Integer)
Function OpDiferenta(ByVal a As Integer, ByVal b As Integer) As Integer
Function OpProdus(ByVal a As Integer, ByVal b As Integer) As Long
End Interface
```

Implementation of multiple interfaces

We will see how to implement of multiple interface in vb.net an interface can obtain one or more methods,properties,indexers and events.

But none of them are implemented in the interface itself.it is the responsibility of the class that implements the interface to define the code for implementation of these membersvb.net does not support directly multiple inheritance but using interface we can use multiple inheritance. interface keywords to create an interface& implement keywords is use to implement the interface. This code implementing two interfaces.The class Computation implement two interfaces Addition and Multiplication.it declare two data members and define the code for the methods Add and multiplication.

Example:

```
Module Module1
Interface Addition
Function Add() As Integer
End Interface
Interface Multiplication
Function Mul() As Integer
```

Visual Basic.Net Programming

```
End Interface
Class Computation
Inherits Addition
Inherits Multiplication
Private x As Integer, y As Integer
Public Sub New(ByVal x As Integer, ByVal y As Integer)
Me.x = x
Me.y = y
End Sub
Public Function Add() As Integer
Return (x + y)
End Function
Public Function Mul() As Integer
Return (x * y)
End Function
End Class
Class interfaceTest1
Sub Main()
Dim com As New Computation(10, 20)
Dim add As Addition = DirectCast(com, Addition)
Console.WriteLine("sum =" & add.Add())
Dim mul As Multiplication = DirectCast(com, Multiplication)
Console.WriteLine("product =+" & mul.Mul())
End Sub
End Class
```


Interface Inheritance

- If an interface uses the Inherits statement, you can specify one or more base interfaces. You can inherit from two interfaces even if they each define a member with the same name. If you do so, the implementing code must use name qualification to specify which member it is implementing.

An interface cannot inherit from another interface with a more restrictive access level. For example, a Public interface cannot inherit from a Friend interface.

An interface cannot inherit from an interface nested within it.

An example of interface inheritance in the .NET Framework is the ICollection interface, which inherits from the IEnumerable interface. This causes ICollection to inherit the definition of the enumerator required to traverse a collection.

Example

```
Public Interface thisInterface
Inherits IComparable, IDisposable, IFormattable
' Add new property, procedure, and event definitions.
End Interface
```

Namespaces

The most common way VB.NET namespaces are used by most programmers is to tell the compiler which .NET Framework libraries are needed for a particular program.

For example, some of the namespaces and the actual files they are in for a Windows Forms Application are:

```
System > in System.dll
System.Data > in System.Data.dll
System.Deployment > System.Deployment.dll
System.Drawing > System.Drawing.dll
System.Windows.Forms > System.Windows.Forms.dll
```

You can see (and change) the namespaces and references for your project in the project properties under the **References** tab.

This way of thinking about namespaces makes them seem to be just the same thing as "code library" but that's only part of the idea. The real benefit of namespaces is organization.

Visual Basic.Net Programming

Namespaces make it possible to organize the tens of thousands of .NET Framework objects and all the objects that VB programmers create in projects, too, so they don't clash.

For example, if you search .NET for a **Color** object, you find two. There is a **Color** object in both:

System.Drawing

System.Windows.Media

If you add an **Imports** statement for both namespaces

Imports System.Drawing

Imports System.Windows.Media

VB.NET uses the name of your project (**WindowsApplication1** for a standard forms application if you don't change it) as the default namespace.

Access Modifiers

AccessSpecifiers describes as the scope of accessibility of an Object and its members. We can control the scope of the member object of a class using access specifiers. We are using access specifiers for providing security of our applications.

Visual Basic .Net provide five access specifiers , they are as follows :

- Public
- Private
- Protected
- Friend
- ProtectedFriend

Public :

Public is the most common access specifier. It can be access from anywhere, hat means there is no restriction on accessibility. The scope of the accessibility is inside class also in outside the class.

Private :

The scope of the accessibility is limited only inside the classes in which they are declared. The Private members can not be accessed outside the class and it is the least permissive access level.

Protected :

The scope of accessibility is limited within the class and the classes derived (Inherited)from this class.

Friend :

The Friend access specifier can access within the program that contain its declarations and also access within the same assembly level. You can use friend instead of Dim keyword.

ProtectedFriend :

ProtectedFriend is same access levels of both Protected and Friend. It can access anywhere in the same assembly and in the same class also the classes inherited from the same class .

Syntax:

```
Public Class SomeClass

Public Sub DoSomething( )
' ...
End Sub

Private Sub InternalHelperSub( )
' ...
End Sub

End Class
```

Delegates

A delegate is a type that safely encapsulates a method, similar to a function pointer in C and C++. Unlike C function pointers, delegates are object-oriented, type safe, and secure. The type of a delegate is defined by the name of the delegate.

Example:

```
public delegate sub myDlg()
```

Then we use the delegate by simply declaring a variable of the delegate and assigning the sub or function to run when called.

```
Private Sub message()
Console.WriteLine ( "show message" )
End Sub
```

now it matches our declaration of MyDlg. it's a sub routine with no parameters. And then our test code:

Visual Basic.Net Programming

```
Dim dlg As mydel
dlg = New mydel(AddressOf message)
dlg.Invoke()
```

When we invoke the delegate, the message sub is run.

Code:

```
Module Module1
```

```
Public Delegate Sub mydel()
```

```
Public Delegate Sub mydel1(ByVal a As Integer)
```

```
Public Sub message()
```

```
Console.WriteLine("show message")
```

```
End Sub
```

```
Public Sub add(ByVal a As Integer)
```

```
Dim b As Integer
```

```
b = a + a
```

```
Console.Write(" Addition is : ")
```

```
Console.WriteLine(b)
```

```
End Sub
```

```
Sub Main()
```

```
Dim dlg As mydel
```

```
dlg = New mydel(AddressOf message)
```

```
dlg.Invoke()
```

```
Dim dlg1 As mydel1
```

```
dlg1 = New mydel1(AddressOf add)
```

```
dlg1.Invoke(10)
```

End Sub

End Module

OUTPUT:

show message

Addition is :20

Event

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

Visual Basic.Net Programming

- **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 **EventArgs**. This object has the following properties –

- **Alt** – it indicates whether the ALT key is pressed
- **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 **EventArgs**. This object has the following properties –

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

Attributes

Attributes are declarative tags that can be used to annotate types or class members, thereby modifying their meaning or customizing their behavior. This descriptive information provided by the attribute is stored as metadata in a .NET assembly and can be extracted either at design time or at runtime using reflection.

To see how attributes might be used, consider the <WebMethod> attribute, which might appear in code as follows:

```
<WebMethod(Description:="Indicates the number of visitors to a page")> _
```

Ordinarily, public methods of a class can be invoked locally from an instance of that class; they are not treated as members of a web service. In contrast, the <WebMethod> attribute marks a method as a function callable over the Internet as part of a web service. This <WebMethod> attribute also includes a single property, Description, which provides the text that will appear in the page describing the web service.

You may wonder why attributes are used on the .NET platform and why they are not simply implemented as language elements. The answer comes from the fact that attributes are stored as metadata in an assembly, rather than as part of its executable code. As an item of metadata, the attribute describes the program element to which it applies and is available through reflection both at design time (if a graphical environment such as Visual Studio .NET is used), at compile time

Reflection

Without adding reference using classLibrary methods at runtime is done through Reflection. In Reflection we have to use an Activator class. An Activator class is a class, which creates the instance of class method at runtime

The generic terms of Reflection are:

- **Assembly:** Which hold the dll of classLibrary.
- **Type:** Hold the class of classLibrary.
- **MethodInfo:** Hold the method of class.
- **Parameterinfo:** Keep the parameter information of Method.

Here I made a class Library, in this class library I made two classes and some methods in these classes. After making this, build this class Library. On building the class Library a dll will generate.

Example:

```
Imports System
Imports System.Collections.Generic
Imports System.Text
Imports System.Windows.Forms
```

```
Namespace reflectionclass
Public Class xx
Public Function sum(ByVal a As Integer, ByVal b As Integer) As Integer
Return a + b
End Function
Public Function [sub](ByVal a As Integer, ByVal b As Integer) As Integer
Return a - b
End Function
End Class
```

```
Public Class yy
Public Function mul(ByVal a As Integer, ByVal b As Integer) As Integer
Return a * b
End Function
End Class
End Namespace
```

Unit-IV Exception Handling

Default Exception Handling :

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

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

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

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

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

Example:

```
Module exceptionProg
Sub division(ByVal num1 As Integer, ByVal num2 As Integer)
Dim result As Integer
```



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

Output:

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

User-Defined Exceptions

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

Example:

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

Visual Basic.Net Programming

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

Output:

TempIsZeroException: Zero Temperature found

Throw Statement

You can throw an object if it is either directly or indirectly derived from the System.Exception class.

You can use a throw statement in the catch block to throw the present object as –

Syntax:

```
Throw [ expression ]
```

Visual Basic.Net Programming

Example:

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

Output:

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

Custom Exceptions

Visual Basic .NET offers structured exception handling that provides a powerful, more readable alternative to "On Error Goto" error handling, which is available in previous versions of Microsoft Visual Basic.

In VB.Net we can handle exceptions with great ease and we can also create our own customized exceptions which can later be used for our applications specific needs. Exceptions are objects that encapsulate an irregular circumstance, such as when an application is out of memory, a file that cannot be opened, or an attempted illegal cast.

You can also throw an exception from within your own code using the keyword Throw.

Example:

