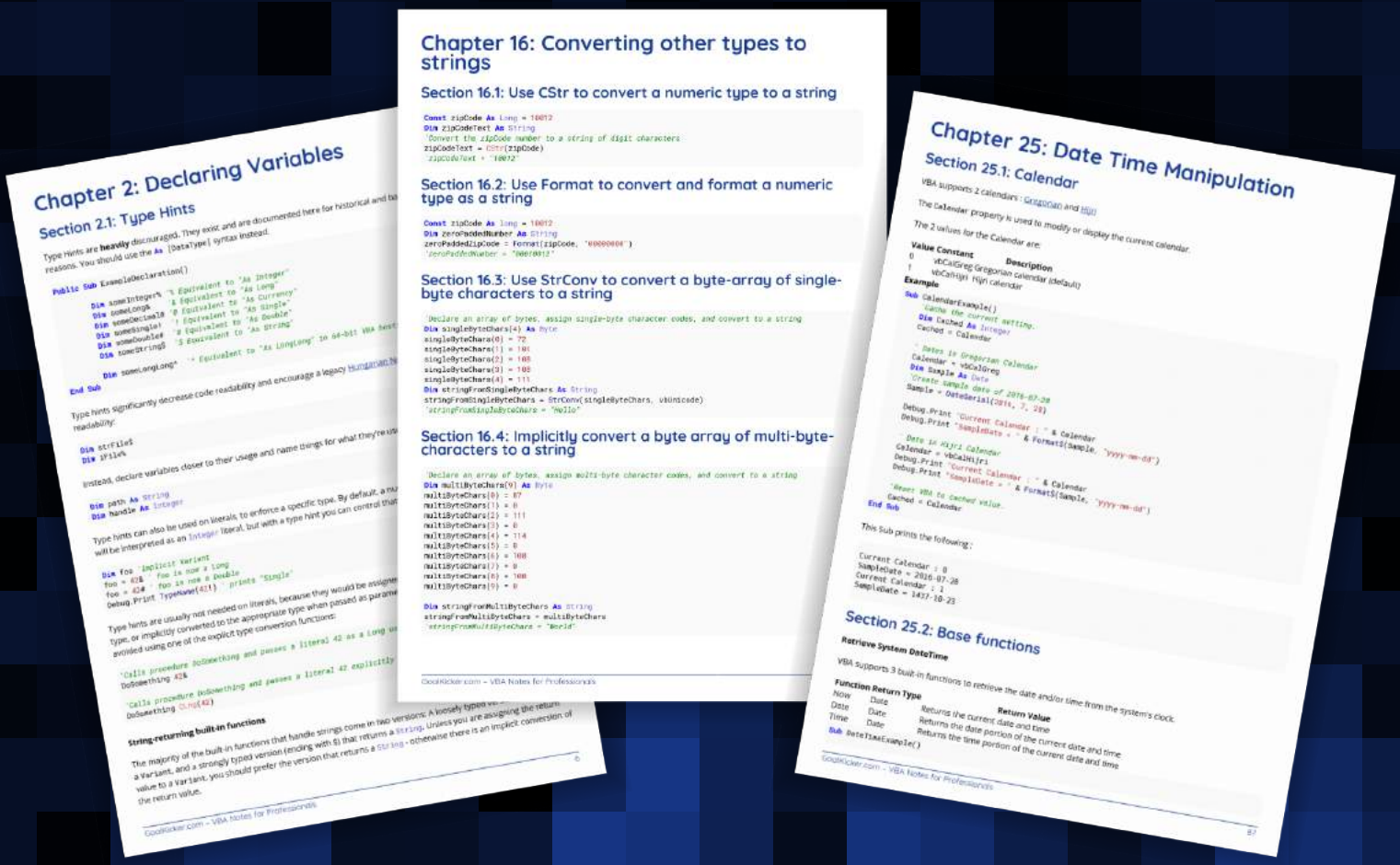
VBA

Notes for Professionals



100+ pages

of professional hints and tricks

Contents

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
http://GoalKicker.com/VBABook

Please send feedback and corrections to web@petercv.com

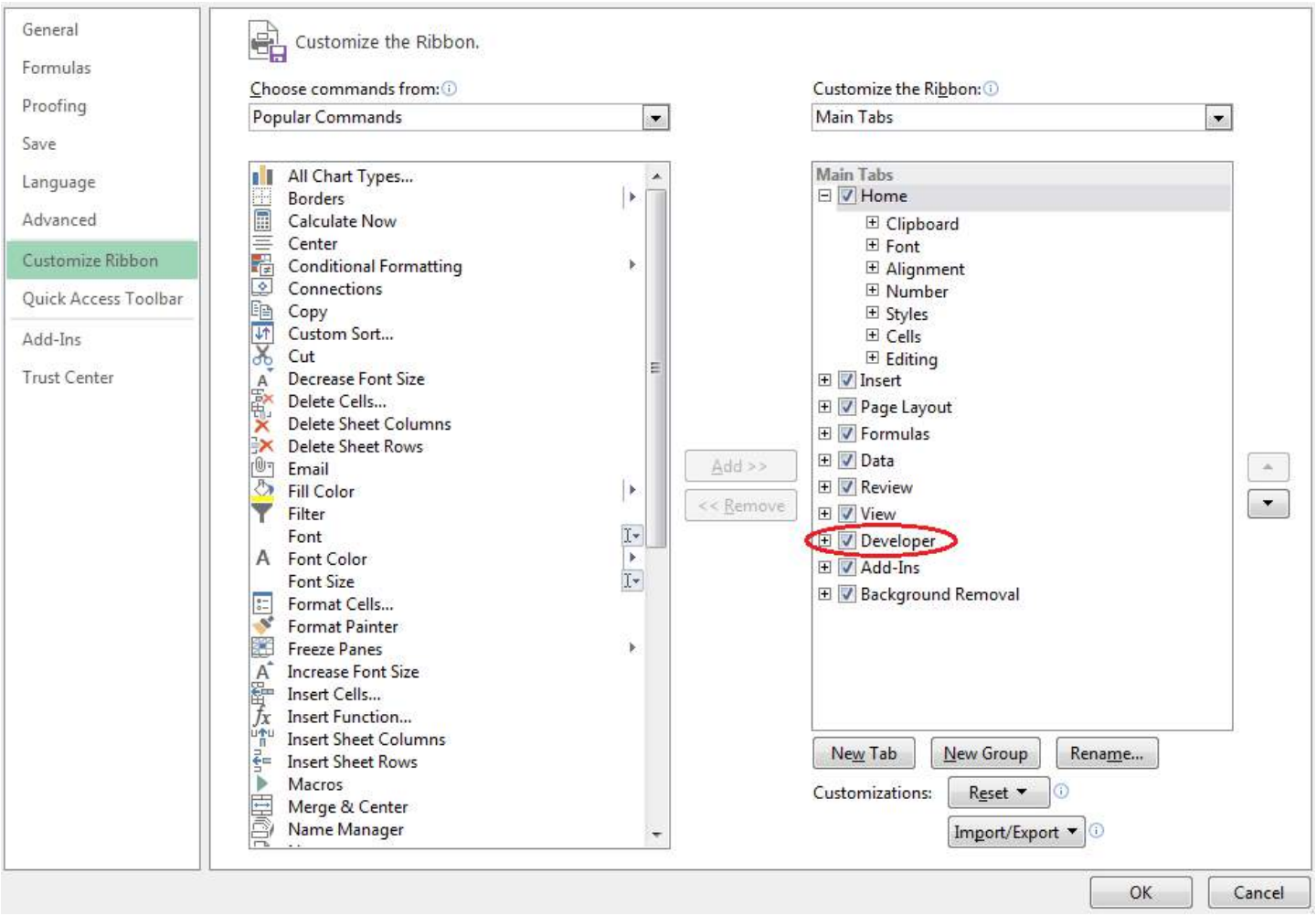
Chapter 1: Getting started with VBA

Version	Office Versions	Release Date Notes	Release Date
Vba6	? - 2007	[Sometime after][1]	1992-06-30
Vba7	2010 - 2016	[blog.techkit.com][2]	2010-04-15
VBA for Mac	2004, 2011 - 2016		2004-05-11

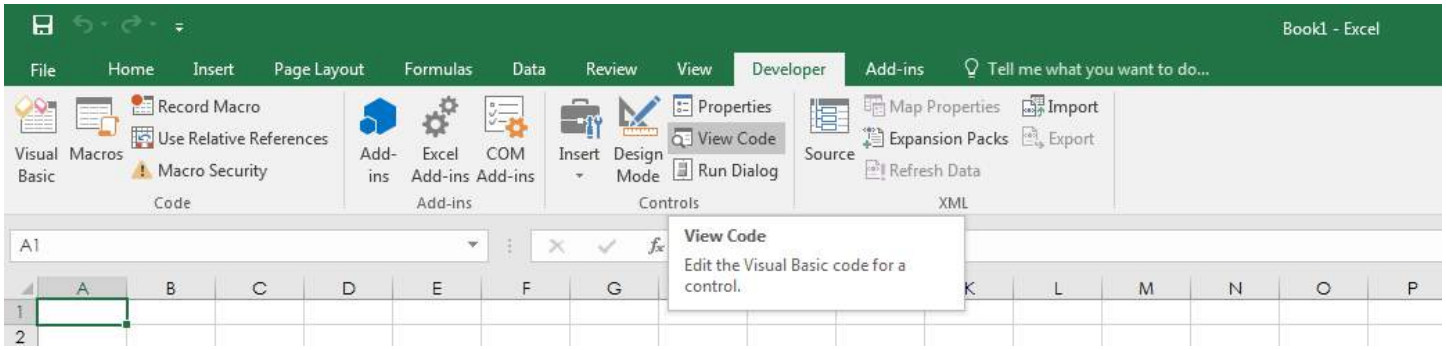
Section 1.1: Accessing the Visual Basic Editor in Microsoft Office

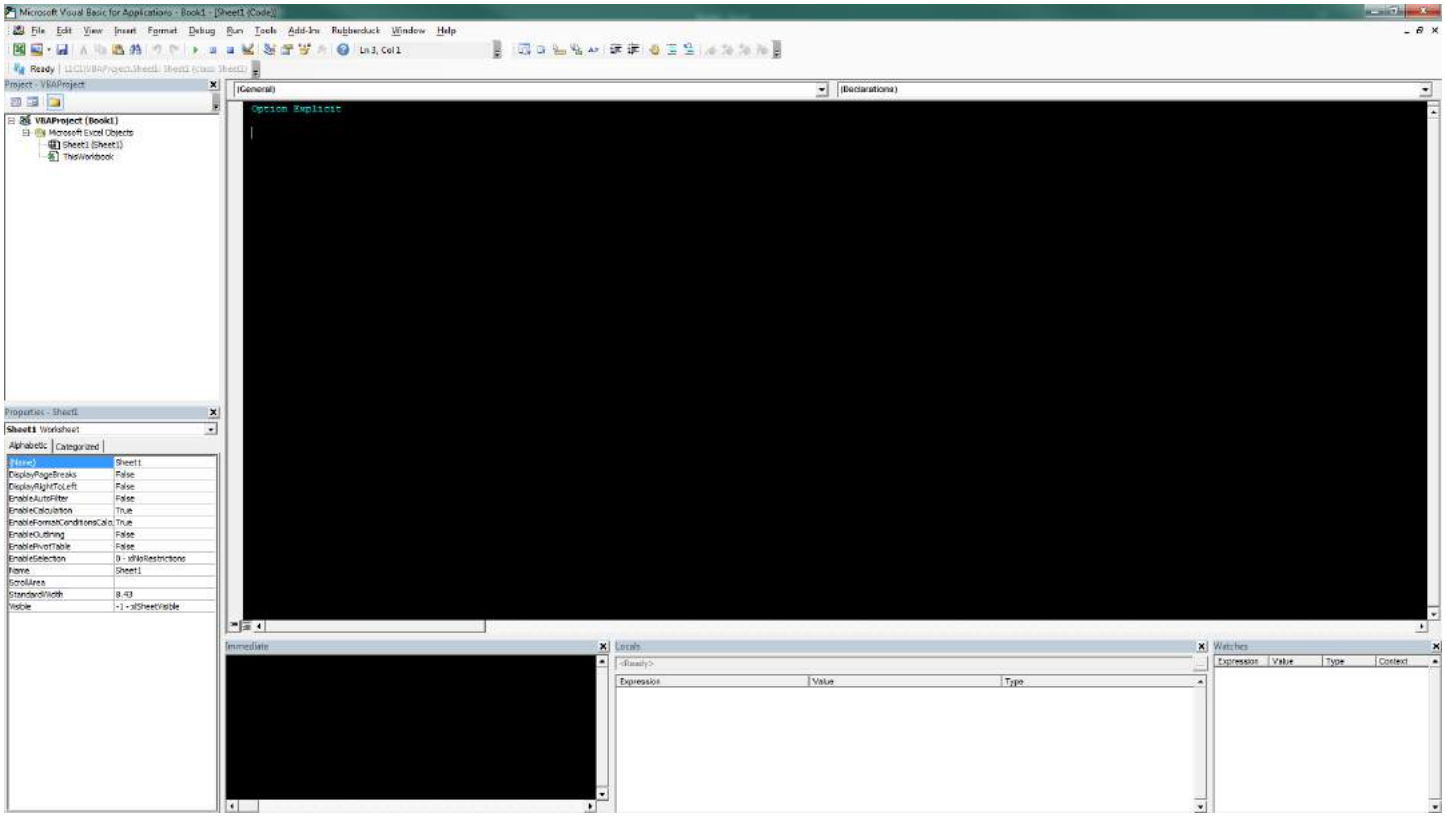
You can open the VB editor in any of the Microsoft Office applications by pressing `Alt` + `F11` or going to the Developer tab and clicking on the "Visual Basic" button. If you don't see the Developer tab in the Ribbon, check if this is enabled.

By default the Developer tab is disabled. To enable the Developer tab go to File -> Options, select Customize Ribbon in the list on the left. In the right "Customize the Ribbon" treeview find the Developer tree item and set the check for the Developer checkbox to checked. Click Ok to close the Options dialog.



The Developer tab is now visible in the Ribbon on which you can click on "Visual Basic" to open the Visual Basic Editor. Alternatively you can click on "View Code" to directly view the code pane of the currently active element, e.g. WorkSheet, Chart, Shape.





You can use VBA to automate almost any action that can be performed interactively (manually) and also provide functionality that is not available in Microsoft Office. VBA can create a document, add text to it, format it, edit it, and save it, all without human intervention.

Section 1.2: Debugging

Debugging is a very powerful way to have a closer look and fix incorrectly working (or non working) code.

Run code step by step

First thing you need to do during debugging is to stop the code at specific locations and then run it line by line to see whether that happens what's expected.

- Breakpoint (`F9` , Debug - Toggle breakpoint): You can add a breakpoint to any executed line (e.g. not to declarations), when execution reaches that point it stops, and gives control to user.
- You can also add the `Stop` keyword to a blank line to have the code stop at that location on runtime. This is useful if, for example, before declaration lines to which you can't add a breakpoint with `F9`
- Step into (`F8` , Debug - Step into): executes only one line of code, if that's a call of a user defined sub / function, then that's executed line by line.
- Step over (`Shift` + `F8` , Debug - Step over): executes one line of code, doesn't enter user defined subs / functions.
- Step out (`Ctrl` + `Shift` + `F8` , Debug - Step out): Exit current sub / function (run code until its end).

- Run to cursor (`Ctrl` + `F8`, Debug - Run to cursor): run code until reaching the line with the cursor.
- You can use `Debug.Print` to print lines to the Immediate Window at runtime. You may also use `Debug.?` as a shortcut for `Debug.Print`

Watches window

Running code line by line is only the first step, we need to know more details and one tool for that is the watch window (View - Watch window), here you can see values of defined expressions. To add a variable to the watch window, either:

- Right-click on it then select "Add watch".
- Right-click in watch window, select "Add watch".
- Go to Debug - Add watch.

When you add a new expression you can choose whether you just want to see it's value, or also break code execution when it's true or when its value changes.

Immediate Window

The immediate window allows you to execute arbitrary code or print items by preceeding them with either the `Print` keyword or a single question mark "`?`"

Some examples:

- `? ActiveSheet.Name` - returns name of the active sheet
- `Print ActiveSheet.Name` - returns the name of the active sheet
- `? foo` - returns the value of `foo`*
- `x = 10` sets `x` to 10*

* Getting/Setting values for variables via the Immediate Window can only be done during runtime

Debugging best practices

Whenever your code doesn't work as expected first thing you should do is to read it again carefully, looking for mistakes.

If that doesn't help, then start debugging it; for short procedures it can be efficient to just execute it line by line, for longer ones you probably need to set breakpoints or breaks on watched expressions, the goal here is to find the line not working as expected.

Once you have the line which gives the incorrect result, but the reason is not yet clear, try to simplify expressions, or replace variables with constants, that can help understanding whether variables' value are wrong.

If you still can't solve it, and ask for help:

- Include as small part of your code as possible for understanding of your problem
- If the problem is not related to the value of variables, then replace them by constants. (so, instead of `Sheets(a*b*c+d^2).Range(addressOfRange)` write `Sheets(4).Range("A2")`)
- Describe which line gives the wrong behaviour, and what it is (error, wrong result...)

Section 1.3: First Module and Hello World

To start coding in the first place, you have to right click your VBA Project in the left list and add a new Module. Your first *Hello-World* Code could look like this:

```
Sub HelloWorld()
    MsgBox "Hello, World!"
End Sub
```

To test it, hit the *Play*-Button in your Toolbar or simply hit the F5 key. Congratulations! You've built your first own VBA Module.

Chapter 2: Declaring Variables

Section 2.1: 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 2.2: 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 2.3: 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 2.4: 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 2.5: 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.

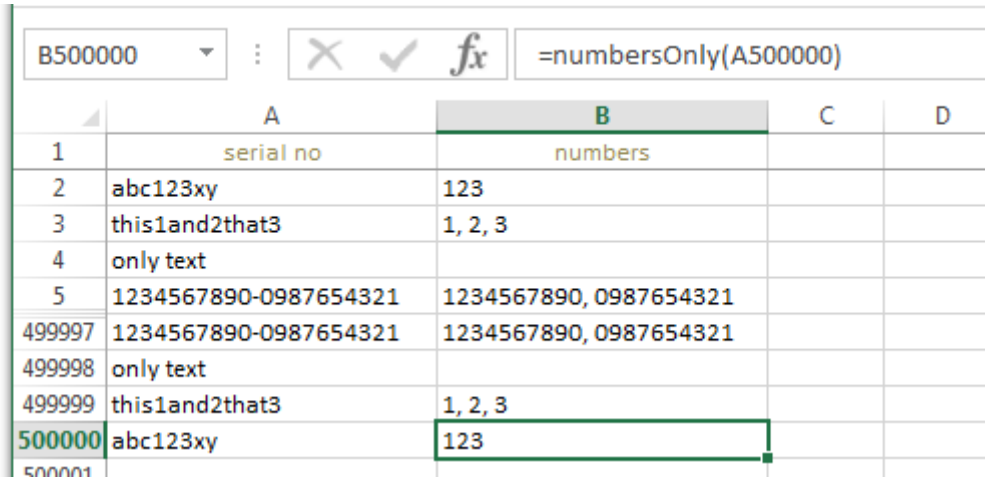
Snippet 1: Reuse a Scripting.Dictionary object across many worksheets

```
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 be 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
```



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 2.6: 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 2.7: 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.

Chapter 3: Scripting.FileSystemObject

Section 3.1: Retrieve only the path from a file path

The GetParentFolderName method returns the parent folder for any path. While this can also be used with folders, it is arguably more useful for extracting the path from an absolute file path:

```
Dim fso As New Scripting.FileSystemObject
Debug.Print fso.GetParentFolderName("C:\Users\Me\My Documents\SomeFile.txt")
```

Prints `C:\Users\Me\My Documents`

Note that the trailing path separator is not included in the returned string.

Section 3.2: Retrieve just the extension from a file name

```
Dim fso As New Scripting.FileSystemObject
Debug.Print fso.GetExtensionName("MyFile.something.txt")
```

Prints `txt` Note that the `GetExtensionName()` method already handles multiple periods in a file name.

Section 3.3: Recursively enumerate folders and files

Early Bound (with a reference to `Microsoft Scripting Runtime`)

```
Sub EnumerateFilesAndFolders( _
    FolderPath As String, _
    Optional MaxDepth As Long = -1, _
    Optional CurrentDepth As Long = 0, _
    Optional Indentation As Long = 2)

    Dim FSO As Scripting.FileSystemObject
    Set FSO = New Scripting.FileSystemObject

    'Check the folder exists
    If FSO.FolderExists(FolderPath) Then
        Dim fldr As Scripting.Folder
        Set fldr = FSO.GetFolder(FolderPath)

        'Output the starting directory path
        If CurrentDepth = 0 Then
          Debug.Print fldr.Path
        End If

        'Enumerate the subfolders
        Dim subFldr As Scripting.Folder
        For Each subFldr In fldr.SubFolders
            Debug.Print Space$((CurrentDepth + 1) * Indentation) & subFldr.Name
            If CurrentDepth < MaxDepth Or MaxDepth = -1 Then
                'Recursively call EnumerateFilesAndFolders
                EnumerateFilesAndFolders subFldr.Path, MaxDepth, CurrentDepth + 1, Indentation
            End If
        Next subFldr

        'Enumerate the files
        Dim fil As Scripting.File
        For Each fil In fldr.Files
```

```
            Debug.Print Space$((CurrentDepth + 1) * Indentation) & fil.Name
        Next fil
    End If
End Sub
```

Output when called with arguments like: `EnumerateFilesAndFolders "C:\Test"`

```
C:\Test
  Documents
    Personal
      Budget.xls
      Recipes.doc
    Work
      Planning.doc
  Downloads
    FooBar.exe
  ReadMe.txt
```

Output when called with arguments like: `EnumerateFilesAndFolders "C:\Test", 0`

```
C:\Test
  Documents
  Downloads
  ReadMe.txt
```

Output when called with arguments like: `EnumerateFilesAndFolders "C:\Test", 1, 4`

```
C:\Test
    Documents
        Personal
        Work
    Downloads
        FooBar.exe
    ReadMe.txt
```

Section 3.4: Strip file extension from a file name

```
Dim fso As New Scripting.FileSystemObject
Debug.Print fso.GetBaseName("MyFile.something.txt")
```

Prints `MyFile.something`

Note that the `GetBaseName()` method already handles multiple periods in a file name.

Section 3.5: Enumerate files in a directory using FileSystemObject

Early bound (requires a reference to Microsoft Scripting Runtime):

```
Public Sub EnumerateDirectory()
    Dim fso As Scripting.FileSystemObject
    Set fso = New Scripting.FileSystemObject

    Dim targetFolder As Folder
    Set targetFolder = fso.GetFolder("C:\")

    Dim foundFile As Variant
```

```
    For Each foundFile In targetFolder.Files
        Debug.Print foundFile.Name
    Next
End Sub
```

Late bound:

```
Public Sub EnumerateDirectory()
    Dim fso As Object
    Set fso = CreateObject("Scripting.FileSystemObject")

    Dim targetFolder As Object
    Set targetFolder = fso.GetFolder("C:\")

    Dim foundFile As Variant
    For Each foundFile In targetFolder.Files
        Debug.Print foundFile.Name
    Next
End Sub
```

Section 3.6: Creating a FileSystemObject

```
Const ForReading = 1
Const ForWriting = 2
Const ForAppending = 8

Sub FsoExample()
    Dim fso As Object ' declare variable
    Set fso = CreateObject("Scripting.FileSystemObject") ' Set it to be a File System Object

    ' now use it to check if a file exists
    Dim myFilePath As String
    myFilePath = "C:\mypath\to\myfile.txt"
    If fso.FileExists(myFilePath) Then
        ' do something
    Else
        ' file doesn't exist
        MsgBox "File doesn't exist"
    End If
End Sub
```

Section 3.7: Reading a text file using a FileSystemObject

```
Const ForReading = 1
Const ForWriting = 2
Const ForAppending = 8

Sub ReadTextFileExample()
    Dim fso As Object
    Set fso = CreateObject("Scripting.FileSystemObject")

    Dim sourceFile As Object
    Dim myFilePath As String
    Dim myFileText As String

    myFilePath = "C:\mypath\to\myfile.txt"
    Set sourceFile = fso.OpenTextFile(myFilePath, ForReading)
    myFileText = sourceFile.ReadAll ' myFileText now contains the content of the text file
    sourceFile.Close ' close the file
```

```
    ' do whatever you might need to do with the text

    ' You can also read it line by line
    Dim line As String
    Set sourceFile = fso.OpenTextFile(myFilePath, ForReading)
    While Not sourceFile.AtEndOfStream ' while we are not finished reading through the file
        line = sourceFile.ReadLine
        ' do something with the line...
    Wend
    sourceFile.Close
End Sub
```

Section 3.8: Creating a text file with FileSystemObject

```
Sub CreateTextFileExample()
    Dim fso As Object
    Set fso = CreateObject("Scripting.FileSystemObject")

    Dim targetFile As Object
    Dim myFilePath As String
    Dim myFileText As String

    myFilePath = "C:\mypath\to\myfile.txt"
    Set targetFile = fso.CreateTextFile(myFilePath, True)  ' this will overwrite any existing file
    targetFile.Write "This is some new text"
    targetFile.Write " And this text will appear right after the first bit of text."
    targetFile.WriteLine "This bit of text includes a newline character to ensure each write takes
its own line."
    targetFile.Close ' close the file
End Sub
```

Section 3.9: Using FSO.BuildPath to build a Full Path from folder path and file name

If you're accepting user input for folder paths, you might need to check for trailing backslashes (`\`) before building a file path. The `FSO.BuildPath` method makes this simpler:

```
  Const sourceFilePath As String = "C:\Temp"  '<-- Without trailing backslash
  Const targetFilePath As String = "C:\Temp\" '<-- With trailing backslash

  Const fileName As String = "Results.txt"

  Dim FSO As FileSystemObject
  Set FSO = New FileSystemObject

  Debug.Print FSO.BuildPath(sourceFilePath, fileName)
  Debug.Print FSO.BuildPath(targetFilePath, fileName)
```

Output:

```
C:\Temp\Results.txt
C:\Temp\Results.txt
```

Section 3.10: Writing to an existing file with FileSystemObject

```
Const ForReading = 1
Const ForWriting = 2
```

```
Const ForAppending = 8


Sub WriteTextFileExample()
    Dim oFso
    Set oFso = CreateObject("Scripting.FileSystemObject")

    Dim oFile as Object
    Dim myFilePath as String
    Dim myFileText as String

    myFilePath = "C:\mypath\to\myfile.txt"
    ' First check if the file exists
    If oFso.FileExists(myFilePath) Then
        ' this will overwrite any existing filecontent with whatever you send the file
        ' to append data to the end of an existing file, use ForAppending instead
        Set oFile = oFso.OpenTextFile(myFilePath, ForWriting)
    Else
        ' create the file instead
        Set oFile = oFso.CreateTextFile(myFilePath) ' skipping the optional boolean for overwrite if
exists as we already checked that the file doesn't exist.
    End If
    oFile.Write "This is some new text"
    oFile.Write " And this text will appear right after the first bit of text."
    oFile.WriteLine "This bit of text includes a newline character to ensure each write takes its
own line."
    oFile.Close ' close the file
End Sub
```

Chapter 4: Procedure Calls

Parameter	Info
IdentifierName	The name of the procedure to call.
arguments	A comma-separated list of arguments to be passed to the procedure.

Section 4.1: This is confusing. Why not just always use parentheses?

Parentheses are used to enclose the arguments of *function calls*. Using them for *procedure calls* can cause unexpected problems.

Because they can introduce bugs, both at run-time by passing a possibly unintended value to the procedure, and at compile-time by simply being invalid syntax.

Run-time

Redundant parentheses can introduce bugs. Given a procedure that takes an object reference as a parameter...

```
Sub DoSomething(ByRef target As Range)
End Sub
```

...and called with parentheses:

```
DoSomething (Application.ActiveCell) 'raises an error at runtime
```

This will raise an "Object Required" runtime error #424. Other errors are possible in other circumstances: here the `Application.ActiveCell` `Range` object reference is being *evaluated* and passed by value **regardless** of the procedure's signature specifying that `target` would be passed `ByRef`. The actual value passed `ByVal` to `DoSomething` in the above snippet, is `Application.ActiveCell.Value`.

Parentheses force VBA to evaluate the value of the bracketed expression, and pass the result `ByVal` to the called procedure. When the type of the evaluated result mismatches the procedure's expected type and cannot be implicitly converted, a runtime error is raised.

Compile-time

This code will fail to compile:

```
MsgBox ("Invalid Code!", vbCritical)
```

Because the expression `("Invalid Code!", vbCritical)` cannot be *evaluated* to a value.

This would compile and work:

```
MsgBox ("Invalid Code!"), (vbCritical)
```

But would definitely look silly. Avoid redundant parentheses.

Section 4.2: Implicit Call Syntax

```
ProcedureName
ProcedureName argument1, argument2
```

Call a procedure by its name without any parentheses.

Edge case

The `Call` keyword is only required in one edge case:

```
Call DoSomething : DoSomethingElse
```

`DoSomething` and `DoSomethingElse` are procedures being called. If the `Call` keyword was removed, then `DoSomething` would be parsed as a *line label* rather than a procedure call, which would break the code:

```
DoSomething: DoSomethingElse 'only DoSomethingElse will run
```

Section 4.3: Optional Arguments

Some procedures have optional arguments. Optional arguments always come after required arguments, but the procedure can be called without them.

For example, if the function, `ProcedureName` were to have two required arguments (`argument1`, `argument2`), and one optional argument, `optArgument3`, it could be called at least four ways:

```
' Without optional argument
result = ProcedureName("A", "B")

' With optional argument
result = ProcedureName("A", "B", "C")

' Using named arguments (allows a different order)
result = ProcedureName(optArgument3:="C", argument1:="A", argument2:="B")

' Mixing named and unnamed arguments
result = ProcedureName("A", "B", optArgument3:="C")
```

The structure of the function header being called here would look something like this:

```
Function ProcedureName(argument1 As String, argument2 As String, Optional optArgument3 As String)
As String
```

The `Optional` keyword indicates that this argument can be omitted. As mentioned before - any optional arguments introduced in the header **must** appear at the end, after any required arguments.

You can also provide a *default* value for the argument in the case that a value isn't passed to the function:

```
Function ProcedureName(argument1 As String, argument2 As String, Optional optArgument3 As String =
"C") As String
```

In this function, if the argument for `c` isn't supplied it's value will default to `"C"`. If a value *is* supplied then this will override the default value.

Section 4.4: Explicit Call Syntax

```
Call ProcedureName
Call ProcedureName(argument1, argument2)
```

The explicit call syntax requires the `Call` keyword and parentheses around the argument list; parentheses are redundant if there are no parameters. This syntax was made obsolete when the more modern implicit call syntax was added to VB.

Section 4.5: Return Values

To retrieve the result of a procedure call (e.g. `Function` or `Property Get` procedures), put the call on the right-hand side of an assignment:

```
result = ProcedureName
result = ProcedureName(argument1, argument2)
```

Parentheses must be present if there are parameters. If the procedure has no parameters, the parentheses are redundant.

Chapter 5: Naming Conventions

Section 5.1: Variable Names

Variables hold data. Name them after what they're used for, **not after their data type** or scope, using a **noun**. If you feel compelled to *number* your variables (e.g. `thing1, thing2, thing3`), then consider using an appropriate data structure instead (e.g. an array, a `Collection`, or a `Dictionary`).

Names of variables that represent an iteratable *set* of values - e.g. an array, a `Collection`, a `Dictionary`, or a `Range` of cells, should be plural.

Some common VBA naming conventions go thus:

For procedure-level Variables:

`camelCase`

```
Public Sub ExampleNaming(ByVal inputValue As Long, ByRef inputVariable As Long)

    Dim procedureVariable As Long
    Dim someOtherVariable As String

End Sub
```

For module-level Variables:

`PascalCase`

```
Public GlobalVariable As Long
Private ModuleVariable As String
```

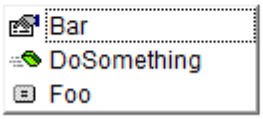
For Constants:

`SHOUTY_SNAKE_CASE` is commonly used to differentiate constants from variables:

```
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 SomeProcedure()

    Const PROCEDURE_CONSTANT As Long = 10

End Sub
```

However `PascalCase` names make cleaner-looking code and are just as good, given IntelliSense uses different icons for variables and constants:

```
Option Explicit
Public Const Foo As String = "foo"
Public Bar As String

Sub DoSomething()
    Module1.
End Sub
```



Hungarian Notation

> Name them after what they're used for, **not after their data type** or scope.

"Hungarian Notation makes it easier to see what the type of a variable is"

If you write your code such as procedures adhere to the *Single Responsibility Principle* (as it should), you should never be looking at a screenful of variable declarations at the top of any procedure; declare variables as close as possible to their first usage, and their data type will always be in plain sight if you declare them with an explicit type. The VBE's `Ctrl`+`i` shortcut can be used to display a variable's type in a tooltip, too.

What a variable is used for is much more useful information than its data type, *especially* in a language such as VBA which happily and implicitly converts a type into another as needed.

Consider `iFile` and `strFile` in this example:

```
Function bReadFile(ByVal strFile As String, ByRef strData As String) As Boolean
    Dim bRetVal As Boolean
    Dim iFile As Integer

    On Error GoTo CleanFail

    iFile = FreeFile
    Open strFile For Input As #iFile
    Input #iFile, strData

    bRetVal = True

CleanExit:
    Close #iFile
    bReadFile = bRetVal
    Exit Function
CleanFail:
    bRetVal = False
    Resume CleanExit
End Function
```

Compare to:

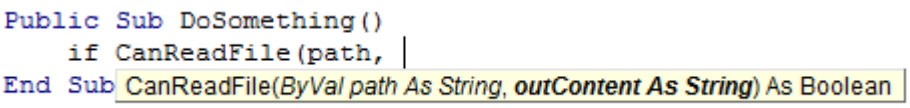
```
Function CanReadFile(ByVal path As String, ByRef outContent As String) As Boolean
    On Error GoTo CleanFail

    Dim handle As Integer
    handle = FreeFile
```

```
    Open path For Input As #handle
    Input #handle, outContent

    Dim result As Boolean
    result = True

CleanExit:
    Close #handle
    CanReadFile = result
    Exit Function
CleanFail:
    result = False
    Resume CleanExit
End Function
```

`strData` is passed **`ByRef`** in the top example, but beside the fact that we're lucky enough to see that it's *explicitly* passed as such, there's no indication that `strData` is actually *returned* by the function.

The bottom example names it `outContent`; this **`out`** prefix is what Hungarian Notation was invented for: to help clarify *what a variable is used for*, in this case to clearly identify it as an "out" parameter.

This is useful, because IntelliSense by itself doesn't display **`ByRef`**, even when the parameter is *explicitly* passed by reference:



Which leads to...

Hungarian Done Right

Hungarian Notation originally didn't have anything to do with variable types. In fact, Hungarian Notation *done right* is actually useful. Consider this small example (**`ByVal`** and **`As Integer`** removed for brevety):

```
Public Sub Copy(iX1, iY1, iX2, iY2)
End Sub
```

Compare to:

```
Public Sub Copy(srcColumn, srcRow, dstColumn, dstRow)
End Sub
```

`src` and `dst` are *Hungarian Notation* prefixes here, and they convey *useful* information that cannot otherwise already be inferred from the parameter names or IntelliSense showing us the declared type.

Of course there's a better way to convey it all, using proper *abstraction* and real words that can be pronounced out loud and make sense - as a contrived example:

```
Type Coordinate
    RowIndex As Long
    ColumnIndex As Long
End Type

Sub Copy(source As Coordinate, destination As Coordinate)
End Sub
```

Section 5.2: Procedure Names

Procedures *do something*. Name them after what they're doing, using a **verb**. If accurately naming a procedure is not possible, likely the procedure is *doing too many things* and needs to be broken down into smaller, more specialized procedures.

Some common VBA naming conventions go thus:

For all Procedures:

`PascalCase`

```
Public Sub DoThing()

End Sub

Private Function ReturnSomeValue() As [DataType]

End Function
```

For event handler procedures:

`ObjectName_EventName`

```
Public Sub Workbook_Open()

End Sub

Public Sub Button1_Click()

End Sub
```

Event handlers are usually automatically named by the VBE; renaming them without renaming the object and/or the handled event will break the code - the code will run and compile, but the handler procedure will be orphaned and will never be executed.

Boolean Members

Consider a Boolean-returning function:

```
Function bReadFile(ByVal strFile As String, ByRef strData As String) As Boolean
End Function
```

Compare to:

```
Function CanReadFile(ByVal path As String, ByRef outContent As String) As Boolean
End Function
```

The `Can` prefix *does* serve the same purpose as the `b` prefix: it identifies the function's return value as a `Boolean`. But `Can` reads better than `b`:

```
If CanReadFile(path, content) Then
```

Compared to:

```
If bReadFile(strFile, strData) Then
```

Consider using prefixes such as `Can`, `Is` or `Has` in front of Boolean-returning members (functions and properties), but only when it adds value. This conforms with the current Microsoft naming guidelines.

Chapter 6: Creating a procedure

Section 6.1: Introduction to procedures

A `Sub` is a procedure that performs a specific task but does not return a specific value.

```
Sub ProcedureName ([argument_list])
    [statements]
End Sub
```

If no access modifier is specified, a procedure is `Public` by default.

A `Function` is a procedure that is given data and returns a value, ideally without global or module-scope side-effects.

```
Function ProcedureName ([argument_list]) [As ReturnType]
    [statements]
End Function
```

A `Property` is a procedure that *encapsulates* module data. A property can have up to 3 accessors: `Get` to return a value or object reference, `Let` to assign a value, and/or `Set` to assign an object reference.

```
Property Get|Let|Set PropertyName([argument_list]) [As ReturnType]
    [statements]
End Property
```

Properties are usually used in class modules (although they are allowed in standard modules as well), exposing accessor to data that is otherwise inaccessible to the calling code. A property that only exposes a `Get` accessor is "read-only"; a property that would only expose a `Let` and/or `Set` accessor is "write-only". Write-only properties are not considered a good programming practice - if the client code can *write* a value, it should be able to *read* it back. Consider implementing a `Sub` procedure instead of making a write-only property.

Returning a value

A `Function` or `Property Get` procedure can (and should!) return a value to its caller. This is done by assigning the identifier of the procedure:

```
Property Get Foo() As Integer
    Foo = 42
End Property
```

Section 6.2: Function With Examples

As stated above Functions are smaller procedures that contain small pieces of code which may be repetitive inside a Procedure.

Functions are used to reduce redundancy in code.

Similar to a Procedure, A function can be declared with or without an arguments list.

Function is declared as a return type, as all functions return a value. The Name and the Return Variable of a function are the Same.

1. Function With Parameter:

```
Function check_even(i as integer) as boolean
if (i mod  2) = 0 then
check_even = True
else
check_even=False
end if
end Function
```

2. Function Without Parameter:

```
Function greet() as String
greet= "Hello Coder!"
end Function
```

The Function can be called in various ways inside a function. Since a Function declared with a return type is basically a variable. it is used similar to a variable.

Functional Calls:

```
call greet() 'Similar to a Procedural call just allows the Procedure to use the
             'variable greet
string_1=greet() 'The Return value of the function is used for variable
                 'assignment
```

Further the function can also be used as conditions for if and other conditional statements.

```
for i = 1 to 10
if check_even(i) then
msgbox i & " is Even"
else
msgbox i & " is Odd"
end if
next i
```

Further more Functions can have modifiers such as By ref and By val for their arguments.

Chapter 7: Flow control structures

Section 7.1: For loop

The `For` loop is used to repeat the enclosed section of code a given number of times. The following simple example illustrates the basic syntax:

```
Dim i as Integer            'Declaration of i
For i = 1 to 10             'Declare how many times the loop shall be executed
    Debug.Print i           'The piece of code which is repeated
Next i                      'The end of the loop
```

The code above declares an Integer `i`. The `For` loop assigns every value between 1 and 10 to `i` and then executes `Debug.Print i` - i.e. the code prints the numbers 1 through 10 to the immediate window. Note that the loop variable is incremented by the `Next` statement, that is after the enclosed code executes as opposed to before it executes.

By default, the counter will be incremented by 1 each time the loop executes. However, a `Step` can be specified to change the amount of the increment as either a literal or the return value of a function. If the starting value, ending value, or `Step` value is a floating point number, it will be rounded to the nearest integer value. `Step` can be either a positive or negative value.

```
Dim i As Integer
For i = 1 To 10 Step 2
    Debug.Print i       'Prints 1, 3, 5, 7, and 9
Next
```

In general a `For` loop would be used in situations where it is known before the loop starts how many times to execute the enclosed code (otherwise a `Do` or `While` loop may be more appropriate). This is because the exit condition is fixed after the first entry into loop, as this code demonstrates:

```
Private Iterations As Long              'Module scope

Public Sub Example()
    Dim i As Long
    Iterations = 10
    For i = 1 To Iterations
        Debug.Print Iterations      'Prints 10 through 1, descending.
        Iterations = Iterations - 1
    Next
End Sub
```

A `For` loop can be exited early with the `Exit For` statement:

```
Dim i As Integer

For i = 1 To 10
    If i > 5 Then
        Exit For
    End If
    Debug.Print i       'Prints 1, 2, 3, 4, 5 before loop exits early.
Next
```

Section 7.2: Select Case

`SELECT CASE` can be used when many different conditions are possible. The conditions are checked from top to bottom and only the first case that match will be executed.

```
Sub TestCase()
    Dim MyVar As String

    Select Case MyVar    'We Select the Variable MyVar to Work with
        Case "Hello"     'Now we simply check the cases we want to check
            MsgBox "This Case"
        Case "World"
            MsgBox "Important"
        Case "How"
            MsgBox "Stuff"
        Case "Are"
            MsgBox "I'm running out of ideas"
        Case "You?", "Today"  'You can separate several conditions with a comma
            MsgBox "Uuuhm..." 'if any is matched it will go into the case
        Case Else             'If none of the other cases is hit
            MsgBox "All of the other cases failed"
    End Select

    Dim i As Integer
    Select Case i
        Case Is > 2 '"Is" can be used instead of the variable in conditions.
            MsgBox "i is greater than 2"
        'Case 2 < Is '"Is" can only be used at the beginning of the condition.
        'Case Else is optional
    End Select
End Sub
```

The logic of the `SELECT CASE` block can be inverted to support testing of different variables too, in this kind of scenario we can also use logical operators:

```
Dim x As Integer
Dim y As Integer

x = 2
y = 5

Select Case True
    Case x > 3
        MsgBox "x is greater than 3"
    Case y < 2
        MsgBox "y is less than 2"
    Case x = 1
        MsgBox "x is equal to 1"
    Case x = 2 Xor y = 3
        MsgBox "Go read about ""Xor"""
    Case Not y = 5
        MsgBox "y is not 5"
    Case x = 3 Or x = 10
        MsgBox "x = 3 or 10"
    Case y < 10 And x < 10
        MsgBox "x and y are less than 10"
    Case Else
        MsgBox "No match found"
End Select
```

Case statements can also use arithmetic operators. Where an arithmetic operator is being used against the `SELECT CASE` value it should be preceded with the `Is` keyword:

```
Dim x As Integer

x = 5

Select Case x
    Case 1
        MsgBox "x equals 1"
    Case 2, 3, 4
        MsgBox "x is 2, 3 or 4"
    Case 7 To 10
        MsgBox "x is between 7 and 10 (inclusive)"
    Case Is < 2
        MsgBox "x is less than one"
    Case Is >= 7
        MsgBox "x is greater than or equal to 7"
    Case Else
        MsgBox "no match found"
End Select
```

Section 7.3: For Each loop

The `For Each` loop construct is ideal for iterating all elements of a collection.

```
Public Sub IterateCollection(ByVal items As Collection)

    'For Each iterator must always be variant
    Dim element As Variant

    For Each element In items
        'assumes element can be converted to a string
        Debug.Print element
    Next

End Sub
```

Use `For Each` when iterating object collections:

```
Dim sheet As Worksheet
For Each sheet In ActiveWorkbook.Worksheets
    Debug.Print sheet.Name
Next
```

Avoid `For Each` when iterating arrays; a `For` loop will offer significantly better performance with arrays. Conversely, a `For Each` loop will offer better performance when iterating a `Collection`.

Syntax

```
For Each [item] In [collection]
    [statements]
Next [item]
```

The `Next` keyword may optionally be followed by the iterator variable; this can help clarify nested loops, although there are better ways to clarify nested code, such as extracting the inner loop into its own procedure.

```
Dim book As Workbook
For Each book In Application.Workbooks
```

```
    Debug.Print book.FullName

    Dim sheet As Worksheet
    For Each sheet In ActiveWorkbook.Worksheets
        Debug.Print sheet.Name
    Next sheet
Next book
```

Section 7.4: Do loop

```
 Public Sub DoLoop()
    Dim entry As String
    entry = ""
    'Equivalent to a While loop will ask for strings until "Stop" in given
    'Prefer using a While loop instead of this form of Do loop
    Do While entry <> "Stop"
        entry = InputBox("Enter a string, Stop to end")
        Debug.Print entry
    Loop

    'Equivalent to the above loop, but the condition is only checked AFTER the
    'first iteration of the loop, so it will execute even at least once even
    'if entry is equal to "Stop" before entering the loop (like in this case)
    Do
        entry = InputBox("Enter a string, Stop to end")
        Debug.Print entry
    Loop While entry <> "Stop"


    'Equivalent to writing Do While Not entry="Stop"
    '
    'Because the Until is at the top of the loop, it will
    'not execute because entry is still equal to "Stop"
    'when evaluating the condition
    Do Until entry = "Stop"
        entry = InputBox("Enter a string, Stop to end")
        Debug.Print entry
    Loop

    'Equivalent to writing Do ... Loop While Not i >= 100
    Do
        entry = InputBox("Enter a string, Stop to end")
        Debug.Print entry
    Loop Until entry = "Stop"
End Sub
```

Section 7.5: While loop

```
'Will return whether an element is present in the array
Public Function IsInArray(values() As String, ByVal whatToFind As String) As Boolean
    Dim i As Integer
    i = 0

    While i < UBound(values) And values(i) <> whatToFind
        i = i + 1
    Wend

    IsInArray = values(i) = whatToFind
End Function
```

Chapter 8: Comments

Section 8.1: Apostrophe Comments

A comment is marked by an apostrophe (`'`), and ignored when the code executes. Comments help explain your code to future readers, including yourself.

