Unit V: Object Oriented Programming: Classes and Objects: Class definition, creating objects, Defining Member functions, Methods and Events, Attaching a class with form, Delegates.

Exceptions Handling: Exception classes in .net framework, Structured and Unstructured exceptions, tracing errors, breakpoints, watch, Quick watch.

Class

Class is user define data type in VB.Net. It contains data members(attributes) and member function(behaviour). Class is a blue print of an object.

Creating Classes

the Class statement:

[<attrlist>] [Public | Private | Protected | Friend | Protected Friend] [Shadows] [MustInherit | NotInheritable] Class name [Implements interfacename]

[statements]

End Class

Here are the various parts of this statement:

- **attrlist**—Optional. This is the list of attributes for this class. Separate multiple attributes by commas.
- **Public**—Optional. Classes declared Public have public access; there are no restrictions on the use of public classes.
- **Private**—Optional. Classes declared Private have private access, which is accessible only within its declaration context.
- **Protected**—Optional. Classes declared Protected have protected access, which means they are accessible only from within their own class or from a derived class.
- **Friend**—Optional. Classes declared Friend have friend access, which means they are accessible only within the program that contains their declaration.
- **Protected Friend**—Optional. Classes declared Protected Friend have both protected and friend accessibility.
- **Shadows**—Optional. Indicates that this class shadows a programming element in a base class.
- **MustInherit**—Optional. Indicates that the class contains methods that must be implemented by a deriving class.
- **NotInheritable**—Optional. Indicates that the class is a class from which no further inheritance is allowed.
- **name**—Required. Name of the class.
- **interfacename**—Optional. The name of the interface implemented by this class. statements—Optional.

The statements that make up the variables, properties, events, and methods of the class.

Each attribute in the attrlist part has the following syntax: Attrlist Here are the parts of the attrlist part:

- **attrname**—Required. Name of the attribute.
- **attrargs**—Optional. List of arguments for this attribute. Separate multiple arguments by commas.
- **attrinit**—Optional. List of field or property initializers. Separate multiple initializers by commas.

You place the members of the class inside the class itself. You also can nest class declarations. here's how we set up a class named DataClass:

```
Public Class DataClass

        Private value As Integer

        Public Sub New(ByVal newValue As Integer)
                value = newValue
        End Sub

        Public Function GetData() As Integer
                Return value
          End Function
End Class
```

Creating Objects

You can create objects of a class using the Dim statement; this statement is used at module, class, structure, procedure, or block level:

[<attrlist>] [{ Public | Protected | Friend | Protected Friend | Private | Static }] [Shared] [Shadows] [ReadOnly] Dim [WithEvents] name [(boundlist)] [As [New] type] [= initex]

Here are the parts of this statement:

- **attrlist**—A list of attributes that apply to the variables you're declaring in this statement. You separate multiple attributes with commas.

- **Public**—Gives variables public access, which means there are no restrictions on their accessibility. You can use Public only at module, namespace, or file level (which means you can't use it inside a procedure). Note that if you specify Public, you can omit the Dim keyword if you want to.

- **Protected**—Gives variables protected access, which means they are accessible only from within their own class or from a class derived from that class. You can use Protected only at class level (which means you can't use it inside a procedure), because you use it to declare members of a class. Note that if you specify Protected, you can omit the Dim keyword if you want to. Friend—Gives variables friend access, which means they are accessible from within the program that contains their declaration, as well as anywhere else in the same assembly. You can use Friend only at module, namespace, or file level (which means you can't use it inside a procedure). Note that if you specify Friend, you can omit the Dim keyword if you want to.

- **Protected Friend**—Gives variables both protected and friend access, which means they can be used by code in the same assembly, as well as by code in derived classes.

- **Private**—Gives variables private access, which means they are accessible only from within their declaration context (usually a class), including any nested procedures. You can use Private only at module, namespace, or file level (which means you can't use it inside a procedure). Note that if you specify Private, you can omit the Dim keyword if you want to.

- **Static**—Makes variables static, which means they'll retain their values, even after the procedure in which they're declared ends. You can declare static variables inside a procedure or a block within a procedure, but not at class or module level. Note that if you specify Static, you can omit the Dim keyword if you want to, but you cannot use either Shadows or Shared.

- **Shared**—Declares a shared variable, which means it is not associated with a specific instance of a class or structure, but can be shared across many instances. You access a shared variable by referring to it either with its class or structure name, or with the variable name of an instance of the class or structure. You can use Shared only at module, namespace, or file level (but not at the procedure level). Note that if you specify Shared, you can omit the Dim keyword if you want to.

- **Shadows**—Makes this variable a shadow of an identically named programming element in a base class. A shadowed element is unavailable in the derived class that shadows it. You can use Shadows only at module, namespace, or file level (but not inside a procedure). This means you can declare shadowing variables in a source file or inside a module, class, or structure, but not inside a procedure. Note that if you specify Shadows, you can omit the Dim keyword if you want to.

- **ReadOnly**—Means this variable can only be read and not written. This can be useful for creating constant members of reference types, such as an object variable with preset data members. You can use ReadOnly only at module, namespace, or file level (but not inside procedures). Note that if you specify Shadows, you can omit the ReadOnly keyword if you want to.

- **WithEvents**—Specifies that this variable is used to respond to events caused by the instance that was assigned to the variable. Note that you cannot specify both WithEvents and New in the same variable declaration. name—The name of the variable. You separate multiple variables by commas. If you specify multiple variables, each variable is declared of the data type given in the first As clause encountered after its name part.

- **boundlist**—Used to declare arrays; gives the upper bounds of the dimensions of an array variable. Multiple upper bounds are separated by commas. An array can have up to 60 dimensions.

- **New**—Means you want to create a new object immediately. If you use New when declaring an object variable, a new instance of the object is created. Note that you cannot use both WithEvents and New in the same declaration.

