Excel
VBA programming
Visual Basic for Applications

Herve Hocquard

http://www.labri.fr/perso/hocquard

Program : after instructions manipulating data

VISUAL BASIC LANGUAGE

Typed data. Visual Basic offers the usual types of programming: integer, reals, booleans, character string.

Advanced data structures. Management of collections of values (enumerations, tables) and structured objects (records, classes).

Instruction sequences: it is the very basis of programming, being able to write and execute a series of commands without having to intervene between the instructions.

Algorithmic structures : conditional branches and loops.

Structured programming tools : be able to group code into procedures and functions. Code organization in modules and the possibility of distributing them.

Visual Basic is not "case sensitive", it does not differentiate between terms written in lower case and upper case.

The data type defines the type of operators that can be applied to them.

- **Digital** which can be real (double) or integer (long). Applicable operators are: +, -, *, / (real division), \ (integer division),mod (modulo)
  Example: 5/2 → 2.5; 5\2 → 2; 5 mod 2 → 1

- **Boolean** (boolean) which takes only two possible values: True and False. The operators are: not, and, or.
  Example: True and False → False

- **String of characters** (string) which corresponds to a sequence of characters delimited by quotes " ". The possible operators are concatenation, deleting a sub-part, copying a sub-part, etc.
  Example: "Toto" is a string, Toto we don't know what it is (for now).

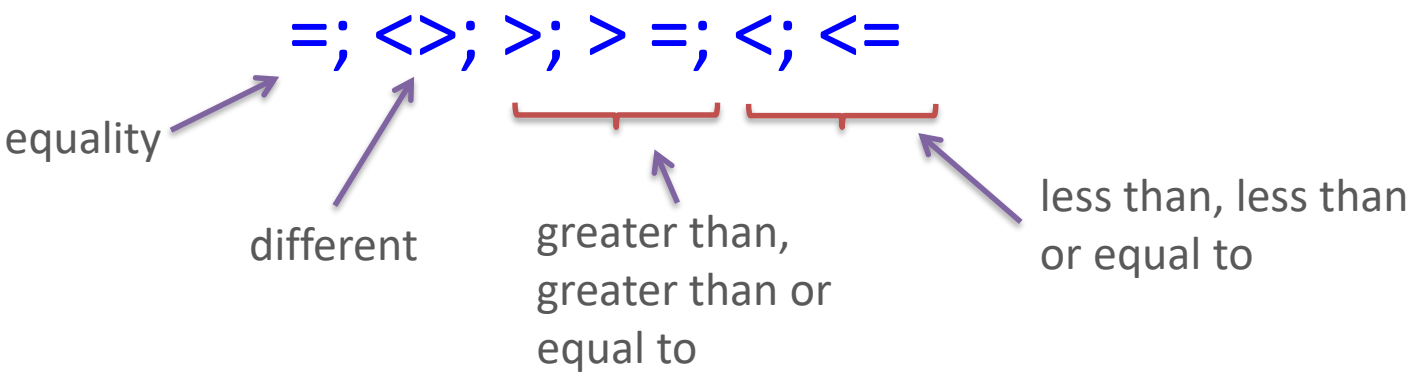
Habitually, the operations involve data of the same type and return a result of the same type.

Type	Values
Boolean	Tue or False
Integer	Integer number
Long	Integer number
Single	Real number
Double	Real number
Currency	4 digits after the,
Date	1/1/100 to 12/31/9999
String	Strings
Object	Any object
Variant	Any type

Data type

Data Type	Bytes Used	Range of Values
Byte	1 byte	0 to 255
Boolean	2 bytes	True or False
Integer	2 bytes	–32,768 to 32,767
Long	4 bytes	–2,147,483,648 to 2,147,483,647
Single	4 bytes	–3.402823E38 to –1.401298E-45 (for negative values); 1.401298E-45 to 3.402823E38 (for positive values)
Double	8 bytes	–1.79769313486232E308 to –4.94065645841247E-324 (negative values); 4.94065645841247E-324 to 1.79769313486232E308 (for positive values)
Currency	8 bytes	–922,337,203,685,477.5808 to 922,337,203,685,477.5807
Decimal	12 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
Date	8 bytes	January 1, 0100 to December 31, 9999
Object	4 bytes	Any object reference
String (variable length)	10 bytes + string length	0 to approximately 2 billion characters
String (fixed length)	Length of string	1 to approximately 65,400 characters
Variant (with numbers)	16 bytes	Any numeric value up to the range of a double data type. It can also hold special values, such as Empty, Error, Nothing, and Null.
Variant (with characters)	22 bytes + string length	0 to approximately 2 billion
User-defined	Varies	Varies by element

Comparison operators compare data of the same type, but the result is a boolean



Examples

5 > 2 → True

5 > ''toto'' → illicit

5 <> 5 → False

''toto'' > ''tata'' → True

Lawful. Comparison from left to right based on ASCII code. Stop comparisons as soon as the indecision is lifted.

