Training Manual

FOR

Microsoft Visual Basic Programming

Compiled by **Technology Ed**

April 2012

CHAPTER-6: VARIABLES

This chapter describes about the variables in Visual Basic. Variables are placeholders used to store values; they have names and data types. The data type of a variable determines how the bits representing those values are stored in the computer's memory. When you declare a variable, you can also supply a data type for it. All variables have a data type that determines what kind of data they can store.

You can think of variables as containers, and you choose to put data in the most suitable container. You would not put a small object in a large box or try to stuff a large object into a small box.

6.1 Data Types

By default Visual Basic variables are of variant data types. The variant data type can store numeric, date/time or string data. When a variable is declared, a data type is supplied for it that determines the kind of data they can store. The fundamental data types in Visual Basic including variant are integer, long, single, double, string, currency, byte and Boolean. Visual Basic supports a vast array of data types. Each data type has limits to the kind of information and the minimum and maximum values it can hold. In addition, some types can interchange with some other types. A list of Visual Basic's simple data types are given below.

6.1.1 Numeric

Byte	Store integer values in the range of 0 - 255
Integer	Store integer values in the range of (-32,768) - (+ 32,767)
Long	Store integer values in the range of (- 2,147,483,468) - (+ 2,147,483,468)
Single	Store floating point value in the range of (-3.4x10-38) - (+ 3.4x1038)
Double	Store large floating value which exceeding the single data type value
Currency	Store monetary values. It supports 4 digits to the right of decimal point and 15 digits to the left

6.1.2 String

Use to store alphanumeric values. A variable length string can store approximately 4 billion characters.

6.1.3 Date

Use to store date and time values. A variable declared as date type can store both date and time values and it can store date values 01/01/0100 up to 12/31/9999.

6.1.4 Boolean

Boolean data types hold either a true or false value. These are not stored as numeric values and cannot be used as such. Values are internally stored as -1 (True) and 0 (False) and any non-zero value is considered as true.

6.1.5 Variant

Stores any type of data and is the default Visual Basic data type. In Visual Basic if we declare a variable without any data type by default the data type is assigned as default.

6.2 Scope and Lifetime of Variables

A variable is scoped to a procedure-level (local) or module-level variable depending on how it is declared. The scope of a variable, procedure or object determines which part of the code in our application is aware of the variable's existence. A variable is declared in general declaration section of e Form, and hence is available to all the procedures. Local variables are recognized only in the procedure in which they are declared. They can be declared with Dim and Static keywords. If we want a variable to be available to all of the procedures within the same module, or to all the procedures in an application, a variable is declared with broader scope.

6.2.1 Local Variables

A local variable is one that is declared inside a procedure. This variable is only available to the code inside the procedure and can be declared using the Dim statements as given below.

```
Dim sum As Integer
```

The local variables exist as long as the procedure in which they are declared, is executing. Once a procedure is executed, the values of its local variables are lost and the memory used by these variables is freed and can be reclaimed. Variables that are declared with keyword Dim exist only as long as the procedure is being executed.

6.2.2 Static Variables

Static variables are not reinitialized each time Visual Invokes a procedure and therefore retain or preserve value even when a procedure ends. In case we need to keep track of the number of times a command button in an application is clicked, a static counter variable has to be declared. These static variables are also ideal for making controls alternately visible or invisible. A static variable is declared as given below.

```
Static intPermanent As Integer
```

Variables have a lifetime in addition to scope. The values in a module-level and public variables are preserved for the lifetime of an application whereas local variables declared with Dim exist only while the procedure in which they are declared is still being executed. The value of a local variable can be preserved using the Static keyword. The following procedure calculates the running total by adding new values to the previous values stored in the static variable value.

```
Function RunningTotal ( )

Static Accumulate

Accumulate = Accumulate + num

RunningTotal = Accumulate

End Function
```

If the variable Accumulate was declared with Dim instead of static, the previously accumulated values would not be preserved across calls to the procedure, and the procedure would return the same value with which it was called. To make all variables in a procedure static, the Static keyword is placed at the beginning of the procedure heading as given in the below statement.

```
Static Function RunningTotal ( )
```

Example

The following is an example of an event procedure for a CommandButton that counts and displays the number of clicks made.

```
Private Sub Command1_Click ( )

Static Counter As Integer

Counter = Counter + 1

Print Counter

End Sub
```

The first time we click the CommandButton, the Counter starts with its default value of zero. Visual Basic then adds 1 to it and prints the result.

