eBook Gratuit

APPRENEZ VBA

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#vba

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: vba

It is an unofficial and free VBA ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official VBA.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec VBA

Remarques

Cette section fournit une vue d'ensemble de ce que vba est et pourquoi un développeur peut vouloir l'utiliser.

Il devrait également mentionner tous les grands sujets au sein de vba, et établir un lien avec les sujets connexes. La documentation de vba étant nouvelle, vous devrez peut-être créer des versions initiales de ces rubriques connexes.

Versions

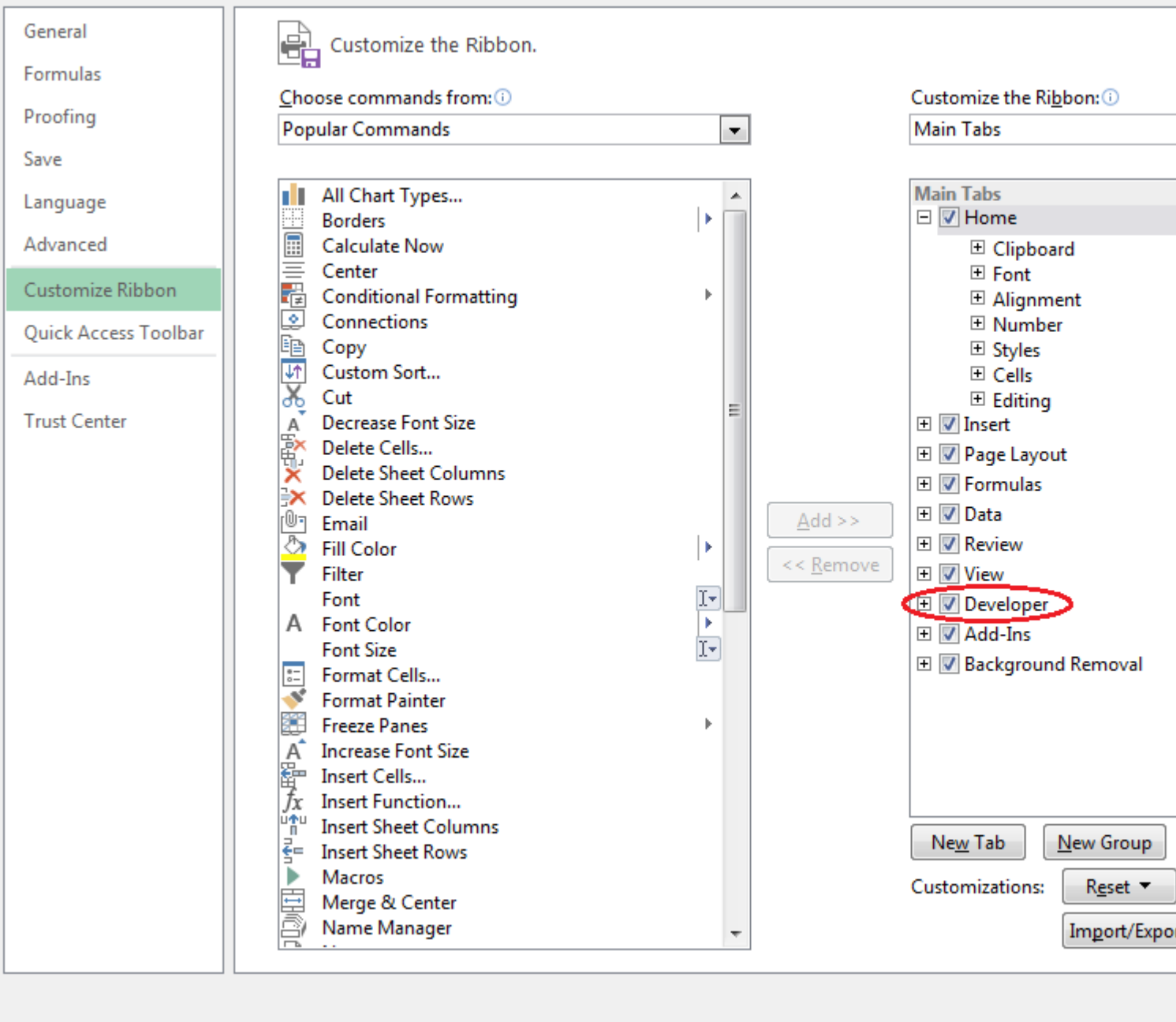
Version	Versions Office	Date de sortie Notes	Date de sortie
Vba6	? - 2007	[Quelque temps après] [1]	1992-06-30
Vba7	2010 - 2016	[blog.techkit.com] [2]	2010-04-15
VBA pour Mac	2004, 2011 - 2016		2004-05-11

Examples

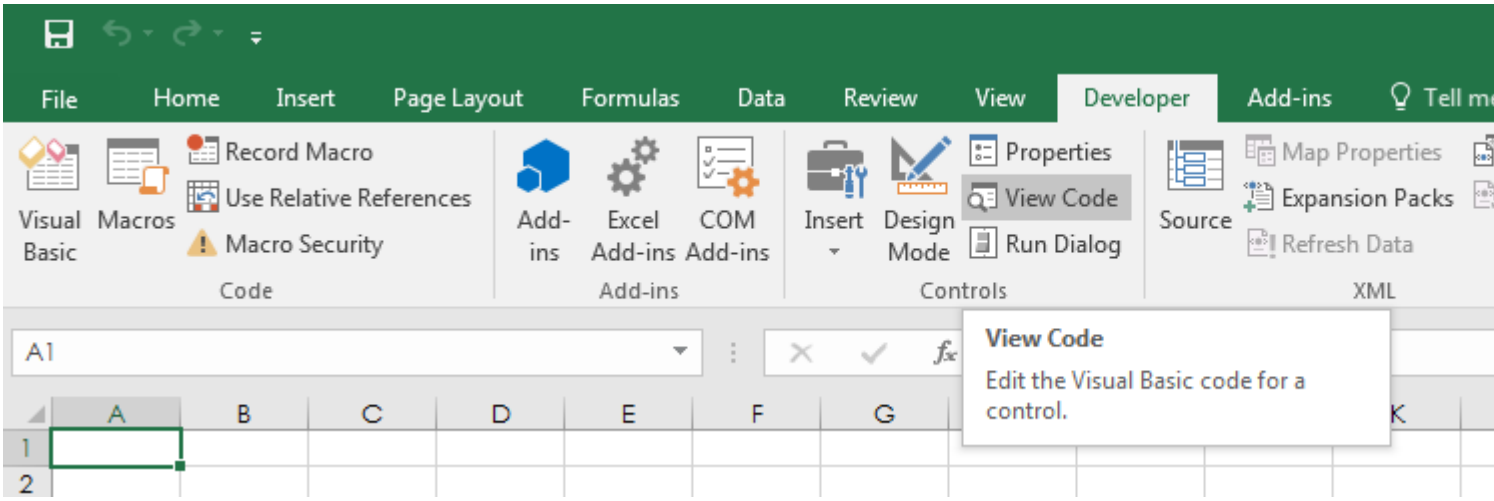
Accès à Visual Basic Editor dans Microsoft Office

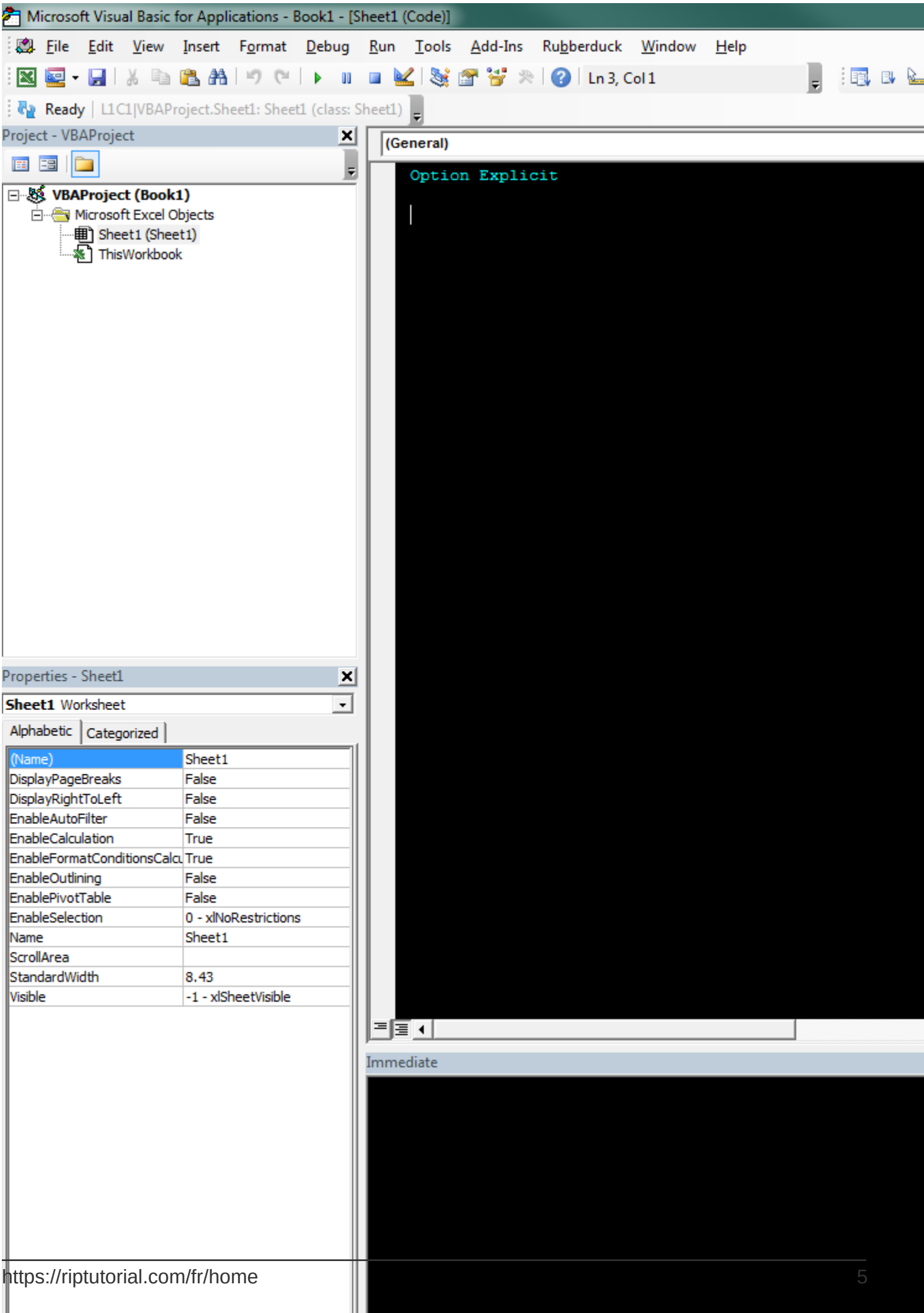
Vous pouvez ouvrir l'éditeur VB dans l'une des applications Microsoft Office en appuyant sur `Alt` + `F11` ou en accédant à l'onglet Développeur et en cliquant sur le bouton "Visual Basic". Si vous ne voyez pas l'onglet Développeur dans le ruban, vérifiez si cette option est activée.

Par défaut, l'onglet Développeur est désactivé. Pour activer l'onglet Développeur, sélectionnez Fichier -> Options, sélectionnez Personnaliser le ruban dans la liste de gauche. Dans l'arborescence de droite "Personnaliser le ruban", recherchez l'élément d'arborescence Developer et définissez la case à cocher pour la case à cocher Developer sur cochée. Cliquez sur OK pour fermer la boîte de dialogue Options.



L'onglet Développeur est maintenant visible dans le ruban sur lequel vous pouvez cliquer sur "Visual Basic" pour ouvrir Visual Basic Editor. Vous pouvez également cliquer sur "Afficher le code" pour afficher directement le volet de code de l'élément actuellement actif, par exemple WorkSheet, Chart, Shape.


