

## UNIT II

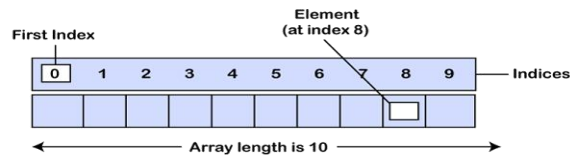
- Conditional statements,
- Loop statements.
- Arrays,
- Types of array
- Exception handling
- Sub procedure
- Functions
- Passing arguments
- Optional argument
- Returning value from function
- MsgBox & inputbox
- String manipulation
- Mathematical function
- Date function

## VB.NET Arrays

- An **array** is a linear data structure that is a collection of data elements of the same type stored on a **contiguous memory** location.
- Each data item is called an element of the array.
- It is a fixed size of sequentially arranged elements in computer memory with the first element being at **index 0** and the last element at index **n - 1**, where **n** represents the total number of elements in the array.

## VB.NET Arrays

- Representation of similar data type



- In the above diagram, we store the Integer type data elements in an array starting at index 0.
- It will continue to store data elements up to a defined number of elements.

## Declaration of VB.NET Array

- `Dim array_name As [Data_Type] ()`
- **array\_name** is the name of an array, **Data\_Type** represents the type of element (Integer, char, String, Decimal) that will store contiguous data elements in the VB.NET array.

## Example to declare an array.

- 'Store only Integer values
- `Dim num As Integer()` or `Dim num(5) As Integer`
- 'Store only String values
- `Dim name As String()` or `Dim name(5) As String`
- 'Store only Double values
- `Dim marks As Double()`

## Initialization of VB.NET Array

- In VB.NET, we can initialize an array with **New** keyword at the time of declaration.

### For example,

- 'Declaration and Initialization of an array elements with size 6
- `Dim num As Integer() = New Integer(5) { }`
- `Dim num As Integer() = New Integer(5) {1, 2, 3, 4, 5, 6}`
- Initialize an array with 5 elements that indicates the size of an array
- `Dim arr_name As Integer() = New Integer() {5, 10, 5, 20, 15}`
- Declare an array
- `Dim array1 As Char()`

`array1 = New Char() { 'A', 'B', 'C', 'D', 'E' }`

## Program to add the elements of an array in VB.NET programming language. (num\_Array.vb)

- Imports System
- Module num\_Array
- Sub Main()
- Dim i As Integer, Sum As Integer = 0
- Dim marks() As Integer = {58, 68, 95, 50, 23, 89}
- Console.WriteLine("Marks in 6 Subjects")
- For i = 0 To marks.Length - 1
- Console.WriteLine(" Marks {0}", marks(i))
- Sum = Sum + marks(i)
- Next
- Console.WriteLine(" Grand total is {0}", Sum)
- Console.WriteLine(" Press any key to exit...")
- Console.ReadKey()
- End Sub
- End Module

```
Marks in 6 Subjects
Marks 58
Marks 68
Marks 95
Marks 50
Marks 23
Marks 89
Grand total is 383
Press any key to exit...
```

### Program to take input values from the user and display them in VB.NET programming language. (Input\_array.vb)

- Imports System
- Module Input\_array
- Sub Main()
- Dim arr As Integer() = New Integer(5) {}
- For i As Integer = 0 To 5
- Console.WriteLine(" Enter the value for arr[{0}] : ", i)
- arr(i) = Console.ReadLine()
- Next
- Console.WriteLine(" The array elements are : ")
- For j As Integer = 0 To 5
- Console.WriteLine("{0}", arr(j))
- Next
- Console.WriteLine(" Press any key to exit...")
- Console.ReadKey()
- End Sub
- End Module

```
Enter the value for arr[1] :  
15  
Enter the value for arr[2] :  
20  
Enter the value for arr[3] :  
25  
Enter the value for arr[4] :  
5  
The array elements are :  
10  
15  
20  
25  
5  
Press any key to exit...
```

## Multidimensional Array

- In VB.NET, a multidimensional array is useful for storing more than one dimension in a tabular form, such as rows and columns.
- The multidimensional array support two or three dimensional in VB NET

### Declaration of two-dimensional array

- Dim twoDimenArray As Integer( , ) = New Integer(3, 2) {}
- Or Dim arr(5, 3) As Integer

