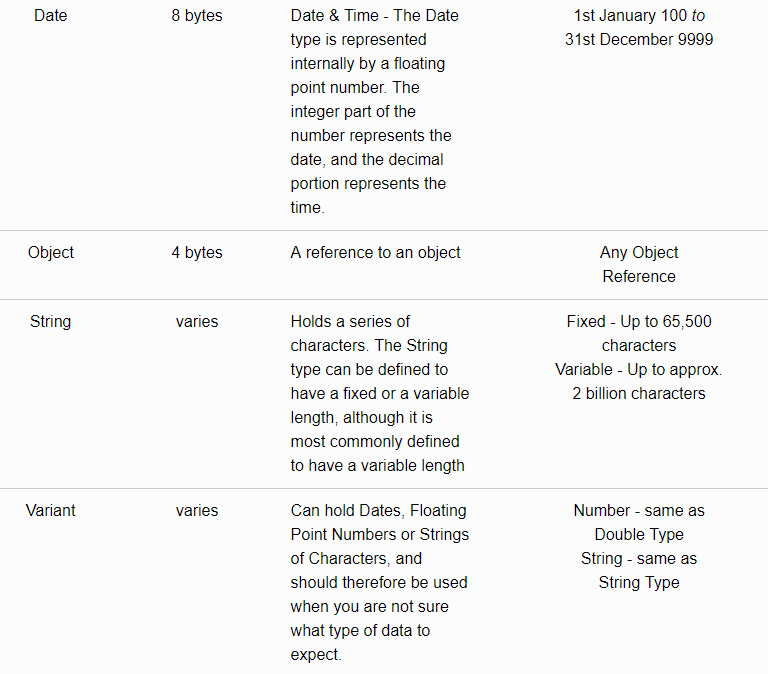


## User-Defined VBA Function & Sub Procedures

In Excel Visual Basic, a set of commands to perform a specific task is placed into a procedure, which can be a Function procedure or a Sub procedure (also known as functions and subroutines).

The main difference between a VBA Function procedure and a Sub procedure is that a Function procedure returns a result, whereas a Sub procedure does not.

Therefore, if you wish to perform a task that returns a result (e.g. summing of a group of numbers), you will generally use a Function procedure, but if you just need a set of actions to be carried out (e.g. formatting a set of cells), you might choose to use a Sub procedure.



From the above table, it is clear that you can save on memory by using specific data types (e.g. Integers rather than Longs, or Singles rather than Doubles). However, if you are planning to use the 'smaller' data types, you must be sure that your code will not encounter larger values than can be handled by the chosen data type.

**Declaring Variables & Constants (Always Declare Variables)**

Before using a variable or constant, you can *declare* it. This is done by adding a simple line of code to your macro, as follows.

To declare a variable:

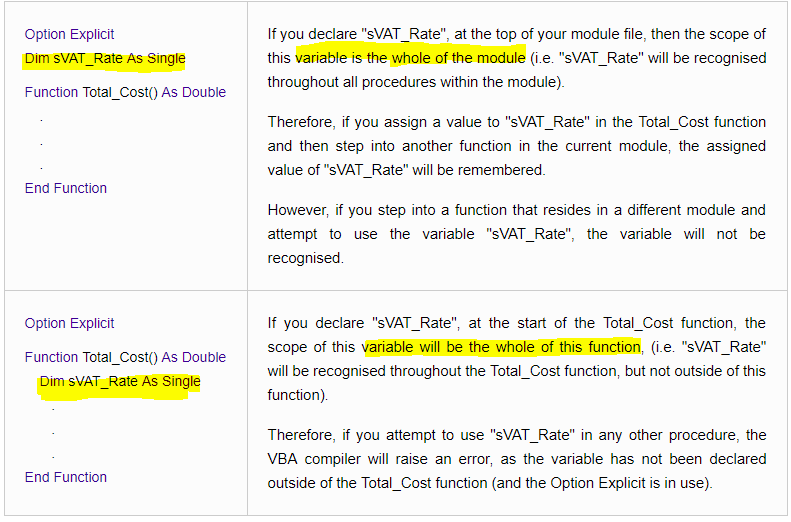
Dim *Variable\_Name* As *Data\_Type*

Note that in the above line of code, *Variable\_Name* should be replaced by your actual variable name and *Data\_Type* should be replaced by one of the above listed data types. For example:

Dim sVAT\_Rate As Single  
Dim i As Integer

Constants are declared in a similar way, except a constant should always be assigned a value when it is declared. Examples of the declaration of constants in VBA are:

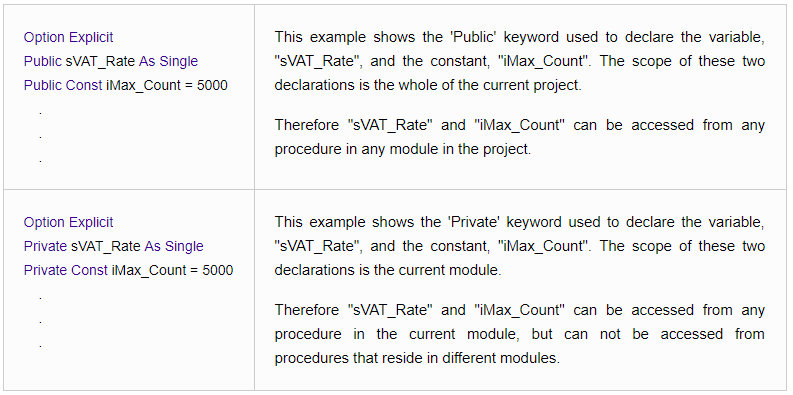
Const iMaxCount=5000  
Const iMaxScore = 100



In the above example, the module level variable has been declared using the 'Dim' keyword. However, it is possible that you may want to declare variables that can be shared with other modules. This can be specified by using the keyword ***Public*** in the declaration, instead of 'Dim'.

