Chapter 5: Declaring Variables

Section 5.1: Type Hints

Type Hints are **heavily** discouraged. They exist and are documented here for historical and backward-compatibility reasons. You should use the `As [DataType]` syntax instead.

```
Public Sub ExampleDeclaration()

        Dim someInteger% '% Equivalent to "As Integer"
        Dim someLong&    '& Equivalent to "As Long"
        Dim someDecimal@ '@ Equivalent to "As Currency"
        Dim someSingle!  '! Equivalent to "As Single"
        Dim someDouble#  '# Equivalent to "As Double"
        Dim someString$  '$ Equivalent to "As String"

        Dim someLongLong^  '^ Equivalent to "As LongLong" in 64-bit VBA hosts
End Sub
```

Type hints significantly decrease code readability and encourage a legacy Hungarian Notation which *also* hinders readability:

```
Dim strFile$
Dim iFile%
```

Instead, declare variables closer to their usage and name things for what they're used, not after their type:

```
Dim path As String
Dim handle As Integer
```

Type hints can also be used on literals, to enforce a specific type. By default, a numeric literal smaller than 32,768 will be interpreted as an `Integer` literal, but with a type hint you can control that:

```
Dim foo 'implicit Variant
foo = 42& ' foo is now a Long
foo = 42# ' foo is now a Double
Debug.Print TypeName(42!) ' prints "Single"
```

Type hints are usually not needed on literals, because they would be assigned to a variable declared with an explicit type, or implicitly converted to the appropriate type when passed as parameters. Implicit conversions can be avoided using one of the explicit type conversion functions:

```
'Calls procedure DoSomething and passes a literal 42 as a Long using a type hint
DoSomething 42&

'Calls procedure DoSomething and passes a literal 42 explicitly converted to a Long
DoSomething CLng(42)
```

String-returning built-in functions

The majority of the built-in functions that handle strings come in two versions: A loosely typed version that returns a `Variant`, and a strongly typed version (ending with $) that returns a `String`. Unless you are assigning the return value to a `Variant`, you should prefer the version that returns a `String` - otherwise there is an implicit conversion of the return value.

```
Debug.Print Left(foo, 2)  'Left returns a Variant
Debug.Print Left$(foo, 2) 'Left$ returns a String
```

These functions are:

- VBA.Conversion.Error -> VBA.Conversion.Error$
- VBA.Conversion.Hex -> VBA.Conversion.Hex$
- VBA.Conversion.Oct -> VBA.Conversion.Oct$
- VBA.Conversion.Str -> VBA.Conversion.Str$
- VBA.FileSystem.CurDir -> VBA.FileSystem.CurDir$
- VBA.[_HiddenModule].Input -> VBA.[_HiddenModule].Input$
- VBA.[_HiddenModule].InputB -> VBA.[_HiddenModule].InputB$
- VBA.Interaction.Command -> VBA.Interaction.Command$
- VBA.Interaction.Environ -> VBA.Interaction.Environ$
- VBA.Strings.Chr -> VBA.Strings.Chr$
- VBA.Strings.ChrB -> VBA.Strings.ChrB$
- VBA.Strings.ChrW -> VBA.Strings.ChrW$
- VBA.Strings.Format -> VBA.Strings.Format$
- VBA.Strings.LCase -> VBA.Strings.LCase$
- VBA.Strings.Left -> VBA.Strings.Left$
- VBA.Strings.LeftB -> VBA.Strings.LeftB$
- VBA.Strings.LTtrim -> VBA.Strings.LTrim$
- VBA.Strings.Mid -> VBA.Strings.Mid$
- VBA.Strings.MidB -> VBA.Strings.MidB$
- VBA.Strings.Right -> VBA.Strings.Right$
- VBA.Strings.RightB -> VBA.Strings.RightB$
- VBA.Strings.RTrim -> VBA.Strings.RTrim$
- VBA.Strings.Space -> VBA.Strings.Space$
- VBA.Strings.Str -> VBA.Strings.Str$
- VBA.Strings.String -> VBA.Strings.String$
- VBA.Strings.Trim -> VBA.Strings.Trim$
- VBA.Strings.UCase -> VBA.Strings.UCase$

Note that these are function *aliases*, not quite *type hints*. The `Left` function corresponds to the hidden `B_Var_Left` function, while the `Left$` version corresponds to the hidden `B_Str_Left` function.