## Representation of Three Dimensional array

- Dim arrThree(2, 4, 3) As Integer
- Or Dim arr1 As Integer( , , ) = New Integer(5, 5, 5) { }
- In the above representation of multidimensional, we have created 2-dimensional array **twoDimenArray with 3 rows and 2 columns** and 3-dimensional array with three dimensions 2, 4 and 3

## Initialization of Multidimensional Array

- Dim intArray As Integer( , ) = New Integer( 3, 2) { {4, 5}, {2, 3}, {6, 7} }
- Dim intArray( , ) As Integer = { {5, 4}, {3, 2}, {4, 7} }

## Initialization of Three Dimensional Array

- Dim threeDimen(3, 3, 2 ) As Integer = { {{1, 3, 2}, {2, 3, 4}}, {{5, 3, 6}, {3, 4, 5}}, {{1, 2, 2}, {5, 2, 3}} }

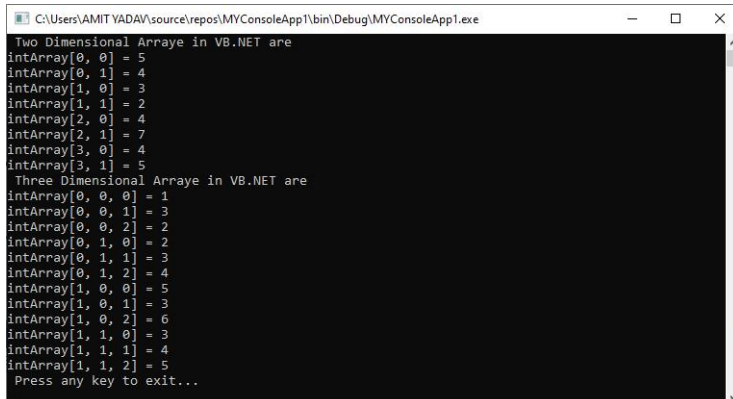
## Multidimensional Array Example (MultidimenArray.vb)

- Sub Module
- Sub Main()
- Dim intArray(,) As Integer = {{5, 4}, {3, 2}, {4, 7}, {4, 5}}
- Dim threeDimen(,,) As Integer =
- {{{1, 3, 2}, {2, 3, 4}},
- {{5, 3, 6}, {3, 4, 5}},
- {{1, 2, 2}, {5, 2, 3}}}
- Console.WriteLine(" Two Dimensional Arraye in VB.NET are")
- For i As Integer = 0 To 3
- For j As Integer = 0 To 1
- Console.WriteLine("intArray[{0}, {1}] = {2}", i, j, intArray(i, j))
- Next j

### Multidimensional Array Example (MultidimenArray.vb)

- Console.WriteLine(" Three Dimensional Array in VB.NET are")
- For i As Integer = 0 To 2 - 1
- For j As Integer = 0 To 2 - 1
- For k As Integer = 0 To 4
- Console.WriteLine("intArray[{0}, {1}, {2}] = {3}", i, j, k, three Dimen(i, j, k))
- Next k
- Next j
- Next i
- Console.WriteLine(" Press any key to exit...")
- Console.ReadKey()
- End Sub
- End Module

### Multidimensional Array Example (MultidimenArray.vb)



```
C:\Users\AMIT YADAV\source\repos\MYConsoleApp1\bin\Debug\MYConsoleApp1.exe
Two Dimensional Array in VB.NET are
intArray[0, 0] = 5
intArray[0, 1] = 4
intArray[1, 0] = 3
intArray[1, 1] = 2
intArray[2, 0] = 4
intArray[2, 1] = 7
intArray[3, 0] = 4
intArray[3, 1] = 5
Three Dimensional Array in VB.NET are
intArray[0, 0, 0] = 1
intArray[0, 0, 1] = 3
intArray[0, 0, 2] = 2
intArray[0, 1, 0] = 2
intArray[0, 1, 1] = 3
intArray[0, 1, 2] = 4
intArray[1, 0, 0] = 5
intArray[1, 0, 1] = 3
intArray[1, 0, 2] = 6
intArray[1, 1, 0] = 3
intArray[1, 1, 1] = 4
intArray[1, 1, 2] = 5
Press any key to exit...
```

## Fixed Size Array

- In VB.NET, a fixed- size array is used to hold a fixed number of elements in memory.
- It means that we have defined the number of elements in the array declaration that will remain the same during the definition of the elements, and its size cannot be changed.
- For example, we need to hold only 5 names in an array; it can be defined and initialized in the array such as,
- Dim names( 0 to 4) As String
- names(0) = "Robert"
- names(1) = "Henry"
- names(2) = "Rock"
- names(3) = "James"
- names(4) = "John"
- The above representation of the fixed array is that we have defined a string array **names 0 to 4**, which stores all the elements in the array from 0 to index 4.