File
Home
Insert
Page Layout
Formulas
Data
Review
View
Developer
Add-ins
Visual Basic
Macros
Record Macro
Use Relative References
Macro Security
Code
Add-ins
Excel Add-ins
COM Add-ins
Add-ins
Insert
Design Mode
Properties
View Code
Run Dialog
Controls
Source
Map Properties
Expansion Packs
Refresh Data
XML
A1
View Code
Edit the Visual Basic code for a control.
A
B
C
D
E
F
G
1
2


Microsoft Visual Basic for Applications - Book1 - [Sheet1 (Code)]
File Edit View Insert Format Debug Run Tools Add-Ins Rubberduck Window Help
Ln 3, Col 1
Ready
L1C1|VBAProject.Sheet1: Sheet1 (class: Sheet1)
Project - VBAProject
VBAProject (Book1)
Microsoft Excel Objects
Sheet1 (Sheet1)
ThisWorkbook
(General)
Option Explicit
Properties - Sheet1
Sheet1 Worksheet
Alphabetic
Categorized
(Name) Sheet1
DisplayPageBreaks False
DisplayRightToLeft False
EnableAutoFilter False
EnableCalculation True
EnableFormatConditionsCalcu True
EnableOutlining False
EnablePivotTable False
EnableSelection 0 - xlNoRestrictions
Name Sheet1
ScrollArea
StandardWidth 8.43
Visible -1 - xlSheetVisible
Immediate

de votre barre d'outils ou appuyez simplement sur la touche `F5` . Toutes nos félicitations! Vous avez construit votre premier module VBA.

Le débogage

Le débogage est un moyen très puissant d'examiner de plus près et de corriger le code qui ne fonctionne pas correctement.

Exécuter le code pas à pas

La première chose à faire pendant le débogage consiste à arrêter le code à des emplacements spécifiques, puis à l'exécuter ligne par ligne pour voir si cela se produit.

- Point d'arrêt (`F9` , Débogage - Activer / désactiver le point d'arrêt): vous pouvez ajouter un point d'arrêt à toute ligne exécutée (par exemple, pas aux déclarations) lorsque l'exécution s'arrête et donne le contrôle à l'utilisateur.
- Vous pouvez également ajouter le mot-clé `Stop` à une ligne vide pour que le code s'arrête à cet emplacement lors de l'exécution. Ceci est utile si, par exemple, avant les lignes de déclaration auxquelles vous ne pouvez pas ajouter de point d'arrêt avec `F9`
- Step into (`F8` , Debug - Step into): exécute une seule ligne de code, si c'est un appel d'une sous-fonction définie par l'utilisateur, alors cela est exécuté ligne par ligne.
- Step over (`Shift` + `F8` , Debug - Step over): exécute une ligne de code, n'entre pas dans les sous-parties / fonctions définies par l'utilisateur.
- Sortir (`Ctrl` + `Maj` + `F8` , Déboguer - Sortir): Quitter la sous-fonction / fonction actuelle (exécuter le code jusqu'à sa fin).
- Exécuter au curseur (`Ctrl` + `F8` , Debug - Exécuter au curseur): exécuter le code jusqu'à atteindre la ligne avec le curseur.
- Vous pouvez utiliser `Debug.Print` pour imprimer des lignes dans la fenêtre immédiate lors de l'exécution. Vous pouvez également utiliser `Debug.?` comme raccourci pour `Debug.Print`

Fenêtre montres

Exécuter le code ligne par ligne n'est que la première étape, nous avons besoin de connaître plus de détails et un outil pour cela est la fenêtre de surveillance (View - Watch window), ici vous pouvez voir les valeurs des expressions définies. Pour ajouter une variable à la fenêtre de surveillance:

- Faites un clic droit dessus puis sélectionnez "Ajouter une montre".
- Cliquez avec le bouton droit dans la fenêtre de surveillance, sélectionnez "Ajouter une montre".
- Aller au débogage - Ajouter regarder.

Lorsque vous ajoutez une nouvelle expression, vous pouvez choisir si vous souhaitez simplement voir sa valeur, ou encore casser l'exécution du code quand elle est vraie ou si sa valeur change.

Fenêtre Immédiate

La fenêtre immédiate vous permet d'exécuter du code arbitraire ou d'imprimer des éléments en les faisant précéder du mot-clé `Print` ou d'un seul point d'interrogation " `?` "

Quelques exemples:

- `? ActiveSheet.Name` - renvoie le nom de la feuille active
- `Print ActiveSheet.Name` - renvoie le nom de la feuille active
- `? foo` - renvoie la valeur de `foo` *
- `x = 10` définit `x` à 10 *

* L'obtention / la définition de valeurs pour les variables via la fenêtre immédiate ne peut être effectuée que pendant l'exécution

Débogage des meilleures pratiques

Chaque fois que votre code ne fonctionne pas comme prévu, la première chose à faire est de le relire attentivement, en recherchant les erreurs.

Si cela ne vous aide pas, commencez à le déboguer. Pour les procédures courtes, il peut être efficace de l'exécuter ligne par ligne. Pour les procédures plus longues, vous devrez probablement définir des points d'arrêt ou des ruptures sur les expressions surveillées. L'objectif est de trouver que la ligne ne fonctionne pas comme prévu.

Une fois que vous avez la ligne qui donne un résultat incorrect, mais que la raison n'est pas encore claire, essayez de simplifier les expressions ou remplacez les variables par des constantes, ce qui peut aider à comprendre si la valeur des variables est incorrecte.

Si vous ne pouvez toujours pas le résoudre, et demander de l'aide:

- Inclure la plus petite partie de votre code possible pour comprendre votre problème
- Si le problème n'est pas lié à la valeur des variables, remplacez-les par des constantes. (donc, au lieu de `Sheets(a*b*c+d^2).Range(addressOfRange)` écrire des `Sheets(4).Range("A2")`)
- Décrivez quelle ligne donne le mauvais comportement, et de quoi il s'agit (erreur, mauvais résultat ...)

Lire Démarrer avec VBA en ligne: https://riptutorial.com/fr/vba/topic/802/demarrer-avec-vba

Chapitre 2: Appels API

Introduction

API signifie Application Programming Interface

Les API pour VBA impliquent un ensemble de méthodes permettant une interaction directe avec le système d'exploitation

Les appels système peuvent être effectués en exécutant des procédures définies dans des fichiers DLL.

Remarques

Fichiers de bibliothèque de l'environnement d'exploitation courants (DLL):

Bibliothèque de liens dynamiques	La description
Advapi32.dll	Bibliothèque de services avancés pour les API, y compris de nombreux appels de sécurité et de registre
Comdlg32.dll	Bibliothèque API de dialogue commun
Gdi32.dll	Bibliothèque API d'interface graphique
Kernel32.dll	Prise en charge des API de base Windows 32 bits
Lz32.dll	Routines de compression 32 bits
Mpr.dll	Bibliothèque de plusieurs fournisseurs
Netapi32.dll	Bibliothèque d'API réseau 32 bits
Shell32.dll	Bibliothèque API 32 bits
User32.dll	Bibliothèque pour les routines d'interface utilisateur
Version.dll	Bibliothèque de versions
Winmm.dll	Bibliothèque multimédia Windows
Winspool.drv	Interface du spouleur d'impression qui contient les appels d'API du spouleur d'impression

Nouveaux arguments utilisés pour le système 64:

Type	Article	La description
Qualificatif	PtrSafe	Indique que l'instruction Declare est compatible avec 64 bits. Cet attribut est obligatoire sur les systèmes 64 bits
Type de données	LongPtr	Un type de données variable qui est un type de données à 4 octets sur les versions 32 bits et un type de données à 8 octets sur les versions 64 bits d'Office 2010. Il est recommandé de déclarer un pointeur ou un handle pour le nouveau code, mais aussi pour le code hérité s'il doit s'exécuter dans la version 64 bits d'Office 2010. Il est uniquement pris en charge dans l'environnement d'exécution VBA 7 sur 32 bits et 64 bits. Notez que vous pouvez lui affecter des valeurs numériques mais pas des types numériques
Type de données	LongLong	Il s'agit d'un type de données à 8 octets, disponible uniquement dans les versions 64 bits d'Office 2010. Vous pouvez attribuer des valeurs numériques, mais pas des types numériques (pour éviter la troncature)
Conversion	Opérateur	CLngPtr Convertit une expression simple en un type de données LongPtr
Conversion	Opérateur	CLngLng Convertit une expression simple en un type de données LongLong
Fonction	VarPtr	Variant convertisseur. Retourne un LongPtr sur les versions 64 bits et un long sur 32 bits (4 octets)
Fonction	ObjPtr	Convertisseur d'objet Retourne un LongPtr sur les versions 64 bits et un long sur 32 bits (4 octets)
Fonction	StrPtr	Convertisseur de chaîne. Retourne un LongPtr sur les versions 64 bits et un long sur 32 bits (4 octets)

Référence complète des signatures d'appel:

- Win32api32.txt pour Visual Basic 5.0 (anciennes déclarations API, dernière révision mars 2005, Microsoft)
- Win32API_PtrSafe avec prise en charge 64 bits (Office 2010, Microsoft)

Examples

Déclaration et utilisation de l'API

Déclaration d'une procédure DLL pour fonctionner avec différentes versions de VBA:

```
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
```

La déclaration ci-dessus indique à VBA comment appeler la fonction "Sleep" définie dans le fichier Kernel32.dll

Win64 et Win32 sont des constantes prédéfinies utilisées pour la compilation conditionnelle

---

Constantes prédéfinies

Certaines constantes de compilation sont déjà prédéfinies. Les versions existantes dépendront de la qualité de la version bureautique dans laquelle vous exécutez VBA. Notez que Vba7 a été introduit avec Office 2010 pour prendre en charge les versions 64 bits d'Office.

Constant	16 bits	32 bits	64 bits
Vba6	Faux	Si vba6	Faux
Vba7	Faux	Si vba7	Vrai
Win16	Vrai	Faux	Faux
Win32	Faux	Vrai	Vrai
Win64	Faux	Faux	Vrai
Mac	Faux	Si mac	Si mac

Ces constantes font référence à la version d'Office, pas à la version Windows. Par exemple, Win32 = TRUE dans Office 32 bits, même si le système d'exploitation est une version 64 bits de Windows.

La principale différence lors de la déclaration des API se situe entre les versions Office 32 bits et 64 bits qui ont introduit de nouveaux types de paramètres (voir la section Remarques pour plus de détails).

Remarques:

- Les déclarations sont placées en haut du module, et en dehors de tout Subs ou Fonctions

- Les procédures déclarées dans les modules standard sont publiques par défaut
- Pour déclarer une procédure privée à un module, faites précéder la déclaration par le mot-clé `Private`
- Les procédures DLL déclarées dans tout autre type de module sont privées à ce module

---

Exemple simple pour l'appel de l'API Sleep:

```
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
```

Il est recommandé de créer un module API dédié pour fournir un accès facile aux fonctions système à partir des wrappers VBA (sous-systèmes VBA normaux ou fonctions) qui encapsulent les détails nécessaires à l'appel système, tels que les paramètres utilisés dans les bibliothèques et l'initialisation de ces paramètres.

Le module peut contenir toutes les déclarations et dépendances:

- Signatures de méthode et structures de données requises
- Les wrappers qui effectuent la validation des entrées et vérifient que tous les paramètres sont transmis comme prévu

---

Pour déclarer une procédure DLL, ajoutez une instruction `Declare` à la section Déclarations de la fenêtre de code.

Si la procédure renvoie une valeur, déclarez-la comme **fonction** :

```
Declare Function publicname Lib "libname" [Alias "alias"] [([[ByVal] variable [As type]
[,[ByVal] variable [As type]]...])] As Type
```

Si une procédure ne renvoie pas de valeur, déclarez-la en tant que **Sub** :

```
Declare Sub publicname Lib "libname" [Alias "alias"] [([[ByVal] variable [As type] [,[ByVal]
variable [As type]]...])]
```

- !! !!! !!!!!!!!!!!!!!!!!!!!!!!

Il convient également de noter que la **plupart des appels non valides à l’API entraîneront un plantage d’Excel** , voire des fichiers de données corrompus.

- !! !! !!!!!!!!!!!!!!!!!!!!!!!!!!