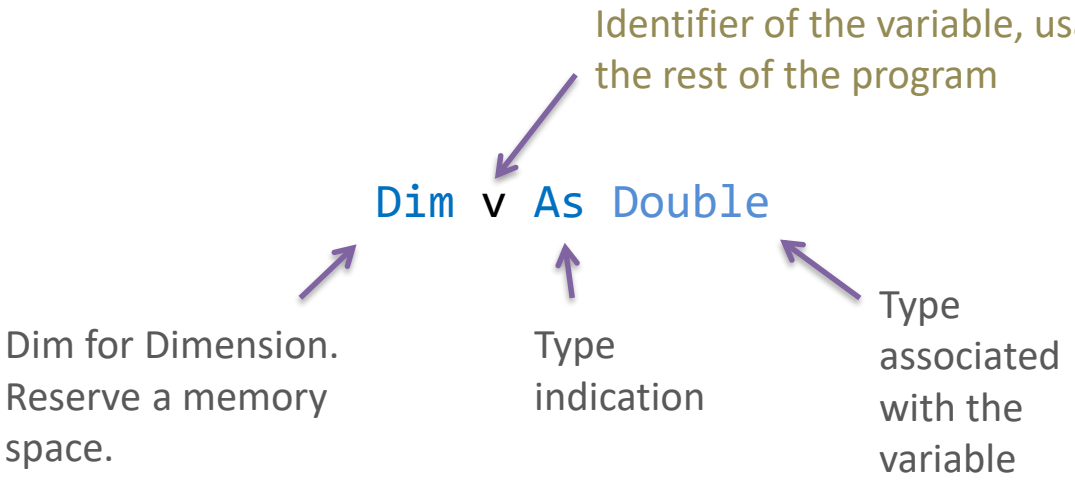
- Absolute value: Abs(-9) returns 9
- Sign: Sgn(-18) returns –1 (or 0 or 1)
- Unit truncation: Fix(-18.3) = -18
  Fix(18.3) = 18
  - Truncate the decimal part
- Integer Part: Int(13.12) returns 13, Int(-14.8) returns -15
  - $E(x) \leq x < E(x) + 1$
  - Truncate to the nearest integer number.

- Sqr, Exp, Log
  - Sqr(4) returns 2, Exp(5) returns 148.413…,
    Log(9) returns 2.197224 ... (in base e)
- Random numbers
  - Rnd returns a random number between 0 (included) and 1 (not included)
  - a = Rnd : a can be 0.12131441
  - Int ((b - a + 1) * Rnd + a) returns an integer random number between a and b
- Sin, Cos, Tan, Atn (arc-tangent)

- Date returns the current date
- Time returns the current time
  - Date and Time can return string of characters
- DateSerial returns a unique value for a given date, in the form Variant
  - dv1 = DateSerial(2003, 4, 22)
    dv2 = DateSerial(1928, 5, 3)
    dv1 - dv2 represents the number of days between these two dates
- Day, Month and Year returns the day, month and year of a date, respectively.
  - Year(Date) returns 2021 this year (in full)

The **variables** correspond to identifiers with associated values of a given type. They materialize a memory space with content that can be read or written.

Declaration of a variable



Assignment. Assign a value to the variable

```
v = 2.5
```

= is the assignment symbol. At left of = we change the content in a variable, to right we read the content of a variable. It is for this reason that the instruction v = v + 1 is lawful.

Operation and assignment

```
x = v * 2
```

The value 5 is written in the variable x which must be declared beforehand.