## VB.NET Exception Handling

- An **exception** is an unwanted error that occurs during the execution of a program and can be a system exception or application exception.
- Exceptions are nothing but some abnormal and typically an event or condition that arises during the execution, which may interrupt the normal flow of the program.
- An exception can occur due to different reasons, including the following:
- 1) A user has entered incorrect data or performs a division operator, such as an attempt to divide by zero.
- 2) A connection has been lost in the middle of communication, or system memory has run out.

## VB.NET Exception Handling

- When an error occurred during the execution of a program, the exception provides a way to transfer control from one part of the program to another using **exception handling** to handle the error.
- VB.NET exception has four built-in keywords such as **Try**, **Catch**, **Finally**, and **Throw** to handle and move controls from one part of the program to another.
- **Try** A try block is used to monitor a particular exception that may throw an exception within the application. And to handle these exceptions, it always follows one or more catch blocks.

## VB.NET Exception Handling

- **Catch** It is a block of code that catches an exception with an exception handler at the place in a program where the problem arises.
- **Finally** It is used to execute a set of statements in a program, whether an exception has occurred.
- **Throw** As the name suggests, a throw handler is used to throw an exception after the occurrence of a problem.



## Exception Classes in VB.NET

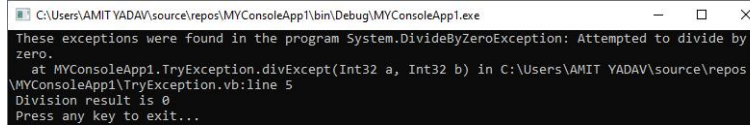
- In VB.net, there are various types of exceptions represented by classes. And these exception classes originate from their parent's class 'System.Exception'.
- **The following are the two exception classes used primarily in VB.NET.**
- **System.SystemException:** It is a base class that includes all predefined exception classes, and some system-generated exception classes that have been generated during a run time such as **DivideByZeroException**, **IndexOutOfRangeException**, **StackOverflowException**, and so on.
- **System.ApplicationException:** It is an exception class that throws an exception defined within the application by the programmer or developer. Furthermore, we can say that it is a user-defined exception that inherits from **System.ApplicationException class**.

## Syntax of exception handler block

- Try
- ' code or statement to be executed
- [ Exit Try block]
- ' **catch** statement followed by Try block
- Catch [ Exception name] As [ Exception Type]
- [Catch1 Statements] Catch [Exception name] As [Exception Type]
- [ Exit Try ]
- [ Finally
- [ Finally Statements ] ]
- End Try
- In the above syntax, the Try/Catch block is always surrounded by a code that can throw an exception. And the code is known as a protected code. Furthermore, we can also use multiple catch statements to catch various types of exceptions in a program, as shown in the syntax.

## Example to Exception Handle (TryException.vb)

```
Module TryException
    Sub divExcept(ByVal a As Integer, ByVal b As Integer)
        Dim res As Integer
        Try
            res = a \ b
        Catch ex As DivideByZeroException
            Console.WriteLine(" These exceptions were found in the program {0}", ex)
        Finally
            Console.WriteLine(" Division result is {0}", res)
        End Try
    End Sub
    Sub Main()
        divExcept(5, 0)
        Console.WriteLine(" Press any key to exit...")
        Console.ReadKey()
    End Sub
End Module
```



```
C:\Users\AMIT YADAV\source\repos\MYConsoleApp1\bin\Debug\MYConsoleApp1.exe
These exceptions were found in the program System.DivideByZeroException: Attempted to divide by
zero.
at MYConsoleApp1.TryException.divExcept(Int32 a, Int32 b) in C:\Users\AMIT YADAV\source\repos
\MYConsoleApp1\TryException.vb:line 5
Division result is 0
Press any key to exit...
```

## Creating User-Defined Exceptions

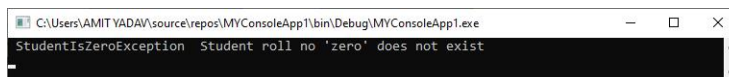
- It allows us to create custom exceptions derived from the **ApplicationException** class.

