PROCEDURES

Procedure
This refers to a self-contained group of Visual Basic instructions/statements that is made to perform a designated task i.e. each procedure's functionality should be limited to the performance of a single, well-defined task. Procedures facilitate code reuse because they can be called by event handlers and other procedures
The classes supplied by the .NET Framework class library all contain procedures and they can appear in a **Class** or **Module** block

Modularity
Effective programs include modularity. Each "module" is composed of Subroutines (Methods, Procedures and Functions). Using modularity in programs increases: clarity, organization, manageability and reusability

Procedures and Access Modifiers
Access modifiers control the visibility of procedures

- A **Private** procedure can only be called from the module containing the procedure declaration
- Procedures declared with the **Public** and **Friend** access modifiers can be called by other modules and classes

Types of Procedures
There are two types of **general procedures**:

1. **Sub procedures** that perform actions, but are not associated with any specific event such as a click event and don't return a value to the calling procedure. The **Exit Sub** statement causes the sub procedure to exit immediately

   Syntax:

```
[Public|Private] Sub ProcedureName([arguments])
     Declarations & statements
End Sub
```

   - ***ProcedureName*** is the name of the procedure being created.
   - ***Arguments*** is a list of optional arguments (separated by commas) to be used in the procedure.
   - ***Declarations*** define variables and constants used within the procedure.
   - ***Statements*** is the block of statements that accomplish the work of the procedure.

2. **Function procedures** (also called **user-defined functions** [UDFs] or simply **functions**) that perform actions, which are not associated with any specific event and also return a value. You use functions to calculate values that are returned to the calling procedure . The **Exit Function** statement causes the function to exit immediately

   Syntax:

```
[Public|Private]Function FunctionName([arguments]) [As Return Type]
     function statements
     [Return expression]
End Function
```

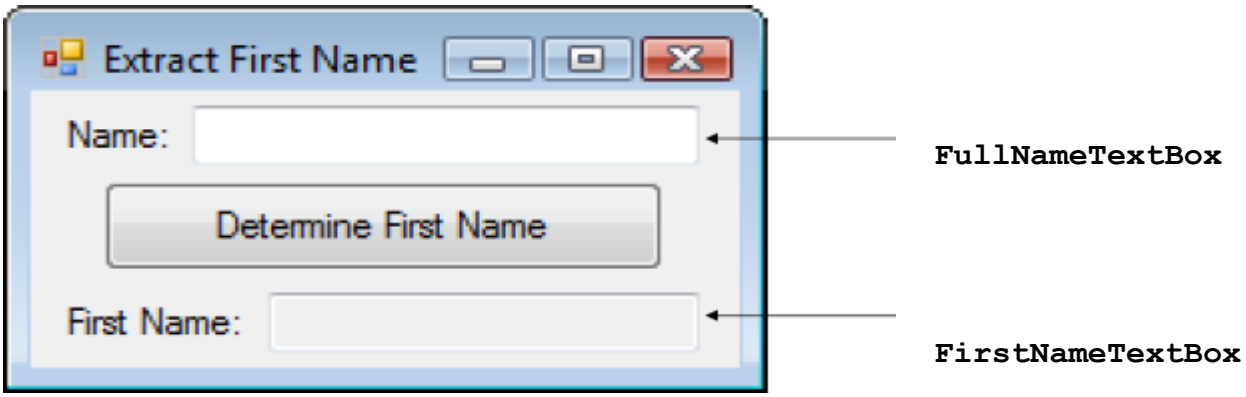
   - ***FunctionName*** is the name of the function being created.
   - ***arguments*** is a list of optional arguments (separated by commas) to be used in the function.
   - ***As Type*** option that specifies the function's return type
   - ***Function statements*** is a block of statements that include the local declarations and accomplish the work of the function

Creating a Sub Procedure
To create a sub procedure, you need to type the following line and press the Enter key (give the sub procedure some unique, meaningful name):

```
Private Sub ProcedureName
```

This causes VB to add parentheses and the **End Sub** statement. You can then add your code inside the procedure.