```
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 j As Integer
Dim k As Integer
j = 10
k = 0
i = j / k
Catch ex As Exception
Throw (New MyCustomException("You can not divide a number by zero"))
End Try

End Sub
End Class
```

```
Public Class MyCustomException
Inherits System.ApplicationException
```

```
Public Sub New(ByVal message As String)
MyBase.New(message)
MsgBox(message)
End Sub
```

```
End Class
```

Usage of a Thread

A thread is a path of execution within a process. A process can contain multiple threads.

- 1. Responsiveness:* If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.
- 2. Faster context switch:* Context switch time between threads is lower compared to process context switch. Process context switching requires more overhead from the CPU.
- 3. Effective utilization of multiprocessor system:* If we have multiple threads in a single process, then we can schedule multiple threads on multiple processor. This will make process execution faster.
- 4. Resource sharing:* Resources like code, data, and files can be shared among all threads within a process.

Note: stack and registers can't be shared among the threads. Each thread has its own stack and registers.

- 5. Communication:* Communication between multiple threads is easier, as the threads shares common address space. while in process we have to follow some specific communication technique for communication between two process.

6. *Enhanced throughput of the system:* If a process is divided into multiple threads, and each thread function is considered as one job, then the number of jobs completed per unit of time is increased, thus increasing the throughput of the system.

Thread Class

Any application makes use of threads to either show you information or a screen; or to allow for long running tasks to complete. Now, threads are tricky and data is tricky. Sometimes you will encounter a situation that may possibly freeze your application due to a long running task. So, instead of slamming all the code into one form, you have several threads. Each thread can run **independently** without affecting any other threads or user process.

Example:

```
Imports System.Threading 'Imports Threading Namespace
Dim F2 As New frmThread2 'Create New Form 2 Object
Dim strText As String = "Thread Is Running!" 'Text To Display
Dim lbList As New ListBox 'Create New ListBox, Used By Form 2
```

```
Private Sub frmThread1_Load(sender As Object, e As EventArgs) Handles Me.Load
```

```
'Create Thread, and Specify Delegate
```

```
Dim tThread1 As New Thread(AddressOf ThreadProcedure)
```

```
'Start A Thread
```

```
tThread1.Start()
```

```
'Show Form 2
```

```
F2.Show()
```

```
F2.TopMost = True
```

```
End Sub
```

Start(), Abort(), Join(), Sleep() Methods

Once you create a new thread object, you must explicitly call its Start() method to have it actually execute the thread method.

Calling the **Start()** method is a non-blocking operation, meaning that control returns immediately to the client that started the thread, even though it may be some time later until the new thread actually starts. As a result, do not make any assumptions in your code that the thread is actually running.

Syntax:

Thread.Start()

Sleep():

The **Thread** class provides two overloaded versions of the static **Sleep()** method, used to put a thread to sleep for a specified timeout:

Because **Sleep()** is a static method, you can only put your own thread to sleep:

Thread.Sleep(20)

Sleep() is a blocking call, meaning that control returns to the calling thread only after the sleep period has elapsed. **Sleep()** puts the thread in a special queue of threads waiting to be awakened by the operating system.

Joining a Thread

The Thread class provides the **Join()** method, which allows one thread to wait for another thread to terminate. Any client that has a reference to a Thread object can call **Join()**, and have the client thread blocked until the thread terminates. Note that you should always check before calling **Join()** that the thread you are trying to join to is not your current thread:

Syntax:

thread.Join()

When you specify a timeout, **Join()** will return when the timeout has expired or when the thread is terminated, whichever happens first.

Aborting a Thread

The Thread class provides an **Abort()** method, intended to forcefully terminate a .NET thread. Calling **Abort()** throws an exception of type **ThreadAbortException** in the thread being aborted.

Syntax:

Thread.Abort()

Suspending and Resuming a Thread

Visual Basic.Net Programming

The **Thread** class provides the **Suspend()** method, used to suspend the execution of a thread, and the **Resume()** method, used to resume a suspended thread:

79

Anybody can call **Suspend()** on a Thread object, including objects running on that thread, and there is no harm in calling **Suspend()** on an already suspended thread. Obviously, only clients on other threads can resume a suspended thread. **Suspend()** is a non-blocking call, meaning that control returns immediately to the caller, and the thread is suspended later, usually at the next safe point.

A *safe point* is a point in the code safe for garbage collection. When garbage collection takes place, .NET must suspend all running threads, so that it can compact the heap, move objects in memory, and patch client-side references. The JIT compiler identifies those points in the code that are safe for suspending the thread (such as returning from method calls or branching for another loop iteration). When **Suspend()** is called, the thread will be suspended once it reaches the next safe point.

Syntax:

```
Thread.Suspend()
```

```
Thread.Resume()
```

Thread Priority

Thread class's ThreadPriority property is used to sets thread's priority. The thread priority can have **Normal**, **AboveNormal**, **BelowNormal**, **Highest**, and **Lowest** values.

```
thread.Priority = ThreadPriority.Lowest
```

Example:

```
Imports System.Threading
Module Module1
Sub Main()
Dim th As New Thread(AddressOf WriteY)
th.Start()
For i As Integer = 0 To 10
Console.WriteLine("Hello")
Next
End Sub

Private Sub WriteY()
For i As Integer = 0 To 9
Console.WriteLine("world")
Next
End Sub
End Module
```

```
Imports System.Threading
Module Module1
Sub Main()
Dim th As New Thread(AddressOf WriteY)
th.Priority = ThreadPriority.Lowest
th.Start()
For i As Integer = 0 To 10
Console.WriteLine("Hello")
Next
End Sub

Private Sub WriteY()
For i As Integer = 0 To 9
Console.WriteLine("world")
Next
End Sub
End Module
```

Synchronization

At times, you might want to control access to a resource, such as an object's properties or methods, so that only one thread at a time can modify or use that resource. Your object is similar to the airplane restroom discussed earlier, and the various threads are like the people waiting in line. Synchronization is provided by a lock on the object, which prevents a second thread from barging in on your object until the first thread is finished with it.

In this section you examine three synchronization mechanisms provided by the CLR: the Interlock class, the Visual Basic .NET Lock statement, and the Monitor class. But first, you need to simulate a shared resource, such as a file or printer, with a simple integer variable: **counter**. Rather than opening the file or accessing the printer, you'll increment **counter** from each of two threads.

To start, declare the member variable and initialize it to 0:

```
Private counter As Integer = 0
```

Modify the Incrementer method to increment the counter member variable:

```
Public Sub Incrementer( )
Try
While counter < 1000
Dim temp As Integer = counter
```