Office 2011 pour 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
```

Les exemples ci-dessous (API Windows - Module dédié (1 et 2)) affichent un module API qui inclut des déclarations communes pour Win64 et Win32

API Windows - Module dédié (1 sur 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-foreground
    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
```

(continué dans le deuxième exemple)

API Windows - Module dédié (2 sur 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-foreground
    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
```

API Mac

Microsoft ne prend pas officiellement en charge les API, mais avec quelques recherches, vous pouvez trouver plus de déclarations en ligne

Office 2016 pour Mac est en bac à sable

Contrairement aux autres versions des applications Office prenant en charge VBA, les applications Office 2016 pour Mac sont en mode bac à sable.

Sandboxing empêche les applications d'accéder aux ressources en dehors du conteneur d'applications. Cela affecte tous les compléments ou macros qui impliquent l'accès aux fichiers ou

la communication entre les processus. Vous pouvez minimiser les effets du sandboxing en utilisant les nouvelles commandes décrites dans la section suivante. Nouvelles commandes VBA pour Office 2016 pour Mac

Les commandes VBA suivantes sont nouvelles et uniques à Office 2016 pour Mac.

Commander	Avoir l'habitude de
GrantAccessToMultipleFiles	Demander à un utilisateur l'autorisation d'accéder à plusieurs fichiers à la fois
AppleScriptTask	Appeler des scripts AppleScript externes à partir de VB
MAC_OFFICE_VERSION	IFDEF entre différentes versions de Mac Office au moment de la compilation

---

Office 2011 pour 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 pour 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
```

Obtenez le total des moniteurs et la résolution de l'écran

```
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
```

API FTP et régionales

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
```

Lire Appels API en ligne: https://riptutorial.com/fr/vba/topic/10569/appels-api

Chapitre 3: Appels de procédure

Syntaxe

- IdentifierName [*arguments*]
- Call IdentifierName [*(arguments)*]
- [Let | Set] *expression* = IdentifierName [*(arguments)*]
- [Let | Set] IdentifierName [*(arguments)*] = *expression*

Paramètres

Paramètre	Info
IdentifiantNom	Le nom de la procédure à appeler.
arguments	Une liste d'arguments séparés par des virgules à transmettre à la procédure.

Remarques

Les deux premières syntaxes servent à appeler des procédures `Sub` ; Notez que la première syntaxe n'implique pas de parenthèses.

Voir c'est déroutant. Pourquoi ne pas toujours utiliser les parenthèses? pour une explication approfondie des différences entre les deux premières syntaxes.

La troisième syntaxe consiste à appeler les procédures `Function` et `Property Get` ; lorsqu'il y a des paramètres, les parenthèses sont toujours obligatoires. Le mot clé `Let` est facultatif lors de l'attribution d'une *valeur* , mais le mot clé `Set` est **requis** lors de l'attribution d'une *référence* .

La quatrième syntaxe consiste à appeler les procédures `Property Let` et `Property Set` ; l' *`expression`* à droite de l'affectation est transmise au paramètre value de la propriété.

Examples

Syntaxe d'appel implicite

```
ProcedureName
ProcedureName argument1, argument2
```

Appelez une procédure par son nom sans parenthèses.

Boîtier Edge

Le mot-clé d' `Call` n'est requis que dans un seul cas:

```
Call DoSomething : DoSomethingElse
```

`DoSomething` et `DoSomethingElse` sont des procédures appelées. Si le mot-clé `Call` été supprimé, alors `DoSomething` sera analysé sous la forme d'une *étiquette de ligne* plutôt que d'un appel de procédure, ce qui briserait le code:

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

Valeurs de retour

Pour récupérer le résultat d'un appel de procédure (par exemple, les procédures `Function` ou `Property Get`), placez l'appel sur le côté droit d'une affectation:

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

Les parenthèses doivent être présentes s'il y a des paramètres. Si la procédure n'a pas de paramètres, les parenthèses sont redondantes.

Ceci est déroutant. Pourquoi ne pas toujours utiliser les parenthèses?

Les parenthèses sont utilisées pour inclure les arguments des *appels* de *fonction* . Leur utilisation pour *les appels de procédure* peut provoquer des problèmes inattendus.

Parce qu'ils peuvent introduire des bogues, à la fois en exécutant en transmettant une valeur éventuellement non intentionnelle à la procédure, et à la compilation en étant simplement une syntaxe non valide.

Temps d'exécution

Les parenthèses redondantes peuvent introduire des bogues. Étant donné une procédure qui prend une référence d'objet comme paramètre ...

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

... et appelé avec des parenthèses:

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

Cela provoquera une erreur d'exécution "Object Required" # 424. D'autres erreurs sont possibles dans d'autres circonstances: ici, la référence d'objet `Application.ActiveCell Range` est en cours d' *évaluation* et transmise par valeur, **quelle que soit** la signature de la procédure spécifiant que la `target` serait transmise `ByRef` . La valeur réelle transmise à `DoSomething` par `ByVal` dans l'extrait de

`DoSomething` ci-dessus est `Application.ActiveCell.Value` .

Les parenthèses forcent VBA à évaluer la valeur de l'expression entre crochets et à transmettre le résultat `ByVal` à la procédure appelée. Lorsque le type du résultat évalué ne correspond pas au type attendu de la procédure et ne peut pas être converti implicitement, une erreur d'exécution est générée.

Compiler-temps

Ce code ne pourra pas être compilé:

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

Parce que l'expression `("Invalid Code!", vbCritical)` ne peut pas être *évaluée* à une valeur.

Cela compilerait et fonctionnerait:

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

Mais serait vraiment idiot. Évitez les parenthèses redondantes.

Syntaxe d'appel explicite

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

La syntaxe d'appel explicite requiert le mot-clé `Call` et les parenthèses autour de la liste d'arguments; les parenthèses sont redondantes s'il n'y a pas de paramètres. Cette syntaxe a été rendue obsolète lorsque la syntaxe d'appel implicite plus moderne a été ajoutée à VB.

Arguments optionnels

Certaines procédures ont des arguments facultatifs. Les arguments facultatifs viennent toujours après les arguments requis, mais la procédure peut être appelée sans eux.

Par exemple, si la fonction `ProcedureName` devait avoir deux arguments obligatoires (`argument1` , `argument2`) et un argument optionnel, `optArgument3` , elle pourrait être appelée au moins de quatre manières:

```
' 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")
```

La structure de l'en-tête de la fonction appelée ici ressemblerait à ceci:

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

Le mot clé `Optional` indique que cet argument peut être omis. Comme mentionné précédemment, tous les arguments facultatifs introduits dans l'en-tête **doivent** apparaître à la fin, après tous les arguments requis.

Vous pouvez également fournir une valeur *par défaut* pour l'argument si une valeur n'est pas transmise à la fonction:

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

Dans cette fonction, si l'argument pour `c` n'est pas fourni, sa valeur par défaut sera `"C"` . Si une valeur *est* fournie, cela remplacera la valeur par défaut.

Lire Appels de procédure en ligne: https://riptutorial.com/fr/vba/topic/1179/appels-de-procedure

Chapitre 4: Arguments de passage ByRef ou ByVal

Introduction

Les modificateurs `ByRef` et `ByVal` font partie de la signature d'une procédure et indiquent comment un argument est transmis à une procédure. Dans VBA, un paramètre est transmis à `ByRef` sauf indication contraire (c'est-à-dire que `ByRef` est implicite s'il est absent).

Remarque Dans de nombreux autres langages de programmation (y compris VB.NET), les paramètres sont implicitement passés par valeur si aucun modificateur n'est spécifié: envisagez de spécifier explicitement les modificateurs `ByRef` pour éviter toute confusion possible.

Remarques

Tableaux passants

Les tableaux **doivent** être transmis par référence. Ce code compile, mais déclenche l'erreur d'exécution 424 "Objet requis":

```
Public Sub Test()
    DoSomething Array(1, 2, 3)
End Sub

Private Sub DoSomething(ByVal foo As Variant)
    foo.Add 42
End Sub
```

Ce code ne compile pas:

```
Private Sub DoSomething(ByVal foo() As Variant) 'ByVal is illegal for arrays
    foo.Add 42
End Sub
```

Examples

Passer des variables simples ByRef et ByVal

Passage `ByRef` ou `ByVal` indique si la valeur réelle d'un argument est transmise à `CalledProcedure` par `CallingProcedure` ou si une référence (appelée pointeur dans d'autres langues) est transmise à `CalledProcedure` .

Si un argument est transmis à `ByRef` , l'adresse mémoire de l'argument est transmise à `CalledProcedure` et toute modification de ce paramètre par `CalledProcedure` est apportée à la valeur de la `CallingProcedure` .

Si un argument est transmis à `ByVal` , la valeur réelle, et non une référence à la variable, est transmise à la `CalledProcedure` .

Un exemple simple illustrera cela clairement:

```
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
```

Un autre exemple:

```
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
```

ByRef

Modificateur par défaut

Si aucun modificateur n'est spécifié pour un paramètre, ce paramètre est implicitement transmis par référence.

```
Public Sub DoSomething1(foo As Long)
End Sub
```

```
Public Sub DoSomething2(ByRef foo As Long)
End Sub
```

Le paramètre `foo` est transmis à `ByRef` à la fois dans `DoSomething1` et `DoSomething2` .

> **Fais attention!** Si vous venez à VBA avec l'expérience d'autres langues, c'est probablement le comportement opposé à celui auquel vous êtes habitué. Dans de nombreux autres langages de programmation (y compris VB.NET), le modificateur implicite / default transmet les paramètres par valeur.

En passant par référence

- Lorsqu'une *valeur* est transmise par `ByRef` , la procédure reçoit **une référence** à la valeur.

```
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
```

L' appel ci - dessus `Test` des sorties de procédure 84. `DoSomething` est donné `foo` et reçoit une *référence* à la valeur, et travaille donc avec la même adresse de mémoire que l'appelant.

- Lorsqu'une *référence* est transmise par `ByRef` , la procédure reçoit **une référence** au pointeur.

```
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
```

Le code ci-dessus génère l' erreur d'exécution 91 , car l'appelant appelle le membre `Count` d'un objet qui n'existe plus, car `DoSomething` a reçu une *référence* au pointeur d'objet et lui a attribué la valeur `Nothing` avant le renvoi.

Forçage de ByVal sur le site d'appel

En utilisant des parenthèses sur le site d'appel, vous pouvez remplacer `ByRef` et forcer le passage d'un argument `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
```

Le code ci-dessus produit 42, que `ByRef` soit spécifié implicitement ou explicitement.

> **Fais attention!** De ce fait, l'utilisation de parenthèses superflues dans les appels de procédure peut facilement introduire des bogues. Faites attention aux espaces entre le nom de la procédure et la liste des arguments:
>
> ```
> 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
> ```

ByVal

En passant par la valeur

- Lorsqu'une *valeur* est transmise par `ByVal` , la procédure reçoit **une copie** de la valeur.

  ```
  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
  ```

  Appeler les `Test` procédure de `Test` ci-dessus 42. `DoSomething` se voit attribuer la valeur `foo` et reçoit **une copie** de la valeur. La copie est multipliée par 2, puis supprimée lorsque la procédure est terminée; la copie de l'appelant n'a jamais été modifiée.

- Lorsqu'une *référence* est transmise à `ByVal` , la procédure reçoit **une copie** du pointeur.

  ```
  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
```

Appel des `Test` procédure de `Test` ci-dessus 1. `DoSomething` se voit attribuer la valeur `foo` et reçoit *une copie* du **pointeur** sur l'objet `Collection` . Étant donné que la variable d'objet `foo` dans la portée `Test` pointe vers le même objet, l'ajout d'un élément dans `DoSomething` ajoute l'élément au même objet. Comme il s'agit d' *une copie* du pointeur, définir sa référence à `Nothing` n'affecte pas la copie de l'appelant.

Lire Arguments de passage ByRef ou ByVal en ligne:
https://riptutorial.com/fr/vba/topic/7363/arguments-de-passage-byref-ou-byval

Chapitre 5: Assigner des chaînes avec des caractères répétés

Remarques

Il y a des moments où vous devez assigner une variable de chaîne avec un caractère spécifique répété un nombre spécifique de fois. VBA propose deux fonctions principales à cet effet:

- `String` / `String$`
- `Space` / `Space$` .

Examples

Utilisez la fonction String pour attribuer une chaîne avec n caractères répétés

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

Utilisez les fonctions String et Space pour attribuer une chaîne de caractères n

```
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, " ")
```

Lire Assigner des chaînes avec des caractères répétés en ligne:
https://riptutorial.com/fr/vba/topic/3581/assigner-des-chaines-avec-des-caracteres-repetes

Chapitre 6: Automatisation ou utilisation d'autres bibliothèques

Introduction

Si vous utilisez les objets dans d'autres applications dans le cadre de votre application Visual Basic, vous souhaiterez peut-être établir une référence aux bibliothèques d'objets de ces applications. Cette documentation fournit une liste, des sources et des exemples d'utilisation des bibliothèques de différents logiciels, tels que Windows Shell, Internet Explorer, XML HttpRequest, etc.

Syntaxe

- expression.CreateObject (ObjectName)
- expression; Champs obligatoires. Une expression qui renvoie un objet Application.
- ObjectName; Chaîne requise. Le nom de classe de l'objet à créer. Pour plus d'informations sur les noms de classe valides, voir Identificateurs programmatiques OLE.

