Procedures

Procedures are units/blocks of code that typically calculate values, manipulate text or control, or interact with database operations. They are very useful for breaking down a large programs into smaller units. Most often you will use a procedure to execute repeated or shared tasks to avoid code duplication - (Code duplication is not wrong merely inefficient). You can also use them as building blocks that let you easily develop other related applications fast.

There are two major benefits of programming with procedures:

- Procedures allow you to break your programs into discrete logical units, each of which you can debug more easily than an entire program without procedures.
- Procedures used in one program can act as building blocks for other programs, usually with little or no modification.

There are several types of procedures used in Visual Basic:

- Sub procedures do not return a value.
- Function procedures return a value.
- Property procedures can return and assign values, and set references to objects.

Sub Procedure

A Sub procedure is a block of code that is executed in response to an event. By breaking the code in a module into Sub procedures, it becomes much easier to find or modify the code in your application.

The syntax for a Sub procedure is: (we will postpone the discussion on Private/Public/Static to later)

```
[Private|Public][Static]Sub procedurename (arguments)

  statements

End Sub
```

Each time the Sub procedure is called, the statements between Sub and End Sub are executed. Sub procedures can be placed in standard modules, class modules, and form modules. Sub procedures are by default Public in all modules, which means they can be called from anywhere in the application.

The arguments for a procedure are like a variable declaration, declaring values that are passed in from the calling procedure. We will explore this after we have defined variables

In VB, it's useful to distinguish between two types of Sub procedures, event procedures and general procedures.

Event Procedures When an object in VB recognizes that an event has occurred, it automatically invokes the event procedure using the name corresponding to the event. Because the name establishes an association between the object and the code, event procedures are said to be attached to forms and controls.

Syntax for a control event

```
Private Sub controlname_eventname (arguments )
   statements
End
```

Syntax for a form event

```
Sub Private Sub Form_eventname (arguments)
      statements
End Sub
```

- An event procedure for a control combines the control's actual name (specified in the Name property), an underscore (_), and the event name. For instance, if you want a command button named cmdPlay to invoke an event procedure when it is clicked, use the procedure cmdPlay_Click.

- An event procedure for a form combines the word "Form" an underscore, and the event name. If you want a form to invoke an event procedure when it is clicked, use the procedure Form_Click. (Like controls, forms do have unique names, but they are not used in the names of event procedures.) If you are using the MDI form, the event procedure combines the word "MDIForm," an underscore, and the event name, as in MDIForm_Load.

All event procedures use the same general syntax.

Although you can write event procedures from scratch, it's easier to use the code procedures provided by VB, which automatically include the correct procedure names. You can select a template in the Code Editor window by selecting an object from the Object box and then selecting a procedure from the Procedure box.

Function Procedures

A Function procedure is another kind of procedure, similar to a Sub procedure for it can take arguments, perform a series of statements, and change the value of its arguments. However, unlike a Sub procedure, a Function procedure can return a value to the calling procedure.

There are three differences between Sub and Function procedures: (books on line)

- Generally, you call a function by including the function procedure name and arguments on the right side of a larger statement or expression

  returnvalue = function()

- Function procedures have data types, just as variables do. This determines the type of the return value. (In the absence of an As clause, the type is the default Variant type.)

- You return a value by assigning it to the procedurename itself. When the Function procedure returns a value, this value can then become part of a larger expression.

Visual Basic includes built-in, or intrinsic functions, like Sqr, Cos or Chr. In addition, you can use the Function statement to write your own Function procedures.

The syntax for a Function procedure is:

[Private|Public][Static]Function procedurename (arguments) [As type]

statements

End Function

Property Procedures

Property procedures are typically use to set an object's property or to get the value of an object's property. This will be useful when we define our own classes and objects.

Here we just accept the following information

Visual Basic provides three kinds of property procedures depending on the purpose

Property Get Returns the value of a property.

Property Let Sets the value of a property.

Property Set Sets the value of an object property (that is, a property that contains a reference to an object).