```
temp += 1 ' increment
```

```
Thread.Sleep(0)
```

```
counter = temp  
Console.WriteLine("Thread {0}. Incrementer: {1}", _  
Thread.CurrentThread.Name, counter)  
End While
```

The idea here is to simulate the work that might be done with a controlled resource. Just as we might open a file, manipulate its contents, and then close it, here we read the value of **counter** into a temporary variable, increment the temporary variable.

Binary Data Files

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

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

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

Sr.No.	Method Name & Purpose
1	Public Overridable Sub Close It closes the BinaryReader object and the underlying stream.
2	Public Overridable Function Read As Integer Reads the characters from the underlying stream and advances the current position of the stream.
3	Public Overridable Function ReadBoolean As Boolean Reads a Boolean value from the current stream and advances the current position of the stream by one byte.
4	Public Overridable Function ReadByte As Byte Reads the next byte from the current stream and advances the current position of the stream by one byte.

5	<p>Public Overridable Function ReadBytes (count As Integer) As Byte()</p> <p>Reads the specified number of bytes from the current stream into a byte array and advances the current position by that number of bytes.</p>
---	--

The BinaryWriter Class

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

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

Sr.No.	Function Name & Description
1	<p>Public Overridable Sub Close</p> <p>It closes the BinaryWriter object and the underlying stream.</p>
2	<p>Public Overridable Sub Flush</p> <p>Clears all buffers for the current writer and causes any buffered data to be written to the underlying device.</p>
3	<p>Public Overridable Function Seek (offset As Integer, origin As SeekOrigin) As Long</p> <p>Sets the position within the current stream.</p>
4	<p>Public Overridable Sub Write (value As Boolean)</p> <p>Writes a one-byte Boolean value to the current stream, with 0 representing false and 1 representing true.</p>
5	<p>Public Overridable Sub Write (value As Byte)</p> <p>Writes an unsigned byte to the current stream and advances the stream position by one byte.</p>

Text Files

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

Visual Basic.Net Programming

The StreamReader Class

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

Sr.No.	Method Name & Purpose
1	Public Overrides Sub Close It closes the <code>StreamReader</code> object and the underlying stream and releases any system resources associated with the reader.
2	Public Overrides Function Peek As Integer Returns the next available character but does not consume it.
3	Public Overrides Function Read As Integer Reads the next character from the input stream and advances the character position by one character.

The StreamWriter Class

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

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

Sr.No.	Method Name & Purpose
1	Public Overrides Sub Close Closes the current <code>StreamWriter</code> object and the underlying stream.
2	Public Overrides Sub Flush Clears all buffers for the current writer and causes any buffered data to be written to the underlying stream.
3	Public Overridable Sub Write (value As Boolean)

	Writes the text representation of a Boolean value to the text string or stream. (Inherited from TextWriter.)
4	Public Overrides Sub Write (value As Char) Writes a character to the stream.
5	Public Overridable Sub Write (value As Decimal) Writes the text representation of a decimal value to the text string or stream.

Example

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

Output:

Zara Ali
Nuha Ali
Amir Sohel
M Amlan

The DirectoryInfo Class

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

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

Sr.No.	Property Name & Description
1	Attributes Gets the attributes for the current file or directory.
2	CreationTime Gets the creation time of the current file or directory.
3	Exists Gets a Boolean value indicating whether the directory exists.
4	Extension Gets the string representing the file extension.
5	FullName Gets the full path of the directory or file.
6	LastAccessTime Gets the time the current file or directory was last accessed.
7	Name

	Gets the name of this DirectoryInfo instance.
--	---

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

Sr.No.	Method Name & Purpose
1	<p>Public Sub Create</p> <p>Creates a directory.</p>
2	<p>Public Function CreateSubdirectory (path As String) As DirectoryInfo</p> <p>Creates a subdirectory or subdirectories on the specified path. The specified path can be relative to this instance of the DirectoryInfo class.</p>
3	<p>Public Overrides Sub Delete</p> <p>Deletes this DirectoryInfo if it is empty.</p>
4	<p>Public Function GetDirectories As DirectoryInfo()</p> <p>Returns the subdirectories of the current directory.</p>
5	<p>Public Function GetFiles As FileInfo()</p> <p>Returns a file list from the current directory.</p>

The FileInfo Class

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

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

Sr.No.	Property Name & Description
--------	-----------------------------

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

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

Sr.No.	Method Name & Purpose
1	<p>Public Function AppendText As StreamWriter Creates a StreamWriter that appends text to the file represented by this instance of the FileInfo.</p>
2	<p>Public Function Create As FileStream Creates a file.</p>
3	<p>Public Overrides Sub Delete Deletes a file permanently.</p>
4	<p>Public Sub MoveTo (destFileName As String) Moves a specified file to a new location, providing the option to specify a new file name.</p>
5	<p>Public Function Open (mode As FileMode) As FileStream Opens a file in the specified mode.</p>
6	<p>Public Function Open (mode As FileMode, access As FileAccess) As FileStream Opens a file in the specified mode with read, write, or read/write access.</p>
7	<p>Public Function Open (mode As FileMode, access As FileAccess, share As FileShare) As FileStream Opens a file in the specified mode with read, write, or read/write access and the specified sharing option.</p>
8	<p>Public Function OpenRead As FileStream Creates a read-only FileStream</p>
9	<p>Public Function OpenWrite As FileStream Creates a write-only FileStream.</p>

Visual Basic.Net Programming

Example

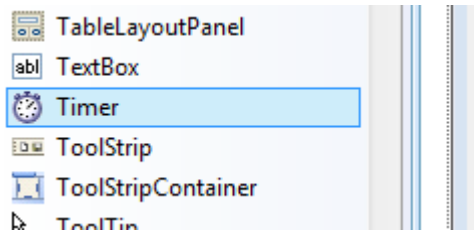
90

```
Imports System.IO
Module fileProg
Sub Main()
'creating a DirectoryInfo object
Dim mydir As DirectoryInfo = New DirectoryInfo("c:\Windows")
' getting the files in the directory, their names and size
Dim f As FileInfo() = mydir.GetFiles()
Dim file As FileInfo
For Each file In f
Console.WriteLine("File Name: {0} Size: {1} ", file.Name, file.Length)
Next file
Console.ReadKey()
End Sub
End Module
```

Unit-V

Timer Control

Timer Control plays an important role in the Client side programming and Server side programming, also used in Windows Services. By using this Timer Control, windows allow you to control when actions take place without the interaction of another thread.



Use of Timer Control

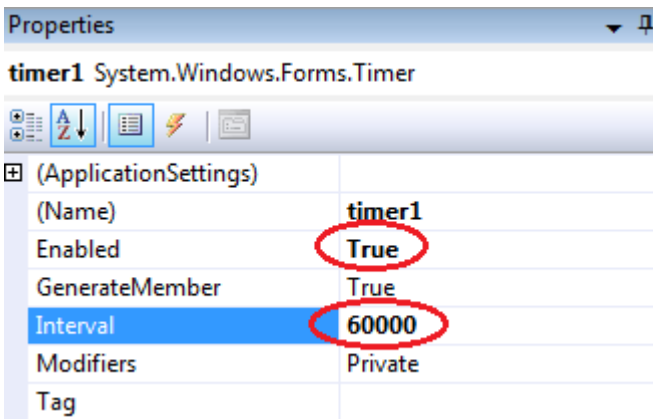
We can use **Timer Control** in many situations in our development environment. If you want to run some code after a certain interval of time continuously, you can use the Timer control. As well as to start a process at a fixed time schedule, to increase or decrease the speed in an animation graphics with time schedule etc. you can use the Timer Control. The Visual Studio toolbox has a Timer Control that allowing you to drag and drop the timer controls directly onto a Windows Forms designer. At runtime it does not have a visual representation and works as a component in the background.



How to Timer Control ?

With the Timer Control, we can control programs in millisecond, seconds, minutes and even in hours. The Timer Control allows us to set Interval property in milliseconds (1 second is equal to 1000 milliseconds). For example, if we want to set an interval of two minute we set the value at Interval property as 120000, means 120x1000 .

The Timer Control starts its functioning only after its Enabled property is set to True, by default Enabled property is False.



Timer example

The following program shows a Timer example that display current system time in a Label control. For doing this, we need one Label control and a Timer Control. Here in this program, we can see the Label Control is updated each seconds because we set Timer Interval as 1 second, that is 1000 milliseconds. After drag and drop the Timer Control in the designer form , double click the Timer control and set the DateTime.Now.ToString to Label control text property.

```
Public Class Form1
```

```
Private Sub Timer1_Tick(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Timer1.Tick
```

```
Label1.Text = DateTime.Now.ToString
```

```
End Sub
```

```
End Class
```

Start and Stop Timer Control

We can control the Timer Control Object that when it start its function as well as when it stop its function. The Timer Control has a start and stop methods to perform these actions.

```
Timer1.Start() 'Timer starts functioning
```

```
Timer1.Stop() 'Timer stops functioning
```

Visual Basic.Net Programming

Here is an example for start and stop methods of the Timer Control. In this example we run this program only 10 seconds. So we start the Timer in the Form_Load event and stop the Timer after 10 seconds. We set timer Interval property as 1000 milliseconds (1 second) and in run time the Timer will execute 10 times its Tick event

93

Example:

```
Public Class Form1

Dim second As Integer

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

Timer1.Interval = 1000

Timer1.Start() 'Timer starts functioning

End Sub

Private Sub Timer1_Tick(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Timer1.Tick

Label1.Text = DateTime.Now.ToString

second = second + 1

If second >= 10 Then

Timer1.Stop() 'Timer stop s functioning

MsgBox("Timer Stopped....")