Since all lines starting with a comment are ignored, they can also be used to prevent code from executing (while you debug or refactor). Placing an apostrophe `'` before your code turns it into a comment. (This is called *commenting out* the line.)

```
Sub InlineDocumentation()
  'Comments start with an "'"

  'They can be place before a line of code, which prevents the line from executing
  'Debug.Print "Hello World"

  'They can also be placed after a statement
  'The statement still executes, until the compiler arrives at the comment
  Debug.Print "Hello World"  'Prints a welcome message

'Comments can have 0 indention....
     '... or as much as needed

  '''' Comments can contain multiple apostrophes ''''

  'Comments can span lines (using line continuations) _
    but this can make for hard to read code

  'If you need to have mult-line comments, it is often easier to
  'use an apostrophe on each line

  'The continued statement syntax (:) is treated as part of the comment, so
  'it is not possible to place an executable statement after a comment
  'This won't run : Debug.Print "Hello World"
End Sub

'Comments can appear inside or outside a procedure
```

Section 8.2: REM Comments

```
Sub RemComments()
  Rem Comments start with "Rem" (VBA will change any alternate casing to "Rem")
  Rem is an abbreviation of Remark, and similar to DOS syntax
  Rem Is a legacy approach to adding comments, and apostrophes should be preferred

  Rem Comments CANNOT appear after a statement, use the apostrophe syntax instead
  Rem Unless they are preceded by the instruction separator token
  Debug.Print "Hello World": Rem prints a welcome message
  Debug.Print "Hello World" 'Prints a welcome message

  'Rem cannot be immediately followed by the following characters "!,@,#,$,%,&"
  'Whereas the apostrophe syntax can be followed by any printable character.

End Sub

Rem Comments can appear inside or outside a procedure
```

Chapter 9: Arrays

Section 9.1: Multidimensional Arrays

Multidimensional Arrays

As the name indicates, multi dimensional arrays are arrays that contain more than one dimension, usually two or three but it can have up to 32 dimensions.

A multi array works like a matrix with various levels, take in example a comparison between one, two, and three Dimensions.

One Dimension is your typical array, it looks like a list of elements.

```
Dim 1D(3) as Variant

*1D - Visually*
(0)
(1)
(2)
```

Two Dimensions would look like a Sudoku Grid or an Excel sheet, when initializing the array you would define how many rows and columns the array would have.

```
Dim 2D(3,3) as Variant
'this would result in a 3x3 grid

*2D - Visually*
(0,0) (0,1) (0,2)
(1,0) (1,1) (1,2)
(2,0) (2,1) (2,2)
```

Three Dimensions would start to look like Rubik's Cube, when initializing the array you would define rows and columns and layers/depths the array would have.

```
Dim 3D(3,3,2) as Variant
'this would result in a 3x3x3 grid

*3D - Visually*
       1st layer                2nd layer                  3rd layer
         front                    middle                     back
(0,0,0) (0,0,1) (0,0,2) ¦ (1,0,0) (1,0,1) (1,0,2) ¦ (2,0,0) (2,0,1) (2,0,2)
(0,1,0) (0,1,1) (0,1,2) ¦ (1,1,0) (1,1,1) (1,1,2) ¦ (2,1,0) (2,1,1) (2,1,2)
(0,2,0) (0,2,1) (0,2,2) ¦ (1,2,0) (1,2,1) (1,2,2) ¦ (2,2,0) (2,2,1) (2,2,2)
```

Further dimensions could be thought as the multiplication of the 3D, so a 4D(1,3,3,3) would be two side-by-side 3D arrays.

Two-Dimension Array

Creating

The example below will be a compilation of a list of employees, each employee will have a set of information on the list (First Name, Surname, Address, Email, Phone ...), the example will essentially be storing on the array

(employee,information) being the (0,0) is the first employee's first name.

```
Dim Bosses As Variant
'set bosses as Variant, so we can input any data type we want

Bosses = [{"Jonh","Snow","President";"Ygritte","Wild","Vice-President"}]
'initialise a 2D array directly by filling it with information, the redult wil be a array(1,2) size
2x3 = 6 elements

Dim Employees As Variant
'initialize your Employees array as variant
'initialize and ReDim the Employee array so it is a dynamic array instead of a static one, hence
treated differently by the VBA Compiler
ReDim Employees(100, 5)
'declaring an 2D array that can store 100 employees with 6 elements of information each, but starts
empty
'the array size is 101 x 6 and contains 606 elements

For employee = 0 To UBound(Employees, 1)
'for each employee/row in the array, UBound for 2D arrays, which will get the last element on the
array
'needs two parameters 1st the array you which to check and 2nd the dimension, in this case 1 =
employee and 2 = information
    For information_e = 0 To UBound(Employees, 2)
    'for each information element/column in the array

        Employees(employee, information_e) = InformationNeeded ' InformationNeeded would be the
data to fill the array
        'iterating the full array will allow for direct attribution of information into the element
coordinates
    Next
Next
```

Resizing

Resizing or `ReDim Preserve` a Multi-Array like the norm for a One-Dimension array would get an error, instead the information needs to be transferred into a Temporary array with the same size as the original plus the number of row/columns to add. In the example below we'll see how to initialize a Temp Array, transfer the information over from the original array, fill the remaining empty elements, and replace the temp array by the original array.

```
Dim TempEmp As Variant
'initialise your temp array as variant
ReDim TempEmp(UBound(Employees, 1) + 1, UBound(Employees, 2))
'ReDim/Resize Temp array as a 2D array with size UBound(Employees)+1 = (last element in Employees 1st
dimension) + 1,
'the 2nd dimension remains the same as the original array. we effectively add 1 row in the Employee
array

'transfer
For emp = LBound(Employees, 1) To UBound(Employees, 1)
    For info = LBound(Employees, 2) To UBound(Employees, 2)
        'to transfer Employees into TempEmp we iterate both arrays and fill TempEmp with the
corresponding element value in Employees
        TempEmp(emp, info) = Employees(emp, info)

    Next
Next

'fill remaining
'after the transfers the Temp array still has unused elements at the end, being that it was increased
```

```
'to fill the remaining elements iterate from the last "row" with values to the last row in the array
'in this case the last row in Temp will be the size of the Employees array rows + 1, as the last row
of Employees array is already filled in the TempArray

For emp = UBound(Employees, 1) + 1 To UBound(TempEmp, 1)
    For info = LBound(TempEmp, 2) To UBound(TempEmp, 2)

        TempEmp(emp, info) = InformationNeeded & "NewRow"

    Next
Next

'erase Employees, attribute Temp array to Employees and erase Temp array
Erase Employees
Employees = TempEmp
Erase TempEmp
```

Changing Element Values

To change/alter the values in a certain element can be done by simply calling the coordinate to change and giving it a new value: `Employees(0, 0) = "NewValue"`

Alternatively iterate through the coordinates use conditions to match values corresponding to the parameters needed:

```
For emp = 0 To UBound(Employees)
    If Employees(emp, 0) = "Gloria" And Employees(emp, 1) = "Stephan" Then
    'if value found
        Employees(emp, 1) = "Married, Last Name Change"
        Exit For
        'don't iterate through a full array unless necessary
    End If
Next
```

Reading

Accessing the elements in the array can be done with a Nested Loop (iterating every element), Loop and Coordinate (iterate Rows and accessing columns directly), or accessing directly with both coordinates.

```
'nested loop, will iterate through all elements
For emp = LBound(Employees, 1) To UBound(Employees, 1)
    For info = LBound(Employees, 2) To UBound(Employees, 2)
        Debug.Print Employees(emp, info)
    Next
Next

'loop and coordinate, iteration through all rows and in each row accessing all columns directly
For emp = LBound(Employees, 1) To UBound(Employees, 1)
    Debug.Print Employees(emp, 0)
    Debug.Print Employees(emp, 1)
    Debug.Print Employees(emp, 2)
    Debug.Print Employees(emp, 3)
    Debug.Print Employees(emp, 4)
    Debug.Print Employees(emp, 5)
Next

'directly accessing element with coordinates
```

```
Debug.Print Employees(5, 5)
```

***Remember**, it's always handy to keep an array map when using Multidimensional arrays, they can easily become confusion.*

Three-Dimension Array

For the 3D array, we'll use the same premise as the 2D array, with the addition of not only storing the Employee and Information but as well Building they work in.

The 3D array will have the Employees (can be thought of as Rows), the Information (Columns), and Building that can be thought of as different sheets on an excel document, they have the same size between them, but every sheets has a different set of information in its cells/elements. The 3D array will contain ***n*** number of 2D arrays.

Creating

A 3D array needs 3 coordinates to be initialized `Dim 3Darray(2,5,5) As Variant` the first coordinate on the array will be the number of Building/Sheets (different sets of rows and columns), second coordinate will define Rows and third Columns. The `Dim` above will result in a 3D array with 108 elements (`3*6*6`), effectively having 3 different sets of 2D arrays.

```
Dim ThreeDArray As Variant
'initialise your ThreeDArray array as variant
ReDim ThreeDArray(1, 50, 5)
'declaring an 3D array that can store two sets of 51 employees with 6 elements of information each,
but starts empty
'the array size is 2 x 51 x 6 and contains 612 elements

For building = 0 To UBound(ThreeDArray, 1)
    'for each building/set in the array
    For employee = 0 To UBound(ThreeDArray, 2)
    'for each employee/row in the array
        For information_e = 0 To UBound(ThreeDArray, 3)
        'for each information element/column in the array

            ThreeDArray(building, employee, information_e) = InformationNeeded ' InformationNeeded
would be the data to fill the array
        'iterating the full array will allow for direct attribution of information into the element
coordinates
        Next
    Next
Next
```

Resizing

Resizing a 3D array is similar to resizing a 2D, first create a Temporary array with the same size of the original adding one in the coordinate of the parameter to increase, the first coordinate will increase the number of sets in the array, the second and third coordinates will increase the number of Rows or Columns in each set.

The example below increases the number of Rows in each set by one, and fills those recently added elements with new information.

```
Dim TempEmp As Variant
'initialise your temp array as variant
ReDim TempEmp(UBound(ThreeDArray, 1), UBound(ThreeDArray, 2) + 1, UBound(ThreeDArray, 3))
```

```
'ReDim/Resize Temp array as a 3D array with size UBound(ThreeDArray)+1 = (last element in Employees
2nd dimension) + 1,
'the other dimension remains the same as the original array. we effectively add 1 row in the for each
set of the 3D array

'transfer
For building = LBound(ThreeDArray, 1) To UBound(ThreeDArray, 1)
    For emp = LBound(ThreeDArray, 2) To UBound(ThreeDArray, 2)
        For info = LBound(ThreeDArray, 3) To UBound(ThreeDArray, 3)
            'to transfer ThreeDArray into TempEmp by iterating all sets in the 3D array and fill
TempEmp with the corresponding element value in each set of each row
            TempEmp(building, emp, info) = ThreeDArray(building, emp, info)

        Next
    Next
Next

'fill remaining
'to fill the remaining elements we need to iterate from the last "row" with values to the last row in
the array in each set, remember that the first empty element is the original array Ubound() plus 1
For building = LBound(TempEmp, 1) To UBound(TempEmp, 1)
    For emp = UBound(ThreeDArray, 2) + 1 To UBound(TempEmp, 2)
        For info = LBound(TempEmp, 3) To UBound(TempEmp, 3)

            TempEmp(building, emp, info) = InformationNeeded & "NewRow"

        Next
    Next
Next

'erase Employees, attribute Temp array to Employees and erase Temp array
Erase ThreeDArray
ThreeDArray = TempEmp
Erase TempEmp
```

Changing Element Values and Reading

Reading and changing the elements on the 3D array can be done similarly to the way we do the 2D array, just adjust for the extra level in the loops and coordinates.

```
Do
' using Do ... While for early exit
    For building = 0 To UBound(ThreeDArray, 1)
        For emp = 0 To UBound(ThreeDArray, 2)
            If ThreeDArray(building, emp, 0) = "Gloria" And ThreeDArray(building, emp, 1) =
"Stephan" Then
            'if value found
                ThreeDArray(building, emp, 1) = "Married, Last Name Change"
                Exit Do
                'don't iterate through all the array unless necessary
            End If
        Next
    Next
Loop While False

'nested loop, will iterate through all elements
For building = LBound(ThreeDArray, 1) To UBound(ThreeDArray, 1)
    For emp = LBound(ThreeDArray, 2) To UBound(ThreeDArray, 2)
        For info = LBound(ThreeDArray, 3) To UBound(ThreeDArray, 3)
            Debug.Print ThreeDArray(building, emp, info)
        Next
```

```
    Next
Next

'loop and coordinate, will iterate through all set of rows and ask for the row plus the value we
choose for the columns
For building = LBound(ThreeDArray, 1) To UBound(ThreeDArray, 1)
    For emp = LBound(ThreeDArray, 2) To UBound(ThreeDArray, 2)
        Debug.Print ThreeDArray(building, emp, 0)
        Debug.Print ThreeDArray(building, emp, 1)
        Debug.Print ThreeDArray(building, emp, 2)
        Debug.Print ThreeDArray(building, emp, 3)
        Debug.Print ThreeDArray(building, emp, 4)
        Debug.Print ThreeDArray(building, emp, 5)
    Next
Next

'directly accessing element with coordinates
Debug.Print Employees(0, 5, 5)
```

Section 9.2: Dynamic Arrays (Array Resizing and Dynamic Handling)

Dynamic Arrays

Adding and reducing variables on an array dynamically is a huge advantage for when the information you are treating does not have a set number of variables.

Adding Values Dynamically

You can simply resize the Array with the `ReDim` Statement, this will resize the array but to if you which to retain the information already stored in the array you'll need the part `Preserve`.

In the example below we create an array and increase it by one more variable in each iteration while preserving the values already in the array.

```
Dim Dynamic_array As Variant
' first we set Dynamic_array as variant

For n = 1 To 100

    If IsEmpty(Dynamic_array) Then
        'isempty() will check if we need to add the first value to the array or subsequent ones

        ReDim Dynamic_array(0)
        'ReDim Dynamic_array(0) will resize the array to one variable only
        Dynamic_array(0) = n

    Else
        ReDim Preserve Dynamic_array(0 To UBound(Dynamic_array) + 1)
        'in the line above we resize the array from variable 0 to the UBound() = last variable, plus
one effectivelly increeasing the size of the array by one
        Dynamic_array(UBound(Dynamic_array)) = n
        'attribute a value to the last variable of Dynamic_array
    End If

Next
```

Removing Values Dynamically

We can utilise the same logic to to decrease the the array. In the example the value "last" will be removed from the array.

```
Dim Dynamic_array As Variant
Dynamic_array = Array("first", "middle", "last")

ReDim Preserve Dynamic_array(0 To UBound(Dynamic_array) - 1)
' Resize Preserve while dropping the last value
```

Resetting an Array and Reusing Dynamically

We can as well re-utilise the arrays we create as not to have many on memory, which would make the run time slower. This is useful for arrays of various sizes. One snippet you could use to re-utilise the array is to `ReDim` the array back to `(0)`, attribute one variable to to the array and freely increase the array again.

In the snippet below I construct an array with the values 1 to 40, empty the array, and refill the array with values 40 to 100, all this done dynamically.

```
Dim Dynamic_array As Variant

For n = 1 To 100

    If IsEmpty(Dynamic_array) Then
        ReDim Dynamic_array(0)
        Dynamic_array(0) = n

    ElseIf Dynamic_array(0) = "" Then
        'if first variant is empty ( = "") then give it the value of n
        Dynamic_array(0) = n
    Else
        ReDim Preserve Dynamic_array(0 To UBound(Dynamic_array) + 1)
        Dynamic_array(UBound(Dynamic_array)) = n
    End If
    If n = 40 Then
        ReDim Dynamic_array(0)
        'Resizing the array back to one variable without Preserving,
        'leaving the first value of the array empty
    End If

Next
```

Section 9.3: Jagged Arrays (Arrays of Arrays)

Jagged Arrays NOT Multidimensional Arrays

Arrays of Arrays(Jagged Arrays) are not the same as Multidimensional Arrays if you think about them visually Multidimensional Arrays would look like Matrices (Rectangular) with defined number of elements on their dimensions(inside arrays), while Jagged array would be like a yearly calendar with the inside arrays having different number of elements, like days in on different months.

Although Jagged Arrays are quite messy and tricky to use due to their nested levels and don't have much type safety, but they are very flexible, allow you to manipulate different types of data quite easily, and don't need to contain unused or empty elements.

Creating a Jagged Array

In the below example we will initialise a jagged array containing two arrays one for Names and another for

Numbers, and then accessing one element of each

```
Dim OuterArray() As Variant
Dim Names() As Variant
Dim Numbers() As Variant
'arrays are declared variant so we can access attribute any data type to its elements

Names = Array("Person1", "Person2", "Person3")
Numbers = Array("001", "002", "003")

OuterArray = Array(Names, Numbers)
'Directly giving OuterArray an array containing both Names and Numbers arrays inside

Debug.Print OuterArray(0)(1)
Debug.Print OuterArray(1)(1)
'accessing elements inside the jagged by giving the coordenades of the element
```

Dynamically Creating and Reading Jagged Arrays

We can as well be more dynamic in our approx to construct the arrays, imagine that we have a customer data sheet in excel and we want to construct an array to output the customer details.

```
   Name -   Phone   -  Email  - Customer Number
Person1 - 153486231 - 1@STACK - 001
Person2 - 153486242 - 2@STACK - 002
Person3 - 153486253 - 3@STACK - 003
Person4 - 153486264 - 4@STACK - 004
Person5 - 153486275 - 5@STACK - 005
```

We will Dynamically construct an Header array and a Customers array, the Header will contain the column titles and the Customers array will contain the information of each customer/row as arrays.

```
Dim Headers As Variant
' headers array with the top section of the customer data sheet
    For c = 1 To 4
        If IsEmpty(Headers) Then
            ReDim Headers(0)
            Headers(0) = Cells(1, c).Value
        Else
            ReDim Preserve Headers(0 To UBound(Headers) + 1)
            Headers(UBound(Headers)) = Cells(1, c).Value
        End If
    Next

Dim Customers As Variant
'Customers array will contain arrays of customer values
Dim Customer_Values As Variant
'Customer_Values will be an array of the customer in its elements (Name-Phone-Email-CustNum)

    For r = 2 To 6
    'iterate through the customers/rows
        For c = 1 To 4
        'iterate through the values/columns

            'build array containing customer values
            If IsEmpty(Customer_Values) Then
                ReDim Customer_Values(0)
                Customer_Values(0) = Cells(r, c).Value
            ElseIf Customer_Values(0) = "" Then
```

```
                Customer_Values(0) = Cells(r, c).Value
            Else
                ReDim Preserve Customer_Values(0 To UBound(Customer_Values) + 1)
                Customer_Values(UBound(Customer_Values)) = Cells(r, c).Value
            End If
        Next

        'add customer_values array to Customers Array
        If IsEmpty(Customers) Then
            ReDim Customers(0)
            Customers(0) = Customer_Values
        Else
            ReDim Preserve Customers(0 To UBound(Customers) + 1)
            Customers(UBound(Customers)) = Customer_Values
        End If

        'reset Custumer_Values to rebuild a new array if needed
        ReDim Customer_Values(0)
    Next

    Dim Main_Array(0 To 1) As Variant
    'main array will contain both the Headers and Customers

    Main_Array(0) = Headers
    Main_Array(1) = Customers
```

To better understand the way to Dynamically construct a one dimensional array please check Dynamic Arrays (Array Resizing and Dynamic Handling) on the Arrays documentation.

The Result of the above snippet is an Jagged Array with two arrays one of those arrays with 4 elements, 2 indention levels, and the other being itself another Jagged Array containing 5 arrays of 4 elements each and 3 indention levels, see below the structure:

```
Main_Array(0) - Headers - Array("Name","Phone","Email","Customer Number")
          (1) - Customers(0) - Array("Person1",153486231,"1@STACK",001)
                Customers(1) - Array("Person2",153486242,"2@STACK",002)
                ...
                Customers(4) - Array("Person5",153486275,"5@STACK",005)
```

To access the information you'll have to bear in mind the structure of the Jagged Array you create, in the above example you can see that the `Main Array` contains an Array of `Headers` and an Array of Arrays (`Customers`) hence with different ways of accessing the elements.

Now we'll read the information of the `Main Array` and print out each of the Customers information as `Info Type`: `Info`.

```
For n = 0 To UBound(Main_Array(1))
    'n to iterate from first to last array in Main_Array(1)

    For j = 0 To UBound(Main_Array(1)(n))
        'j will iterate from first to last element in each array of Main_Array(1)

        Debug.Print Main_Array(0)(j) & ": " & Main_Array(1)(n)(j)
        'print Main_Array(0)(j) which is the header and Main_Array(0)(n)(j) which is the element in
the customer array
        'we can call the header with j as the header array has the same structure as the customer
array
    Next
Next
```

REMEMBER to keep track of the structure of your Jagged Array, in the example above to access the Name of a customer is by accessing `Main_Array -> Customers -> CustomerNumber -> Name` which is three levels, to return `"Person4"` you'll need the location of Customers in the Main_Array, then the Location of customer four on the Customers Jagged array and lastly the location of the element you need, in this case `Main_Array(1)(3)(0)` which is `Main_Array(Customers)(CustomerNumber)(Name)`.

Section 9.4: Declaring an Array in VBA

Declaring an array is very similar to declaring a variable, except you need to declare the dimension of the Array right after its name:

```
Dim myArray(9) As String 'Declaring an array that will contain up to 10 strings
```

By default, Arrays in VBA are **indexed from ZERO**, thus, the number inside the parenthesis doesn't refer to the size of the array, but rather to **the index of the last element**

Accessing Elements

Accessing an element of the Array is done by using the name of the Array, followed by the index of the element, inside parenthesis:

```
myArray(0) = "first element"
myArray(5) = "sixth element"
myArray(9) = "last element"
```

Array Indexing

You can change Arrays indexing by placing this line at the top of a module:

```
Option Base 1
```

With this line, all Arrays declared in the module will be **indexed from ONE**.

Specific Index

You can also declare each Array with its own index by using the `To` keyword, and the lower and upper bound (= index):

```
Dim mySecondArray(1 To 12) As String 'Array of 12 strings indexed from 1 to 12
Dim myThirdArray(13 To 24) As String 'Array of 12 strings indexed from 13 to 24
```

Dynamic Declaration

When you do not know the size of your Array prior to its declaration, you can use the dynamic declaration, and the `ReDim` keyword:

```
Dim myDynamicArray() As Strings 'Creates an Array of an unknown number of strings
ReDim myDynamicArray(5) 'This resets the array to 6 elements
```

Note that using the `ReDim` keyword will wipe out any previous content of your Array. To prevent this, you can use the `Preserve` keyword after `ReDim`:

```
Dim myDynamicArray(5) As String
myDynamicArray(0) = "Something I want to keep"
```

```
ReDim Preserve myDynamicArray(8) 'Expand the size to up to 9 strings
Debug.Print myDynamicArray(0) ' still prints the element
```

Section 9.5: Use of Split to create an array from a string

Split Function

returns a zero-based, one dimensional array containing a specified number of substrings.

Syntax

Split(expression [, **delimiter** [, **limit** [, **compare**]]])

Part	Description
expression	Required. String expression containing substrings and delimiters. If *expression* is a zero-length string("" or vbNullString), **Split** returns an empty array containing no elements and no data. In this case, the returned array will have a LBound of 0 and a UBound of -1.
delimiter	Optional. String character used to identify substring limits. If omitted, the space character (" ") is assumed to be the delimiter. If ***delimiter*** is a zero-length string, a single-element array containing the entire ***expression*** string is returned.
limit	Optional. Number of substrings to be returned; -1 indicates that all substrings are returned.
compare	Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. See Settings section for values.

Settings

The ***compare*** argument can have the following values:

Constant	Value	Description
Description	-1	Performs a comparison using the setting of the **Option Compare** statement.
vbBinaryCompare	0	Performs a binary comparison.
vbTextCompare	1	Performs a textual comparison.
vbDatabaseCompare	2	Microsoft Access only. Performs a comparison based on information in your database.

Example

In this example it is demonstrated how Split works by showing several styles. The comments will show the result set for each of the different performed Split options. Finally it is demonstrated how to loop over the returned string array.

```
Sub Test

    Dim textArray() as String

    textArray = Split("Tech on the Net")
    'Result: {"Tech", "on", "the", "Net"}

    textArray = Split("172.23.56.4", ".")
    'Result: {"172", "23", "56", "4"}

    textArray = Split("A;B;C;D", ";")
    'Result: {"A", "B", "C", "D"}

    textArray = Split("A;B;C;D", ";", 1)
    'Result: {"A;B;C;D"}

    textArray = Split("A;B;C;D", ";", 2)
    'Result: {"A", "B;C;D"}
```

```
    textArray = Split("A;B;C;D", ";", 3)
    'Result: {"A", "B", "C;D"}

    textArray = Split("A;B;C;D", ";", 4)
    'Result: {"A", "B", "C", "D"}

    'You can iterate over the created array
    Dim counter As Long

    For counter = LBound(textArray) To UBound(textArray)
        Debug.Print textArray(counter)
    Next
End Sub
```

Section 9.6: Iterating elements of an array

For...Next

Using the iterator variable as the index number is the fastest way to iterate the elements of an array:

```
Dim items As Variant
items = Array(0, 1, 2, 3)

Dim index As Integer
For index = LBound(items) To UBound(items)
    'assumes value can be implicitly converted to a String:
    Debug.Print items(index)
Next
```

Nested loops can be used to iterate multi-dimensional arrays:

```
Dim items(0 To 1, 0 To 1) As Integer
items(0, 0) = 0
items(0, 1) = 1
items(1, 0) = 2
items(1, 1) = 3

Dim outer As Integer
Dim inner As Integer
For outer = LBound(items, 1) To UBound(items, 1)
    For inner = LBound(items, 2) To UBound(items, 2)
        'assumes value can be implicitly converted to a String:
        Debug.Print items(outer, inner)
    Next
Next
```

For Each...Next

A `For Each...Next` loop can also be used to iterate arrays, if performance doesn't matter:

```
Dim items As Variant
items = Array(0, 1, 2, 3)

Dim item As Variant 'must be variant
For Each item In items
    'assumes value can be implicitly converted to a String:
    Debug.Print item
Next
```

A `For Each` loop will iterate all dimensions from outer to inner (the same order as the elements are laid out in memory), so there is no need for nested loops:

```
Dim items(0 To 1, 0 To 1) As Integer
items(0, 0) = 0
items(1, 0) = 1
items(0, 1) = 2
items(1, 1) = 3

Dim item As Variant 'must be Variant
For Each item In items
    'assumes value can be implicitly converted to a String:
    Debug.Print item
Next
```

Note that `For Each` loops are best used to iterate `Collection` objects, if performance matters.

All 4 snippets above produce the same output:

```
0
1
2
3
```

Chapter 10: Error Handling

Section 10.1: Avoiding error conditions

When a runtime error occurs, good code should handle it. The best error handling strategy is to write code that checks for error conditions and simply avoids executing code that results in a runtime error.

One key element in reducing runtime errors, is writing small procedures that *do one thing*. The fewer reasons procedures have to fail, the easier the code as a whole is to debug.

Avoiding runtime error 91 - Object or With block variable not set:

This error will be raised when an object is used before its reference is assigned. One might have a procedure that receives an object parameter:

```
Private Sub DoSomething(ByVal target As Worksheet)
    Debug.Print target.Name
End Sub
```

If `target` isn't assigned a reference, the above code will raise an error that is easily avoided by checking if the object contains an actual object reference:

```
Private Sub DoSomething(ByVal target As Worksheet)
    If target Is Nothing Then Exit Sub
    Debug.Print target.Name
End Sub
```

If `target` isn't assigned a reference, then the unassigned reference is never used, and no error occurs.

This way of early-exiting a procedure when one or more parameter isn't valid, is called a *guard clause*.

Avoiding runtime error 9 - Subscript out of range:

This error is raised when an array is accessed outside of its boundaries.

```
Private Sub DoSomething(ByVal index As Integer)
    Debug.Print ActiveWorkbook.Worksheets(index)
End Sub
```

Given an index greater than the number of worksheets in the `ActiveWorkbook`, the above code will raise a runtime error. A simple guard clause can avoid that:

```
Private Sub DoSomething(ByVal index As Integer)
    If index > ActiveWorkbook.Worksheets.Count Or index <= 0 Then Exit Sub
    Debug.Print ActiveWorkbook.Worksheets(index)
End Sub
```

Most runtime errors can be avoided by carefully verifying the values we're using *before* we use them, and branching on another execution path accordingly using a simple `If` statement - in guard clauses that makes no assumptions and validates a procedure's parameters, or even in the body of larger procedures.

Section 10.2: Custom Errors

Often when writing a specialized class, you'll want it to raise its own specific errors, and you'll want a clean way for

user/calling code to handle these custom errors. A neat way to achieve this is by defining a dedicated `Enum` type:

```
Option Explicit
Public Enum FoobarError
    Err_FooWasNotBarred = vbObjectError + 1024
    Err_BarNotInitialized
    Err_SomethingElseHappened
End Enum
```

Using the `vbObjectError` built-in constant ensures the custom error codes don't overlap with reserved/existing error codes. Only the first enum value needs to be explicitly specified, for the underlying value of each `Enum` member is `1` greater than the previous member, so the underlying value of `Err_BarNotInitialized` is implicitly `vbObjectError + 1025`.

Raising your own runtime errors

A runtime error can be raised using the `Err.Raise` statement, so the custom `Err_FooWasNotBarred` error can be raised as follows:

```
Err.Raise Err_FooWasNotBarred
```

The `Err.Raise` method can also take custom `Description` and `Source` parameters - for this reason it's a good idea to also define constants to hold each custom error's description:

```
Private Const Msg_FooWasNotBarred As String = "The foo was not barred."
Private Const Msg_BarNotInitialized As String = "The bar was not initialized."
```

And then create a dedicated private method to raise each error:

```
Private Sub OnFooWasNotBarredError(ByVal source As String)
    Err.Raise Err_FooWasNotBarred, source, Msg_FooWasNotBarred
End Sub

Private Sub OnBarNotInitializedError(ByVal source As String)
    Err.Raise Err_BarNotInitialized, source, Msg_BarNotInitialized
End Sub
```

The class' implementation can then simply call these specialized procedures to raise the error:

```
Public Sub DoSomething()
    'raises the custom 'BarNotInitialized' error with "DoSomething" as the source:
    If Me.Bar Is Nothing Then OnBarNotInitializedError "DoSomething"
    '...
End Sub
```

The client code can then handle `Err_BarNotInitialized` as it would any other error, inside its own error-handling subroutine.

Note: the legacy `Error` keyword can also be used in place of `Err.Raise`, but it's obsolete/deprecated.

Section 10.3: Resume keyword

An error-handling subroutine will either:

- run to the end of the procedure, in which case execution resumes in the calling procedure.
- or, use the `Resume` keyword to *resume* execution inside the same procedure.

The `Resume` keyword should only ever be used inside an error handling subroutine, because if VBA encounters `Resume` without being in an error state, runtime error 20 "Resume without error" is raised.

There are several ways an error-handling subroutine may use the `Resume` keyword:

- `Resume` used alone, execution continues **on the statement that caused the error**. If the error isn't *actually* handled before doing that, then the same error will be raised again, and execution might enter an infinite loop.
- `Resume Next` continues execution **on the statement immediately following** the statement that caused the error. If the error isn't *actually* handled before doing that, then execution is permitted to continue with potentially invalid data, which may result in logical errors and unexpected behavior.
- `Resume [line label]` continues execution **at the specified line label** (or line number, if you're using legacy-style line numbers). This would typically allow executing some cleanup code before cleanly exiting the procedure, such as ensuring a database connection is closed before returning to the caller.

On Error Resume Next

The `On Error` statement itself can use the `Resume` keyword to instruct the VBA runtime to effectively **ignore all errors**.

> *If the error isn't **actually handled** before doing that, then execution is permitted to continue with potentially invalid data, which may result in **logical errors and unexpected behavior**.*

The emphasis above cannot be emphasized enough. ***On Error Resume Next* effectively ignores all errors and shoves them under the carpet**. A program that blows up with a runtime error given invalid input is a better program than one that keeps running with unknown/unintended data - be it only because the bug is much more easily identifiable. `On Error Resume Next` can easily **hide bugs**.

The `On Error` statement is procedure-scoped - that's why there should *normally* be only **one**, single such `On Error` statement in a given procedure.

However *sometimes* an error condition can't quite be avoided, and jumping to an error-handling subroutine only to `Resume Next` just doesn't feel right. In this specific case, the known-to-possibly-fail statement can be **wrapped** between two `On Error` statements:

```
On Error Resume Next
[possibly-failing statement]
Err.Clear 'resets current error
On Error GoTo 0
```

The `On Error GoTo 0` instruction resets error handling in the current procedure, such that any further instruction causing a runtime error *would be unhandled within that procedure* and instead passed up the call stack until it is caught by an active error handler. If there is no active error handler in the call stack, it will be treated as an unhandled exception.

```
Public Sub Caller()
    On Error GoTo Handler

    Callee

    Exit Sub
Handler:
```

```
    Debug.Print "Error " & Err.Number & " in Caller."
End Sub

Public Sub Callee()
    On Error GoTo Handler

    Err.Raise 1     'This will be handled by the Callee handler.
    On Error GoTo 0 'After this statement, errors are passed up the stack.
    Err.Raise 2     'This will be handled by the Caller handler.

    Exit Sub
Handler:
    Debug.Print "Error " & Err.Number & " in Callee."
    Resume Next
End Sub
```

Section 10.4: On Error statement

Even with *guard clauses*, one cannot realistically *always* account for all possible error conditions that could be raised in the body of a procedure. The `On Error GoTo` statement instructs VBA to jump to a *line label* and enter "error handling mode" whenever an unexpected error occurs at runtime. After handling an error, code can *resume* back into "normal" execution using the `Resume` keyword.

Line labels denote *subroutines*: because subroutines originate from legacy BASIC code and uses `GoTo` and `GoSub` jumps and `Return` statements to jump back to the "main" routine, it's fairly easy to write hard-to-follow *spaghetti code* if things aren't rigorously structured. For this reason, it's best that:

- a procedure has **one and only one** error-handling subroutine
- the error-handling subroutine **only ever runs in an error state**

This means a procedure that handles its errors, should be structured like this:

```
Private Sub DoSomething()
    On Error GoTo CleanFail

    'procedure code here

CleanExit:
    'cleanup code here
    Exit Sub

CleanFail:
    'error-handling code here
    Resume CleanExit
End Sub
```

Error Handling Strategies

Sometimes you want to handle different errors with different actions. In that case you will inspect the global `Err` object, which will contain information about the error that was raised - and act accordingly:

```
CleanExit:
    Exit Sub

CleanFail:
    Select Case Err.Number
        Case 9
            MsgBox "Specified number doesn't exist. Please try again.", vbExclamation
```

```
            Resume
        Case 91
            'woah there, this shouldn't be happening.
            Stop 'execution will break here
            Resume 'hit F8 to jump to the line that raised the error
        Case Else
            MsgBox "An unexpected error has occurred:" & vbNewLine & Err.Description, vbCritical
            Resume CleanExit
    End Select
End Sub
```

As a general guideline, consider turning on the error handling for entire subroutine or function, and handle all the errors that may occur within its scope. If you need to only handle errors in the small section section of the code -- turn error handling on and off a the same level:

```
Private Sub DoSomething(CheckValue as Long)

    If CheckValue = 0 Then
        On Error GoTo ErrorHandler    ' turn error handling on
        ' code that may result in error
        On Error GoTo 0               ' turn error handling off - same level
    End If

CleanExit:
    Exit Sub

ErrorHandler:
    ' error handling code here
    ' do not turn off error handling here
    Resume

End Sub
```

Line numbers

VBA supports legacy-style (e.g. QBASIC) line numbers. The `Erl` hidden property can be used to identify the line number that raised the last error. If you're not using line numbers, `Erl` will only ever return 0.

```
Sub DoSomething()
10 On Error GoTo 50
20 Debug.Print 42 / 0
30 Exit Sub
40
50 Debug.Print "Error raised on line " & Erl ' returns 20
End Sub
```

If you *are* using line numbers, but not consistently, then `Erl` will return *the last line number before the instruction that raised the error*.

```
Sub DoSomething()
10 On Error GoTo 50
   Debug.Print 42 / 0
30 Exit Sub

50 Debug.Print "Error raised on line " & Erl 'returns 10
End Sub
```

Keep in mind that `Erl` also only has `Integer` precision, and will silently overflow. This means that line numbers

outside of the integer range will give incorrect results:

```
Sub DoSomething()
99997 On Error GoTo 99999
99998 Debug.Print 42 / 0
99999
      Debug.Print Erl   'Prints 34462
End Sub
```

The line number isn't quite as relevant as the statement that caused the error, and numbering lines quickly becomes tedious and not quite maintenance-friendly.

Chapter 11: Recursion

A function that calls itself is said to be *recursive*. Recursive logic can often be implemented as a loop, too. Recursion must be controlled with a parameter, so that the function knows when to stop recursing and deepening the call stack. *Infinite recursion* eventually causes a run-time error '28': "Out of stack space".

See Recursion.

Section 11.1: Factorials

```
Function Factorial(Value As Long) As Long
    If Value = 0 Or Value = 1 Then
         Factorial = 1
    Else
       Factorial = Factorial(Value - 1) * Value
    End If
End Function
```

Section 11.2: Folder Recursion

Early Bound (with a reference to `Microsoft Scripting Runtime`)

```
    Sub EnumerateFilesAndFolders( _
        FolderPath As String, _
        Optional MaxDepth As Long = -1, _
        Optional CurrentDepth As Long = 0, _
        Optional Indentation As Long = 2)

        Dim FSO As Scripting.FileSystemObject
        Set FSO = New Scripting.FileSystemObject

        'Check the folder exists
        If FSO.FolderExists(FolderPath) Then
            Dim fldr As Scripting.Folder
            Set fldr = FSO.GetFolder(FolderPath)

            'Output the starting directory path
            If CurrentDepth = 0 Then
              Debug.Print fldr.Path
            End If

            'Enumerate the subfolders
            Dim subFldr As Scripting.Folder
            For Each subFldr In fldr.SubFolders
                Debug.Print Space$((CurrentDepth + 1) * Indentation) & subFldr.Name
                If CurrentDepth < MaxDepth Or MaxDepth = -1 Then
                    'Recursively call EnumerateFilesAndFolders
                    EnumerateFilesAndFolders subFldr.Path, MaxDepth, CurrentDepth + 1, Indentation
                End If
            Next subFldr

            'Enumerate the files
            Dim fil As Scripting.File
            For Each fil In fldr.Files
                Debug.Print Space$((CurrentDepth + 1) * Indentation) & fil.Name
            Next fil
        End If
```

```
End Sub
```

Chapter 12: Conditional Compilation

Section 12.1: Changing code behavior at compile time

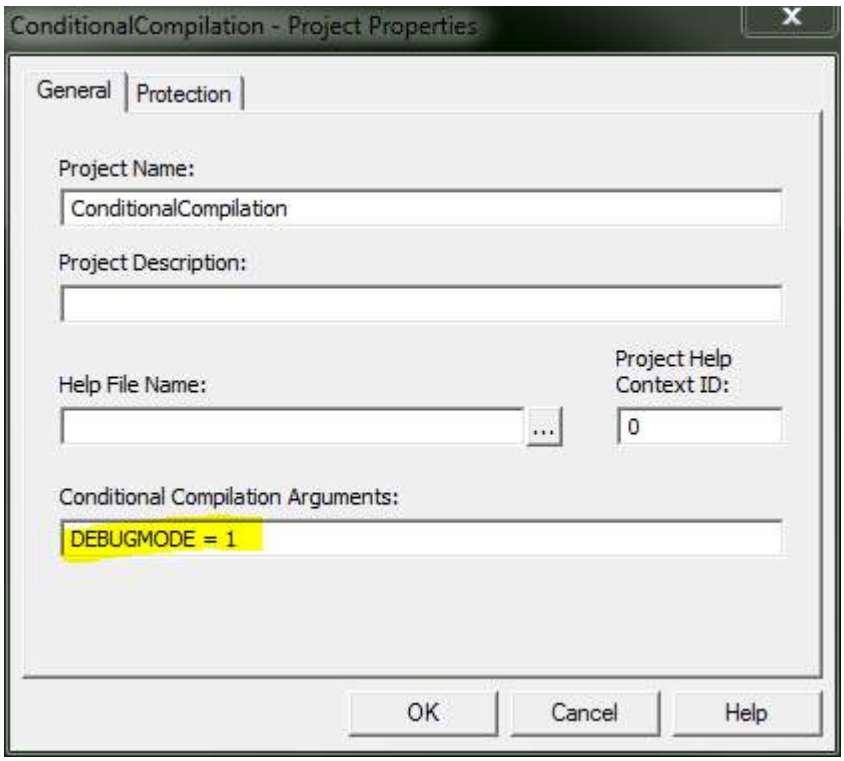
The `#Const` directive is used to define a custom preprocessor constant. These can later be used by `#If` to control which blocks of code get compiled and executed.

```
#Const DEBUGMODE = 1

#If DEBUGMODE Then
    Const filepath As String = "C:\Users\UserName\Path\To\File.txt"
#Else
    Const filepath As String = "\\server\share\path\to\file.txt"
#End If
```

This results in the value of `filepath` being set to `"C:\Users\UserName\Path\To\File.txt"`. Removing the `#Const` line, or changing it to `#Const DEBUGMODE = 0` would result in the `filepath` being set to `"\\server\share\path\to\file.txt"`.

#Const Scope

The `#Const` directive is only effective for a single code file (module or class). It must be declared for each and every file you wish to use your custom constant in. Alternatively, you can declare a `#Const` globally for your project by going to Tools >> [Your Project Name] Project Properties. This will bring up the project properties dialog box where we'll enter the constant declaration. In the "Conditional Compilation Arguments" box, type in `[constName] = [value]`. You can enter more than 1 constant by separating them with a colon, like `[constName1] = [value1] : [constName2] = [value2]`.



Pre-defined Constants

Some compilation constants are already pre-defined. Which ones exist will depend on the bitness of the office version you're running VBA in. Note that Vba7 was introduced alongside Office 2010 to support 64 bit versions of Office.

Constant	16 bit	32 bit	64 bit
Vba6	False	If Vba6	False
Vba7	False	If Vba7	True
Win16	True	False	False
Win32	False	True	True
Win64	False	False	True
Mac	False	If Mac	If Mac

Note that Win64/Win32 refer to the Office version, not the Windows version. For example Win32 = TRUE in 32-bit Office, even if the OS is a 64-bit version of Windows.

Section 12.2: Using Declare Imports that work on all versions of Office

```
#If Vba7 Then
    ' It's important to check for Win64 first,
    ' because Win32 will also return true when Win64 does.

    #If Win64 Then
        Declare PtrSafe Function GetFoo64 Lib "exampleLib32" () As LongLong
    #Else
        Declare PtrSafe Function GetFoo Lib "exampleLib32" () As Long
    #End If
#Else
    ' Must be Vba6, the PtrSafe keyword didn't exist back then,
    ' so we need to declare Win32 imports a bit differently than above.

    #If Win32 Then
        Declare Function GetFoo Lib "exampleLib32"() As Long
    #Else
        Declare Function GetFoo Lib "exampleLib"() As Integer
    #End If
#End If
```

This can be simplified a bit depending on what versions of office you need to support. For example, not many people are still supporting 16 bit versions of Office. The last version of 16 bit office was version 4.3, released in 1994, so the following declaration is sufficient for nearly all modern cases (including Office 2007).

```
#If Vba7 Then
    ' It's important to check for Win64 first,
    ' because Win32 will also return true when Win64 does.

    #If Win64 Then
        Declare PtrSafe Function GetFoo64 Lib "exampleLib32" () As LongLong
    #Else
        Declare PtrSafe Function GetFoo Lib "exampleLib32" () As Long
    #End If
#Else
    ' Must be Vba6. We don't support 16 bit office, so must be Win32.

    Declare Function GetFoo Lib "exampleLib32"() As Long
#End If
```

If you don't have to support anything older than Office 2010, this declaration works just fine.

```
' We only have 2010 installs, so we already know we have Vba7.

#If Win64 Then
```

```
    Declare PtrSafe Function GetFoo64 Lib "exampleLib32" () As LongLong
#Else
    Declare PtrSafe Function GetFoo Lib "exampleLib32" () As Long
#End If
```

Chapter 13: Data Types and Limits

Section 13.1: Variant

```
Dim Value As Variant    'Explicit
Dim Value               'Implicit
```

A Variant is a COM data type that is used for storing and exchanging values of arbitrary types, and any other type in VBA can be assigned to a Variant. Variables declared without an explicit type specified by `As [Type]` default to Variant.

Variants are stored in memory as a VARIANT structure that consists of a byte type descriptor (VARTYPE) followed by 6 reserved bytes then an 8 byte data area. For numeric types (including Date and Boolean), the underlying value is stored in the Variant itself. For all other types, the data area contains a pointer to the underlying value.

VARTYPE		Reserved						Data area			
0	1	2	3	4	5	6	7	8	9	10	11

The underlying type of a Variant can be determined with either the `VarType()` function which returns the numeric value stored in the type descriptor, or the `TypeName()` function which returns the string representation:

```
Dim Example As Variant
Example = 42
Debug.Print VarType(Example)    'Prints 2 (VT_I2)
Debug.Print TypeName(Example)   'Prints "Integer"
Example = "Some text"
Debug.Print VarType(Example)    'Prints 8 (VT_BSTR)
Debug.Print TypeName(Example)   'Prints "String"
```

Because Variants can store values of any type, assignments from literals without type hints will be implicitly cast to a Variant of the appropriate type according to the table below. Literals with type hints will be cast to a Variant of the hinted type.

Value	Resulting type
String values	String
Non-floating point numbers in Integer range	Integer
Non-floating point numbers in Long range	Long
Non-floating point numbers outside of Long range	Double
All floating point numbers	Double

Note: Unless there is a specific reason to use a Variant (i.e. an iterator in a For Each loop or an API requirement), the type should generally be avoided for routine tasks for the following reasons:

- They are not type safe, increasing the possibility of runtime errors. For example, a Variant holding an Integer value will silently change itself into a Long instead of overflowing.
- They introduce processing overhead by requiring at least one additional pointer dereference.
- The memory requirement for a Variant is always ***at least*** 8 bytes higher than needed to store the underlying type.