In very early versions of VBA the $ sign isn't an allowed character and the function name had to be enclosed in square brackets. In Word Basic, there were many, many more functions that returned strings that ended in $.

Section 5.2: Variables

Scope

A variable can be declared (in increasing visibility level):

- At procedure level, using the **`Dim`** keyword in any procedure; a *local variable*.
- At module level, using the **`Private`** keyword in any type of module; a *private field*.
- At instance level, using the **`Friend`** keyword in any type of class module; a *friend field*.
- At instance level, using the **`Public`** keyword in any type of class module; a *public field*.
- Globally, using the **`Public`** keyword in a *standard module*; a *global variable*.

Variables should always be declared with the smallest possible scope: prefer passing parameters to procedures, rather than declaring global variables.

See Access Modifiers for more information.

Local variables

Use the `Dim` keyword to declare a *local variable*:

```
Dim identifierName [As Type][, identifierName [As Type], ...]
```

The `[As Type]` part of the declaration syntax is optional. When specified, it sets the variable's data type, which determines how much memory will be allocated to that variable. This declares a `String` variable:

```
Dim identifierName As String
```

When a type is not specified, the type is implicitly `Variant`:

```
Dim identifierName 'As Variant is implicit
```

The VBA syntax also supports declaring multiple variables in a single statement:

```
Dim someString As String, someVariant, someValue As Long
```

Notice that the `[As Type]` has to be specified for each variable (other than 'Variant' ones). This is a relatively common trap:

```
Dim integer1, integer2, integer3 As Integer 'Only integer3 is an Integer.
                                            'The rest are Variant.
```

Static variables

Local variables can also be `Static`. In VBA the `Static` keyword is used to make a variable "remember" the value it had, last time a procedure was called:

```
Private Sub DoSomething()
    Static values As Collection
    If values Is Nothing Then
        Set values = New Collection
        values.Add "foo"
        values.Add "bar"
    End If
    DoSomethingElse values
End Sub
```

Here the `values` collection is declared as a `Static` local; because it's an *object variable*, it is initialized to `Nothing`. The condition that follows the declaration verifies if the object reference was `Set` before - if it's the first time the procedure runs, the collection gets initialized. `DoSomethingElse` might be adding or removing items, and they'll still be in the collection next time `DoSomething` is called.

Alternative

> VBA's `Static` keyword can easily be misunderstood - *especially* by seasoned programmers that usually work in other languages. In many languages, `static` is used to make a class member (field, property, method, ...) belong to the *type* rather than to the *instance*. Code in `static` context cannot reference code in *instance* context. The VBA `Static` keyword means something wildly different.

Often, a `Static` local could just as well be implemented as a `Private`, module-level variable (field) - however this challenges the principle by which a variable should be declared with the smallest possible scope; trust your instincts, use whichever you prefer - both will work... but using `Static` without understanding what it does could lead to interesting bugs.

Dim vs. Private

The `Dim` keyword is legal at procedure and module levels; its usage at module level is equivalent to using the `Private` keyword:

```
Option Explicit
Dim privateField1 As Long 'same as Private privateField2 as Long
Private privateField2 As Long 'same as Dim privateField2 as Long
```

The `Private` keyword is only legal at module level; this invites reserving `Dim` for local variables and declaring module variables with `Private`, especially with the contrasting `Public` keyword that would have to be used anyway to declare a public member. Alternatively use `Dim` *everywhere* - what matters is *consistency*:

"Private fields"

- **DO** use `Private` to declare a module-level variable.
- **DO** use `Dim` to declare a local variable.
- **DO NOT** use `Dim` to declare a module-level variable.