Example of a Program with General Sub Procedure
Suppose the program below was divided between two or more computer programmers. One programmer might be assigned to write the code to extract the first name from the fullname – this divides up the work load for a large programming project. The **FirstName** procedure will be executed by calling it from the **Determine Button**




```
Public Class ExtractFirstNameForm
    Private Sub DetermineButton_Click(ByVal sender As Object, ByVal
    e As EventArgs) Handles DetermineButton.Click
        Dim name As String
        name = FullNameTextBox.Text
        Call FirstName(name)
    End Sub

    Private Sub FirstName(ByVal name As String)
        Dim firstSpace As Integer
        firstSpace = name.IndexOf(" ")
        FirstNameTextBox.Text = name.Substring(0, firstSpace)
    End Sub
End Class
```

Creating a Function Procedure
Create a function by typing:

```
Private Function FunctionName() As Datatype
```

You must specify the **Datatype** that the function returns – this is done with the **As Datatype** clause in the function declaration.

Example
Write a complete Visual Basic program that accepts the radius of a sphere using a textbox control upon the click of a button control. The program should then use a function procedure to calculate its volume, which is then displayed in a textbox control. Use the formula shown below for volume calculation:

$$V = \frac{4}{3}\pi r^3$$

```
Public Class SpherForm
    Private Function ComputeVolume(ByVal SRadius As Double) As Double
      ComputeVolume = 4/3 * Math.PI * SRadius ^ 3
    End Function

    Private Sub VolumeButton_Click(ByVal sender As Object, ByVal e As
    System.EventArgs) Handles VolumeButton.Click
         Dim Radius, Volume As Double
         Radius = Double.Parse(RadiusTextBox.Text,
              Globalization.NumberStyles.Number)
         Volume = ComputeVolume(Radius)
         VolumeTextBox.Text = Volume.ToString("N2")
     End Sub
End Class
```

In the example above the function named **ComputeVolume** receives a **double parameter** that is passed to the function. The value incoming to the function is stored in the parameter variable named **SRadius**.

Function Return

There are two ways to return a value from a function:

1. Set the function name to the value to be returned inside the function.
2. Use the optional **Return** statement as is done below.

```
Private Function ComputeVolume(ByVal CRadius As Double) As Double
  Dim SphereVolume As Double
  SphereVolume = 4/3 * Math.PI * Radius ^ 3
  Return SphereVolume
End Function
```

Notice that the Return statement terminates execution of the method and returns the value. It can occur anywhere in method/procure

Calling a Function

Procedures are called by name and optionally a list of arguments follows this name.

- A function that returns a value must be used in on the right-hand side of an assignment statement.
- The value returned is stored to the variable in the assignment statement
- The parameter(s) must be passed to the function by the calling statement.

Both sub and function procedures can have zero, one, or more arguments in the parameter list. The calling and called procedure must pass/receive the same type of data for each argument in the parameter list.

Types of Parameters

There are two types of parameters:

i). **Actual Parameters: -** This are substituted for the formal parameter at the time the procedure is called. Each data type must be assignment compatible with the data type of its corresponding formal parameter.

ii). **Formal Parameters: -** This a list of "place marker" names used in the procedure's declaration. It can include the data type of the valued parameters.

Rules about Passing Arguments (Parameters)

- Arguments passed to a sub/function procedure must be the **same type of data** that the sub/function procedure receives into a parameter.
- Argument values **must be passed** if the sub/function procedure has an argument list.
- In the procedure call, the number and type of arguments sent to the sub/function procedure must match the number and type of arguments in the sub declaration.

Example:

Consider the following sub procedure:

```
Private Sub PrintPay(hours As Short, wage As Double)
        Dim GrossPay As Double
        GrossPay = hours * wage
End Sub
```

The following will be the expected procedure call

```
PrintPay(40, 10.00)
```

Or

```
Call PrintPay(40, 10.00)
```

Argument Passing Mechanism

Parameters can receive an argument value using either **ByVal** or **ByRef**.