End If

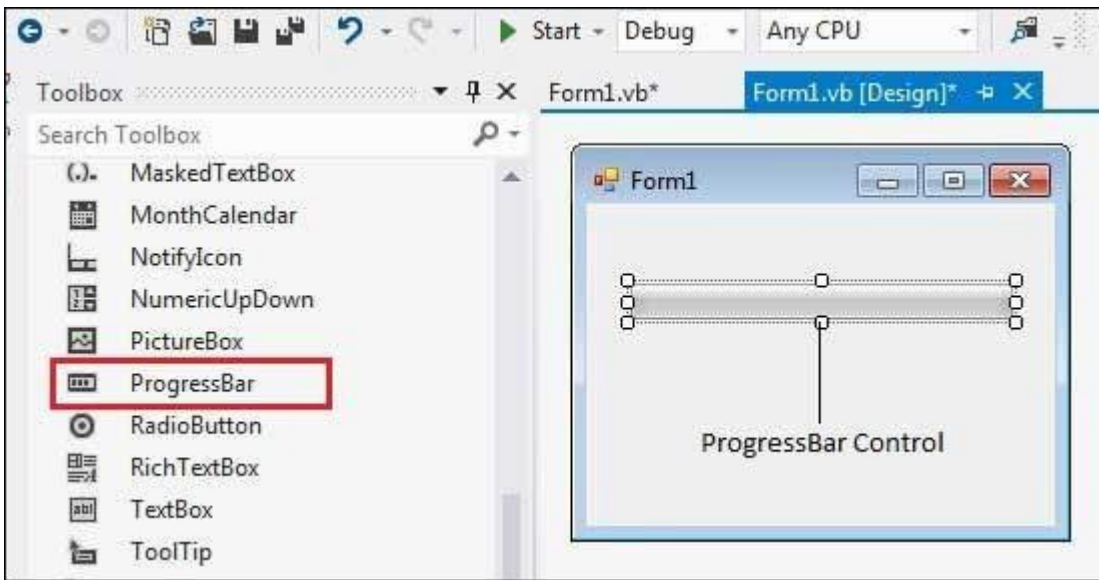
End Sub

End Class
```

ProgressBar Control

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

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



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

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

Properties of the *ProgressBar* Control

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

Sr.No.	Property & Description
1	AllowDrop Overrides <i>Control.AllowDrop</i> .
2	BackgroundImage Gets or sets the background image for the <i>ProgressBar</i> control.
3	BackgroundImageLayout Gets or sets the layout of the background image of the progress bar.

4	<p>CausesValidation</p> <p>Gets or sets a value indicating whether the control, when it receives focus, causes validation to be performed on any controls that require validation.</p>
5	<p>Font</p> <p>Gets or sets the font of text in the ProgressBar.</p>
6	<p>ImeMode</p> <p>Gets or sets the input method editor (IME) for the ProgressBar.</p>

Methods of the ProgressBar Control

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

Sr.No.	Method Name & Description
1	<p>Increment</p> <p>Increments the current position of the ProgressBar control by specified amount.</p>
2	<p>PerformStep</p> <p>Increments the value by the specified step.</p>
3	<p>ResetText</p> <p>Resets the Text property to its default value.</p>
4	<p>ToString</p> <p>Returns a string that represents the progress bar control.</p>

Events of the ProgressBar Control

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

Sr.No.	Event & Description
--------	---------------------

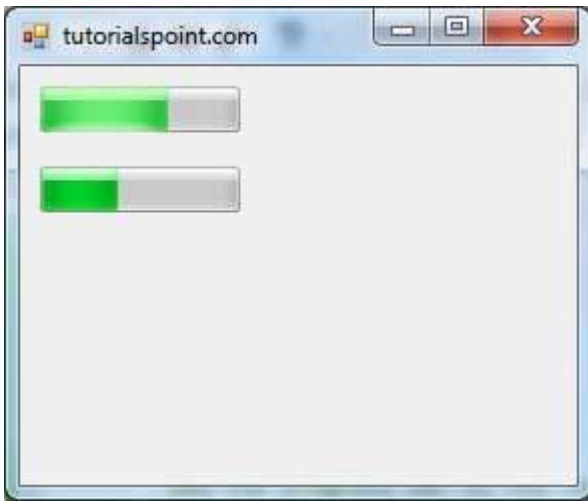
1	BackgroundImageChanged Occurs when the value of the BackgroundImage property changes.
2	BackgroundImageLayoutChanged Occurs when the value of the BackgroundImageLayout property changes.
3	CausesValidationChanged Occurs when the value of the CausesValidation property changes.
4	Click Occurs when the control is clicked.
5	DoubleClick Occurs when the user double-clicks the control

Visual Basic.Net Programming

Example

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

Output:



LinkLabel

LinkLabel Control is designed such that it provides the functionality of Hyperlink in window application. It is derived from label Control so it also provides all the functionality of Label control.

Properties of Linklabel Control

Property	Purpose
LinkColor	It is used to get or set Fore color of the Hyperlink in its default state.
ActiveLinkColor	It is used to get or set Fore color of the Hyperlink when user clicks it.
DisabledLinkColor	It is used to get or set Fore color of the Hyperlink when LinkLabel is disabled.
VisitedLinkColor	It is used to get or set Fore color of the Hyperlink when LinkVisited property of LinkLabel is set to true.
LinkVisited	It is used to specify weather Hyperlink is already visited or not. It has Boolean value. Default value is false.
Text	It is used to get or set text associated with LinkLabel Control.
TextAlign	It is used to get or set alignment of the text associated with LinkLabel Control.

Methods

Method	Purpose
Show	It is used to Show LinkLabel Control at runtime.
Hide	It is used to Hide LinkLabel Control at runtime.
Focus	It is used to set input focus on LinkLabel Control.

Events

Event	Purpose
Link Clicked	It is the default event of LinkLabel Control. It fires each time a user click on a hyperlink of LinkLabel Control.

Visual Basic.Net Programming

Example:

```
Private Sub LinkLabel1_LinkClicked(ByVal sender As System.Object, ByVal e As System.Windows.Forms.LinkLabelLinkClickedEventArgs) Handles LinkLabel1.LinkClicked
    'after visiting site LinkLabel color changed which will indicate that you have visited this site
    LinkLabel1.LinkVisited = True
    System.Diagnostics.Process.Start("www.mindstick.com")
    'using the start method of system.diagnostics.process class
    'process class gives access to local and remote processes
End Sub
```

Output:



Panel control

The Panel control is a container of other controls. The Panel control is displayed by default without any borders at run time.

How to use Panel control

Drag and drop Panel control from toolbox on the window Form.

Collection of control can be placed in side Panel.

Transparent Panel

First set BackColor of Panel suppose you set green then set Form's TransparencyKey property to the same color as Panel's background color –red in this case.

```
Private Sub Form25_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
```

```
Panel1.BackColor = Color.Red
```

```
Me.TransparencyKey = Color.Red
```

```
End Sub
```

Panel properties

BackColor: Panel BackColor can be changed through BackColor property.

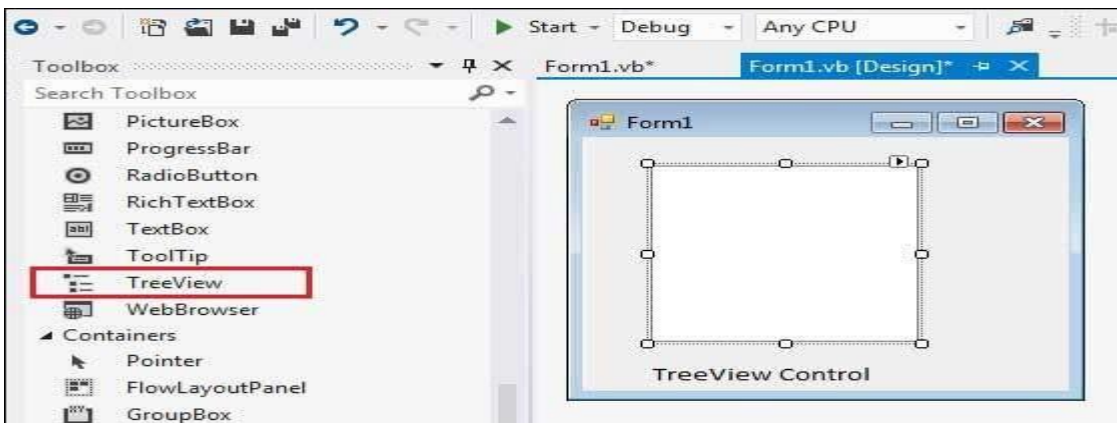
BorderStyle: Get or set BorderStyle of Panel.

Visible: You can hide all control inside panel through visible property of Panel. If you want to hide then set visible to false.

TreeView Control

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

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



Visual Basic.Net Programming

Properties of the TreeView Control

The following are some of the commonly used properties of the TreeView control –

Sr.No.	Property & Description
1	BackColor Gets or sets the background color for the control.
2	BackgroundImage Gets or set the background image for the TreeView control.
3	BackgroundImageLayout Gets or sets the layout of the background image for the TreeView control.
4	BorderStyle Gets or sets the border style of the tree view control.
5	CheckBoxes Gets or sets a value indicating whether check boxes are displayed next to the tree nodes in the tree view control.
6	DataBindings Gets the data bindings for the control.

Methods of the TreeView Control

The following are some of the commonly used methods of the TreeView control –

Sr.No.	Method Name & Description
1	CollapseAll Collapses all the nodes including all child nodes in the tree view control.

2	ExpandAll Expands all the nodes.
3	GetNodeAt Gets the node at the specified location.
4	GetNodeCount Gets the number of tree nodes.
5	Sort Sorts all the items in the tree view control.
6	ToString Returns a string containing the name of the control.

Events of the TreeView Control

The following are some of the commonly used events of the TreeView control –

Sr.No.	Event & Description
1	AfterCheck Occurs after the tree node check box is checked.
2	AfterCollapse Occurs after the tree node is collapsed.
3	AfterExpand Occurs after the tree node is expanded.
4	AfterSelect

	Occurs after the tree node is selected.
5	<p>BeforeCheck</p> <p>Occurs before the tree node check box is checked.</p>

Example

Public Class Form1

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

'create a new TreeView

Dim TreeView1 As TreeView

TreeView1 = New TreeView()

TreeView1.Location = New Point(10, 10)

TreeView1.Size = New Size(150, 150)

Me.Controls.Add(TreeView1)

TreeView1.Nodes.Clear()

'Creating the root node

Dim root = New TreeNode("Application")

TreeView1.Nodes.Add(root)

TreeView1.Nodes(0).Nodes.Add(New TreeNode("Project 1"))

'Creating child nodes under the first child

For loopindex As Integer = 1 To 4

TreeView1.Nodes(0).Nodes(0).Nodes.Add(New _

TreeNode("Sub Project" & Str(loopindex)))

Next loopindex

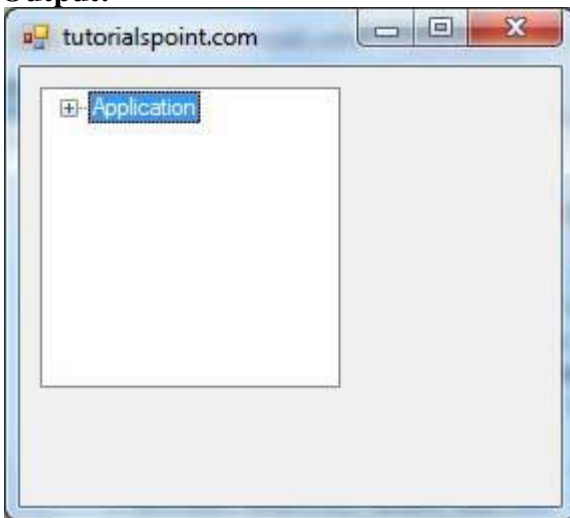
' creating child nodes under the root

TreeView1.Nodes(0).Nodes.Add(New TreeNode("Project 6"))

'creating child nodes under the created child node