"Dim everywhere"

- **DO** use `Dim` to declare anything private/local.
- **DO NOT** use `Private` to declare a module-level variable.
- **AVOID** declaring `Public` fields.*

*In general, one should avoid declaring `Public` or `Global` fields anyway.

Fields

A variable declared at module level, in the *declarations section* at the top of the module body, is a *field*. A `Public` field declared in a *standard module* is a *global variable*:

```
Public PublicField As Long
```

A variable with a global scope can be accessed from anywhere, including other VBA projects that would reference the project it's declared in.

To make a variable global/public, but only visible from within the project, use the `Friend` modifier:

```
Friend FriendField As Long
```

This is especially useful in add-ins, where the intent is that other VBA projects reference the add-in project and can consume the public API.

```
Friend FriendField As Long 'public within the project, aka for "friend" code
Public PublicField As Long 'public within and beyond the project
```

Friend fields are not available in standard modules.

Instance Fields

A variable declared at module level, in the *declarations section* at the top of the body of a class module (including `ThisWorkbook`, `ThisDocument`, `Worksheet`, `UserForm` and *class modules*), is an *instance field*: it only exists as long as there's an *instance* of the class around.

```
'> Class1
Option Explicit
Public PublicField As Long
```

```
'> Module1
Option Explicit
Public Sub DoSomething()
    'Class1.PublicField means nothing here
    With New Class1
        .PublicField = 42
    End With
    'Class1.PublicField means nothing here
End Sub
```

Encapsulating fields

Instance data is often kept `Private`, and dubbed *encapsulated*. A private field can be exposed using a `Property` procedure. To expose a private variable publicly without giving write access to the caller, a class module (or a standard module) implements a `Property Get` member:

```
Option Explicit
Private encapsulated As Long

Public Property Get SomeValue() As Long
    SomeValue = encapsulated
End Property

Public Sub DoSomething()
    encapsulated = 42
End Sub
```

The class itself can modify the encapsulated value, but the calling code can only access the `Public` members (and `Friend` members, if the caller is in the same project).

To allow the caller to modify:

- An encapsulated **value**, a module exposes a `Property Let` member.
- An encapsulated **object reference**, a module exposes a `Property Set` member.

Section 5.3: Constants (Const)

If you have a value that never changes in your application, you can define a named constant and use it in place of a literal value.

You can use Const only at module or procedure level. This means the declaration context for a variable must be a class, structure, module, procedure, or block, and cannot be a source file, namespace, or interface.

```
Public Const GLOBAL_CONSTANT As String = "Project Version #1.000.000.001"
Private Const MODULE_CONSTANT As String = "Something relevant to this Module"

Public Sub ExampleDeclaration()
```

```
    Const SOME_CONSTANT As String = "Hello World"

    Const PI As Double = 3.141592653

End Sub
```

Whilst it can be considered good practice to specify Constant types, it isn't strictly required. Not specifying the type will still result in the correct type:

```
Public Const GLOBAL_CONSTANT = "Project Version #1.000.000.001" 'Still a string
Public Sub ExampleDeclaration()

    Const SOME_CONSTANT = "Hello World"           'Still a string
    Const DERIVED_CONSTANT = SOME_CONSTANT        'DERIVED_CONSTANT is also a string
    Const VAR_CONSTANT As Variant = SOME_CONSTANT 'VAR_CONSTANT is Variant/String

    Const PI = 3.141592653        'Still a double
    Const DERIVED_PI = PI         'DERIVED_PI is also a double
    Const VAR_PI As Variant = PI  'VAR_PI is Variant/Double

End Sub
```

Note that this is specific to Constants and in contrast to variables where not specifying the type results in a Variant type.