6.2.3 Module Level Variables

A module level variable is available to all the procedures in the module. They are declared using the Public or the Private keyword. If you declare a variable using a Private or a Dim statement in the declaration section of a module—a standard BAS module, a form module, a class module, and so on—you're creating a private module-level variable. Such variables are visible only from within the module they belong to and can't be accessed from the outside. In general, these variables are useful for sharing data among procedures in the same module:

```
' In the declarative section of any module

Private LoginTime As Date ' A private module-level variable

Dim LoginPassword As String ' Another private module-level variable
```

You can also use the Public attribute for module-level variables, for all module types except BAS modules. (Public variables in BAS modules are global variables.) In this case, you're creating a strange beast: a Public module-level variable that can be accessed by all procedures in the module to share data and that also can be accessed from outside the module. In this case, however, it's more appropriate to describe such a variable as a property:

' In the declarative section of Form1 module

```
Public CustomerName As String ' A Public property
```

You can access a module property as a regular variable from inside the module and as a custom property from the outside:

' From outside Form1 module...

```
Form1.CustomerName = "John Smith"
```

The lifetime of a module-level variable coincides with the lifetime of the module itself. Private variables in standard BAS modules live for the entire life of the application, even if they can be accessed only while Visual Basic is executing code in that module. Variables in form and class modules exist only when that module is loaded in memory. In other words, while a form is active (but not necessarily visible to the user) all its variables take some memory, and this memory is released only when the form is completely unloaded from memory. The next time the form is re-created, Visual Basic reallocates memory for all variables and resets them to their default values (0 for numeric values, "" for strings, Nothing for object variables).

6.2.4 Public Vs Local Variables

A variable can have the same name and different scope. For example, we can have a public variable named R and within a procedure we can declare a local variable R. References to the name R within the procedure would access the local variable and references to R outside the procedure would access the public variable.

6.3 Constants

Constants are named storage locations in memory, the value of which does not change during program Execution. They remain the same throughout the program execution. When the user wants to use a value that never changes, a constant can be declared and created. The Const statement is used to create a constant. Constants can be declared in local, form, module or global scope and can be public or private as for variables. Constants can be declared as illustrated below.

```
Public Const gravityconstant As Single = 9.81.
```

6.3.1 Predefined Visual Basic Constants

The predefined constants can be used anywhere in the code in place of the actual numeric values. This makes the code easier to read and write.

For example consider a statement that will set the window state of a form to be maximized.

```
Form1.Windowstate = 2
```

The same task can be performed using a Visual Basic constant

```
Form1.WindowState = vbMaximized
```

6.4 Arrays and User Defined Types

6.4.1 Arrays in Visual Basic

An array is a consecutive group of memory locations that all have the same name and the same type. To refer to a particular location or element in the array, we specify the array name and the array element position number.

The Individual elements of an array are identified using an index. Arrays have upper and lower bounds and the elements have to lie within those bounds. Each index number in an array is allocated individual memory space and therefore users must evade declaring arrays of larger size than required. We can declare an array of any of the basic data types including variant, user-defined types and object variables. The individual elements of an array are all of the same data type.

6.4.1.1 Declaring Arrays

Arrays occupy space in memory. The programmer specifies the array type and the number of elements required by the array so that the compiler may reserve the appropriate amount of memory. Arrays may be declared as Public (in a code module), module or local. Module arrays are declared in the general declarations using keyword Dim or Private. Local arrays are declared in a procedure using Dim or Static. Array must be declared explicitly with keyword "As".

There are two types of arrays in Visual Basic namely:

- **Fixed-size array**: The size of array always remains the same-size doesn't change during the program execution.
- **Dynamic array**: The size of the array can be changed at the run time- size changes during the program execution.

6.4.1.2 Fixed Sized Arrays

When an upper bound is specified in the declaration, a Fixed-array is created. The upper limit should always be within the range of long data type.

Declaring a fixed-array

```
Dim numbers(5) As Integer
```

In the above illustration, numbers is the name of the array, and the number 6 included in the parentheses is the upper limit of the array. The above declaration creates an array with 6 elements, with index numbers running from 0 to 5.

If we want to specify the lower limit, then the parentheses should include both the lower and upper limit along with the To keyword. An example for this is given below.

```
Dim numbers (1 To 6 ) As Integer
```

In the above statement, an array of 10 elements is declared but with indexes running from 1 to 6.

A public array can be declared using the keyword Public instead of Dim as shown below.

```
Public numbers(5) As Integer
```

6.4.1.3 Multidimensional Arrays

Arrays can have multiple dimensions. A common use of multidimensional arrays is to represent tables of values consisting of information arranged in rows and columns. To identify a particular table element, we must specify two indexes: The first (by convention) identifies the element's row and the second (by convention) identifies the element's column.

Tables or arrays that require two indexes to identify a particular element are called two dimensional arrays. Note that multidimensional arrays can have more than two dimensions. Visual Basic supports at least 60 array dimensions, but most people will need to use more than two or three dimensional-arrays.

The following statement declares a two-dimensional array 50 by 50 array within a procedure.

```
Dim AvgMarks ( 50, 50)
```

It is also possible to define the lower limits for one or both the dimensions as for fixed size arrays. An example for this is given here.

```
Dim Marks ( 101 To 200, 1 To 100)
```

An example for three dimensional-array with defined lower limits is given below.

```
Dim Details( 101 To 200, 1 To 100, 1 To 100)
```

6.4.1.4 Static and Dynamic Arrays

Basically, you can create either static or dynamic arrays. Static arrays must include a fixed number of items, and this number must be known at compile time so that the compiler can set aside the necessary amount of memory. You create a static array using a Dim statement with a constant argument:

' This is a static array.

```
Dim Names(100) As String
```

Visual Basic starts indexing the array with 0. Therefore, the preceding array actually holds 101 items.