The casting function to convert to a Variant is `CVar()`.

Section 13.2: Boolean

```
Dim Value As Boolean
```

A Boolean is used to store values that can be represented as either True or False. Internally, the data type is stored as a 16 bit value with 0 representing False and any other value representing True.

It should be noted that when a Boolean is cast to a numeric type, all of the bits are set to 1. This results in an internal representation of -1 for signed types and the maximum value for an unsigned type (Byte).

```
Dim Example As Boolean
Example = True
Debug.Print CInt(Example)  'Prints -1
Debug.Print CBool(42)      'Prints True
Debug.Print CByte(True)    'Prints 255
```

The casting function to convert to a Boolean is `CBool()`. Even though it is represented internally as a 16 bit number, casting to a Boolean from values outside of that range is safe from overflow, although it sets all 16 bits to 1:

```
Dim Example As Boolean
Example = CBool(2 ^ 17)
Debug.Print CInt(Example)   'Prints -1
Debug.Print CByte(Example)  'Prints 255
```

Section 13.3: String

A String represents a sequence of characters, and comes in two flavors:

Variable length

```
Dim Value As String
```

A variable length String allows appending and truncation and is stored in memory as a COM BSTR. This consists of a 4 byte unsigned integer that stores the length of the String in bytes followed by the string data itself as wide characters (2 bytes per character) and terminated with 2 null bytes. Thus, the maximum string length that can be handled by VBA is 2,147,483,647 characters.

The internal pointer to the structure (retrievable by the `StrPtr()` function) points to the memory location of the *data*, not the length prefix. This means that a VBA String can be passed directly API functions that require a pointer to a character array.

Because the length can change, VBA reallocates memory for a String *every time the variable is assigned to*, which can impose performance penalties for procedures that alter them repeatedly.

Fixed length

```
Dim Value As String * 1024    'Declares a fixed length string of 1024 characters.
```

Fixed length strings are allocated 2 bytes for each character and are stored in memory as a simple byte array. Once allocated, the length of the String is immutable. They are ***not*** null terminated in memory, so a string that fills the memory allocated with non-null characters is unsuitable for passing to API functions expecting a null terminated string.

Fixed length strings carry over a legacy 16 bit index limitation, so can only be up to 65,535 characters in length. Attempting to assign a value longer than the available memory space will not result in a runtime error - instead the resulting value will simply be truncated:

```
Dim Foobar As String * 5
```

```
Foobar = "Foo" & "bar"
Debug.Print Foobar          'Prints "Fooba"
```

The casting function to convert to a String of either type is `CStr()`.

Section 13.4: Byte

```
Dim Value As Byte
```

A Byte is an unsigned 8 bit data type. It can represent integer numbers between 0 and 255 and attempting to store a value outside of that range will result in runtime error 6: `Overflow`. Byte is the only intrinsic unsigned type available in VBA.

The casting function to convert to a Byte is `CByte()`. For casts from floating point types, the result is rounded to the nearest integer value with .5 rounding up.

Byte Arrays and Strings

Strings and byte arrays can be substituted for one another through simple assignment (no conversion functions necessary).

For example:

```
Sub ByteToStringAndBack()

Dim str As String
str = "Hello, World!"

Dim byt() As Byte
byt = str

Debug.Print byt(0)  ' 72

Dim str2 As String
str2 = byt

Debug.Print str2    ' Hello, World!

End Sub
```

In order to be able to encode Unicode characters, each character in the string takes up two bytes in the array, with the least significant byte first. For example:

```
Sub UnicodeExample()

Dim str As String
str = ChrW(&H2123) & "."  ' Versicle character and a dot

Dim byt() As Byte
byt = str

Debug.Print byt(0), byt(1), byt(2), byt(3)  ' Prints: 35,33,46,0

End Sub
```

Section 13.5: Currency

```
Dim Value As Currency
```

A Currency is a signed 64 bit floating point data type similar to a Double, but scaled by 10,000 to give greater precision to the 4 digits to the right of the decimal point. A Currency variable can store values from -922,337,203,685,477.5808 to 922,337,203,685,477.5807, giving it the largest capacity of any intrinsic type in a 32 bit application. As the name of the data type implies, it is considered best practice to use this data type when representing monetary calculations as the scaling helps to avoid rounding errors.

The casting function to convert to a Currency is `CCur()`.

Section 13.6: Decimal

```
Dim Value As Variant
Value = CDec(1.234)

'Set Value to the smallest possible Decimal value
Value = CDec("0.0000000000000000000000000001")
```

The `Decimal` data-type is *only* available as a sub-type of `Variant`, so you must declare any variable that needs to contain a `Decimal` as a `Variant` and *then* assign a `Decimal` value using the `CDec` function. The keyword `Decimal` is a reserved word (which suggests that VBA was eventually going to add first-class support for the type), so `Decimal` cannot be used as a variable or procedure name.

The `Decimal` type requires 14 bytes of memory (in addition to the bytes required by the parent Variant) and can store numbers with up to 28 decimal places. For numbers without any decimal places, the range of allowed values is -79,228,162,514,264,337,593,543,950,335 to +79,228,162,514,264,337,593,543,950,335 inclusive. For numbers with the maximum 28 decimal places, the range of allowed values is -7.9228162514264337593543950335 to +7.9228162514264337593543950335 inclusive.

Section 13.7: Integer

```
Dim Value As Integer
```

An Integer is a signed 16 bit data type. It can store integer numbers in the range of -32,768 to 32,767 and attempting to store a value outside of that range will result in runtime error 6: Overflow.

Integers are stored in memory as little-endian values with negatives represented as a two's complement.

Note that in general, it is better practice to use a Long rather than an Integer unless the smaller type is a member of a Type or is required (either by an API calling convention or some other reason) to be 2 bytes. In most cases VBA treats Integers as 32 bit internally, so there is usually no advantage to using the smaller type. Additionally, there is a performance penalty incurred every time an Integer type is used as it is silently cast as a Long.

The casting function to convert to an Integer is `CInt()`. For casts from floating point types, the result is rounded to the nearest integer value with .5 rounding up.

Section 13.8: Long

```
Dim Value As Long
```

A Long is a signed 32 bit data type. It can store integer numbers in the range of -2,147,483,648 to 2,147,483,647 and

attempting to store a value outside of that range will result in runtime error 6: Overflow.

Longs are stored in memory as little-endian values with negatives represented as a two's complement.

Note that since a Long matches the width of a pointer in a 32 bit operating system, Longs are commonly used for storing and passing pointers to and from API functions.

The casting function to convert to a Long is `CLng()`. For casts from floating point types, the result is rounded to the nearest integer value with .5 rounding up.

Section 13.9: Single

```
Dim Value As Single
```

A Single is a signed 32 bit floating point data type. It is stored internally using a little-endian IEEE 754 memory layout. As such, there is not a fixed range of values that can be represented by the data type - what is limited is the precision of value stored. A Single can store a value ***integer*** values in the range of -16,777,216 to 16,777,216 without a loss of precision. The precision of floating point numbers depends on the exponent.

A Single will overflow if assigned a value greater than roughly 2128. It will not overflow with negative exponents, although the usable precision will be questionable before the upper limit is reached.

As with all floating point numbers, care should be taken when making equality comparisons. Best practice is to include a delta value appropriate to the required precision.

The casting function to convert to a Single is `CSng()`.

Section 13.10: Double

```
Dim Value As Double
```

A Double is a signed 64 bit floating point data type. Like the Single, it is stored internally using a little-endian IEEE 754 memory layout and the same precautions regarding precision should be taken. A Double can store ***integer*** values in the range of -9,007,199,254,740,992 to 9,007,199,254,740,992 without a loss of precision. The precision of floating point numbers depends on the exponent.

A Double will overflow if assigned a value greater than roughly 21024. It will not overflow with negative exponents, although the usable precision will be questionable before the upper limit is reached.

The casting function to convert to a Double is `CDbl()`.

Section 13.11: Date

```
Dim Value As Date
```

A Date type is represented internally as a signed 64 bit floating point data type with the value to the left of the decimal representing the number of days from the epoch date of December 30th, 1899 (although see the note below). The value to the right of the decimal represents the time as a fractional day. Thus, an integer Date would have a time component of 12:00:00AM and x.5 would have a time component of 12:00:00PM.

Valid values for Dates are between January 1st 100 and December 31st 9999. Since a Double has a larger range, it is possible to overflow a Date by assigning values outside of that range.

As such, it can be used interchangeably with a Double for Date calculations:

```
Dim MyDate As Double
MyDate = 0                                  'Epoch date.
Debug.Print Format$(MyDate, "yyyy-mm-dd")   'Prints 1899-12-30.
MyDate = MyDate + 365
Debug.Print Format$(MyDate, "yyyy-mm-dd")   'Prints 1900-12-30.
```

The casting function to convert to a Date is `CDate()`, which accepts any numeric type string date/time representation. It is important to note that string representations of dates will be converted based on the current locale setting in use, so direct casts should be avoided if the code is meant to be portable.

Section 13.12: LongLong

```
Dim Value As LongLong
```

A LongLong is a signed 64 bit data type and is only available in 64 bit applications. It is **not** available in 32 bit applications running on 64 bit operating systems. It can store integer values in the range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 and attempting to store a value outside of that range will result in runtime error 6: Overflow.

LongLongs are stored in memory as little-endian values with negatives represented as a two's complement.

The LongLong data type was introduced as part of VBA's 64 bit operating system support. In 64 bit applications, this value can be used to store and pass pointers to 64 bit APIs.

The casting function to convert to a LongLong is `CLngLng()`. For casts from floating point types, the result is rounded to the nearest integer value with .5 rounding up.

Section 13.13: LongPtr

```
Dim Value As LongPtr
```

The LongPtr was introduced into VBA in order to support 64 bit platforms. On a 32 bit system, it is treated as a Long and on 64 bit systems it is treated as a LongLong.

It's primary use is in providing a portable way to store and pass pointers on both architectures (See Changing code behavior at compile time.

Although it is treated by the operating system as a memory address when used in API calls, it should be noted that VBA treats it like signed type (and therefore subject to unsigned to signed overflow). For this reason, any pointer arithmetic performed using LongPtrs should not use > or < comparisons. This "quirk" also makes it possible that adding simple offsets pointing to valid addresses in memory can cause overflow errors, so caution should be taken when working with pointers in VBA.

The casting function to convert to a LongPtr is `CLngPtr()`. For casts from floating point types, the result is rounded to the nearest integer value with .5 rounding up (although since it is usually a memory address, using it as an assignment target for a floating point calculation is dangerous at best).

Chapter 14: String Literals - Escaping, non-printable characters and line-continuations

Section 14.1: Escaping the " character

VBA syntax requires that a string-literal appear within `"` marks, so when your string needs to *contain* quotation marks, you'll need to escape/prepend the `"` character with an extra `"` so that VBA understands that you intend the `""` to be interpreted as a `"` string.

```
'The following 2 lines produce the same output
Debug.Print "The man said, ""Never use air-quotes"""
Debug.Print "The man said, " & """ & "Never use air-quotes" & """

'Output:
'The man said, "Never use air-quotes"
'The man said, "Never use air-quotes"
```

Section 14.2: Assigning long string literals

The VBA editor only allows 1023 characters per line, but typically only the first 100-150 characters are visible without scrolling. If you need to assign long string literals, but you want to keep your code readable, you'll need to use line-continuations and concatenation to assign your string.

```
Debug.Print "Lorem ipsum dolor sit amet, consectetur adipiscing elit. " & _
            "Integer hendrerit maximus arcu, ut elementum odio varius " & _
            "nec. Integer ipsum enim, iaculis et egestas ac, condiment" & _
            "um ut tellus."
'Output:
'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer hendrerit maximus arcu, ut
elementum odio varius nec. Integer ipsum enim, iaculis et egestas ac, condimentum ut tellus.
```

VBA will let you use a limited number of line-continuations (the actual number varies by the length of each line within the continued-block), so if you have very long strings, you'll need to assign and re-assign with concatenation.

```
Dim loremIpsum As String

'Assign the first part of the string
loremIpsum = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. " & _
             "Integer hendrerit maximus arcu, ut elementum odio varius "
'Re-assign with the previous value AND the next section of the string
loremIpsum = loremIpsum & _
            "nec. Integer ipsum enim, iaculis et egestas ac, condiment" & _
            "um ut tellus."

Debug.Print loremIpsum

'Output:
'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer hendrerit maximus arcu, ut
elementum odio varius nec. Integer ipsum enim, iaculis et egestas ac, condimentum ut tellus.
```

Section 14.3: Using VBA string constants

VBA defines a number of string constants for special characters like:

- vbCr : Carriage-Return 'Same as "\r" in C style languages.
- vbLf : Line-Feed 'Same as "\n" in C style languages.
- vbCrLf : Carriage-Return & Line-Feed (a new-line in Windows)
- vbTab: Tab Character
- vbNullString: an empty string, like ""

You can use these constants with concatenation and other string functions to build string-literals with special-characters.

```
Debug.Print "Hello " & vbCrLf & "World"
'Output:
'Hello
'World

Debug.Print vbTab & "Hello" & vbTab & "World"
'Output:
'    Hello    World

Dim EmptyString As String
EmptyString = vbNullString
Debug.Print EmptyString = ""
'Output:
'True
```

Using `vbNullString` is considered better practice than the equivalent value of `""` due to differences in how the code is compiled. Strings are accessed via a pointer to an allocated area of memory, and the VBA compiler is smart enough to use a null pointer to represent `vbNullString`. The literal `""` is allocated memory as if it were a String typed Variant, making the use of the constant much more efficient:

```
Debug.Print StrPtr(vbNullString)    'Prints 0.
Debug.Print StrPtr("")              'Prints a memory address.
```

Chapter 15: Declaring and assigning strings

Section 15.1: Assignment to and from a byte array

Strings can be assigned directly to byte arrays and visa-versa. Remember that Strings are stored in a Multi-Byte Character Set (see Remarks below) so only every other index of the resulting array will be the portion of the character that falls within the ASCII range.

```
Dim bytes() As Byte
Dim example As String

example = "Testing."
bytes = example             'Direct assignment.

'Loop through the characters. Step 2 is used due to wide encoding.
Dim i As Long
For i = LBound(bytes) To UBound(bytes) Step 2
    Debug.Print Chr$(bytes(i))  'Prints T, e, s, t, i, n, g, .
Next

Dim reverted As String
reverted = bytes            'Direct assignment.
Debug.Print reverted        'Prints "Testing."
```

Section 15.2: Declare a string constant

```
Const appName As String = "The App For That"
```

Section 15.3: Declare a variable-width string variable

```
Dim surname As String 'surname can accept strings of variable length
surname = "Smith"
surname = "Johnson"
```

Section 15.4: Declare and assign a fixed-width string

```
'Declare and assign a 1-character fixed-width string
Dim middleInitial As String * 1 'middleInitial must be 1 character in length
middleInitial = "M"

'Declare and assign a 2-character fixed-width string `stateCode`,
'must be 2 characters in length
Dim stateCode As String * 2
stateCode = "TX"
```

Section 15.5: Declare and assign a string array

```
'Declare, dimension and assign a string array with 3 elements
Dim departments(2) As String
departments(0) = "Engineering"
departments(1) = "Finance"
departments(2) = "Marketing"
```

```
'Declare an undimensioned string array and then dynamically assign with
'the results of a function that returns a string array
Dim stateNames() As String
stateNames = VBA.Strings.Split("Texas;California;New York", ";")

'Declare, dimension and assign a fixed-width string array
Dim stateCodes(2) As String * 2
stateCodes(0) = "TX"
stateCodes(1) = "CA"
stateCodes(2) = "NY"
```

Section 15.6: Assign specific characters within a string using Mid statement

VBA offers a Mid function for *returning* substrings within a string, but it also offers the Mid *Statement* which can be used to assign substrings or individual characters withing a string.

The `Mid` function will typically appear on the right-hand-side of an assignment statement or in a condition, but the `Mid` Statement typically appears on the left hand side of an assignment statement.

```
Dim surname As String
surname = "Smith"

'Use the Mid statement to change the 3rd character in a string
Mid(surname, 3, 1) = "y"
Debug.Print surname

'Output:
'Smyth
```

Note: If you need to assign to individual *bytes* in a string instead of individual *characters* within a string (see the Remarks below regarding the Multi-Byte Character Set), the `MidB` statement can be used. In this instance, the second argument for the `MidB` statement is the 1-based position of the byte where the replacement will start so the equivalent line to the example above would be `MidB(surname, 5, 2) = "y"`.

Chapter 16: Converting other types to strings

Section 16.1: Use CStr to convert a numeric type to a string

```
Const zipCode As Long = 10012
Dim zipCodeText As String
'Convert the zipCode number to a string of digit characters
zipCodeText = CStr(zipCode)
'zipCodeText = "10012"
```

Section 16.2: Use Format to convert and format a numeric type as a string

```
Const zipCode As long = 10012
Dim zeroPaddedNumber As String
zeroPaddedZipCode = Format(zipCode, "00000000")
'zeroPaddedNumber = "00010012"
```

Section 16.3: Use StrConv to convert a byte-array of single-byte characters to a string

```
'Declare an array of bytes, assign single-byte character codes, and convert to a string
Dim singleByteChars(4) As Byte
singleByteChars(0) = 72
singleByteChars(1) = 101
singleByteChars(2) = 108
singleByteChars(3) = 108
singleByteChars(4) = 111
Dim stringFromSingleByteChars As String
stringFromSingleByteChars = StrConv(singleByteChars, vbUnicode)
'stringFromSingleByteChars = "Hello"
```

Section 16.4: Implicitly convert a byte array of multi-byte-characters to a string

```
'Declare an array of bytes, assign multi-byte character codes, and convert to a string
Dim multiByteChars(9) As Byte
multiByteChars(0) = 87
multiByteChars(1) = 0
multiByteChars(2) = 111
multiByteChars(3) = 0
multiByteChars(4) = 114
multiByteChars(5) = 0
multiByteChars(6) = 108
multiByteChars(7) = 0
multiByteChars(8) = 100
multiByteChars(9) = 0

Dim stringFromMultiByteChars As String
stringFromMultiByteChars = multiByteChars
'stringFromMultiByteChars = "World"
```

Chapter 17: Searching within strings for the presence of substrings

Section 17.1: Use InStr to determine if a string contains a substring

```
Const baseString As String = "Foo Bar"
Dim containsBar As Boolean

'Check if baseString contains "bar" (case insensitive)
containsBar = InStr(1, baseString, "bar", vbTextCompare) > 0
'containsBar = True

'Check if baseString contains bar (case insensitive)
containsBar = InStr(1, baseString, "bar", vbBinaryCompare) > 0
'containsBar = False
```

Section 17.2: Use InStrRev to find the position of the last instance of a substring

```
Const baseString As String = "Foo Bar"
Dim containsBar As Boolean

'Find the position of the last "B"
Dim posX As Long
'Note the different number and order of the parameters for InStrRev
posX = InStrRev(baseString, "X", -1, vbBinaryCompare)
'posX = 0
```

Section 17.3: Use InStr to find the position of the first instance of a substring

```
Const baseString As String = "Foo Bar"
Dim containsBar As Boolean

Dim posB As Long
posB = InStr(1, baseString, "B", vbBinaryCompare)
'posB = 5
```

Chapter 18: Substrings

Section 18.1: Use Left or Left$ to get the 3 left-most characters in a string

```
Const baseString As String = "Foo Bar"

Dim leftText As String
leftText = Left$(baseString, 3)
'leftText = "Foo"
```

Section 18.2: Use Right or Right$ to get the 3 right-most characters in a string

```
Const baseString As String = "Foo Bar"
Dim rightText As String
rightText = Right$(baseString, 3)
'rightText = "Bar"
```

Section 18.3: Use Mid or Mid$ to get specific characters from within a string

```
Const baseString As String = "Foo Bar"

'Get the string starting at character 2 and ending at character 6
Dim midText As String
midText = Mid$(baseString, 2, 5)
'midText = "oo Ba"
```

Section 18.4: Use Trim to get a copy of the string without any leading or trailing spaces

```
'Trim the leading and trailing spaces in a string
Const paddedText As String = "    Foo Bar    "
Dim trimmedText As String
trimmedText = Trim$(paddedText)
'trimmedText = "Foo Bar"
```

Chapter 19: Measuring the length of strings

Section 19.1: Use the Len function to determine the number of characters in a string

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

Dim charLength As Long

charLength = Len(baseString)
'charlength = 11
```

Section 19.2: Use the LenB function to determine the number of bytes in a string

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

Dim byteLength As Long

byteLength = LenB(baseString)
'byteLength = 22
```

Section 19.3: Prefer `If Len(myString) = 0 Then` over `If myString = "" Then`

When checking if a string is zero-length, it is better practice, and more efficient, to inspect the length of the string rather than comparing the string to an empty string.

```
Const myString As String = vbNullString

'Prefer this method when checking if myString is a zero-length string
If Len(myString) = 0 Then
    Debug.Print "myString is zero-length"
End If

'Avoid using this method when checking if myString is a zero-length string
If myString = vbNullString Then
    Debug.Print "myString is zero-length"
End If
```

Chapter 20: Working with ADO

Section 20.1: Making a connection to a data source

The first step in accessing a data source via ADO is creating an ADO `Connection` object. This is typically done using a connection string to specify the data source parameters, although it is also possible to open a DSN connection by passing the DSN, user ID, and password to the `.Open` method.

Note that a DSN is not required to connect to a data source via ADO - any data source that has an ODBC provider can be connected to with the appropriate connection string. While specific connection strings for different providers are outside of the scope of this topic, ConnectionStrings.com is an excellent reference for finding the appropriate string for your provider.

```
Const SomeDSN As String = "DSN=SomeDSN;Uid=UserName;Pwd=MyPassword;"

Public Sub Example()
    Dim database As ADODB.Connection
    Set database = OpenDatabaseConnection(SomeDSN)
    If Not database Is Nothing Then
        '... Do work.
        database.Close          'Make sure to close all database connections.
    End If
End Sub

Public Function OpenDatabaseConnection(ConnString As String) As ADODB.Connection
    On Error GoTo Handler
    Dim database As ADODB.Connection
    Set database = New ADODB.Connection

    With database
        .ConnectionString = ConnString
        .ConnectionTimeout = 10             'Value is given in seconds.
        .Open
    End With

    OpenDatabaseConnection = database

    Exit Function
Handler:
    Debug.Print "Database connection failed. Check your connection string."
End Function
```

Note that the database password is included in the connection string in the example above only for the sake of clarity. Best practices would dictate ***not*** storing database passwords in code. This can be accomplished by taking the password via user input or using Windows authentication.

Section 20.2: Creating parameterized commands

Any time SQL executed through an ADO connection needs to contain user input, it is considered best practice to parameterize it in order to minimize the chance of SQL injection. This method is also more readable than long concatenations and facilitates more robust and maintainable code (i.e. by using a function that returns an array of `Parameter`).

In standard ODBC syntax, parameters are given ? "placeholders" in the query text, and then parameters are appended to the `Command` in the same order that they appear in the query.

Note that the example below uses the `OpenDatabaseConnection` function from the Making a connection to a data source for brevity.

```
Public Sub UpdateTheFoos()
    On Error GoTo Handler
    Dim database As ADODB.Connection
    Set database = OpenDatabaseConnection(SomeDSN)

    If Not database Is Nothing Then
        Dim update As ADODB.Command
        Set update = New ADODB.Command
        'Build the command to pass to the data source.
        With update
            .ActiveConnection = database
            .CommandText = "UPDATE Table SET Foo = ? WHERE Bar = ?"
            .CommandType = adCmdText

            'Create the parameters.
            Dim fooValue As ADODB.Parameter
            Set fooValue = .CreateParameter("FooValue", adNumeric, adParamInput)
            fooValue.Value = 42

            Dim condition As ADODB.Parameter
            Set condition = .CreateParameter("Condition", adBSTR, adParamInput)
            condition.Value = "Bar"

            'Add the parameters to the Command
            .Parameters.Append fooValue
            .Parameters.Append condition
            .Execute
        End With
    End If
CleanExit:
    If Not database Is Nothing And database.State = adStateOpen Then
        database.Close
    End If
    Exit Sub
Handler:
    Debug.Print "Error " & Err.Number & ": " & Err.Description
    Resume CleanExit
End Sub
```

Note: The example above demonstrates a parameterized UPDATE statement, but any SQL statement can be given parameters.

Section 20.3: Retrieving records with a query

Queries can be performed in two ways, both of which return an ADO `Recordset` object which is a collection of returned rows. Note that both of the examples below use the `OpenDatabaseConnection` function from the Making a connection to a data source example for the purpose of brevity. Remember that the syntax of the SQL passed to the data source is provider specific.

The first method is to pass the SQL statement directly to the Connection object, and is the easiest method for executing simple queries:

```
Public Sub DisplayDistinctItems()
    On Error GoTo Handler
    Dim database As ADODB.Connection
    Set database = OpenDatabaseConnection(SomeDSN)
```

```
    If Not database Is Nothing Then
        Dim records As ADODB.Recordset
        Set records = database.Execute("SELECT DISTINCT Item FROM Table")
        'Loop through the returned Recordset.
        Do While Not records.EOF      'EOF is false when there are more records.
            'Individual fields are indexed either by name or 0 based ordinal.
            'Note that this is using the default .Fields member of the Recordset.
            Debug.Print records("Item")
            'Move to the next record.
            records.MoveNext
        Loop
    End If
CleanExit:
    If Not records Is Nothing Then records.Close
    If Not database Is Nothing And database.State = adStateOpen Then
        database.Close
    End If
    Exit Sub
Handler:
    Debug.Print "Error " & Err.Number & ": " & Err.Description
    Resume CleanExit
End Sub
```

The second method is to create an ADO `Command` object for the query you want to execute. This requires a little more code, but is necessary in order to use parametrized queries:

```
Public Sub DisplayDistinctItems()
    On Error GoTo Handler
    Dim database As ADODB.Connection
    Set database = OpenDatabaseConnection(SomeDSN)

    If Not database Is Nothing Then
        Dim query As ADODB.Command
        Set query = New ADODB.Command
        'Build the command to pass to the data source.
        With query
            .ActiveConnection = database
            .CommandText = "SELECT DISTINCT Item FROM Table"
            .CommandType = adCmdText
        End With
        Dim records As ADODB.Recordset
        'Execute the command to retrieve the recordset.
        Set records = query.Execute()

        Do While Not records.EOF
            Debug.Print records("Item")
            records.MoveNext
        Loop
    End If
CleanExit:
    If Not records Is Nothing Then records.Close
    If Not database Is Nothing And database.State = adStateOpen Then
        database.Close
    End If
    Exit Sub
Handler:
    Debug.Print "Error " & Err.Number & ": " & Err.Description
    Resume CleanExit
End Sub
```

Note that commands sent to the data source are **vulnerable to SQL injection**, either intentional or unintentional. In general, queries should not be created by concatenating user input of any kind. Instead, they should be parameterized (see Creating parameterized commands).

Section 20.4: Executing non-scalar functions

ADO connections can be used to perform pretty much any database function that the provider supports via SQL. In this case it isn't always necessary to use the `Recordset` returned by the `Execute` function, although it can be useful for obtaining key assignments after INSERT statements with @@Identity or similar SQL commands. Note that the example below uses the `OpenDatabaseConnection` function from the Making a connection to a data source example for the purpose of brevity.

```
Public Sub UpdateTheFoos()
    On Error GoTo Handler
    Dim database As ADODB.Connection
    Set database = OpenDatabaseConnection(SomeDSN)

    If Not database Is Nothing Then
        Dim update As ADODB.Command
        Set update = New ADODB.Command
        'Build the command to pass to the data source.
        With update
            .ActiveConnection = database
            .CommandText = "UPDATE Table SET Foo = 42 WHERE Bar IS NULL"
            .CommandType = adCmdText
            .Execute        'We don't need the return from the DB, so ignore it.
        End With
    End If
CleanExit:
    If Not database Is Nothing And database.State = adStateOpen Then
        database.Close
    End If
    Exit Sub
Handler:
    Debug.Print "Error " & Err.Number & ": " & Err.Description
    Resume CleanExit
End Sub
```

Note that commands sent to the data source are **vulnerable to SQL injection**, either intentional or unintentional. In general, SQL statements should not be created by concatenating user input of any kind. Instead, they should be parameterized (see Creating parameterized commands).

Chapter 21: Concatenating strings

Section 21.1: Concatenate an array of strings using the Join function

```
'Declare and assign a string array
Dim widgetNames(2) As String
widgetNames(0) = "foo"
widgetNames(1) = "bar"
widgetNames(2) = "fizz"

'Concatenate with Join and separate each element with a 3-character string
concatenatedString = VBA.Strings.Join(widgetNames, " > ")
'concatenatedString = "foo > bar > fizz"

'Concatenate with Join and separate each element with a zero-width string
concatenatedString = VBA.Strings.Join(widgetNames, vbNullString)
'concatenatedString = "foobarfizz"
```

Section 21.2: Concatenate strings using the & operator

```
Const string1 As String = "foo"
Const string2 As String = "bar"
Const string3 As String = "fizz"
Dim concatenatedString As String

'Concatenate two strings
concatenatedString = string1 & string2
'concatenatedString = "foobar"

'Concatenate three strings
concatenatedString = string1 & string2 & string3
'concatenatedString = "foobarfizz"
```

Chapter 22: Assigning strings with repeated characters

Section 22.1: Use the String function to assign a string with n repeated characters

```
Dim lineOfHyphens As String
'Assign a string with 80 repeated hyphens
lineOfHyphens = String$(80, "-")
```

Section 22.2: Use the String and Space functions to assign an n-character string

```
Dim stringOfSpaces As String

'Assign a string with 255 repeated spaces using Space$
stringOfSpaces = Space$(255)

'Assign a string with 255 repeated spaces using String$
stringOfSpaces = String$(255, " ")
```

Chapter 23: Scripting.Dictionary object

You must add Microsoft Scripting Runtime to the VBA project through the VBE's Tools → References command in order to implement early binding of the Scripting Dictionary object. This library reference is carried with the project; it does not have to be re-referenced when the VBA project is distributed and run on another computer.

Section 23.1: Properties and Methods

A Scripting Dictionary object stores information in Key/Item pairs. The Keys must be unique and not an array but the associated Items can be repeated (their uniqueness is held by the companion Key) and can be of any type of variant or object.

A dictionary can be thought of as a two field in-memory database with a primary unique index on the first 'field' (the *Key*). This unique index on the Keys property allows very fast 'lookups' to retrieve a Key's associated Item value.

Properties

name	read/write	type	description
CompareMode	*read / write*	CompareMode constant	Setting the CompareMode can only be performed on an empty dictionary. Accepted values are 0 (vbBinaryCompare), 1 (vbTextCompare), 2 (vbDatabaseCompare).
Count	*read only*	unsigned long integer	A one-based count of the key/item pairs in the scripting dictionary object.
Key	*read / write*	non-array variant	Each individual unique key in the dictionary.
Item(*Key*)	*read / write*	any variant	Default property. Each individual item associated with a key in the dictionary. Note that attempting to retrieve an item with a key that does not exist in the dictionary will *implicitly add* the passed key.

Methods

name	description
Add(*Key*,*Item*)	Adds a new Key and Item to the dictionary. The new key must not exist in the dictionary's current Keys collection but an item can be repeated among many unique keys.
Exists(*Key*)	Boolean test to determine if a Key already exists in the dictionary.
Keys	Returns the array or collection of unique keys.
Items	Returns the array or collection of associated items.
Remove(*Key*)	Removes an individual dictionary key and its associated item.
RemoveAll	Clears all of a dictionary object's keys and items.

Sample Code

```
'Populate, enumerate, locate and remove entries in a dictionary that was created
'with late binding
Sub iterateDictionaryLate()
    Dim k As Variant, dict As Object

    Set dict = CreateObject("Scripting.Dictionary")
    dict.CompareMode = vbTextCompare          'non-case sensitive compare model

    'populate the dictionary
    dict.Add Key:="Red", Item:="Balloon"
    dict.Add Key:="Green", Item:="Balloon"
    dict.Add Key:="Blue", Item:="Balloon"

    'iterate through the keys
    For Each k In dict.Keys
```

```
        Debug.Print k & " - " & dict.Item(k)
    Next k

    'locate the Item for Green
    Debug.Print dict.Item("Green")

    'remove key/item pairs from the dictionary
    dict.Remove "blue"      'remove individual key/item pair by key
    dict.RemoveAll          'remove all remaining key/item pairs

End Sub

'Populate, enumerate, locate and remove entries in a dictionary that was created
'with early binding (see Remarks)
Sub iterateDictionaryEarly()
    Dim d As Long, k As Variant
    Dim dict As New Scripting.Dictionary

    dict.CompareMode = vbTextCompare          'non-case sensitive compare model

    'populate the dictionary
    dict.Add Key:="Red", Item:="Balloon"
    dict.Add Key:="Green", Item:="Balloon"
    dict.Add Key:="Blue", Item:="Balloon"
    dict.Add Key:="White", Item:="Balloon"

    'iterate through the keys
    For Each k In dict.Keys
        Debug.Print k & " - " & dict.Item(k)
    Next k

    'iterate through the keys by the count
    For d = 0 To dict.Count - 1
        Debug.Print dict.Keys(d) & " - " & dict.Items(d)
    Next d

    'iterate through the keys by the boundaries of the keys collection
    For d = LBound(dict.Keys) To UBound(dict.Keys)
        Debug.Print dict.Keys(d) & " - " & dict.Items(d)
    Next d

    'locate the Item for Green
    Debug.Print dict.Item("Green")
    'locate the Item for the first key
    Debug.Print dict.Item(dict.Keys(0))
    'locate the Item for the last key
    Debug.Print dict.Item(dict.Keys(UBound(dict.Keys)))

    'remove key/item pairs from the dictionary
    dict.Remove "blue"                          'remove individual key/item pair by key
    dict.Remove dict.Keys(0)                    'remove first key/item by index position
    dict.Remove dict.Keys(UBound(dict.Keys))    'remove last key/item by index position
    dict.RemoveAll                              'remove all remaining key/item pairs

End Sub
```

Chapter 24: VBA Option Keyword

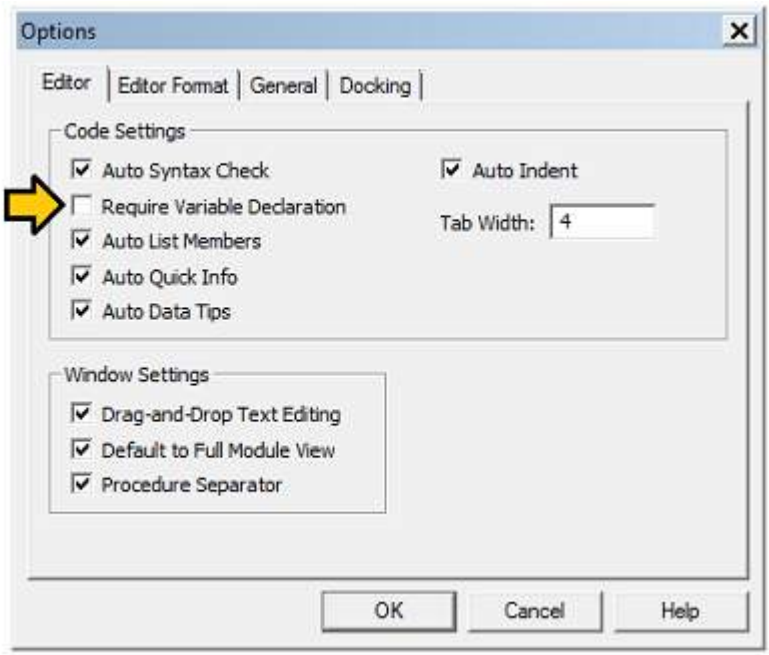
Option	Detail
Explicit	*Require variable declaration* in the module it's specified in (ideally all of them); with this option specified, using an undeclared (/mispelled) variable becomes a compilation error.
Compare Text	Makes the module's string comparisons be case-insensitive, based on system locale, prioritizing alphabetical equivalency (e.g. "a" = "A").
Compare Binary	Default string comparison mode. Makes the module's string comparisons be case sensitive, comparing strings using the binary representation / numeric value of each character (e.g. ASCII).
Compare Database	(MS-Access only) Makes the module's string comparisons work the way they would in an SQL statement.
Private Module	Prevents the module's `Public` member from being accessed from outside of the project that the module resides in, effectively hiding procedures from the host application (i.e. not available to use as macros or user-defined functions).
Option Base 0	Default setting. Sets the implicit array lower bound to `0` in a module. When an array is declared without an explicit lower boundary value, `0` will be used.
Option Base 1	Sets the implicit array lower bound to `1` in a module. When an array is declared without an explicit lower boundary value, `1` will be used.

Section 24.1: Option Explicit

It is deemed best practice to always use `Option Explicit` in VBA as it forces the developer to declare all their variables before use. This has other benefits too, such as auto-capitalization for declared variable names and IntelliSense.

```
Option Explicit

Sub OptionExplicit()
    Dim a As Integer
    a = 5
    b = 10 '// Causes compile error as 'b' is not declared
End Sub
```

Setting **Require Variable Declaration** within the VBE's Tools ▸ Options ▸ Editor property page will put the **Option Explicit** statement at the top of each newly created code sheet.



This will avoid silly coding mistakes like misspellings as well as influencing you to use the correct variable type in the variable declaration. (Some more examples are given at ALWAYS Use "Option Explicit".)

Section 24.2: Option Base {0 | 1}

`Option Base` is used to declare the default lower bound of **array** elements. It is declared at module level and is valid only for the current module.

By default (and thus if no Option Base is specified), the Base is 0. Which means that the first element of any array declared in the module has an index of 0.

If `Option Base 1` is specified, the first array element has the index 1

Example in Base 0 :

```
Option Base 0

Sub BaseZero()

    Dim myStrings As Variant

    ' Create an array out of the Variant, having 3 fruits elements
    myStrings = Array("Apple", "Orange", "Peach")

    Debug.Print LBound(myStrings) ' This Prints "0"
    Debug.Print UBound(myStrings) ' This print "2", because we have 3 elements beginning at 0 ->
0,1,2

    For i = 0 To UBound(myStrings)

        Debug.Print myStrings(i) ' This will print "Apple", then "Orange", then "Peach"

    Next i

End Sub
```

Same Example with Base 1

```
Option Base 1

Sub BaseOne()

    Dim myStrings As Variant

    ' Create an array out of the Variant, having 3 fruits elements
    myStrings = Array("Apple", "Orange", "Peach")

    Debug.Print LBound(myStrings) ' This Prints "1"
    Debug.Print UBound(myStrings) ' This print "3", because we have 3 elements beginning at 1 ->
1,2,3

    For i = 0 To UBound(myStrings)

        Debug.Print myStrings(i) ' This triggers an error 9 "Subscript out of range"

    Next i

End Sub
```

The second example generated a Subscript out of range (Error 9) at the first loop stage because an attempt to

access the index 0 of the array was made, and this index doesn't exists as the module is declared with `Base 1`

The correct code with Base 1 is :

```
    For i = 1 To UBound(myStrings)

        Debug.Print myStrings(i) ' This will print "Apple", then "Orange", then "Peach"

    Next i
```

It should be noted that the Split function **always** creates an array with a zero-based element index regardless of any `Option Base` setting. Examples on how to use the **Split** function can be found here

> Split Function
>
> Returns a zero-based, one-dimensional array containing a specified number of substrings.

In Excel, the `Range.Value` and `Range.Formula` properties for a multi-celled range *always* returns a 1-based 2D Variant array.

Likewise, in ADO, the `Recordset.GetRows` method *always* returns a 1-based 2D array.

One recommended 'best practice' is to always use the LBound and UBound functions to determine the extents of an array.

```
'for single dimensioned array
Debug.Print LBound(arr) & ":" & UBound(arr)
Dim i As Long
For i = LBound(arr) To UBound(arr)
    Debug.Print arr(i)
Next i

'for two dimensioned array
Debug.Print LBound(arr, 1) & ":" & UBound(arr, 1)
Debug.Print LBound(arr, 2) & ":" & UBound(arr, 2)
Dim i As long, j As Long
For i = LBound(arr, 1) To UBound(arr, 1)
    For j = LBound(arr, 2) To UBound(arr, 2)
         Debug.Print arr(i, j)
    Next j
Next i
```

The `Option Base 1` must be at the top of every code module where an array is created or re-dimensioned if arrays are to be consistently created with an lower boundary of 1.

Section 24.3: Option Compare {Binary | Text | Database}

Option Compare Binary

Binary comparison makes all checks for string equality within a module/class case *sensitive*. Technically, with this option, string comparisons are performed using sort order of the binary representations of each character.

> A < B < E < Z < a < b < e < z

If no Option Compare is specified in a module, Binary is used by default.

```
Option Compare Binary
```

```
Sub CompareBinary()

    Dim foo As String
    Dim bar As String

    '// Case sensitive
    foo = "abc"
    bar = "ABC"

    Debug.Print (foo = bar) '// Prints "False"

    '// Still differentiates accented characters
    foo = "ábc"
    bar = "abc"

    Debug.Print (foo = bar) '// Prints "False"

    '// "b" (Chr 98) is greater than "a" (Chr 97)
    foo = "a"
    bar = "b"

    Debug.Print (bar > foo) '// Prints "True"

    '// "b" (Chr 98) is NOT greater than "á" (Chr 225)
    foo = "á"
    bar = "b"

    Debug.Print (bar > foo) '// Prints "False"

End Sub
```

Option Compare Text

Option Compare Text makes all string comparisons within a module/class use a case *insensitive* comparison.

> (A | a) < (B | b) < (Z | z)

```
Option Compare Text

Sub CompareText()

    Dim foo As String
    Dim bar As String

    '// Case insensitivity
    foo = "abc"
    bar = "ABC"

    Debug.Print (foo = bar) '// Prints "True"

    '// Still differentiates accented characters
    foo = "ábc"
    bar = "abc"

    Debug.Print (foo = bar) '// Prints "False"

    '// "b" still comes after "a" or "á"
    foo = "á"
```

```
    bar = "b"

    Debug.Print (bar > foo) '// Prints "True"

End Sub
```

Option Compare Database

Option Compare Database is only available within MS Access. It sets the module/class to use the current database settings to determine whether to use Text or Binary mode.

Note: The use of this setting is discouraged unless the module is used for writing custom Access UDFs (User defined functions) that should treat text comparisons in the same manner as SQL queries in that database.

Chapter 25: Date Time Manipulation

Section 25.1: Calendar

VBA supports 2 calendars : Gregorian and Hijri

The `Calendar` property is used to modify or display the current calendar.

The 2 values for the Calendar are:

Value	Constant	Description
0	vbCalGreg	Gregorian calendar (default)
1	vbCalHijri	Hijri calendar

Example

```
Sub CalendarExample()
    'Cache the current setting.
    Dim Cached As Integer
    Cached = Calendar

    ' Dates in Gregorian Calendar
    Calendar = vbCalGreg
    Dim Sample As Date
    'Create sample date of 2016-07-28
    Sample = DateSerial(2016, 7, 28)

    Debug.Print "Current Calendar : " & Calendar
    Debug.Print "SampleDate = " & Format$(Sample, "yyyy-mm-dd")

    ' Date in Hijri Calendar
    Calendar = vbCalHijri
    Debug.Print "Current Calendar : " & Calendar
    Debug.Print "SampleDate = " & Format$(Sample, "yyyy-mm-dd")

    'Reset VBA to cached value.
    Cached = Calendar
End Sub
```

This Sub prints the following ;

```
Current Calendar : 0
SampleDate = 2016-07-28
Current Calendar : 1
SampleDate = 1437-10-23
```

Section 25.2: Base functions

Retrieve System DateTime

VBA supports 3 built-in functions to retrieve the date and/or time from the system's clock.

Function	Return Type	Return Value
Now	Date	Returns the current date and time
Date	Date	Returns the date portion of the current date and time
Time	Date	Returns the time portion of the current date and time

```
Sub DateTimeExample()
```

```
    ' Note : EU system with default date format DD/MM/YYYY

    Debug.Print Now    ' prints 28/07/2016 10:16:01 (output below assumes this date and time)
    Debug.Print Date   ' prints 28/07/2016
    Debug.Print Time   ' prints 10:16:01

    ' Apply a custom format to the current date or time
    Debug.Print Format$(Now, "dd mmmm yyyy hh:nn")   ' prints 28 July 2016 10:16
    Debug.Print Format$(Date, "yyyy-mm-dd")          ' prints 2016-07-28
    Debug.Print Format$(Time, "hh") & " hour " & _
                Format$(Time, "nn") & " min " & _
                Format$(Time, "ss") & " sec "        ' prints 10 hour 16 min 01 sec

End Sub
```

Timer Function

The `Timer` function returns a Single representing the number of seconds elapsed since midnight. The precision is one hundredth of a second.

```
Sub TimerExample()

    Debug.Print Time    ' prints 10:36:31  (time at execution)
    Debug.Print Timer   ' prints 38191,13  (seconds since midnight)

End Sub
```

Because `Now` and `Time` functions are only precise to seconds, `Timer` offers a convenient way to increase accuracy of time measurement:

```
Sub GetBenchmark()

    Dim StartTime  As Single
    StartTime = Timer       'Store the current Time

    Dim i As Long
    Dim temp As String
    For i = 1 To 1000000    'See how long it takes Left$ to execute 1,000,000 times
        temp = Left$("Text", 2)
    Next i

    Dim Elapsed As Single
    Elapsed = Timer - StartTime
    Debug.Print "Code completed in " & CInt(Elapsed * 1000) & " ms"

End Sub
```

IsDate()

IsDate() tests whether an expression is a valid date or not. Returns a `Boolean`.

```
Sub IsDateExamples()

    Dim anything As Variant

    anything = "September 11, 2001"
```

```
    Debug.Print IsDate(anything)    'Prints True

    anything = #9/11/2001#

    Debug.Print IsDate(anything)    'Prints True

    anything = "just a string"

    Debug.Print IsDate(anything)    'Prints False

    anything = vbNull

    Debug.Print IsDate(anything)    'Prints False

End Sub
```

Section 25.3: Extraction functions

These functions take a `Variant` that can be cast to a `Date` as a parameter and return an `Integer` representing a portion of a date or time. If the parameter can not be cast to a `Date`, it will result in a run-time error 13: Type mismatch.