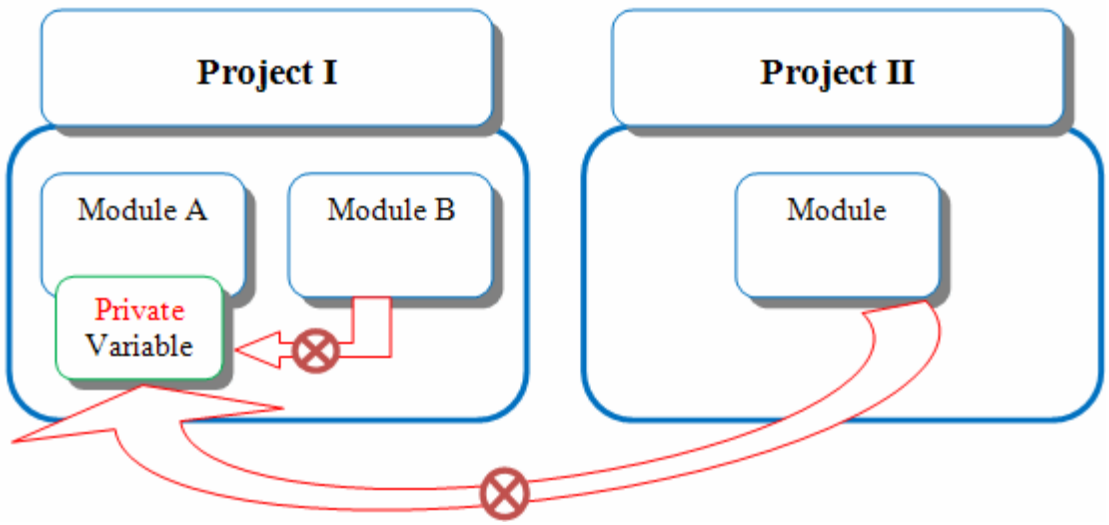
Each of these property procedures has a particular role to play in defining a property. The typical property will be made up of a pair of property procedures: A Property Get to retrieve the property value, and a Property Let or Property Set to assign a new value.

Passing Arguments

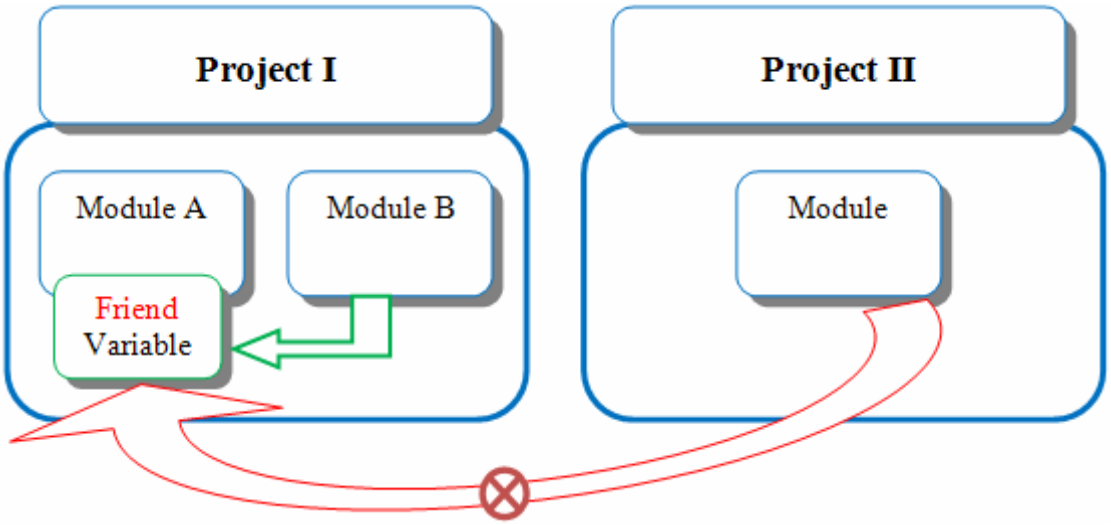
Using Global Variables

In the previous lesson, we saw that you could declare a global variable outside of any procedure. When using various procedures in a code file, one of the characteristics of a global variable is that it is automatically accessible to other procedures. Still, a global variable can use access modifiers that would control its access:

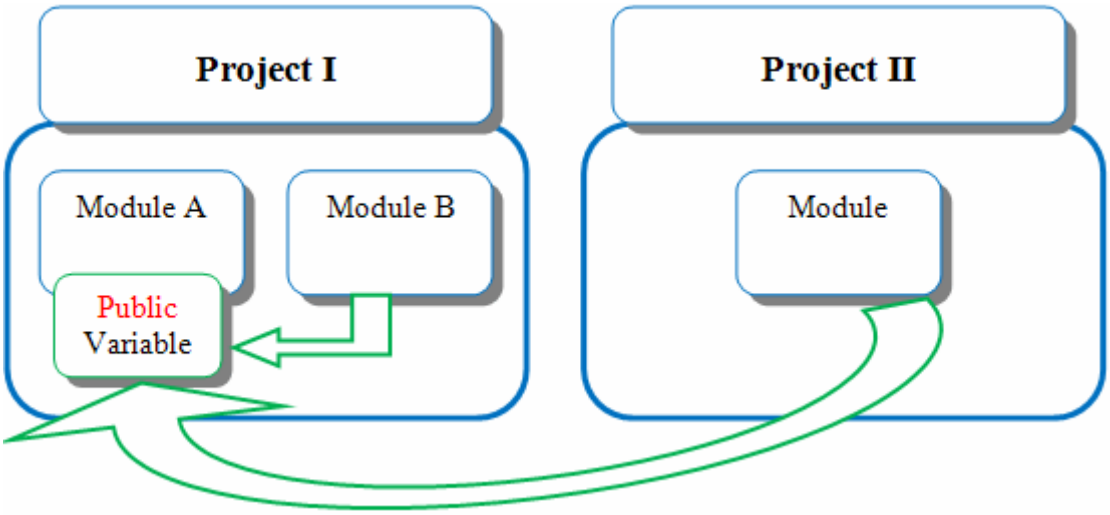
- **Private**: A private global variable can be accessed by any procedure of the same module. No procedure of another module, even of the same program, can access it