- **type**—The data type of the variable. Can be Boolean, Byte, Char, Date, Decimal, Double, Integer, Long, Object, Short, Single, or String ; or the name of an enumeration, structure, class, or interface. To specify the type, you use a separate As clause for each variable, or you can declare a number of variables of the same type by using common As clauses. If you do not specify type, the variable takes the data type of initexpr. Note that if you don't specify either type or initexpr, the data type is set to Object.

- **initexpr**—An initialization expression which is evaluated and the result is assigned to the variable when it is created. Note that if you declare more than one variable with the same As clause, you cannot supply initexpr for those variables.

Each attribute in the attrlist list must use this syntax:

Here are the parts of the attrlist list:

- **attrname**—Name of the attribute.
- **attrargs**—List of arguments for this attribute. Separate multiple arguments with commas.
- **attrinit**—List of field or property initializers for this attribute. Separate multiple arguments with commas.

When you create a new object from a class, you use the New keyword. You can do that in either of these ways:

Dim employee As New EmployeeClass()

Dim employee As EmployeeClass = New EmployeeClass()

If you omit the New keyword, you're just declaring a new object, and it's not yet created:

Dim employee As EmployeeClass

Before using the object, you must explicitly create it with New:

employee = New EmployeeClass()

Member Function in VB.Net

There are three member function associated with class in VB.Net .

- Methods
- Properties
- Events

Method

Methods represent the object's built-in procedures. We mean that methods are functions or procedures associated with objects, and they define the behavior of those objects. They are built into the structure of the object and can be called to perform specific actions or manipulate the object's data. For example, a class named Animal may have methods named Sleeping and Eating. **You define methods by adding procedures, either Sub procedures or functions, to your class.** We might implement the Sleeping and Eating methods as follows:

```
Public Class Animal
        Public Sub Eating()
                MsgBox("Eating...")
         End Sub
         Public Sub Sleeping()
                MsgBox("Sleeping...")
          End Sub
 End Class
```

Now we can create a new object of the Animal class and call the Eating method in the familiar way:

```
Dim pet As New Animal()
pet.Eating()
```

In general, we can add whatever functions or Sub procedures you want to your class, including those that accept parameters and those that return values. **Sub procedures** are used when you want to execute a set of instructions **without returning a value**.

Functions are used when you want to **calculate a value** or perform an operation and then **provide that result to the caller**.
Defining Events in VB.Net

You can design and support your own events using OOP in Visual Basic, using the Event statement:

[<attrlist>] [Public | Private | Protected | Friend | Protected Friend] [Shadows] Event eventname[(arglist)] [Implements interfacename.interfaceeventname]

Here are the parts of this statement:

- **attrlist**—Optional. List of attributes that apply to this event. Separate multiple attributes by commas.
- **Public**—Optional. Events declared Public have public access, which means there are no restrictions on their use.
- **Private**—Optional. Events declared Private have private access, which means they are accessible only within their declaration context.
- **Protected**—Optional. Events declared Protected have protected access, which means they are accessible only from within their own class or from a derived class.
- **Friend**—Optional. Events declared Friend have friend access, which means they are accessible only within the program that contains the its declaration.
- **Protected Friend**—Optional. Events declared Protected Friend have both protected and friend accessibility.
- **Shadows**—Optional. Indicates that this event shadows an identically named programming element in a base class.
- **eventname**—Required. Name of the event.
- **interfacename**—The name of an interface.
- **interfaceeventname**—The name of the event being implemented.

Customized Event Example

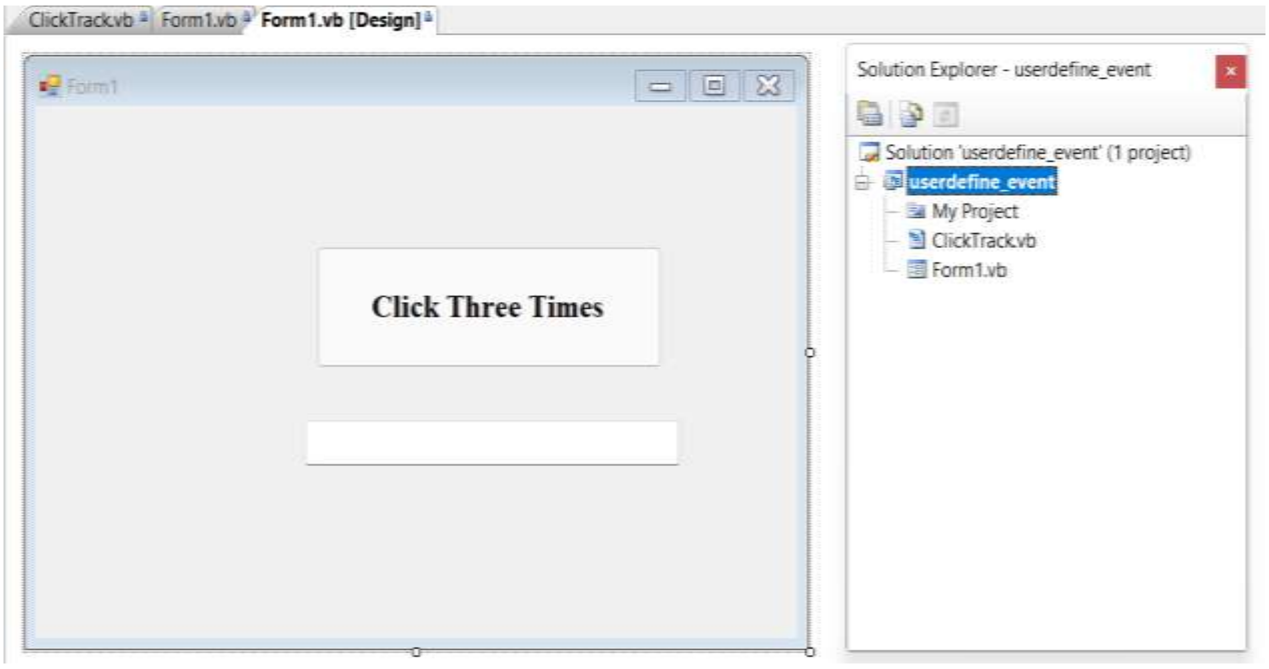
ClickTrack.vb Class File

```
Public Class ClickTrack
    Public Event ThreeClick(ByVal Message As String)
    Public Sub Click()
        Static ClickCount As Integer = 0
        ClickCount += 1
        If ClickCount >= 3 Then
            ClickCount = 0
            RaiseEvent ThreeClick("You clicked three times")
        End If
    End Sub
End Class
```