Remarques

- MSDN-Understanding Automation

  Lorsqu'une application prend en charge l'automatisation, Visual Basic permet d'accéder aux objets exposés par l'application. Utilisez Visual Basic pour manipuler ces objets en appelant des méthodes sur l'objet ou en obtenant et en définissant les propriétés de l'objet.

- MSDN-Check ou Ajouter une référence de bibliothèque d'objets

  Si vous utilisez les objets dans d'autres applications dans le cadre de votre application Visual Basic, vous souhaiterez peut-être établir une référence aux bibliothèques d'objets de ces applications. Avant de pouvoir le faire, vous devez d'abord vous assurer que l'application fournit une bibliothèque d'objets.

- Boîte de dialogue MSDN-References

  Vous permet de sélectionner les objets d'une autre application que vous souhaitez mettre à disposition dans votre code en définissant une référence à la bibliothèque d'objets de cette application.

- Méthode MSDN-CreateObject

  Crée un objet Automation de la classe spécifiée. Si l'application est déjà en cours d'exécution, CreateObject va créer une nouvelle instance.

Examples

Expressions régulières VBScript

```
Set createVBScriptRegExObject = CreateObject("vbscript.RegExp")
```

Outils> Références> Expressions régulières Microsoft VBScript #. #
DLL associée: VBScript.dll
Source: Internet Explorer 1.0 et 5.5

- MSDN-Microsoft enrichit VBScript avec des expressions régulières
- Syntaxe des expressions MSDN-Regular (Scripting)
- experts-exchange - Utilisation d'expressions régulières dans Visual Basic pour Applications et Visual Basic 6
- Comment utiliser les expressions régulières (Regex) dans Microsoft Excel à la fois dans la cellule et les boucles sur SO.
- regular-expressions.info/vbscript
- regular-expressions.info/vbscriptexample
- WIKI-Expression régulière

Code

Vous pouvez utiliser cette fonction pour obtenir des résultats RegEx, concaténer toutes les correspondances (si plus de 1) en 1 chaîne et afficher le résultat dans la cellule Excel.

```
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
```

Objet de système de fichiers de script

```
Set createScriptingFileSystemObject = CreateObject("Scripting.FileSystemObject")
```

Outils> Références> Microsoft Scripting Runtime
DLL associée: ScrRun.dll
Source: Windows OS

MSDN-Accès aux fichiers avec FileSystemObject

> Le modèle FSO (File System Object) fournit un outil basé sur des objets pour travailler avec des dossiers et des fichiers. Il vous permet d'utiliser la syntaxe object.method familière avec un ensemble riche de propriétés, de méthodes et d'événements pour traiter les dossiers et les fichiers. Vous pouvez également utiliser les instructions et les commandes Visual Basic traditionnelles.
>
> Le modèle FSO permet à votre application de créer, modifier, déplacer et supprimer des dossiers ou de déterminer si et où des dossiers particuliers existent. Il vous permet également d'obtenir des informations sur les dossiers, tels que leurs noms et la date à laquelle ils ont été créés ou modifiés pour la dernière fois.

Rubriques MSDN-FileSystemObject : " *... explique le concept de l'objet FileSystemObject et comment l'utiliser* " exceltrick-FileSystemObject dans VBA Scripting.FileSystemObject

Objet de script de script

```
Set dict = CreateObject("Scripting.Dictionary")
```

Outils> Références> Microsoft Scripting Runtime

DLL associée: ScrRun.dll
Source: Windows OS

Objet Scripting.Dictionary
Objet Dictionnaire MSDN

Objet Internet Explorer

```
Set createInternetExplorerObject = CreateObject("InternetExplorer.Application")
```

Outils> Références> Microsoft Internet Controls
DLL associée: ieframe.dll
Source: Navigateur Internet Explorer

Objet MSDN-InternetExplorer

> Contrôle une instance de Windows Internet Explorer via l'automatisation.

Membres Basic Internet Explorer Objec

Le code ci-dessous devrait indiquer comment fonctionne l'objet IE et comment le manipuler via VBA. Je vous recommande de passer à travers, sinon il pourrait y avoir une erreur lors de plusieurs 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

La chose la plus courante avec IE est de collecter des informations sur un site Web ou de remplir un formulaire de site Web et de soumettre des informations. Nous allons voir comment le faire.

Considérons le code source *example.com* :

```
<!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>
```

Nous pouvons utiliser le code ci-dessous pour obtenir et définir des informations:

```
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
```

Que se passe-t-il? Le joueur clé ici est la **.Document,** qui est le code source HTML. Nous pouvons appliquer certaines requêtes pour obtenir les collections ou l'objet que nous voulons. Par exemple, `IE.Document.GetElementsByTagName("title")(0).innerHtml` . `GetElementsByTagName` renvoie une **collection** d'éléments HTML ayant la balise " *title* ". Il n'y a qu'une seule balise dans le code source. La **collection** est basée sur 0. Donc, pour obtenir le premier élément, nous ajoutons `(0)` . Maintenant, dans notre cas, nous ne voulons que le `innerHtml` (une chaîne), pas l'objet Object lui-même. Donc, nous spécifions la propriété que nous voulons.

Cliquez sur

Pour suivre un lien sur un site, nous pouvons utiliser plusieurs méthodes:

```
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 ou IE Best friend

Pour tirer le meilleur parti du code HTML qui est chargé dans Internet Explorer, vous pouvez (ou devriez) utiliser une autre bibliothèque, par exemple *Microsoft HTML Object Library* . Plus d'informations à ce sujet dans un autre exemple.

IE Problèmes principaux

Le problème principal avec IE est de vérifier que la page est en cours de chargement et prête à interagir avec. The `Do While... Loop` aide, mais n'est pas fiable.

En outre, utiliser IE uniquement pour effacer le contenu HTML est OVERKILL. Pourquoi? Parce que le navigateur est conçu pour naviguer, c'est-à-dire afficher la page Web avec tous les CSS, JavaScripts, Images, Popups, etc. Si vous n'avez besoin que des données brutes, envisagez une approche différente. Par exemple, en utilisant XML HTTPRequest . Plus d'informations à ce sujet dans un autre exemple.

Lire Automatisation ou utilisation d'autres bibliothèques en ligne:
https://riptutorial.com/fr/vba/topic/8916/automatisation-ou-utilisation-d-autres-bibliotheques

Chapitre 7: Caractères non latins

Introduction

VBA peut lire et écrire des chaînes dans n'importe quel langage ou script utilisant Unicode . Cependant, des règles plus strictes sont en place pour les jetons d'identifiant .

Examples

Texte non latin dans le code VBA

Dans la cellule de feuille de calcul A1, nous avons le pangram arabe suivant:

صِفْ خَلْقَ خَوْدٍ كَمِثْلِ الشَّمْسِ إِذْ بَزَغَتْ ـ يَحْظَى الضَّجِيعُ بِهَا نَجْلَاءَ مِعْطَارِ

VBA fournit les fonctions `AscW` et `ChrW` pour travailler avec des codes de caractères multi-octets. Nous pouvons également utiliser des tableaux d' `Byte` pour manipuler directement la variable chaîne:

```
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
```

Cela produit la sortie suivante pour la lettre arabe Sad :

> Premier caractère (ChrW): 1589
> Premier caractère (binaire): 00011000110101
> Valeur de chaîne (texte):?
> Valeur de chaîne (AscW): 1589
> Valeurs d'octets (en décimal): 53 | 6
> Valeurs d'octets (binaires): 00110101 | 00000110
>
> Chaîne entière
> ??? ????? ????? ??????? ??????? ??? ??????? - ????? ???????? ???? ???????
> ???????

Notez que VBA est incapable d'imprimer du texte non latin dans la fenêtre immédiate même si les fonctions de chaîne fonctionnent correctement. Ceci est une limitation de l'EDI et non du langage.

Identificateurs non latins et couverture linguistique

Les identifiants VBA (noms de variables et de fonctions) peuvent utiliser le script latin et peuvent également utiliser des scripts japonais , coréen , chinois simplifié et chinois traditionnel .

Le script latin étendu a une couverture complète pour de nombreuses langues:
Anglais, français, espagnol, allemand, italien, breton, catalan, danois, estonien, finnois, islandais, indonésien, irlandais, lojban, mapudungun, norvégien, portugais, gaélique écossais, suédois, tagalog

Certaines langues ne sont que partiellement couvertes:
Azéri, croate, tchèque, espéranto, hongrois, letton, lituanien, polonais, roumain, serbe, slovaque, slovène, turc, yoruba, gallois

Certaines langues ont peu ou pas de couverture:
Arabe, bulgare, cherokee, dzongkha, grec, hindi, macédonien, malayalam, mongol, russe, sanscrit, thaï, tibétain, ourdou, ouïghour

Les déclarations de variables suivantes sont toutes valides:

```
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
```

Notez que dans l'IDE VBA, une seule apostrophe dans un nom de variable ne transforme pas la ligne en commentaire (comme c'est le cas avec Stack Overflow).

En outre, les langages qui utilisent deux angles pour indiquer une citation «» sont autorisés à utiliser ceux des noms de variables pour la suppression du fait que les guillemets de type "" ne le sont pas.

Lire Caractères non latins en ligne: https://riptutorial.com/fr/vba/topic/10555/caracteres-non-latins

Chapitre 8: Chaînes concaténantes

Remarques

Les chaînes peuvent être concaténées ou jointes à l'aide d'un ou plusieurs opérateurs de concaténation `&` .

Les tableaux de chaînes peuvent également être concaténés à l'aide de la fonction `Join` et en fournissant une chaîne (pouvant être de longueur nulle) à utiliser entre chaque élément du tableau.

Examples

Concaténer des chaînes à l'aide de l'opérateur &

```
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"
```

Concaténer un tableau de chaînes à l'aide de la fonction Join

```
'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"
```

Lire Chaînes concaténantes en ligne: https://riptutorial.com/fr/vba/topic/3580/chaines-concatenantes

Chapitre 9: Collections

Remarques

Une `Collection` est un objet conteneur inclus dans le runtime VBA. Aucune référence supplémentaire n'est requise pour l'utiliser. Une `Collection` peut être utilisée pour stocker des éléments de n'importe quel type de données et permet leur récupération par l'index ordinal de l'élément ou en utilisant une clé unique facultative.

Comparaison des fonctionnalités avec les tableaux et les dictionnaires

	Collection	Tableau	dictionnaire
Peut être redimensionné	Oui	Parfois [1]	Oui
Les articles sont commandés	Oui	Oui	Oui [2]
Les articles sont fortement typés	Non	Oui	Non
Les objets peuvent être récupérés par ordinal	Oui	Oui	Non
De nouveaux éléments peuvent être insérés à l'ordinal	Oui	Non	Non
Comment déterminer si un article existe	Itérer tous les articles	Itérer tous les articles	Itérer tous les articles
Les éléments peuvent être récupérés par clé	Oui	Non	Oui
Les clés sont sensibles à la casse	Non	N / A	Facultatif [3]
Comment déterminer si une clé existe	Gestionnaire d'erreur	N / A	`.Exists` fonction
Supprimer tous les articles	`.Remove` et `.Remove`	`Erase` , `ReDim`	`.RemoveAll` fonction

[1] Seuls les tableaux dynamiques peuvent être redimensionnés et seule la dernière dimension des tableaux multidimensionnels.

[2] Les `.Keys` et `.Items` sous- `.Items` sont ordonnés.

[3] Déterminé par la propriété `.CompareMode` .

Examples

Ajout d'éléments à une collection

Les éléments sont ajoutés à une `Collection` en appelant sa méthode `.Add` :

Syntaxe:

```
.Add(item, [key], [before, after])
```

Paramètre	La description
article	L'article à stocker dans la `Collection` . Cela peut être essentiellement n'importe quelle valeur à laquelle une variable peut être affectée, y compris les types primitifs, les tableaux, les objets et `Nothing` .
clé	Optionnel. Une `String` qui sert d'identificateur unique pour récupérer des éléments de la `Collection` . Si la clé spécifiée existe déjà dans la `Collection` , cela entraînera une erreur d'exécution 457: "cette clé est déjà associée à un élément de cette collection".
avant	Optionnel. Une clé existante (valeur de `String`) *ou un* index (valeur numérique) pour insérer l'élément avant dans la `Collection` . Si une valeur est donnée, le paramètre *after* ***doit*** être vide ou une erreur d'exécution 5: "appel ou argument de procédure non valide". Si une clé `String` est transmise qui n'existe pas dans la `Collection` , une erreur d'exécution 5: «appel ou argument de procédure non valide» est générée. Si un index numérique est transmis qui n'existe pas dans la `Collection` , il en résultera une erreur d'exécution 9: "indice hors limites".
après	Optionnel. Une clé existante (valeur de `String`) *ou un* index (valeur numérique) pour insérer l'élément après dans la `Collection` . Si une valeur est donnée, le paramètre *before* ***doit*** être vide. Les erreurs soulevées sont identiques au paramètre *before* .

Remarques:

- Les clés ***ne*** sont ***pas*** sensibles à la casse. `.Add "Bar", "Foo"` et `.Add "Baz", "foo"` entraînera une collision de clé.
- Si aucun des paramètres facultatifs *avant* ou *après* n'est indiqué, l'élément sera ajouté après le dernier élément de la `Collection` .
- Les insertions effectuées en spécifiant un paramètre *avant* ou *après* modifieront les index numériques des membres existants pour correspondre à leur nouvelle position. Cela signifie que des précautions doivent être prises lors de l'insertion de boucles dans des index

numériques.

Échantillon utilisation:

```
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
```

Suppression d'éléments d'une collection

Les éléments sont supprimés d'une `Collection` en appelant sa méthode `.Remove` :

Syntaxe:

```
.Remove(index)
```

Paramètre	La description
indice	L'article à retirer de la `Collection` . Si la valeur passée est un type numérique ou un `Variant` avec un sous-type numérique, il sera interprété comme un index numérique. Si la valeur passée est une `String` ou un `Variant` contenant une chaîne, elle sera interprétée comme la clé. Si une clé String est transmise qui n'existe pas dans la `Collection` , une erreur d'exécution 5: «appel ou argument de procédure non valide» est générée. Si un index numérique est transmis qui n'existe pas dans la `Collection` , il en résultera une erreur d'exécution 9: "indice hors limites".

Remarques:

- La suppression d'un élément d'une `Collection` modifie les index numériques de tous les éléments après ceux-ci dans la `Collection` . `For` boucles qui utilisent des index numériques et des éléments à supprimer, exécutez *les* `Step -1` *vers l'arrière* (`Step -1`) pour empêcher les exceptions d'index et les éléments ignorés.
- Les éléments ne doivent généralement ***pas*** être retirés d'une `Collection` à l'intérieur d'une boucle `For Each` car ils peuvent donner des résultats imprévisibles.

Échantillon utilisation:

```
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
```

Obtenir le nombre d'articles d'une collection

Le nombre d'éléments dans une `Collection` peut être obtenu en appelant sa fonction `.Count` :

Syntaxe:

```
.Count()
```

Échantillon utilisation:

```
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
```

Récupération d'éléments d'une collection

Les éléments peuvent être extraits d'une `Collection` en appelant la fonction `.Item` .

Syntaxe:

```
.Item(index)
```

Paramètre	La description
indice	L'article à récupérer de la `Collection` . Si la valeur passée est un type numérique ou un `Variant` avec un sous-type numérique, il sera interprété comme un index numérique. Si la valeur passée est une `String` ou un `Variant` contenant une chaîne, elle sera interprétée comme la clé. Si une clé String est transmise qui n'existe pas dans la `Collection` , une erreur d'exécution 5: «appel ou argument de procédure non valide» est générée. Si un index numérique est transmis qui n'existe pas dans la `Collection` , il en résultera une erreur d'exécution 9: "indice hors limites".

Remarques:

- `.Item` est le membre par défaut de `Collection` . Cela permet une flexibilité dans la syntaxe, comme le montre l'exemple d'utilisation ci-dessous.
- Les index numériques sont basés sur 1.
- Les clés ***ne*** sont ***pas*** sensibles à la casse. `.Item("Foo")` et `.Item("foo")` font référence à la même clé.
- Le paramètre d' *index* n'est ***pas*** implicitement converti en nombre à partir d'une `String` ou vice-versa. Il est tout à fait possible que `.Item(1)` et `.Item("1")` réfèrent à différents éléments de la `Collection` .

Exemple d'utilisation (index):

```
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
```

Exemple d'utilisation (clés):

```
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
```

Exemple d'utilisation (syntaxe alternative):

```
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
```

Notez que la syntaxe bang (`!`) Est autorisée car `.Item` est le membre par défaut et peut prendre un seul argument `String` . L'utilité de cette syntaxe est discutable.

Déterminer si une clé ou un objet existe dans une collection

Clés

Contrairement à un scripting.Dictionary , une `Collection` ne possède pas de méthode pour déterminer si une clé donnée existe *ou* pour récupérer des clés présentes dans la `Collection` . La seule méthode pour déterminer si une clé est présente est d'utiliser le gestionnaire d'erreurs:

```
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
```

Articles

La seule façon de déterminer si un élément est contenu dans une `Collection` consiste à parcourir la `Collection` jusqu'à ce que l'élément soit localisé. Notez qu'une `Collection` peut contenir des primitives ou des objets, une manipulation supplémentaire est nécessaire pour éviter les erreurs d'exécution lors des comparaisons:

```
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
```

Effacer tous les articles d'une collection

Le moyen le plus simple d’effacer tous les éléments d’une `Collection` est de simplement le remplacer par une nouvelle `Collection` et de laisser l’ancien hors de portée:

```
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
```

Cependant, s'il existe plusieurs références à la `Collection` , cette méthode ne vous donnera qu'une `Collection` vide *pour la variable affectée* .

```
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
```

Dans ce cas, la méthode la plus simple pour effacer le contenu consiste à parcourir le nombre d'éléments de la `Collection` et à supprimer de manière répétée l'élément le plus bas:

```
Public Sub ClearCollection(ByRef container As Collection)
    Dim index As Long
    For index = 1 To container.Count
        container.Remove 1
    Next
End Sub
```

Lire Collections en ligne: https://riptutorial.com/fr/vba/topic/5838/collections

Chapitre 10: commentaires

Remarques

Blocs de commentaires

Si vous avez besoin de commenter ou de commenter plusieurs lignes à la fois, vous pouvez utiliser les boutons d' **édition de la barre d'outils de** l'EDI:

> **Bloc de commentaire** - Ajoute une seule apostrophe au début de toutes les lignes sélectionnées

> **Uncomment Block** - Supprime la première apostrophe du début de toutes les lignes sélectionnées

Commentaires multilignes De nombreux autres langages prennent en charge les commentaires de bloc multi-lignes, mais VBA n'autorise que les commentaires sur une seule ligne.

Examples

Commentaires Apostrophe

Un commentaire est marqué par une apostrophe (`'`) et ignoré lorsque le code est exécuté. Les commentaires aident à expliquer votre code aux futurs lecteurs, y compris vous-même.

Comme toutes les lignes commençant par un commentaire sont ignorées, elles peuvent également être utilisées pour empêcher l'exécution du code (pendant le débogage ou le refactorisation). Placer une apostrophe `'` avant votre code en fait un commentaire. (Ceci s'appelle *commenter* la ligne.)

```
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
```

Commentaires de REM

```
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
```

Lire commentaires en ligne: https://riptutorial.com/fr/vba/topic/2059/commentaires

Chapitre 11: Compilation conditionnelle

Examples

Modification du comportement du code au moment de la compilation

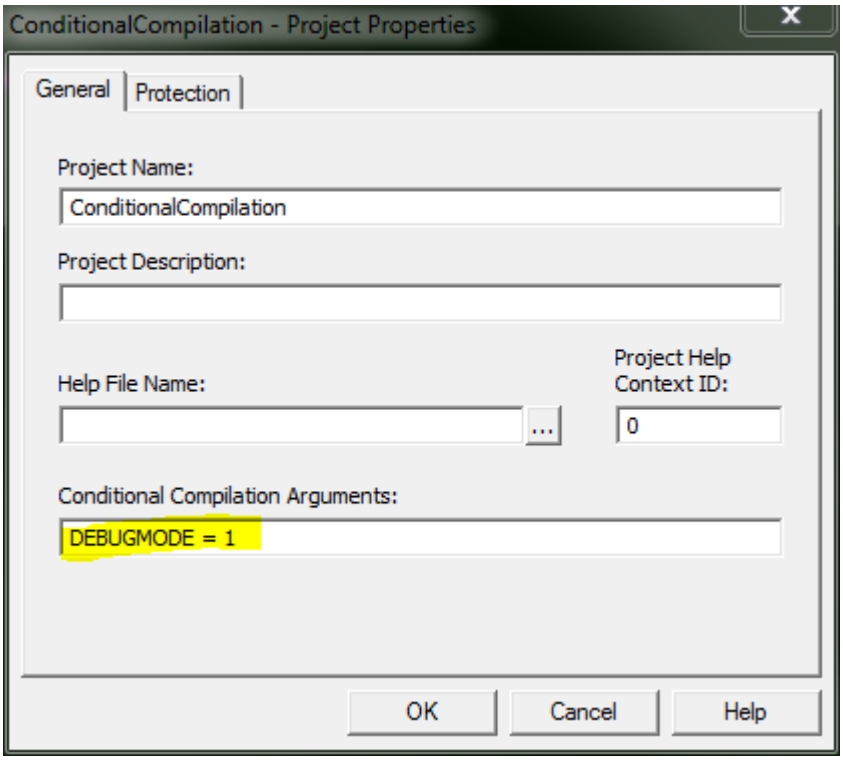
La directive `#Const` est utilisée pour définir une constante de préprocesseur personnalisée. Ceux-ci peuvent ensuite être utilisés par `#If` pour contrôler les blocs de code compilés et exécutés.

```
#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
```

Cela se traduit par la valeur de `filepath` définie sur `"C:\Users\UserName\Path\To\File.txt"` . La suppression de la ligne `#Const` ou son `#Const DEBUGMODE = 0` par `#Const DEBUGMODE = 0` entraînerait la `filepath` du `filepath` au `filepath` `"\\server\share\path\to\file.txt"` .

#Const Scope

La directive `#Const` n'est efficace que pour un seul fichier de code (module ou classe). Il doit être déclaré pour chaque fichier que vous souhaitez utiliser avec votre constante personnalisée. Vous pouvez également déclarer un `#Const` globalement pour votre projet en allant dans Outils >> [Nom de votre projet] Propriétés du projet. Cela fera apparaître la boîte de dialogue des propriétés du projet dans laquelle nous entrerons la déclaration de la constante. Dans la zone «Arguments de compilation conditionnelle», tapez `[constName] = [value]` . Vous pouvez entrer plus d'une constante en les séparant par deux points, comme `[constName1] = [value1] : [constName2] = [value2]` .



Constantes prédéfinies

Certaines constantes de compilation sont déjà prédéfinies. Les versions existantes dépendront de la qualité de la version bureautique dans laquelle vous exécutez VBA. Notez que Vba7 a été introduit avec Office 2010 pour prendre en charge les versions 64 bits d'Office.

Constant	16 bits	32 bits	64 bits
Vba6	Faux	Si vba6	Faux
Vba7	Faux	Si vba7	Vrai
Win16	Vrai	Faux	Faux
Win32	Faux	Vrai	Vrai
Win64	Faux	Faux	Vrai
Mac	Faux	Si mac	Si mac

Notez que Win64 / Win32 fait référence à la version d'Office, pas à la version de Windows. Par exemple, Win32 = TRUE dans Office 32 bits, même si le système d'exploitation est une version 64 bits de Windows.

Utilisation des déclarations de déclaration qui fonctionnent sur toutes les versions d'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
```

Cela peut être simplifié un peu en fonction des versions de bureau à prendre en charge. Par exemple, peu de personnes prennent en charge les versions 16 bits d'Office. La dernière version de bureau 16 bits était la version 4.3, publiée en 1994. La déclaration suivante est donc suffisante pour presque tous les cas modernes (y compris 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
```

Si vous n'êtes pas obligé de prendre en charge une version antérieure à Office 2010, cette déclaration fonctionne correctement.