- **Friend**: A friendly global variable can be accessed by any procedure of any module of the same project. A procedure of another program cannot access that variable



- **Public**: A public global variable can be accessed by any procedure of its project and procedures of other projects



Based on this characteristic of the procedures of a module having access to global variables of the same program, you can declare such variables and initialize or modify them in any procedure of the same code file.

Practical Learning: Using Global Variables

To use global variables, change the document as follows:

```
Module Geometry
    Private Length As Double
    Private Width As Double

    Private Sub GetLength()
        Length = InputBox(&amp;quot;Enter Rectangle Length:&amp;quot;)
    End Sub

    Private Sub GetWidth()
        Width = InputBox(&amp;quot;Enter Rectangle Width:&amp;quot;)
    End Sub
```

```
    Private Function CalculatePerimeter() As Double
        CalculatePerimeter = (Length + Width) * 2
    End Function

    Public Function Main() As Integer
        Dim Perimeter As Double
        GetLength()
        GetWidth()
        Perimeter = CalculatePerimeter()

        MsgBox("=-= Square Characteristics=-=" & vbCrLf &
               "Length: " & vbTab & Length & vbCrLf &
               "Width:  " & vbTab & Width & vbCrLf &
               "Perimeter: " & Perimeter)
        Return 0
    End Function

End Module
```

To execute the program, on the Standard toolbar, click the Start Debugging button 

Enter the length as **32.08** and the with as **24.84**

Close the message box and return to your programming environment

Introduction to Arguments

So far, to use a value in a procedure, we had to declare it. In some cases, a procedure may need an external value in order to carry its assignment. A value that is supplied to a procedure is called an argument.

When creating a procedure that will use an external value, declare the argument that represents that value between the parentheses of the procedure. For a procedure, the syntax you use would be:

Sub *ProcedureName*(*Argument*)

End Sub

If you are creating a function, the syntax would be:

Function *ProcedureName*(*Argument*) As *DataType*

Function Sub

The argument must be declared as a normal variable, omitting the **Dim** keyword. Here is an example that creates a function that takes a string as argument:

Function CalculatePayroll(strName As String) As Double

Function Sub

A certain procedure can take more than one argument. In this case, in the parentheses of the procedure, separate the arguments with a comma. Here is an example of a procedure that takes two arguments:

```
Sub EvaluateInvoice(EmplName As String, HourlySalary As Currency)

End Sub
```

In the body of a procedure that takes one or more arguments, use the argument(s) as you see fit as if they were locally declared variables. For example, you can involve them with values inside of the procedure. You can also exclusively use the values of the arguments to perform the assignment.

Calling a Procedure With Argument

To call a procedure that takes an argument, type its name and a space, followed by a value for each argument between parentheses. The value provided for an argument is also called a parameter. If there is more than one argument, separate them with a comma. Here is an example:

```
Module Exercise

    Private Function GetFullName$(strFirst As String,
                                  strLast As String)
        Dim FName As String

        FName = strFirst & " " & strLast
        GetFullName = FName
    End Function

    Public Function Main() As Integer
        Dim FirstName, LastName As String
        Dim FullName As String
        Dim ComputerLanguage As String = "Visual Basic"

        FirstName = inputbox("Enter First Name: ")
        LastName = inputbox("Enter Last Name: ")

        FullName = GetFullName(FirstName, LastName)

        msgbox("Hello, " & FullName)
        Welcome(ComputerLanguage)
            Return 0
    End Function

    Sub Welcome(ByVal strLanguage As String)
        msgbox("Welcome to the wonderful world of " & strLanguage)
    End Sub
```

```
End Module
```

As mentioned previously, you can also use the **Call** keyword to call a procedure.