Function	Description	Returned value
Year()	Returns the year portion of the date argument.	Integer (100 to 9999)
Month()	Returns the month portion of the date argument.	Integer (1 to 12)
Day()	Returns the day portion of the date argument.	Integer (1 to 31)
WeekDay()	Returns the day of the week of the date argument. Accepts an optional second argument defininig the first day of the week	Integer (1 to 7)
Hour()	Returns the hour portion of the date argument.	Integer (0 to 23)
Minute()	Returns the minute portion of the date argument.	Integer (0 to 59)
Second()	Returns the second portion of the date argument.	Integer (0 to 59)

Examples:

```
Sub ExtractionExamples()

    Dim MyDate As Date

    MyDate = DateSerial(2016, 7, 28) + TimeSerial(12, 34, 56)

    Debug.Print Format$(MyDate, "yyyy-mm-dd hh:nn:ss") ' prints 2016-07-28 12:34:56

    Debug.Print Year(MyDate)                           ' prints 2016
    Debug.Print Month(MyDate)                          ' prints 7
    Debug.Print Day(MyDate)                            ' prints 28
    Debug.Print Hour(MyDate)                           ' prints 12
    Debug.Print Minute(MyDate)                         ' prints 34
    Debug.Print Second(MyDate)                         ' prints 56

    Debug.Print Weekday(MyDate)                        ' prints 5
    'Varies by locale - i.e. will print 4 in the EU and 5 in the US
    Debug.Print Weekday(MyDate, vbUseSystemDayOfWeek)
    Debug.Print Weekday(MyDate, vbMonday)              ' prints 4
    Debug.Print Weekday(MyDate, vbSunday)              ' prints 5

End Sub
```

DatePart() Function

`DatePart()` is also a function returning a portion of a date, but works differently and allow more possibilities than the functions above. It can for instance return the Quarter of the year or the Week of the year.

Syntax:

```
DatePart ( interval, date  [, firstdayofweek] [, firstweekofyear] )
```

interval argument can be :

Interval	Description
"yyyy"	Year (100 to 9999)
"y"	Day of the year (1 to 366)
"m"	Month (1 to 12)
"q"	Quarter (1 to 4)
"ww"	Week (1 to 53)
"w"	Day of the week (1 to 7)
"d"	Day of the month (1 to 31)
"h"	Hour (0 to 23)
"n"	Minute (0 to 59)
"s"	Second (0 to 59)

firstdayofweek is optional. it is a constant that specifies the first day of the week. If not specified, `vbSunday` is assumed.

firstweekofyear is optional. it is a constant that specifies the first week of the year. If not specified, the first week is assumed to be the week in which January 1 occurs.

Examples:

```
Sub DatePartExample()

    Dim MyDate As Date

    MyDate = DateSerial(2016, 7, 28) + TimeSerial(12, 34, 56)

    Debug.Print Format$(MyDate, "yyyy-mm-dd hh:nn:ss") ' prints 2016-07-28 12:34:56

    Debug.Print DatePart("yyyy", MyDate)              ' prints 2016
    Debug.Print DatePart("y", MyDate)                 ' prints 210
    Debug.Print DatePart("h", MyDate)                 ' prints 12
    Debug.Print DatePart("Q", MyDate)                 ' prints 3
    Debug.Print DatePart("w", MyDate)                 ' prints 5
    Debug.Print DatePart("ww", MyDate)                ' prints 31

End Sub
```

Section 25.4: Calculation functions

DateDiff()

`DateDiff()` returns a `Long` representing the number of time intervals between two specified dates.

Syntax

```
DateDiff ( interval, date1, date2  [, firstdayofweek] [, firstweekofyear] )
```

- *interval* can be any of the intervals defined in the `DatePart()` function

- *date1* and *date2* are the two dates you want to use in the calculation
- *firstdayofweek* and *firstweekofyear* are optional. Refer to `DatePart()` function for explanations

Examples

```
Sub DateDiffExamples()

    ' Check to see if 2016 is a leap year.
    Dim NumberOfDays As Long
    NumberOfDays = DateDiff("d", #1/1/2016#, #1/1/2017#)

    If NumberOfDays = 366 Then
        Debug.Print "2016 is a leap year."              'This will output.
    End If

    ' Number of seconds in a day
    Dim StartTime As Date
    Dim EndTime As Date
    StartTime = TimeSerial(0, 0, 0)
    EndTime = TimeSerial(24, 0, 0)
    Debug.Print DateDiff("s", StartTime, EndTime)       'prints 86400

End Sub
```

DateAdd()

`DateAdd()` returns a `Date` to which a specified date or time interval has been added.

Syntax

```
DateAdd ( interval, number, date  )
```

- *interval* can be any of the intervals defined in the `DatePart()` function
- *number* Numeric expression that is the number of intervals you want to add. It can be positive (to get dates in the future) or negative (to get dates in the past).
- *date* is a `Date` or literal representing date to which the interval is added

Examples :

```
Sub DateAddExamples()

    Dim Sample As Date
    'Create sample date and time of 2016-07-28 12:34:56
    Sample = DateSerial(2016, 7, 28) + TimeSerial(12, 34, 56)

    ' Date 5 months previously (prints 2016-02-28):
    Debug.Print Format$(DateAdd("m", -5, Sample), "yyyy-mm-dd")

    ' Date 10 months previously (prints 2015-09-28):
    Debug.Print Format$(DateAdd("m", -10, Sample), "yyyy-mm-dd")

    ' Date in 8 months (prints 2017-03-28):
    Debug.Print Format$(DateAdd("m", 8, Sample), "yyyy-mm-dd")

    ' Date/Time 18 hours previously (prints 2016-07-27 18:34:56):
    Debug.Print Format$(DateAdd("h", -18, Sample), "yyyy-mm-dd hh:nn:ss")

    ' Date/Time in 36 hours (prints 2016-07-30 00:34:56):
    Debug.Print Format$(DateAdd("h", 36, Sample), "yyyy-mm-dd hh:nn:ss")
```

```
End Sub
```

Section 25.5: Conversion and Creation

CDate()

`CDate()` converts something from any datatype to a `Date` datatype

```
Sub CDateExamples()

    Dim sample As Date

    ' Converts a String representing a date and time to a Date
    sample = CDate("September 11, 2001 12:34")
    Debug.Print Format$(sample, "yyyy-mm-dd hh:nn:ss")      ' prints 2001-09-11 12:34:00

    ' Converts a String containing a date to a Date
    sample = CDate("September 11, 2001")
    Debug.Print Format$(sample, "yyyy-mm-dd hh:nn:ss")      ' prints 2001-09-11 00:00:00

    ' Converts a String containing a time to a Date
    sample = CDate("12:34:56")
    Debug.Print Hour(sample)                        ' prints 12
    Debug.Print Minute(sample)                      ' prints 34
    Debug.Print Second(sample)                      ' prints 56

    ' Find the 10000th day from the epoch date of 1899-12-31
    sample = CDate(10000)
    Debug.Print Format$(sample, "yyyy-mm-dd")       ' prints 1927-05-18

End Sub
```

Note that VBA also has a loosely typed `CVDate()` that functions in the same way as the `CDate()` function other than returning a date typed `Variant` instead of a strongly typed `Date`. The `CDate()` version should be preferred when passing to a `Date` parameter or assigning to a `Date` variable, and the `CVDate()` version should be preferred when when passing to a `Variant` parameter or assigning to a `Variant` variable. This avoids implicit type casting.

DateSerial()

`DateSerial()` function is used to create a date. It returns a `Date` for a specified year, month, and day.

Syntax:

```
DateSerial ( year, month, day )
```

With year, month and day arguments being valid Integers (Year from 100 to 9999, Month from 1 to 12, Day from 1 to 31).

Examples

```
Sub DateSerialExamples()

    ' Build a specific date
    Dim sample As Date
    sample = DateSerial(2001, 9, 11)
    Debug.Print Format$(sample, "yyyy-mm-dd")                   ' prints 2001-09-11
```

```
    ' Find the first day of the month for a date.
    sample = DateSerial(Year(sample), Month(sample), 1)
    Debug.Print Format$(sample, "yyyy-mm-dd")                    ' prints 2001-09-11

    ' Find the last day of the previous month.
    sample = DateSerial(Year(sample), Month(sample), 1) - 1
    Debug.Print Format$(sample, "yyyy-mm-dd")                    ' prints 2001-09-11

End Sub
```

Note that `DateSerial()` will accept "invalid" dates and calculate a valid date from it. This can be used creatively for good:

Positive Example

```
Sub GoodDateSerialExample()

    'Calculate 45 days from today
    Dim today As Date
    today = DateSerial (2001, 9, 11)
    Dim futureDate As Date
    futureDate = DateSerial(Year(today), Month(today), Day(today) + 45)
    Debug.Print Format$(futureDate, "yyyy-mm-dd")             'prints 2009-10-26

End Sub
```

However, it is more likely to cause grief when attempting to create a date from unvalidated user input:

Negative Example

```
Sub BadDateSerialExample()

    'Allow user to enter unvalidate date information
    Dim myYear As Long
    myYear = InputBox("Enter Year")
            'Assume user enters 2009
    Dim myMonth As Long
    myMonth = InputBox("Enter Month")
            'Assume user enters 2
    Dim myDay As Long
    myDay = InputBox("Enter Day")
            'Assume user enters 31
    Debug.Print Format$(DateSerial(myYear, myMonth, myDay), "yyyy-mm-dd")
            'prints  2009-03-03

End Sub
```

Chapter 26: Creating a Custom Class

Section 26.1: Adding a Property to a Class

A `Property` procedure is a series of statement that retrieves or modifies a custom property on a module.

There are three types of property accessors:

1. A `Get` procedure that returns the value of a property.
2. A `Let` procedure that assigns a (non-`Object`) value to an object.
3. A `Set` procedure that assigns an `Object` reference.

Property accessors are often defined in pairs, using both a `Get` and `Let`/`Set` for each property. A property with only a `Get` procedure would be read-only, while a property with only a `Let`/`Set` procedure would be write-only.

In the following example, four property accessors are defined for the `DateRange` class:

1. `StartDate` (*read/write*). Date value representing the earlier date in a range. Each procedure uses the value of the module variable, `mStartDate`.
2. `EndDate` (*read/write*). Date value representing the later date in a range. Each procedure uses the value of the module variable, `mEndDate`.
3. `DaysBetween` (*read-only*). Calculated Integer value representing the number of days between the two dates. Because there is only a `Get` procedure, this property cannot be modified directly.
4. `RangeToCopy` (*write-only*). A `Set` procedure used to copy the values of an existing `DateRange` object.

```
Private mStartDate As Date                ' Module variable to hold the starting date
Private mEndDate As Date                  ' Module variable to hold the ending date

' Return the current value of the starting date
Public Property Get StartDate() As Date
    StartDate = mStartDate
End Property

' Set the starting date value. Note that two methods have the name StartDate
Public Property Let StartDate(ByVal NewValue As Date)
    mStartDate = NewValue
End Property

' Same thing, but for the ending date
Public Property Get EndDate() As Date
    EndDate = mEndDate
End Property

Public Property Let EndDate(ByVal NewValue As Date)
    mEndDate = NewValue
End Property

' Read-only property that returns the number of days between the two dates
Public Property Get DaysBetween() As Integer
    DaysBetween = DateDiff("d", mStartDate, mEndDate)
End Function

' Write-only property that passes an object reference of a range to clone
Public Property Set RangeToCopy(ByRef ExistingRange As DateRange)

Me.StartDate = ExistingRange.StartDate
Me.EndDate = ExistingRange.EndDate
```

```
End Property
```

Section 26.2: Class module scope, instancing and re-use

By default, a new class module is a Private class, so it is *only* available for instantiation and use within the VBProject in which it is defined. You can declare, instantiate and use the class anywhere in the *same* project:

```
'Class List has Instancing set to Private
'In any other module in the SAME project, you can use:

Dim items As List
Set items = New List
```

But often you'll write classes that you'd like to use in other projects *without* copying the module between projects. If you define a class called `List` in `ProjectA`, and want to use that class in `ProjectB`, then you'll need to perform 4 actions:

1. Change the instancing property of the `List` class in `ProjectA` in the Properties window, from **`Private`** to `PublicNotCreatable`
2. Create a public "factory" function in `ProjectA` that creates and returns an instance of a `List` class. Typically the factory function would include arguments for the initialization of the class instance. The factory function is required because the class can be used by `ProjectB` but `ProjectB` cannot directly create an instance of `ProjectA`'s class.

```
Public Function CreateList(ParamArray values() As Variant) As List
    Dim tempList As List
    Dim itemCounter As Long
    Set tempList = New List
    For itemCounter = LBound(values) to UBound(values)
        tempList.Add values(itemCounter)
    Next itemCounter
    Set CreateList = tempList
End Function
```

3. In `ProjectB` add a reference to `ProjectA` using the `Tools..References...` menu.
4. In `ProjectB`, declare a variable and assign it an instance of `List` using the factory function from `ProjectA`

```
Dim items As ProjectA.List
Set items = ProjectA.CreateList("foo","bar")

'Use the items list methods and properties
items.Add "fizz"
Debug.Print items.ToString()
'Destroy the items object
Set items = Nothing
```

Section 26.3: Adding Functionality to a Class

Any public **`Sub`**, **`Function`**, or **`Property`** inside a class module can be called by preceding the call with an object reference:

```
Object.Procedure
```

In a `DateRange` class, a `Sub` could be used to add a number of days to the end date:

```
Public Sub AddDays(ByVal NoDays As Integer)
    mEndDate = mEndDate + NoDays
End Sub
```

A `Function` could return the last day of the next month-end (note that `GetFirstDayOfMonth` would not be visible outside the class because it is private):

```
Public Function GetNextMonthEndDate() As Date
    GetNextMonthEndDate = DateAdd("m", 1, GetFirstDayOfMonth())
End Function

Private Function GetFirstDayOfMonth() As Date
    GetFirstDayOfMonth = DateAdd("d", -DatePart("d", mEndDate), mEndDate)
End Function
```

Procedures can accept arguments of any type, including references to objects of the class being defined.

The following example tests whether the current `DateRange` object has a starting date and ending date that includes the starting and ending date of another `DateRange` object.

```
Public Function ContainsRange(ByRef TheRange As DateRange) As Boolean
    ContainsRange = TheRange.StartDate >= Me.StartDate And TheRange.EndDate <= Me.EndDate
End Function
```

Note the use of the `Me` notation as a way to access the value of the object running the code.

Chapter 27: Events

Section 27.1: Sources and Handlers

What are events?

VBA is *event-driven*: VBA code runs in response to events raised by the host application or the host document - understanding events is fundamental to understanding VBA.

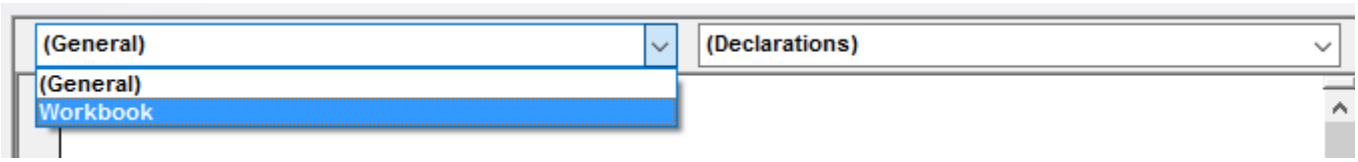
APIs often expose objects that raise a number of *events* in response to various states. For example an `Excel.Application` object raises an event whenever a new workbook is created, opened, activated, or closed. Or whenever a worksheet gets calculated. Or just before a file is saved. Or immediately after. A button on a form raises a `Click` event when the user clicks it, the user form itself raises an event just after it's activated, and another just before it's closed.

From an API perspective, events are *extension points*: the client code can chose to implement code that *handles* these events, and execute custom code whenever these events are fired: that's how you can execute your custom code automatically every time the selection changes on any worksheet - by handling the event that gets fired when the selection changes on any worksheet.

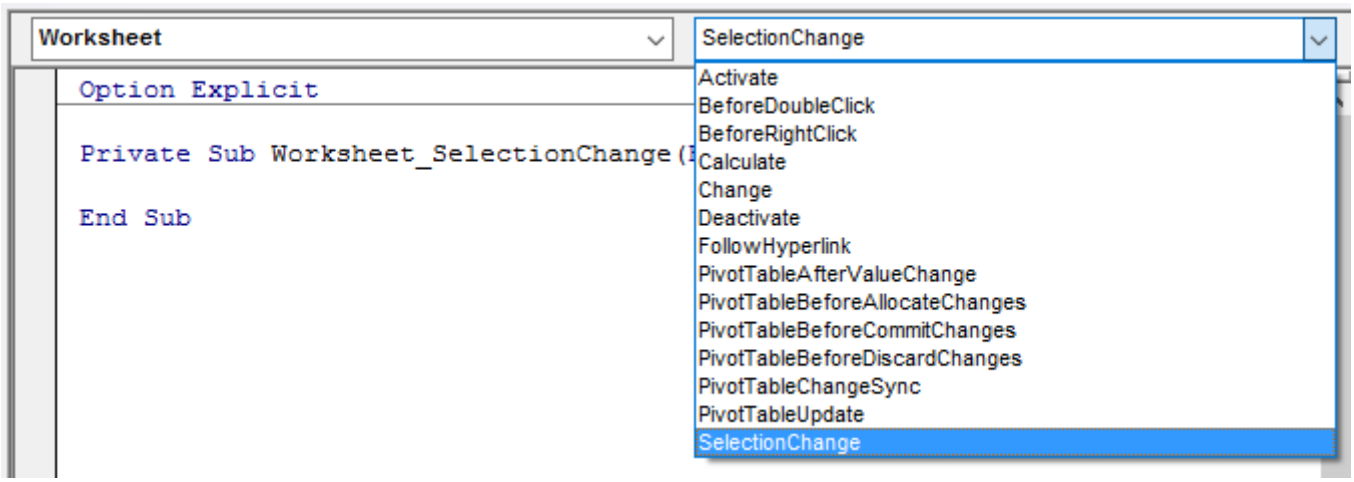
An object that exposes events is an *event source*. A method that handles an event is a *handler*.

Handlers

VBA document modules (e.g. `ThisDocument`, `ThisWorkbook`, `Sheet1`, etc.) and `UserForm` modules are *class modules* that *implement* special interfaces that expose a number of *events*. You can browse these interfaces in the left-side dropdown at the top of the code pane:



The right-side dropdown lists the members of the interface selected in the left-side dropdown:



The VBE automatically generates an event handler stub when an item is selected on the right-side list, or navigates there if the handler exists.

You can define a module-scoped `WithEvents` variable in any module:

```
Private WithEvents Foo As Workbook
Private WithEvents Bar As Worksheet
```

Each `WithEvents` declaration becomes available to select from the left-side dropdown. When an event is selected in the right-side dropdown, the VBE generates an event handler stub named after the `WithEvents` object and the name of the event, joined with an underscore:

```
Private WithEvents Foo As Workbook
Private WithEvents Bar As Worksheet

Private Sub Foo_Open()

End Sub

Private Sub Bar_SelectionChange(ByVal Target As Range)

End Sub
```

Only types that expose at least one event can be used with `WithEvents`, and `WithEvents` declarations cannot be assigned a reference on-the-spot with the `New` keyword. This code is illegal:

```
Private WithEvents Foo As New Workbook 'illegal
```

The object reference must be `Set` explicitly; in a class module, a good place to do that is often in the `Class_Initialize` handler, because then the class handles that object's events for as long as its instance exists.

Sources

Any class module (or document module, or user form) can be an event source. Use the `Event` keyword to define the *signature* for the event, in the *declarations section* of the module:

```
Public Event SomethingHappened(ByVal something As String)
```

The signature of the event determines how the event is raised, and what the event handlers will look like.

Events can only be *raised* within the class they're defined in - client code can only *handle* them. Events are raised with the `RaiseEvent` keyword; the event's arguments are provided at that point:

```
Public Sub DoSomething()
    RaiseEvent SomethingHappened("hello")
End Sub
```

Without code that handles the `SomethingHappened` event, running the `DoSomething` procedure will still raise the event, but nothing will happen. Assuming the event source is the above code in a class named `Something`, this code in `ThisWorkbook` would show a message box saying "hello" whenever `test.DoSomething` gets called:

```
Private WithEvents test As Something

Private Sub Workbook_Open()
    Set test = New Something
    test.DoSomething
End Sub

Private Sub test_SomethingHappened(ByVal bar As String)
'this procedure runs whenever 'test' raises the 'SomethingHappened' event
```

```
    MsgBox bar
End Sub
```

Section 27.2: Passing data back to the event source

Using parameters passed by reference

An event may define a `ByRef` parameter meant to be returned to the caller:

```
Public Event BeforeSomething(ByRef cancel As Boolean)
Public Event AfterSomething()

Public Sub DoSomething()
    Dim cancel As Boolean
    RaiseEvent BeforeSomething(cancel)
    If cancel Then Exit Sub

    'todo: actually do something

    RaiseEvent AfterSomething
End Sub
```

If the `BeforeSomething` event has a handler that sets its `cancel` parameter to `True`, then when execution returns from the handler, `cancel` will be `True` and `AfterSomething` will never be raised.

```
Private WithEvents foo As Something

Private Sub foo_BeforeSomething(ByRef cancel As Boolean)
    cancel = MsgBox("Cancel?", vbYesNo) = vbYes
End Sub

Private Sub foo_AfterSomething()
    MsgBox "Didn't cancel!"
End Sub
```

Assuming the `foo` object reference is assigned somewhere, when `foo.DoSomething` runs, a message box prompts whether to cancel, and a second message box says "didn't cancel" only when [No] was selected.

Using mutable objects

You could also pass a copy of a mutable object `ByVal`, and let handlers modify that object's properties; the caller can then read the modified property values and act accordingly.

```
'class module ReturnBoolean
Option Explicit
Private encapsulated As Boolean

Public Property Get ReturnValue() As Boolean
'Attribute ReturnValue.VB_UserMemId = 0
    ReturnValue = encapsulated
End Property

Public Property Let ReturnValue(ByVal value As Boolean)
    encapsulated = value
End Property
```

Combined with the `Variant` type, this can be used to create rather non-obvious ways to return a value to the caller:

```
Public Event SomeEvent(ByVal foo As Variant)

Public Sub DoSomething()
    Dim result As ReturnBoolean
    result = New ReturnBoolean

    RaiseEvent SomeEvent(result)

    If result Then ' If result.ReturnValue Then
        'handler changed the value to True
    Else
        'handler didn't modify the value
    End If
End Sub
```

The handler would look like this:

```
Private Sub source_SomeEvent(ByVal foo As Variant) 'foo is actually a ReturnBoolean object
    foo = True 'True is actually assigned to foo.ReturnValue, the class' default member
End Sub
```

Chapter 28: Attributes

Section 28.1: VB_PredeclaredId

Creates a Global Default Instance of a class. The default instance is accessed via the name of the class.

Declaration

```
VERSION 1.0 CLASS
BEGIN
  MultiUse = -1  'True
END
Attribute VB_Name = "Class1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Option Explicit

Public Function GiveMeATwo() As Integer
    GiveMeATwo = 2
End Function
```

Call

```
Debug.Print Class1.GiveMeATwo
```

In some ways, this simulates the behavior of static classes in other languages, but unlike other languages, you can still create an instance of the class.

```
Dim cls As Class1
Set cls = New Class1
Debug.Print cls.GiveMeATwo
```

Section 28.2: VB_[Var]UserMemId

`VB_VarUserMemId` (for module-scope variables) and `VB_UserMemId` (for procedures) attributes are used in VBA mostly for two things.

Specifying the default member of a class

A `List` class that would encapsulate a `Collection` would want to have an `Item` property, so the client code can do this:

```
For i = 1 To myList.Count 'VBA Collection Objects are 1-based
    Debug.Print myList.Item(i)
Next
```

But with a `VB_UserMemId` attribute set to 0 on the `Item` property, the client code can do this:

```
For i = 1 To myList.Count 'VBA Collection Objects are 1-based
    Debug.Print myList(i)
Next
```

Only one member can legally have `VB_UserMemId = 0` in any given class. For properties, specify the attribute in the `Get` accessor:

```
Option Explicit
Private internal As New Collection

Public Property Get Count() As Long
    Count = internal.Count
End Property

Public Property Get Item(ByVal index As Long) As Variant
Attribute Item.VB_Description = "Gets or sets the element at the specified index."
Attribute Item.VB_UserMemId = 0
'Gets the element at the specified index.
    Item = internal(index)
End Property

Public Property Let Item(ByVal index As Long, ByVal value As Variant)
'Sets the element at the specified index.
    With internal
        If index = .Count + 1 Then
            .Add item:=value
        ElseIf index = .Count Then
            .Remove index
            .Add item:=value
        ElseIf index < .Count Then
            .Remove index
            .Add item:=value, before:=index
        End If
    End With
End Property
```

Making a class iteratable with a `For Each` loop construct

With the magic value `-4`, the `VB_UserMemId` attribute tells VBA that this member yields an enumerator - which allows the client code to do this:

```
Dim item As Variant
For Each item In myList
    Debug.Print item
Next
```

The easiest way to implement this method is by calling the hidden `[_NewEnum]` property getter on an internal/encapsulated `Collection`; the identifier needs to be enclosed in square brackets because of the leading underscore that makes it an illegal VBA identifier:

```
Public Property Get NewEnum() As IUnknown
Attribute NewEnum.VB_Description = "Gets an enumerator that iterates through the List."
Attribute NewEnum.VB_UserMemId = -4
Attribute NewEnum.VB_MemberFlags = "40" 'would hide the member in VB6. not supported in VBA.
'Gets an enumerator that iterates through the List.
    Set NewEnum = internal.[_NewEnum]
End Property
```

Section 28.3: VB_Exposed

Controls the instancing characteristics of a class.

```
Attribute VB_Exposed = False
```

Makes the class `Private`. It cannot be accessed outside of the current project.

```
Attribute VB_Exposed = True
```

Exposes the class `Public`ly, outside of the project. However, since `VB_Createable` is ignored in VBA, instances of the class can not be created directly. This is equivalent to a the following VB.Net class.

```
Public Class Foo
    Friend Sub New()
    End Sub
End Class
```

In order to get an instance from outside the project, you must expose a factory to create instances. One way of doing this is with a regular `Public` module.

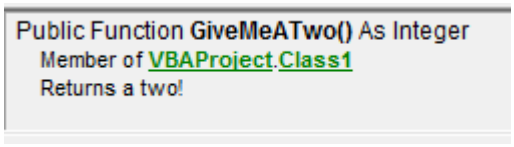
```
Public Function CreateFoo() As Foo
    CreateFoo = New Foo
End Function
```

Since public modules are accessible from other projects, this allows us to create new instances of our `Public` - `Not Createable` classes.

Section 28.4: VB_Description

Adds a text description to a class or module member that becomes visible in the Object Explorer. Ideally, all public members of a public interface / API should have a description.

```
Public Function GiveMeATwo() As Integer
    Attribute GiveMeATwo.VB_Description = "Returns a two!"
    GiveMeATwo = 2
End Property
```



Note: all accessor members of a property (`Get`, `Let`, `Set`) use the same description.

Section 28.5: VB_Name

VB_Name specifies the class or module name.

```
Attribute VB_Name = "Class1"
```

A new instance of this class would be created with

```
Dim myClass As Class1
myClass = new Class1
```

Section 28.6: VB_GlobalNameSpace

In VBA, this attribute is ignored. It was not ported over from VB6.

In VB6, it creates a Default Global Instance of the class (a "shortcut") so that class members can be accessed without using the class name. For example, `DateTime` (as in `DateTime.Now`) is actually part of the `VBA.Conversion`

class.

```
Debug.Print VBA.Conversion.DateTime.Now
Debug.Print DateTime.Now
```

Section 28.7: VB_Createable

This attribute is ignored. It was not ported over from VB6.

In VB6, it was used in combination with the `VB_Exposed` attribute to control accessibility of classes outside of the current project.

```
VB_Exposed=True
VB_Creatable=True
```

Would result in a `Public Class`, that could be accessed from other projects, but this functionality does not exist in VBA.

Chapter 29: User Forms

Section 29.1: Best Practices

A `UserForm` is a class module with a designer and a **default instance**. The *designer* can be accessed by pressing `Shift`+`F7` while viewing the *code-behind*, and the *code-behind* can be accessed by pressing `F7` while viewing the *designer*.

Work with a new instance every time.

Being a *class module*, a form is therefore a *blueprint* for an *object*. Because a form can hold state and data, it's a better practice to work with a new *instance* of the class, rather than with the default/global one:

```
With New UserForm1
    .Show vbModal
    If Not .IsCancelled Then
        '...
    End If
End With
```

Instead of:

```
UserForm1.Show vbModal
If Not UserForm1.IsCancelled Then
    '...
End If
```

Working with the default instance can lead to subtle bugs when the form is closed with the red "X" button and/or when `Unload Me` is used in the code-behind.

Implement the logic elsewhere.

A form should be concerned with nothing but *presentation*: a button `Click` handler that connects to a database and runs a parameterized query based on user input, is **doing too many things**.

Instead, implement the *applicative logic* in the code that's responsible for displaying the form, or even better, in dedicated modules and procedures.

Write the code in such a way that the UserForm is only ever responsible for knowing how to display and collect data: where the data comes from, or what happens with the data afterwards, is none of its concern.

Caller shouldn't be bothered with controls.

Make a well-defined *model* for the form to work with, either in its own dedicated class module, or encapsulated within the form's code-behind itself - expose the *model* with `Property Get` procedures, and have the client code work with these: this makes the form an *abstraction* over controls and their nitty-gritty details, exposing only the relevant data to the client code.

This means code that looks like this:

```
With New UserForm1
```

```
    .Show vbModal
    If Not .IsCancelled Then
        MsgBox .Message, vbInformation
    End If
End With
```

Instead of this:

```
With New UserForm1
    .Show vbModal
    If Not .IsCancelled Then
        MsgBox .txtMessage.Text, vbInformation
    End If
End With
```

Handle the QueryClose event.

Forms typically have a `Close` button, and prompts/dialogs have `Ok` and `Cancel` buttons; the user may close the form using the form's *control box* (the red "X" button), which destroys the form instance by default (another good reason to *work with a new instance every time*).

```
With New UserForm1
    .Show vbModal
    If Not .IsCancelled Then 'if QueryClose isn't handled, this can raise a runtime error.
        '...
    End With
End With
```

The simplest way to handle the `QueryClose` event is to set the `Cancel` parameter to `True`, and then to *hide* the form instead of *closing* it:

```
Private Sub UserForm_QueryClose(Cancel As Integer, CloseMode As Integer)
    Cancel = True
    Me.Hide
End Sub
```

That way the "X" button will never destroy the instance, and the caller can safely access all the public members.

Hide, don't close.

The code that creates an object should be responsible for destroying it: it's not the form's responsibility to unload and terminate itself.

Avoid using `Unload Me` in a form's code-behind. Call `Me.Hide` instead, so that the calling code can still use the object it created when the form closes.

Name things.

Use the *properties* toolwindow (`F4`) to carefully name each control on a form. The name of a control is used in the code-behind, so unless you're using a refactoring tool that can handle this, **renaming a control will break the code** - so it's much easier to do things right in the first place, than try to puzzle out exactly which of the 20 textboxes `TextBox12` stands for.

Traditionally, UserForm controls are named with Hungarian-style prefixes:

- `lblUserName` for a `Label` control that indicates a user name.
- `txtUserName` for a `TextBox` control where the user can enter a user name.
- `cboUserName` for a `ComboBox` control where the user can enter or pick a user name.
- `lstUserName` for a `ListBox` control where the user can pick a user name.
- `btnOk` or `cmdOk` for a `Button` control labelled "Ok".

The problem is that when e.g. the UI gets redesigned and a `ComboBox` changes to a `ListBox`, the name needs to change to reflect the new control type: it's better to name controls for what they represent, rather than after their control type - to *decouple* the code from the UI as much as possible.

- `UserNameLabel` for a read-only label that indicates a user name.
- `UserNameInput` for a control where the user can enter or pick a user name.
- `OkButton` for a command button labelled "Ok".

Whichever style is chosen, anything is better than leaving all controls their default names. Consistency in naming style is ideal, too.

Section 29.2: Handling QueryClose

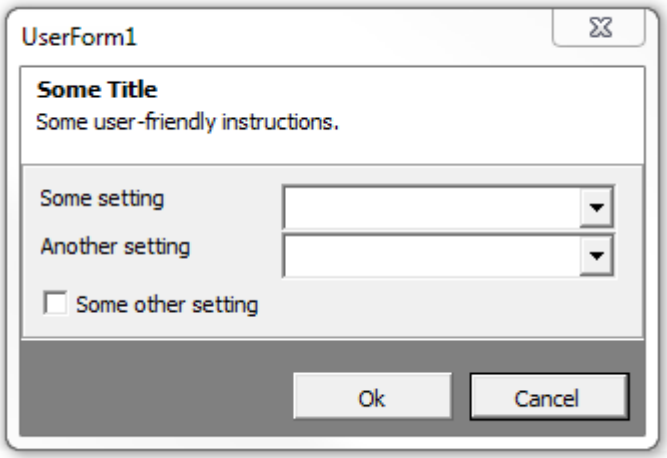
The `QueryClose` event is raised whenever a form is about to be closed, whether it's via user action or programmatically. The `CloseMode` parameter contains a `VbQueryClose` enum value that indicates how the form was closed:

Constant	Description	Value
`vbFormControlMenu`	Form is closing in response to user action	0
`vbFormCode`	Form is closing in response to an `Unload` statement	1
`vbAppWindows`	Windows session is ending	2
`vbAppTaskManager`	Windows Task Manager is closing the host application	3
`vbFormMDIForm`	Not supported in VBA	4

For better readability, it's best to use these constants instead of using their value directly.

A Cancellable UserForm

Given a form with a `Cancel` button



The form's code-behind could look like this:

```
Option Explicit
```

```
Private Type TView
    IsCancelled As Boolean
    SomeOtherSetting As Boolean
    'other properties skipped for brievety
End Type
Private this As TView

Public Property Get IsCancelled() As Boolean
    IsCancelled = this.IsCancelled
End Property

Public Property Get SomeOtherSetting() As Boolean
    SomeOtherSetting = this.SomeOtherSetting
End Property

'...more properties...

Private Sub SomeOtherSettingInput_Change()
    this.SomeOtherSetting = CBool(SomeOtherSettingInput.Value)
End Sub

Private Sub OkButton_Click()
    Me.Hide
End Sub

Private Sub CancelButton_Click()
    this.IsCancelled = True
    Me.Hide
End Sub

Private Sub UserForm_QueryClose(Cancel As Integer, CloseMode As Integer)
    If CloseMode = VbQueryClose.vbFormControlMenu Then
        Cancel = True
        this.IsCancelled = True
        Me.Hide
    End If
End Sub
```

The calling code could then display the form, and know whether it was cancelled:

```
Public Sub DoSomething()
    With New UserForm1
        .Show vbModal
        If .IsCancelled Then Exit Sub
        If .SomeOtherSetting Then
            'setting is enabled
        Else
            'setting is disabled
        End If
    End With
End Sub
```

The `IsCancelled` property returns `True` when the `Cancel` button is clicked, or when the user closes the form using the *control box*.

Chapter 30: Object-Oriented VBA

Section 30.1: Abstraction

Abstraction levels help determine when to split things up.

Abstraction is achieved by implementing functionality with increasingly detailed code. The entry point of a macro should be a small procedure with a *high abstraction level* that makes it easy to grasp at a glance what's going on:

```
Public Sub DoSomething()
    With New SomeForm
        Set .Model = CreateViewModel
        .Show vbModal
        If .IsCancelled Then Exit Sub
        ProcessUserData .Model
    End With
End Sub
```

The `DoSomething` procedure has a high *abstraction level*: we can tell that it's displaying a form and creating some model, and passing that object to some `ProcessUserData` procedure that knows what to do with it - how the model is created is the job of another procedure:

```
Private Function CreateViewModel() As ISomeModel
    Dim result As ISomeModel
    Set result = SomeModel.Create(Now, Environ$("UserName"))
    result.AvailableItems = GetAvailableItems
    Set CreateViewModel = result
End Function
```

The `CreateViewModel` function is only responsible for creating some `ISomeModel` instance. Part of that responsibility is to acquire an array of *available items* - how these items are acquired is an implementation detail that's abstracted behind the `GetAvailableItems` procedure:

```
Private Function GetAvailableItems() As Variant
    GetAvailableItems = DataSheet.Names("AvailableItems").RefersToRange
End Function
```

Here the procedure is reading the available values from a named range on a `DataSheet` worksheet. It could just as well be reading them from a database, or the values could be hard-coded: it's an *implementation detail* that's none of a concern for any of the higher abstraction levels.

Section 30.2: Encapsulation

Encapsulation hides implementation details from client code.

The Handling QueryClose example demonstrates encapsulation: the form has a checkbox control, but its client code doesn't work with it directly - the checkbox is an *implementation detail*, what the client code needs to know is whether the setting is enabled or not.

When the checkbox value changes, the handler assigns a private field member:

```
Private Type TView
    IsCancelled As Boolean
    SomeOtherSetting As Boolean
    'other properties skipped for brievety
```

```
End Type
Private this As TView

'...

Private Sub SomeOtherSettingInput_Change()
    this.SomeOtherSetting = CBool(SomeOtherSettingInput.Value)
End Sub
```

And when the client code wants to read that value, it doesn't need to worry about a checkbox - instead it simply uses the `SomeOtherSetting` property:

```
Public Property Get SomeOtherSetting() As Boolean
    SomeOtherSetting = this.SomeOtherSetting
End Property
```

The `SomeOtherSetting` property *encapsulates* the checkbox' state; client code doesn't need to know that there's a checkbox involved, only that there's a setting with a Boolean value. By *encapsulating* the `Boolean` value, we've added an *abstraction layer* around the checkbox.

Using interfaces to enforce immutability

Let's push that a step further by *encapsulating* the form's *model* in a dedicated class module. But if we made a `Public Property` for the `UserName` and `Timestamp`, we would have to expose `Property Let` accessors, making the properties mutable, and we don't want the client code to have the ability to change these values after they're set.

The `CreateViewModel` function in the **Abstraction** example returns an `ISomeModel` class: that's our *interface*, and it looks something like this:

```
Option Explicit

Public Property Get Timestamp() As Date
End Property

Public Property Get UserName() As String
End Property

Public Property Get AvailableItems() As Variant
End Property

Public Property Let AvailableItems(ByRef value As Variant)
End Property

Public Property Get SomeSetting() As String
End Property

Public Property Let SomeSetting(ByVal value As String)
End Property

Public Property Get SomeOtherSetting() As Boolean
End Property

Public Property Let SomeOtherSetting(ByVal value As Boolean)
End Property
```

Notice `Timestamp` and `UserName` properties only expose a `Property Get` accessor. Now the `SomeModel` class can implement that interface:

```
Option Explicit
Implements ISomeModel

Private Type TModel
    Timestamp As Date
    UserName As String
    SomeSetting As String
    SomeOtherSetting As Boolean
    AvailableItems As Variant
End Type
Private this As TModel

Private Property Get ISomeModel_Timestamp() As Date
    ISomeModel_Timestamp = this.Timestamp
End Property

Private Property Get ISomeModel_UserName() As String
    ISomeModel_UserName = this.UserName
End Property

Private Property Get ISomeModel_AvailableItems() As Variant
    ISomeModel_AvailableItems = this.AvailableItems
End Property

Private Property Let ISomeModel_AvailableItems(ByRef value As Variant)
    this.AvailableItems = value
End Property

Private Property Get ISomeModel_SomeSetting() As String
    ISomeModel_SomeSetting = this.SomeSetting
End Property

Private Property Let ISomeModel_SomeSetting(ByVal value As String)
    this.SomeSetting = value
End Property

Private Property Get ISomeModel_SomeOtherSetting() As Boolean
    ISomeModel_SomeOtherSetting = this.SomeOtherSetting
End Property

Private Property Let ISomeModel_SomeOtherSetting(ByVal value As Boolean)
    this.SomeOtherSetting = value
End Property

Public Property Get Timestamp() As Date
    Timestamp = this.Timestamp
End Property

Public Property Let Timestamp(ByVal value As Date)
    this.Timestamp = value
End Property

Public Property Get UserName() As String
    UserName = this.UserName
End Property

Public Property Let UserName(ByVal value As String)
    this.UserName = value
End Property

Public Property Get AvailableItems() As Variant
    AvailableItems = this.AvailableItems
```

```
End Property

Public Property Let AvailableItems(ByRef value As Variant)
    this.AvailableItems = value
End Property

Public Property Get SomeSetting() As String
    SomeSetting = this.SomeSetting
End Property

Public Property Let SomeSetting(ByVal value As String)
    this.SomeSetting = value
End Property

Public Property Get SomeOtherSetting() As Boolean
    SomeOtherSetting = this.SomeOtherSetting
End Property

Public Property Let SomeOtherSetting(ByVal value As Boolean)
    this.SomeOtherSetting = value
End Property
```

The interface members are all `Private`, and all members of the interface must be implemented for the code to compile. The `Public` members are not part of the interface, and are therefore not exposed to code written against the `ISomeModel` interface.

Using a Factory Method to simulate a constructor

Using a VB_PredeclaredId attribute, we can make the `SomeModel` class have a *default instance*, and write a function that works like a type-level (`Shared` in VB.NET, `static` in C#) member that the client code can call without needing to first create an instance, like we did here:

```
Private Function CreateViewModel() As ISomeModel
    Dim result As ISomeModel
    Set result = SomeModel.Create(Now, Environ$("UserName"))
    result.AvailableItems = GetAvailableItems
    Set CreateViewModel = result
End Function
```

This *factory method* assigns the property values that are read-only when accessed from the `ISomeModel` interface, here `Timestamp` and `UserName`:

```
Public Function Create(ByVal pTimeStamp As Date, ByVal pUserName As String) As ISomeModel
    With New SomeModel
        .Timestamp = pTimeStamp
        .UserName = pUserName
        Set Create = .Self
    End With
End Function

Public Property Get Self() As ISomeModel
    Set Self = Me
End Property
```

And now we can code against the `ISomeModel` interface, which exposes `Timestamp` and `UserName` as read-only properties that can never be reassigned (as long as the code is written against the interface).

Section 30.3: Polymorphism

Polymorphism is the ability to present the same interface for different underlying implementations.

The ability to implement interfaces allows completely decoupling the application logic from the UI, or from the database, or from this or that worksheet.

Say you have an `ISomeView` interface that the form itself implements:

```
Option Explicit

Public Property Get IsCancelled() As Boolean
End Property

Public Property Get Model() As ISomeModel
End Property

Public Property Set Model(ByVal value As ISomeModel)
End Property

Public Sub Show()
End Sub
```

The form's code-behind could look like this:

```
Option Explicit
Implements ISomeView

Private Type TView
    IsCancelled As Boolean
    Model As ISomeModel
End Type
Private this As TView

Private Property Get ISomeView_IsCancelled() As Boolean
    ISomeView_IsCancelled = this.IsCancelled
End Property

Private Property Get ISomeView_Model() As ISomeModel
    Set ISomeView_Model = this.Model
End Property

Private Property Set ISomeView_Model(ByVal value As ISomeModel)
    Set this.Model = value
End Property

Private Sub ISomeView_Show()
    Me.Show vbModal
End Sub

Private Sub SomeOtherSettingInput_Change()
    this.Model.SomeOtherSetting = CBool(SomeOtherSettingInput.Value)
End Sub

'...other event handlers...

Private Sub OkButton_Click()
    Me.Hide
End Sub
```

```
Private Sub CancelButton_Click()
    this.IsCancelled = True
    Me.Hide
End Sub

Private Sub UserForm_QueryClose(Cancel As Integer, CloseMode As Integer)
    If CloseMode = VbQueryClose.vbFormControlMenu Then
        Cancel = True
        this.IsCancelled = True
        Me.Hide
    End If
End Sub
```

But then, nothing forbids creating another class module that implements the `ISomeView` interface *without being a user form* - this could be a `SomeViewMock` class:

```
Option Explicit
Implements ISomeView

Private Type TView
    IsCancelled As Boolean
    Model As ISomeModel
End Type
Private this As TView

Public Property Get IsCancelled() As Boolean
    IsCancelled = this.IsCancelled
End Property

Public Property Let IsCancelled(ByVal value As Boolean)
    this.IsCancelled = value
End Property

Private Property Get ISomeView_IsCancelled() As Boolean
    ISomeView_IsCancelled = this.IsCancelled
End Property

Private Property Get ISomeView_Model() As ISomeModel
    Set ISomeView_Model = this.Model
End Property

Private Property Set ISomeView_Model(ByVal value As ISomeModel)
    Set this.Model = value
End Property

Private Sub ISomeView_Show()
    'do nothing
End Sub
```

And now we can change the code that works with a `UserForm` and make it work off the `ISomeView` interface, e.g. by giving it the form as a parameter instead of instantiating it:

```
Public Sub DoSomething(ByVal view As ISomeView)
    With view
        Set .Model = CreateViewModel
        .Show
        If .IsCancelled Then Exit Sub
        ProcessUserData .Model
    End With
End Sub
```

Because the `DoSomething` method depends on an interface (i.e. an *abstraction*) and not a *concrete class* (e.g. a specific `UserForm`), we can write an automated unit test that ensures that `ProcessUserData` isn't executed when `view.IsCancelled` is `True`, by making our test create a `SomeViewMock` instance, setting its `IsCancelled` property to `True`, and passing it to `DoSomething`.

Testable code depends on abstractions

Writing unit tests in VBA can be done, there are add-ins out there that even integrate it into the IDE. But when code is *tightly coupled* with a worksheet, a database, a form, or the file system, then the unit test starts requiring an actual worksheet, database, form, or file system - and these *dependencies* are new out-of-control failure points that testable code should isolate, so that unit tests *don't* require an actual worksheet, database, form, or file system.

By writing code against interfaces, in a way that allows test code to *inject* stub/mock implementations (like the above `SomeViewMock` example), you can write tests in a "controlled environment", and simulate what happens when every single one of the 42 possible permutations of user interactions on the form's data, without even once displaying a form and manually clicking on a form control.