While it is possible to explicitly declare a constant as a String, it is not possible to declare a constant as a string using fixed-width string syntax

```
'This is a valid 5 character string constant
Const FOO As String = "ABCDE"

'This is not valid syntax for a 5 character string constant
Const FOO As String * 5 = "ABCDE"
```

Section 5.4: Declaring Fixed-Length Strings

In VBA, Strings can be declared with a specific length; they are automatically padded or truncated to maintain that length as declared.

```
Public Sub TwoTypesOfStrings()

    Dim FixedLengthString As String * 5 ' declares a string of 5 characters
    Dim NormalString As String

    Debug.Print FixedLengthString       ' Prints "     "
    Debug.Print NormalString            ' Prints ""

    FixedLengthString = "123"           ' FixedLengthString now equals "123  "
    NormalString = "456"                ' NormalString now equals "456"

    FixedLengthString = "123456"        ' FixedLengthString now equals "12345"
    NormalString = "456789"             ' NormalString now equals "456789"

End Sub
```

Section 5.5: When to use a Static variable

A Static variable declared locally is not destructed and does not lose its value when the Sub procedure is exited. Subsequent calls to the procedure do not require re-initialization or assignment although you may want to 'zero' any remembered value(s).

These are particularly useful when late binding an object in a 'helper' sub that is called repeatedly.

Snippet 1: Reuse a Scripting.Dictionary object across many worksheets

```
Option Explicit

Sub main()
    Dim w As Long

    For w = 1 To Worksheets.Count
        processDictionary ws:=Worksheets(w)
    Next w
End Sub

Sub processDictionary(ws As Worksheet)
    Dim i As Long, rng As Range
    Static dict As Object

    If dict Is Nothing Then
        'initialize and set the dictionary object
        Set dict = CreateObject("Scripting.Dictionary")
        dict.CompareMode = vbTextCompare
    Else
        'remove all pre-existing dictionary entries
        ' this may or may not desired if a single dictionary of entries
        ' from all worksheets is preferred
        dict.RemoveAll
    End If

    With ws

        'work with a fresh dictionary object for each worksheet
        ' without constructing/destructing a new object each time
        ' or do not clear the dictionary upon subsequent uses and
        ' build a dictionary containing entries from all worksheets

    End With
End Sub
```

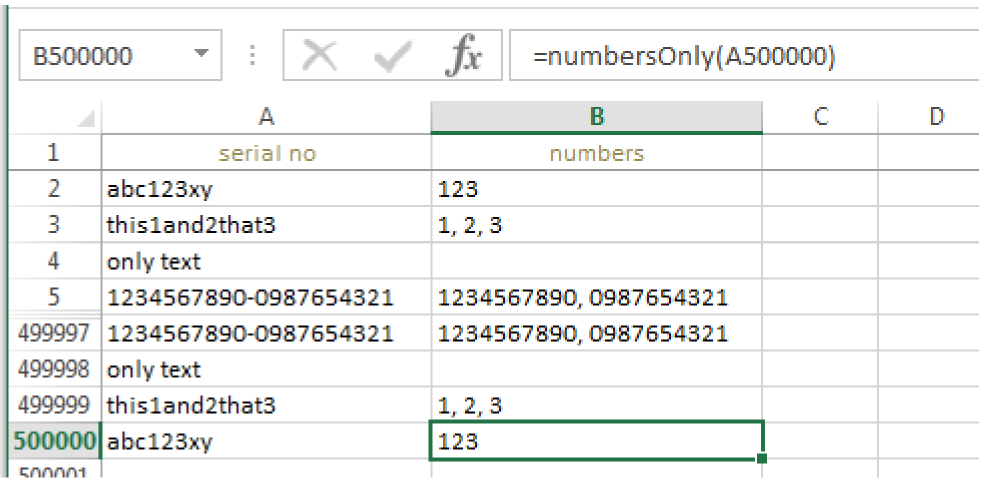
Snippet 2: Create a worksheet UDF that late binds the VBScript.RegExp object

```
Option Explicit

Function numbersOnly(str As String, _
                     Optional delim As String = ", ")
    Dim n As Long, nums() As Variant
    Static rgx As Object, cmat As Object

    'with rgx as static, it only has to be created once
    'this is beneficial when filling a long column with this UDF
    If rgx Is Nothing Then
        Set rgx = CreateObject("VBScript.RegExp")
    Else
        Set cmat = Nothing
    End If

    With rgx
        .Global = True
        .MultiLine = True
        .Pattern = "[0-9]{1,999}"
        If .Test(str) Then
            Set cmat = .Execute(str)
            'resize the nums array to accept the matches
            ReDim nums(cmat.Count - 1)
            'populate the nums array with the matches
            For n = LBound(nums) To UBound(nums)
                nums(n) = cmat.Item(n)
            Next n
            'convert the nums array to a delimited string
            numbersOnly = Join(nums, delim)
        Else
            numbersOnly = vbNullString
        End If
    End With
End Function
```