When you call a procedure that takes more than one argument, you must provide the values of the arguments in the exact order they are listed inside of the parentheses of the function. Fortunately, you don't have to. If you know the names of the arguments, you can type them in any order and provide a value for each. To do that, on the right side of each argument, type the := operator followed by the desired value for the argument. Here are examples:

```
Public Module Exercise

    Private Function GetFullName$(MI As String,
                                  LastName As String,
                                  FirstName As String)
        GetFullName = FirstName & " " & MI & " " & LastName
    End Function

    Public Function Main() As Integer
        Dim FullName As String
        Dim ComputerLanguage As String = "VBasic"

        FullName = GetFullName(LastName:="Roberts", FirstName:="Alan", MI:="R.")

        MsgBox("Hello " & FullName)
        Call Welcome(ComputerLanguage)
            Return 0
    End Function

    Private Sub Welcome(ByVal strLanguage As String)
        MsgBox("Welcome to the wonderful world of " & strLanguage)
    End Sub

End Module
```

Practical Learning: Passing Arguments to a Procedure

To pass arguments to a function, change the file as follows (when you type the argument, Microsoft Visual Studio, actually the Visual Basic language parser, will add the **ByVal** keywords; in the next sections, we will learn what that keyword means; for now, keep it but ignore it):

```
Module Geometry
    Private Function GetValue(TypeOfValue As String) As Double
        Dim Value As Double

        Value = InputBox("Enter the " & TypeOfValue & ":")
        Return Value
    End Function
```

```
    Private Function CalculatePerimeter(ByVal Length As Double,
                                        ByVal Width As Double) As Double
        CalculatePerimeter = (Length + Width) * 2
    End Function

    Public Function Main() As Integer
        Dim L As Double, W As Double
        Dim Perimeter As Double

        L = GetValue("Length")
        W = GetValue("Width")
        Perimeter = CalculatePerimeter(L, W)

        MsgBox("=-= Square Characteristics=-=" & vbCrLf &
               "Length: " & L & vbCrLf &
               "Width: " & W & vbCrLf &
               "Perimeter: " & Perimeter)
        Return 0
    End Function

End Module
```

To execute the program, on the main menu, click Debug -> Start Debugging

Enter the length as **44.14** and the with as **30.76**

Close the message box and return to your programming environment

Techniques of Passing Arguments

Passing an Argument By Value

When calling a procedure that takes an argument, we were supplying a value for that argument. When this is done, the procedure that is called makes a copy of the value of the argument and makes that copy available to the calling procedure. That way, the argument itself is not accessed. This is referred to as passing an argument by value. This can be reinforced by typing the **ByVal** keyword on the left side of the argument. Here is an example:

```
Private Sub Welcome(ByVal strLanguage As String)
    MsgBox("Welcome to the wonderful world of " & strLanguage)
End Sub
```

If you create a procedure that takes an argument by value and you have used the **ByVal** keyword on the argument, when calling the procedure, you don't need to use the **ByVal** keyword; just the name of the argument is enough, as done in the examples on arguments so far. Here is an example:

```
Public Module Exercise

    Public Function Main() As Integer
        Dim ComputerLanguage As String = "Visual Basic"
```

```
    Welcome(ComputerLanguage)
        Return 0
End Function

Private Sub Welcome(ByVal strLanguage As String)
    MsgBox("Welcome to the wonderful world of " & strLanguage)
End Sub

End Module
```

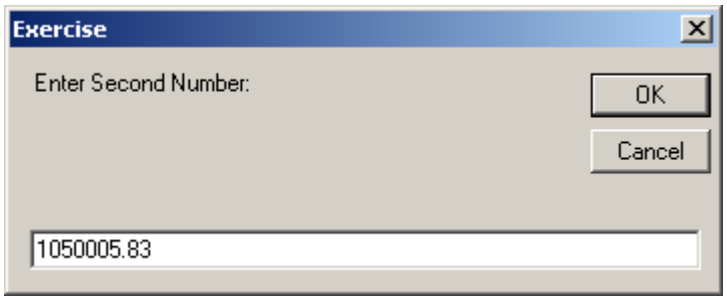
This would produce:

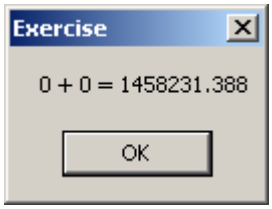
Passing an Argument By Reference

An alternative to passing an argument as done so far is to pass the address of the argument to the procedure. When this is done, the procedure doesn't receive a simple copy of the value of the argument: the argument is accessed by its address. That is, at its memory address. With this technique, any action carried on the argument will be kept. If the value of the argument is modified, the argument would now have the new value, dismissing or losing the original value it had. This technique is referred to as passing an argument by reference. Consider the following program:

```
Public Module Exercise

    Private Function Addition#(ByVal Value1 As Double, ByVal Value2 As Double)
        Value1 = InputBox("Enter First Number:  ")
        Value2 = InputBox("Enter Second Number: ")

        Addition = Value1 + Value2
    End Function

    Public Function Main() As Integer
        Dim Result As String
        Dim Number1, Number2 As Double

        Result = Addition(Number1, Number2)
        MsgBox(Number1 & " + " & Number2 & " = " & Result)
            Return 0
    End Function

End Module
```