Chapter 31: Working With Files and Directories Without Using FileSystemObject

Section 31.1: Determining If Folders and Files Exist

Files:

To determine if a file exists, simply pass the filename to the `Dir$` function and test to see if it returns a result. Note that `Dir$` supports wild-cards, so to test for a *specific* file, the passed `pathName` should to be tested to ensure that it does not contain them. The sample below raises an error - if this isn't the desired behavior, the function can be changed to simply return `False`.

```
Public Function FileExists(pathName As String) As Boolean
    If InStr(1, pathName, "*") Or InStr(1, pathName, "?") Then
        'Exit Function   'Return False on wild-cards.
        Err.Raise 52    'Raise error on wild-cards.
    End If
    FileExists = Dir$(pathName) <> vbNullString
End Function
```

Folders (Dir$ method):

The `Dir$()` function can also be used to determine if a folder exists by specifying passing `vbDirectory` for the optional `attributes` parameter. In this case, the passed `pathName` value must end with a path separator (`\`), as matching *filenames* will cause false positives. Keep in mind that wild-cards are only allowed after the last path separator, so the example function below will throw a run-time error 52 - "Bad file name or number" if the input contains a wild-card. If this isn't the desired behavior, uncomment `On Error Resume Next` at the top of the function. Also remember that `Dir$` supports relative file paths (i.e. `..\Foo\Bar`), so results are only guaranteed to be valid as long as the current working directory is not changed.

```
Public Function FolderExists(ByVal pathName As String) As Boolean
    'Uncomment the "On Error" line if paths with wild-cards should return False
    'instead of raising an error.
    'On Error Resume Next
    If pathName = vbNullString Or Right$(pathName, 1) <> "\" Then
        Exit Function
    End If
    FolderExists = Dir$(pathName, vbDirectory) <> vbNullString
End Function
```

Folders (ChDir method):

The `ChDir` statement can also be used to test if a folder exists. Note that this method will temporarily change the environment that VBA is running in, so if that is a consideration, the `Dir$` method should be used instead. It does have the advantage of being much less forgiving with its parameter. This method also supports relative file paths, so has the same caveat as the `Dir$` method.

```
Public Function FolderExists(ByVal pathName As String) As Boolean
    'Cache the current working directory
    Dim cached As String
    cached = CurDir$

    On Error Resume Next
```

```
    ChDir pathName
    FolderExists = Err.Number = 0
    On Error GoTo 0
    'Change back to the cached working directory.
    ChDir cached
End Function
```

Section 31.2: Creating and Deleting File Folders

NOTE: For brevity, the examples below use the FolderExists function from the **Determining If Folders and Files Exist** example in this topic.

The `MkDir` statement can be used to create a new folder. It accepts paths containing drive letters (`C:\Foo`), UNC names (`\\Server\Foo`), relative paths (`..\Foo`), or the current working directory (`Foo`).

If the drive or UNC name is omitted (i.e. `\Foo`), the folder is created on the current drive. This may or may not be the same drive as the current working directory.

```
Public Sub MakeNewDirectory(ByVal pathName As String)
    'MkDir will fail if the directory already exists.
    If FolderExists(pathName) Then Exit Sub
    'This may still fail due to permissions, etc.
    MkDir pathName
End Sub
```

The `RmDir` statement can be used to delete existing folders. It accepts paths in the same forms as `MkDir` and uses the same relationship to the current working directory and drive. Note that the statement is similar to the Windows `rd` shell command, so will throw a run-time error 75: "Path/File access error" if the target directory is not empty.

```
Public Sub DeleteDirectory(ByVal pathName As String)
    If Right$(pathName, 1) <> "\" Then
        pathName = pathName & "\"
    End If
    'Rmdir will fail if the directory doesn't exist.
    If Not FolderExists(pathName) Then Exit Sub
    'Rmdir will fail if the directory contains files.
    If Dir$(pathName & "*") <> vbNullString Then Exit Sub

    'Rmdir will fail if the directory contains directories.
    Dim subDir As String
    subDir = Dir$(pathName & "*", vbDirectory)
    Do
        If subDir <> "." And subDir <> ".." Then Exit Sub
        subDir = Dir$(, vbDirectory)
    Loop While subDir <> vbNullString

    'This may still fail due to permissions, etc.
    RmDir pathName
End Sub
```

Chapter 32: Operators

Section 32.1: Concatenation Operators

VBA supports 2 different concatenation operators, `+` and `&` and both perform the exact same function when used with `String` types - the right-hand `String` is appended to the end of the left-hand `String`.

If the `&` operator is used with a variable type other than a `String`, it is implicitly cast to a `String` before being concatenated.

Note that the `+` concatenation operator is an overload of the `+` addition operator. The behavior of `+` is determined by the variable types of the operands and precedence of operator types. If both operands are typed as a `String` or `Variant` with a sub-type of `String`, they are concatenated:

```
Public Sub Example()
    Dim left As String
    Dim right As String

    left = "5"
    right = "5"

    Debug.Print left + right    'Prints "55"
End Sub
```

If *either* side is a numeric type and the other side is a `String` that can be coerced into a number, the type precedence of mathematical operators causes the operator to be treated as the addition operator and the numeric values are added:

```
Public Sub Example()
    Dim left As Variant
    Dim right As String

    left = 5
    right = "5"

    Debug.Print left + right    'Prints 10
End Sub
```

This behavior can lead to subtle, hard to debug errors - especially if `Variant` types are being used, so only the `&` operator should typically be used for concatenation.

Section 32.2: Comparison Operators

Token	Name	Description
`=`	Equal to	Returns `True` if the left-hand and right-hand operands are equal. Note that this is an overload of the assignment operator.
`<>`	Not equal to	Returns `True` if the left-hand and right-hand operands are not equal.
`>`	Greater than	Returns `True` if the left-hand operand is greater than the right-hand operand.
`<`	Less than	Returns `True` if the left-hand operand is less than the right-hand operand.
`>=`	Greater than or equal	Returns `True` if the if the left-hand operand is greater than or equal to the right-hand operand.
`<=`	Less than or equal	Returns `True` if the if the left-hand operand is less than or equal to the right-hand operand.

`Is`	Reference equity	Returns `True` if the left-hand object reference is the same instance as the right-hand object reference. It can also be used with `Nothing` (the null object reference) on either side. **Note:** The `Is` operator will attempt to coerce both operands into an `Object` before performing the comparison. If either side is a primitive type *or* a `Variant` that does not contain an object (either a non-object subtype or `vtEmpty`), the comparison will result in a Run-time error 424 - "Object required". If either operand belongs to a different *interface* of the same object, the comparison will return `True`. If you need to test for equity of both the instance *and* the interface, use `ObjPtr(left) = ObjPtr(right)` instead.

Notes

The VBA syntax allows for "chains" of comparison operators, but these constructs should generally be avoided. Comparisons are always performed from left to right on only 2 operands at a time, and each comparison results in a `Boolean`. For example, the expression...

```
a = 2: b = 1: c = 0
expr = a > b > c
```

...may be read in some contexts as a test of whether `b` is between `a` and `c`. In VBA, this evaluates as follows:

```
a = 2: b = 1: c = 0
expr = a > b > c
expr = (2 > 1) > 0
expr = True > 0
expr = -1 > 0 'CInt(True) = -1
expr = False
```

Any comparison operator other than `Is` used with an `Object` as an operand will be performed on the return value of the `Object`'s default member. If the object does not have a default member, the comparison will result in a Run-time error 438 - "Object doesn't support his property or method".

If the `Object` is unintitialized, the comparison will result in a Run-time error 91 - "Object variable or With block variable not set".

If the literal `Nothing` is used with any comparison operator other than `Is`, it will result in a Compile error - "Invalid use of object".

If the default member of the `Object` is *another* `Object`, VBA will continually call the default member of each successive return value until a primitive type is returned or an error is raised. For example, assume `SomeClass` has a default member of `Value`, which is an instance of `ChildClass` with a default member of `ChildValue`. The comparison...

```
Set x = New SomeClass
Debug.Print x > 42
```

...will be evaluated as:

```
Set x = New SomeClass
Debug.Print x.Value.ChildValue > 42
```

If either operand is a numeric type and the *other* operand is a `String` or `Variant` of subtype `String`, a numeric comparison will be performed. In this case, if the `String` cannot be cast to a number, a Run-time error 13 - "Type mismatch" will result from the comparison.

If ***both*** operands are a `String` or a `Variant` of subtype `String`, a string comparison will be performed based on the

Option Compare setting of the code module. These comparisons are performed on a character by character basis. Note that the *character representation* of a `String` containing a number is ***not*** the same as a comparison of the numeric values:

```
Public Sub Example()
    Dim left As Variant
    Dim right As Variant

    left = "42"
    right = "5"
    Debug.Print left > right              'Prints False
    Debug.Print Val(left) > Val(right)    'Prints True
End Sub
```

For this reason, make sure that `String` or `Variant` variables are cast to numbers before performing numeric inequity comparisons on them.

If one operand is a `Date`, a numeric comparison on the underlying Double value will be performed if the other operand is numeric or can be cast to a numeric type.

If the other operand is a `String` or a `Variant` of subtype `String` that can be cast to a `Date` using the current locale, the `String` will be cast to a `Date`. If it cannot be cast to a `Date` in the current locale, a Run-time error 13 - "Type mismatch" will result from the comparison.

Care should be taken when making comparisons between `Double` or `Single` values and Booleans. Unlike other numeric types, non-zero values cannot be assumed to be **`True`** due to VBA's behavior of promoting the data type of a comparison involving a floating point number to `Double`:

```
Public Sub Example()
    Dim Test As Double

    Test = 42        Debug.Print CBool(Test)             'Prints True.
    'True is promoted to Double - Test is not cast to Boolean
    Debug.Print Test = True             'Prints False

    'With explicit casts:
    Debug.Print CBool(Test) = True      'Prints True
    Debug.Print CDbl(-1) = CDbl(True)   'Prints True
End Sub
```

Section 32.3: Bitwise \ Logical Operators

All of the logical operators in VBA can be thought of as "overrides" of the bitwise operators of the same name. Technically, they are *always* treated as bitwise operators. All of the comparison operators in VBA return a Boolean, which will always have none of its bits set (**`False`**) or *all* of its bits set (**`True`**). But it will treat a value with *any* bit set as **`True`**. This means that the result of the casting the bitwise result of an expression to a `Boolean` (see Comparison Operators) will always be the same as treating it as a logical expression.

Assigning the result of an expression using one of these operators will give the bitwise result. Note that in the truth tables below, `0` is equivalent to **`False`** and `1` is equivalent to **`True`**.

`And`

Returns **`True`** if the expressions on both sides evaluate to **`True`**.

Left-hand Operand Right-hand Operand Result

0	0	0
0	1	0
1	0	0
1	1	1

`Or`

Returns `True` if either side of the expression evaluates to `True`.

Left-hand Operand	Right-hand Operand	Result
0	0	0
0	1	1
1	0	1
1	1	1

`Not`

Returns `True` if the expression evaluates to `False` and `False` if the expression evaluations to `True`.

Right-hand Operand	Result
0	1
1	0

`Not` is the only operand without a Left-hand operand. The Visual Basic Editor will automatically simplify expressions with a left hand argument. If you type...

```
Debug.Print x Not y
```

...the VBE will change the line to:

```
Debug.Print Not x
```

Similar simplifications will be made to any expression that contains a left-hand operand (including expressions) for `Not`.

`Xor`

Also known as "exclusive or". Returns `True` if both expressions evaluate to different results.

Left-hand Operand	Right-hand Operand	Result
0	0	0
0	1	1
1	0	1
1	1	0

Note that although the `Xor` operator can be *used* like a logical operator, there is absolutely no reason to do so as it gives the same result as the comparison operator `<>`.

`Eqv`

Also known as "equivalence". Returns `True` when both expressions evaluate to the same result.

Left-hand Operand	Right-hand Operand	Result
0	0	1
0	1	0
1	0	0
1	1	1

Note that the `Eqv` function is *very* rarely used as `x Eqv y` is equivalent to the much more readable `Not (x Xor y)`.

`Imp`

Also known as "implication". Returns `True` if both operands are the same *or* the second operand is `True`.

Left-hand Operand	Right-hand Operand	Result
0	0	1
0	1	1
1	0	0
1	1	1

Note that the `Imp` function is very rarely used. A good rule of thumb is that if you can't explain what it means, you should use another construct.

Section 32.4: Mathematical Operators

Listed in order of precedence:

Token	Name	Description
`^`	Exponentiation	Return the result of raising the left-hand operand to the power of the right-hand operand. Note that the value returned by exponentiation is *always* a `Double`, regardless of the value types being divided. Any coercion of the result into a variable type takes place ***after*** the calculation is performed.
`/`	Division1	Returns the result of dividing the left-hand operand by the right-hand operand. Note that the value returned by division is *always* a `Double`, regardless of the value types being divided. Any coercion of the result into a variable type takes place ***after*** the calculation is performed.
`*`	Multiplication1	Returns the product of 2 operands.
`\`	Integer Division	Returns the integer result of dividing the left-hand operand by the right-hand operand ***after*** rounding both sides with .5 rounding down. Any remainder of the division is ignored. If the right-hand operand (the divisor) is `0`, a Run-time error 11: Division by zero will result. Note that this is ***after*** all rounding is performed - expressions such as `3 \ 0.4` will also result in a division by zero error.
`Mod`	Modulo	Returns the integer remainder of dividing the left-hand operand by the right-hand operand. The operand on each side is rounded to an integer *before* the division, with .5 rounding down. For example, both `8.6 Mod 3` and `12 Mod 2.6` result in `0`. If the right-hand operand (the divisor) is `0`, a Run-time error 11: Division by zero will result. Note that this is ***after*** all rounding is performed - expressions such as `3 Mod 0.4` will also result in a division by zero error.
`-`	Subtraction2	Returns the result of subtracting the right-hand operand from the left-hand operand.
`+`	Addition2	Returns the sum of 2 operands. Note that this token also treated as a concatenation operator when it is applied to a `String`. See **Concatenation Operators**.

1 Multiplication and division are treated as having the same precedence.

2 Addition and subtraction are treated as having the same precedence.

Chapter 33: Collections

Section 33.1: Getting the Item Count of a Collection

The number of items in a `Collection` can be obtained by calling its `.Count` function:

Syntax:

```
.Count()
```

Sample Usage:

```
Public Sub Example()
    Dim foo As New Collection

    With foo
        .Add "One"
        .Add "Two"
        .Add "Three"
        .Add "Four"
    End With

    Debug.Print foo.Count   'Prints 4
End Sub
```

Section 33.2: Determining if a Key or Item Exists in a Collection

Keys

Unlike a Scripting.Dictionary, a `Collection` does not have a method for determining if a given key exists *or* a way to retrieve keys that are present in the `Collection`. The only method to determine if a key is present is to use the error handler:

```
Public Function KeyExistsInCollection(ByVal key As String, _
                                      ByRef container As Collection) As Boolean
    With Err
        If container Is Nothing Then .Raise 91
        On Error Resume Next
        Dim temp As Variant
        temp = container.Item(key)
        On Error GoTo 0

        If .Number = 0 Then
            KeyExistsInCollection = True
        ElseIf .Number <> 5 Then
            .Raise .Number
        End If
    End With
End Function
```

Items

The only way to determine if an item is contained in a `Collection` is to iterate over the `Collection` until the item is located. Note that because a `Collection` can contain either primitives or objects, some extra handling is needed to avoid run-time errors during the comparisons:

```
Public Function ItemExistsInCollection(ByRef target As Variant, _
                                       ByRef container As Collection) As Boolean
    Dim candidate As Variant
    Dim found As Boolean

    For Each candidate In container
        Select Case True
            Case IsObject(candidate) And IsObject(target)
                found = candidate Is target
            Case IsObject(candidate), IsObject(target)
                found = False
            Case Else
                found = (candidate = target)
        End Select
        If found Then
            ItemExistsInCollection = True
            Exit Function
        End If
    Next
End Function
```

Section 33.3: Adding Items to a Collection

Items are added to a `Collection` by calling its `.Add` method:

Syntax:

```
.Add(item, [key], [before, after])
```

Parameter	Description
item	The item to store in the `Collection`. This can be essentially any value that a variable can be assigned to, including primitive types, arrays, objects, and `Nothing`.
key	Optional. A `String` that serves as a unique identifier for retrieving items from the `Collection`. If the specified key already exists in the `Collection`, it will result in a Run-time error 457: "This key is already associated with an element of this collection".
before	Optional. An existing key (`String` value) *or* index (numeric value) to insert the item before in the `Collection`. If a value is given, the *after* parameter ***must*** be empty or a Run-time error 5: "Invalid procedure call or argument" will result. If a `String` key is passed that does not exist in the `Collection`, a Run-time error 5: "Invalid procedure call or argument" will result. If a numeric index is passed that is does not exist in the `Collection`, a Run-time error 9: "Subscript out of range" will result.
after	Optional. An existing key (`String` value) *or* index (numeric value) to insert the item after in the `Collection`. If a value is given, the *before* parameter ***must*** be empty. Errors raised are identical to the *before* parameter.

Notes:

- Keys are ***not*** case-sensitive. `.Add "Bar", "Foo"` and `.Add "Baz", "foo"` will result in a key collision.
- If neither of the optional *before* or *after* parameters are given, the item will be added after the last item in the `Collection`.
- Insertions made by specifying a *before* or *after* parameter will alter the numeric indexes of existing members to match thier new position. This means that care should be taken when making insertions in loops using numeric indexes.

Sample Usage:

```
Public Sub Example()
    Dim foo As New Collection
```

```
    With foo
        .Add "One"             'No key. This item can only be retrieved by index.
        .Add "Two", "Second"   'Key given. Can be retrieved by key or index.
        .Add "Three", , 1      'Inserted at the start of the collection.
        .Add "Four", , , 1     'Inserted at index 2.
    End With

    Dim member As Variant
    For Each member In foo
        Debug.Print member     'Prints "Three, Four, One, Two"
    Next
End Sub
```

Section 33.4: Removing Items From a Collection

Items are removed from a `Collection` by calling its `.Remove` method:

Syntax:

```
.Remove(index)
```

Parameter	Description
index	The item to remove from the `Collection`. If the value passed is a numeric type or `Variant` with a numeric sub-type, it will be interpreted as a numeric index. If the value passed is a `String` or `Variant` containing a string, it will be interpreted as the a key. If a String key is passed that does not exist in the `Collection`, a Run-time error 5: "Invalid procedure call or argument" will result. If a numeric index is passed that is does not exist in the `Collection`, a Run-time error 9: "Subscript out of range" will result.

Notes:

- Removing an item from a `Collection` will change the numeric indexes of all the items after it in the `Collection`. `For` loops that use numeric indexes and remove items should run *backwards* (`Step -1`) to prevent subscript exceptions and skipped items.
- Items should generally ***not*** be removed from a `Collection` from inside of a `For Each` loop as it can give unpredictable results.

Sample Usage:

```
Public Sub Example()
    Dim foo As New Collection

    With foo
        .Add "One"
        .Add "Two", "Second"
        .Add "Three"
        .Add "Four"
    End With

    foo.Remove 1             'Removes the first item.
    foo.Remove "Second"      'Removes the item with key "Second".
    foo.Remove foo.Count     'Removes the last item.

    Dim member As Variant
    For Each member In foo
        Debug.Print member   'Prints "Three"
    Next
End Sub
```

Section 33.5: Retrieving Items From a Collection

Items can be retrieved from a `Collection` by calling the `.Item` function.

Syntax:

```
.Item(index)
```

Parameter	Description
index	The item to retrieve from the `Collection`. If the value passed is a numeric type or `Variant` with a numeric sub-type, it will be interpreted as a numeric index. If the value passed is a `String` or `Variant` containing a string, it will be interpreted as the a key. If a String key is passed that does not exist in the `Collection`, a Run-time error 5: "Invalid procedure call or argument" will result. If a numeric index is passed that is does not exist in the `Collection`, a Run-time error 9: "Subscript out of range" will result.

Notes:

- `.Item` is the default member of `Collection`. This allows flexibility in syntax as demonstrated in the sample usage below.
- Numeric indexes are 1-based.
- Keys are ***not*** case-sensitive. `.Item("Foo")` and `.Item("foo")` refer to the same key.
- The *index* parameter is ***not*** implicitly cast to a number from a `String` or visa-versa. It is entirely possible that `.Item(1)` and `.Item("1")` refer to different items of the `Collection`.

Sample Usage (Indexes):

```
Public Sub Example()
    Dim foo As New Collection

    With foo
        .Add "One"
        .Add "Two"
        .Add "Three"
        .Add "Four"
    End With

    Dim index As Long
    For index = 1 To foo.Count
        Debug.Print foo.Item(index) 'Prints One, Two, Three, Four
    Next
End Sub
```

Sample Usage (Keys):

```
Public Sub Example()
    Dim keys() As String
    keys = Split("Foo,Bar,Baz", ",")
    Dim values() As String
    values = Split("One,Two,Three", ",")

    Dim foo As New Collection
    Dim index As Long
    For index = LBound(values) To UBound(values)
        foo.Add values(index), keys(index)
    Next

    Debug.Print foo.Item("Bar") 'Prints "Two"
End Sub
```

Sample Usage (Alternate Syntax):

```
Public Sub Example()
    Dim foo As New Collection

    With foo
        .Add "One", "Foo"
        .Add "Two", "Bar"
        .Add "Three", "Baz"
    End With

    'All lines below print "Two"
    Debug.Print foo.Item("Bar")     'Explicit call syntax.
    Debug.Print foo("Bar")          'Default member call syntax.
    Debug.Print foo!Bar             'Bang syntax.
End Sub
```

Note that bang (`!`) syntax is allowed because `.Item` is the default member and can take a single `String` argument. The utility of this syntax is questionable.

Section 33.6: Clearing All Items From a Collection

The easiest way to clear all of the items from a `Collection` is to simply replace it with a new `Collection` and let the old one go out of scope:

```
Public Sub Example()
    Dim foo As New Collection

    With foo
        .Add "One"
        .Add "Two"
        .Add "Three"
    End With

    Debug.Print foo.Count   'Prints 3
    Set foo = New Collection
    Debug.Print foo.Count   'Prints 0
End Sub
```

However, if there are multiple references to the `Collection` held, this method will only give you an empty `Collection` *for the variable that is assigned*.

```
Public Sub Example()
    Dim foo As New Collection
    Dim bar As Collection

    With foo
        .Add "One"
        .Add "Two"
        .Add "Three"
    End With

    Set bar = foo
    Set foo = New Collection

    Debug.Print foo.Count   'Prints 0
    Debug.Print bar.Count   'Prints 3
End Sub
```

In this case, the easiest way to clear the contents is by looping through the number of items in the `Collection` and repeatedly remove the lowest item:

```
Public Sub ClearCollection(ByRef container As Collection)
    Dim index As Long
    For index = 1 To container.Count
        container.Remove 1
    Next
End Sub
```

Chapter 34: Passing Arguments ByRef or ByVal

The `ByRef` and `ByVal` modifiers are part of a procedure's signature and indicate how an argument is passed to a procedure. In VBA a parameter is passed `ByRef` unless specified otherwise (i.e. `ByRef` is implicit if absent).

Note In many other programming languages (including VB.NET), parameters are implicitly passed by value if no modifier is specified: consider specifying `ByRef` modifiers explicitly to avoid possible confusion.

Section 34.1: Passing Simple Variables ByRef And ByVal

Passing `ByRef` or `ByVal` indicates whether the actual value of an argument is passed to the `CalledProcedure` by the `CallingProcedure`, or whether a reference (called a pointer in some other languages) is passed to the `CalledProcedure`.

If an argument is passed `ByRef`, the memory address of the argument is passed to the `CalledProcedure` and any modification to that parameter by the `CalledProcedure` is made to the value in the `CallingProcedure`.

If an argument is passed `ByVal`, the actual value, not a reference to the variable, is passed to the `CalledProcedure`.

A simple example will illustrate this clearly:

```
Sub CalledProcedure(ByRef X As Long, ByVal Y As Long)
    X = 321
    Y = 654
End Sub

Sub CallingProcedure()
    Dim A As Long
    Dim B As Long
    A = 123
    B = 456

    Debug.Print "BEFORE CALL => A: " & CStr(A), "B: " & CStr(B)
    ''Result : BEFORE CALL => A: 123 B: 456

    CalledProcedure X:=A, Y:=B

    Debug.Print "AFTER CALL =  A: " & CStr(A), "B: " & CStr(B)
    ''Result : AFTER CALL => A: 321 B: 456
End Sub
```

Another example:

```
Sub Main()
    Dim IntVarByVal As Integer
    Dim IntVarByRef As Integer

    IntVarByVal = 5
    IntVarByRef = 10

    SubChangeArguments IntVarByVal, IntVarByRef '5 goes in as a "copy". 10 goes in as a reference
    Debug.Print "IntVarByVal: " & IntVarByVal 'prints 5 (no change made by SubChangeArguments)
    Debug.Print "IntVarByRef: " & IntVarByRef 'prints 99 (the variable was changed in
SubChangeArguments)
End Sub
```

```
Sub SubChangeArguments(ByVal ParameterByVal As Integer, ByRef ParameterByRef As Integer)
    ParameterByVal = ParameterByVal + 2 ' 5 + 2 = 7 (changed only inside this Sub)
    ParameterByRef = ParameterByRef + 89 ' 10 + 89 = 99 (changes the IntVarByRef itself - in the
Main Sub)
End Sub
```

Section 34.2: ByRef

Default modifier

If no modifier is specified for a parameter, that parameter is implicitly passed by reference.

```
Public Sub DoSomething1(foo As Long)
End Sub
```

```
Public Sub DoSomething2(ByRef foo As Long)
End Sub
```

The `foo` parameter is passed `ByRef` in both `DoSomething1` and `DoSomething2`.

> **Watch out!** If you're coming to VBA with experience from other languages, this is very likely the exact opposite behavior to the one you're used to. In many other programming languages (including VB.NET), the implicit/default modifier passes parameters by value.

Passing by reference

- When a *value* is passed `ByRef`, the procedure receives **a reference** to the value.

```
Public Sub Test()
    Dim foo As Long
    foo = 42
    DoSomething foo
    Debug.Print foo
End Sub

Private Sub DoSomething(ByRef foo As Long)
    foo = foo * 2
End Sub
```

Calling the above `Test` procedure outputs 84. `DoSomething` is given `foo` and receives a *reference* to the value, and therefore works with the same memory address as the caller.

- When a *reference* is passed `ByRef`, the procedure receives **a reference** to the pointer.

```
Public Sub Test()
    Dim foo As Collection
    Set foo = New Collection
    DoSomething foo
    Debug.Print foo.Count
End Sub

Private Sub DoSomething(ByRef foo As Collection)
    foo.Add 42
```

```
    Set foo = Nothing
End Sub
```

The above code raises run-time error 91, because the caller is calling the `Count` member of an object that no longer exists, because `DoSomething` was given a *reference* to the object pointer and assigned it to `Nothing` before returning.

Forcing ByVal at call site

Using parentheses at the call site, you can override `ByRef` and force an argument to be passed `ByVal`:

```
Public Sub Test()
    Dim foo As Long
    foo = 42
    DoSomething (foo)
    Debug.Print foo
End Sub

Private Sub DoSomething(ByRef foo As Long)
    foo = foo * 2
End Sub
```

The above code outputs 42, regardless of whether `ByRef` is specified implicitly or explicitly.

> **Watch out!** Because of this, using extraneous parentheses in procedure calls can easily introduce bugs. Pay attention to the whitespace between the procedure name and the argument list:
>
> ```
> bar = DoSomething(foo) 'function call, no whitespace; parens are part of args list
> DoSomething (foo) 'procedure call, notice whitespace; parens are NOT part of args list
> DoSomething foo 'procedure call does not force the foo parameter to be ByVal
> ```

Section 34.3: ByVal

Passing by value

- When a *value* is passed `ByVal`, the procedure receives **a copy** of the value.

```
Public Sub Test()
    Dim foo As Long
    foo = 42
    DoSomething foo
    Debug.Print foo
End Sub

Private Sub DoSomething(ByVal foo As Long)
    foo = foo * 2
End Sub
```

Calling the above `Test` procedure outputs 42. `DoSomething` is given `foo` and receives **a copy** of the value. The copy is multiplied by 2, and then discarded when the procedure exits; the caller's copy was never altered.

- When a *reference* is passed `ByVal`, the procedure receives **a copy** of the pointer.

```
Public Sub Test()
    Dim foo As Collection
    Set foo = New Collection
    DoSomething foo
    Debug.Print foo.Count
End Sub

Private Sub DoSomething(ByVal foo As Collection)
    foo.Add 42
    Set foo = Nothing
End Sub
```

Calling the above `Test` procedure outputs 1. `DoSomething` is given `foo` and receives *a copy* of **the pointer** to the `Collection` object. Because the `foo` object variable in the `Test` scope points to the same object, adding an item in `DoSomething` adds the item to the same object. Because it's *a copy* of the pointer, setting its reference to `Nothing` does not affect the caller's own copy.

Chapter 35: CreateObject vs. GetObject

Section 35.1: Demonstrating GetObject and CreateObject

MSDN-GetObject Function

> Returns a reference to an object provided by an ActiveX component.

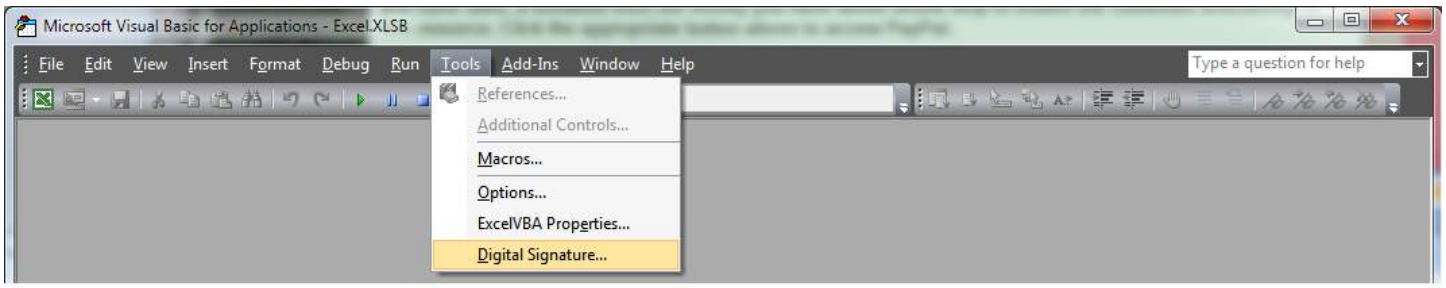
> Use the GetObject function when there is a current instance of the object or if you want to create the object with a file already loaded. If there is no current instance, and you don't want the object started with a file loaded, use the CreateObject function.

```
Sub CreateVSGet()
    Dim ThisXLApp As Excel.Application 'An example of early binding
    Dim AnotherXLApp As Object 'An example of late binding
    Dim ThisNewWB As Workbook
    Dim AnotherNewWB As Workbook
    Dim wb As Workbook

    'Get this instance of Excel
    Set ThisXLApp = GetObject(ThisWorkbook.Name).Application
    'Create another instance of Excel
    Set AnotherXLApp = CreateObject("Excel.Application")
    'Make the 2nd instance visible
    AnotherXLApp.Visible = True
    'Add a workbook to the 2nd instance
    Set AnotherNewWB = AnotherXLApp.Workbooks.Add
    'Add a sheet to the 2nd instance
    AnotherNewWB.Sheets.Add

    'You should now have 2 instances of Excel open
    'The 1st instance has 1 workbook: Book1
    'The 2nd instance has 1 workbook: Book2

    'Lets add another workbook to our 1st instance
    Set ThisNewWB = ThisXLApp.Workbooks.Add
    'Now loop through the workbooks and show their names
    For Each wb In ThisXLApp.Workbooks
        Debug.Print wb.Name
    Next
    'Now the 1st instance has 2 workbooks: Book1 and Book3
    'If you close the first instance of Excel,
    'Book1 and Book3 will close, but book2 will still be open

End Sub
```

Chapter 36: Macro security and signing of VBA-projects/-modules

Section 36.1: Create a valid digital self-signed certificate SELFCERT.EXE

To run macros and maintain the security Office applications provide against malicious code, it is necessary to digitally sign the VBAProject.OTM from the *VBA editor* > *Tools* > *Digital Signature*.



Office comes with a utility to create a self-signed digital certificate that you can employ on the PC to sign your projects.

This utility **SELFCERT.EXE** is in the Office program folder,

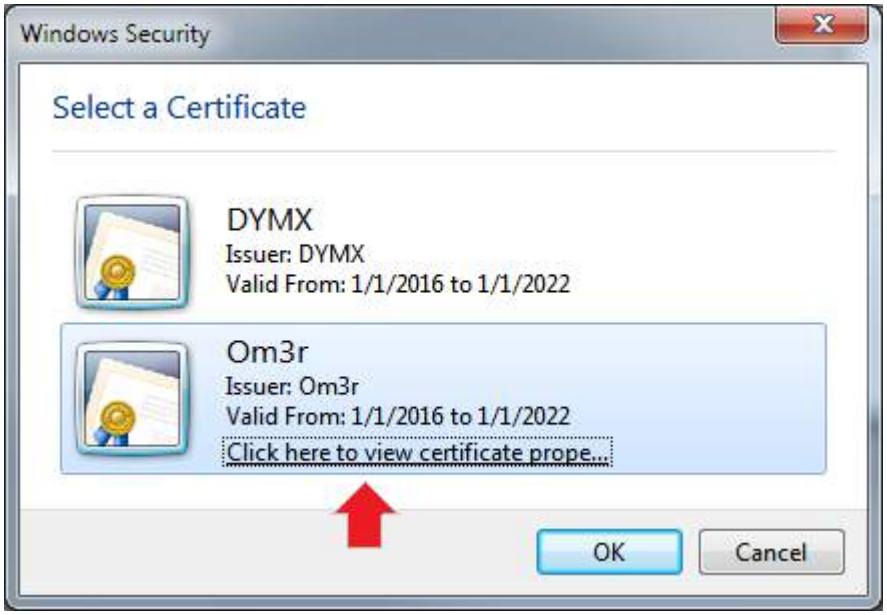
Click on Digital Certificate for VBA Projects to open the certificate *wizard*.

In the dialog enter a suitable name for the certificate and click OK.

If all goes well you will see a confirmation:

You can now close the **SELFCERT** wizard and turn your attention to the certificate you have created.

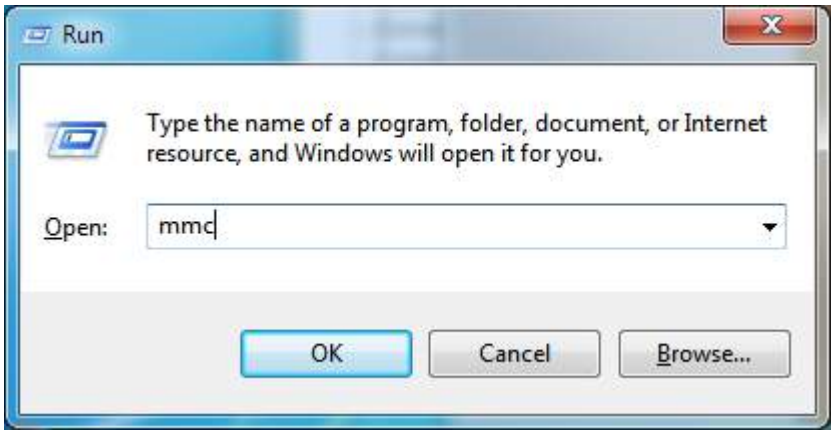
If you try to employ the certificate you have just created and you check its properties



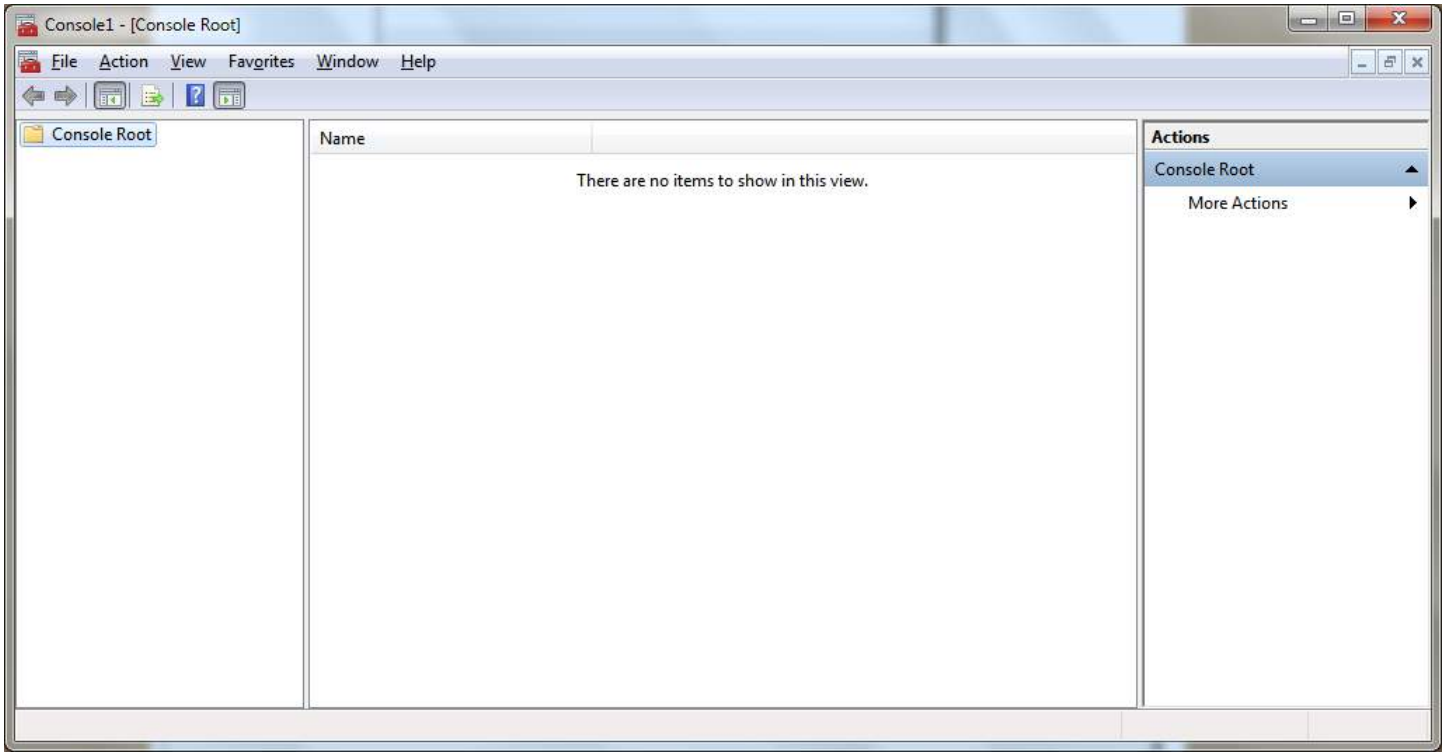
You will see that the certificate is not trusted and the reason is indicated in the dialog.

The certificate has been created in the Current User > Personal > Certificates store. It needs to go in Local Computer > Trusted Root Certificate Authorities > Certificates store, so you need to export from the former and import to the latter.

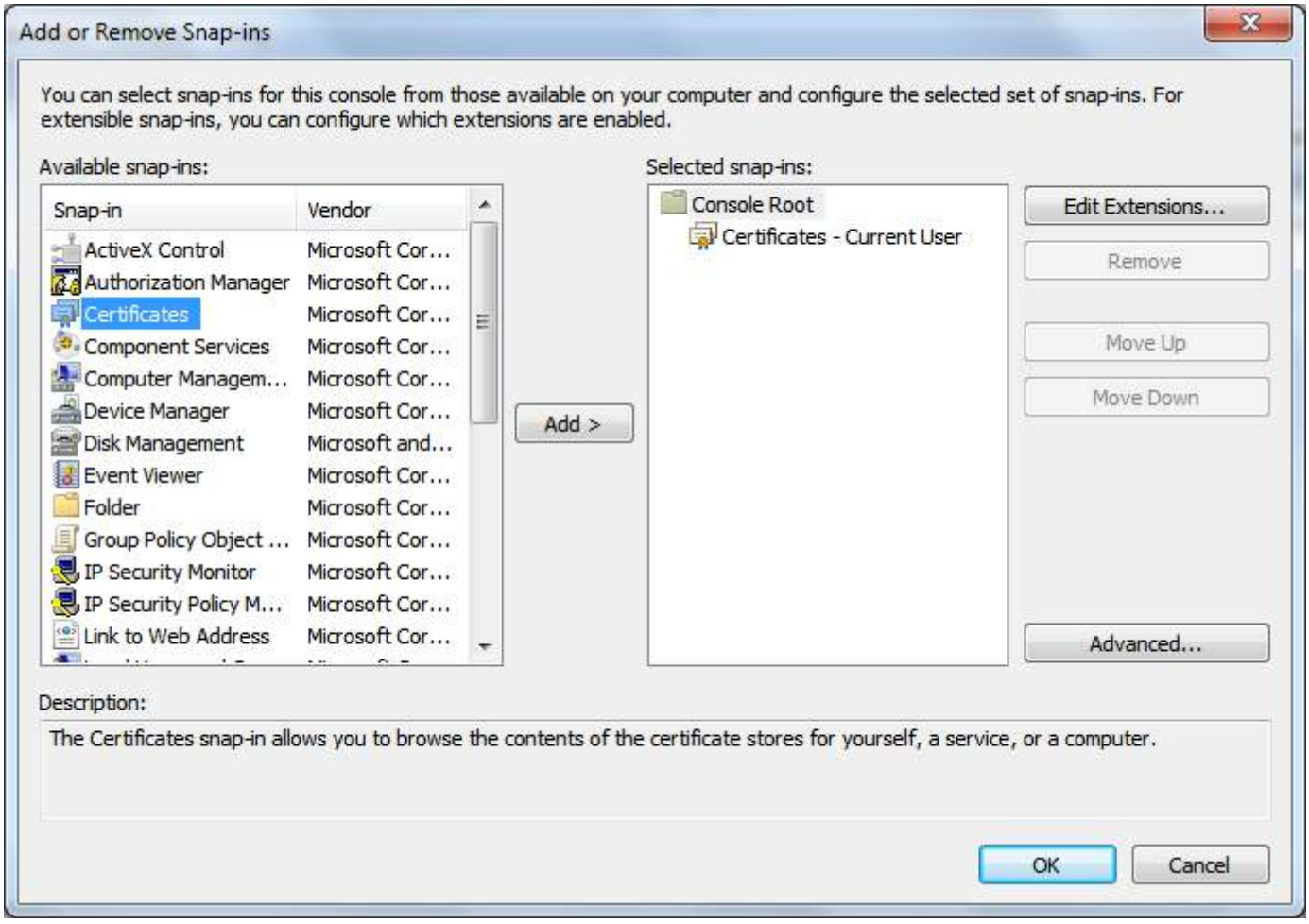
Pressing the Windows **Key+R** which will open the 'Run' Window. then Enter 'mmc' in the window as shown below and click 'OK '.



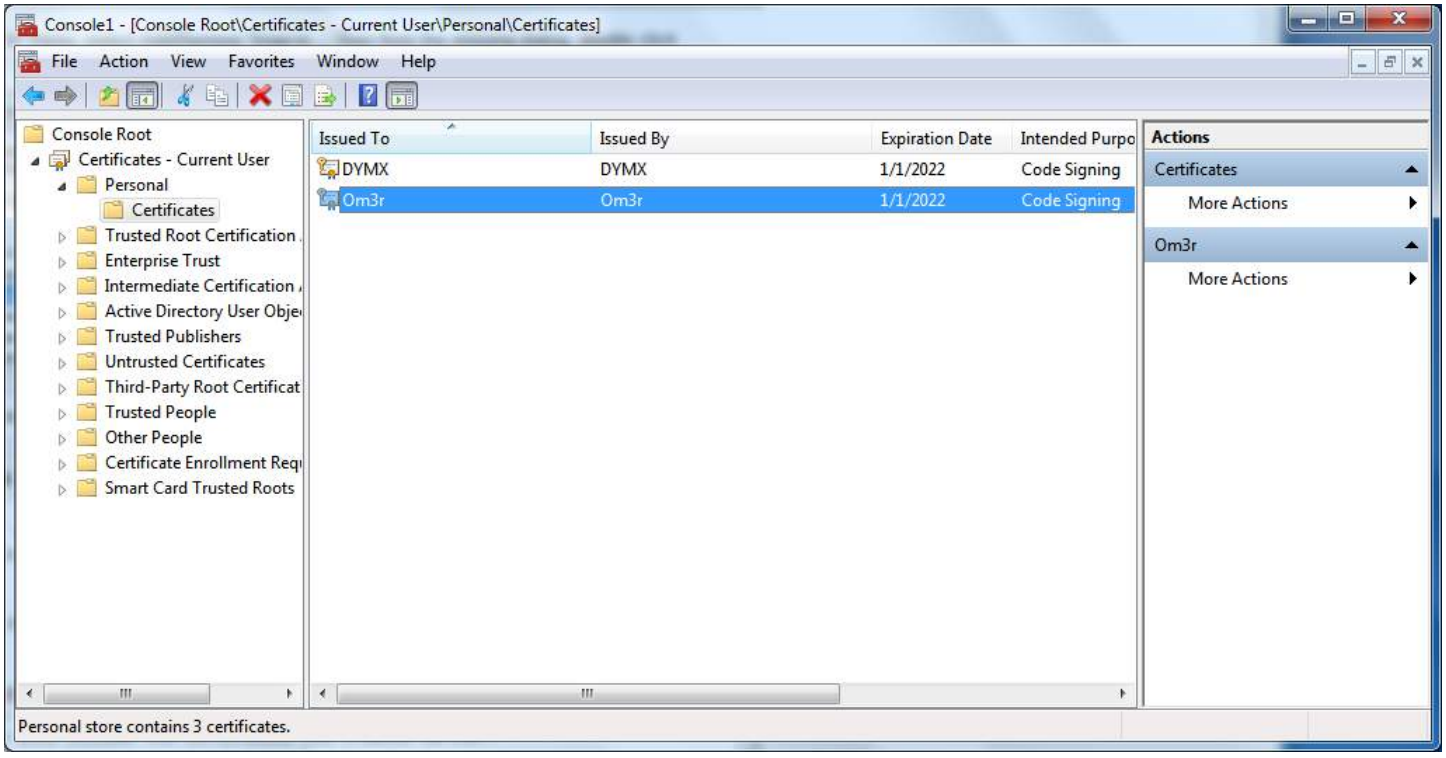
The Microsoft Management Console will open and look like the following.