```
For loopindex As Integer = 1 To 3
TreeView1.Nodes(0).Nodes(1).Nodes.Add(New _
TreeNode("Project File" & Str(loopindex)))
Next loopindex
' Set the caption bar text of the form.
Me.Text = "tutorialspoint.com"
End Sub
End Class
```

Output:



SplitContainer Control

A SplitContainer has two panels. The first panel is represented by Panel1 and second panel is represented by Panel2. These panels can have their own properties and events. Drag and drop SplitContainer control from toolbox on the window Form.

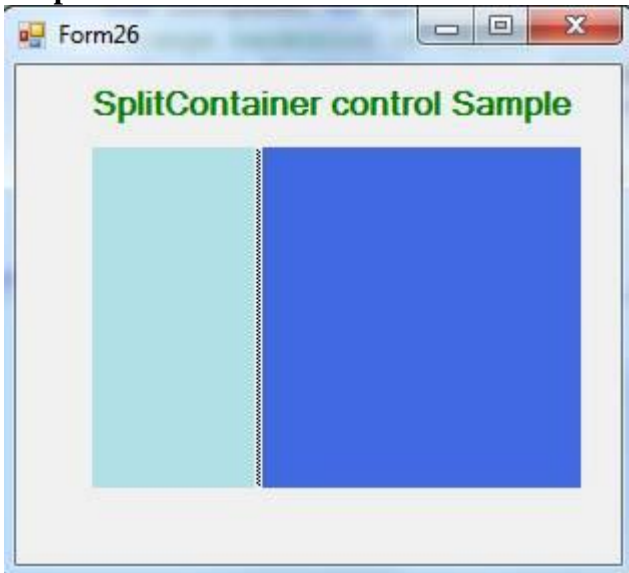
Properties of SplitContainer control:

- **Orientation** - Gets or sets a value indicating the Horizontal or Vertical orientation of the SplitContainer panels.
- **BackgroundImage** - Instead of a single color, an image can be displayed as the background. The image only appears in the splitter bar.

Example:

```
Private Sub Form26_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles MyBase.Load
Dim leftpanel As SplitterPanel = SplitContainer1.Panel1
'change backcolor of Panle1
leftpanel.BackColor = Color.PowderBlue
Dim rightpanel As SplitterPanel = SplitContainer1.Panel2
'change backcolor of Panle2
rightpanel.BackColor = Color.RoyalBlue
End Sub
```

Output:

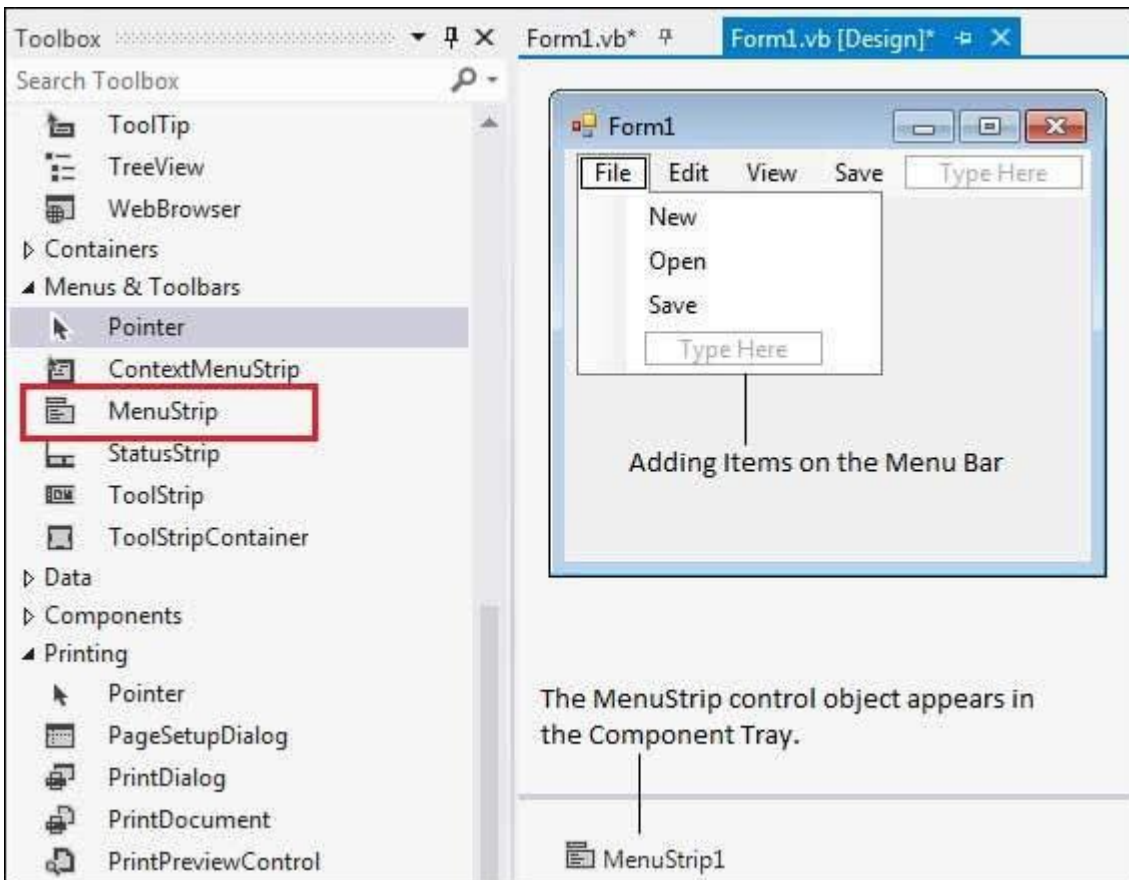


MenuStrip Control

The **MenuStrip** control represents the container for the menu structure.

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

The following diagram shows adding a MenuStrip control on the form –



Properties of the MenuStrip Control

The following are some of the commonly used properties of the MenuStrip control –

Sr.No.	Property & Description
1	<p>CanOverflow</p> <p>Gets or sets a value indicating whether the MenuStrip supports overflow functionality.</p>
2	<p>GripStyle</p> <p>Gets or sets the visibility of the grip used to reposition the control.</p>
3	<p>MdiWindowListItem</p> <p>Gets or sets the ToolStripMenuItem that is used to display a list of Multiple-document interface (MDI) child forms.</p>

4	<p>ShowItemToolTips</p> <p>Gets or sets a value indicating whether ToolTips are shown for the MenuStrip.</p>
5	<p>Stretch</p> <p>Gets or sets a value indicating whether the MenuStrip stretches from end to end in its container.</p>

Events of the MenuStrip Control

The following are some of the commonly used events of the MenuStrip control –

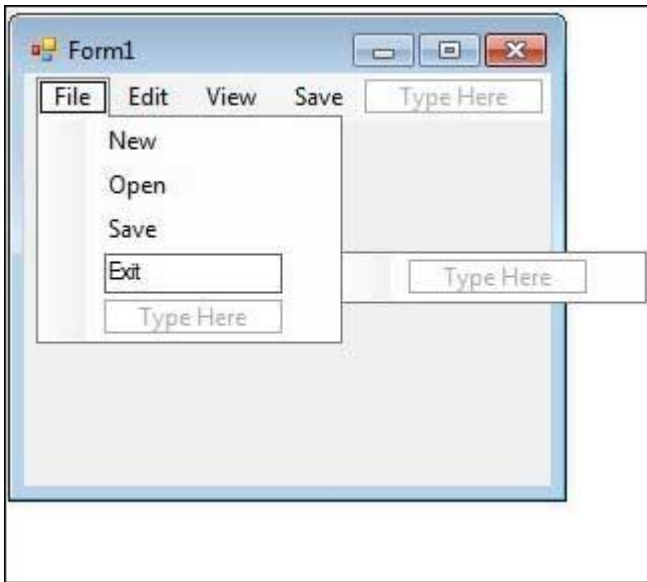
Sr.No.	Event & Description
1	<p>MenuActivate</p> <p>Occurs when the user accesses the menu with the keyboard or mouse.</p>
2	<p>MenuDeactivate</p> <p>Occurs when the MenuStrip is deactivated.</p>

Example

In this example, let us add menu and sub-menu items.

Take the following steps –

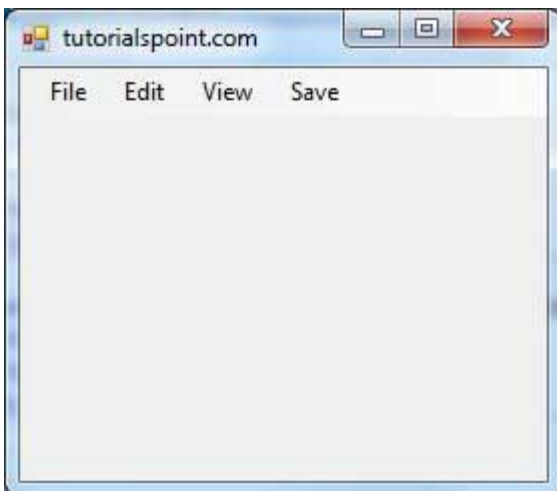
- Drag and drop or double click on a MenuStrip control, to add it to the form.
- Click the Type Here text to open a text box and enter the names of the menu items or sub-menu items you want. When you add a sub-menu, another text box with 'Type Here' text opens below it.
- Complete the menu structure shown in the diagram above.
- Add a sub menu **Exit** under the **File** menu.



- Double-Click the Exit menu created and add the following code to the **Click** event of **ExitToolStripMenuItem** –