Here is an example of running the program:



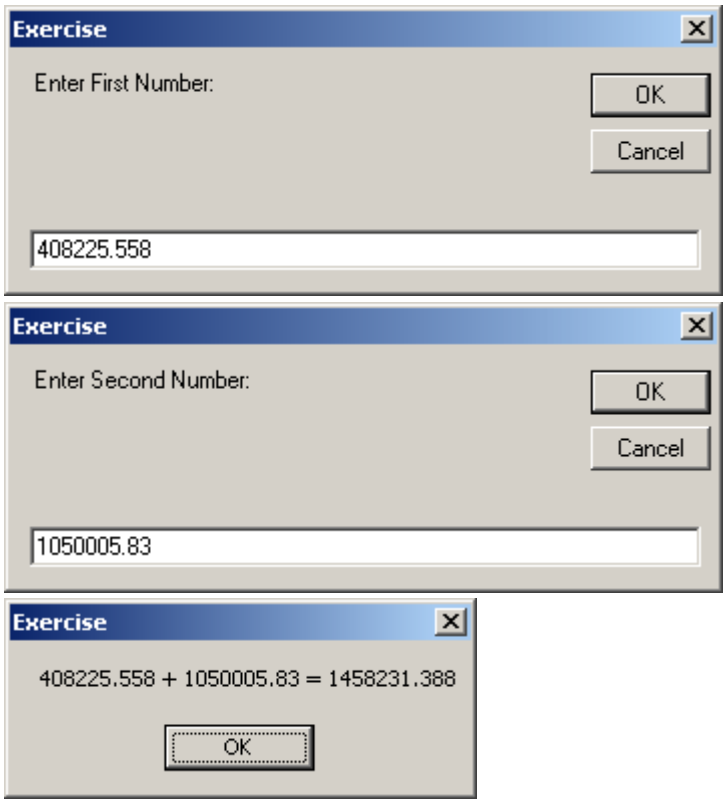


Notice that, although the values of the arguments were changed in the Addition() procedure, at the end of the procedure, they lose the value they got in the function. If you want a procedure to change the value of an argument, you can pass the argument by reference.

To pass an argument by reference, on its left, type the **ByRef** keyword. This is done only when defining the procedure. When the procedure finishes with the argument, the argument would keep whatever modification was made on its value. Now consider the same program as above but with arguments passed by reference:

```
Public Module Exercise

    Private Function Addition#(ByRef Value1 As Double, ByRef Value2 As Double)
        Value1 = InputBox("Enter First Number:  ")
        Value2 = InputBox("Enter Second Number: ")

        Addition = Value1 + Value2
    End Function

    Public Function Main() As Integer
        Dim Result As String
        Dim Number1, Number2 As Double

        Result = Addition(Number1, Number2)
        MsgBox(Number1 & " + " & Number2 & " = " & Result)
            Return 0
    End Function

End Module
```

Here is an example of running the program:



Using this technique, you can pass as many arguments by reference and as many arguments by value as you want. As you may guess already, this technique is also used to make a procedure return a value, which a regular procedure cannot do. Furthermore, passing arguments by reference allows a procedure to return as many values as possible while a regular function can return only one value.

Practical Learning: Passing Arguments by Reference

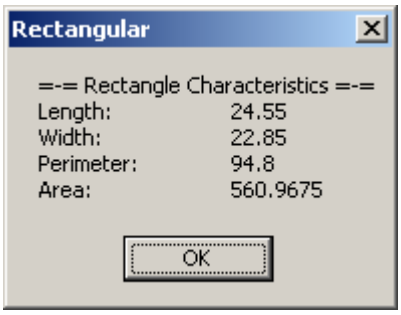
To pass an argument by reference, change the file as follows:

```
Module Geometry

Private Sub GetValues(ByRef Length As Double, ByRef Width As Double)

Length = InputBox("Enter the length:")

Width = InputBox("Enter the width:")

End Sub


Private Function CalculatePerimeter(ByVal Length As Double,
                                    ByVal Width As Double) As Double

CalculatePerimeter = (Length + Width) * 2

End Function
```

```
Private Function CalculateArea(ByVal Length As Double, _
                    ByVal Width As Double) As Double
CalculateArea = Length * Width
End Function

Private Sub ShowCharacteristics(ByVal Length As Double,
                    ByVal Width As Double)
Dim Result As String

Result = "=-= Rectangle Characteristics =-=" & vbCrLf &
                "Length: " & vbTab & vbTab & CStr(Length) & vbCrLf &
                "Width: " & vbTab & vbTab & CStr(Width) & vbCrLf &
                "Perimeter: " & vbTab &
                CalculatePerimeter(Length, Width) & vbCrLf &
                "Area: " & vbTab & vbTab &
CalculateArea(Length, Width)
MsgBox(Result)
End Sub

Public Function Main() As Integer
Dim L As Double, W As Double

GetValues(L, W)
ShowCharacteristics(L, W)

Return 0
End Function

    End Module
```