Most programs don't use static arrays because programmers rarely know at compile time how many items you need and also because static arrays can't be resized during execution. Both these issues are solved by dynamic arrays. You declare and create dynamic arrays in two distinct steps. In general, you declare the array to account for its visibility (for example, at the beginning of a module if you want to make it visible by all the procedures of the module) using a Dim command with an empty pair of brackets. Then you create the array when you actually need it, using a ReDim statement:

' An array defined in a BAS module (with Private scope)

```
Dim Customers() As String

...

Sub Main()

' Here you create the array.

ReDim Customer(1000) As String

End Sub
```

If you're creating an array that's local to a procedure, you can do everything with a single ReDim statement:

```
Sub PrintReport()

' This array is visible only to the procedure.

ReDim Customers(1000) As String

' ...

End Sub
```

If you don't specify the lower index of an array, Visual Basic assumes it to be 0, unless an Option Base 1 statement is placed at the beginning of the module. My suggestion is this: Never use an Option Base statement because it makes code reuse more difficult. (You can't cut and paste routines without worrying about the current Option Base.) If you want to explicitly use a lower index different from 0, use this syntax instead:

```
ReDim Customers(1 To 1000) As String
```

Dynamic arrays can be re-created at will, each time with a different number of items. When you re-create a dynamic array, its contents are reset to 0 (or to an empty string) and you lose the data it contains. If you want to resize an array without losing its contents, use the ReDim Preserve command:

```
ReDim Preserve Customers(2000) As String
```

When you're resizing an array, you can't change the number of its dimensions nor the type of the values it contains. Moreover, when you're using ReDim Preserve on a multidimensional array, you can resize only its last dimension:

```
ReDim Cells(1 To 100, 10) As Integer
...
ReDim Preserve Cells(1 To 100, 20) As Integer ' This works.
ReDim Preserve Cells(1 To 200, 20) As Integer ' This doesn't.
```

Finally, you can destroy an array using the Erase statement. If the array is dynamic, Visual Basic releases the memory allocated for its elements (and you can't read or write them any longer); if the array is static, its elements are set to 0 or to empty strings.

You can use the LBound and UBound functions to retrieve the lower and upper indices. If the array has two or more dimensions, you need to pass a second argument to these functions to specify the dimension you need:

```
Print LBound(Cells, 1) ' Displays 1, lower index of 1st dimension
Print LBound(Cells) ' Same as above
Print UBound(Cells, 2) ' Displays 20, upper index of 2nd dimension
' Evaluate total number of elements.
NumEls = (UBound(Cells) _ LBound(Cells) + 1) * _
(UBound(Cells, 2) _ LBound(Cells, 2) + 1)
```

6.4.1.5 Arrays within UDTs

UDT structures can include both static and dynamic arrays. Here's a sample structure that contains both types:

```
Type MyUDT
StaticArr(100) As Long
DynamicArr() As Long
```

```
End Type

...

Dim udt As MyUDT
```

' You must DIMension the dynamic array before using it.

```
ReDim udt.DynamicArr(100) As Long
```

' You don't have to do that with static arrays.

```
udt.StaticArr(1) = 1234
```

The memory needed by a static array is allocated within the UDT structure; for example, the StaticArr array in the preceding code snippet takes exactly 400 bytes. Conversely, a dynamic array in a UDT takes only 4 bytes, which form a pointer to the memory area where the actual data is stored. Dynamic arrays are advantageous when each individual UDT variable might host a different number of array items. As with all dynamic arrays, if you don't dimension a dynamic array within a UDT before accessing its items, you get an error 9—"Subscript out of range."

6.4.2 User Defined Data Types

Variables of different data types when combined as a single variable to hold several related information is called a User-Defined data type.

A Type statement is used to define a user-defined type in the General declaration section of a form or module. User-defined data types can only be private in form while in standard modules can be public or private. An example for a user defined data type to hold the product details is as given below.

```
Private Type ProductDetails

ProdID as String

ProdName as String

Price as Currency

End Type
```

The user defined data type can be declared with a variable using the Dim statement as in any other variable declaration statement. An array of these user-defined data types can also be declared. An example to consolidate these two features is given below.

```
Dim ElectronicGoods as ProductDetails ' One Record

Dim ElectronicGoods(10) as ProductDetails ' An array of 11 records
```

A User-Defined data type can be referenced in an application by using the variable name in the procedure along with the item name in the Type block. Say, for example if the text property of a TextBox namely text1 is to be assigned the name of the electronic good, the statement can be written as given below.

```
Text1.Text = ElectronicGoods.ProdName
```

If the same is implemented as an array, then the statement becomes

```
Text1.Text = ElectronicGoods(i).ProdName
```

User-defined data types can also be passed to procedures to allow many related items as one argument.

```
Sub ProdData( ElectronicGoods as ProductDetails)

Text1.Text = ElectronicGoods.ProdName

Text1.Text = ElectronicGoods.Price

End Sub.
```