The following procedure demonstrates how a variable can assume different data types:

```
Sub VariantDemo()
    MyVar = True
    MyVar = MyVar * 100
    MyVar = MyVar / 4
    MyVar = "Answer: " & MyVar
    MsgBox MyVar
End Sub
```

In the VariantDemo procedure, MyVar starts out as a Boolean. The multiplication operation converts it to an Integer. The division operation converts it to a Double. And finally, it's concatenated with text to make it a String. The MsgBox statement displays the final string: Answer: -25.

To further demonstrate the potential problems in dealing with Variant data types, try executing this procedure:

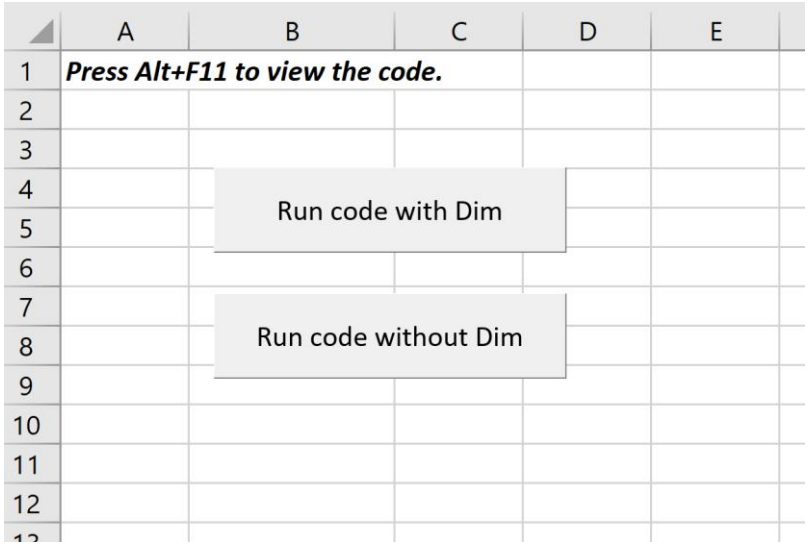
```
Sub VariantDemo2()
    MyVar = "123"
    MyVar = MyVar + MyVar
    MyVar = "Answer: " & MyVar
    MsgBox MyVar
End Sub
```

This is probably not what you wanted. When dealing with variants that contain text strings, the + operator performs string concatenation.

You can use the VBA TypeName function to determine the data type of a variable. Here's a modified version of the VariantDemo procedure. This version displays the data type of MyVar at each step.

```
Sub VariantDemo3()
    MyVar = True
    MsgBox TypeName(MyVar)
    MyVar = MyVar * 100
    MsgBox TypeName(MyVar)
    MyVar = MyVar / 4
    MsgBox TypeName(MyVar)
    MyVar = "Answer: " & MyVar
    MsgBox TypeName(MyVar)
    MsgBox MyVar
End Sub
```

Declaring variables




```
Sub TimeTest2()
'    VARIABLES NOT DECLARED
'    Store the starting time
     StartTime = Timer
'    Perform some calculations
     x = 0
     y = 0
     For i = 1 To 10000
          x = x + 1
          y = x + 1
          For j = 1 To 10000
               A = x + y + i
               B = y - x - i
               C = x / y * i
          Next j
     Next i
'    Get ending time
     EndTime = Timer
'    Display total time in seconds
     MsgBox Format(EndTime - StartTime, "0.0")
End Sub
```

```
Sub TimeTest1()
'    VARIABLES DECLARED
     Dim x As Long, y As Long
     Dim A As Double, B As Double, C As Double
     Dim i As Long, j As Long
     Dim StartTime As Date, EndTime As Date
'    Store the starting time
     StartTime = Timer
'    Perform some calculations
     x = 0
     y = 0
     For i = 1 To 10000
          x = x + 1
          y = x + 1
          For j = 1 To 10000
               A = x + y + i
               B = y - x - i
               C = x / y * i
          Next j
     Next i
'    Get ending time
     EndTime = Timer
'    Display total time in seconds
     MsgBox Format(EndTime - StartTime, "0.0")
End Sub
```