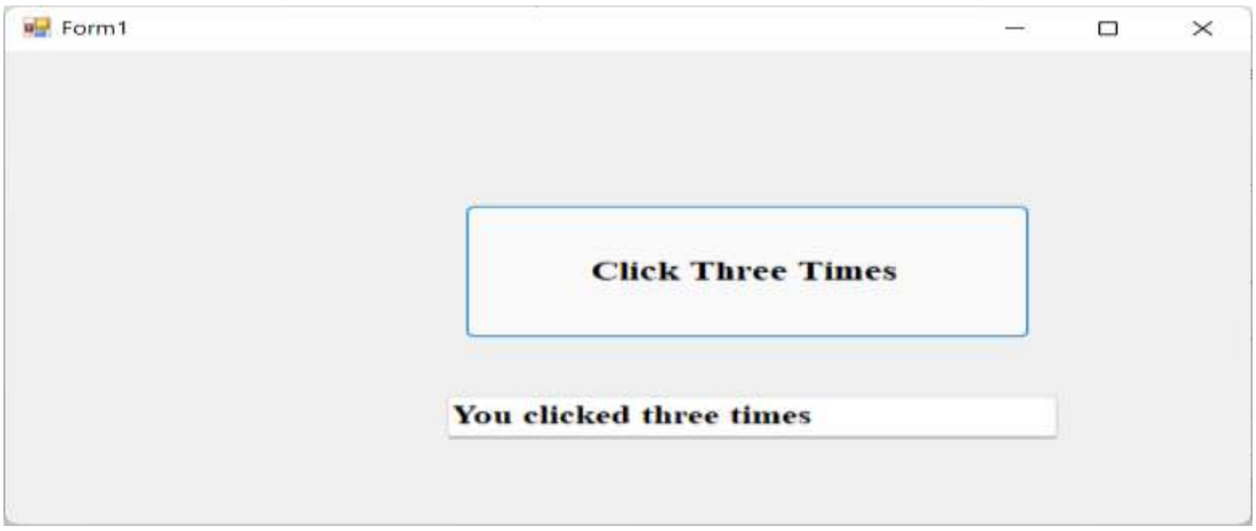
Form1.vb File

```
Public Class Form1
    Dim WithEvents tracker As New ClickTrack()
    Private Sub tracker_ThreeClick(ByVal Message As String) Handles
tracker.ThreeClick
        TextBox1.Text = Message
    End Sub
    Private Sub Button1_Click(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Button1.Click

      tracker.Click()
End Sub
End Class
```

Design Time:



Run Time:



You clicked three times

Creating Properties

Visual Basic objects can have methods, fields, and properties. If you've worked with Visual Basic before, you're familiar with properties, which you use to set configuration data for objects, such as the text in a text box or the width of a list box. Using properties provides you with an interface to set or get the value of data internal to an object. You declare properties using **Get** and **Set** procedures in a **Property** statement.

```
[ Public | Private | Protected | Friend | Protected Friend ] [ ReadOnly | WriteOnly ]
Property varname([ parameter list ]) [ As typename ]
        Get
                [ block ]
          End Get
        Set(ByVal Value As typename )
                [ block ]
        End Set
End Property
```

Here are the parts of this statement:

- **Public**—Optional. Events declared Public have public access, which means there are no restrictions on their use.
- **Private**—Optional. Events declared Private have private access, which means they are accessible only within their declaration context.
- **Protected**—Optional. Events declared Protected have protected access, which means they are accessible only from within their own class or from a derived class.
- **Friend**—Optional. Events declared Friend have friend access, which means they are accessible only within the program that contains the its declaration.
- **Protected Friend**—Optional. Events declared Protected Friend have both protected and friend accessibility.
- **ReadOnly**—Specifies that a properties value can be retrieved, but it cannot be the modified. ReadOnly properties contain Get blocks but no Set blocks.
- **WriteOnly**—Specifies that a property can be set but its value cannot be retrieved. WriteOnly properties contain Set blocks but no Get blocks.
- **varname**—A name that identifies the Property.
- **parameter list**—The parameters you use with the property. The list default is ByVal.
- **typename**—The type of the property. If you don't specify a data type, the default type is Object.
- **Get**—Starts a Get property procedure used to return the value of a property. Get blocks are **optional unless** the property is **ReadOnly**.
- **End Get**—Ends a Get property procedure.
- **Set**—Starts a Set property procedure used to set the value of a property. Set blocks are **optional unless** the property is **WriteOnly**. Note that the new value

of the property is passed to the Set property procedure in a parameter named Value when the value of the property changes.

- **End Set**—Ends a Set property procedure.

Visual Basic passes a parameter named Value to the Set block during property assignments, and the Value parameter contains the value that was assigned to the property when the Set block was called. Here's an example where I'm creating a read/write property.