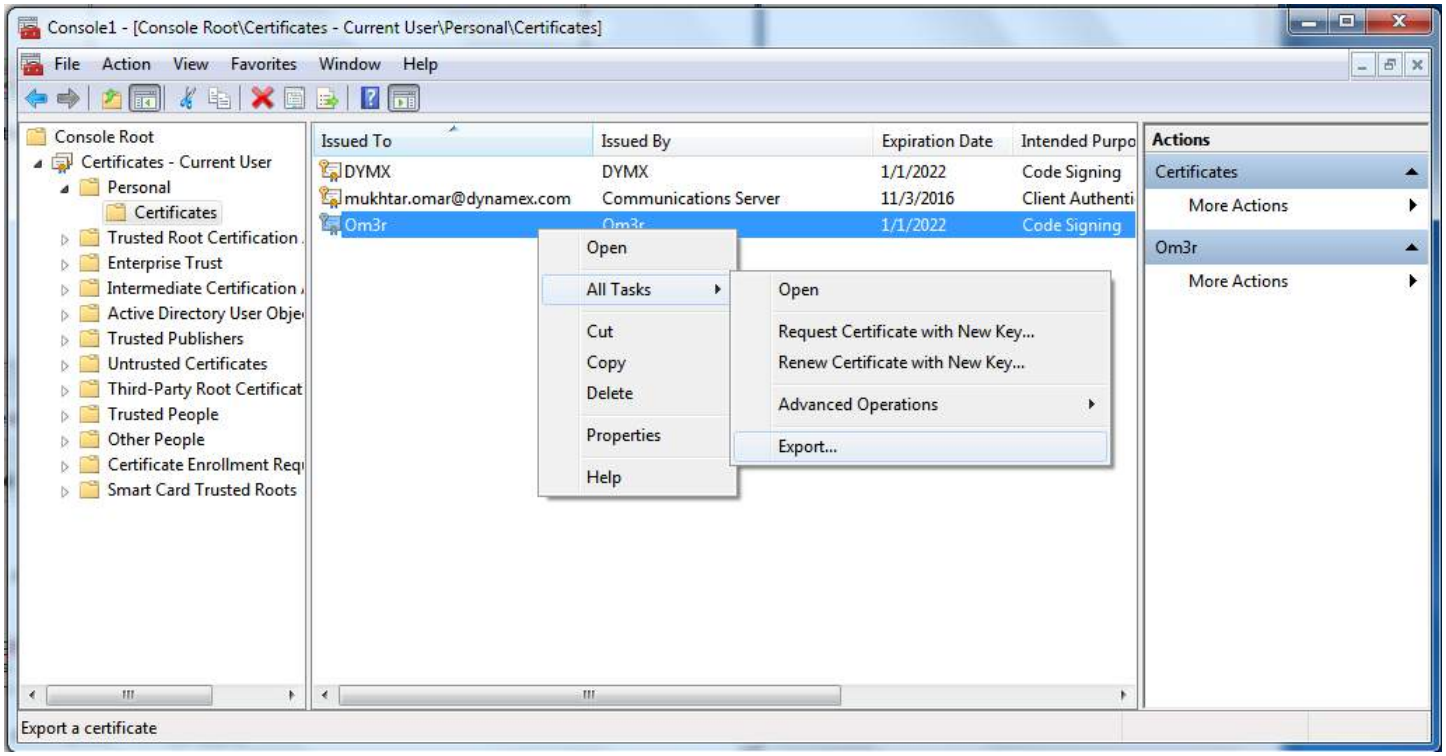
From the File menu, select Add/Remove Snap-in... Then from the ensuing dialog, double click Certificates and then click OK



Expand the dropdown in the left window for *Certificates - Current User*' and select certificates as shown below. The center panel will then show the certificates in that location, which will include the certificate you created earlier:

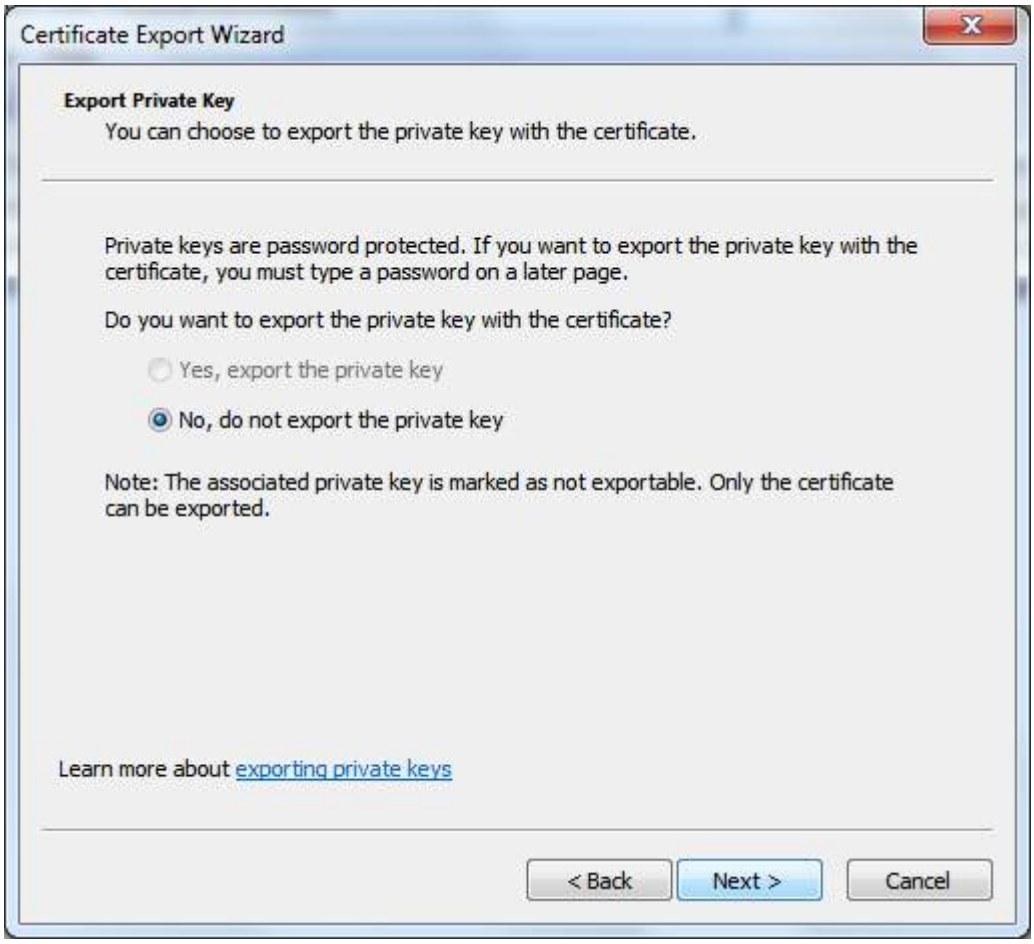


Right click the certificate and select All Tasks > Export:

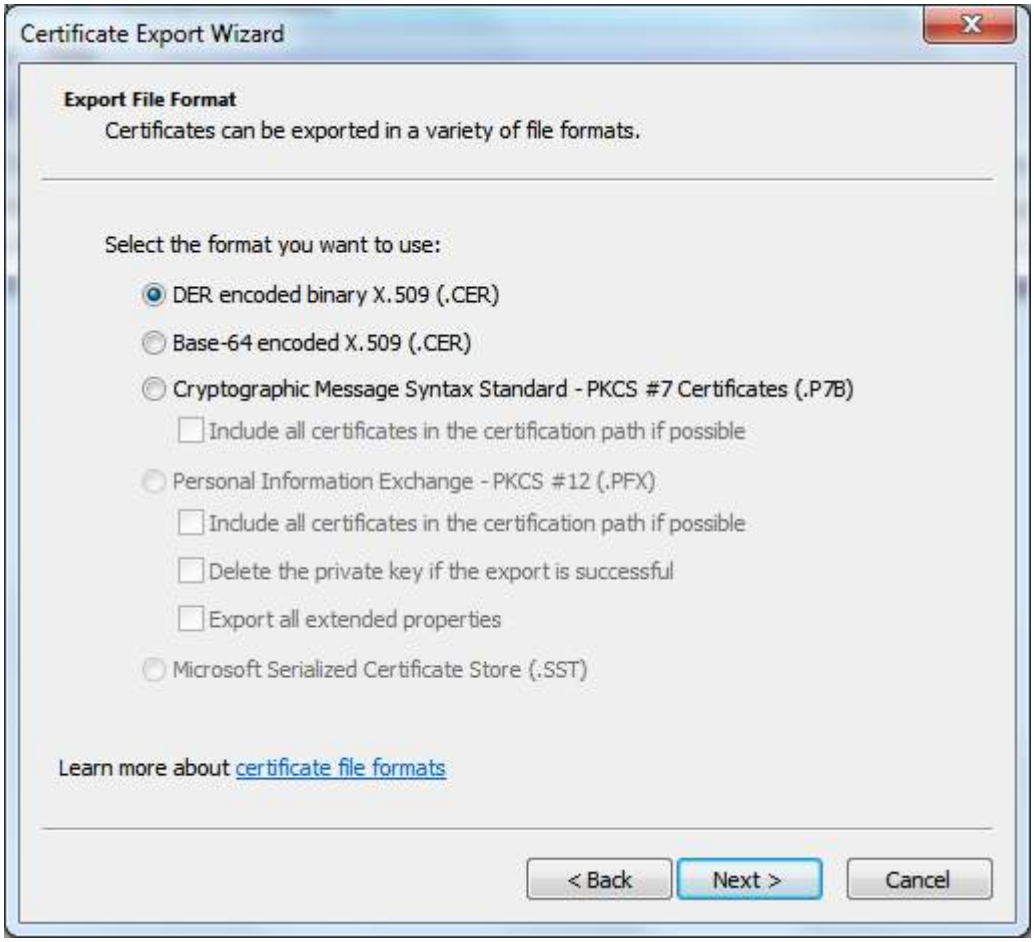


Export Wizard

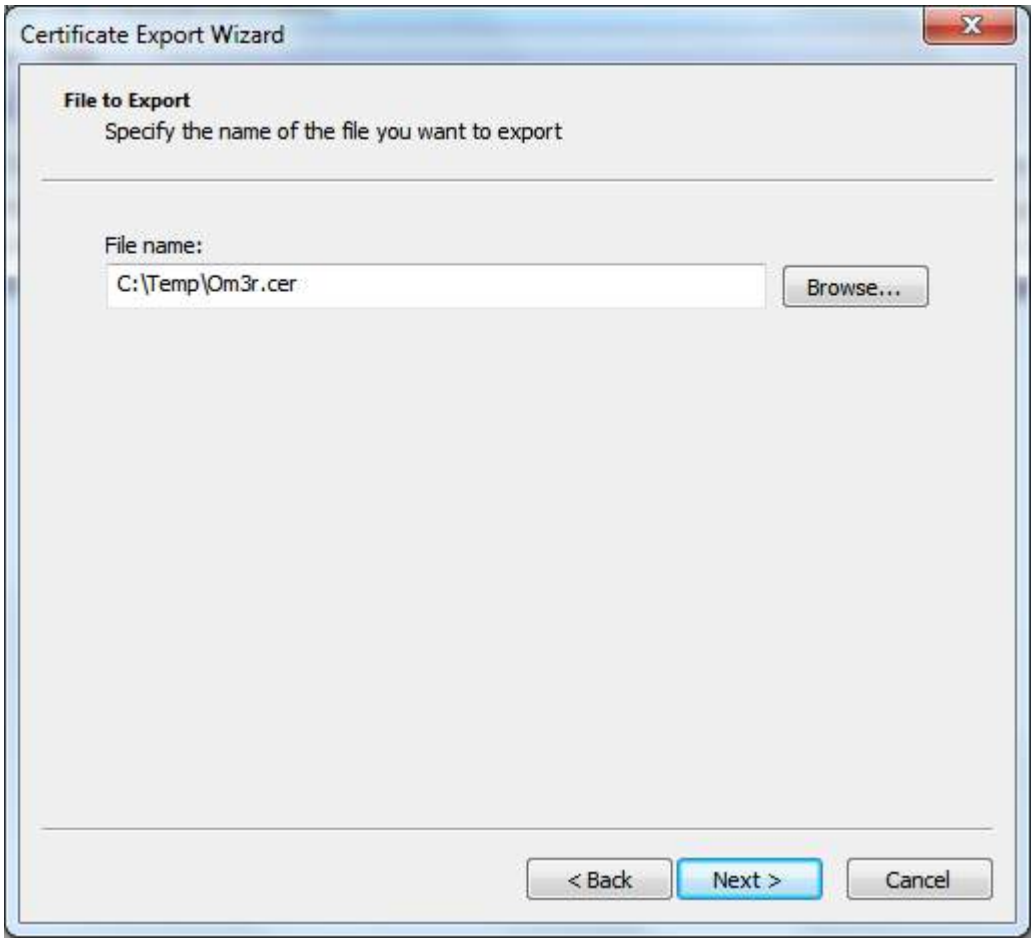
Click Next



the Only one pre-selected option will be available, so click 'Next' again:



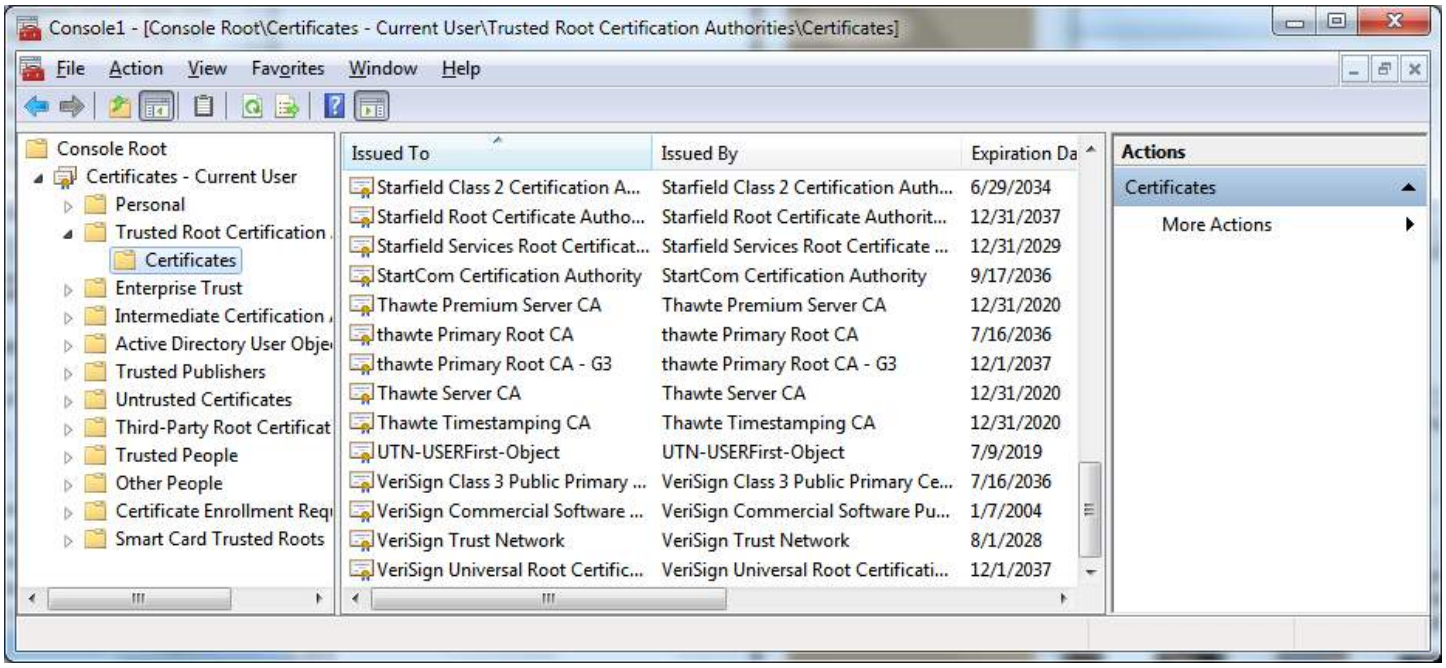
The top item will already be pre-selected. Click Next again and choose a name and location to save the exported certificate.



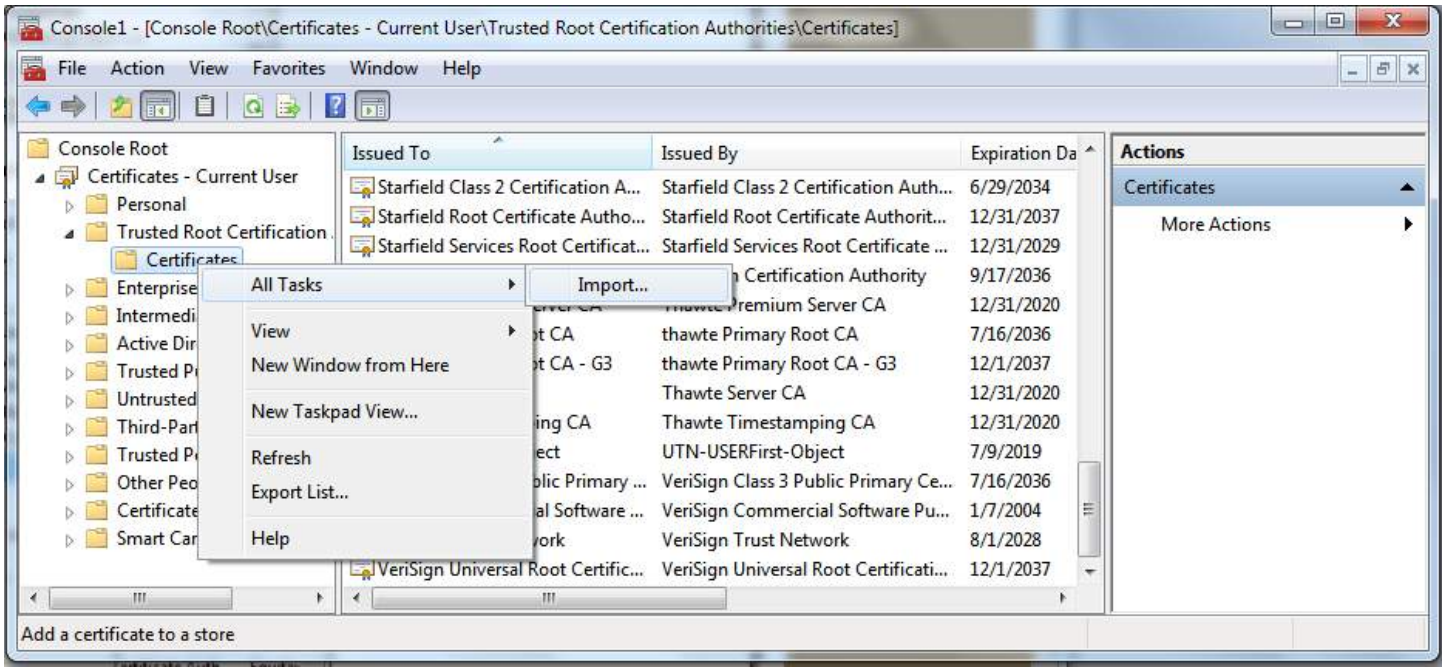
Click Next again to save the certificate

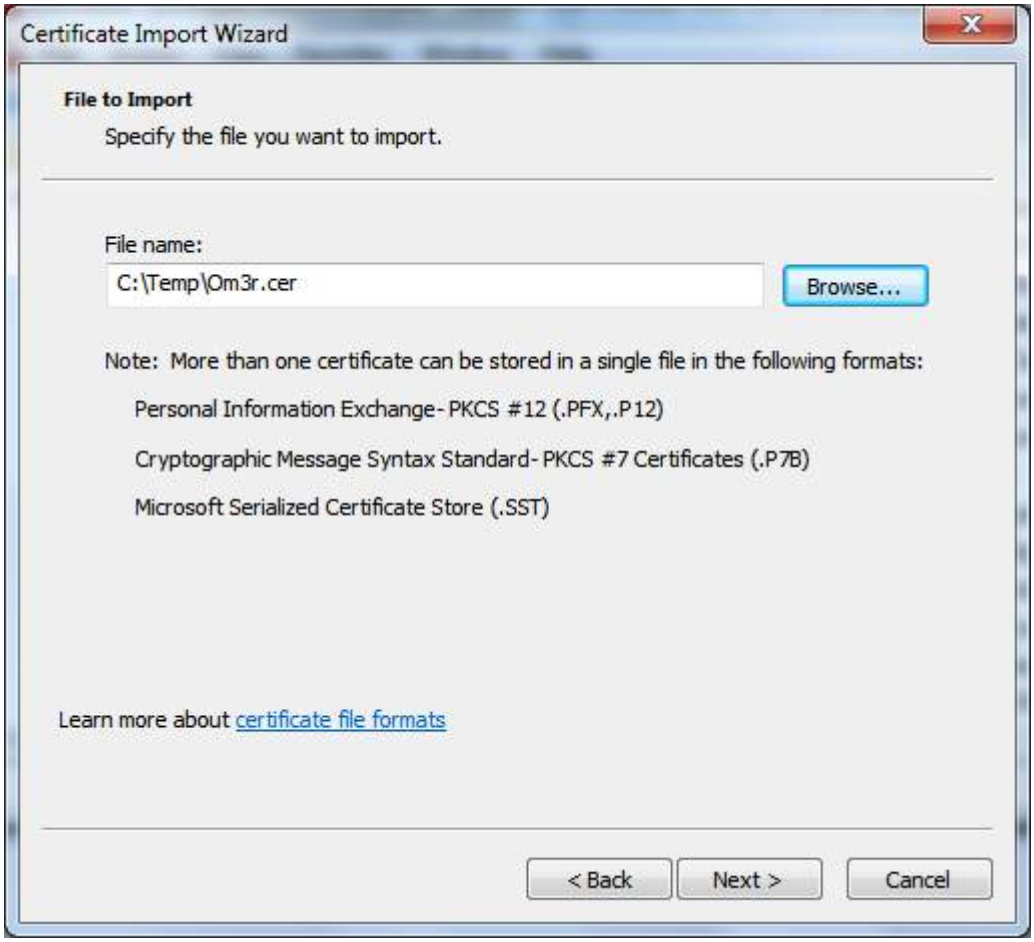
Once focus is returned to the Management Console.

Expand the *Certificates* menu and from the Trusted Root Certification Authorities menu, select *Certificates*.

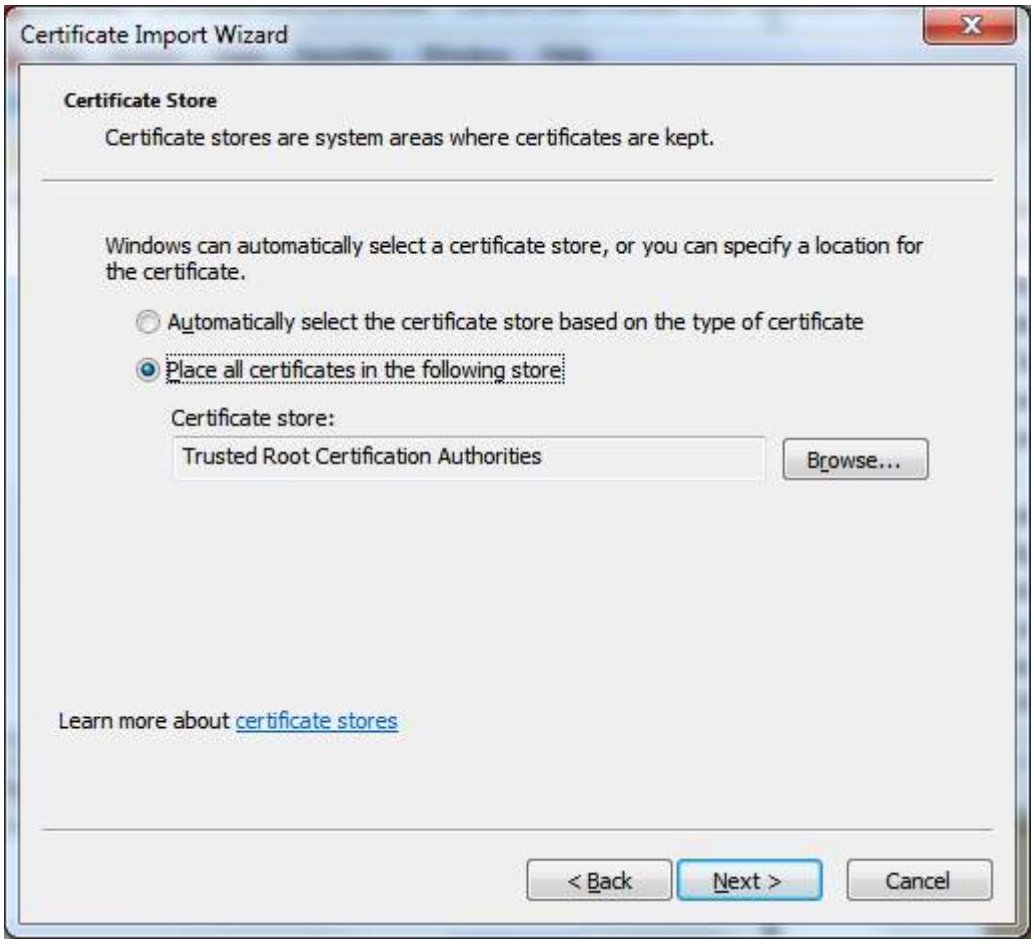


Right click. Select *All Tasks* and *Import*





Click next and Save to the *Trusted Root Certification Authorities store*:



Then Next > Finish, now close the Console.

If you now use the certificate and check its properties, you will see that it is a trusted certificate and you can use it to sign your project:

Chapter 37: Data Structures

[TODO: This topic should be an example of all the basic CS 101 data structures along with some explanation as an overview of how data structures can be implemented in VBA. This would be a good opportunity to tie in and reinforce concepts introduced in Class-related topics in VBA documentation.]

Section 37.1: Linked List

This linked list example implements Set abstract data type operations.

SinglyLinkedNode class

```
Option Explicit

Private Value As Variant
Private NextNode As SinglyLinkedNode '"Next" is a keyword in VBA and therefore is not a valid
variable name
```

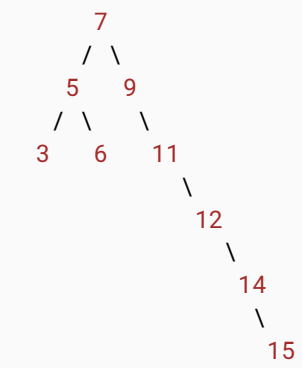
LinkedList class

```
Option Explicit

Private head As SinglyLinkedNode

'Set type operations

Public Sub Add(value As Variant)
    Dim node As SinglyLinkedNode

    Set node = New SinglyLinkedNode
    node.value = value
    Set node.nextNode = head

    Set head = node
End Sub

Public Sub Remove(value As Variant)
    Dim node As SinglyLinkedNode
    Dim prev As SinglyLinkedNode

    Set node = head

    While Not node Is Nothing
        If node.value = value Then
            'remove node
            If node Is head Then
                Set head = node.nextNode
            Else
                Set prev.nextNode = node.nextNode
            End If
            Exit Sub
        End If
        Set prev = node
        Set node = node.nextNode
    Wend

End Sub
```

```
Public Function Exists(value As Variant) As Boolean
    Dim node As SinglyLinkedNode

    Set node = head
    While Not node Is Nothing
        If node.value = value Then
            Exists = True
            Exit Function
        End If
        Set node = node.nextNode
    Wend
End Function

Public Function Count() As Long
    Dim node As SinglyLinkedNode

    Set node = head

    While Not node Is Nothing
        Count = Count + 1
        Set node = node.nextNode
    Wend

End Function
```

Section 37.2: Binary Tree

This is an example of an unbalanced binary search tree. A binary tree is structured conceptually as a hierarchy of nodes descending downward from a common root, where each node has two children: left and right. For example, suppose the numbers 7, 5, 9, 3, 11, 6, 12, 14 and 15 were inserted into a BinaryTree. The structure would be as below. Note that this binary tree is not balanced, which can be a desirable characteristic for guaranteeing the performance of lookups - see AVL trees for an example of a self-balancing binary search tree.



BinaryTreeNode class

```
Option Explicit

Public left As BinaryTreeNode
Public right As BinaryTreeNode
Public key As Variant
Public value As Variant
```

BinaryTree class

[TODO]

Chapter 38: Interfaces

An **Interface** is a way to define a set of behaviors that a class will perform. The definition of an interface is a list of method signatures (name, parameters, and return type). A class having all of the methods is said to "implement" that interface.

In VBA, using interfaces lets the compiler check that a module implements all of its methods. A variable or parameter can be defined in terms of an interface instead of a specific class.

Section 38.1: Multiple Interfaces in One Class - Flyable and Swimable

Using the `Flyable` example as a starting point, we can add a second interface, `Swimmable`, with the following code:

```
Sub Swim()
    ' No code
End Sub
```

The `Duck` object can `Implement` both flying and swimming:

```
Implements Flyable
Implements Swimmable

Public Sub Flyable_Fly()
    Debug.Print "Flying With Wings!"
End Sub

Public Function Flyable_GetAltitude() As Long
    Flyable_GetAltitude = 30
End Function

Public Sub Swimmable_Swim()
    Debug.Print "Floating on the water"
End Sub
```

A `Fish` class can implement `Swimmable`, too:

```
Implements Swimmable

Public Sub Swimmable_Swim()
    Debug.Print "Swimming under the water"
End Sub
```

Now, we can see that the `Duck` object can be passed to a Sub as a `Flyable` on one hand, and a `Swimmable` on the other:

```
Sub InterfaceTest()

Dim MyDuck As New Duck
Dim MyAirplane As New Airplane
Dim MyFish As New Fish

Debug.Print "Fly Check..."

FlyAndCheckAltitude MyDuck
FlyAndCheckAltitude MyAirplane
```

```
Debug.Print "Swim Check..."

TrySwimming MyDuck
TrySwimming MyFish

End Sub

Public Sub FlyAndCheckAltitude(F As Flyable)
    F.Fly
    Debug.Print F.GetAltitude
End Sub

Public Sub TrySwimming(S As Swimmable)
    S.Swim
End Sub
```

The output of this code is:

> Fly Check...
>
> Flying With Wings!
>
> 30
>
> Flying With Jet Engines!
>
> 10000
>
> Swim Check...
>
> Floating on the water
>
> Swimming under the water

Section 38.2: Simple Interface - Flyable

The interface `Flyable` is a class module with the following code:

```
Public Sub Fly()
    ' No code.
End Sub

Public Function GetAltitude() As Long
    ' No code.
End Function
```

A class module, `Airplane`, uses the **`Implements`** keyword to tell the compiler to raise an error unless it has two methods: a `Flyable_Fly()` sub and a `Flyable_GetAltitude()` function that returns a `Long`.

```
Implements Flyable

Public Sub Flyable_Fly()
    Debug.Print "Flying With Jet Engines!"
End Sub

Public Function Flyable_GetAltitude() As Long
    Flyable_GetAltitude = 10000
```

```
End Function
```

A second class module, `Duck`, also implements `Flyable`:

```
Implements Flyable

Public Sub Flyable_Fly()
    Debug.Print "Flying With Wings!"
End Sub

Public Function Flyable_GetAltitude() As Long
    Flyable_GetAltitude = 30
End Function
```

We can write a routine that accepts any `Flyable` value, knowing that it will respond to a command of `Fly` or `GetAltitude`:

```
Public Sub FlyAndCheckAltitude(F As Flyable)
    F.Fly
    Debug.Print F.GetAltitude
End Sub
```

Because the interface is defined, the IntelliSense popup window will show `Fly` and `GetAltitude` for `F`.

When we run the following code:

```
Dim MyDuck As New Duck
Dim MyAirplane As New Airplane

FlyAndCheckAltitude MyDuck
FlyAndCheckAltitude MyAirplane
```

The output is:

```
Flying With Wings!
30
Flying With Jet Engines!
10000
```

Note that even though the subroutine is named `Flyable_Fly` in both `Airplane` and `Duck`, it can be called as `Fly` when the variable or parameter is defined as `Flyable`. If the variable is defined specifically as a `Duck`, it would have to be called as `Flyable_Fly`.

Chapter 39: Reading 2GB+ files in binary in VBA and File Hashes

There is a built in easy way to read files in binary within VBA, however it has a restriction of 2GB (2,147,483,647 bytes - max of Long data type). As technology evolves, this 2GB limit is easily breached. e.g. an ISO image of Operating System install DVD disc. Microsoft does provide a way to overcome this via low level Windows API and here is a backup of it.

Also demonstrate (Read part) for calculating File Hashes without external program like `fciv.exe` from Microsoft.

Section 39.1: This have to be in a Class module, examples later referred as "Random"

```
' How To Seek Past VBA's 2GB File Limit
' Source: https://support.microsoft.com/en-us/kb/189981 (Archived)
' This must be in a Class Module

Option Explicit

Public Enum W32F_Errors
    W32F_UNKNOWN_ERROR = 45600
    W32F_FILE_ALREADY_OPEN
    W32F_PROBLEM_OPENING_FILE
    W32F_FILE_ALREADY_CLOSED
    W32F_Problem_seeking
End Enum

Private Const W32F_SOURCE = "Win32File Object"
Private Const GENERIC_WRITE = &H40000000
Private Const GENERIC_READ = &H80000000
Private Const FILE_ATTRIBUTE_NORMAL = &H80
Private Const CREATE_ALWAYS = 2
Private Const OPEN_ALWAYS = 4
Private Const INVALID_HANDLE_VALUE = -1

Private Const FILE_BEGIN = 0, FILE_CURRENT = 1, FILE_END = 2

Private Const FORMAT_MESSAGE_FROM_SYSTEM = &H1000

Private Declare Function FormatMessage Lib "kernel32" Alias "FormatMessageA" ( _
    ByVal dwFlags As Long, _
    lpSource As Long, _
    ByVal dwMessageId As Long, _
    ByVal dwLanguageId As Long, _
    ByVal lpBuffer As String, _
    ByVal nSize As Long, _
    Arguments As Any) As Long

Private Declare Function ReadFile Lib "kernel32" ( _
    ByVal hFile As Long, _
    lpBuffer As Any, _
    ByVal nNumberOfBytesToRead As Long, _
    lpNumberOfBytesRead As Long, _
    ByVal lpOverlapped As Long) As Long

Private Declare Function CloseHandle Lib "kernel32" (ByVal hObject As Long) As Long

Private Declare Function WriteFile Lib "kernel32" ( _
```

```
    ByVal hFile As Long, _
    lpBuffer As Any, _
    ByVal nNumberOfBytesToWrite As Long, _
    lpNumberOfBytesWritten As Long, _
    ByVal lpOverlapped As Long) As Long

Private Declare Function CreateFile Lib "kernel32" Alias "CreateFileA" ( _
    ByVal lpFileName As String, _
    ByVal dwDesiredAccess As Long, _
    ByVal dwShareMode As Long, _
    ByVal lpSecurityAttributes As Long, _
    ByVal dwCreationDisposition As Long, _
    ByVal dwFlagsAndAttributes As Long, _
    ByVal hTemplateFile As Long) As Long

Private Declare Function SetFilePointer Lib "kernel32" ( _
    ByVal hFile As Long, _
    ByVal lDistanceToMove As Long, _
    lpDistanceToMoveHigh As Long, _
    ByVal dwMoveMethod As Long) As Long

Private Declare Function FlushFileBuffers Lib "kernel32" (ByVal hFile As Long) As Long

Private hFile As Long, sFName As String, fAutoFlush As Boolean

Public Property Get FileHandle() As Long
    If hFile = INVALID_HANDLE_VALUE Then
        RaiseError W32F_FILE_ALREADY_CLOSED
    End If
    FileHandle = hFile
End Property

Public Property Get FileName() As String
    If hFile = INVALID_HANDLE_VALUE Then
        RaiseError W32F_FILE_ALREADY_CLOSED
    End If
    FileName = sFName
End Property

Public Property Get IsOpen() As Boolean
    IsOpen = hFile <> INVALID_HANDLE_VALUE
End Property

Public Property Get AutoFlush() As Boolean
    If hFile = INVALID_HANDLE_VALUE Then
        RaiseError W32F_FILE_ALREADY_CLOSED
    End If
    AutoFlush = fAutoFlush
End Property

Public Property Let AutoFlush(ByVal NewVal As Boolean)
    If hFile = INVALID_HANDLE_VALUE Then
        RaiseError W32F_FILE_ALREADY_CLOSED
    End If
    fAutoFlush = NewVal
End Property

Public Sub OpenFile(ByVal sFileName As String)
    If hFile <> INVALID_HANDLE_VALUE Then
        RaiseError W32F_FILE_ALREADY_OPEN, sFName
    End If
    hFile = CreateFile(sFileName, GENERIC_WRITE Or GENERIC_READ, 0, 0, OPEN_ALWAYS,
```

```
FILE_ATTRIBUTE_NORMAL, 0)
    If hFile = INVALID_HANDLE_VALUE Then
        RaiseError W32F_PROBLEM_OPENING_FILE, sFileName
    End If
    sFName = sFileName
End Sub

Public Sub CloseFile()
    If hFile = INVALID_HANDLE_VALUE Then
        RaiseError W32F_FILE_ALREADY_CLOSED
    End If
    CloseHandle hFile
    sFName = ""
    fAutoFlush = False
    hFile = INVALID_HANDLE_VALUE
End Sub

Public Function ReadBytes(ByVal ByteCount As Long) As Variant
    Dim BytesRead As Long, Bytes() As Byte
    If hFile = INVALID_HANDLE_VALUE Then
        RaiseError W32F_FILE_ALREADY_CLOSED
    End If
    ReDim Bytes(0 To ByteCount - 1) As Byte
    ReadFile hFile, Bytes(0), ByteCount, BytesRead, 0
    ReadBytes = Bytes
End Function

Public Sub WriteBytes(DataBytes() As Byte)
    Dim fSuccess As Long, BytesToWrite As Long, BytesWritten As Long
    If hFile = INVALID_HANDLE_VALUE Then
        RaiseError W32F_FILE_ALREADY_CLOSED
    End If
    BytesToWrite = UBound(DataBytes) - LBound(DataBytes) + 1
    fSuccess = WriteFile(hFile, DataBytes(LBound(DataBytes)), BytesToWrite, BytesWritten, 0)
    If fAutoFlush Then Flush
End Sub

Public Sub Flush()
    If hFile = INVALID_HANDLE_VALUE Then
        RaiseError W32F_FILE_ALREADY_CLOSED
    End If
    FlushFileBuffers hFile
End Sub

Public Sub SeekAbsolute(ByVal HighPos As Long, ByVal LowPos As Long)
    If hFile = INVALID_HANDLE_VALUE Then
        RaiseError W32F_FILE_ALREADY_CLOSED
    End If
    LowPos = SetFilePointer(hFile, LowPos, HighPos, FILE_BEGIN)
End Sub

Public Sub SeekRelative(ByVal Offset As Long)
    Dim TempLow As Long, TempErr As Long
    If hFile = INVALID_HANDLE_VALUE Then
        RaiseError W32F_FILE_ALREADY_CLOSED
    End If
    TempLow = SetFilePointer(hFile, Offset, ByVal 0&, FILE_CURRENT)
    If TempLow = -1 Then
        TempErr = Err.LastDllError
        If TempErr Then
            RaiseError W32F_Problem_seeking, "Error " & TempErr & "." & vbCrLf & CStr(TempErr)
        End If
```

```
    End If
End Sub

Private Sub Class_Initialize()
    hFile = INVALID_HANDLE_VALUE
End Sub

Private Sub Class_Terminate()
    If hFile <> INVALID_HANDLE_VALUE Then CloseHandle hFile
End Sub

Private Sub RaiseError(ByVal ErrorCode As W32F_Errors, Optional sExtra)
    Dim Win32Err As Long, Win32Text As String
    Win32Err = Err.LastDllError
    If Win32Err Then
        Win32Text = vbCrLf & "Error " & Win32Err & vbCrLf & _
        DecodeAPIErrors(Win32Err)
    End If
    Select Case ErrorCode
        Case W32F_FILE_ALREADY_OPEN
            Err.Raise W32F_FILE_ALREADY_OPEN, W32F_SOURCE, "The file '" & sExtra & "' is already
open." & Win32Text
        Case W32F_PROBLEM_OPENING_FILE
            Err.Raise W32F_PROBLEM_OPENING_FILE, W32F_SOURCE, "Error opening '" & sExtra & "'." &
Win32Text
        Case W32F_FILE_ALREADY_CLOSED
            Err.Raise W32F_FILE_ALREADY_CLOSED, W32F_SOURCE, "There is no open file."
        Case W32F_Problem_seeking
            Err.Raise W32F_Problem_seeking, W32F_SOURCE, "Seek Error." & vbCrLf & sExtra
        Case Else
            Err.Raise W32F_UNKNOWN_ERROR, W32F_SOURCE, "Unknown error." & Win32Text
    End Select
End Sub

Private Function DecodeAPIErrors(ByVal ErrorCode As Long) As String
    Dim sMessage As String, MessageLength As Long
    sMessage = Space$(256)
    MessageLength = FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, 0&, ErrorCode, 0&, sMessage, 256&,
0&)
    If MessageLength > 0 Then
        DecodeAPIErrors = Left(sMessage, MessageLength)
    Else
        DecodeAPIErrors = "Unknown Error."
    End If
End Function
```

Section 39.2: Code for Calculating File Hash in a Standard module

```
Private Const HashTypeMD5 As String = "MD5" '
https://msdn.microsoft.com/en-us/library/system.security.cryptography.md5cryptoserviceprovider(v=vs.1
10).aspx
Private Const HashTypeSHA1 As String = "SHA1" '
https://msdn.microsoft.com/en-us/library/system.security.cryptography.sha1cryptoserviceprovider(v=vs.
110).aspx
Private Const HashTypeSHA256 As String = "SHA256" '
https://msdn.microsoft.com/en-us/library/system.security.cryptography.sha256cryptoserviceprovider(v=v
s.110).aspx
Private Const HashTypeSHA384 As String = "SHA384" '
https://msdn.microsoft.com/en-us/library/system.security.cryptography.sha384cryptoserviceprovider(v=v
s.110).aspx
```

```
Private Const HashTypeSHA512 As String = "SHA512" '
https://msdn.microsoft.com/en-us/library/system.security.cryptography.sha512cryptoserviceprovider(v=vs.110).aspx

Private uFileSize As Double ' Comment out if not testing performance by FileHashes()

Sub FileHashes()
    Dim tStart As Date, tFinish As Date, sHash As String, aTestFiles As Variant, oTestFile As
Variant, aBlockSizes As Variant, oBlockSize As Variant
    Dim BLOCKSIZE As Double

    ' This performs performance testing on different file sizes and block sizes
    aBlockSizes = Array("2^12-1", "2^13-1", "2^14-1", "2^15-1", "2^16-1", "2^17-1", "2^18-1",
"2^19-1", "2^20-1", "2^21-1", "2^22-1", "2^23-1", "2^24-1", "2^25-1", "2^26-1")
    aTestFiles = Array("C:\ISO\clonezilla-live-2.2.2-37-amd64.iso",
"C:\ISO\HPIP201.2014_0902.29.iso", "C:\ISO\SW_DVD5_Windows_Vista_Business_W32_32BIT_English.ISO",
"C:\ISO\Win10_1607_English_x64.iso",
"C:\ISO\SW_DVD9_Windows_Svr_Std_and_DataCtr_2012_R2_64Bit_English.ISO")
    Debug.Print "Test files: " & Join(aTestFiles, " | ")
    Debug.Print "BlockSizes: " & Join(aBlockSizes, " | ")
    For Each oTestFile In aTestFiles
        Debug.Print oTestFile
        For Each oBlockSize In aBlockSizes
            BLOCKSIZE = Evaluate(oBlockSize)
            tStart = Now
            sHash = GetFileHash(CStr(oTestFile), BLOCKSIZE, HashTypeMD5)
            tFinish = Now
            Debug.Print sHash, uFileSize, Format(tFinish - tStart, "hh:mm:ss"), oBlockSize & " (" &
BLOCKSIZE & ")"
        Next
    Next
End Sub

Private Function GetFileHash(ByVal sFile As String, ByVal uBlockSize As Double, ByVal sHashType As
String) As String
    Dim oFSO As Object ' "Scripting.FileSystemObject"
    Dim oCSP As Object ' One of the "CryptoServiceProvider"
    Dim oRnd As Random ' "Random" Class by Microsoft, must be in the same file
    Dim uBytesRead As Double, uBytesToRead As Double, bDone As Boolean
    Dim aBlock() As Byte, aBytes As Variant ' Arrays to store bytes
    Dim aHash() As Byte, sHash As String, i As Long
    'Dim uFileSize As Double ' Un-Comment if GetFileHash() is to be used individually

    Set oRnd = New Random ' Class by Microsoft: Random
    Set oFSO = CreateObject("Scripting.FileSystemObject")
    Set oCSP = CreateObject("System.Security.Cryptography." & sHashType & "CryptoServiceProvider")

    If oFSO Is Nothing Or oRnd Is Nothing Or oCSP Is Nothing Then
        MsgBox "One or more required objects cannot be created"
        GoTo CleanUp
    End If

    uFileSize = oFSO.GetFile(sFile).Size ' FILELEN() has 2GB max!
    uBytesRead = 0
    bDone = False
    sHash = String(oCSP.HashSize / 4, "0") ' Each hexadecimal has 4 bits

    Application.ScreenUpdating = False
    ' Process the file in chunks of uBlockSize or less
    If uFileSize = 0 Then
        ReDim aBlock(0)
        oCSP.TransformFinalBlock aBlock, 0, 0
```

```
        bDone = True
    Else
        With oRnd
            .OpenFile sFile
            Do
                If uBytesRead + uBlockSize < uFileSize Then
                    uBytesToRead = uBlockSize
                Else
                    uBytesToRead = uFileSize - uBytesRead
                    bDone = True
                End If
                ' Read in some bytes
                aBytes = .ReadBytes(uBytesToRead)
                aBlock = aBytes
                If bDone Then
                    oCSP.TransformFinalBlock aBlock, 0, uBytesToRead
                    uBytesRead = uBytesRead + uBytesToRead
                Else
                    uBytesRead = uBytesRead + oCSP.TransformBlock(aBlock, 0, uBytesToRead, aBlock,
0)
                End If
                DoEvents
            Loop Until bDone
            .CloseFile
        End With
    End If
    If bDone Then
        ' convert Hash byte array to an hexadecimal string
        aHash = oCSP.hash
        For i = 0 To UBound(aHash)
            Mid$(sHash, i * 2 + (aHash(i) > 15) + 2) = Hex(aHash(i))
        Next
    End If
    Application.ScreenUpdating = True
    ' Clean up
    oCSP.Clear
CleanUp:
    Set oFSO = Nothing
    Set oRnd = Nothing
    Set oCSP = Nothing
    GetFileHash = sHash
End Function
```

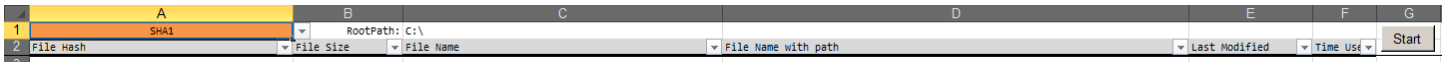
> The output is pretty interesting, my test files indicates that **`BLOCKSIZE = 131071` (2^17-1)** gives overall best performance with 32bit Office 2010 on Windows 7 x64, next best is *2^16-1 (65535)*. Note `2^27-1` yields *Out of memory*.