```
Private Sub ExitToolStripMenuItem_Click(sender As Object, e As EventArgs) _  
Handles ExitToolStripMenuItem.Click  
End  
End Sub
```

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



The SDI and MDI forms are the interface design for document handling within a single Windows application. The MDI stands for Multiple Document Interface whereas SDI stands for Single Document Interface.

MDI: A multiple document Interface is one that allows viewing multiple windows within a large window.

SDI: A single Document Interface is one where all Windows appear independently of one another without the unification of a single parent window.

The Visual Basic IDE can be viewed in two ways:

1. With the Multiple Document Interface (MDI)
2. Single Document Interface (SDI)

MDI view shows all the distinct windows of Visual Basic IDE as child windows within on large IDE Window.

In the SDI view, distinct windows of the Visual Basic IDE exist independently of each other.

MDI Forms

- This is the main form or parent form which is not duplicated, but acts like a container for all the Windows which is also called the primary window.
- The windows in which the individual documents are displayed are called Child Windows.

- An MDI application must have atleast two form, the primary parent form and one or more child forms.

- The parent form may not contain any controls. While the parent Form is open in design mode, the icon on the tool box are not displayed, but you can't place any control on the form.

- The parent form usually have a menu.

To create an MDI form follow these steps

- Start a new project and then choose: Project -> Add MDI Form to add the parent form.

- Set the Forms caption to MDI window.

- Choose Project -> Add Form to add a SDI window.

- Make this form as child of MDI form by setting the MDI child property of the SDI form to True. Set the caption property to MDI child window.

Dialog Boxes

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

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

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

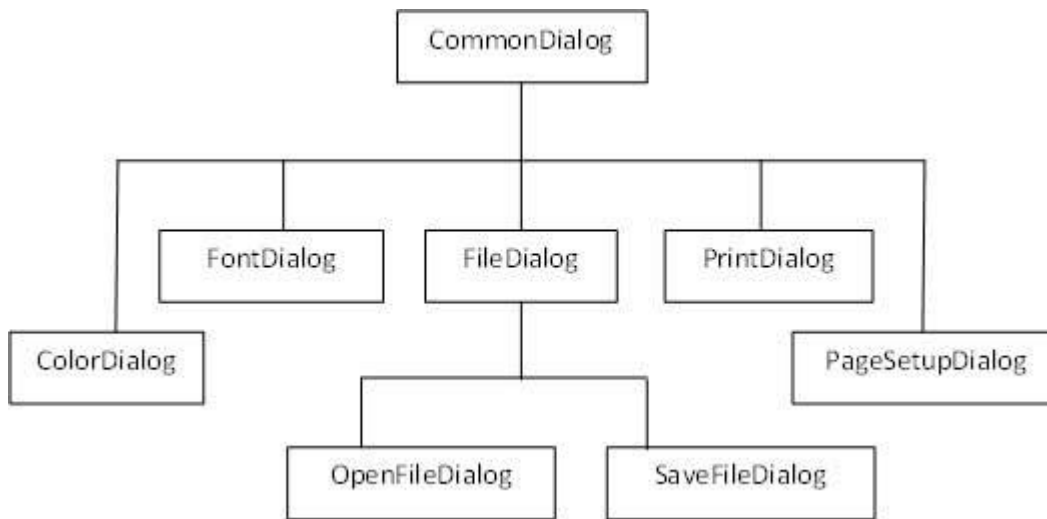
The **ShowDialog** method is used to display all the dialog box controls at run-time. It returns a value of the type of **DialogResult** enumeration. The values of DialogResult enumeration are –

- **Abort** – returns DialogResult.Abort value, when user clicks an Abort button.
- **Cancel** – returns DialogResult.Cancel, when user clicks a Cancel button.
- **Ignore** – returns DialogResult.Ignore, when user clicks an Ignore button.

Visual Basic.Net Programming

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

The following diagram shows the common dialog class inheritance –



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

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

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

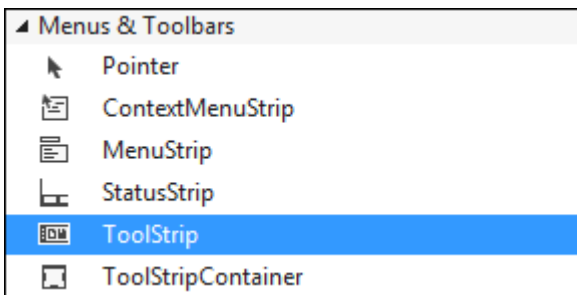
Sr.No.	Control & Description
1	<p>ColorDialog</p> <p>It represents a common dialog box that displays available colors along with controls that enable the user to define custom colors.</p>
2	<p>FontDialog</p> <p>It prompts the user to choose a font from among those installed on the local computer and</p>

	lets the user select the font, font size, and color.
3	<p>OpenFileDialog</p> <p>It prompts the user to open a file and allows the user to select a file to open.</p>
4	<p>SaveFileDialog</p> <p>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.</p>
5	<p>PrintDialog</p> <p>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.</p>

Toolbar

The toolbar is a very popular and much-used addition to a programme. It's difficult to think of a piece of software that doesn't make use of them. VB.NET lets you add toolbars to your forms, and the process is quite straightforward. Let's see how it's done:

Either start a new Windows project, or keep the one you currently have. To add a toolbar to the top of your form, expand the Toolbox and locate the ToolStrip control:



Double click the ToolStrip control, and it will be added to the top of your form: You should also notice the ToolStrip object that appears at the bottom of the window:

ToolStrips work by adding buttons and images to them. The button is then clicked, and an action performed.

Visual Basic.Net Programming

Click on your ToolStrip to select it. In the property box for the ToolStrip, you'll notice that it has the default Name of **ToolStrip1**. We'll keep this Name. But locate the Items (Collection) property:

114

Click the button with the three dots in it. This brings up the Items Collection Editor:

To add a new button to your ToolStrip, click the **Add** button at the top. The button appears in the Members box (ToolStripButton1):

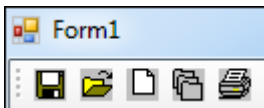
Notice that the new button has its own list of properties, just to the right. To add an image to this new button, locate the Image property:

Click the small button with the 3 dots in it to bring up the Select Resource box:

We then navigated to some Bitmap images and imported the five that you can see in the screenshot above (these are in the BITMAP folder that you download at the top of this tutorial). Click OK when you have imported some images. You will be returned to the Item Collection Editor. Click OK on this, as well.

To add a new button to the toolstrip, click on Button from the drop down menu in the image above. A default button is added called ToolStripButton2. (The first button is called ToolStripButton1.)

Repeat the steps above to add more buttons to the toolstrip. It should then look something like ours:



You can place any code you like, here. Try a message box, as in the image below:

```
Private Sub ToolStripButton1_Click(sender As Object, e As EventArgs) _
    Handles ToolStripButton1.Click
    MessageBox.Show("Save button clicked")
End Sub
```

Run your programme and click your ToolStrip button. You should see the message box display.

StatusBar

A StatusBar control is a combination of StatusBar panels where each panel can be used to display different information. In this article, I will discuss how to create and use a StatusBar using StatusBar class in a Windows Forms application.

StatusBar control is not available in Toolbox of Visual Studio 2010. StatusStrip control replaces StatusBar in Visual Studio 2010. But for backward compatibility support, StatusBar class is available in Windows Forms. In this article, I will discuss how to create and use a StatusBar using StatusBar class in a Windows Forms application.

A StatusBar control is a combination of StatusBar panels where each panel can be used to display different information. For example, one panel can display current application status and other can display date and other information and so on. A typical StatusBar sits at the bottom of a form.

Creating a StatusBar

StatusBar class represents a StatusBar.

```
Dim mainStatusBar As New StatusBar()
```

A StatusBar is a combination of StatusBar panels. StatusBarPanel class represents a StatusBar panel. The following code snippet creates two panels and adds them to the StatusBar.

```
Dim statusPanel As New StatusBarPanel()
```

```
Dim datetimePanel As New StatusBarPanel()
```

```
statusPanel.BorderStyle = StatusBarPanelBorderStyle.Sunken
```

```
statusPanel.Text = "Application started. No action yet."
```

```
statusPanel.ToolTipText = "Last Activity"
```

```
statusPanel.AutoSize = StatusBarPanelAutoSize.Spring
```

```
mainStatusBar.Panels.Add(statusPanel)
```

```
datetimePanel.BorderStyle = StatusBarPanelBorderStyle.Raised
```

```
datetimePanel.ToolTipText = "DateTime: " + System.DateTime.Today.ToString()
```