1). **Call-by-Value (ByVal**) sends a copy of the argument's value to the procedure so the procedure does not alter the original copy of the value. Parameters can also be defined as ByVal in the Call Statement by using two sets of parentheses. Because Call-by-value makes a copy of the value within a variable, it could be memory consuming if the value is large.

2). **Call-by-Reference (ByRef)** sends the memory location of the value to the procedure so that if the value is modified, the original copy is modified.

Examples:

```
Private Function ComputeVolume(ByVal CRadius As Double) As Double

Private Function ComputeVolume(ByRef CRadius As Double) As Double
```

Using Regions to Organize Code

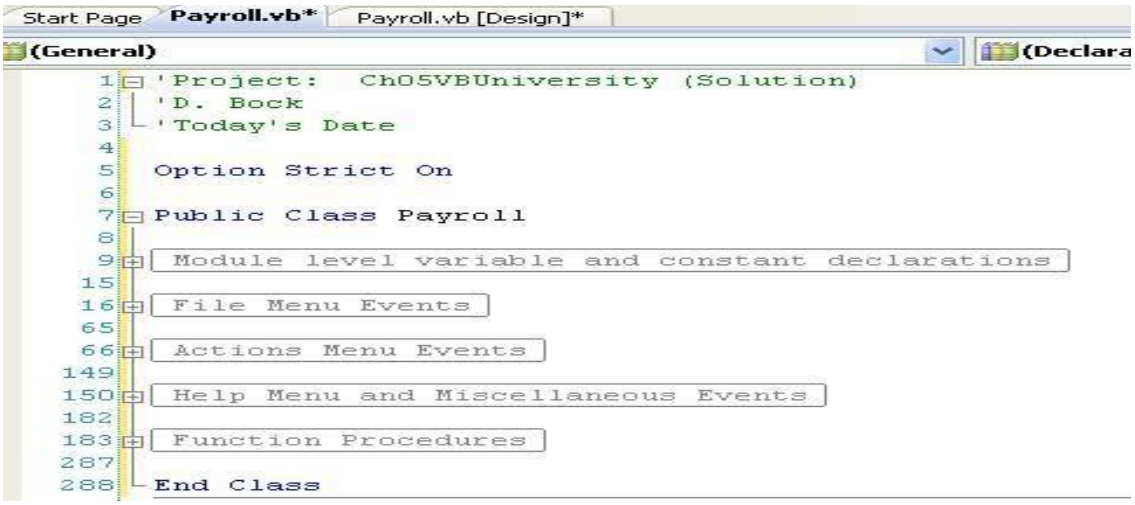
The **#Region** statement can be used to organize the code in a program as the size of the program grows.

- This **#Region** statement is used to create a region for the storage of sub and function procedures.
- The sub and function procedures can be cut/pasted into the region and the region can be collapsed.
- One approach for organizing code is to organize according to functionality.

```
#Region "Function Procedures"

#End Region
```

This figure shows the main program organized with regions.



E.g

```
'Project:  VBUniversity (Solution)
```

```
'Author name
'Today's Date
Option Strict On
 Public Class Payroll
 #Region "Module level variable and constant declarations"
     'Module level variable/constant declarations
    Private RetirementRateDecimal As Decimal
 #End Region
```

Optional Arguments

The caller has the *option* of passing a particular argument.

- Must `specify` a default value that is assigned to the parameter if the optional argument is not passed
- All optional parameters must be placed to the right of the method's non-optional parameters
- Specified in the procedure header with the keyword *Optional*.