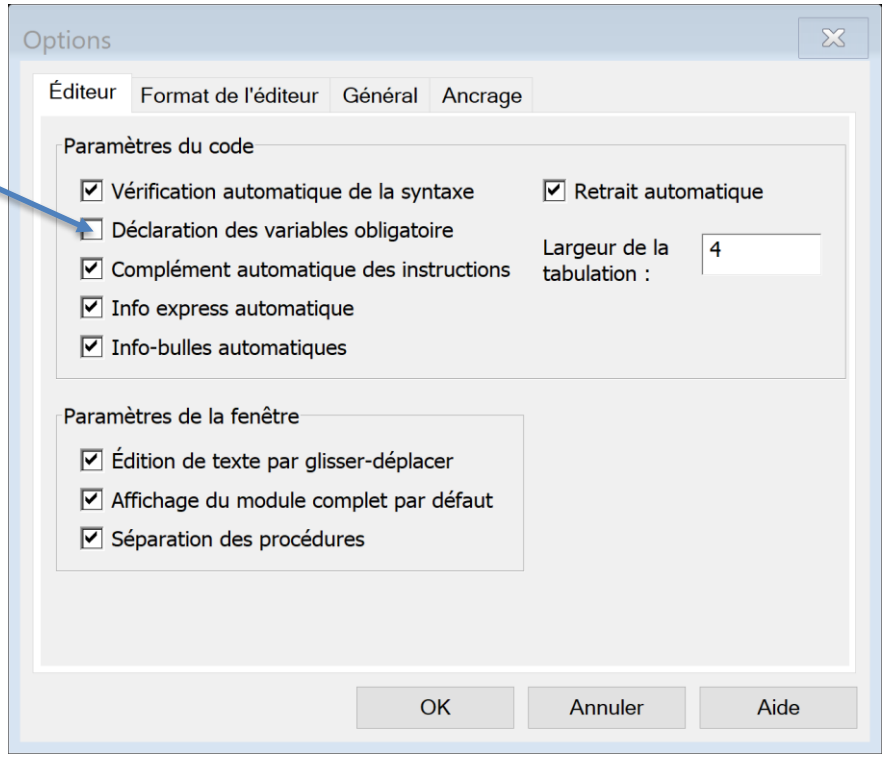
On my system, this routine took 5.6 seconds to run. (The time will vary, depending on your system's processor speed.) I then turned the Dim statements, which declare the data types, into comments by adding an apostrophe at the beginning of the lines. As a result, VBA used the default data type, Variant. I ran the procedure again. It took 9.3 seconds, less than two times as long as before.

The moral is simple: If you want your VBA applications to run as fast as possible, declare your variables!

To force yourself to declare all the variables that you use, include the following as the first instruction in your VBA module: Option Explicit.

If the Require Variable Declaration option is set, VBE inserts the following statement at the beginning of each new VBA module that you insert

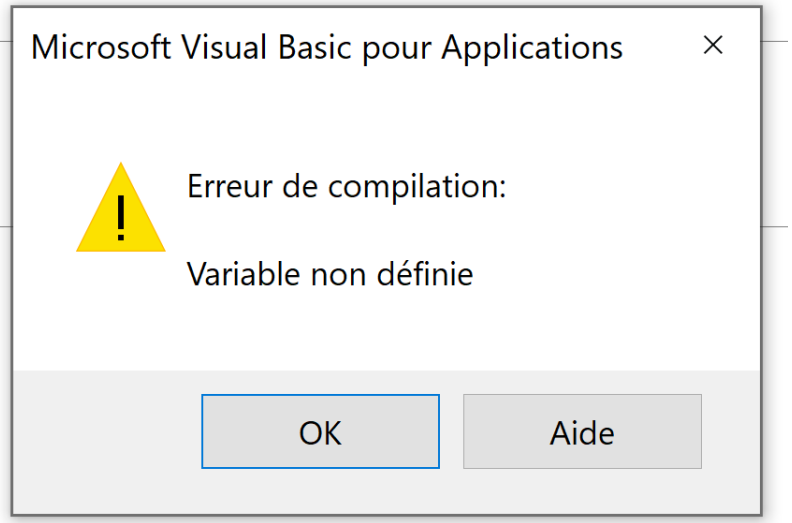
Option Explicit



When the statement “Option Explicit” is present, VBA won’t even execute a procedure if it contains an undeclared variable name. VBA issues the error message shown below, and you must declare the variable before you can proceed.

```
Option Explicit

Sub product()
    x = 2
    y = 3
    Debug.Print x * y
End Sub
```



To ensure that the Option Explicit statement is inserted automatically whenever you insert a new VBA module, enable the Require Variable Declaration option in the Editor tab of the VBE Options dialog box (choose Tools➜Options). I highly recommend doing so. Be aware, however, that this option doesn’t affect existing modules.