```
datetimePanel.Text = System.DateTime.Today.ToLongDateString()
```

```
datetimePanel.AutoSize = StatusBarPanelAutoSize.Contents
```

```
mainStatusBar.Panels.Add(datetimePanel)
```

Now, make sure ShowPanels property is true.

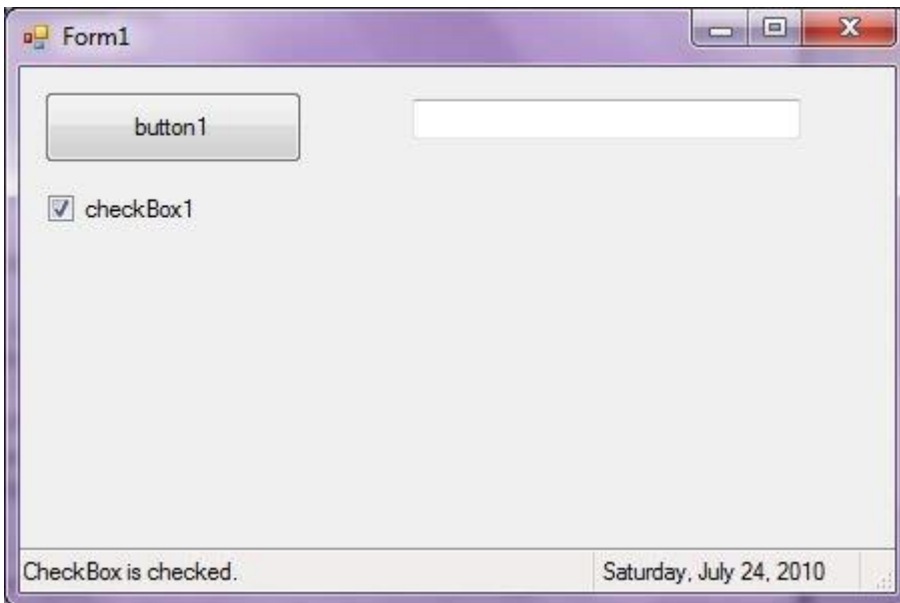
```
mainStatusBar.ShowPanels = True
```

In the end, we add StatusBar to the Form.

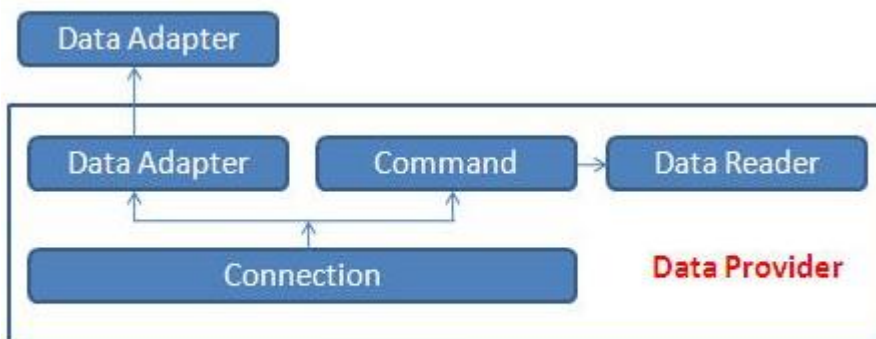
Visual Basic.Net Programming

Controls.Add(mainStatusBar)

Now let's create a Windows Forms application with a few controls on it. We are going to show current activity and date on the status bar. The Form looks like following.



Advantage of ADO.Net



ADO.NET

Single Object-Oriented API

ADO.NET provides a single object-oriented set of classes. Different data providers work with different data sources, but the programming model for all these data providers work in the same way. So, if you

Visual Basic.Net Programming

know how to work with one data provider, you can easily work with others. It's just a matter of changing class names and connection strings.

The ADO.NET classes are easy to use and easy to understand because of their object-oriented nature.

Managed Code

The ADO.NET classes are *managed classes*. They have all the advantages of the .NET CLR, such as language independency and automatic resource management. All .NET languages access the same API. So, if you know how to use these classes in C#, you will have no problem using them in VB .NET. Another big advantage is you don't have to worry about memory allocation and freeing it. The CLR takes care of it for you.

XML Support

Today, XML is an industry standard and the most widely used method of sharing data among applications over the Internet. As mentioned earlier, in ADO.NET data is cached and transferred in XML format. All components and applications can share this data, and data can be transferred via different protocols such as HTTP. We explain this topic in more detail in Chapters 6 and 7.

Visual Data Components

VS .NET offers ADO.NET components and data-bound controls to work in visual form. That means you can use these components as you use any Windows controls. You drag and drop these components on Windows and Web Forms, set their properties, and write events. This helps programmers to write less code and develop applications in no time. VS .NET also offers the Data Form Wizard, which helps you create full-fledged database applications without writing a single line of code. Using these components, you can directly bind these components with data-bound controls by setting these control's properties at design-time. Chapter 4 explains this in detail.

Performance and Scalability

Performance and scalability are two major factors when developing Web-based applications and services. Transferring data from one source to another is a costly affair over the Internet because of connection bandwidth limitations and rapidly increasing traffic. Using disconnected cached data in XML takes care of both of these problems.

Managed Data Providers

When speaking of data access, it's useful to distinguish between providers of data and consumers of data. A *data provider* encapsulates data and provides access to it in a generic way. The data itself can be in any form or location. For example, the data may be in a typical database management system such as SQL Server, or it may be distributed around the world and accessed via web services. The data

Visual Basic.Net Programming

provider shields the data consumer from having to know how to reach the data. In ADO.NET, data providers are referred to as *managed providers* .

118

A *data consumer* is an application that uses the services of a data provider for the purposes of storing, retrieving, and manipulating data. A customer-service application that manipulates a customer database is a typical example of a data consumer. To consume data, the application must know how to access one or more data providers.

ADO.NET is comprised of many classes, but five take center stage:

Connection

Represents a connection to a data source.

Command

Represents a query or a command that is to be executed by a data source.

DataSet

Represents data. The DataSet can be filled either from a data source (using a DataAdapter object) or dynamically.

DataAdapter

Used for filling a DataSet from a data source.

DataReader

Used for fast, efficient, forward-only reading of a data source.

With the exception of DataSet, these five names are not the actual classes used for accessing data sources. Each managed provider exposes classes.

Developing a Simple ADO.Net Application

```
Imports System.Data.SqlClient
```

```
Public Class Form1
```

Visual Basic.Net Programming

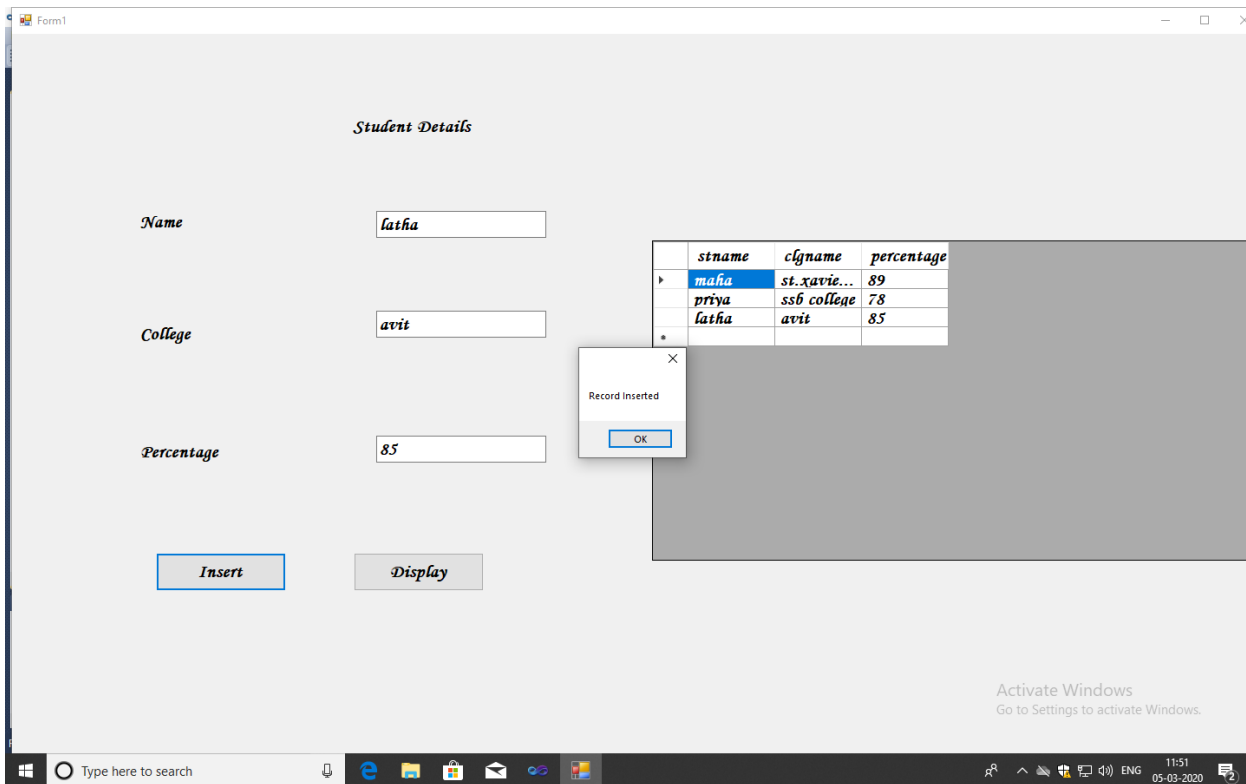
119

```
Dim con As New SqlConnection
Dim cmd As New SqlCommand
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
Button1.Click
cmd = con.CreateCommand()
cmd.CommandType = CommandType.Text
cmd.CommandText = "insert into Table1 values('" + TextBox1.Text + "','" + TextBox2.Text + "','" +
TextBox3.Text + "')"
cmd.ExecuteNonQuery()
disp_data()
MessageBox.Show("Record Inserted")
EndSub
```

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles MyBase.Load
con.ConnectionString = "Data
Source=.\SQLEXPRESS;AttachDbFilename=C:\Users\cse\documents\visual
2010\Projects\WindowsApplication1\WindowsApplication1\student.mdf;Integrated
Security=True;User Instance=True"
If con.State = ConnectionState.Open Then
con.Close()
EndIf
con.Open()
disp_data()
EndSub
```

```
Public Sub disp_data()
cmd = con.CreateCommand()
cmd.CommandType = CommandType.Text
cmd.CommandText = "select * from Table1"
cmd.ExecuteNonQuery()
Dim dt As New DataTable()
Dim da As New SqlDataAdapter(cmd)
da.Fill(dt)
DataGridView1.DataSource = dt
EndSub
EndClass
```

Output:



Creating a DataTable

A **DataTable**, which represents one table of in-memory relational data, can be created and used independently, or can be used by other .NET Framework objects, most commonly as a member of a **DataSet**.

