Practical-01: Familiarization with IDE

.NET provides a fast and modular platform for creating many different types of applications that run on Windows, Linux, and macOS. Use Visual Studio Code with the C# and F# extensions to get a powerful editing experience with C# IntelliSense, F# IntelliSense (smart code completion), and debugging.

Setting up VS Code for .NET development

.NET Coding Pack

To help you set up quickly, you can install the **.NET Coding Pack**, which includes VS Code, the .NET Software Development Kit, and essential .NET extensions. The Coding Pack can be used as a clean installation, or to update or repair an existing development environment.

Install the .NET Coding Pack - Windows

Install the .NET Coding Pack - macOS

Installing extensions

If you are an existing VS Code user, you can also add .NET support by installing the .NET Extension Pack, which includes these extensions:

- C# for Visual Studio Code
- Ionide for F#
- Jupyter Notebooks
- Polyglot Notebooks

You can also install extensions separately.

Installing the .NET Software Development Kit

If you download the extensions separately, ensure that you also have the .NET SDK on your local environment. The .NET SDK is a software development environment used for developing .NET applications.

Install the .NET SDK

VB.NET COMPONENTS

VISUAL BASIC IDE contains different components. These components are:

- Tool Bar
- Form Window
- Tool Box
- Property Window
- Project Explorer Window
- Menu Bar

Tool Bar : It provides quick access to commonly used commands in the programming environment. You click a button on the toolbar to carry out the action represented by that button. The Standard toolbar is displayed when you start Visual Basic**.**

Form Window: Form objects are the basic building blocks of Visual Basic application. It is the actual window with which a user interacts at the start of application.

Tool Box: You use special tools, called controls, to add elements of a program user interface to a form. You can find these resources in the toolbox, which is typically located along the left side of the screen. If the toolbox is not open, display it by using the Toolbox command on the View menu.

Property Window: With the Properties window, you change the characteristics (property settings) of the user interface elements on a form. A property setting is a characteristic of a user interface object. For example, you can change the text displayed by a text box control to a different font, point size, or alignment**.**

Project Explorer Window: A Visual Basic program consists of several files that are linked together to make the program run. The Visual Basic 6.0 development environment includes a Project window to help you switch back and forth between these components as you work on a program.

Menu Bar: It is a horizontal strip that appears across the top of the screen. Menu Bar lists the menus that you can use in the active window. You can modify the menu bar using the Commands tab of the Customize dialog box.

Examples of code:

```
'Program to print "Hello World" in VB.NET.
Module Module1
       Sub Main()
        Console.WriteLine("Hello World")
        Console.ReadLine()
    End Sub
End Module
```

Output:

```
Hello world
```

Explanation:

In the above program, we created a *Module* that contains the *Main()* method, here we printed the **"Hello World"** message using *WriteLine()* method of *Console* class on the console screen.......

Practical-02: Programming console applications using vb.net covering all the aspects of vb.net fundamental

A VB.Net program consists of the following modules:

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

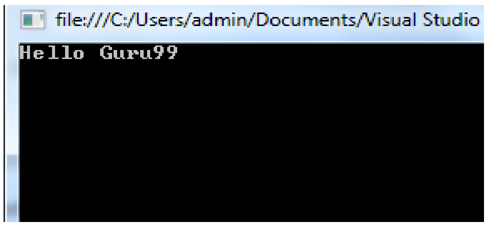
Hello World Program Example in VB.Net

Step 1) Create a new console application.

Step 2) Add the following code:

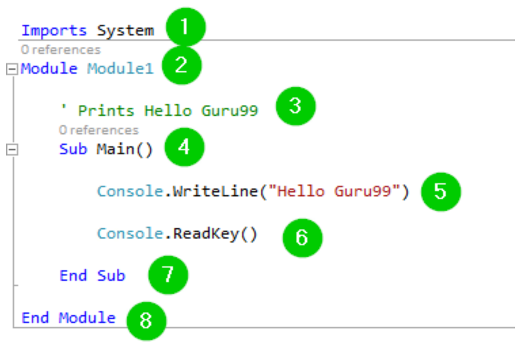
```
Imports System
Module Module1

    'Prints Hello Guru99
    Sub Main()


Console.WriteLine("Hello
Guru99")
        Console.ReadKey()

    End Sub
End Module
```