```
' 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
```

Lire Compilation conditionnelle en ligne: https://riptutorial.com/fr/vba/topic/3364/compilation-conditionnelle

Chapitre 12: Conventions de nommage

Examples

Noms variables

Les variables contiennent des données. Nommez-les après ce qu'ils ont utilisé, **pas après leur type de données** ou leur étendue, en utilisant un **nom** . Si vous vous sentez obligé de *numéroter* vos variables (par exemple, `thing1, thing2, thing3`), envisagez plutôt d'utiliser une structure de données appropriée (par exemple, un tableau, une `Collection` ou un `Dictionary`).

Les noms des variables qui représentent un *ensemble* de valeurs itérables - par exemple, un tableau, une `Collection` , un `Dictionary` ou une `Range` de cellules, doivent être au pluriel.

Certaines conventions de nommage VBA courantes vont donc:

Pour les variables de niveau procédure :

`camelCase`

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

    Dim procedureVariable As Long
    Dim someOtherVariable As String

End Sub
```

Pour les variables de niveau module:

`PascalCase`

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

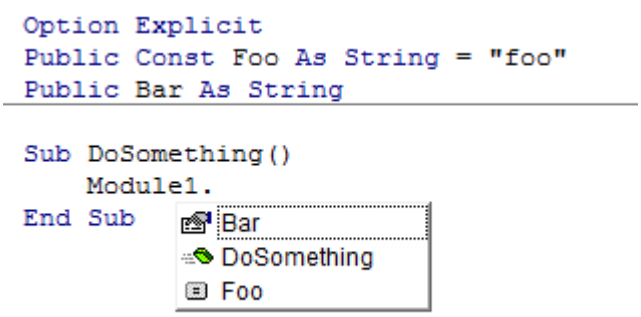
Pour les constantes:

`SHOUTY_SNAKE_CASE` est couramment utilisé pour différencier les constantes des 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
```

Cependant, les noms `PascalCase` font un code plus propre et sont tout aussi bons, étant donné

qu'IntelliSense utilise des icônes différentes pour les variables et les constantes:



Notation hongroise

> Nommez-les après leur utilisation, et **non après leur type de données** ou leur étendue.

"La notation hongroise permet de voir plus facilement le type d'une variable"

Si vous écrivez votre code tel que les procédures respectent le *principe de la responsabilité unique* (comme il se doit), vous ne devriez jamais regarder un écran de déclarations de variables en tête de procédure; déclare les variables aussi proches que possible de leur première utilisation et leur type de données sera toujours visible si vous les déclarez avec un type explicite. Le raccourci `Ctrl` + `i` du VBE peut également être utilisé pour afficher le type d'une variable dans une info-bulle.

A quoi sert une variable est beaucoup plus d'informations utiles que son type de données, en *particulier* dans un langage tel que VBA qui convertit joyeusement et implicitement un type en un autre si nécessaire.

Considérez `iFile` et `strFile` dans cet exemple:

```
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
```

Comparer aux:

```
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` est passé `ByRef` dans le premier exemple, mais à part le fait que nous avons la chance de voir qu'il est *explicitement* passé en tant que tel, rien n'indique que `strData` soit réellement *renvoyé* par la fonction.

L'exemple inférieur le nomme `outContent` ; ce préfixe `out` est ce que la notation hongroise a été inventée pour: aider à clarifier à *quoi sert une variable* , dans ce cas-ci pour l'identifier clairement comme un paramètre "out".

Ceci est utile, car IntelliSense en lui-même n'affiche pas `ByRef` , même si le paramètre est *explicitement* passé par référence:

```
Public Sub DoSomething()
    if CanReadFile(path, 
End Sub
```
CanReadFile(*ByVal path As String*, ***outContent As String***) As Boolean

Qui conduit à...

Hongrois Fait Bien

La notation hongroise n'avait à l'origine rien à voir avec les types de variables . En fait, la notation hongroise *faite correctement* est réellement utile. Considérez ce petit exemple (`ByVal` et `As Integer` supprimés pour des raisons de brièveté):

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

Comparer aux:

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

`src` et `dst` sont *les* préfixes de *notation hongrois* ici, et ils transmettent *des* informations *utiles* qui ne peuvent pas déjà être déduites des noms de paramètres ou d'IntelliSense, nous indiquant le type déclaré.

Bien sûr, il existe un meilleur moyen de tout transmettre, en utilisant une *abstraction* correcte et des mots réels qui peuvent être prononcés à haute voix et qui ont du sens - comme un exemple artificiel:

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

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

Noms de procédure

Les procédures *font quelque chose* . Nommez-les après ce qu'ils font, en utilisant un **verbe** . S'il est impossible de nommer avec précision une procédure, il est probable que la procédure *fait trop de choses* et doit être divisée en procédures plus petites et plus spécialisées.

Certaines conventions de nommage VBA courantes vont donc:

Pour toutes les procédures:

`PascalCase`

```
Public Sub DoThing()

End Sub

Private Function ReturnSomeValue() As [DataType]

End Function
```

Pour les procédures de gestionnaire d'événement:

`ObjectName_EventName`

```
Public Sub Workbook_Open()

End Sub

Public Sub Button1_Click()

End Sub
```

Les gestionnaires d'événements sont généralement nommés automatiquement par le VBE; les

renommer sans renommer l'objet et / ou l'événement manipulé cassera le code - le code s'exécutera et se compilera, mais la procédure du gestionnaire sera orpheline et ne sera jamais exécutée.

Membres booléens

Considérons une fonction booléenne:

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

Comparer aux:

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

Le `Can` préfixe *ne* sert le même but que le `b` préfixe: il identifie la valeur de retour de la fonction comme `Boolean` . Mais `Can` lit mieux que `b` :

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

Par rapport à:

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

Envisagez d'utiliser des préfixes tels que `Can` , `Is` ou `Has` devant les membres renvoyant des booléens (fonctions et propriétés), mais uniquement lorsque cela ajoute de la valeur. Cela est conforme aux directives de dénomination actuelles de Microsoft .

Lire Conventions de nommage en ligne: https://riptutorial.com/fr/vba/topic/1184/conventions-de-nommage

Chapitre 13: Conversion d'autres types en chaînes

Remarques

VBA convertira implicitement certains types en chaîne si nécessaire et sans travail supplémentaire de la part du programmeur, mais VBA fournit également un certain nombre de fonctions de conversion de chaînes explicites, et vous pouvez également écrire vos propres fonctions.

Les trois fonctions les plus fréquemment utilisées sont `CStr` , `Format` et `StrConv` .

Examples

Utilisez CStr pour convertir un type numérique en chaîne

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

Utilisez Format pour convertir et formater un type numérique en chaîne

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

Utiliser StrConv pour convertir un tableau d'octets de caractères à un octet en chaîne

```
'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"
```

Convertit implicitement un tableau d'octets de caractères multi-octets en chaîne

```
'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"
```

Lire Conversion d'autres types en chaînes en ligne:
https://riptutorial.com/fr/vba/topic/3467/conversion-d-autres-types-en-chaines

Chapitre 14: Copier, retourner et passer des tableaux

Examples

Copier des tableaux

Vous pouvez copier un tableau VBA dans un tableau du même type en utilisant l'opérateur `=` . Les tableaux doivent être du même type, sinon le code lancera une erreur de compilation "Impossible d'affecter au tableau".

```
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
```

Le tableau source peut être fixe ou dynamique, mais le tableau de destination doit être dynamique. Essayer de copier dans un tableau fixe lancera une erreur de compilation "Impossible d'affecter au tableau". Toutes les données préexistantes dans le tableau de réception sont perdues et ses limites et dimenions sont remplacées par celles du tableau source.

```
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)
```

Une fois la copie effectuée, les deux tableaux sont séparés en mémoire, c'est-à-dire que les deux variables ne sont pas des références aux mêmes données sous-jacentes, de sorte que les modifications apportées à un tableau n'apparaissent pas dans l'autre.

```
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
```

Copie de tableaux d'objets

Avec les tableaux d'objets, les *références* à ces objets sont copiées, pas les objets eux-mêmes. Si une modification est apportée à un objet dans un tableau, il semblera également être modifié dans l'autre tableau - ils font tous deux référence au même objet. Cependant, définir un élément sur un objet différent dans un tableau ne le définira pas sur cet autre tableau.

```
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
```

Variantes contenant un tableau

Vous pouvez également copier un tableau dans et à partir d'une variable variante. Lors de la copie à partir d'une variante, il doit contenir un tableau du même type que le tableau de réception, sinon une erreur d'exécution "Incompatibilité de type" apparaîtra.

```
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
```

Retour de tableaux à partir de fonctions

Une fonction dans un module normal (mais pas un module Class) peut retourner un tableau en mettant `()` après le type de données.

```
Function arrayOfPiDigits() As Long()
    Dim outputArray(0 To 2) As Long
```

```
    outputArray(0) = 3
    outputArray(1) = 1
    outputArray(2) = 4

    arrayOfPiDigits = outputArray
End Function
```

Le résultat de la fonction peut alors être placé dans un tableau dynamique du même type ou d'une variante. Les éléments peuvent également être accédés directement en utilisant un deuxième ensemble de parenthèses, mais cela appellera la fonction à chaque fois, il est donc préférable de stocker les résultats dans un nouveau tableau si vous prévoyez de les utiliser plusieurs fois.

```
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
```

Notez que ce qui est retourné est en fait une copie du tableau à l'intérieur de la fonction, pas une référence. Donc, si la fonction renvoie le contenu d'un tableau statique, ses données ne peuvent pas être modifiées par la procédure d'appel.

Sortie d'un tableau via un argument de sortie

Normalement, il est conseillé de coder pour que les arguments d'une procédure soient des entrées et une sortie via la valeur de retour. Cependant, les limitations de VBA obligent parfois une procédure à produire des données via un argument `ByRef` .

Sortie vers un tableau fixe

```
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
```

Sortie d'un tableau à partir d'une méthode de classe

Un argument de sortie peut également être utilisé pour générer un tableau à partir d'une méthode / procédure dans un module de classe.

```
' 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
```

Passer des tableaux à des procédures

Les tableaux peuvent être passés aux procédures en mettant `()` après le nom de la variable de tableau.

```
Function countElements(ByRef arr() As Double) As Long
    countElements = UBound(arr) - LBound(arr) + 1
End Function
```

Les tableaux *doivent* être transmis par référence. Si aucun mécanisme de transmission n'est spécifié, par exemple `myFunction(arr())` , alors VBA prendra `ByRef` par défaut, mais il est `ByRef` de le coder pour le rendre explicite. Essayer de passer un tableau par valeur, par exemple `myFunction(ByVal arr())` provoquera une erreur de compilation "Array argument doit être ByRef" (ou une erreur de compilation "Syntax error" si l'option `Auto Syntax Check` n'est pas cochée dans les options VBE) .

Passer par référence signifie que toute modification apportée à la matrice sera préservée dans la procédure d'appel.

```
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
```

Si vous voulez éviter de changer le tableau d'origine, faites attention à écrire la fonction pour qu'elle ne change aucun élément.

```
Function doubleAndSum(ByRef arr() As Long)
    doubleAndSum = arr(0) * 2 + arr(1) * 2
End Function
```

Vous pouvez également créer une copie de travail du tableau et travailler avec la copie.

```
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
```

Lire Copier, retourner et passer des tableaux en ligne:
https://riptutorial.com/fr/vba/topic/9069/copier--retourner-et-passer-des-tableaux

Chapitre 15: CreateObject vs. GetObject

Remarques

Dans sa forme la plus simple, `CreateObject` crée une instance d'un objet alors que `GetObject` obtient une instance existante d'un objet. Déterminer si un objet peut être créé ou obtenu dépend de sa propriété Instancing . Certains objets sont SingleUse (par exemple, WMI) et ne peuvent pas être créés s'ils existent déjà. D'autres objets (par exemple, Excel) sont MultiUse et permettent à plusieurs instances de s'exécuter simultanément. Si une instance d'un objet n'existe pas déjà et que vous tentez `GetObject` , vous recevrez le message récupérable suivant: Erreur d'exécution `Run-time error '429': ActiveX component can't create object` .

GetObject nécessite qu'au moins un de ces deux paramètres facultatifs soit présent:

1. *Pathname* - Variant (String): chemin d'accès complet, y compris nom de fichier, du fichier contenant l'objet. Ce paramètre est facultatif, mais *Class* est requis si le *chemin d'accès* est omis.
2. *Class* - Variant (String): Chaîne représentant la définition formelle (Application et ObjectType) de l'objet. *La classe* est obligatoire si le *chemin d'accès* est omis.

CreateObject a un paramètre requis et un paramètre facultatif:

1. *Class* - Variant (String): Chaîne représentant la définition formelle (Application et ObjectType) de l'objet. *La classe* est un paramètre obligatoire.
2. *Servername* - Variant (String): nom de l'ordinateur distant sur lequel l'objet sera créé. S'il est omis, l'objet sera créé sur la machine locale.