E.g.

```
Function Power (ByVal base As Integer, Optional exponent As Long = 2)
```

Function calls

```
Call Power()    'Causes syntax error
Call Power(2)
Call Power(10,3)
```

DIALOG BOXES

1. MessageBox class and MsgBox function

Visual Studio provides both the *MsgBox* function and the *MessageBox* class for displaying text in a message box/ dialog box (i.e. displaying output to the user). The results of the function call can be assigned to a variable.

The *MessageBox* class is part of the *System.Windows.Forms* namespace; it takes arguments much like *MsgBox*, and it is displayed by using the *Show* method. This dialogs are extremely handy for software-user communication and for debugging purposes

The MessageBox Show Method

This is a method that generates a standard dialog box, which displays a message, a caption, an icon and one or more standard button groups.

Syntax

Public Shared Function Show(ByVal caption As String, ByVal title As String, ByVal buttons As MessageBoxButtons, ByVal icon As MessageBoxIcon) As DialogResult

- *caption* contains the message
- *title* appears in the title bar
- *buttons* defines the button(s)
- *icon* defines the icon

The following are enumerations that work with the MessageBox class

Enumeration	Members
MessageBoxButtons	OK, OKCancel, YesNo, YesNoCancel, AbortRetryIgnore
MessageBoxIcon	None, Information, Error, Warning, Exclamation, Question, Asterisk, Hand, Stop
MessageBoxDefaultButton	Button1, Button2, Button3
DialogResult	OK, Cancel, Yes, No, Abort, Retry, Ignore

Example 1
The following is a procedure that prompts the user by displaying a message box with Yes and No buttons on whether to exit application or not

```
Private Sub ExitButton_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles ExitButton.Click
      'Close the form if the system user responds Yes
      Dim MessageString As String = "Do you want to close the form?"
      Dim ButtonDialogResult As DialogResult =
      MessageBox.Show(MessageString, "Quit?", MessageBoxButtons.YesNo,
      MessageBoxIcon.Question, MessageBoxDefaultButton.Button2)
      If ButtonDialogResult = Windows.Forms.DialogResult.Yes Then
            Me.Close()
      End If
 End Sub
```

MsgBox Function
This is a function used to display a dialog box, wait for the user to click a button and returns an integer value indicating which button the user clicked.

Syntax: *MsgBox (Prompt, Button, Title, HelpFile, Context)*

Buttons: This is a number expression that is the sum of values specifying the number and type of buttons to display, the icon style to use, the identity of the default button and the modality of the message box. If omitted, the default value is 0. This argument is optional.

Example 2
The following is a procedure that prompts the user by displaying a message box with Yes and No buttons on whether to exit application or not

```
Private Sub ExitButton_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles ExitButton.Click
    'Close the form if the system user responds Yes
    Dim MessageString As String = "Do you want to close the form?"
    Dim response As Integer =
    MsgBox(MessageString, vbYesNo + vbQuestion + vbDefaultButton2, "Quit?")
    If response = vbYes Then
        Me.Close()
    End If
 End Sub
```