Step 3) Click the Start button from the toolbar to run it. It should print the following on the console:



```
Hello Guru99
```

Let us discuss the various parts of the above program:



```
Imports System
0 references
Module Module1

    ' Prints Hello Guru99
    0 references
    Sub Main()

        Console.WriteLine("Hello Guru99")

        Console.ReadKey()

    End Sub

End Module
```

Explanation of Code:

1. This is called the namespace declaration. What we are doing is that we are including a namespace with the name System into our programming structure. After that, we will be able to access all the methods that have been defined in that namespace without getting an error.
2. This is called a module declaration. Here, we have declared a module named Module1. VB.Net is an object-oriented language. Hence we must have a class module in every program. It is inside this module that you will be able to define the data and methods to be used by your program.
3. This is a comment. To mark it as a comment, we added a single quote (‘) to the beginning of the sentence. The VB.Net compiler will not process this part. The purpose of comments is to improve the readability of the code. Use them to explain the meaning of various statements in your code. Anyone reading through your code will find it easy to understand.
4. A VB.Net module or class can have more than one procedures. It is inside procedures where you should define your executable code. This means that the procedure will define the class behavior. A procedure can be a Function, Sub, Get, Set, AddHandler, Operator, RemoveHandler, or RaiseEvent. In this line, we defined the Main sub-procedure. This marks the entry point in all VB.Net programs. It defines what the module will do when it is executed.
5. This is where we have specified the behavior of the primary method. The WriteLine method belongs to the Console class, and it is defined inside the System namespace. Remember this was imported into the code. This statement makes the program print the text Hello Guru99 on the console when executed.
6. This line will prevent the screen from closing or exiting soon after the program has been executed. The screen will pause and wait for the user to perform an action to close it.
7. Closing the main sub-procedure.
8. Ending the module.

Practical-03: Object oriented programming using vb.net covering objects, inheritance, polymorphism. constructors, static classes, and interfaces.

Visual Basic provides full support for object-oriented programming including encapsulation, inheritance, and polymorphism.

Classes and objects: The terms *class* and *object* are sometimes used interchangeably, but in fact, classes describe the *type* of objects, while objects are usable *instances* of classes. So, the act of creating an object is called *instantiation*. Using the blueprint analogy, a class is a blueprint, and an object is a building made from that blueprint.

***To define a class*:**

```
Class SampleClass

End Class
```

Inheritance: Inheritance enables you to create a new class that reuses, extends, and modifies the behavior that is defined in another class. The class whose members are inherited is called the *base class*, and the class that inherits those members is called the *derived class*. However, all classes in Visual Basic implicitly inherit from the Object class that supports .NET class hierarchy and provides low-level services to all classes.

First basic program:

```
Base class
Class Hello
Public Sub sayHelloWorld()
Console.WriteLine("Hello World")
End Sub
End Class
Derived class
Class Welcome: Inherits Hello
Public Sub sayWelcome()
Console.WriteLine("Welcome")
End Sub
End Class
Class CallAllFunctions
Shared Sub Main()
Dim a As Welcome = new Welcome()
a.sayHelloWorld()
```

Output:

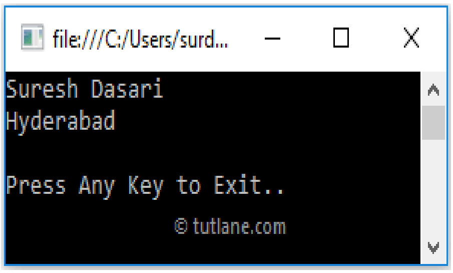
```
Hello World
Welcome
```

Constructors: Constructors are class methods that are executed automatically when an object of a given type is created. Constructors usually initialize the data members of the new object. A constructor can run only once when a class is created. Furthermore, the code in the constructor always runs before any other code in a class. However, you can create multiple constructor overloads in the same way as for *any other method.*