La classe est toujours composée de deux parties sous la forme de `Application.ObjectType` :

1. *Application* - Nom de l'application dont l'objet fait partie. |
2. *Type d'objet* - Le type d'objet en cours de création. |

Quelques exemples de classes sont:

1. Word.Application
2. Feuille de calcul Excel
3. Scripting.FileSystemObject

Examples

Démonstration de GetObject et CreateObject

Fonction MSDN-GetObject

> Renvoie une référence à un objet fourni par un composant ActiveX.

Utilisez la fonction GetObject lorsqu'il existe une instance actuelle de l'objet ou si vous souhaitez créer l'objet avec un fichier déjà chargé. S'il n'y a pas d'instance actuelle et que vous ne voulez pas que l'objet démarre avec un fichier chargé, utilisez la fonction CreateObject.

```
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
```

Lire CreateObject vs. GetObject en ligne: https://riptutorial.com/fr/vba/topic/7729/createobject-vs--getobject

Chapitre 16: Créer une classe personnalisée

Remarques

Cet article montre comment créer une classe personnalisée complète dans VBA. Il utilise l'exemple d'un objet `DateRange` , car les dates de début et de fin sont souvent transmises aux fonctions.

Examples

Ajout d'une propriété à une classe

Une procédure de `Property` est une série d'instructions qui récupère ou modifie une propriété personnalisée sur un module.

Il existe trois types d'accesseurs de propriétés:

1. Une procédure `Get` qui renvoie la valeur d'une propriété.
2. Une procédure `Let` qui affecte une valeur (non `Object`) à un objet.
3. Une procédure `Set` qui attribue une référence à un `Object` .

Les accesseurs de propriétés sont souvent définis par paires, en utilisant à la fois un `Get` et un `Let` / `Set` pour chaque propriété. Une propriété avec uniquement une procédure `Get` serait en lecture seule, tandis qu'une propriété avec uniquement une procédure `Let` / `Set` serait en écriture seule.

Dans l'exemple suivant, quatre accesseurs de propriétés sont définis pour la classe `DateRange` :

1. `StartDate` (*lecture / écriture*). Valeur de date représentant la date antérieure dans une plage. Chaque procédure utilise la valeur de la variable de module, `mStartDate` .
2. `EndDate` (*lecture / écriture*) Valeur de date représentant la date ultérieure dans une plage. Chaque procédure utilise la valeur de la variable de module, `mEndDate` .
3. `DaysBetween` (*lecture seule*). Valeur entière calculée représentant le nombre de jours entre les deux dates. Comme il n'y a qu'une procédure `Get` , cette propriété ne peut pas être modifiée directement.
4. `RangeToCopy` (en *écriture seule*). Une procédure `Set` permet de copier les valeurs d'un objet `DateRange` existant.

```
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
```

Ajout de fonctionnalités à une classe

Toute `Sub` - `Function` , `Function` ou `Property` publique à l'intérieur d'un module de classe peut être appelée en précédant l'appel par une référence d'objet:

```
Object.Procedure
```

Dans une classe `DateRange` , un `Sub` pourrait être utilisé pour ajouter un nombre de jours à la date de fin:

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

Une `Function` pourrait renvoyer le dernier jour du mois suivant (notez que `GetFirstDayOfMonth` ne serait pas visible en dehors de la classe car elle est privée):

```
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
```

Les procédures peuvent accepter des arguments de tout type, y compris des références à des objets de la classe en cours de définition.

L'exemple suivant teste si l'objet `DateRange` actuel a une date de début et une date de fin qui incluent la date de début et de fin d'un autre objet `DateRange` .

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

Notez l'utilisation de la notation `Me` pour accéder à la valeur de l'objet exécutant le code.

Portée du module de classe, instanciation et réutilisation

Par défaut, un nouveau module de classe est une classe Private, il est donc *uniquement* disponible pour l'instanciation et l'utilisation dans VBProject dans lequel il est défini. Vous pouvez déclarer, instancier et utiliser la classe n'importe où dans le *même* projet:

```
'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
```

Mais souvent, vous écrirez des classes que vous souhaitez utiliser dans d'autres projets *sans* copier le module entre les projets. Si vous définissez une classe appelée `List` in `ProjectA` et que vous souhaitez utiliser cette classe dans `ProjectB` , vous devrez effectuer 4 actions:

1. Modifier la propriété d'instanciation de la classe `List` dans `ProjectA` dans la fenêtre Propriétés, de `Private` à `PublicNotCreatable`

2. Créez une fonction "factory" publique dans `ProjectA` qui crée et renvoie une instance d'une classe `List` . Généralement, la fonction de fabrique inclurait des arguments pour l'initialisation de l'instance de classe. La fonction factory est requise car la classe peut être utilisée par `ProjectB` mais `ProjectB` ne peut pas créer directement une instance de la classe `ProjectA` .

   ```
   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. Dans `ProjectB` ajoutez une référence à `ProjectA` utilisant le menu `Tools..References...`

4. Dans `ProjectB` , déclarez une variable et assignez-lui une instance de `List` utilisant la fonction factory de `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
```

Lire Créer une classe personnalisée en ligne: https://riptutorial.com/fr/vba/topic/4464/creer-une-classe-personnalisee

Chapitre 17: Créer une procédure

Examples

Introduction aux procédures

Un `Sub` est une procédure qui effectue une tâche spécifique mais ne renvoie pas de valeur spécifique.

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

Si aucun modificateur d'accès n'est spécifié, une procédure est `Public` par défaut.

Une `Function` est une procédure qui reçoit des données et renvoie une valeur, idéalement sans effets secondaires globaux ou de portée de module.

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

Une `Property` est une procédure qui *encapsule des* données de module. Une propriété peut avoir jusqu'à 3 accesseurs: `Get` une valeur ou une référence d'objet, `Let` pour affecter une valeur et / ou `Set` pour affecter une référence d'objet.

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

Les propriétés sont généralement utilisées dans les modules de classe (bien qu'elles soient également autorisées dans les modules standard), exposant l'accesseur à des données inaccessibles au code appelant. Une propriété qui expose uniquement un accesseur `Get` est "en lecture seule"; une propriété qui exposerait uniquement un accesseur `Let` et / ou `Set` est "en écriture seule". Les propriétés en écriture seule ne sont pas considérées comme une bonne pratique de programmation - si le code client peut *écrire* une valeur, il doit pouvoir le *lire* . Envisagez d'implémenter une procédure `Sub` au lieu de créer une propriété en écriture seule.

Retourner une valeur

Une `Function` ou une procédure d' `Property Get` peut (et devrait!) Renvoyer une valeur à son appelant. Cela se fait en attribuant l'identifiant de la procédure:

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

Fonction avec exemples

Comme indiqué ci-dessus, les fonctions sont des procédures plus petites qui contiennent de petits morceaux de code qui peuvent être répétitifs dans une procédure.

Les fonctions permettent de réduire la redondance dans le code.

Semblable à une procédure, une fonction peut être déclarée avec ou sans liste d'arguments.

La fonction est déclarée comme type de retour, toutes les fonctions renvoyant une valeur. Le nom et la variable de retour d'une fonction sont identiques.

1. Fonction avec paramètre:

```
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. Fonction sans paramètre:

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

La fonction peut être appelée de différentes manières dans une fonction. Comme une fonction déclarée avec un type de retour est essentiellement une variable. il est similaire à une variable.

Appels fonctionnels:

```
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
```

De plus, la fonction peut également être utilisée comme condition pour if et d'autres instructions conditionnelles.

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

De plus, les fonctions peuvent avoir des modificateurs tels que By ref et By val pour leurs arguments.

Lire Créer une procédure en ligne: https://riptutorial.com/fr/vba/topic/1474/creer-une-procedure

Chapitre 18: Date Manipulation

Examples

Calendrier

VBA prend en charge 2 calendriers: Gregorian et Hijri

La propriété `Calendar` est utilisée pour modifier ou afficher le calendrier actuel.

Les 2 valeurs du calendrier sont les suivantes:

Valeur	Constant	La description
0	vbCalGreg	Calendrier grégorien (par défaut)
1	vbCalHijri	Calendrier Hijri

Exemple

```
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
```

Ce sous-groupe imprime les éléments suivants:

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

Récupérer la date et l'heure du système

VBA prend en charge 3 fonctions intégrées pour récupérer la date et / ou l'heure de l'horloge du système.

Fonction	Type de retour	Valeur de retour
À présent	Rendez-vous amoureux	Renvoie la date et l'heure actuelles
Rendez-vous amoureux	Rendez-vous amoureux	Renvoie la date de la date et de l'heure actuelles
Temps	Rendez-vous amoureux	Renvoie la partie heure de la date et de l'heure actuelles

```
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
```

Fonction minuterie

La fonction `Timer` renvoie un Single représentant le nombre de secondes écoulées depuis minuit. La précision est un centième de seconde.

```
Sub TimerExample()

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

End Sub
```

Parce que les fonctions `Now` and `Time` ne sont précises qu'en secondes, `Timer` offre un moyen

pratique d'augmenter la précision de la mesure du temps:

```
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 () teste si une expression est une date valide ou non. Renvoie un `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
```

Fonctions d'extraction

Ces fonctions prennent en paramètre un `Variant` pouvant être converti en `Date` et renvoient un `Integer` représentant une partie d'une date ou d'une heure. Si le paramètre ne peut pas être converti en une `Date` , il en résultera une erreur d'exécution 13: incompatibilité de type.

Fonction	La description	Valeur retournée
An()	Renvoie la partie année de l'argument date.	Entier (100 à 9999)
Mois()	Renvoie la partie mois de l'argument de date.	Entier (1 à 12)
Journée()	Renvoie la partie du jour de l'argument date.	Entier (1 à 31)
Jour de la semaine()	Renvoie le jour de la semaine de l'argument date. Accepte un second argument optionnel définissant le premier jour de la semaine	Entier (1 à 7)
Heure()	Renvoie la partie heure de l'argument date.	Entier (0 à 23)
Minute()	Renvoie la partie minute de l'argument date.	Entier (0 à 59)
Seconde()	Renvoie la deuxième partie de l'argument date.	Entier (0 à 59)

Exemples:

```
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
```

Fonction DatePart ()

`DatePart()` est également une fonction renvoyant une partie d'une date, mais fonctionne différemment et offre plus de possibilités que les fonctions ci-dessus. Il peut par exemple retourner le trimestre de l'année ou la semaine de l'année.

Syntaxe:

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

L' argument d' *intervalle* peut être:

Intervalle	La description
"yyyy"	Année (100 à 9999)
"y"	Jour de l'année (1 à 366)
"m"	Mois (1 à 12)
"q"	Trimestre (1 à 4)
"ww"	Semaine (1 à 53)
"w"	Jour de la semaine (1 à 7)
"ré"	Jour du mois (1 à 31)
"h"	Heure (0 à 23)
"n"	Minute (0 à 59)
"s"	Deuxième (0 à 59)

firstdayofweek est facultatif. c'est une constante qui spécifie le premier jour de la semaine. Si non spécifié, `vbSunday` est supposé.

firstweekofyear est facultatif. c'est une constante qui spécifie la première semaine de l'année. Si elle n'est pas spécifiée, la première semaine est supposée être la semaine au cours de laquelle le 1er janvier se produit.

Exemples:

```
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
```

Fonctions de calcul

DateDiff ()

`DateDiff()` renvoie un `Long` représentant le nombre d'intervalles de temps entre deux dates spécifiées.

Syntaxe

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

- *intervalle* peut être l'un des intervalles définis dans la fonction `DatePart()`
- *date1* et *date2* sont les deux dates que vous souhaitez utiliser dans le calcul
- *firstdayofweek* et *firstweekofyear* sont facultatifs. Reportez-vous à la fonction `DatePart()` pour obtenir des explications

Exemples

```
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()` renvoie une `Date` à laquelle une date ou un intervalle de temps spécifié a été ajouté.

Syntaxe

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

- *intervalle* peut être l'un des intervalles définis dans la fonction `DatePart()`
- *nombre* Expression numérique correspondant au nombre d'intervalles à ajouter. Cela peut être positif (pour obtenir des dates dans le futur) ou négatif (pour obtenir des dates dans le passé).
- *date* est une `Date` ou un littéral représentant la date à laquelle l'intervalle est ajouté

Exemples :

```
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
```

Conversion et création

CDate ()

`CDate()` convertit quelque chose de n'importe quel type de données en un type de données `Date`

```
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
```

Notez que VBA a également faiblement typé `CVDate()` qui fonctionne de la même manière que le `CDate()` autre fonction que de retourner une date tapés `Variant` au lieu d'un fortement typé `Date` . La version de `CDate()` doit être préférée lors du passage à un paramètre `Date` ou à une variable `Date` , et la version `CVDate()` doit être préférée lors du passage à un paramètre `Variant` ou lors de l'attribution à une variable `Variant` . Cela évite le casting de type implicite.