### Program to understand the uses of User-Defined Exceptions (User\_Exception.vb)

```
Module User_Exception
    Public Class StudentIsZeroException : Inherits Exception
        Public Sub New(ByVal stdetails As String)
            MyBase.New(stdetails)
        End Sub
    End Class
    Public Class StudentManagement
        Dim stud As Integer = 0
        Sub ShowDetail()
            If (stud = 0) Then
                Throw (New StudentIsZeroException(" Student roll no 'zero' does not exist"))
            Else
                Console.WriteLine(" Student is {0}", stud)
            End If
        End Sub
    End Class
```

## Creating User-Defined Exceptions

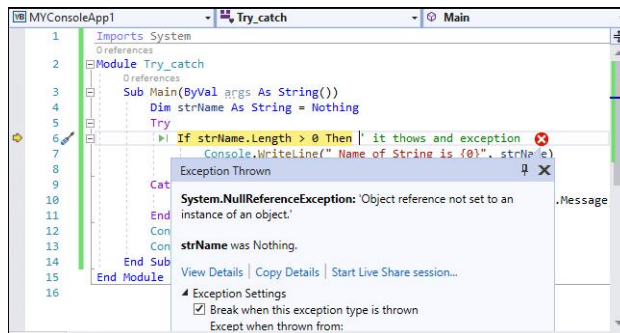
```
Sub Main()  
    Dim stdmg As StudentManagement = New StudentManagement()  
    Try  
        stdmg.ShowDetail()  
    Catch ex As StudentIsZeroException  
        Console.WriteLine(" StudentIsZeroException {0}", ex.message)  
    End Try  
    Console.ReadKey()  
End Sub  
End Module
```



## Using Try-Catch Statement

### Try\_catch.vb

```
Imports System  
Module Try_catch  
    Sub Main(ByVal args As String())  
        Dim strName As String = Nothing  
        Try  
            If strName.Length > 0 Then ' it throws and exception  
                Console.WriteLine(" Name of String is {0}", strName)  
            End If  
        Catch ex As Exception ' it catches an exception  
            Console.WriteLine(" Catch exception in a program {0}", ex.Message)  
        End Try  
        Console.WriteLine(" Press any key to exit...")  
        Console.ReadKey()  
    End Sub  
End Module
```



## Throwing Objects

- In VB.NET exception handling, we can throw an object exception directly or indirectly derived from the **System.Exception** class.
- To throw an object using the throw statement in a catch block, such as:  
Throw [ expression ]

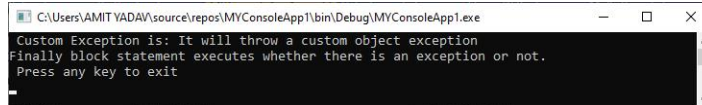
## Program to throw an object in VB.NET exception (throwexcent.vb)

```
Imports System
Module throwexcent
    Sub Main()
        Try
            Throw New ApplicationException("It will throw a custom object exception")

        Catch ex As Exception
            Console.WriteLine(" Custom Exception is: {0}", ex.Message)

        Finally
            Console.WriteLine("Finally block statement executes whether there is an exception or not.")

        End Try
        Console.WriteLine(" Press any key to exit")
        Console.ReadKey()
    End Sub
End Module
```



## VB.NET Sub procedure

- A Sub procedure is a separate set of codes that are used in VB.NET programming to execute a specific task, and it does not return any values.
- The Sub procedure is enclosed by the Sub and End Sub statement.
- The Sub procedure is similar to the function procedure for executing a specific task except that it does not return any value, while the function procedure returns a value.

### Defining the Sub procedure

```
[Access_Specifier] Sub Sub_name [ (parameterList) ]  
[ Block of Statement to be executed ]  
End Sub
```

Where,

**Access\_Specifier:** It defines the access level of the procedure such as public, private or friend, Protected, etc. and information about the overloading, overriding, shadowing to access the method.