To execute the program, on the Standard toolbar, click the Start Debugging button 

Enter the length as **24.55** and the width as **22.85**



Close the message box and return to your programming environment

Other Techniques of Passing Arguments

Optional Arguments

If you create a procedure that takes one or more arguments, whenever you call that procedure, you must provide a value for the argument(s). Otherwise,, you would receive an error. If such an argument is passed with the same value over and over again, you may be tempted to remove the argument altogether. In some cases, although a certain argument is passed with the same value most of the time, you still have situations in which you want the user to decide whether to pass a value or not for the argument, you can declare the value optional. In other words, you can create the argument with a default value so that the user can call the procedure without passing a value for the argument, thus passing a value only when necessary. Such an argument is called default or optional.

Imagine you write a procedure that will be used to calculate the final price of an item after discount. The procedure would need the discount rate in order to perform the calculation. Such a procedure could look like this:

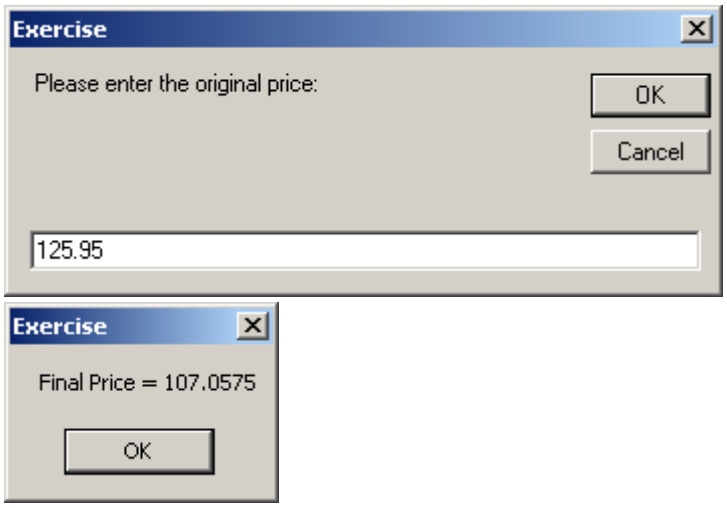
```
Function CalculateNetPrice#(ByVal DiscountRate As Double)
    Dim OrigPrice#

    OrigPrice = InputBox("Please enter the original price:")

    Return OrigPrice - (OrigPrice * DiscountRate / 100)
End Function
```

Since this procedure expects an argument, if you do not supply it, the following program would not compile:

```
Public Module Exercise

    Function CalculateNetPrice#(ByVal DiscountRate As Double)
        Dim OrigPrice#

        OrigPrice = InputBox("Please enter the original price:")
```

```
    Return OrigPrice - (OrigPrice * DiscountRate / 100)
End Function

Public Function Main() As Integer
    Dim FinalPrice#
    Dim Discount# = 15 ' That is 25% = 25

    FinalPrice = CalculateNetPrice(Discount)

    MsgBox("Final Price = " & FinalPrice)
    Return 0
End Function

End Module
```

Here is an example of running the program:



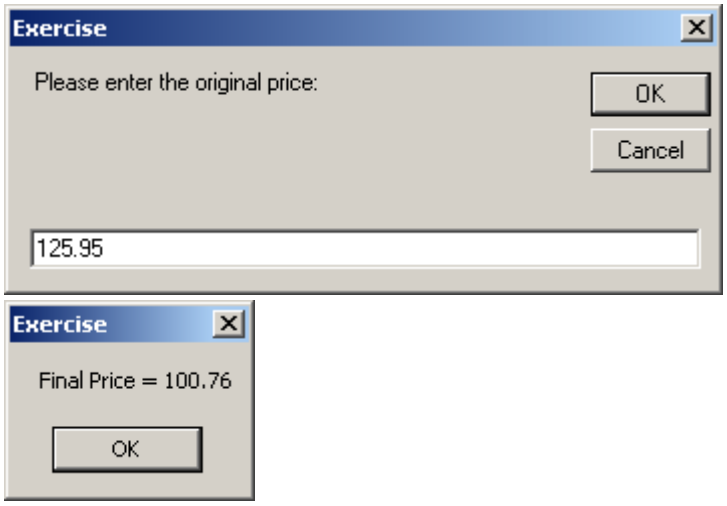
Most of the time, a procedure such as ours would use the same discount rate over and over again. Therefore, instead of supplying an argument all the time, you can define an argument whose value would be used whenever the function is not provided with the argument.