File Size (bytes)	File Name
146,800,640	clonezilla-live-2.2.2-37-amd64.iso
798,210,048	HPIP201.2014_0902.29.iso
2,073,016,320	SW_DVD5_Windows_Vista_Business_W32_32BIT_English.ISO
4,380,387,328	Win10_1607_English_x64.iso
5,400,115,200	SW_DVD9_Windows_Svr_Std_and_DataCtr_2012_R2_64Bit_English.ISO

Section 39.3: Calculating all Files Hash from a root Folder

Another variation from the code above gives you more performance when you want to get hash codes of all files from a root folder including all sub folders.

Example of Worksheet:



Code

```
Option Explicit

Private Const HashTypeMD5 As String = "MD5" '
https://msdn.microsoft.com/en-us/library/system.security.cryptography.md5cryptoserviceprovider(v=vs.1
10).aspx
Private Const HashTypeSHA1 As String = "SHA1" '
https://msdn.microsoft.com/en-us/library/system.security.cryptography.sha1cryptoserviceprovider(v=vs.
110).aspx
Private Const HashTypeSHA256 As String = "SHA256" '
https://msdn.microsoft.com/en-us/library/system.security.cryptography.sha256cryptoserviceprovider(v=v
s.110).aspx
Private Const HashTypeSHA384 As String = "SHA384" '
https://msdn.microsoft.com/en-us/library/system.security.cryptography.sha384cryptoserviceprovider(v=v
s.110).aspx
Private Const HashTypeSHA512 As String = "SHA512" '
https://msdn.microsoft.com/en-us/library/system.security.cryptography.sha512cryptoserviceprovider(v=v
s.110).aspx

Private Const BLOCKSIZE As Double = 131071 ' 2^17-1

Private oFSO As Object
Private oCSP As Object
Private oRnd As Random ' Requires the Class from Microsoft
https://support.microsoft.com/en-us/kb/189981
Private sHashType As String
Private sRootFDR As String
Private oRng As Range
Private uFileCount As Double

Sub AllFileHashes() ' Active-X button calls this
    Dim oWS As Worksheet
    ' | A: FileHash | B: FileSize | C: FileName | D: FilaName and Path | E: File Last Modification
Time | F: Time required to calculate has code (seconds)
    With ThisWorkbook
        ' Clear All old entries on all worksheets
        For Each oWS In .Worksheets
            Set oRng = Intersect(oWS.UsedRange, oWS.UsedRange.Offset(2))
            If Not oRng Is Nothing Then oRng.ClearContents
        Next
        With .Worksheets(1)
            sHashType = Trim(.Range("A1").Value) ' Range(A1)
            sRootFDR = Trim(.Range("C1").Value) ' Range(C1) Column B for file size
            If Len(sHashType) = 0 Or Len(sRootFDR) = 0 Then Exit Sub
            Set oRng = .Range("A3") ' First entry on First Page
        End With
    End With

    uFileCount = 0
    If oRnd Is Nothing Then Set oRnd = New Random ' Class by Microsoft: Random
    If oFSO Is Nothing Then Set oFSO = CreateObject("Scripting.FileSystemObject") ' Just to get
correct FileSize
    If oCSP Is Nothing Then Set oCSP = CreateObject("System.Security.Cryptography." & sHashType &
"CryptoServiceProvider")

    ProcessFolder oFSO.GetFolder(sRootFDR)
```

```
    Application.StatusBar = False
    Application.ScreenUpdating = True
    oCSP.Clear
    Set oCSP = Nothing
    Set oRng = Nothing
    Set oFSO = Nothing
    Set oRnd = Nothing
    Debug.Print "Total file count: " & uFileCount
End Sub

Private Sub ProcessFolder(ByRef oFDR As Object)
    Dim oFile As Object, oSubFDR As Object, sHash As String, dStart As Date, dFinish As Date
    Application.ScreenUpdating = False
    For Each oFile In oFDR.Files
        uFileCount = uFileCount + 1
        Application.StatusBar = uFileCount & ": " & Right(oFile.Path, 255 - Len(uFileCount) - 2)
        oCSP.Initialize ' Reinitialize the CryptoServiceProvider
        dStart = Now
        sHash = GetFileHash(oFile, BLOCKSIZE, sHashType)
        dFinish = Now
        With oRng
            .Value = sHash
            .Offset(0, 1).Value = oFile.Size ' File Size in bytes
            .Offset(0, 2).Value = oFile.Name ' File name with extension
            .Offset(0, 3).Value = oFile.Path ' Full File name and Path
            .Offset(0, 4).Value = FileDateTime(oFile.Path) ' Last modification timestamp of file
            .Offset(0, 5).Value = dFinish - dStart ' Time required to calculate hash code
        End With
        If oRng.Row = Rows.Count Then
            ' Max rows reached, start on Next sheet
            If oRng.Worksheet.Index + 1 > ThisWorkbook.Worksheets.Count Then
                MsgBox "All rows in all worksheets have been used, please create more sheets"
                End
            End If
            Set oRng = ThisWorkbook.Sheets(oRng.Worksheet.Index + 1).Range("A3")
            oRng.Worksheet.Activate
        Else
            ' Move to next row otherwise
            Set oRng = oRng.Offset(1)
        End If
    Next
    'Application.StatusBar = False
    Application.ScreenUpdating = True
    oRng.Activate
    For Each oSubFDR In oFDR.SubFolders
        ProcessFolder oSubFDR
    Next
End Sub

Private Function GetFileHash(ByVal sFile As String, ByVal uBlockSize As Double, ByVal sHashType As
String) As String
    Dim uBytesRead As Double, uBytesToRead As Double, bDone As Boolean
    Dim aBlock() As Byte, aBytes As Variant ' Arrays to store bytes
    Dim aHash() As Byte, sHash As String, i As Long, oTmp As Variant
    Dim uFileSize As Double ' Un-Comment if GetFileHash() is to be used individually

    If oRnd Is Nothing Then Set oRnd = New Random ' Class by Microsoft: Random
    If oFSO Is Nothing Then Set oFSO = CreateObject("Scripting.FileSystemObject") ' Just to get
correct FileSize
    If oCSP Is Nothing Then Set oCSP = CreateObject("System.Security.Cryptography." & sHashType &
"CryptoServiceProvider")
```

```
    If oFSO Is Nothing Or oRnd Is Nothing Or oCSP Is Nothing Then
        MsgBox "One or more required objects cannot be created"
        Exit Function
    End If

    uFileSize = oFSO.GetFile(sFile).Size ' FILELEN() has 2GB max
    uBytesRead = 0
    bDone = False
    sHash = String(oCSP.HashSize / 4, "0") ' Each hexadecimal is 4 bits

    ' Process the file in chunks of uBlockSize or less
    If uFileSize = 0 Then
        ReDim aBlock(0)
        oCSP.TransformFinalBlock aBlock, 0, 0
        bDone = True
    Else
        With oRnd
            On Error GoTo CannotOpenFile
            .OpenFile sFile
            Do
                If uBytesRead + uBlockSize < uFileSize Then
                    uBytesToRead = uBlockSize
                Else
                    uBytesToRead = uFileSize - uBytesRead
                    bDone = True
                End If
                ' Read in some bytes
                aBytes = .ReadBytes(uBytesToRead)
                aBlock = aBytes
                If bDone Then
                    oCSP.TransformFinalBlock aBlock, 0, uBytesToRead
                    uBytesRead = uBytesRead + uBytesToRead
                Else
                    uBytesRead = uBytesRead + oCSP.TransformBlock(aBlock, 0, uBytesToRead, aBlock,
0)
                End If
                DoEvents
            Loop Until bDone
            .CloseFile
CannotOpenFile:
            If Err.Number <> 0 Then ' Change the hash code to the Error description
                oTmp = Split(Err.Description, vbCrLf)
                sHash = oTmp(1) & ":" & oTmp(2)
            End If
        End With
    End If
    If bDone Then
        ' convert Hash byte array to an hexadecimal string
        aHash = oCSP.hash
        For i = 0 To UBound(aHash)
            Mid$(sHash, i * 2 + (aHash(i) > 15) + 2) = Hex(aHash(i))
        Next
    End If
    GetFileHash = sHash
End Function
```

Chapter 40: Sorting

Unlike the .NET framework, the Visual Basic for Applications library does not include routines to sort arrays.

There are two types of workarounds: 1) implementing a sorting algorithm from scratch, or 2) using sorting routines in other commonly-available libraries.

Section 40.1: Algorithm Implementation - Quick Sort on a One-Dimensional Array

From VBA array sort function?

```
Public Sub QuickSort(vArray As Variant, inLow As Long, inHi As Long)

  Dim pivot   As Variant
  Dim tmpSwap As Variant
  Dim tmpLow  As Long
  Dim tmpHi   As Long

  tmpLow = inLow
  tmpHi = inHi

  pivot = vArray((inLow + inHi) \ 2)

  While (tmpLow <= tmpHi)

     While (vArray(tmpLow) < pivot And tmpLow < inHi)
        tmpLow = tmpLow + 1
     Wend

     While (pivot < vArray(tmpHi) And tmpHi > inLow)
        tmpHi = tmpHi - 1
     Wend

     If (tmpLow <= tmpHi) Then
        tmpSwap = vArray(tmpLow)
        vArray(tmpLow) = vArray(tmpHi)
        vArray(tmpHi) = tmpSwap
        tmpLow = tmpLow + 1
        tmpHi = tmpHi - 1
     End If

  Wend

  If (inLow < tmpHi) Then QuickSort vArray, inLow, tmpHi
  If (tmpLow < inHi) Then QuickSort vArray, tmpLow, inHi

End Sub
```

Section 40.2: Using the Excel Library to Sort a One-Dimensional Array

This code takes advantage of the `Sort` class in the Microsoft Excel Object Library.

For further reading, see:

- Copy a range to a virtual range

- How to copy selected range into given array?

```
Sub testExcelSort()

Dim arr As Variant

InitArray arr
ExcelSort arr

End Sub

Private Sub InitArray(arr As Variant)

Const size = 10
ReDim arr(size)

Dim i As Integer

' Add descending numbers to the array to start
For i = 0 To size
    arr(i) = size - i
Next i

End Sub

Private Sub ExcelSort(arr As Variant)

' Ininitialize the Excel objects (required)
Dim xl As New Excel.Application
Dim wbk As Workbook
Set wbk = xl.Workbooks.Add
Dim sht As Worksheet
Set sht = wbk.ActiveSheet

' Copy the array to the Range object
Dim rng As Range
Set rng = sht.Range("A1")
Set rng = rng.Resize(UBound(arr, 1), 1)
rng.Value = xl.WorksheetFunction.Transpose(arr)

' Run the worksheet's sort routine on the Range
Dim MySort As Sort
Set MySort = sht.Sort

With MySort
    .SortFields.Clear
    .SortFields.Add rng, xlSortOnValues, xlAscending, xlSortNormal
    .SetRange rng
    .Header = xlNo
    .Apply
End With

' Copy the results back to the array
CopyRangeToArray rng, arr

' Clear the objects
Set rng = Nothing
wbk.Close False
xl.Quit

End Sub
```

```
Private Sub CopyRangeToArray(rng As Range, arr)

Dim i As Long
Dim c As Range

' Can't just set the array to Range.value (adds a dimension)
For Each c In rng.Cells
    arr(i) = c.Value
    i = i + 1
Next c

End Sub
```

Chapter 41: Frequently used string manipulation

Quick examples for MID LEFT and RIGHT string functions using INSTR FIND and LEN.

How do you find the text between two search terms (Say: after a colon and before a comma)? How do you get the remainder of a word (using MID or using RIGHT)? Which of these functions use Zero-based params and return codes vs One-based? What happens when things go wrong? How do they handle empty strings, unfound results and negative numbers?

Section 41.1: String manipulation frequently used examples

Better MID() and other string extraction examples, currently lacking from the web. Please help me make a good example, or complete this one here. Something like this:

```
DIM strEmpty as String, strNull as String, theText as String
DIM idx as Integer
DIM letterCount as Integer
DIM result as String

strNull = NOTHING
strEmpty = ""
theText = "1234, 78910"

' -----------------
' Extract the word after the comma ", "  and before "910"  result: "78" ***
' -----------------

' Get index (place) of comma using INSTR
idx = ...    ' some explanation here
if idx < ... ' check if no comma found in text

' or get index of comma using FIND
idx = ...    ' some explanation here... Note: The difference is...
if idx < ...  ' check if no comma found in text

result = MID(theText, ..., LEN(...

' Retrieve remaining word after the comma
result = MID(theText, idx+1, LEN(theText) - idx+1)

' Get word until the comma using LEFT
result = LEFT(theText, idx - 1)

' Get remaining text after the comma-and-space using RIGHT
result = ...

' What happens when things go wrong
result = MID(strNothing, 1, 2)     ' this causes ...
result = MID(strEmpty, 1, 2) ' which causes...
result = MID(theText, 30, 2) ' and now...
result = MID(theText, 2, 999) ' no worries...
result = MID(theText, 0, 2)
result = MID(theText, 2, 0)
result = MID(theText -1, 2)
result = MID(theText 2, -1)
idx = INSTR(strNothing, "123")
idx = INSTR(theText, strNothing)
```

```
idx = INSTR(theText, strEmpty)
i = LEN(strEmpty)
i = LEN(strNothing) '...
```

Please feel free to edit this example and make it better. As long as it remains clear, and has in it common usage practices.

Chapter 42: Automation or Using other applications Libraries

If you use the objects in other applications as part of your Visual Basic application, you may want to establish a reference to the object libraries of those applications. This Documentation provides a list, sources and examples of how to use libraries of different softwares, like Windows Shell, Internet Explorer, XML HttpRequest, and others.

Section 42.1: VBScript Regular Expressions

```
Set createVBScriptRegExObject = CreateObject("vbscript.RegExp")
```

Tools> References> Microsoft VBScript Regular Expressions #.#
Associated DLL: VBScript.dll
Source: Internet Explorer 1.0 and 5.5

- MSDN-Microsoft Beefs Up VBScript with Regular Expressions
- MSDN-Regular Expression Syntax (Scripting)
- experts-exchange - Using Regular Expressions in Visual Basic for Applications and Visual Basic 6
- How to use Regular Expressions (Regex) in Microsoft Excel both in-cell and loops on SO.
- regular-expressions.info/vbscript
- regular-expressions.info/vbscriptexample
- WIKI-Regular expression

Code

You can use this functions to get RegEx results, concatenate all matches (if more than 1) into 1 string, and display result in excel cell.

```
Public Function getRegExResult(ByVal SourceString As String, Optional ByVal RegExPattern As String
= "\d+", _
    Optional ByVal isGlobalSearch As Boolean = True, Optional ByVal isCaseSensitive As Boolean =
False, Optional ByVal Delimiter As String = ";") As String

    Static RegExObject As Object
    If RegExObject Is Nothing Then
        Set RegExObject = createVBScriptRegExObject
    End If

    getRegExResult = removeLeadingDelimiter(concatObjectItems(getRegExMatches(RegExObject,
SourceString, RegExPattern, isGlobalSearch, isCaseSensitive), Delimiter), Delimiter)

End Function

Private Function getRegExMatches(ByRef RegExObj As Object, _
    ByVal SourceString As String, ByVal RegExPattern As String, ByVal isGlobalSearch As Boolean,
ByVal isCaseSensitive As Boolean) As Object

    With RegExObj
        .Global = isGlobalSearch
        .IgnoreCase = Not (isCaseSensitive) 'it is more user friendly to use positive meaning of
argument, like isCaseSensitive, than to use negative IgnoreCase
        .Pattern = RegExPattern
        Set getRegExMatches = .Execute(SourceString)
    End With

End Function
```

```
Private Function concatObjectItems(ByRef Obj As Object, Optional ByVal DelimiterCustom As String =
";") As String
    Dim ObjElement As Variant
    For Each ObjElement In Obj
        concatObjectItems = concatObjectItems & DelimiterCustom & ObjElement.Value
    Next
End Function

Public Function removeLeadingDelimiter(ByVal SourceString As String, ByVal Delimiter As String) As
String
    If Left$(SourceString, Len(Delimiter)) = Delimiter Then
        removeLeadingDelimiter = Mid$(SourceString, Len(Delimiter) + 1)
    End If
End Function

Private Function createVBScriptRegExObject() As Object
    Set createVBScriptRegExObject = CreateObject("vbscript.RegExp") 'ex.:
createVBScriptRegExObject.Pattern
End Function
```

Section 42.2: Scripting File System Object

```
Set createScriptingFileSystemObject = CreateObject("Scripting.FileSystemObject")
```

Tools> References> Microsoft Scripting Runtime
Associated DLL: ScrRun.dll
Source: Windows OS

MSDN-Accessing Files with FileSystemObject

> The File System Object (FSO) model provides an object-based tool for working with folders and files. It allows you to use the familiar object.method syntax with a rich set of properties, methods, and events to process folders and files. You can also employ the traditional Visual Basic statements and commands.

> The FSO model gives your application the ability to create, alter, move, and delete folders, or to determine if and where particular folders exist. It also enables you to get information about folders, such as their names and the date they were created or last modified.

MSDN-FileSystemObject topics: "*...explain the concept of the FileSystemObject and how to use it.*" exceltrick-FileSystemObject in VBA – Explained
Scripting.FileSystemObject

Section 42.3: Scripting Dictionary object

```
Set dict = CreateObject("Scripting.Dictionary")
```

Tools> References> Microsoft Scripting Runtime
Associated DLL: ScrRun.dll
Source: Windows OS

Scripting.Dictionary object
MSDN-Dictionary Object

Section 42.4: Internet Explorer Object

```
Set createInternetExplorerObject = CreateObject("InternetExplorer.Application")
```

Tools> References> Microsoft Internet Controls
Associated DLL: ieframe.dll
Source: Internet Explorer Browser

MSDN-InternetExplorer object

> Controls an instance of Windows Internet Explorer through automation.

Internet Explorer Objec Basic Members

The code below should introduce how the IE object works and how to manipulate it through VBA. I recommend stepping through it, otherwise it might error out during multiple navigations.

```
Sub IEGetToKnow()
    Dim IE As InternetExplorer 'Reference to Microsoft Internet Controls
    Set IE = New InternetExplorer

    With IE
        .Visible = True 'Sets or gets a value that indicates whether the object is visible or
hidden.

        'Navigation
        .Navigate2 "http://www.example.com" 'Navigates the browser to a location that might not be
expressed as a URL, such as a PIDL for an entity in the Windows Shell namespace.
        Debug.Print .Busy 'Gets a value that indicates whether the object is engaged in a navigation
or downloading operation.
        Debug.Print .ReadyState 'Gets the ready state of the object.
        .Navigate2 "http://www.example.com/2"
        .GoBack 'Navigates backward one item in the history list
        .GoForward 'Navigates forward one item in the history list.
        .GoHome 'Navigates to the current home or start page.
        .Stop 'Cancels a pending navigation or download, and stops dynamic page elements, such as
background sounds and animations.
        .Refresh 'Reloads the file that is currently displayed in the object.

        Debug.Print .Silent 'Sets or gets a value that indicates whether the object can display
dialog boxes.
        Debug.Print .Type 'Gets the user type name of the contained document object.

        Debug.Print .Top 'Sets or gets the coordinate of the top edge of the object.
        Debug.Print .Left 'Sets or gets the coordinate of the left edge of the object.
        Debug.Print .Height 'Sets or gets the height of the object.
        Debug.Print .Width 'Sets or gets the width of the object.
    End With

    IE.Quit 'close the application window
End Sub
```

Web Scraping

The most common thing to do with IE is to scrape some information of a website, or to fill a website form and submit information. We will look at how to do it.

Let us consider *example.com* source code:

```
<!doctype html>
<html>
    <head>
        <title>Example Domain</title>
        <meta charset="utf-8" />
        <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1" />
        <style ... </style>
    </head>

    <body>
        <div>
            <h1>Example Domain</h1>
            <p>This domain is established to be used for illustrative examples in documents. You
may use this
            domain in examples without prior coordination or asking for permission.</p>
            <p><a href="http://www.iana.org/domains/example">More information...</a></p>
        </div>
    </body>
</html>
```

We can use code like below to get and set information:

```
Sub IEWebScrape1()
    Dim IE As InternetExplorer 'Reference to Microsoft Internet Controls
    Set IE = New InternetExplorer

    With IE
        .Visible = True
        .Navigate2 "http://www.example.com"

        'we add a loop to be sure the website is loaded and ready.
        'Does not work consistently. Cannot be relied upon.
        Do While .Busy = True Or .ReadyState <> READYSTATE_COMPLETE 'Equivalent = .ReadyState <> 4
            ' DoEvents - worth considering. Know implications before you use it.
            Application.Wait (Now + TimeValue("00:00:01")) 'Wait 1 second, then check again.
        Loop

        'Print info in immediate window
        With .Document 'the source code HTML "below" the displayed page.
            Stop 'VBE Stop. Continue line by line to see what happens.
            Debug.Print .GetElementsByTagName("title")(0).innerHtml 'prints "Example Domain"
            Debug.Print .GetElementsByTagName("h1")(0).innerHtml 'prints "Example Domain"
            Debug.Print .GetElementsByTagName("p")(0).innerHtml 'prints "This domain is
established..."
            Debug.Print .GetElementsByTagName("p")(1).innerHtml 'prints "<a
href="http://www.iana.org/domains/example">More information...</a>"
            Debug.Print .GetElementsByTagName("p")(1).innerText 'prints "More information..."
            Debug.Print .GetElementsByTagName("a")(0).innerText 'prints "More information..."

            'We can change the localy displayed website. Don't worry about breaking the site.
            .GetElementsByTagName("title")(0).innerHtml = "Psst, scraping..."
            .GetElementsByTagName("h1")(0).innerHtml = "Let me try something fishy." 'You have just
changed the local HTML of the site.
            .GetElementsByTagName("p")(0).innerHtml = "Lorem ipsum.......... The End"
            .GetElementsByTagName("a")(0).innerText = "iana.org"
        End With '.document

        .Quit 'close the application window
```

```
    End With 'ie

End Sub
```

What is going on? The key player here is the **.Document**, that is the HTML source code. We can apply some queries to get the Collections or Object we want.
For example the `IE.Document.GetElementsByTagName("title")(0).innerHtml`. `GetElementsByTagName` returns a **Collection** of HTML Elements, that have the "*title*" tag. There is only one such tag in the source code. The **Collection** is 0-based. So to get the first element we add `(0)`. Now, in our case, we want only the `innerHtml` (a String), not the Element Object itself. So we specify the property we want.

Click

To follow a link on a site, we can use multiple methods:

```
Sub IEGoToPlaces()
    Dim IE As InternetExplorer 'Reference to Microsoft Internet Controls
    Set IE = New InternetExplorer

    With IE
        .Visible = True
        .Navigate2 "http://www.example.com"
        Stop 'VBE Stop. Continue line by line to see what happens.

        'Click
        .Document.GetElementsByTagName("a")(0).Click
        Stop 'VBE Stop.

        'Return Back
        .GoBack
        Stop 'VBE Stop.

        'Navigate using the href attribute in the <a> tag, or "link"
        .Navigate2 .Document.GetElementsByTagName("a")(0).href
        Stop 'VBE Stop.

        .Quit 'close the application window
    End With
End Sub
```

Microsoft HTML Object Library or IE Best friend

To get the most out of the HTML that gets loaded into the IE, you can (or should) use another Library, i.e. *Microsoft HTML Object Library*. More about this in another example.

IE Main issues

The main issue with IE is verifying that the page is done loading and is ready to be interacted with. The `Do While...Loop` helps, but is not reliable.

Also, using IE just to scrape HTML content is OVERKILL. Why? Because the Browser is meant for browsing, i.e. displaying the web page with all the CSS, JavaScripts, Pictures, Popups, etc. If you only need the raw data, consider different approach. E.g. using XML HTTPRequest. More about this in another example.

Chapter 43: VBA Run-Time Errors

Code that compiles can still run into errors, at run-time. This topic lists the most common ones, their causes, and how to avoid them.

Section 43.1: Run-time error '6': Overflow

Incorrect code

```
Sub DoSomething()
    Dim row As Integer
    For row = 1 To 100000
        'do stuff
    Next
End Sub
```

Why doesn't this work?

The `Integer` data type is a 16-bit signed integer with a maximum value of 32,767; assigning it to anything larger than that will *overflow* the type and raise this error.

Correct code

```
Sub DoSomething()
    Dim row As Long
    For row = 1 To 100000
        'do stuff
    Next
End Sub
```

Why does this work?

By using a `Long` (32-bit) integer instead, we can now make a loop that iterates more than 32,767 times without overflowing the counter variable's type.

Other notes

See Data Types and Limits for more information.

Section 43.2: Run-time error '9': Subscript out of range

Incorrect code

```
Sub DoSomething()
    Dim foo(1 To 10)
    Dim i As Long
    For i = 1 To 100
        foo(i) = i
    Next
End Sub
```

Why doesn't this work?

`foo` is an array that contains 10 items. When the `i` loop counter reaches a value of 11, `foo(i)` is *out of range*. This error occurs whenever an array or collection is accessed with an index that doesn't exist in that array or collection.

Correct code

```
Sub DoSomething()
    Dim foo(1 To 10)
    Dim i As Long
```

```
    For i = LBound(foo) To UBound(foo)
        foo(i) = i
    Next
End Sub
```

Why does this work?

Use `LBound` and `UBound` functions to determine the lower and upper boundaries of an array, respectively.

Other notes

When the index is a string, e.g. `ThisWorkbook.Worksheets("I don't exist")`, this error means the supplied name doesn't exist in the queried collection.

The actual error is implementation-specific though; `Collection` will raise run-time error 5 "Invalid procedure call or argument" instead:

```
Sub RaisesRunTimeError5()
    Dim foo As New Collection
    foo.Add "foo", "foo"
    Debug.Print foo("bar")
End Sub
```

Section 43.3: Run-time error '13': Type mismatch

Incorrect code

```
Public Sub DoSomething()
    DoSomethingElse "42?"
End Sub

Private Sub DoSomethingElse(foo As Date)
'    Debug.Print MonthName(Month(foo))
End Sub
```

Why doesn't this work?

VBA is trying really hard to convert the `"42?"` argument into a `Date` value. When it fails, the call to `DoSomethingElse` cannot be executed, because VBA doesn't know what date to pass, so it raises run-time error 13 *type mismatch*, because the type of the argument doesn't match the expected type (and can't be implicitly converted either).

Correct code

```
Public Sub DoSomething()
    DoSomethingElse Now
End Sub

Private Sub DoSomethingElse(foo As Date)
'    Debug.Print MonthName(Month(foo))
End Sub
```

Why does this work?

By passing a `Date` argument to a procedure that expects a `Date` parameter, the call can succeed.

Section 43.4: Run-time error '91': Object variable or With block variable not set

Incorrect code

```
Sub DoSomething()
```

```
    Dim foo As Collection
    With foo
        .Add "ABC"
        .Add "XYZ"
    End With
End Sub
```

Why doesn't this work?

Object variables hold a *reference*, and references need to be *set* using the `Set` keyword. This error occurs whenever a member call is made on an object whose reference is `Nothing`. In this case `foo` is a `Collection` reference, but it's not initialized, so the reference contains `Nothing` - and we can't call `.Add` on `Nothing`.

Correct code

```
Sub DoSomething()
    Dim foo As Collection
    Set foo = New Collection
    With foo
        .Add "ABC"
        .Add "XYZ"
    End With
End Sub
```

Why does this work?

By assigning the object variable a valid reference using the `Set` keyword, the `.Add` calls succeed.

Other notes

Often, a function or property can return an object reference - a common example is Excel's `Range.Find` method, which returns a `Range` object:

```
Dim resultRow As Long
resultRow = SomeSheet.Cells.Find("Something").Row
```

However the function can very well return `Nothing` (if the search term isn't found), so it's likely that the chained `.Row` member call fails.

Before calling object members, verify that the reference is set with a `If Not xxxx Is Nothing` condition:

```
Dim result As Range
Set result = SomeSheet.Cells.Find("Something")

Dim resultRow As Long
If Not result Is Nothing Then resultRow = result.Row
```

Section 43.5: Run-time error '20': Resume without error

Incorrect code

```
Sub DoSomething()
    On Error GoTo CleanFail
    DoSomethingElse

CleanFail:
    Debug.Print Err.Number
    Resume Next
End Sub
```

Why doesn't this work?

If the `DoSomethingElse` procedure raises an error, execution jumps to the `CleanFail` line label, prints the error number, and the `Resume Next` instruction jumps back to the instruction that immediately follows the line where the error occurred, which in this case is the `Debug.Print` instruction: the error-handling subroutine is executing without an error context, and when the `Resume Next` instruction is reached, run-time error 20 is raised because there is nowhere to resume to.

Correct Code

```
Sub DoSomething()
    On Error GoTo CleanFail
    DoSomethingElse

    Exit Sub
CleanFail:
    Debug.Print Err.Number
    Resume Next
End Sub
```

Why does this work?

By introducing an `Exit Sub` instruction before the `CleanFail` line label, we have segregated the `CleanFail` error-handling subroutine from the rest of the procedure body - the only way to execute the error-handling subroutine is via an `On Error` jump; therefore, no execution path reaches the `Resume` instruction outside of an error context, which avoids run-time error 20.

Other notes

This is very similar to Run-time error '3': Return without GoSub; in both situations, the solution is to ensure that the *normal execution path* cannot enter a sub-routine (identified by a line label) without an explicit jump (assuming `On Error GoTo` is considered an *explicit jump*).

Section 43.6: Run-time error '3': Return without GoSub

Incorrect Code

```
Sub DoSomething()
    GoSub DoThis
DoThis:
    Debug.Print "Hi!"
    Return
End Sub
```

Why doesn't this work?

Execution enters the `DoSomething` procedure, jumps to the `DoThis` label, prints "Hi!" to the debug output, *returns* to the instruction immediately after the `GoSub` call, prints "Hi!" again, and then encounters a `Return` statement, but there's nowhere to *return* to now, because we didn't get here with a `GoSub` statement.

Correct Code

```
Sub DoSomething()
    GoSub DoThis
    Exit Sub
DoThis:
    Debug.Print "Hi!"
    Return
End Sub
```

Why does this work?

By introducing an `Exit Sub` instruction *before* the `DoThis` line label, we have segregated the `DoThis` subroutine from

the rest of the procedure body - the only way to execute the `DoThis` subroutine is via the `GoSub` jump.

Other notes

`GoSub`/`Return` is deprecated, and should be avoided in favor of actual procedure calls. A procedure should not contain subroutines, other than error handlers.

This is very similar to Run-time error '20': Resume without error; in both situations, the solution is to ensure that the *normal execution path* cannot enter a sub-routine (identified by a line label) without an explicit jump (assuming `On Error GoTo` is considered an *explicit jump*).

Chapter 44: Copying, returning and passing arrays

Section 44.1: Passing Arrays to Proceedures

Arrays can be passed to proceedures by putting `()` after the name of the array variable.

```
Function countElements(ByRef arr() As Double) As Long
    countElements = UBound(arr) - LBound(arr) + 1
End Function
```

Arrays *must* be passed by reference. If no passing mechanism is specified, e.g. `myFunction(arr())`, then VBA will assume `ByRef` by default, however it is good coding practice to make it explicit. Trying to pass an array by value, e.g. `myFunction(ByVal arr())` will result in an "Array argument must be ByRef" compilation error (or a "Syntax error" compilation error if `Auto Syntax Check` is not checked in the VBE options).

Passing by reference means that any changes to the array will be preserved in the calling proceedure.

```
Sub testArrayPassing()
    Dim source(0 To 1) As Long
    source(0) = 3
    source(1) = 1

    Debug.Print doubleAndSum(source)  ' outputs 8
    Debug.Print source(0); source(1)  ' outputs 6 2
End Sub

Function doubleAndSum(ByRef arr() As Long)
    arr(0) = arr(0) * 2
    arr(1) = arr(1) * 2
    doubleAndSum = arr(0) + arr(1)
End Function
```

If you want to avoid changing the original array then be careful to write the function so that it doesn't change any elements.

```
Function doubleAndSum(ByRef arr() As Long)
    doubleAndSum = arr(0) * 2 + arr(1) * 2
End Function
```

Alternatively create a working copy of the array and work with the copy.

```
Function doubleAndSum(ByRef arr() As Long)
    Dim copyOfArr() As Long
    copyOfArr = arr

    copyOfArr(0) = copyOfArr(0) * 2
    copyOfArr(1) = copyOfArr(1) * 2

    doubleAndSum = copyOfArr(0) + copyOfArr(1)
End Function
```

Section 44.2: Copying Arrays

You can copy a VBA array into an array of the same type using the `=` operator. The arrays must be of the same type

otherwise the code will throw a "Can't assign to array" compilation error.

```
Dim source(0 to 2) As Long
Dim destinationLong() As Long
Dim destinationDouble() As Double

destinationLong = source     ' copies contents of source into destinationLong
destinationDouble = source   ' does not compile
```

The source array can be fixed or dynamic, but the destination array must be dynamic. Trying to copy to a fixed array will throw a "Can't assign to array" compilation error. Any preexisting data in the receiving array is lost and its bounds and dimenions are changed to the same as the source array.

```
Dim source() As Long
ReDim source(0 To 2)

Dim fixed(0 To 2) As Long
Dim dynamic() As Long

fixed = source   ' does not compile
dynamic = source ' does compile

Dim dynamic2() As Long
ReDim dynamic2(0 to 6, 3 to 99)

dynamic2 = source ' dynamic2 now has dimension (0 to 2)
```

Once the copy is made the two arrays are separate in memory, i.e. the two variables are not references to same underlying data, so changes made to one array do not appear in the other.

```
Dim source(0 To 2) As Long
Dim destination() As Long

source(0) = 3
source(1) = 1
source(2) = 4

destination = source
destination(0) = 2

Debug.Print source(0); source(1); source(2)                ' outputs: 3 1 4
Debug.Print destination(0); destination(1); destination(2) ' outputs: 2 1 4
```

Copying Arrays of Objects

With arrays of objects the *references* to those objects are copied, not the objects themselves. If a change is made to an object in one array it will also appear to be changed in the other array - they are both referencing the same object. However, setting an element to a different object in one array won't set it to that object the other array.

```
Dim source(0 To 2) As Range
Dim destination() As Range

Set source(0) = Range("A1"): source(0).Value = 3
Set source(1) = Range("A2"): source(1).Value = 1
Set source(2) = Range("A3"): source(2).Value = 4

destination = source
```

```
Set destination(0) = Range("A4")    'reference changed in destination but not source

destination(0).Value = 2            'affects an object only in destination
destination(1).Value = 5            'affects an object in both source and destination

Debug.Print source(0); source(1); source(2)                  ' outputs 3 5 4
Debug.Print destination(0); destination(1); destination(2)   ' outputs 2 5 4
```

Variants Containing an Array

You can also copy an array into and from a variant variable. When copying from a variant, it must contain an array of the same type as the receiving array otherwise it will throw a "Type mismatch" runtime error.

```
Dim var As Variant
Dim source(0 To 2) As Range
Dim destination() As Range

var = source
destination = var

var = 5
destination = var   ' throws runtime error
```

Section 44.3: Returning Arrays from Functions

A function in a normal module (but not a Class module) can return an array by putting `()` after the data type.

```
Function arrayOfPiDigits() As Long()
    Dim outputArray(0 To 2) As Long

    outputArray(0) = 3
    outputArray(1) = 1
    outputArray(2) = 4

    arrayOfPiDigits = outputArray
End Function
```

The result of the function can then be put into a dynamic array of the same type or a variant. The elements can also be accessed directly by using a second set of brackets, however this will call the function each time, so its best to store the results in a new array if you plan to use them more than once

```
Sub arrayExample()

    Dim destination() As Long
    Dim var As Variant

    destination = arrayOfPiDigits()
    var = arrayOfPiDigits

    Debug.Print destination(0)          ' outputs 3
    Debug.Print var(1)                  ' outputs 1
    Debug.Print arrayOfPiDigits()(2)    ' outputs 4

End Sub
```

Note that what is returned is actually a copy of the array inside the function, not a reference. So if the function returns the contents of a Static array its data can't be changed by the calling procedure.

Outputting an Array via an output argument

It is normally good coding practice for a procedure's arguments to be inputs and to output via the return value. However, the limitations of VBA sometimes make it necessary for a procedure to output data via a `ByRef` argument.

Outputting to a fixed array

```
Sub threePiDigits(ByRef destination() As Long)
    destination(0) = 3
    destination(1) = 1
    destination(2) = 4
End Sub

Sub printPiDigits()
    Dim digits(0 To 2) As Long

    threePiDigits digits
    Debug.Print digits(0); digits(1); digits(2) ' outputs 3 1 4
End Sub
```

Outputting an Array from a Class method

An output argument can also be used to output an array from a method/proceedure in a Class module

```
' Class Module 'MathConstants'
Sub threePiDigits(ByRef destination() As Long)
    ReDim destination(0 To 2)

    destination(0) = 3
    destination(1) = 1
    destination(2) = 4
End Sub

' Standard Code Module
Sub printPiDigits()
    Dim digits() As Long
    Dim mathConsts As New MathConstants

    mathConsts.threePiDigits digits
    Debug.Print digits(0); digits(1); digits(2) ' outputs 3 1 4
End Sub
```

Chapter 45: Non-Latin Characters

VBA can read and write strings in any language or script using Unicode. However, there are stricter rules in place for Identifier Tokens.

Section 45.1: Non-Latin Text in VBA Code

In spreadsheet cell A1, we have the following Arabic pangram:

صِف خَلقَ خَودِ كَمِثلِ الشَمسِ إِذ بَزَغَت — يَحظى الضَجيعُ بِها نَجلاءَ مِعطارِ

VBA provides the `AscW` and `ChrW` functions to work with multi-byte character codes. We can also use `Byte` arrays to manipulate the string variable directly:

```
Sub NonLatinStrings()

Dim rng As Range
Set rng = Range("A1")
Do Until rng = ""
    Dim MyString As String
    MyString = rng.Value

    ' AscW functions
    Dim char As String
    char = AscW(Left(MyString, 1))
    Debug.Print "First char (ChrW): " & char
    Debug.Print "First char (binary): " & BinaryFormat(char, 12)

    ' ChrW functions
    Dim uString As String
    uString = ChrW(char)
    Debug.Print "String value (text): " & uString        ' Fails! Appears as '?'
    Debug.Print "String value (AscW): " & AscW(uString)

    ' Using a Byte string
    Dim StringAsByt() As Byte
    StringAsByt = MyString
    Dim i As Long
    For i = 0 To 1 Step 2
        Debug.Print "Byte values (in decimal): " & _
            StringAsByt(i) & "|" & StringAsByt(i + 1)
        Debug.Print "Byte values (binary): " & _
            BinaryFormat(StringAsByt(i)) & "|" & BinaryFormat(StringAsByt(i + 1))
    Next i
    Debug.Print ""

    ' Printing the entire string to the immediate window fails (all '?'s)
    Debug.Print "Whole String" & vbNewLine & rng.Value
    Set rng = rng.Offset(1)
Loop

End Sub
```

This produces the following output for the Arabic Letter Sad:

> First char (ChrW): 1589
> First char (binary): 00011000110101

> String value (text): ?
> String value (AscW): 1589
> Byte values (in decimal): 53|6
> Byte values (binary): 00110101|00000110
>
> Whole String
> ??? ????? ????? ??????? ??????? ??? ??????? — ????? ???????? ???? ??????? ???????

Note that VBA is unable to print non-Latin text to the immediate window even though the string functions work correctly. This is a limitation of the IDE and not the language.

Section 45.2: Non-Latin Identifiers and Language Coverage

VBA Identifiers (variable and function names) can use the Latin script and may also be able to use Japanese, Korean, Simplified Chinese, and Traditional Chinese scripts.

The extended Latin script has full coverage for many languages:
English, French, Spanish, German, Italian, Breton, Catalan, Danish, Estonian, Finnish, Icelandic, Indonesian, Irish, Lojban, Mapudungun, Norwegian, Portuguese, Scottish Gaelic, Swedish, Tagalog

Some languages are only partially covered:
Azeri, Croatian, Czech, Esperanto, Hungarian, Latvian, Lithuanian, Polish, Romanian, Serbian, Slovak, Slovenian, Turkish, Yoruba, Welsh

Some languages have little or no coverage:
Arabic, Bulgarian, Cherokee, Dzongkha, Greek, Hindi, Macedonian, Malayalam, Mongolian, Russian, Sanskrit, Thai, Tibetan, Urdu, Uyghur

The following variable declarations are all valid:

```
Dim Yec'hed As String 'Breton
Dim «Dóna» As String 'Catalan
Dim fræk As String 'Danish
Dim tšellomängija As String 'Estonian
Dim Törkylempijävongahdus As String 'Finnish
Dim j'examine As String 'French
Dim Paß As String 'German
Dim þjófum As String 'Icelandic
Dim hÓighe As String 'Irish
Dim sofybakni As String 'Lojban (.o'i does not work)
Dim ñizol As String 'Mapudungun
Dim Vår As String 'Norwegian
Dim «brações» As String 'Portuguese
Dim d'fhàg As String 'Scottish Gaelic
```

Note that in the VBA IDE, a single apostrophe within a variable name does not turn the line into a comment (as it does on Stack Overflow).

Also, languages that use two angles to indicate a quote «» are allowed to use those in variable names desipte the fact that the ""-type quotes are not.

Chapter 46: API Calls

API stands for Application Programming Interface

API's for VBA imply a set of methods that allow direct interaction with the operating system

System calls can be made by executing procedures defined in DLL files

Section 46.1: Mac APIs

Microsoft doesn't officially support APIs but with some research more declarations can be found online

Office 2016 for Mac is sandboxed

Unlike other versions of Office apps that support VBA, Office 2016 for Mac apps are sandboxed.

Sandboxing restricts the apps from accessing resources outside the app container. This affects any add-ins or macros that involve file access or communication across processes. You can minimize the effects of sandboxing by using the new commands described in the following section. New VBA commands for Office 2016 for Mac

The following VBA commands are new and unique to Office 2016 for Mac.