To define a constructor for a class:

```
Module Module1
    Class User
        Public name, location As String
        ' Default Constructor
        Public Sub New()
            name = "Suresh Dasari"
            location = "Hyderabad"
        End Sub
    End Class
    Sub Main()
        ' The constructor will be called automatically
once the instance of the class created
        Dim user As User = New User()
        Console.WriteLine(user.name)
        Console.WriteLine(user.location)
        Console.WriteLine("Press Enter Key to Exit..")
        Console.ReadLine()
    End Sub
End Module
```

Output:



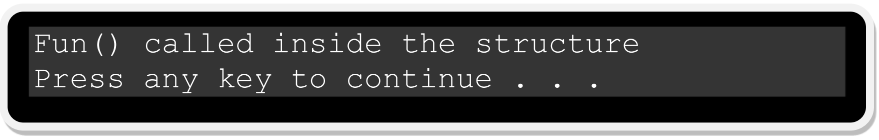
```
Suresh Dasari
Hyderabad

Press Any Key to Exit..
```

Interfaces: Interfaces, like classes, define a set of properties, methods, and events. But unlike classes, interfaces do not provide implementation. They are implemented by classes, and defined as separate entities from classes. An interface represents a contract, in that a class that implements an interface must implement every aspect of that interface exactly as it is defined.

To define an interface:

```
Module Module1
    Interface ISample
        Sub Fun()
    End Interface

    Structure Sample
        Implements ISample
        Sub Fun() Implements ISample.Fun
            ' Method Implementation
            Console.WriteLine("Fun()
called inside the structure")
        End Sub
    End Structure

    Sub Main()
        Dim S As New Sample()
```

Output:



```
Fun() called inside the structure
Press any key to continue . . .
```

Polymorphism: Polymorphism is the ability to define a method or property in a set of derived classes with matching method signatures but provide different implementations and then distinguish the objects' matching interface from one another at runtime when you call the method on the base class For example:

```
Module Module1

Sub Main()
Dim two As New One()
WriteLine(two.add(10))
'calls the function with one argument
WriteLine(two.add(10, 20))
'calls the function with two arguments
WriteLine(two.add(10, 20, 30))
'calls the function with three arguments

End Sub

End Module

Public Class One
Public i, j, k As Integer

Public Function add(ByVal i As Integer) As Integer
'function with one argument
Return i
End Function

Public Function add(ByVal i As Integer, ByVal j As Integer) As Integer
'function with two arguments
Return i + j
End Function

Public Function add(ByVal i As Integer, ByVal j As Integer, ByVal k As Integer) As
Integer
'function with three arguments
Return i + j + k
End Function
```

Program-04: Programme to illustrate exception handling concepts in VB.NET

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

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

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

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

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

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

Some of the exception classes derived from the System.Exception class are the **System.ApplicationException** and **System.SystemException** classes.

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

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

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

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

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

Throwing Objects:

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

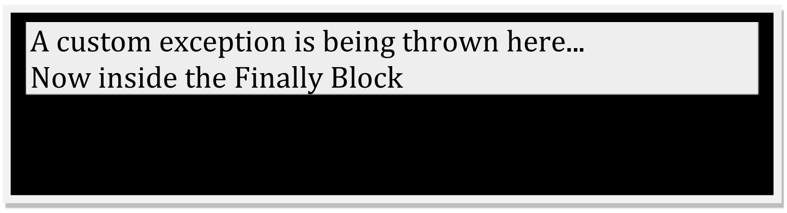
You can use a throw statement in the catch block to throw the present object as −

```
Throw [ expression ]
```

The following program demonstrates this −

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

Output:



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

Practical-05: Programme to illustrate use of collections in VB.NET

A collection is a class, so you must declare an instance of the class before you can add elements to that collection.

Collections provide a more flexible way to work with groups of objects. Unlike arrays, the group of objects you work with can grow and shrink dynamically as the needs of the application change. For some collections, you can assign a key to any object that you put into the collection so that you can quickly retrieve the object by using the key.