**Sub\_name:** The Sub\_name indicates the name of the Sub that should be unique.

**ParameterList:** It defines the list of the parameters to send or retrieve data from a method.

## VB.NET Sub procedure

The following are the different ways to define the types of Sub method.

```
Public Sub getDetails()  
' Statement to be executed  
End Sub
```

```
Private Sub GetData( ByVal username As String) As String  
' Statement to be executed  
End Sub
```

```
Public Function GetData1( ByRef username As String, ByRef userId As Integer)  
' Statement to be executed  
End Sub
```

**Example:** Write a simple program to pass the empty, a single or double parameter of Sub procedure in the VB.NET.

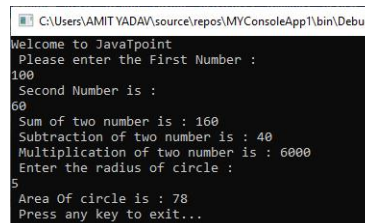
### Sub\_Program.vb

Module Sub\_Program

```
Sub sample()  
    Console.WriteLine("Welcome to JavaTpoint")  
End Sub
```

```
Sub circle(ByVal r As Integer)  
    Dim Area As Integer  
    Const PI = 3.14  
    Area = PI * r * r  
    Console.WriteLine(" Area Of circle is : {0}", Area)  
End Sub
```

```
' Create the SumOfTwo() Function and pass the parameters.  
Sub SumOfTwo(ByVal n1 As Integer, ByVal n2 As Integer)  
    ' Define the local variable.  
    Dim sum As Integer = 0  
    sum = n1 + n2  
    Console.WriteLine(" Sum of two number is : {0}", sum)  
End Sub
```



C:\Users\AMIT\YADAV\source\repos\MYConsoleApp1\bin\Debug  
Welcome to JavaTpoint  
Please enter the First Number :  
100  
Second Number is :  
60  
Sum of two number is : 160  
Subtraction of two number is : 40  
Multiplication of two number is : 6000  
Enter the radius of circle :  
5  
Area Of circle is : 78  
Press any key to exit...

**Example:** Write a simple program to pass the empty, a single or double parameter of Sub procedure in the VB.NET.

```
Sub SubtractionOfTwo(ByVal n1 As Integer, ByVal n2 As Integer)
    ' Define the local variable.
    Dim subtract As Integer
    subtract = n1 - n2
    Console.WriteLine(" Subtraction of two number is : {0}", subtract)
End Sub

Sub MultiplicationOfTwo(ByVal n1 As Integer, ByVal n2 As Integer)
    ' Define the local variable.
    Dim multiply As Integer
    multiply = n1 * n2
    Console.WriteLine(" Multiplication of two number is : {0}", multiply)
End Sub
```

**Example:** Write a simple program to pass the empty, a single or double parameter of Sub procedure in the VB.NET.

```
Sub Main()
    ' Define the local variable a, b and rad.
    Dim a, b, rad As Integer
    sample() ' call sample() procedure
    Console.WriteLine(" Please enter the First Number : ")
    a = Console.ReadLine()
    Console.WriteLine(" Second Number is : ")
    b = Console.ReadLine()

    SumOfTwo(a, b) 'call SumOfTwo() Function
    SubtractionOfTwo(a, b) 'call SubtractionOfTwo() Function
    MultiplicationOfTwo(a, b) 'call MultiplicationOfTwo() Function

    Console.WriteLine(" Enter the radius of circle : ")
    rad = Console.ReadLine()
    circle(rad)
    Console.WriteLine(" Press any key to exit...")
    Console.ReadKey()
End Sub
End Module
```

## Pass by Value

In the VB.NET programming language, we can pass parameters in two different ways:

### Passing parameter by Value

- In the VB.NET, passing parameter by value is the default mechanism to pass a value in the Sub method.
- When the method is called, it simply copies the actual value of an argument into the formal method of Sub procedure for creating a new storage location for each parameter.
- Therefore, the changes made to the main function's actual parameter that do not affect the Sub procedure's formal argument.

### Syntax:

```
Sub Sub_method( ByVal parameter_name As datatype )  
[ Statement to be executed ]  
End Sub
```

In the above syntax, the **ByVal** is used to declare parameters in a Sub procedure.

