

# Data Types and Structures



[ [Variables](#gjdgxs) | [Static Variables](#30j0zll) | [Constants](#1fob9te) | [Arrays](#3znysh7) | [Dynamic Arrays](#2et92p0) ]

## ***Variables***

Variables are data structures which are used to store information. There are two main types of information which can be stored: numbers and text. Before using a variable it must first be created:

**Dim** *variable\_name* **As *Type***

Example:

**Dim price As Long  
 Dim registration\_plate As String**

| | Type | Size | Allowable Ranges | | --- | --- | --- | | **Boolean** | 2 bytes | TRUE or FALSE | | **Byte** | 1 byte | 0 to 255 | | **Currency** | 8 bytes | -922337203685477.5808 to 922337203685477.5807 | | **Date** | 8 bytes | January 1, 100 to December 31, 9999 | | **Decimal** | 14 bytes | +/-79,228,162,514,264,337,593,543,950,335 with no decimal point;  +/-7.9228162514264337593543950335 with 28 places to the right of the decimal; smallest non-zero number is  +/-0.0000000000000000000000000001 | | **Double** | 8 bytes | -1.79769313486232D308 to -4.94065645841247D-324  4.94065645841247D-324 to 1.79769313486232D308 | | **Integer** | 2 bytes | -32,768 to 32,767 | | **Long** | 4 bytes | -2,147,483,648 to 2,147,483,647 | | **Single** | 4 bytes | -3.402823D38 to -1.401298D-45  1.401298E-45 to 3.402823E38 | | **String** | 10 bytes + string length | 0 to 65,000 bytes | | **Variant** | 16 bytes | Numeric: same as Double | | 22 bytes + string length | String: same as String | | Number required by elements | User defined type | | *If a new variable is declared without specifying a type then by default it will be made type Variant.* | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

Once a new variable has been created a value can be assigned to it for storage. To do this the ‘=’ operator is used. The first example assigns a constant to a variable, while the second assigns the contents of another variable multiplied by ten.

Example 1: **price = 29.95**

Example 2: **total\_price = price \* 10**

The *scope* of a variable is defined as its operating range. There are three types of varible scope:

1. **Local** - Variable can only be used in the current procedure (use Dim within the required procedure).
2. **Module** - Variable can only be accessed in any procedure in the current form of module (use Private within the *Declarations* section on a form or module).
3. **Public** - Can be accessed in any procedure in any module (use Public within the *Declarations* section of a module).

## ***Static Variables***

Declaring variables and arrays ([see below](#3znysh7)) as *local* to the current procedure/function is useful because it minimises strange side-effects which can occur with *public* variables. However, when using a local variable in a procedure VB creates memory space to hold the variable’s contents when it reads the Dim statement, but when it gets to the end of the procedure (End Sub) it frees the space allocated to the local variable. Add the following code to a command button and see what values are printed:

**Private Sub Command1\_Click ()  
 Dim number As Integer ' Create a normal local variable  
  
 number = number + 1  
 Print number  
 End Sub**

After clicking on the command button a few times you should see a whole column of 1’s down the left hand side of the form. The value never gets above 1 even though 1 is added to the variable each time. This is because each time the procedure is called, by clicking on the command button, VB is working with a different variable. It has exactly the same name in the program but it is a completely different variable. To stop this happening enter the keyword *Static* instead of *Dim*:

**Private Sub Command1\_Click ()  
 Static number As Integer ' Create a static local variable  
  
 number = number + 1  
 Print number  
 End Sub**

This time instead of the local variable being destroyed when the procedure terminates its value remains until the whole program ends. Thus, we can now see a list of numbers being incremented by 1 every time the command button is clicked.

**Note:** The new static variable is still local in scope, if any other procedure tries to access this variable it will be unable to do so. Add a second command button, which will add a new ‘Click’ procedure, and try amending or printing the value of the static variable.

The contents of local arrays can also be preserved while the program is still running. To do this add the ‘Static’ keyword instead of ‘Dim’ just like the variable example above.

**Static salaries(199) As Long**

## ***Constants***

Constants are similar to variables but only have a single value throughout the execution of a program. The contents of variables can change as often as necessary. So why use a constant if it can only hold one value? Well often in a program it is neccassary to use the same number or string repeatedly. For example, in a program to calculate end of year accounts there will be several parts of the program refering to the current value of V.A.T. We could hard code the value into the program everywhere it is used, but it would be tedious to change if the government altered the rate. Alternatively we could use a standard variable called ‘VAT’ to hold the value and set it on the Form\_Load event. However, what happens if there is a bug in the program and the contents of this variable is changed accidentally to something else.

One solution which solves the above two problems is to use a constant. In the following example a constant named ‘VAT’ is declared and assigned the value 1.175. It is used in the Print statement with the variable bill\_total to calculate the total bill amount. Notice that instead of writing 1.175 in the formula we refer to the constant by name.

Example:

**Const VAT = 1.175 ' Declare and set the value of the constant  
 Dim bill\_total As Currency ' Declare a local variable to hold bill total  
  
 bill\_total = 560.95  
  
 Print "Total = "; bill\_total \* VAT**

Like variables, constants also have scoping rules. There are public constants available to any module or unit in the current project, module wide constants available to any unit within the current module, and local constants accessable only within the current unit (procedure/function).

1. **Local** - use ‘Const’ within the required procedure.
2. **Module** - use ‘Private Const’ within the *Declarations* section on a form or module.
3. **Public** - use ‘Public Const’ within the *Declarations* section of a module (e.g. Module1.bas).

## ***Arrays***

Variables are useful for storing small pieces of information, but not good for large amounts of very similar information. For example, to hold the salaries of two hundred employees would require 200 different variable names. A much more efficient way is to use a data structure called an *array*.

An array is similar to a row of pigeon-holes. The whole array has one name, and each pigeon-hole has an *address*. For the above salaries problem we need an array which has 200 *elements* (pigeon-holes). To do this we use the Dim command that we used to create new variables. However, a size is also allocated.

Dim *array\_name* (*size*) [As *Type*]

Example: **Dim salaries(199) As Long**

The above example creates an array with 200 elements. The size is set to 199 because by default VB starts numbering from 0.

If we know that ‘Fred’ is employee number 24 and has a salary of 25,000 then we can enter this amount into the array using:

**salaries(23) = 25000**

Conversely, if we want to know how much employee 189 earns then we can use:

**lblEarnings.Caption = salaries(188)**

**Note:** The above two examples both access the element one below the required number. This is because VB numbers the array from 0, not 1. However, VB can be forced to start at 1 by entering the statement ‘Option Base 1’ in the declarations section of a form or a module.

## ***Dynamic Arrays***

There may be times when writing an application, however, when a program needs to change the size of the array. To do this a ‘dynamic’ array can be used. First the array must be declared in the same way but without a number of elements specified:

**Dim books() As String**

To change the size of this array use the ‘ReDim’ command and a specific number of elements:

**ReDim books(99)**

Normally the contents of an array is erased when redimensioning, however to stop this use:

**ReDim Preserve books(99)**

Like variables, arrays can also have different scopes:

1. **Public** - available to any form or module contained in the project.
2. **Module** - available to any procedure on the form on which it is placed.
3. **Procedure** - available only within the procedure in which it is declared.

 [Menu](http://docs.google.com/dvbhome.html)