You can create a **DataTable** object by using the appropriate **DataTable** constructor. You can add it to the **DataSet** by using the **Add** method to add it to the **DataTable** object's **Tables** collection.

You can also create **DataTable** objects within a **DataSet** by using the **Fill** or **FillSchema** methods of the **DataAdapter** object, or from a predefined or inferred XML schema using the **ReadXml**, **ReadXmlSchema**, or **InferXmlSchema** methods of the **DataSet**. Note that after you have added a **DataTable** as a member of the **Tables** collection of one **DataSet**, you cannot add it to the collection of tables of any other **DataSet**.

When you first create a **DataTable**, it does not have a schema (that is, a structure). To define the schema of the table, you must create and add **DataColumn** objects to the **Columns** collection of the table. You can also define a primary key column for the table, and create and add **Constraint** objects to the **Constraints** collection of the table. After you have defined the schema for a **DataTable**, you can add rows of data to the table by adding **DataRow** objects to the **Rows** collection of the table.

Visual Basic.Net Programming

You are not required to supply a value for the **TableName** property when you create a **DataTable**; you can specify the property at another time, or you can leave it empty. However, when you add a table without a **TableName** value to a **DataSet**, the table will be given an incremental default name of **TableN**, starting with "Table" for Table0. 121

The following example creates an instance of a **DataTable** object and assigns it the name "Customers."

```
DataTable workTable = new DataTable("Customers");
```

The following example creates an instance of a **DataTable** by adding it to the **Tables** collection of a **DataSet**.

```
DataSet customers = new DataSet();  
DataTable customersTable = customers.Tables.Add("CustomersTable");
```

Retrieve Data from A Table

Use **ado.net** to connect to a database and retrieve the row from the database table. To do that we use **DataAdapter** to retrieve the data from the database and place the data into **DataSet**. To fill the data into the **DataSet** use **Fill** method.

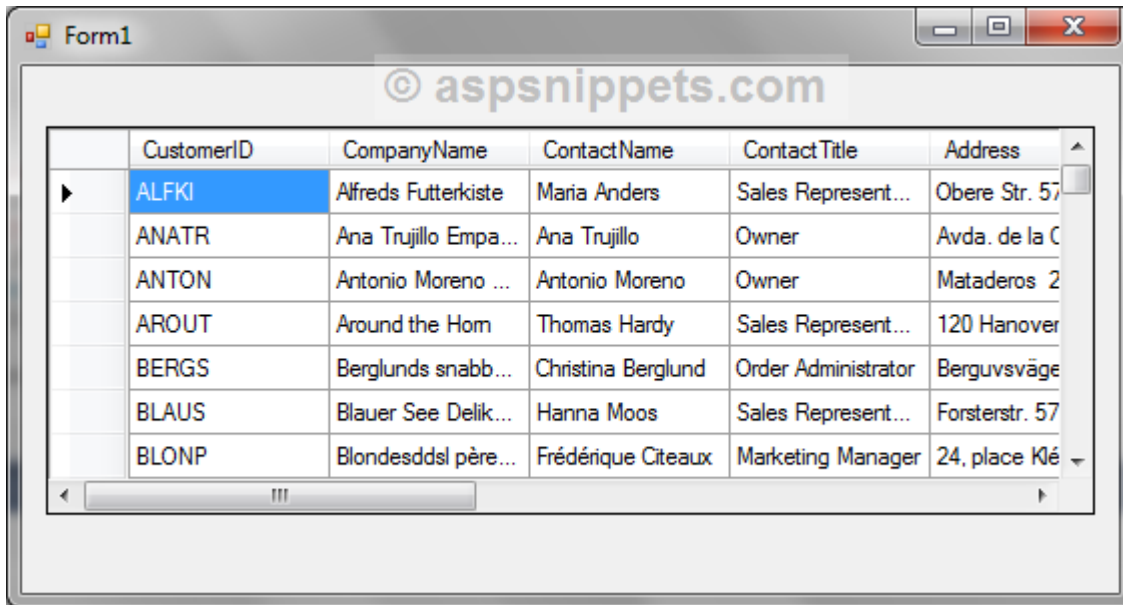
Example:

```
Imports System.Data  
Imports System.Data.SqlClient  
Public Class Form1  
Public Sub New()  
InitializeComponent()  
BindGrid()  
End Sub  
  
Private Sub BindGrid()  
Dim constring As String = "Data Source=.\SQL2005;Initial Catalog=Northwind;User id =  
sa;password=pass@123"  
Using con As New SqlConnection(constring)  
Using cmd As New SqlCommand("SELECT * FROM Customers", con)  
cmd.CommandType = CommandType.Text  
Using sda As New SqlDataAdapter(cmd)  
Using dt As New DataTable()  
sda.Fill(dt)  
dataGridView1.DataSource = dt  
End Using  
End Using  
End Using
```

Visual Basic.Net Programming

End Using
End Sub
End Class

Output:



CustomerID	CompanyName	ContactName	ContactTitle	Address
ALFKI	Alfreds Futterkiste	Maria Anders	Sales Represent...	Obere Str. 57
ANATR	Ana Trujillo Empa...	Ana Trujillo	Owner	Avda. de la C
ANTON	Antonio Moreno ...	Antonio Moreno	Owner	Mataderos 2
AROUT	Around the Hom	Thomas Hardy	Sales Represent...	120 Hanover
BERGS	Berglunds snabb...	Christina Berglund	Order Administrator	Berguvsväge
BLAUS	Blauer See Delik...	Hanna Moos	Sales Represent...	Forsterstr. 57
BLONP	Blondesddsl père...	Frédérique Citeaux	Marketing Manager	24, place Klé

Update data

After the data in your dataset has been modified and validated, you can send the updated data back to a database by calling the Update method of a TableAdapter. The Update method updates a single data table and runs the correct command (INSERT, UPDATE, or DELETE) based on the RowState of each data row in the table.

When a dataset has related tables, Visual Studio generates a TableAdapterManager class that you use to do the updates. The TableAdapterManager class ensures that updates are made in the correct order based on the foreign-key constraints that are defined in the database.

When you use data-bound controls, the databinding architecture creates a member variable of the TableAdapterManager class called tableAdapterManager.

The exact procedure for updating a data source can vary depending on business needs, but includes the following steps:

1. Call the adapter's Update method in a try/catch block.
2. If an exception is caught, locate the data row that caused the error.

3. Reconcile the problem in the data row (programmatically if you can, or by presenting the invalid row to the user for modification), and then try the update again.

Example:

```
Imports System.Data
Imports System.Data.SqlClient
Partial Class Default2
Inherits System.Web.UI.Page
Dim con As New SqlConnection
Dim cmd As New SqlCommand
Dim ds As New DataSet
Dim adap As New SqlDataAdapter

Protected Sub Button2_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles
Button2.Click
con.ConnectionString = "Data
Source=.\SQLEXPRESS;AttachDbFilename=C:\Users\Acer\Documents\Visual
Studio
2010\WebSites\WebSite2\App_Data\student.mdf;Integrated Security=True;User Instance=True"
con.Open()
cmd.Connection = con
cmd.CommandText = "update stud set name=" & TextBox2.Text & ",percentage=" & TextBox3.Text
& ",college=" & TextBox4.Text & " where regno=" & TextBox1.Text & ""
cmd.ExecuteNonQuery()
MsgBox("Record Updated")
End Sub
```

Disconnected Data Access through DataSet Object

The ADO.NET Framework supports two models of Data Access Architecture, Connection Oriented Data Access Architecture and Disconnected Data Access Architecture. The ADO.NET Disconnected Data Access Architecture far more flexible and powerful than ADOs Connection Oriented Data Access.

In Connection Oriented Data Access Architecture the application makes a connection to the Data Source and then interact with it through SQL requests using the same connection. In this case the application stays connected to the database system even when it is not using any Database Operations. On the other hand the disconnected approach makes no attempt to maintain a connection to the data source.

Visual Basic.Net Programming

ADO.Net provides a new solution by introduce a new component called Dataset. The DataSet is the central component in the ADO.NET Disconnected Data Access Architecture. A DataSet is an in-memory data store that can hold multiple tables at the same time. DataSets only hold data and do not interact with a Data Source. One of the key characteristics of the DataSet is that it has no knowledge of the underlying Data Source that might have been used to populate it. 124

Example:

```
Dim ds As New DataSet
```

In Connection Oriented Data Access, when you read data from a database by using a DataReader object, an open connection must be maintained between your application and the Data Source. Unlike the DataReader, the DataSet is not connected directly to a Data Source through a Connection object when you populate it.

It is the DataAdapter that manages connections between Data Source and Dataset by fill the data from Data Source to the Dataset and giving a disconnected behavior to the Dataset. The DataAdapter acts as a bridge between the Connected and Disconnected Objects.

Example:

```
Dim adapter As New SqlDataAdapter("sql", "connection")  
Dim ds As New DataSet  
adapter.Fill(ds, "Src Table")
```

By keeping connections open for only a minimum period of time, ADO .NET conserves system resources and provides maximum security for databases and also has less impact on system performance.