Command	Use to
GrantAccessToMultipleFiles	Request a user's permission to access multiple files at once
AppleScriptTask	Call external AppleScript scripts from VB
MAC_OFFICE_VERSION	IFDEF between different Mac Office versions at compile time

Office 2011 for Mac

```
Private Declare Function system Lib "libc.dylib" (ByVal command As String) As Long
Private Declare Function popen Lib "libc.dylib" (ByVal command As String, ByVal mode As String) As
Long
Private Declare Function pclose Lib "libc.dylib" (ByVal file As Long) As Long
Private Declare Function fread Lib "libc.dylib" (ByVal outStr As String, ByVal size As Long, ByVal
items As Long, ByVal stream As Long) As Long
Private Declare Function feof Lib "libc.dylib" (ByVal file As Long) As Long
```

Office 2016 for Mac

```
Private Declare PtrSafe Function popen Lib "libc.dylib" (ByVal command As String, ByVal mode As
String) As LongPtr
Private Declare PtrSafe Function pclose Lib "libc.dylib" (ByVal file As LongPtr) As Long
Private Declare PtrSafe Function fread Lib "libc.dylib" (ByVal outStr As String, ByVal size As
LongPtr, ByVal items As LongPtr, ByVal stream As LongPtr) As Long
Private Declare PtrSafe Function feof Lib "libc.dylib" (ByVal file As LongPtr) As LongPtr
```

Section 46.2: Get total monitors and screen resolution

```
Option Explicit

'GetSystemMetrics32 info: http://msdn.microsoft.com/en-us/library/ms724385(VS.85).aspx
#If Win64 Then
    Private Declare PtrSafe Function GetSystemMetrics32 Lib "User32" Alias "GetSystemMetrics"
(ByVal nIndex As Long) As Long
#ElseIf Win32 Then
```

```
    Private Declare Function GetSystemMetrics32 Lib "User32" Alias "GetSystemMetrics" (ByVal nIndex
As Long) As Long
#End If

'VBA Wrappers:
Public Function dllGetMonitors() As Long
    Const SM_CMONITORS = 80
    dllGetMonitors = GetSystemMetrics32(SM_CMONITORS)
End Function

Public Function dllGetHorizontalResolution() As Long
    Const SM_CXVIRTUALSCREEN = 78
    dllGetHorizontalResolution = GetSystemMetrics32(SM_CXVIRTUALSCREEN)
End Function

Public Function dllGetVerticalResolution() As Long
    Const SM_CYVIRTUALSCREEN = 79
    dllGetVerticalResolution = GetSystemMetrics32(SM_CYVIRTUALSCREEN)
End Function

Public Sub ShowDisplayInfo()
    Debug.Print "Total monitors: " & vbTab & vbTab & dllGetMonitors
    Debug.Print "Horizontal Resolution: " & vbTab & dllGetHorizontalResolution
    Debug.Print "Vertical Resolution: " & vbTab & dllGetVerticalResolution

    'Total monitors:         1
    'Horizontal Resolution:  1920
    'Vertical Resolution:    1080
End Sub
```

Section 46.3: FTP and Regional APIs

modFTP

```
Option Explicit
Option Compare Text
Option Private Module

'http://msdn.microsoft.com/en-us/library/aa384180(v=VS.85).aspx
'http://www.dailydoseofexcel.com/archives/2006/01/29/ftp-via-vba/
'http://www.15seconds.com/issue/981203.htm

'Open the Internet object
Private Declare Function InternetOpen Lib "wininet.dll" Alias "InternetOpenA" ( _
    ByVal sAgent As String, _
    ByVal lAccessType As Long, _
    ByVal sProxyName As String, _
    ByVal sProxyBypass As String, _
    ByVal lFlags As Long _
) As Long
'ex: lngINet = InternetOpen("MyFTP Control", 1, vbNullString, vbNullString, 0)

'Connect to the network
Private Declare Function InternetConnect Lib "wininet.dll" Alias "InternetConnectA" ( _
    ByVal hInternetSession As Long, _
    ByVal sServerName As String, _
    ByVal nServerPort As Integer, _
    ByVal sUsername As String, _
    ByVal sPassword As String, _
    ByVal lService As Long, _
```

```
    ByVal lFlags As Long, _
    ByVal lContext As Long _
) As Long
'ex: lngINetConn = InternetConnect(lngINet, "ftp.microsoft.com", 0, "anonymous",
"wally@wallyworld.com", 1, 0, 0)

'Get a file
Private Declare Function FtpGetFile Lib "wininet.dll" Alias "FtpGetFileA" ( _
    ByVal hFtpSession As Long, _
    ByVal lpszRemoteFile As String, _
    ByVal lpszNewFile As String, _
    ByVal fFailIfExists As Boolean, _
    ByVal dwFlagsAndAttributes As Long, _
    ByVal dwFlags As Long, _
    ByVal dwContext As Long _
) As Boolean
'ex: blnRC = FtpGetFile(lngINetConn, "dirmap.txt", "c:\dirmap.txt", 0, 0, 1, 0)

'Send a file
Private Declare Function FtpPutFile Lib "wininet.dll" Alias "FtpPutFileA" _
( _
    ByVal hFtpSession As Long, _
    ByVal lpszLocalFile As String, _
    ByVal lpszRemoteFile As String, _
    ByVal dwFlags As Long, ByVal dwContext As Long _
) As Boolean
'ex: blnRC = FtpPutFile(lngINetConn, "c:\dirmap.txt", "dirmap.txt", 1, 0)

'Delete a file
Private Declare Function FtpDeleteFile Lib "wininet.dll" Alias "FtpDeleteFileA" _
( _
    ByVal hFtpSession As Long, _
    ByVal lpszFileName As String _
) As Boolean
'ex: blnRC = FtpDeleteFile(lngINetConn, "test.txt")

'Close the Internet object
Private Declare Function InternetCloseHandle Lib "wininet.dll" (ByVal hInet As Long) As Integer
'ex: InternetCloseHandle lngINetConn
'ex: InternetCloseHandle lngINet



Private Declare Function FtpFindFirstFile Lib "wininet.dll" Alias "FtpFindFirstFileA" _
( _
    ByVal hFtpSession As Long, _
    ByVal lpszSearchFile As String, _
    lpFindFileData As WIN32_FIND_DATA, _
    ByVal dwFlags As Long, _
    ByVal dwContent As Long _
) As Long
Private Type FILETIME
        dwLowDateTime As Long
        dwHighDateTime As Long
End Type
Private Type WIN32_FIND_DATA
        dwFileAttributes As Long
        ftCreationTime As FILETIME
        ftLastAccessTime As FILETIME
        ftLastWriteTime As FILETIME
        nFileSizeHigh As Long
        nFileSizeLow As Long
```

```
        dwReserved0 As Long
        dwReserved1 As Long
        cFileName As String * MAX_FTP_PATH
        cAlternate As String * 14
End Type
'ex: lngHINet = FtpFindFirstFile(lngINetConn, "*.*", pData, 0, 0)

Private Declare Function InternetFindNextFile Lib "wininet.dll" Alias "InternetFindNextFileA" _
( _
    ByVal hFind As Long, _
    lpvFindData As WIN32_FIND_DATA _
) As Long
'ex: blnRC = InternetFindNextFile(lngHINet, pData)


Public Sub showLatestFTPVersion()
    Dim ftpSuccess As Boolean, msg As String, lngFindFirst As Long
    Dim lngINet As Long, lngINetConn As Long
    Dim pData As WIN32_FIND_DATA
    'init the filename buffer
    pData.cFileName = String(260, 0)

    msg = "FTP Error"
    lngINet = InternetOpen("MyFTP Control", 1, vbNullString, vbNullString, 0)
    If lngINet > 0 Then
        lngINetConn = InternetConnect(lngINet, FTP_SERVER_NAME, FTP_SERVER_PORT, FTP_USER_NAME,
FTP_PASSWORD, 1, 0, 0)
        If lngINetConn > 0 Then
                FtpPutFile lngINetConn, "C:\Tmp\ftp.cls", "ftp.cls", FTP_TRANSFER_BINARY, 0
                'lngFindFirst = FtpFindFirstFile(lngINetConn, "ExcelDiff.xlsm", pData, 0, 0)
            If lngINet = 0 Then
                msg = "DLL error: " & Err.LastDllError & ", Error Number: " & Err.Number & ", Error
Desc: " & Err.Description
            Else
                msg = left(pData.cFileName, InStr(1, pData.cFileName, String(1, 0),
vbBinaryCompare) - 1)
            End If
            InternetCloseHandle lngINetConn
        End If
    InternetCloseHandle lngINet
    End If
        MsgBox msg
End Sub
```

modRegional:

```
Option Explicit

Private Const LOCALE_SDECIMAL = &HE
Private Const LOCALE_SLIST = &HC

Private Declare Function GetLocaleInfo Lib "Kernel32" Alias "GetLocaleInfoA" (ByVal Locale As Long,
ByVal LCType As Long, ByVal lpLCData As String, ByVal cchData As Long) As Long
Private Declare Function SetLocaleInfo Lib "Kernel32" Alias "SetLocaleInfoA" (ByVal Locale As Long,
ByVal LCType As Long, ByVal lpLCData As String) As Boolean
Private Declare Function GetUserDefaultLCID% Lib "Kernel32" ()

Public Function getTimeSeparator() As String
    getTimeSeparator = Application.International(xlTimeSeparator)
End Function
Public Function getDateSeparator() As String
```

```
    getDateSeparator = Application.International(xlDateSeparator)
End Function
Public Function getListSeparator() As String
    Dim ListSeparator As String, iRetVal1 As Long, iRetVal2 As Long, lpLCDataVar As String,
Position As Integer, Locale As Long
    Locale = GetUserDefaultLCID()
    iRetVal1 = GetLocaleInfo(Locale, LOCALE_SLIST, lpLCDataVar, 0)
    ListSeparator = String$(iRetVal1, 0)
    iRetVal2 = GetLocaleInfo(Locale, LOCALE_SLIST, ListSeparator, iRetVal1)
    Position = InStr(ListSeparator, Chr$(0))
    If Position > 0 Then ListSeparator = Left$(ListSeparator, Position - 1) Else ListSeparator =
vbNullString
    getListSeparator = ListSeparator
End Function

Private Sub ChangeSettingExample()  'change the setting of the character displayed as the decimal
separator.
    Call SetLocalSetting(LOCALE_SDECIMAL, ",")  'to change to ","
    Stop                                        'check your control panel to verify or use the
GetLocaleInfo API function
    Call SetLocalSetting(LOCALE_SDECIMAL, ".")  'to back change to "."
End Sub

Private Function SetLocalSetting(LC_CONST As Long, Setting As String) As Boolean
    Call SetLocaleInfo(GetUserDefaultLCID(), LC_CONST, Setting)
End Function
```

Section 46.4: API declaration and usage

Declaring a DLL procedure to work with different VBA versions:

```
Option Explicit

#If Win64 Then

    Private Declare PtrSafe Sub xLib "Kernel32" Alias "Sleep" (ByVal dwMilliseconds As Long)

#ElseIf Win32 Then

    Private Declare Sub apiSleep Lib "Kernel32" Alias "Sleep" (ByVal dwMilliseconds As Long)

#End If
```

The above declaration tells VBA how to call the function "Sleep" defined in file Kernel32.dll

Win64 and Win32 are predefined constants used for conditional compilation

Pre-defined Constants

Some compilation constants are already pre-defined. Which ones exist will depend on the bitness of the office version you're running VBA in. Note that Vba7 was introduced alongside Office 2010 to support 64 bit versions of Office.

Constant	16 bit	32 bit	64 bit
Vba6	False	If Vba6	False
Vba7	False	If Vba7	True
Win16	True	False	False
Win32	False	True	True
Win64	False	False	True

Mac	False	If Mac	If Mac

These constants refer to the Office version, not the Windows version. For example Win32 = TRUE in 32-bit Office, even if the OS is a 64-bit version of Windows.

The main difference when declaring APIs is between 32 bit and 64 bit Office versions which introduced new parameter types (see Remarks section for more details)

> Notes:
>
> - Declarations are placed at the top of the module, and outside any Subs or Functions
> - Procedures declared in standard modules are public by default
> - To declare a procedure private to a module precede the declaration with the `Private` keyword
> - DLL procedures declared in any other type of module are private to that module

Simple example for the Sleep API call:

```
Public Sub TestPause()

    Dim start As Double

    start = Timer

    Sleep 9000      'Pause execution for 9 seconds

    Debug.Print "Paused for " & Format(Timer - start, "#,###.000") & " seconds"

    'Immediate window result: Paused for 9.000 seconds

End Sub
```

It is recommended to create a dedicated API module to provide easy access to the system functions from VBA wrappers -- normal VBA Subs or Functions that encapsulate the details needed for the actual system call such as parameters used in libraries, and initialization of those parameters

The module can contain all declarations and dependencies:

- Method signatures and required data structures
- Wrappers that perform input validation, and ensure all parameters are passed as expected

To declare a DLL procedure, add a `Declare` statement to the Declarations section of the code window.

If the procedure returns a value, declare it as a **Function**:

```
Declare Function publicname Lib "libname" [Alias "alias"] [([[ByVal] variable [As type] [,[ByVal]
variable [As type]]...])] As Type
```

If a procedure does not return a value, declare it as a **Sub**:

```
Declare Sub publicname Lib "libname" [Alias "alias"] [([[ByVal] variable [As type] [,[ByVal]
variable [As type]]...])]
```

- !!!

> Also of note is that **most invalid calls to the API's will crash Excel**, and possibly corrupt data files

- !!!

Office 2011 for Mac

```
Private Declare Function system Lib "libc.dylib" (ByVal command As String) As Long

Sub RunSafari()
    Dim result As Long
    result = system("open -a Safari --args http://www.google.com")
    Debug.Print Str(result)
End Sub
```

The examples bellow (Windows API - Dedicated Module (1 and 2)) show an API module that includes common declarations for Win64 and Win32

Section 46.5: Windows API - Dedicated Module (1 of 2)

```
Option Explicit

#If Win64 Then  'Win64 = True, Win32 = False, Win16 = False
    Private Declare PtrSafe Sub apiCopyMemory Lib "Kernel32" Alias "RtlMoveMemory" (MyDest As Any,
MySource As Any, ByVal MySize As Long)
    Private Declare PtrSafe Sub apiExitProcess Lib "Kernel32" Alias "ExitProcess" (ByVal uExitCode
As Long)
    Private Declare PtrSafe Sub apiSetCursorPos Lib "User32" Alias "SetCursorPos" (ByVal X As
Integer, ByVal Y As Integer)
    Private Declare PtrSafe Sub apiSleep Lib "Kernel32" Alias "Sleep" (ByVal dwMilliseconds As
Long)
    Private Declare PtrSafe Function apiAttachThreadInput Lib "User32" Alias "AttachThreadInput"
(ByVal idAttach As Long, ByVal idAttachTo As Long, ByVal fAttach As Long) As Long
    Private Declare PtrSafe Function apiBringWindowToTop Lib "User32" Alias "BringWindowToTop"
(ByVal lngHWnd As Long) As Long
    Private Declare PtrSafe Function apiCloseWindow Lib "User32" Alias "CloseWindow" (ByVal hWnd As
Long) As Long
    Private Declare PtrSafe Function apiDestroyWindow Lib "User32" Alias "DestroyWindow" (ByVal
hWnd As Long) As Boolean
    Private Declare PtrSafe Function apiEndDialog Lib "User32" Alias "EndDialog" (ByVal hWnd As
Long, ByVal result As Long) As Boolean
    Private Declare PtrSafe Function apiEnumChildWindows Lib "User32" Alias "EnumChildWindows"
(ByVal hWndParent As Long, ByVal pEnumProc As Long, ByVal lParam As Long) As Long
    Private Declare PtrSafe Function apiExitWindowsEx Lib "User32" Alias "ExitWindowsEx" (ByVal
uFlags As Long, ByVal dwReserved As Long) As Long
    Private Declare PtrSafe Function apiFindExecutable Lib "Shell32" Alias "FindExecutableA" (ByVal
lpFile As String, ByVallpDirectory As String, ByVal lpResult As String) As Long
    Private Declare PtrSafe Function apiFindWindow Lib "User32" Alias "FindWindowA" (ByVal
lpClassName As String, ByVal lpWindowName As String) As Long
    Private Declare PtrSafe Function apiFindWindowEx Lib "User32" Alias "FindWindowExA" (ByVal
hWnd1 As Long, ByVal hWnd2 As Long, ByVal lpsz1 As String, ByVal lpsz2 As String) As Long
    Private Declare PtrSafe Function apiGetActiveWindow Lib "User32" Alias "GetActiveWindow" () As
Long
    Private Declare PtrSafe Function apiGetClassNameA Lib "User32" Alias "GetClassNameA" (ByVal
hWnd As Long, ByVal szClassName As String, ByVal lLength As Long) As Long
    Private Declare PtrSafe Function apiGetCommandLine Lib "Kernel32" Alias "GetCommandLineW" () As
Long
```

```
    Private Declare PtrSafe Function apiGetCommandLineParams Lib "Kernel32" Alias "GetCommandLineA"
() As Long
    Private Declare PtrSafe Function apiGetDiskFreeSpaceEx Lib "Kernel32" Alias
"GetDiskFreeSpaceExA" (ByVal lpDirectoryName As String, lpFreeBytesAvailableToCaller As Currency,
lpTotalNumberOfBytes As Currency, lpTotalNumberOfFreeBytes As Currency) As Long
    Private Declare PtrSafe Function apiGetDriveType Lib "Kernel32" Alias "GetDriveTypeA" (ByVal
nDrive As String) As Long
    Private Declare PtrSafe Function apiGetExitCodeProcess Lib "Kernel32" Alias
"GetExitCodeProcess" (ByVal hProcess As Long, lpExitCode As Long) As Long
    Private Declare PtrSafe Function apiGetForegroundWindow Lib "User32" Alias
"GetForegroundWindow" () As Long
    Private Declare PtrSafe Function apiGetFrequency Lib "Kernel32" Alias
"QueryPerformanceFrequency" (cyFrequency As Currency) As Long
    Private Declare PtrSafe Function apiGetLastError Lib "Kernel32" Alias "GetLastError" () As
Integer
    Private Declare PtrSafe Function apiGetParent Lib "User32" Alias "GetParent" (ByVal hWnd As
Long) As Long
    Private Declare PtrSafe Function apiGetSystemMetrics Lib "User32" Alias "GetSystemMetrics"
(ByVal nIndex As Long) As Long
    Private Declare PtrSafe Function apiGetSystemMetrics32 Lib "User32" Alias "GetSystemMetrics"
(ByVal nIndex As Long) As Long
    Private Declare PtrSafe Function apiGetTickCount Lib "Kernel32" Alias "QueryPerformanceCounter"
(cyTickCount As Currency) As Long
    Private Declare PtrSafe Function apiGetTickCountMs Lib "Kernel32" Alias "GetTickCount" () As
Long
    Private Declare PtrSafe Function apiGetUserName Lib "AdvApi32" Alias "GetUserNameA" (ByVal
lpBuffer As String, nSize As Long) As Long
    Private Declare PtrSafe Function apiGetWindow Lib "User32" Alias "GetWindow" (ByVal hWnd As
Long, ByVal wCmd As Long) As Long
    Private Declare PtrSafe Function apiGetWindowRect Lib "User32" Alias "GetWindowRect" (ByVal
hWnd As Long, lpRect As winRect) As Long
    Private Declare PtrSafe Function apiGetWindowText Lib "User32" Alias "GetWindowTextA" (ByVal
hWnd As Long, ByVal szWindowText As String, ByVal lLength As Long) As Long
    Private Declare PtrSafe Function apiGetWindowThreadProcessId Lib "User32" Alias
"GetWindowThreadProcessId" (ByVal hWnd As Long, lpdwProcessId As Long) As Long
    Private Declare PtrSafe Function apiIsCharAlphaNumericA Lib "User32" Alias
"IsCharAlphaNumericA" (ByVal byChar As Byte) As Long
    Private Declare PtrSafe Function apiIsIconic Lib "User32" Alias "IsIconic" (ByVal hWnd As Long)
As Long
    Private Declare PtrSafe Function apiIsWindowVisible Lib "User32" Alias "IsWindowVisible" (ByVal
hWnd As Long) As Long
    Private Declare PtrSafe Function apiIsZoomed Lib "User32" Alias "IsZoomed" (ByVal hWnd As Long)
As Long
    Private Declare PtrSafe Function apiLStrCpynA Lib "Kernel32" Alias "lstrcpynA" (ByVal
pDestination As String, ByVal pSource As Long, ByVal iMaxLength As Integer) As Long
    Private Declare PtrSafe Function apiMessageBox Lib "User32" Alias "MessageBoxA" (ByVal hWnd As
Long, ByVal lpText As String, ByVal lpCaption As String, ByVal wType As Long) As Long
    Private Declare PtrSafe Function apiOpenIcon Lib "User32" Alias "OpenIcon" (ByVal hWnd As Long)
As Long
    Private Declare PtrSafe Function apiOpenProcess Lib "Kernel32" Alias "OpenProcess" (ByVal
dwDesiredAccess As Long, ByVal bInheritHandle As Long, ByVal dwProcessId As Long) As Long
    Private Declare PtrSafe Function apiPathAddBackslashByPointer Lib "ShlwApi" Alias
"PathAddBackslashW" (ByVal lpszPath As Long) As Long
    Private Declare PtrSafe Function apiPathAddBackslashByString Lib "ShlwApi" Alias
"PathAddBackslashW" (ByVal lpszPath As String) As Long
'http://msdn.microsoft.com/en-us/library/aa155716%28office.10%29.aspx
    Private Declare PtrSafe Function apiPostMessage Lib "User32" Alias "PostMessageA" (ByVal hWnd
As Long, ByVal wMsg As Long, ByVal wParam As Long, ByVal lParam As Long) As Long
    Private Declare PtrSafe Function apiRegQueryValue Lib "AdvApi32" Alias "RegQueryValue" (ByVal
hKey As Long, ByVal sValueName As String, ByVal dwReserved As Long, ByRef lValueType As Long, ByVal
sValue As String, ByRef lResultLen As Long) As Long
    Private Declare PtrSafe Function apiSendMessage Lib "User32" Alias "SendMessageA" (ByVal hWnd
```

```
As Long, ByVal wMsg As Long, ByVal wParam As Long, lParam As Any) As Long
    Private Declare PtrSafe Function apiSetActiveWindow Lib "User32" Alias "SetActiveWindow" (ByVal
hWnd As Long) As Long
    Private Declare PtrSafe Function apiSetCurrentDirectoryA Lib "Kernel32" Alias
"SetCurrentDirectoryA" (ByVal lpPathName As String) As Long
    Private Declare PtrSafe Function apiSetFocus Lib "User32" Alias "SetFocus" (ByVal hWnd As Long)
As Long
    Private Declare PtrSafe Function apiSetForegroundWindow Lib "User32" Alias
"SetForegroundWindow" (ByVal hWnd As Long) As Long
    Private Declare PtrSafe Function apiSetLocalTime Lib "Kernel32" Alias "SetLocalTime" (lpSystem
As SystemTime) As Long
    Private Declare PtrSafe Function apiSetWindowPlacement Lib "User32" Alias "SetWindowPlacement"
(ByVal hWnd As Long, ByRef lpwndpl As winPlacement) As Long
    Private Declare PtrSafe Function apiSetWindowPos Lib "User32" Alias "SetWindowPos" (ByVal hWnd
As Long, ByVal hWndInsertAfter As Long, ByVal X As Long, ByVal Y As Long, ByVal cx As Long, ByVal
cy As Long, ByVal wFlags As Long) As Long
    Private Declare PtrSafe Function apiSetWindowText Lib "User32" Alias "SetWindowTextA" (ByVal
hWnd As Long, ByVal lpString As String) As Long
    Private Declare PtrSafe Function apiShellExecute Lib "Shell32" Alias "ShellExecuteA" (ByVal
hWnd As Long, ByVal lpOperation As String, ByVal lpFile As String, ByVal lpParameters As String,
ByVal lpDirectory As String, ByVal nShowCmd As Long) As Long
    Private Declare PtrSafe Function apiShowWindow Lib "User32" Alias "ShowWindow" (ByVal hWnd As
Long, ByVal nCmdShow As Long) As Long
    Private Declare PtrSafe Function apiShowWindowAsync Lib "User32" Alias "ShowWindowAsync" (ByVal
hWnd As Long, ByVal nCmdShow As Long) As Long
    Private Declare PtrSafe Function apiStrCpy Lib "Kernel32" Alias "lstrcpynA" (ByVal pDestination
As String, ByVal pSource As String, ByVal iMaxLength As Integer) As Long
    Private Declare PtrSafe Function apiStringLen Lib "Kernel32" Alias "lstrlenW" (ByVal lpString
As Long) As Long
    Private Declare PtrSafe Function apiStrTrimW Lib "ShlwApi" Alias "StrTrimW" () As Boolean
    Private Declare PtrSafe Function apiTerminateProcess Lib "Kernel32" Alias "TerminateProcess"
(ByVal hWnd As Long, ByVal uExitCode As Long) As Long
    Private Declare PtrSafe Function apiTimeGetTime Lib "Winmm" Alias "timeGetTime" () As Long
    Private Declare PtrSafe Function apiVarPtrArray Lib "MsVbVm50" Alias "VarPtr" (Var() As Any) As
Long
    Private Type browseInfo     'used by apiBrowseForFolder
        hOwner As Long
        pidlRoot As Long
        pszDisplayName As String
        lpszTitle As String
        ulFlags As Long
        lpfn As Long
        lParam As Long
        iImage As Long
    End Type
    Private Declare PtrSafe Function apiBrowseForFolder Lib "Shell32" Alias "SHBrowseForFolderA"
(lpBrowseInfo As browseInfo) As Long
    Private Type CHOOSECOLOR    'used by apiChooseColor; http://support.microsoft.com/kb/153929 and
http://www.cpearson.com/Excel/Colors.aspx
        lStructSize As Long
        hWndOwner As Long
        hInstance As Long
        rgbResult As Long
        lpCustColors As String
        flags As Long
        lCustData As Long
        lpfnHook As Long
        lpTemplateName As String
    End Type
    Private Declare PtrSafe Function apiChooseColor Lib "ComDlg32" Alias "ChooseColorA"
(pChoosecolor As CHOOSECOLOR) As Long
    Private Type FindWindowParameters   'Custom structure for passing in the parameters in/out of
```

```
the hook enumeration function; could use global variables instead, but this is nicer
        strTitle As String  'INPUT
        hWnd As Long        'OUTPUT
    End Type                            'Find a specific window with dynamic caption from a list of
all open windows:
http://www.everythingaccess.com/tutorials.asp?ID=Bring-an-external-application-window-to-the-foregrou
nd
    Private Declare PtrSafe Function apiEnumWindows Lib "User32" Alias "EnumWindows" (ByVal
lpEnumFunc As LongPtr, ByVal lParam As LongPtr) As Long
    Private Type lastInputInfo  'used by apiGetLastInputInfo, getLastInputTime
        cbSize As Long
        dwTime As Long
    End Type
    Private Declare PtrSafe Function apiGetLastInputInfo Lib "User32" Alias "GetLastInputInfo"
(ByRef plii As lastInputInfo) As Long
    'http://www.pgacon.com/visualbasic.htm#Take%20Advantage%20of%20Conditional%20Compilation
    'Logical and Bitwise Operators in Visual Basic:
http://msdn.microsoft.com/en-us/library/wz3k228a(v=vs.80).aspx and
http://stackoverflow.com/questions/1070863/hidden-features-of-vba
    Private Type SystemTime
          wYear          As Integer
          wMonth         As Integer
          wDayOfWeek     As Integer
          wDay           As Integer
          wHour          As Integer
          wMinute        As Integer
          wSecond        As Integer
          wMilliseconds  As Integer
    End Type
    Private Declare PtrSafe Sub apiGetLocalTime Lib "Kernel32" Alias "GetLocalTime" (lpSystem As
SystemTime)
    Private Type pointAPI       'used by apiSetWindowPlacement
         X As Long
         Y As Long
    End Type
    Private Type rectAPI       'used by apiSetWindowPlacement
        Left_Renamed As Long
        Top_Renamed As Long
        Right_Renamed As Long
        Bottom_Renamed As Long
    End Type
    Private Type winPlacement   'used by apiSetWindowPlacement
        length As Long
        flags As Long
        showCmd As Long
        ptMinPosition As pointAPI
        ptMaxPosition As pointAPI
        rcNormalPosition As rectAPI
    End Type
    Private Declare PtrSafe Function apiGetWindowPlacement Lib "User32" Alias "GetWindowPlacement"
(ByVal hWnd As Long, ByRef lpwndpl As winPlacement) As Long
    Private Type winRect     'used by apiMoveWindow
        Left As Long
        Top As Long
        Right As Long
        Bottom As Long
    End Type
    Private Declare PtrSafe Function apiMoveWindow Lib "User32" Alias "MoveWindow" (ByVal hWnd As
Long, xLeft As Long, ByVal yTop As Long, wWidth As Long, ByVal hHeight As Long, ByVal repaint As
Long) As Long

    Private Declare PtrSafe Function apiInternetOpen Lib "WiniNet" Alias "InternetOpenA" (ByVal
```

```
sAgent As String, ByVal lAccessType As Long, ByVal sProxyName As String, ByVal sProxyBypass As
String, ByVal lFlags As Long) As Long    'Open the Internet object    'ex: lngINet =
InternetOpen("MyFTP Control", 1, vbNullString, vbNullString, 0)
    Private Declare PtrSafe Function apiInternetConnect Lib "WiniNet" Alias "InternetConnectA"
(ByVal hInternetSession As Long, ByVal sServerName As String, ByVal nServerPort As Integer, ByVal
sUsername As String, ByVal sPassword As String, ByVal lService As Long, ByVal lFlags As Long, ByVal
lContext As Long) As Long  'Connect to the network  'ex: lngINetConn = InternetConnect(lngINet,
"ftp.microsoft.com", 0, "anonymous", "wally@wallyworld.com", 1, 0, 0)
    Private Declare PtrSafe Function apiFtpGetFile Lib "WiniNet" Alias "FtpGetFileA" (ByVal
hFtpSession As Long, ByVal lpszRemoteFile As String, ByVal lpszNewFile As String, ByVal
fFailIfExists As Boolean, ByVal dwFlagsAndAttributes As Long, ByVal dwFlags As Long, ByVal
dwContext As Long) As Boolean    'Get a file 'ex: blnRC = FtpGetFile(lngINetConn, "dirmap.txt",
"c:\dirmap.txt", 0, 0, 1, 0)
    Private Declare PtrSafe Function apiFtpPutFile Lib "WiniNet" Alias "FtpPutFileA" (ByVal
hFtpSession As Long, ByVal lpszLocalFile As String, ByVal lpszRemoteFile As String, ByVal dwFlags
As Long, ByVal dwContext As Long) As Boolean  'Send a file  'ex: blnRC = FtpPutFile(lngINetConn,
"c:\dirmap.txt", "dirmap.txt", 1, 0)
    Private Declare PtrSafe Function apiFtpDeleteFile Lib "WiniNet" Alias "FtpDeleteFileA" (ByVal
hFtpSession As Long, ByVal lpszFileName As String) As Boolean 'Delete a file 'ex: blnRC =
FtpDeleteFile(lngINetConn, "test.txt")
    Private Declare PtrSafe Function apiInternetCloseHandle Lib "WiniNet" (ByVal hInet As Long) As
Integer  'Close the Internet object  'ex: InternetCloseHandle lngINetConn    'ex: InternetCloseHandle
lngINet
    Private Declare PtrSafe Function apiFtpFindFirstFile Lib "WiniNet" Alias "FtpFindFirstFileA"
(ByVal hFtpSession As Long, ByVal lpszSearchFile As String, lpFindFileData As WIN32_FIND_DATA,
ByVal dwFlags As Long, ByVal dwContent As Long) As Long
    Private Type FILETIME
        dwLowDateTime As Long
        dwHighDateTime As Long
    End Type
    Private Type WIN32_FIND_DATA
        dwFileAttributes As Long
        ftCreationTime As FILETIME
        ftLastAccessTime As FILETIME
        ftLastWriteTime As FILETIME
        nFileSizeHigh As Long
        nFileSizeLow As Long
        dwReserved0 As Long
        dwReserved1 As Long
        cFileName As String * 1 'MAX_FTP_PATH
        cAlternate As String * 14
    End Type    'ex: lngHINet = FtpFindFirstFile(lngINetConn, "*.*", pData, 0, 0)
    Private Declare PtrSafe Function apiInternetFindNextFile Lib "WiniNet" Alias
"InternetFindNextFileA" (ByVal hFind As Long, lpvFindData As WIN32_FIND_DATA) As Long  'ex: blnRC =
InternetFindNextFile(lngHINet, pData)
#ElseIf Win32 Then  'Win32 = True, Win16 = False
```

(continued in second example)

Section 46.6: Windows API - Dedicated Module (2 of 2)

```
#ElseIf Win32 Then  'Win32 = True, Win16 = False
    Private Declare Sub apiCopyMemory Lib "Kernel32" Alias "RtlMoveMemory" (MyDest As Any, MySource
As Any, ByVal MySize As Long)
    Private Declare Sub apiExitProcess Lib "Kernel32" Alias "ExitProcess" (ByVal uExitCode As Long)
    'Private Declare Sub apiGetStartupInfo Lib "Kernel32" Alias "GetStartupInfoA" (lpStartupInfo As
STARTUPINFO)
    Private Declare Sub apiSetCursorPos Lib "User32" Alias "SetCursorPos" (ByVal X As Integer,
ByVal Y As Integer)  'Logical and Bitwise Operators in Visual Basic:
http://msdn.microsoft.com/en-us/library/wz3k228a(v=vs.80).aspx and
http://stackoverflow.com/questions/1070863/hidden-features-of-vba
```

```
'http://www.pgacon.com/visualbasic.htm#Take%20Advantage%20of%20Conditional%20Compilation
    Private Declare Sub apiSleep Lib "Kernel32" Alias "Sleep" (ByVal dwMilliseconds As Long)
    Private Declare Function apiAttachThreadInput Lib "User32" Alias "AttachThreadInput" (ByVal
idAttach As Long, ByVal idAttachTo As Long, ByVal fAttach As Long) As Long
    Private Declare Function apiBringWindowToTop Lib "User32" Alias "BringWindowToTop" (ByVal
lngHWnd As Long) As Long
    Private Declare Function apiCloseHandle Lib "Kernel32" (ByVal hObject As Long) As Long
    Private Declare Function apiCloseWindow Lib "User32" Alias "CloseWindow" (ByVal hWnd As Long)
As Long
    'Private Declare Function apiCreatePipe Lib "Kernel32" (phReadPipe As Long, phWritePipe As Long,
lpPipeAttributes As SECURITY_ATTRIBUTES, ByVal nSize As Long) As Long
    'Private Declare Function apiCreateProcess Lib "Kernel32" Alias "CreateProcessA" (ByVal
lpApplicationName As Long, ByVal lpCommandLine As String, lpProcessAttributes As Any,
lpThreadAttributes As Any, ByVal bInheritHandles As Long, ByVal dwCreationFlags As Long,
lpEnvironment As Any, ByVal lpCurrentDriectory As String, lpStartupInfo As STARTUPINFO,
lpProcessInformation As PROCESS_INFORMATION) As Long
    Private Declare Function apiDestroyWindow Lib "User32" Alias "DestroyWindow" (ByVal hWnd As
Long) As Boolean
    Private Declare Function apiEndDialog Lib "User32" Alias "EndDialog" (ByVal hWnd As Long, ByVal
result As Long) As Boolean
    Private Declare Function apiEnumChildWindows Lib "User32" Alias "EnumChildWindows" (ByVal
hWndParent As Long, ByVal pEnumProc As Long, ByVal lParam As Long) As Long
    Private Declare Function apiExitWindowsEx Lib "User32" Alias "ExitWindowsEx" (ByVal uFlags As
Long, ByVal dwReserved As Long) As Long
    Private Declare Function apiFindExecutable Lib "Shell32" Alias "FindExecutableA" (ByVal lpFile
As String, ByVallpDirectory As String, ByVal lpResult As String) As Long
    Private Declare Function apiFindWindow Lib "User32" Alias "FindWindowA" (ByVal lpClassName As
String, ByVal lpWindowName As String) As Long
    Private Declare Function apiFindWindowEx Lib "User32" Alias "FindWindowExA" (ByVal hWnd1 As
Long, ByVal hWnd2 As Long, ByVal lpsz1 As String, ByVal lpsz2 As String) As Long
    Private Declare Function apiGetActiveWindow Lib "User32" Alias "GetActiveWindow" () As Long
    Private Declare Function apiGetClassNameA Lib "User32" Alias "GetClassNameA" (ByVal hWnd As
Long, ByVal szClassName As String, ByVal lLength As Long) As Long
    Private Declare Function apiGetCommandLine Lib "Kernel32" Alias "GetCommandLineW" () As Long
    Private Declare Function apiGetCommandLineParams Lib "Kernel32" Alias "GetCommandLineA" () As
Long
    Private Declare Function apiGetDiskFreeSpaceEx Lib "Kernel32" Alias "GetDiskFreeSpaceExA"
(ByVal lpDirectoryName As String, lpFreeBytesAvailableToCaller As Currency, lpTotalNumberOfBytes As
Currency, lpTotalNumberOfFreeBytes As Currency) As Long
    Private Declare Function apiGetDriveType Lib "Kernel32" Alias "GetDriveTypeA" (ByVal nDrive As
String) As Long
    Private Declare Function apiGetExitCodeProcess Lib "Kernel32" (ByVal hProcess As Long,
lpExitCode As Long) As Long
    Private Declare Function apiGetFileSize Lib "Kernel32" (ByVal hFile As Long, lpFileSizeHigh As
Long) As Long
    Private Declare Function apiGetForegroundWindow Lib "User32" Alias "GetForegroundWindow" () As
Long
    Private Declare Function apiGetFrequency Lib "Kernel32" Alias "QueryPerformanceFrequency"
(cyFrequency As Currency) As Long
    Private Declare Function apiGetLastError Lib "Kernel32" Alias "GetLastError" () As Integer
    Private Declare Function apiGetParent Lib "User32" Alias "GetParent" (ByVal hWnd As Long) As
Long
    Private Declare Function apiGetSystemMetrics Lib "User32" Alias "GetSystemMetrics" (ByVal
nIndex As Long) As Long
    Private Declare Function apiGetTickCount Lib "Kernel32" Alias "QueryPerformanceCounter"
(cyTickCount As Currency) As Long
    Private Declare Function apiGetTickCountMs Lib "Kernel32" Alias "GetTickCount" () As Long
    Private Declare Function apiGetUserName Lib "AdvApi32" Alias "GetUserNameA" (ByVal lpBuffer As
String, nSize As Long) As Long
    Private Declare Function apiGetWindow Lib "User32" Alias "GetWindow" (ByVal hWnd As Long, ByVal
wCmd As Long) As Long
    Private Declare Function apiGetWindowRect Lib "User32" Alias "GetWindowRect" (ByVal hWnd As
```

```
Long, lpRect As winRect) As Long
    Private Declare Function apiGetWindowText Lib "User32" Alias "GetWindowTextA" (ByVal hWnd As
Long, ByVal szWindowText As String, ByVal lLength As Long) As Long
    Private Declare Function apiGetWindowThreadProcessId Lib "User32" Alias
"GetWindowThreadProcessId" (ByVal hWnd As Long, lpdwProcessId As Long) As Long
    Private Declare Function apiIsCharAlphaNumericA Lib "User32" Alias "IsCharAlphaNumericA" (ByVal
byChar As Byte) As Long
    Private Declare Function apiIsIconic Lib "User32" Alias "IsIconic" (ByVal hWnd As Long) As Long
    Private Declare Function apiIsWindowVisible Lib "User32" Alias "IsWindowVisible" (ByVal hWnd As
Long) As Long
    Private Declare Function apiIsZoomed Lib "User32" Alias "IsZoomed" (ByVal hWnd As Long) As Long
    Private Declare Function apiLStrCpynA Lib "Kernel32" Alias "lstrcpynA" (ByVal pDestination As
String, ByVal pSource As Long, ByVal iMaxLength As Integer) As Long
    Private Declare Function apiMessageBox Lib "User32" Alias "MessageBoxA" (ByVal hWnd As Long,
ByVal lpText As String, ByVal lpCaption As String, ByVal wType As Long) As Long
    Private Declare Function apiOpenIcon Lib "User32" Alias "OpenIcon" (ByVal hWnd As Long) As Long
    Private Declare Function apiOpenProcess Lib "Kernel32" Alias "OpenProcess" (ByVal
dwDesiredAccess As Long, ByVal bInheritHandle As Long, ByVal dwProcessId As Long) As Long
    Private Declare Function apiPathAddBackslashByPointer Lib "ShlwApi" Alias "PathAddBackslashW"
(ByVal lpszPath As Long) As Long
    Private Declare Function apiPathAddBackslashByString Lib "ShlwApi" Alias "PathAddBackslashW"
(ByVal lpszPath As String) As Long
'http://msdn.microsoft.com/en-us/library/aa155716%28office.10%29.aspx
    Private Declare Function apiPostMessage Lib "User32" Alias "PostMessageA" (ByVal hWnd As Long,
ByVal wMsg As Long, ByVal wParam As Long, ByVal lParam As Long) As Long
    Private Declare Function apiReadFile Lib "Kernel32" (ByVal hFile As Long, lpBuffer As Any,
ByVal nNumberOfBytesToRead As Long, lpNumberOfBytesRead As Long, lpOverlapped As Any) As Long
    Private Declare Function apiRegQueryValue Lib "AdvApi32" Alias "RegQueryValue" (ByVal hKey As
Long, ByVal sValueName As String, ByVal dwReserved As Long, ByRef lValueType As Long, ByVal sValue
As String, ByRef lResultLen As Long) As Long
    Private Declare Function apiSendMessage Lib "User32" Alias "SendMessageA" (ByVal hWnd As Long,
ByVal wMsg As Long, ByVal wParam As Long, lParam As Any) As Long
    Private Declare Function apiSetActiveWindow Lib "User32" Alias "SetActiveWindow" (ByVal hWnd As
Long) As Long
    Private Declare Function apiSetCurrentDirectoryA Lib "Kernel32" Alias "SetCurrentDirectoryA"
(ByVal lpPathName As String) As Long
    Private Declare Function apiSetFocus Lib "User32" Alias "SetFocus" (ByVal hWnd As Long) As Long
    Private Declare Function apiSetForegroundWindow Lib "User32" Alias "SetForegroundWindow" (ByVal
hWnd As Long) As Long
    Private Declare Function apiSetLocalTime Lib "Kernel32" Alias "SetLocalTime" (lpSystem As
SystemTime) As Long
    Private Declare Function apiSetWindowPlacement Lib "User32" Alias "SetWindowPlacement" (ByVal
hWnd As Long, ByRef lpwndpl As winPlacement) As Long
    Private Declare Function apiSetWindowPos Lib "User32" Alias "SetWindowPos" (ByVal hWnd As Long,
ByVal hWndInsertAfter As Long, ByVal X As Long, ByVal Y As Long, ByVal cx As Long, ByVal cy As
Long, ByVal wFlags As Long) As Long
    Private Declare Function apiSetWindowText Lib "User32" Alias "SetWindowTextA" (ByVal hWnd As
Long, ByVal lpString As String) As Long
    Private Declare Function apiShellExecute Lib "Shell32" Alias "ShellExecuteA" (ByVal hWnd As
Long, ByVal lpOperation As String, ByVal lpFile As String, ByVal lpParameters As String, ByVal
lpDirectory As String, ByVal nShowCmd As Long) As Long
    Private Declare Function apiShowWindow Lib "User32" Alias "ShowWindow" (ByVal hWnd As Long,
ByVal nCmdShow As Long) As Long
    Private Declare Function apiShowWindowAsync Lib "User32" Alias "ShowWindowAsync" (ByVal hWnd As
Long, ByVal nCmdShow As Long) As Long
    Private Declare Function apiStrCpy Lib "Kernel32" Alias "lstrcpynA" (ByVal pDestination As
String, ByVal pSource As String, ByVal iMaxLength As Integer) As Long
    Private Declare Function apiStringLen Lib "Kernel32" Alias "lstrlenW" (ByVal lpString As Long)
As Long
    Private Declare Function apiStrTrimW Lib "ShlwApi" Alias "StrTrimW" () As Boolean
    Private Declare Function apiTerminateProcess Lib "Kernel32" Alias "TerminateProcess" (ByVal
hWnd As Long, ByVal uExitCode As Long) As Long
```

```
    Private Declare Function apiTimeGetTime Lib "Winmm" Alias "timeGetTime" () As Long
    Private Declare Function apiVarPtrArray Lib "MsVbVm50" Alias "VarPtr" (Var() As Any) As Long
    Private Declare Function apiWaitForSingleObject Lib "Kernel32" (ByVal hHandle As Long, ByVal
dwMilliseconds As Long) As Long
    Private Type browseInfo     'used by apiBrowseForFolder
        hOwner As Long
        pidlRoot As Long
        pszDisplayName As String
        lpszTitle As String
        ulFlags As Long
        lpfn As Long
        lParam As Long
        iImage As Long
    End Type
    Private Declare Function apiBrowseForFolder Lib "Shell32" Alias "SHBrowseForFolderA"
(lpBrowseInfo As browseInfo) As Long
    Private Type CHOOSECOLOR    'used by apiChooseColor; http://support.microsoft.com/kb/153929 and
http://www.cpearson.com/Excel/Colors.aspx
        lStructSize As Long
        hWndOwner As Long
        hInstance As Long
        rgbResult As Long
        lpCustColors As String
        flags As Long
        lCustData As Long
        lpfnHook As Long
        lpTemplateName As String
    End Type
    Private Declare Function apiChooseColor Lib "ComDlg32" Alias "ChooseColorA" (pChoosecolor As
CHOOSECOLOR) As Long
    Private Type FindWindowParameters   'Custom structure for passing in the parameters in/out of
the hook enumeration function; could use global variables instead, but this is nicer
        strTitle As String  'INPUT
        hWnd As Long        'OUTPUT
    End Type                            'Find a specific window with dynamic caption from a list of
all open windows:
http://www.everythingaccess.com/tutorials.asp?ID=Bring-an-external-application-window-to-the-foregrou
nd
    Private Declare Function apiEnumWindows Lib "User32" Alias "EnumWindows" (ByVal lpEnumFunc As
Long, ByVal lParam As Long) As Long
    Private Type lastInputInfo  'used by apiGetLastInputInfo, getLastInputTime
        cbSize As Long
        dwTime As Long
    End Type
    Private Declare Function apiGetLastInputInfo Lib "User32" Alias "GetLastInputInfo" (ByRef plii
As lastInputInfo) As Long
    Private Type SystemTime
          wYear          As Integer
          wMonth         As Integer
          wDayOfWeek     As Integer
          wDay           As Integer
          wHour          As Integer
          wMinute        As Integer
          wSecond        As Integer
          wMilliseconds  As Integer
    End Type
    Private Declare Sub apiGetLocalTime Lib "Kernel32" Alias "GetLocalTime" (lpSystem As
SystemTime)
    Private Type pointAPI
        X As Long
        Y As Long
    End Type
```

```
    Private Type rectAPI
        Left_Renamed As Long
        Top_Renamed As Long
        Right_Renamed As Long
        Bottom_Renamed As Long
    End Type
    Private Type winPlacement
        length As Long
        flags As Long
        showCmd As Long
        ptMinPosition As pointAPI
        ptMaxPosition As pointAPI
        rcNormalPosition As rectAPI
    End Type
    Private Declare Function apiGetWindowPlacement Lib "User32" Alias "GetWindowPlacement" (ByVal
hWnd As Long, ByRef lpwndpl As winPlacement) As Long
    Private Type winRect
        Left As Long
        Top As Long
        Right As Long
        Bottom As Long
    End Type
    Private Declare Function apiMoveWindow Lib "User32" Alias "MoveWindow" (ByVal hWnd As Long,
xLeft As Long, ByVal yTop As Long, wWidth As Long, ByVal hHeight As Long, ByVal repaint As Long) As
Long
#Else    ' Win16 = True
#End If
```

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

0m3r	Chapters 1 and 36
Andre Terra	Chapter 1
Benno Grimm	Chapters 1 and 7
Blackhawk	Chapter 37
Bookeater	Chapter 1
Branislav Kollár	Chapters 34, 35 and 42
Comintern	Chapters 2, 3, 6, 9, 10, 13, 14, 15, 20, 23, 25, 31, 32 and 33
dadde	Chapter 2
Dave	Chapters 2, 3 and 9
Derpcode	Chapter 1
FreeMan	Chapters 13 and 25
Hosch250	Chapter 8
Hubisan	Chapter 9
hymced	Chapter 28
IvenBach	Chapter 30
Jeeped	Chapters 2, 23 and 24
Kaz	Chapters 2 and 5
Kyle	Chapter 23
LiamH	Chapter 6
litelite	Chapters 1 and 7
Logan Reed	Chapter 10
Máté Juhász	Chapter 1
Maarten van Stam	Chapters 1, 9 and 24
Macro Man	Chapters 1, 4, 7, 12 and 24
Mark.R	Chapters 2 and 44
Martin	Chapter 7
Mat's Mug	Chapters 2, 4, 5, 6, 7, 9, 10, 18, 24, 27, 28, 29, 30, 34 and 43
Miguel_Ryu	Chapter 9
Neil Mussett	Chapters 2, 4, 13, 26, 38, 40 and 45
Nick Dewitt	Chapter 1
pashute	Chapter 41
PatricK	Chapter 39
paul bica	Chapter 46
R3uK	Chapter 34
Roland	Chapter 7
RubberDuck	Chapters 3, 12, 24 and 28
SandPiper	Chapter 31
Shawn V. Wilson	Chapters 2 and 8
Sivaprasath Vadivel	Chapter 6
Stefan Pinnow	Chapters 1 and 24
Steve Rindsberg	Chapters 3, 12 and 19
SWa	Chapter 2
Tazaf	Chapter 9
Thierry Dalon	Chapter 2
Thomas G	Chapters 24 and 25
ThunderFrame	Chapters 2, 3, 7, 8, 11, 13, 14, 15, 16, 17, 18, 19, 21, 22, 24 and 26
Tim	Chapter 35
Tom	Chapter 2
Zev Spitz	Chapter 23

You may also like