If your collection contains elements of only one data type, you can use one of the classes in the System.Collections.Generic namespace. A generic collection enforces type safety so that no other data type can be added to it. When you retrieve an element from a generic collection, you do not have to determine its data type or convert it.

```
' Create a list of strings.
Dim salmons As New List(Of String)
salmons.Add("chinook")
salmons.Add("coho")
salmons.Add("pink")
salmons.Add("sockeye")

' Iterate through the list.
For Each salmon As String In salmons
    Console.Write(salmon & " ")
Next


'Output: chinook coho pink sockeye
```

If the contents of a collection are known in advance, you can use a *collection initializer* to initialize the collection. For more information, see Collection Initializers.The following example is the same as the previous example, except a collection initializer is used to add elements to the collection.

```
' Create a list of strings by using a
' collection initializer.
Dim salmons As New List(Of String) From
    {"chinook", "coho", "pink",
"sockeye"}

For Each salmon As String In salmons
    Console.Write(salmon & " ")
Next

    'Output: chinook coho pink sockeye
```

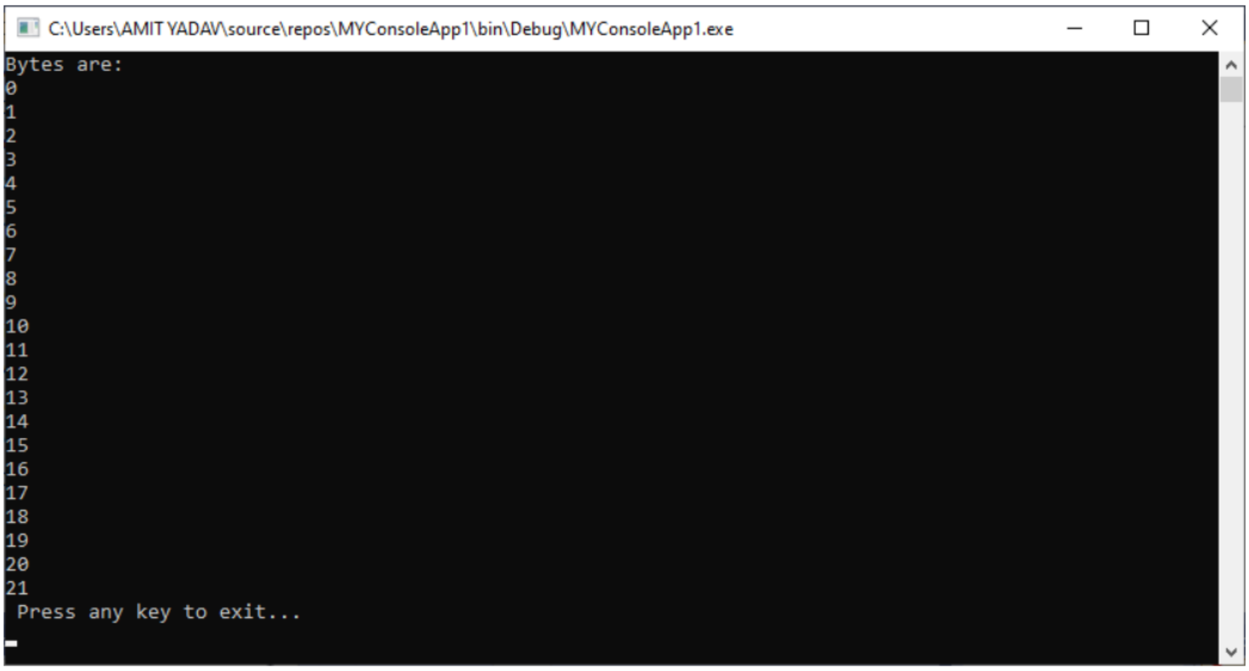
Practical-06: Programme to perform File I/O operations.

File and stream I/O (input/output) refers to the transfer of data either to or from a storage medium. In .NET, the System.IO namespaces contain types that enable reading and writing, both synchronously and asynchronously, on data streams and files. These namespaces also contain types that perform compression and decompression on files, and types that enable communication through pipes and serial ports.