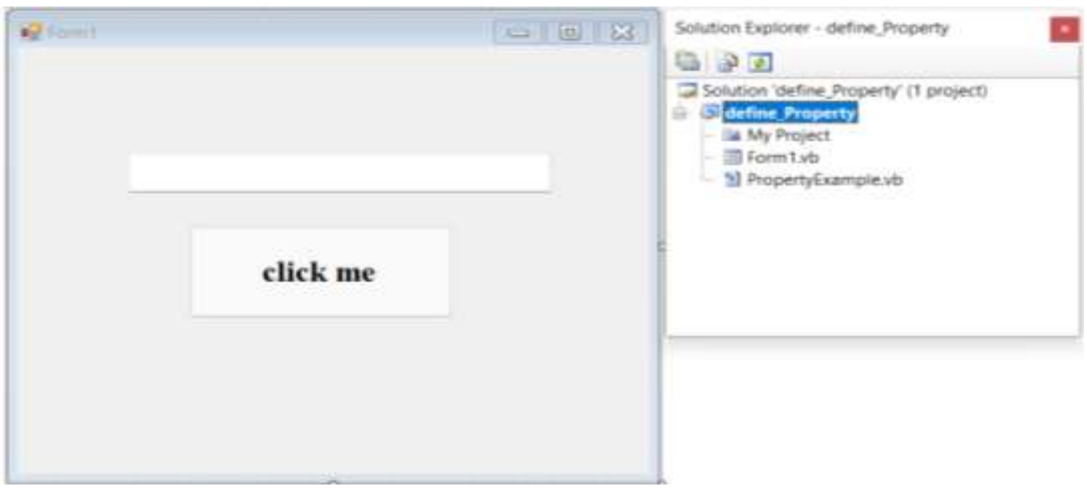
PropertyExample.vb class File

```
Public Class PropertyExample
    Dim message As String
    Public Property Input() As String
        Get
            Return message
        End Get
        Set(ByVal value As String)

            message = value

        End Set
    End Property

End Class
```

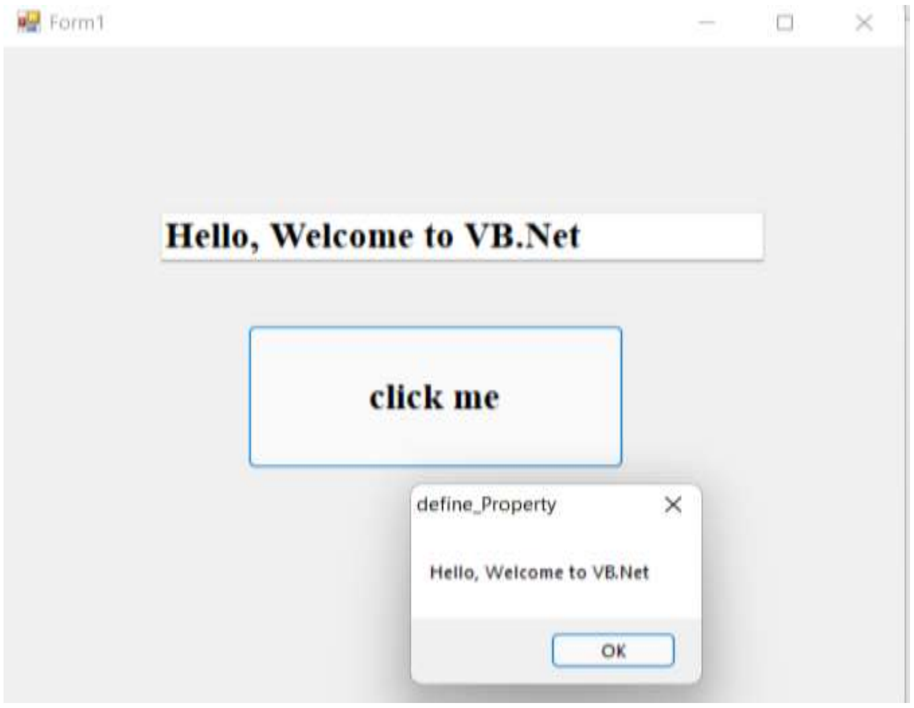
Form1.vb

```
Public Class Form1

    Dim p As PropertyExample = New PropertyExample()
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Button1.Click

        p.Input = TextBox1.Text
        MsgBox(p.Input)

    End Sub
End Class
```

1. Design Time


Form1
click me
Solution Explorer - define_Property
Solution 'define_Property' (1 project)
define_Property
My Project
Form1.vb
PropertyExample.vb

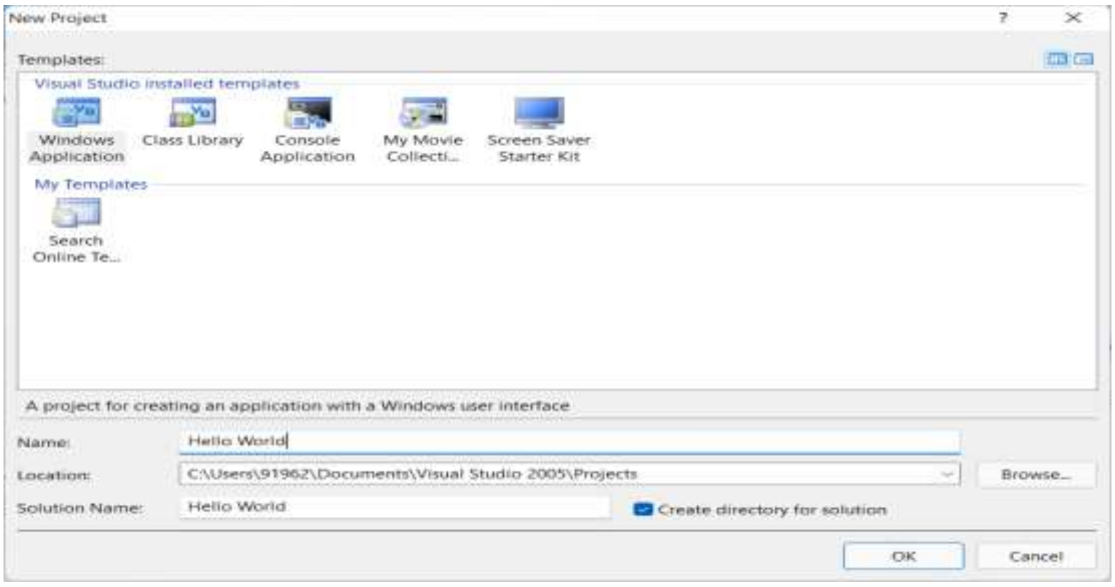
2. **Run Time**



Hello, Welcome to VB.Net

Steps to Attaching a class with form

1. Select **Programs** → **Microsoft Visual Studio .Net** from **start** menu.
2. Click on the **new Project** in **start** page → Select **Project** from file **menu**. This will display **New Project** dialog box as shown below fig.