Note that, for a module-level variable, the 'Dim' keyword could also be replaced with the keyword ***Private*** to indicate that the scope of the variable is limited to the current module.

Constants can also use the 'Public' and 'Private' keywords, but in this case, the 'Public' or 'Private' keyword is used in addition to the 'Const' keyword (not instead of).



**Introduction to Visual Basic Arrays**

Excel Visual Basic arrays are structures which are used to store a set of related variables of the same type. Each of the entries in the array can be accessed by an index number.

For example, if you had 20 members of a team and you wanted to store all the names for use in your VBA code. You could declare 20 variables to hold the team member names, as follows:

|  |
| --- |
| Dim Team\_Member1 As String Dim Team\_Member2 As String  **.** **.** **.**  Dim Team\_Member20 As String |

Alternatively, you could use the simpler and more organised method of storing the Team members in an array of 20 String variables:

|  |
| --- |
| Dim Team\_Members(1 To 20) As String |

Once you have declared the array as above, each entry of the array can be populated as follows:

|  |
| --- |
| Team\_Members(1) = "John Smith"  **.** **.** **.** |

A further advantage of storing data in an array, rather than in individual variables, arises if you want to perform the same action on every member of the list. If the team member names were stored in 20 individual variables, you would need 20 lines of code to carry out a specific action on each name. However, if you have stored the names in an array, you can use a simple loop to carry out the action for each entry in the array.

This is shown in the example code below, which prints out each name in the Team\_Members array to Column A of the current Excel Worksheet:

|  |
| --- |
| For i = 1 To 20  Cells(i, 1).Value = Team\_Members(i)  Next i |

Even with just 20 names, the advantages of using an Array are clear, but imagine if you had 1,000 names to store! And imagine you wanted to store Surnames separately from Forenames! It would soon become almost impossible to handle this amount of data without the use of Arrays in your VBA code.

**Declaring Excel Visual Basic Arrays**

The above sections have already given some examples of Visual Basic Array declarations, but it is worth discussing this further. As seen above, a one-dimensional array can be declared as follows:

|  |
| --- |
| Dim Team\_Members(1 To 20) As String |

This declaration tells the VBA compiler that the array 'Team\_Members' has 20 variables, which are referenced by indices 1 to 20. However, we could also decide to index the 20 array variables from 0 to 19, in which case the declaration would be:

|  |
| --- |
| Dim Team\_Members(0 To 19) As String |

In fact, the default form of array indexing is to start at 0, so if you omit the start index from the declaration, and simply declare the array as:

|  |
| --- |
| Dim Team\_Members(19) As String |

Then the VBA compiler will understand this to be an array of 20 variables, which are indexed from 0 to 19.

The same rules are applied to declarations of multi-dimensional Visual Basic arrays. As shown in the previous example, a two-dimensional array is declared by separating the dimension indices by a comma:

|  |
| --- |
| Dim Jan\_Sales\_Figures(1 To 31, 1 To 5) As Currency |

However, if we omit the start indices from both dimensions, as follows:

|  |
| --- |
| Dim Jan\_Sales\_Figures(31, 5) As Currency |

this is understood to be a two-dimensional array in which the first dimension has 32 entries, indexed from 0 to 31 and the second dimension has 6 entries, indexed from 0 to 5.

**Dynamic Arrays (when the size of the table is unkown)**

In the above examples, the arrays all have fixed dimensions. However, in many cases, you may not know how big an array is going to be before run time. One way to solve this is to declare a huge array, in an attempt to cover the maximum possible size needed, but this would use up an unnecessarily large amount of memory and could slow down your program. A better option would be to use a Dynamic array, which is an array that can be sized and re-sized as many times as you like, during the execution of a macro.

A dynamic array is declared with empty parentheses, as follows:

|  |
| --- |
| Dim Team\_Members() As String |

It is then necessary to declare the dimension of the array during the execution of the code, using the ReDimstatement:

|  |
| --- |
| ReDim Team\_Members(1 To 20) |

If, during the execution of the code, you need to extend the size of an array, you can use ReDim again:

|  |
| --- |
| If Team\_Size > 20 Then  ReDim Team\_Members(1 To Team\_Size)  End If |

It should be noted that resizing a dynamic array in this way will result in the loss of all the values that had previously been stored in the array. If you want to avoid this loss, and keep the values that have previously been assigned to the array, you need to use the "Preserve" keyword, as shown below:

|  |
| --- |
| If Team\_Size > 20 Then  ReDim Preserve Team\_Members(1 To Team\_Size)  End If |

The disadvantage of using the "Preserve" keyword when resizing Visual Basic Arrays is that you can only change the upper bound of an array, not the lower bound. Also, if you have a multi-dimensional array, the use of the "Preserve" keyword limits you to changing only the last dimension of the array.