Thanks to VBA, the data type conversion of undeclared variables is automatic. This process may seem like an easy way out, but remember that you sacrifice speed and memory — and you run the risk of errors that you may not even know about. Declaring each variable in a procedure before you use it is an excellent habit. Declaring a variable tells VBA its name and data type. Declaring variables provides two main benefits:

- **Your programs run faster** and use memory more efficiently. The default data type, Variant, causes VBA to repeatedly perform time-consuming checks and reserve more memory than necessary. If VBA knows the data type, it doesn't have to investigate, and it can reserve just enough memory to store the data.

- **You avoid problems involving misspelled variable names.** This benefit assumes that you use Option Explicit to force yourself to declare all. Say that you use an undeclared variable named CurrentRate. At some point in your routine, however, you insert the statement CurentRate = .075. This misspelled variable name, which is very difficult to spot, will likely cause your routine to give incorrect results.

A local variable is a variable declared within a procedure. You can use local variables only in the procedure in which they're declared. When the procedure ends, the variable no longer exists, and Excel frees up the memory that the variable used. If you need the variable to retain its value when the procedure ends, declare it as a Static variable.

The most common way to declare a local variable is to place a Dim statement between a Sub statement and an End Sub statement. Dim statements usually are placed right after the Sub statement, before the procedure's code.

Dim is a shortened form of Dimension. In old versions of BASIC, this statement was used exclusively to declare the dimensions for an array. In VBA, the Dim keyword is used to declare any variable, not just arrays.

The following procedure uses six local variables declared by using Dim statements:

```
Sub MySub()
    Dim x As Integer
    Dim First As Long
    Dim InterestRate As Single
    Dim TodaysDate As Date
    Dim UserName As String
    Dim MyValue
    ' - [The procedure's code goes here] -
End Sub
```

Notice that the last Dim statement in the preceding example doesn't declare a data type; it simply names the variable. As a result, that variable becomes a variant.

You also can declare several variables with a single Dim statement. For example:

```
Dim x As Integer, y As Integer, z As Integer
Dim First As Long, Last As Double
```

Unlike some languages, VBA doesn't let you declare a group of variables to be a particular data type by separating the variables with commas. For example, the following statement, although valid, does not declare all the variables as integers:

```
Dim i, j, k As Integer
```

In VBA, only k is declared to be an integer; the other variables are declared variants. To declare i, j, and k as integers, use this statement:

```
Dim i As Integer, j As Integer, k As Integer
```

Sometimes, you want a variable to be available to all procedures in a module. If so, just declare the variable before the module's first procedure (outside of any procedures or functions).
In the following example, the Dim statement is the first instruction in the module. Both Procedure1 and Procedure2 have access to the CurrentValue variable.

```
Dim CurrentValue As Long

Sub Procedure1()
    '- [Code goes here] -
End Sub

Sub Procedure2()
    '- [Code goes here] -
End Sub
```

The value of a module-wide variable retains its value when a procedure ends normally (that is, when it reaches the **End Sub** or **End Function** statement). An exception is if the procedure is halted with an **End** statement. When VBA encounters an **End** statement, all module-wide variables in all modules lose their values.

To make a variable available to all the procedures in all the VBA modules in a project, declare the variable at the module level (before the first procedure declaration) by using the **Public** keyword rather than Dim. Here's an example:

```
Public CurrentRate As Long
```

The Public keyword makes the CurrentRate variable available to any procedure in the VBA project, even those in other modules in the project. You must insert this statement before the first procedure in a module (any module). This type of declaration must appear in a standard VBA module, not in a code module for a sheet or a UserForm.

Static variables are a special case. They're declared at the procedure level, and they retain their value when the procedure ends normally. However, if the procedure is halted by an **End** statement, static variables do lose their values. Note that an **End** statement is not the same as an **End Sub** statement.

You declare static variables by using the **Static keyword**:

```
Sub MySub()
    Static Counter As Long
    '- [Code goes here] -
End Sub
```

A variable's value may change while a procedure is executing (that's why it's called a variable). Sometimes, you need to refer to a named value or string that never changes: a constant.

Using constants throughout your code in place of hard-coded values or strings is an excellent programming practice. For example, if your procedure needs to refer to a specific value (such as an interest rate) several times, it's better to declare the value as a constant and use the constant's name rather than its value in your expressions.