A project for creating an application with a Windows user interface

Name: Hello World

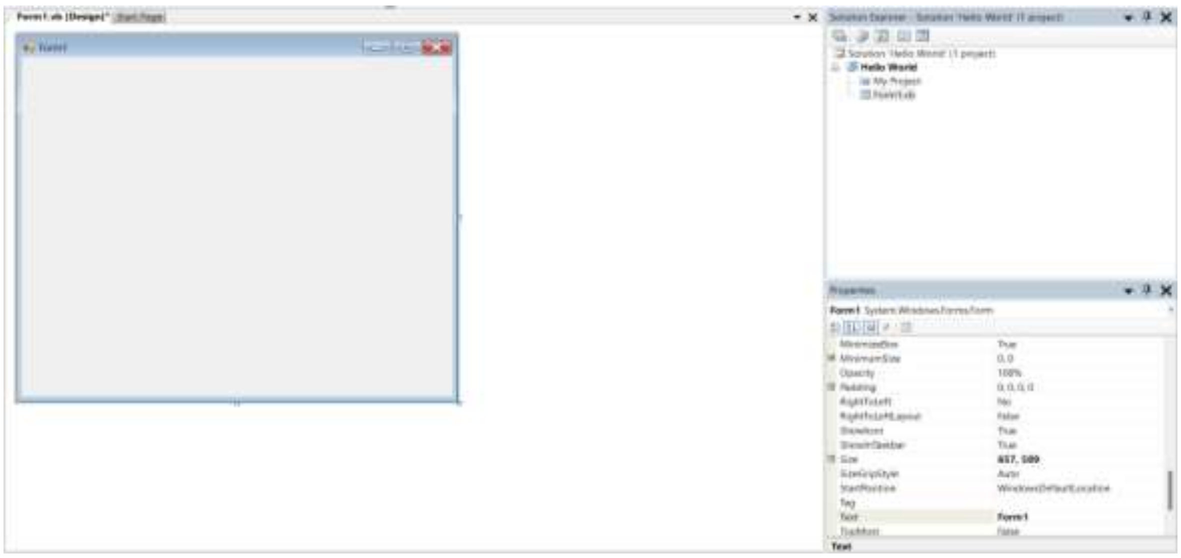
Location: C:\Users\91962\Documents\Visual Studio 2005\Projects

Solution Name: Hello World

☑ Create directory for solution

[Fig: New Project Dialog Box]

3. Now select **Windows Application** from **Templates** pane. Give name for project as **“Hello World”**. Accept the default location or browse the location where you want to save your project. Click on **OK** button. This will display an empty designer code window as shown in below fig.



Form1.vb [Design]*
Start Page
Form1
Solution Explorer - Solution 'Hello World' (1 project)
Solution 'Hello World' (1 project)
Hello World
My Project
Form1.vb
Properties

Form1 System.Windows.Forms.Form

MinimizeBox	True
MinimumSize	0, 0
Opacity	100%
Padding	0, 0, 0, 0
RightToLeft	No
RightToLeftLayout	False
ShowIcon	True
ShowInTaskbar	True
Size	457, 509
SizeGripStyle	Auto
StartPosition	WindowsDefaultLocation
Tag	
Text	Form1
TopMost	False

Text

[Fig: Form Designer]

4. Now to attach class to form Go to **Solution Explorer** → just **right click** on **Project name**(“Hello World in my case”) from pop up window select **Add** → Then Add **Class** from another pop up window. As shown in Fig

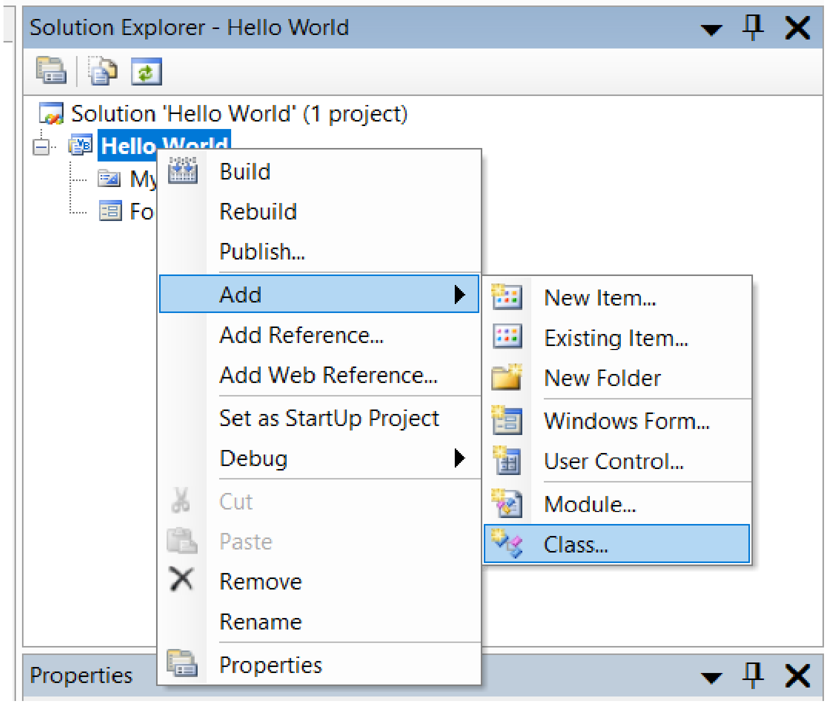
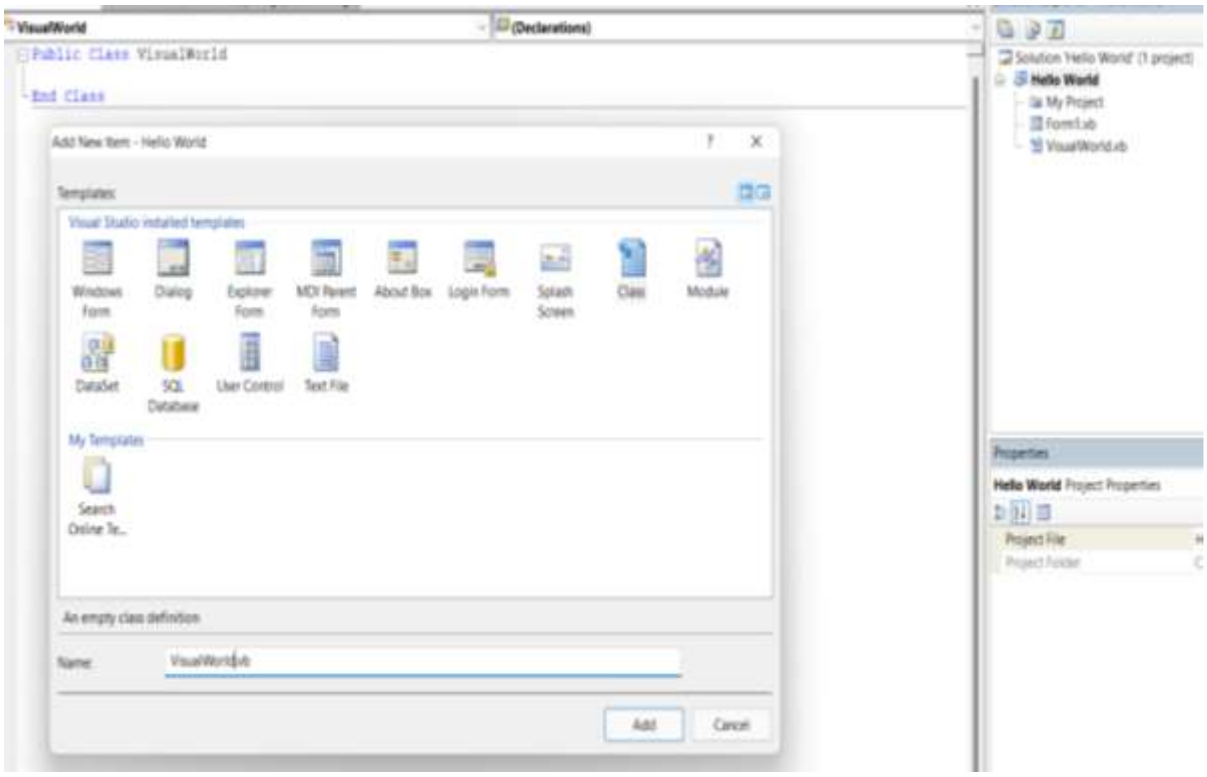