## Program to understand the concept of passing parameter by value

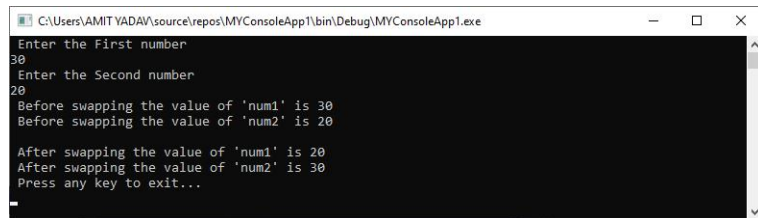
### Passing\_value.vb

```
Imports System  
Module Passing_value  
    Sub Main()  
        ' declaration of local variable  
        Dim num1, num2 As Integer  
        Console.WriteLine(" Enter the First number")  
        num1 = Console.ReadLine()  
        Console.WriteLine(" Enter the Second number")  
        num2 = Console.ReadLine()  
        Console.WriteLine(" Before swapping the value of 'num1' is {0}", num1)  
        Console.WriteLine(" Before swapping the value of 'num2' is {0}", num2)  
        Console.WriteLine()  
  
        'Call a function to pass the parameter for swapping the numbers.  
        swap_value(num1, num2)  
        Console.WriteLine(" After swapping the value of 'num1' is {0}", num1)  
        Console.WriteLine(" After swapping the value of 'num2' is {0}", num2)  
        Console.WriteLine(" Press any key to exit...")  
        Console.ReadKey()  
    End Sub
```



## Program to understand the concept of passing parameter by value

```
' Create a swap_value() method
Sub swap_value(ByVal a As Integer, ByVal b As Integer)
    ' Declare a temp variable
    Dim temp As Integer
    temp = a ' save the value of a to temp
    b = a ' put the value of b into a
    b = temp ' put the value of temp into b
End Sub
End Module
```



The screenshot shows a console window titled "C:\Users\AMIT YADAV\source\repos\MYConsoleApp1\bin\Debug\MYConsoleApp1.exe". The output is as follows:

```
Enter the First number:
30
Enter the Second number:
20
Before swapping the value of 'num1' is 30
Before swapping the value of 'num2' is 20

After swapping the value of 'num1' is 20
After swapping the value of 'num2' is 30
Press any key to exit...
```

## Passing parameter by Reference

- A Reference parameter is a reference of a variable in the memory location.
- The reference parameter is used to pass a reference of a variable with **ByRef** in the Sub procedure.
- When we pass a reference parameter, it does not create a new storage location for the sub method's formal parameter.
- Furthermore, the reference parameters represent the same memory location as the actual parameters supplied to the method.
- So, when we changed the value of the formal parameter, the actual parameter value is automatically changed in the memory.
- The syntax for the passing parameter by Reference:

```
Sub Sub_method( ByRef parameter_name, ByRef Parameter_name2 )
    [ Statement to be executed ]
End Sub
```
- In the above syntax, the **ByRef** keyword is used to pass the Sub procedure's reference parameters.

## Passing parameter by Reference

Let's create a program to swap the values of two variables using the ByRef keyword.

```
Passing_ByRef.vb
Imports System
Module Passing_ByRef
    Sub Main()
        ' declaration of local variable
        Dim num1, num2 As Integer
        Console.WriteLine(" Enter the First number")
        num1 = Console.ReadLine()
        Console.WriteLine(" Enter the Second number")
        num2 = Console.ReadLine()
        Console.WriteLine(" Before swapping the value of 'num1' is {0}", num1)
        Console.WriteLine(" Before swapping the value of 'num2' is {0}", num2)
        Console.WriteLine()

        'Call a function to pass the parameter for swapping the numbers.
        swap_Ref(num1, num2)
        Console.WriteLine(" After swapping the value of 'num1' is {0}", num1)
        Console.WriteLine(" After swapping the value of 'num2' is {0}", num2)
        Console.WriteLine(" Press any key to exit...")
        Console.ReadKey()
```

## Program to swap the values of two variables using the ByRef keyword

```
Module Passing_ByRef
    Sub Main()
        Dim num1, num2 As Integer
        Console.WriteLine(" Enter the First number")
        num1 = Console.ReadLine()
        Console.WriteLine(" Enter the Second number")
        num2 = Console.ReadLine()
        Console.WriteLine(" Before swapping the value of 'num1' is {0}", num1)
        Console.WriteLine(" Before swapping the value of 'num2' is {0}", num2)
        Console.WriteLine()
        swap_Ref(num1, num2)
        Console.WriteLine(" After swapping the value of 'num1' is {0}", num1)
        Console.WriteLine(" After swapping the value of 'num2' is {0}", num2)
        Console.WriteLine(" Press any key to exit...")
        Console.ReadKey()
    End Sub
    Sub swap_Ref(ByRef a As Integer, ByRef b As Integer)
        Dim temp As Integer
        temp = a
        a = b
        b = temp
    End Sub End Module
```