This technique not only makes your code more readable, it also makes it easier to change should the need arise — you have to change only one instruction rather than several.

You declare constants with the **Const** statement. Here are some examples:

```
Const NumQuarters As Integer = 4
Const Rate = .0725, Period = 12
Const ModName As String = "Budget Macros"‘
Public Const AppName As String = "Budget Application"‘
```

The second example doesn’t declare a data type. Consequently, VBA determines the data type from the value. The Rate variable is a Double, and the Period variable is an Integer. Because a constant never changes its value, you normally want to declare your constants as a specific data type.

If you want a constant to be available within a single procedure only, declare it after the Sub or Function statement to make it a local constant.

To make a constant available to all procedures in a module, declare it before the first procedure in the module.

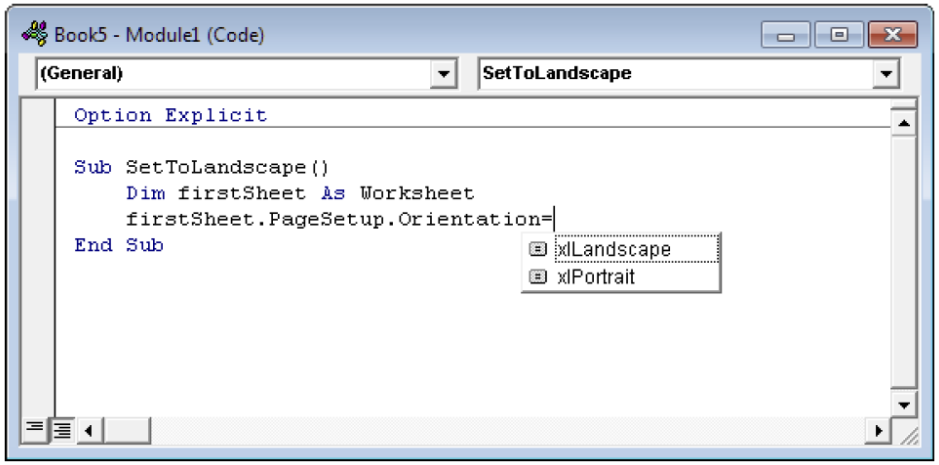
To make a constant available to all modules in the workbook, use the **Public keyword** and declare the constant before the first procedure in a module.
For example:

```
Public Const InterestRate As Double = 0.0725
```

Excel and VBA make available many predefined constants, which you can use without declaring. In fact, you don't even need to know the value of these constants to use them. The macro recorder generally uses constants rather than actual values.
The following procedure uses a built-in constant (xlLandscape) to set the page orientation to landscape for the active sheet:

```
Sub SetToLandscape()
    ActiveSheet.PageSetup.Orientation = xlLandscape
End Sub
```

In many cases, VBA lists all the constants that you can assign to a property.



Like Excel, VBA can manipulate both numbers and text (strings). There are two types of strings in VBA:

- Fixed-length strings are declared with a specified number of characters. The maximum length is 65,535 characters.

- Variable-length strings theoretically can hold up to 2 billion characters.

Each character in a string requires 1 byte of storage, plus a small amount of storage for the header of each string. When you declare a variable with a Dim statement as data type String, you can specify the length if you know it (that is, a fixed-length string), or you can let VBA handle it dynamically (a variable-length string).

In the following example, the MyString variable is declared to be a string with a maximum length of 50 characters. YourString is also declared as a string; but it's a variable-length string, so its length is not fixed.

```
Dim MyString As String * 50
Dim YourString As String
```

You can use a string variable to store a date, but if you do, it's not a real date (meaning you can't perform date calculations with it). Using the Date data type is a better way to work with dates.
A variable defined as a date uses 8 bytes of storage and can hold dates ranging from January 1, 0100, to December 31, 9999.
That's a span of nearly 10,000 years — more than enough for even the most aggressive financial forecast! The Date data type is also useful for storing time-related data. In VBA, you specify dates and times by enclosing them between two hash marks (#).