Fig: Attaching Class to form

5. Then, enter the name you want to use for the new class and click open. When you complete the process .Now, it will appear class file in the solution Explorer window with *vb* as the extension.

6. The class and End class statements are automatically added to the class. Then, you can enter the code for class between those statements.



```
Public Class VisualWorld

End Class
```

[Fig : Class added in window form]

Delegates

Delegates are objects that refers to methods. They are sometimes described as **type-safe** function pointer because its points the function or method indirectly through its memory address. Delegates cannot be used as **private** delegates.

Delegates usage:

- **Event Handling**: Delegates are commonly used in event-driven programming. For example, in Windows Forms applications, you often use delegates to handle events such as button clicks.
- **Callback Mechanisms**: Delegates are used in callback scenarios where a method needs to call another method asynchronously or to provide a callback mechanism.

1. **Delegate Declaration:**

   You declare a delegate using the Delegate keyword followed by the signature of the method that the delegate will reference. For example:

   ```
   Public Delegate Sub MyDelegate(ByVal message As String)
   ```

   This declares a delegate named **MyDelegate** that can reference methods accepting a **String** parameter and returning **Sub**.
2. **Delegate Instance Creation:**

   You can create an instance of a delegate and associate it with a method that has a compatible signature. For example:

   ```
   Dim del As MyDelegate = AddressOf MyMethod
   ```

   Here, MyMethod is a method with the same signature as MyDelegate.
3. **Invoking a Delegate:**

   You can invoke a delegate instance just like you would invoke a regular method:

   ```
   del("Hello, world!")
   ```

   This will call the method that del references, passing the string "Hello, world!" as an argument.

Exception Handling

Exception is an unwanted or unexpected event which occurs during the execution of a program i.e., at run time that disturbs the normal flow of the program.

Exception classes in .net framework

- **System.Exception Class**: This is the base class for all exceptions in the .NET Framework. It provides properties such as Message, StackTrace, InnerException, and HelpLink to assist in handling exceptions effectively.

  Common Exception Classes:
- **System.NullReferenceException:** Occurs when attempting to dereference a null object reference.

- **System.ArgumentException:** Thrown when one or more arguments provided to a method are not valid.
- **System.IO.IOException:** Represents an error related to file I/O operations.
- **System.InvalidOperationException:** Indicates that a method was called at an invalid or inappropriate time or state.
- **System.DivideByZeroException:** Occurs when attempting to divide an integer or decimal by zero.
- **System.FormatException:** Thrown when a format of an argument is invalid.

Types of Exception Handling

- Unstructured Exception Handling
- Structured Exception Handling

Unstructured Exception Handling

In Unstructured exception handling is implemented using the Error object and three statements On Error Resume and Next. The On Error statements to tell VB.NET where to transfer control to in case there's been exception.

```
On Error Goto Handler
 ⋮
 Exit Sub
Handler:
⋮
End Sub
```

Parts of On Error

On Error GoTo Line:-

Error-handling code starts at the line specified in the required line argument. The line argument is any line label or line number. If an Exception occurs, program execution goes to the given location.

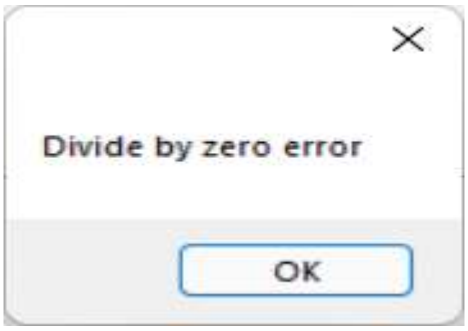
Resume Next :-

The Resume Next statement specifies that in the event of a run-time error, control passes to the statement immediately following the one in which the error occurred. At that point, execution continues.