B500000 | =numbersOnly(A500000)

	A	B	C	D
1	serial no	numbers		
2	abc123xy	123		
3	this1and2that3	1, 2, 3		
4	only text			
5	1234567890-0987654321	1234567890, 0987654321		
499997	1234567890-0987654321	1234567890, 0987654321		
499998	only text			
499999	this1and2that3	1, 2, 3		
500000	abc123xy	123		
500001				

Example of UDF with Static object filled through a half-million rows

> *Elapsed times to fill 500K rows with UDF:
> - with **Dim rgx As Object**: 148.74 seconds
> - with **Static rgx As Object**: 26.07 seconds
>
> * These should be considered for relative comparison only. Your own results will vary according to the complexity and
> scope of the operations performed.

Remember that a UDF is not calculated once in the lifetime of a workbook. Even a non-volatile UDF will recalculate whenever the values within the range(s) it references are subject to change. Each subsequent recalculation event only increases the benefits of a statically declared variable.

- A Static variable is available for the lifetime of the module, not the procedure or function in which it was declared and assigned.
- Static variables can only be declared locally.
- Static variable hold many of the same properties of a private module level variable but with a more restricted scope.

Related reference: Static (Visual Basic)

Section 5.6: Implicit And Explicit Declaration

If a code module does not contain `Option Explicit` at the top of the module, then the compiler will automatically (that is, "implicitly") create variables for you when you use them. They will default to variable type `Variant`.

```
Public Sub ExampleDeclaration()

    someVariable = 10                  '
    someOtherVariable = "Hello World"
    'Both of these variables are of the Variant type.

End Sub
```

In the above code, if `Option Explicit` is specified, the code will interrupt because it is missing the required `Dim` statements for `someVariable` and `someOtherVariable`.

```
Option Explicit

Public Sub ExampleDeclaration()

    Dim someVariable As Long
    someVariable = 10

    Dim someOtherVariable As String
    someOtherVariable = "Hello World"

End Sub
```

It is considered best practice to use Option Explicit in code modules, to ensure that you declare all variables.

See VBA Best Practices how to set this option by default.

Section 5.7: Access Modifiers

The `Dim` statement should be reserved for local variables. At module-level, prefer explicit access modifiers:

- `Private` for private fields, which can only be accessed within the module they're declared in.
- `Public` for public fields and global variables, which can be accessed by any calling code.
- `Friend` for variables public within the project, but inaccessible to other referencing VBA projects (relevant for add-ins)
- `Global` can also be used for `Public` fields in standard modules, but is illegal in class modules and is obsolete anyway - prefer the `Public` modifier instead. This modifier isn't legal for procedures either.

Access modifiers are applicable to variables and procedures alike.

```
Private ModuleVariable As String
Public GlobalVariable As String

Private Sub ModuleProcedure()

    ModuleVariable = "This can only be done from within the same Module"

End Sub

Public Sub GlobalProcedure()

    GlobalVariable = "This can be done from any Module within this Project"
```

```
End Sub
```

Option Private Module

Public parameterless `Sub` procedures in standard modules are exposed as macros and can be attached to controls and keyboard shortcuts in the host document.

Conversely, public `Function` procedures in standard modules are exposed as user-defined functions (UDF's) in the host application.

Specifying `Option Private Module` at the top of a standard module prevents its members from being exposed as macros and UDF's to the host application.