It is commonly known that Excel has a date bug: It incorrectly assumes that the year 1900 is a leap year. Even though there was no February 29, 1900, Excel accepts the following formula and displays the result as the 29th day of February, 1900: `=Date(1900,2,29)`
VBA doesn't have this date bug. The VBA equivalent of Excel's DATE function is DateSerial. The following expression (correctly) returns March 1, 1900:

```
DateSerial(1900,2,29)
```

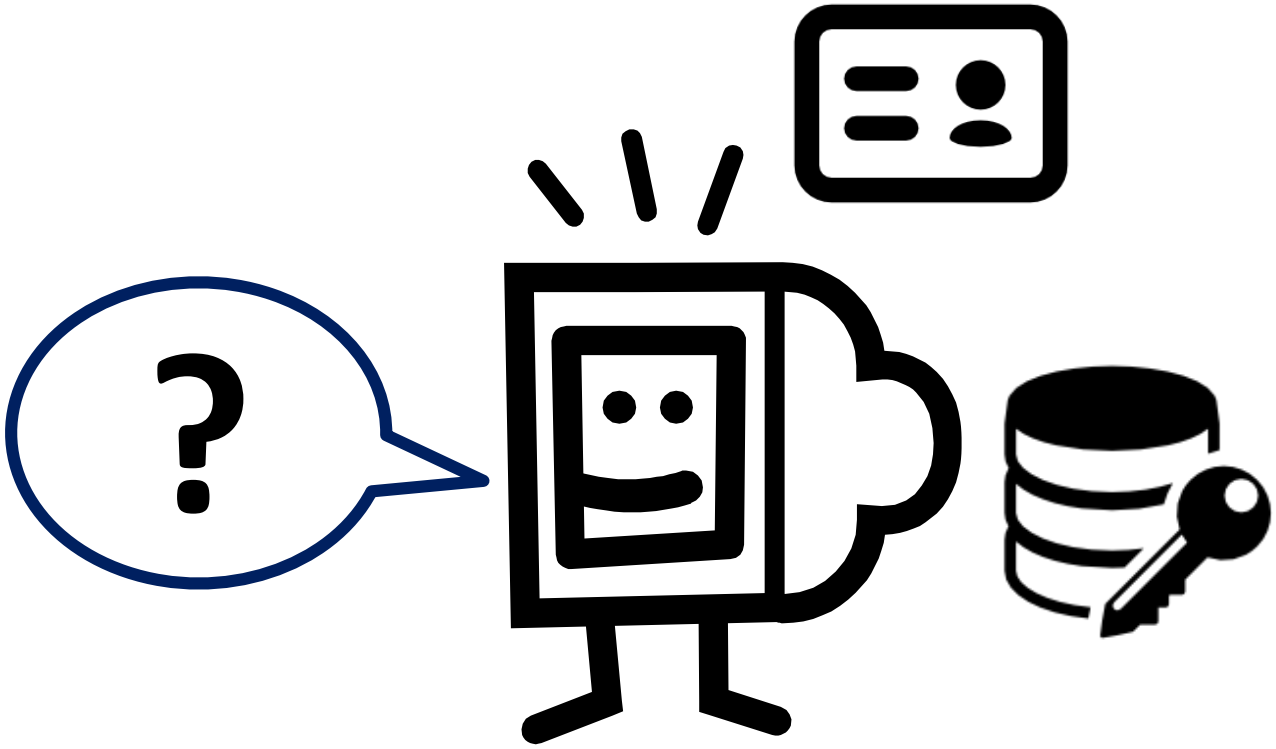
Therefore, Excel's date serial number system doesn't correspond exactly to the VBA date serial number system. These two systems return different values for dates between January 1, 1900, and February 28, 1900.

The range of dates that VBA can handle is much larger than Excel's own date range, which begins with January 1, 1900, and extends through December 31, 9999. Therefore, be careful that you don't attempt to use a date in a worksheet that is outside Excel's acceptable date range.

Here are some examples of declaring variables and constants as Date data types:

```
Dim Today As Date
Dim StartTime As Date
Const FirstDay As Date = #1/1/2013#
Const Noon = #12:00:00#
```

Dates are always defined using month/day/year format, even if your system is set to display dates in a different format (for example, day/month/year).
If you use a message box to display a date, it's displayed according to your system's short date format. Similarly, a time is displayed according to your system's time format (either 12- or 24-hour). You can modify these system settings by using the Regional Settings option in the Windows Control Panel.



To be continued with Custom Functions…

Thank you

Herve Hocquard(hocquard@labri.fr)

http://www.labri.fr/perso/hocquard/Teaching.html