To specify that an argument is optional, when creating its procedure, type the **Optional** keyword to the left of the argument's name and assign it the default value. Here is an example:

```
Public Module Exercise

Function CalculateNetPrice(Optional ByVal DiscountRate As Double = 20)
    Dim OrigPrice#

    OrigPrice = InputBox("Please enter the original price:")

    Return OrigPrice - (OrigPrice * DiscountRate / 100)
End Function

Public Function Main() As Integer
    Dim FinalPrice#
```

```
    Dim Discount# = 15 ' That is 25% = 25

    FinalPrice = CalculateNetPrice()

    MsgBox("Final Price = " & FinalPrice)
    Return 0
  End Function

End Module
```

Here is an example of running the program:

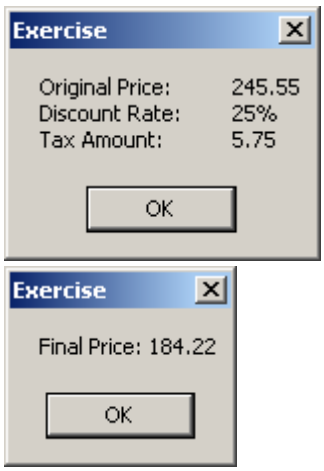


If a procedure takes more than one argument, you can provide a default argument for each and select which ones would have default values. If you want all arguments to have default values, when defining the procedure , provide the **Optional** keyword for each and assign it the desired default value. Here is an example:

```
Public Module Exercise

  Function CalculateNetPrice#(Optional ByVal Tax As Double = 5.75,
                              Optional ByVal Discount As Double = 25,
                              Optional ByVal OrigPrice As Double = 245.55)
    Dim DiscountValue As Double = OrigPrice * Discount / 100
    Dim TaxValue As Double = Tax / 100
    Dim NetPrice As Double = OrigPrice - DiscountValue + TaxValue
    Dim Result As String

    Result = "Original Price: " & vbTab & CStr(OrigPrice) & vbCrLf &
             "Discount Rate: " & vbTab & CStr(Discount) & "%" & vbCrLf &
             "Tax Amount: " & vbTab & CStr(Tax)
    MsgBox(Result)

    Return NetPrice
  End Function

  Public Function Main() As Integer
```

```
    Dim FinalPrice As Double

    FinalPrice = CalculateNetPrice()
    MsgBox("Final Price: " & CStr(FinalPrice))
    Return 0
End Function

End Module
```

This would produce:



If a procedure takes more than one argument as above, remember that some arguments can be specified as optional. In this case, when calling the procedure, any argument that does not have a default value must be passed with a value. When creating a procedure that takes more than one argument, the argument(s) that has(have) default value(s) must be the last in the procedure. This means that:

- If a procedure takes two arguments and one argument has a default value, this optional argument must be the second
- If a procedure is taking three or more arguments and two or more arguments have default values, these optional arguments must by placed to the right of the non-optional argument(s).

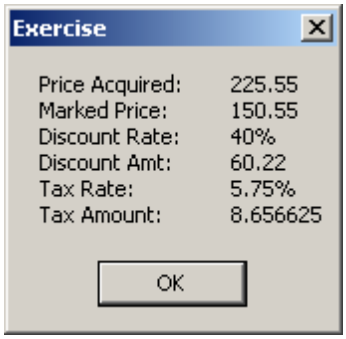
Because of this, when calling any procedure in the Visual Basic language, you must know what, if any, argument is optional and which one is not.