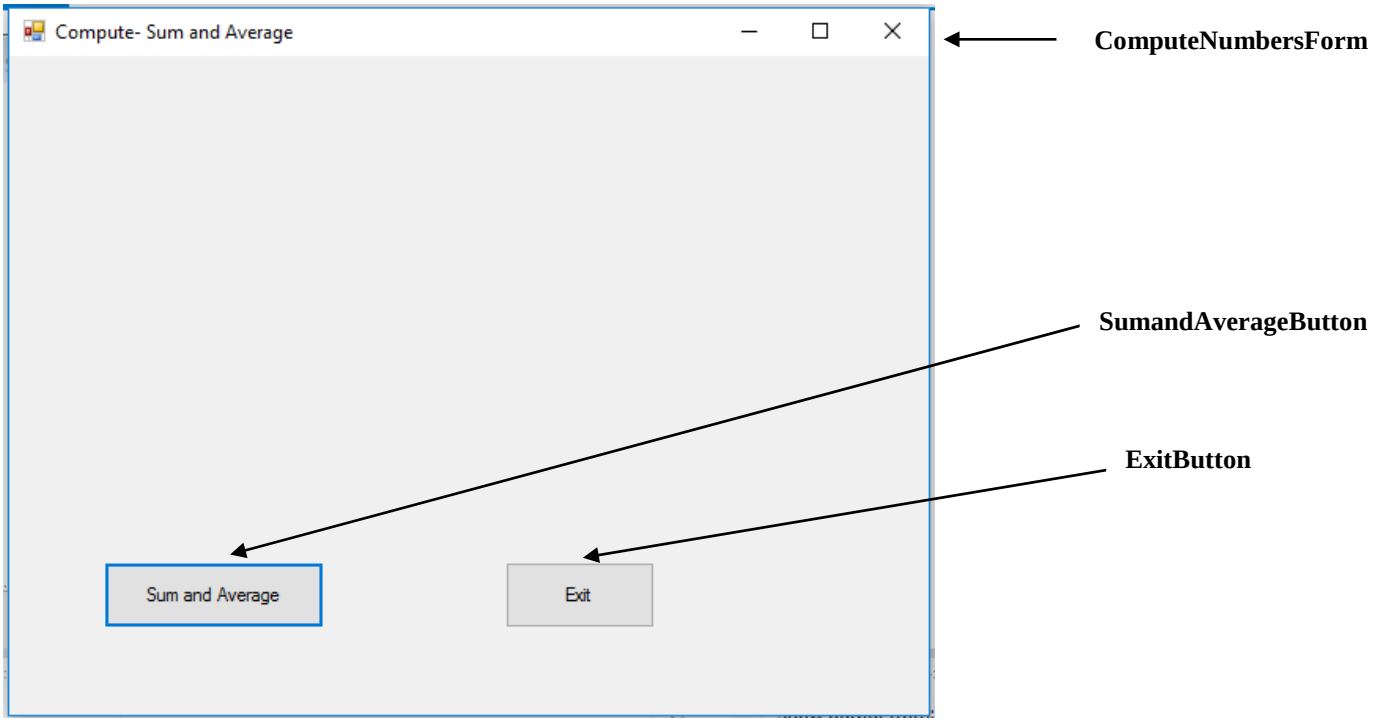
2. InputBox Dialog
This is a named Visual Basic function that displays a standard dialog box on the screen and prompts the user for input. The input is returned as a text string to the program. In addition to a prompt string, the *InputBox* function supports other arguments that you might want to use occasionally.

Syntax
Public Shared Function InputBox (ByVal prompt As String, Optional ByVal title As String, Optional ByVal defaultResponse As String, Optional ByVal xPos As Integer, Optional ByVal yPos As Integer) As String

- *prompt* contains a descriptive prompt
- *title* appears on the title bar
- *defaultResponse* contains the default value
- *xPos* and *YPos* contain the coordinate values where the input box will appear

Example
Write a Visual Basic program that inputs two integer numbers using Inputbox dialogs and computes the sum and average of the numbers after a click of a button control. The program should display the sum and average using a message box dialog. Design your GUI

Appropriate GUI



Program Code

```
Public Class ComputeNumbersForm

    Private Sub SumandAverageButton_Click(ByVal sender As Object, ByVal e As EventArgs)
    Handles SumandAverageButton.Click
        Dim Number1Integer, Number2Integer As Integer
        Dim SumInteger As Integer, AverageSingle As Single
        Number1Integer = Integer.Parse(InputBox("Enter the first number", "Inputs"))
        Number2Integer = Integer.Parse(InputBox("Enter the second number", "Inputs"))
        SumInteger = Number1Integer + Number2Integer
        AverageSingle = Convert.ToSingle(SumInteger) / 2
        MessageBox.Show("Sum is " & SumInteger.ToString() & ControlChars.NewLine &
        "Average is " & AverageSingle.ToString(), "Results", MessageBoxButtons.OK,
        MessageBoxIcon.Information)
     End Sub

End Class
```