Example:

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

```
Dim int1 As Integer=0
Dim int2 As Integer=10
Dim int3 As Integer
 On Error Goto Handler
int3 = int2 / int1
MessageBox.Show("The answer is {0}" & int3)
Handler:
 MessageBox.Show ("Divide by zero error")
 Resume Next
End Sub
 End class
```



Divide by zero error

Structured Exception Handling

VB.NET utilizes the .NET Framework's standard mechanism for error reporting, called Structured Exception Handling.

Structured exception handling provides the following components in the code:

Try section: The block of code that may result in an exception and always gets executed

Catch section: The block of code that attempts to act on an exception and is only executed when an exception takes place.

Finally section: The "Finally" block is useful for releasing resources, closing files or connections, or performing any other cleanup tasks that need to be done whether or not an exception occurs. It helps ensure that your application maintains its integrity and releases resources properly, even in the event of errors.

The Try...Catch Block

The purpose of the Try... Catch block is to allow catching errors and specifying a resolution for them. The sample code looks like this:

Try

'Code to be executed

Catch

'Error resolution code

End Try

Use the Try section to write the code that should be executed and the Catch section to catch and act on any errors that may have been generated while executing the code in the Try section. The protected code appearing in the Try section always gets executed; however, the code in the Catch section is executed only if an error occurs.

The Try...Catch... Finally Block

The purpose of the Try... Catch... Finally block is to allow executing the protected code under the Try e section, acting on any errors that may arise in the Catch block, and following up with the cleanup code in the Finally block.

The sample code looks like this:

Try

'Code to be executed

Catch

'Error resolution code

Finally

'Cleanup code

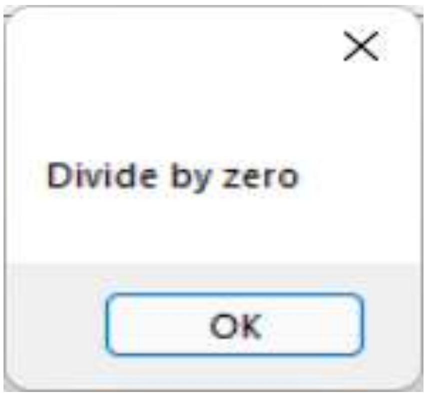
End Try

The Try and Finally sections of the code always get executed. However, the Catch section of the code will be executed only if an error occurred.

Example:

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


Dim int1 As Integer=0

Dim int2 As Integer=10

Dim int3 As Integer

Try

int3 = int2 / int1
```

```
MsgBox.show ("The answer is {0}" & int3)
 Catch ex As ArithmeticException
MsgBox.show (“Divide by zero”)
End Try
End Sub
End Class
```


Divide by zero

Tracing Errors

Tracing in a VB.NET Windows application involves monitoring and recording the execution flow, variable values, and other relevant information during runtime for debugging and analysis purposes. Trace class available from **System.Diagnostics.Trace.** This class allows you to write trace messages to various outputs such as the Output window, log files, or custom listeners.

```
' Enable tracing in your application (e.g., in the application startup)
System.Diagnostics.Trace.Listeners.Add(New
System.Diagnostics.TextWriterTraceListener("trace.log"))
System.Diagnostics.Trace.AutoFlush = True
```

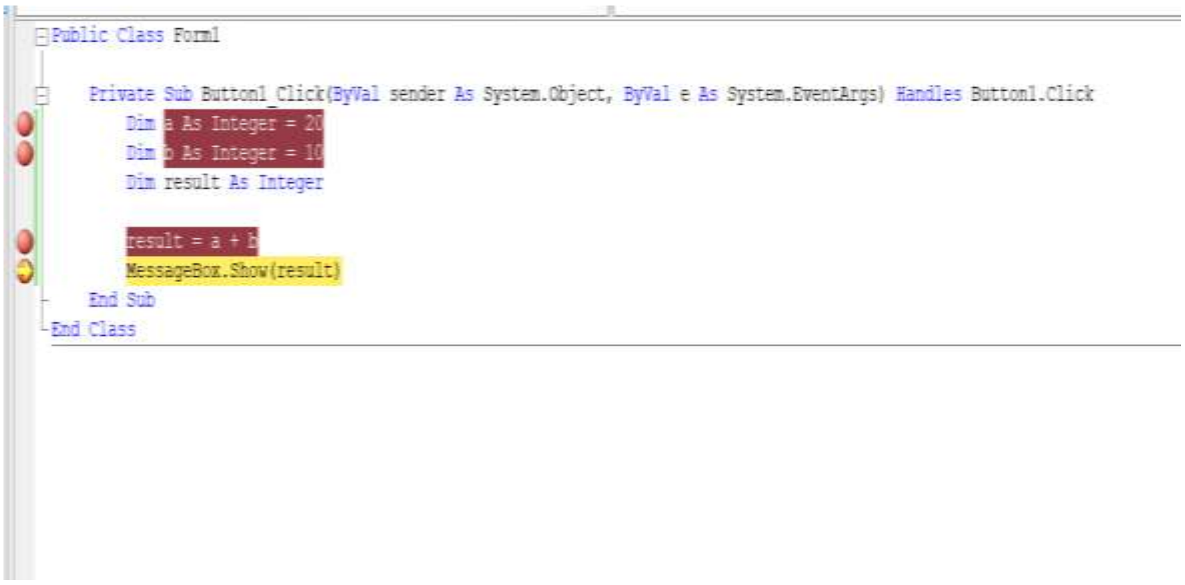
Writing Trace Messages: Use the Trace.WriteLine method to write trace messages at different points in your code.