A file is an ordered and named collection of bytes that has persistent storage. When you work with files, you work with directory paths, disk storage, and file and directory names. In contrast, a stream is a sequence of bytes that you can use to read from and write to a backing store, which can be one of several storage mediums (for example, disks or memory). Just as there are several backing stores other than disks, there are several kinds of streams other than file streams, such as network, memory, and pipe streams.

```
Imports System.IO
Module File_Prog
  Sub Main()
' Create an object FS for the FileStream class along with the name of the text file to perform operation like create, read or write.
    Dim FS As FileStream = New FileStream("myFile.txt", FileMode.OpenOrCreate, FileAccess.ReadWrite)
    Dim c As Integer
    ' use for loop to read character
    For c = 0 To 21
      FS.WriteByte(CByte(c))  'write byte to the file
    Next c
    FS.Position = 0
    Console.WriteLine("Bytes are:")
    For c = 0 To 21
    Console.WriteLine("{0} ", FS.ReadByte()) ' ReadByte() to read byte form the fie.
    Next c
    FS.Close()  'Close the file
    Console.WriteLine(" Press any key to exit...")
    Console.ReadKey()
  End Sub
End Module
```

Output:



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

```
Bytes are:
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
 Press any key to exit...
```

Practical-07: Programming Windows applications using VB.NET covering all major controls and components Menus, MDI, Event Handling.

MDI Applications

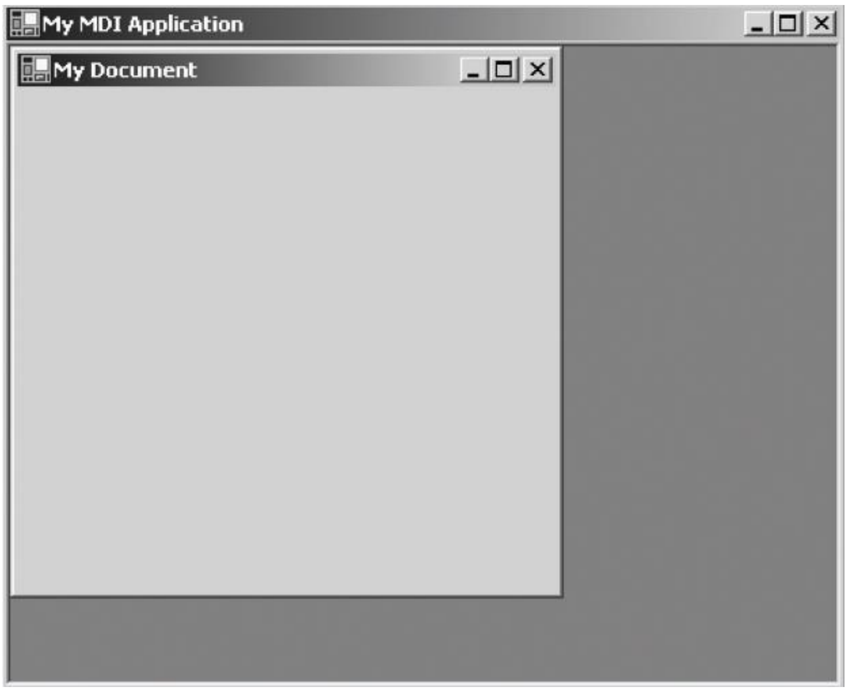
Multiple document interface (MDI) applications permit more than one document to be open at a time. This is in contrast to *single document interface* (SDI) applications, which can manipulate only one document at a time. Visual Studio .NET is an example of an MDI application—many source files and design views can be open at once. In contrast, Notepad is an example of an SDI application—opening a document closes any previously opened document.

There is more to MDI applications than their ability to have multiple files open at once. The Microsoft Windows platform SDK specifies several UI behaviors that MDI applications should implement. The Windows operating system provides support for these behaviors, and this support is exposed through Windows Forms as well.

```
Imports System
Imports System.Windows.Forms

Public Module AppModule
  Public Sub Main( )
    Application.Run(New MainForm( ))
  End Sub
End Module

Public Class MainForm
  Inherits Form

  Public Sub New( )
    ' Set the main window caption.
    Text = "My MDI Application"
    ' Set this to be an MDI parent form.
    IsMdiContainer = True
    ' Create a child form.
    Dim myChild As New DocumentForm("My Document", Me)
    myChild.Show
  End Sub

End Class

Public Class DocumentForm
  Inherits Form
```

```
    Public Sub New(ByVal name As String, ByVal parent As Form)
      ' Set the document window caption.
      Text = name
      ' Set this to be an MDI child form.
      MdiParent = parent
    End Sub

End Class
```

Output:


My MDI Application
My Document