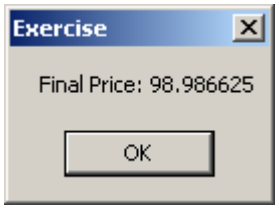
If a procedure takes two arguments and one argument has a default value, when calling this procedure, you can pass only one value. In this case, the passed value would be applied on the first argument. If a procedure takes more than two arguments and two or more arguments have a default value, when calling this procedure, you can provide only the value(s) of the argument that is (are) not optional. If you want to provide the value of one of the arguments but that argument is not the first optional, you can leave empty the position(s) of the other argument(s) but remember to type a comma to indicate that the position is that of an argument that has a default value. Here is an example:

```
Public Module Exercise

    Function CalculateNetPrice(ByVal AcquiredPrice As Double,
```

```
                    ByVal MarkedPrice As Double,
                    Optional ByVal TaxRate As Double = 5.75,
                    Optional ByVal DiscountRate As Double = 25) As Double
    Dim DiscountAmount As Double = MarkedPrice * DiscountRate / 100
    Dim TaxAmount As Double = MarkedPrice * TaxRate / 100
    Dim NetPrice As Double = MarkedPrice - DiscountAmount + TaxAmount
    Dim Result As String

    Result = "Price Acquired: " & vbTab & CStr(AcquiredPrice) & vbCrLf &
        "Marked Price: " & vbTab & CStr(MarkedPrice) & vbCrLf &
        "Discount Rate: " & vbTab & CStr(DiscountRate) & "%" & vbCrLf &
        "Discount Amt: " & vbTab & CStr(DiscountAmount) & vbCrLf &
        "Tax Rate: " & vbTab & CStr(TaxRate) & "%" & vbCrLf &
        "Tax Amount: " & vbTab & CStr(TaxAmount)
    MsgBox(Result)

    Return NetPrice
  End Function

  Public Function Main() As Integer
    Dim FinalPrice As Double

    FinalPrice = CalculateNetPrice(225.55, 150.55, , 40)
    MsgBox("Final Price: " & CStr(FinalPrice))
    Return 0
  End Function

End Module
```

This would produce:





Procedure Overloading

A program involves a great deal of names that represent variables and procedures of various kinds. The compiler does not allow two variables to have the same name in the same procedure

(or in the same scope). Although two procedures should have unique names in the same program, you are allowed to use the same name for different procedures of the same program following certain rules.

The ability to have various procedures with the same name in the same program is referred to as overloading. The most important rule about procedure overloading is to make sure that each one of these procedures has a different number or different type(s) of arguments.

METHOD or PROCEDURE OVERLOADING :

Method Overloading allows us to write different versions of a method (i.e. a single class can contain more than one methods (Sub / Functions) of same name but of different implementation). And compiler will automatically select the appropriate method based on parameters passed.

Consider following points in order to implement/understand method overloading :

All the overloaded methods must be of same procedure type (either only Sub or only Function).
Allowed :
Public Overloads Function add(ByVal a As Integer, ByVal b As Integer)
Public Overloads Function add(ByVal a As Long, ByVal b As Long)
Not Allowed :
Public Overloads Function add(ByVal a As Integer, ByVal b As Integer)
Public Overloads Sub add(ByVal a As Long, ByVal b As Long)

Don't repeat same parameters with same data types in parameter, **it will show an error**
Example :
Public Overloads Function add(ByVal a As Integer, ByVal b As Integer)
Public Overloads Function add(ByVal a As Integer, ByVal b As Integer)

Overloaded methods must have different number parameters and/or Different data types in the parameters
Different number of parameters
Public Overloads Function add(ByVal a As Integer, ByVal b As Integer)
Public Overloads Function add(ByVal a As Integer, ByVal b As Integer, ByVal c As Integer)
Different data type in parameters
Public Overloads Function add(ByVal a As Integer, ByVal b As Integer)
Public Overloads Function add(ByVal a As Long, ByVal b As Long)

Inherited Classes can also overload their own methods or their base class methods

Example 1 :

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

```
    Public Class demo
        Public Overloads Function add(ByVal a As Integer, ByVal b As Integer)
            WriteLine("You are in function add(integer, integer)")
            Return a + b
        End Function
        Public Overloads Function add(ByVal a As Long, ByVal b As Long)
            WriteLine("You are in function add(long, long)")
            Return a + b
        End Function
    End Class
    Sub Main()
        Dim obj As New demo
        WriteLine(obj.add(2147483640, 4))
        WriteLine(obj.add(2147483648, 1))
        WriteLine("press return to exit...")
        Read()
    End Sub
End Module
```

Example 2 :

```
Imports System.Console
Module Module1
    Public Class demo
        Public Overloads Function add(ByVal a As Integer, ByVal b As Integer)
            WriteLine("You are in function add(a,b)")
            Return a + b
        End Function
        Public Overloads Function add(ByVal a As Integer, ByVal b As Integer, ByVal c As Integer)
            WriteLine("You are in function add(a, b, c)")
            Return a + b + c
        End Function
    End Class
    Sub Main()
        Dim obj As New demo
        WriteLine(obj.add(4, 2))
        WriteLine(obj.add(4, 5, 1))
        WriteLine("press return to exit...")
        Read()
    End Sub
End Module
```