```
' Write trace messages at different points in your code
System.Diagnostics.Trace.WriteLine("Entering SomeMethod")
' Your code logic
System.Diagnostics.Trace.WriteLine("Exiting SomeMethod")
```

Breakpoints

Breakpoint is place to purposely pause debugger execution. Breakpoint are activated only in debug mode. Breakpoints are used to observe the values of variables, stepping through a code i.e., debugging our code into single line, checking whether branch of code is running or not. Breakpoints can be set on executable Statements like variable assignment(x=3.14), loop or inside loop etc. Breakpoints cannot be set on declarative Statements like method signatures, declaration for namespace or class, variable declaration if there is no assignment.

When you click in the margin to left of a line of code, a brown circle appears. The line where you want to break is highlighted as brown shown in below fig.



```
Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        Dim a As Integer = 20
        Dim b As Integer = 10
        Dim result As Integer

        result = a + b
        MessageBox.Show(result)
    End Sub
End Class
```

[Fig: code with Breakpoint added]

Watch Window

The Watch window allows to watch the value of variables, objects, collections(Array elements, List elements) etc. It is also used to watch expressions that includes arithmetic expressions, calling methods etc. It executed during debugging mode. It provides real-time updates on the state of these elements as your code runs.

Example:

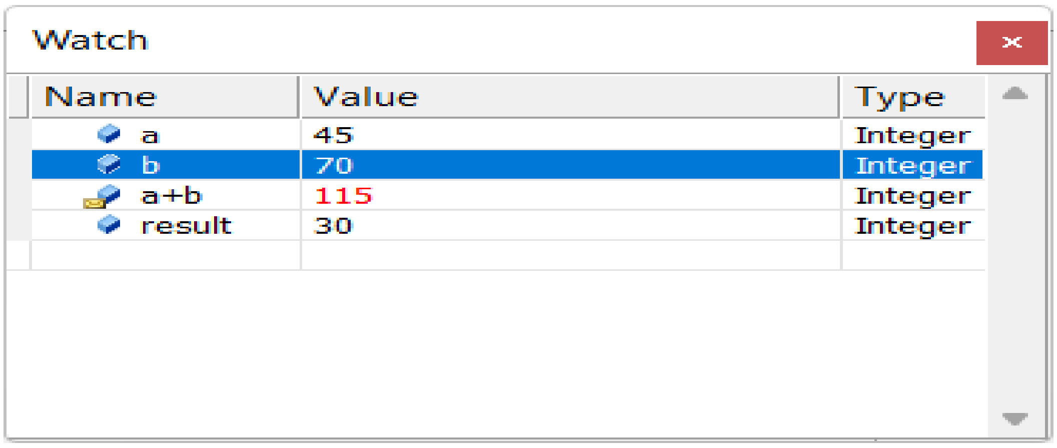
Public Class Form1

Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click

Dim a As Integer = 5

```
        Dim b As Integer = 10
        Dim result As Integer

        ' Set a breakpoint on the next line to observe variables in the Watch window
        result = a + b
    End Sub
End Class
```

We have a simple Windows Form (Form1) with a button (Button1). When the button is clicked, it triggers the Button1_Click event handler. Inside the event handler, we have three variables: a, b, and result. We perform a simple addition operation result = a + b. To observe the values of a, b, and result during execution, you can set a breakpoint on the line result = a + b. Then, while debugging, you can open the Watch window, add these variables to it, and observe how their values change as the code executes. As shown in fig below



Name	Value	Type
a	45	Integer
b	70	Integer
a+b	115	Integer
result	30	Integer

[Fig: Watch Window]

Quick Watch

In VB.NET Windows applications, the Quick Watch window allows you to quickly inspect the value of a variable or an expression during debugging without adding it to the Watch window permanently. It is use to observe a single variable at a time. It will show a pop up dialog box so we cannot continue debugging before closing the Quick watch window itself.

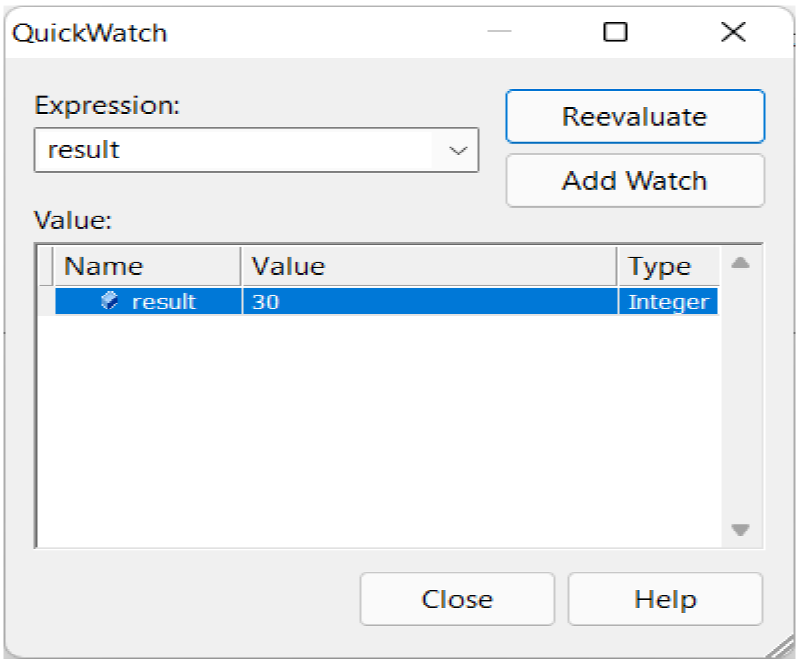
```
Public Class Form1
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

```
        Dim a As Integer = 20

        Dim b As Integer = 10

        Dim result As Integer


        ' Set a breakpoint on the next line to observe variables in the Watch window

        result = a + b

    End Sub

End Class
```

Here's how you can use the Quick Watch window with the provided code:

1. Set a breakpoint on the line result = a + b by clicking in the margin to the left of the line number in Visual Studio.
2. Start debugging your application and click the button to trigger the breakpoint.
3. Once the debugger stops at the breakpoint, right-click on the variable result.
4. Select "Quick Watch" from the context menu.

In the Quick Watch window, you'll see the value of result, which should be 30 in this case, as result = a + b. As shown in below fig:



Expression: result

Value:

Name	Value	Type
result	30	Integer

[Fig: Quick Watch window]