DateSerial ()

`DateSerial()` est utilisée pour créer une date. Il renvoie une `Date` pour une année, un mois et un jour spécifiés.

Syntaxe:

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

Les arguments année, mois et jour étant valides Entiers (Année de 100 à 9999, Mois de 1 à 12, Jour de 1 à 31).

Exemples

```
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
```

Notez que `DateSerial()` acceptera les dates "invalides" et en calculera une date valide. Cela peut être utilisé de manière créative pour de bon:

Exemple positif

```
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
```

Cependant, il est plus probable que vous tentiez de créer une date à partir d'une entrée utilisateur non validée:

Exemple négatif

```
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
```

Lire Date Manipulation en ligne: https://riptutorial.com/fr/vba/topic/4452/date-manipulation

Chapitre 19: Déclaration des variables

Examples

Déclaration implicite et explicite

Si un module de code ne contient pas `Option Explicit` en haut du module, le compilateur créera automatiquement (c'est-à-dire "implicitement") des variables lorsque vous les utiliserez. Ils utiliseront par défaut le type de variable `Variant` .

```
Public Sub ExampleDeclaration()

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

End Sub
```

Dans le code ci-dessus, si `Option Explicit` est spécifié, le code sera interrompu car il manque les instructions `Dim` requises pour `someVariable` et `someOtherVariable` .

```
Option Explicit

Public Sub ExampleDeclaration()

    Dim someVariable As Long
    someVariable = 10

    Dim someOtherVariable As String
    someOtherVariable = "Hello World"

End Sub
```

Il est recommandé d'utiliser Option Explicit dans les modules de code pour vous assurer que vous déclarez toutes les variables.

Voir Meilleures pratiques de VBA pour définir cette option par défaut.

Les variables

Portée

Une variable peut être déclarée (au niveau de visibilité croissant):

- Au niveau de la procédure, en utilisant le mot-clé `Dim` dans n'importe quelle procédure; une *variable locale* .
- Au niveau du module, en utilisant le mot-clé `Private` dans tout type de module; un *domaine privé* .
- Au niveau de l'instance, en utilisant le mot clé `Friend` dans tout type de module de classe; un

champ ami .
- Au niveau de l'instance, en utilisant le mot clé `Public` dans tout type de module de classe; un *domaine public* .
- Globalement, en utilisant le mot-clé `Public` dans un *module standard* ; une *variable globale* .

Les variables doivent toujours être déclarées avec la plus petite portée possible: préférer transmettre des paramètres aux procédures plutôt que de déclarer des variables globales.

Voir Modificateurs d'accès pour plus d'informations.

Variables locales

Utilisez le mot-clé `Dim` pour déclarer une *variable locale* :

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

La partie `[As Type]` de la syntaxe de déclaration est facultative. Une fois spécifié, il définit le type de données de la variable, qui détermine la quantité de mémoire allouée à cette variable. Cela déclare une variable `String` :

```
Dim identifierName As String
```

Lorsqu'un type n'est pas spécifié, le type est implicitement `Variant` :

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

La syntaxe VBA prend également en charge la déclaration de plusieurs variables dans une seule instruction:

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

Notez que le type `[As Type]` doit être spécifié pour chaque variable (autre que les variantes). C'est un piège relativement courant:

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

Variables statiques

Les variables locales peuvent également être `Static` . Dans VBA, le mot-clé `Static` est utilisé pour créer une variable "rappelant" la valeur obtenue, la dernière fois qu'une procédure a été appelée:

```
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
```

Ici, la collection de `values` est déclarée en tant que local `Static` ; parce que c'est une *variable objet* , elle est initialisée à `Nothing` . La condition qui suit la déclaration vérifie si la référence à l'objet a été `Set` avant - si c'est la première fois que la procédure est exécutée, la collection est initialisée. `DoSomethingElse` peut ajouter ou supprimer des éléments, et ils seront toujours dans la collection la prochaine fois que `DoSomething` est appelé.

Alternative

> Le mot-clé `Static` de VBA peut facilement être mal compris, en *particulier* par les programmeurs expérimentés qui travaillent généralement dans d'autres langues. Dans de nombreux langages, `static` est utilisé pour faire qu'un membre de classe (field, property, method, ...) appartienne au *type* plutôt qu'à l' *instance* . Le code en contexte `static` ne peut pas référencer le code dans le contexte de l' *instance* . Le mot-clé VBA `Static` signifie quelque chose de très différent.

Souvent, un local `Static` pourrait tout aussi bien être implémenté en tant que variable `Private` niveau module (module), mais cela remet en cause le principe selon lequel une variable doit être déclarée avec la plus petite portée possible; Faites confiance à vos instincts, utilisez celui que vous préférez - les deux fonctionneront ... mais l'utilisation de `Static` sans comprendre ce qu'il fait pourrait conduire à des bogues intéressants.

Dim vs privé

Le mot-clé `Dim` est légal au niveau des procédures et des modules. son utilisation au niveau du module équivaut à utiliser le mot clé `Private` :

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

Le mot clé `Private` est uniquement légal au niveau du module. Cela invite à réserver `Dim` pour les variables locales et à déclarer les variables de module avec `Private` , en particulier avec le mot-clé `Public` contrastant qui devrait être utilisé pour déclarer un membre public. Vous pouvez également utiliser `Dim` *partout* - ce qui compte, c'est la *cohérence* :

"Champs privés"

- **NE** PAS utiliser `Private` de déclarer une variable au niveau du module.
- **NE** PAS utiliser `Dim` pour déclarer une variable locale.
- **N'utilisez pas** `Dim` pour déclarer une variable de niveau module.

"Dim partout"

- **NE** PAS utiliser `Dim` pour déclarer quoi que ce soit privé / local.

- **N'utilisez PAS** `Private` pour déclarer une variable de niveau module.
- **ÉVITER de** déclarer `Public` champs `Public` . *

* En général, il faut éviter de déclarer `Public` champs `Public` ou `Global` toute façon.

Des champs

Une variable déclarée au niveau du module, dans la *section déclarations* en haut du corps du module, est un *champ* . Un champ `Public` déclaré dans un *module standard* est une *variable globale* :

```
Public PublicField As Long
```

Une variable avec une portée globale est accessible de n'importe où, y compris d'autres projets VBA qui référenceraient le projet dans lequel elle a été déclarée.

Pour rendre une variable globale / publique, mais uniquement visible depuis le projet, utilisez le modificateur `Friend` :

```
Friend FriendField As Long
```

Cela est particulièrement utile dans les compléments, où l'intention est que d'autres projets VBA référencent le projet de complément et puissent consommer l'API publique.

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

Les champs d'amis ne sont pas disponibles dans les modules standard.

Champs d'instance

Une variable déclarée au niveau du module, dans la *section déclarations* en haut du corps d'un module de classe (y compris `ThisWorkbook` , `ThisDocument` , `Worksheet` , `UserForm` et *les modules de classe*), est un *champ instance* : il existe seulement tant qu'il y a une *instance* de la classe autour.

```
'> 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
```

Champs d'encapsulation

Les données d'instance sont souvent conservées en mode `Private` et *copiées sous forme de* doublure. Un champ privé peut être exposé à l'aide d'une procédure `Property` . Pour exposer publiquement une variable privée sans donner d'accès en écriture à l'appelant, un module de classe (ou un module standard) implémente un membre `Property Get` :

```
Option Explicit
Private encapsulated As Long

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

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

La classe elle-même peut modifier la valeur encapsulée, mais le code appelant peut uniquement accéder aux membres `Public` (et aux membres `Friend` , si l'appelant est dans le même projet).

Permettre à l'appelant de modifier:

- Une **valeur** encapsulée, un module expose un membre `Property Let` .
- Une **référence d'objet** encapsulé, un module expose un membre de `Property Set` .

Constantes (Const)

Si vous avez une valeur qui ne change jamais dans votre application, vous pouvez définir une constante nommée et l'utiliser à la place d'une valeur littérale.

Vous pouvez utiliser Const uniquement au niveau du module ou de la procédure. Cela signifie que le contexte de déclaration d'une variable doit être une classe, une structure, un module, une procédure ou un bloc, et ne peut pas être un fichier source, un espace de noms ou une 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
```

Bien qu'il puisse être considéré comme une bonne pratique de spécifier des types de constantes, cela n'est pas obligatoire. Ne pas spécifier le type entraînera toujours le type correct:

```
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
```

Notez que ceci est spécifique aux constantes et que contrairement aux variables ne spécifiant pas le type, cela donne un type de variante.

Bien qu'il soit possible de déclarer explicitement une constante en tant que chaîne, il n'est pas possible de déclarer une constante en tant que chaîne en utilisant une syntaxe de chaîne à largeur fixe

```
'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"
```

Modificateurs d'accès

L'instruction `Dim` doit être réservée aux variables locales. Au niveau du module, préférez les modificateurs d'accès explicite:

- `Private` pour les champs privés, auxquels on ne peut accéder que dans le module dans lequel ils ont été déclarés.
- `Public` pour les champs publics et les variables globales, accessibles par n'importe quel code d'appel.
- `Friend` pour les variables publiques dans le projet, mais inaccessible aux autres projets VBA de référence (pertinent pour les compléments)
- `Global` peut également être utilisé pour `Public` champs `Public` dans les modules standard, mais est illégal dans les modules de classe et est de toute façon obsolète - préférez plutôt le modificateur `Public` . Ce modificateur n'est pas légal pour les procédures non plus.

Les modificateurs d'accès sont applicables aux variables et aux procédures.

```
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
```

Module privé d'option

Les `Sub` procédures publiques sans paramètre dans les modules standard sont exposées en tant que macros et peuvent être associées à des contrôles et à des raccourcis clavier dans le document hôte.

Inversement, les procédures de `Function` publique dans les modules standard sont exposées en tant que fonctions définies par l'utilisateur dans l'application hôte.

La spécification `Option Private Module` en haut d'un module standard empêche ses membres d'être exposés en tant que macros et UDF à l'application hôte.

Type conseils

Les indicateurs de type sont **fortement** déconseillés. Ils existent et sont documentés ici pour des raisons historiques et de compatibilité descendante. Vous devez utiliser la syntaxe `As [DataType]` place.

```
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
```

Les indications de type réduisent considérablement la lisibilité du code et encouragent une notation hongroise héritée, ce qui entrave *également la* lisibilité:

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

Au lieu de cela, déclarez les variables plus proches de leur utilisation et nommez les choses pour ce qu'elles sont utilisées, et non après leur type:

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

Les indications de type peuvent également être utilisées sur des littéraux pour appliquer un type spécifique. Par défaut, un littéral numérique inférieur à 32 768 sera interprété comme un littéral `Integer` , mais avec un indice de type, vous pouvez contrôler cela:

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

Les indications de type ne sont généralement pas nécessaires sur les littéraux, car ils seraient affectés à une variable déclarée avec un type explicite ou convertis implicitement dans le type approprié lorsqu'ils sont transmis en tant que paramètres. Les conversions implicites peuvent être évitées en utilisant l'une des fonctions de conversion de type explicite:

```
'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)
```

Fonctions intégrées renvoyant des chaînes

La majorité des fonctions intégrées qui gèrent les chaînes sont disponibles en deux versions: une version faiblement typée qui renvoie un `Variant` et une version fortement typée (se terminant par `$`) qui renvoie une `String` . A moins que vous n'attribuez la valeur de retour à un `Variant` , vous devez préférer la version qui renvoie une `String` - sinon il y a une conversion implicite de la valeur de retour.

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

Ces fonctions sont:

- 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 $

Notez que ce sont des *alias de* fonctions, pas des *indications de type* . La fonction `Left` correspond à la fonction `B_Var_Left` masquée, tandis que la version `Left$` correspond à la fonction `B_Str_Left` masquée.

Dans les premières versions de VBA, le signe `$` n'est pas un caractère autorisé et le nom de la fonction doit être placé entre crochets. Dans Word Basic, il y avait beaucoup plus de fonctions qui renvoyaient des chaînes se terminant par $.

Déclaration de chaînes de longueur fixe

En VBA, les chaînes peuvent être déclarées avec une longueur spécifique; ils sont automatiquement remplis ou tronqués pour conserver cette longueur comme déclaré.

```
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
```

Quand utiliser une variable statique

Une variable statique déclarée localement n'est pas détruite et ne perd pas sa valeur lorsque vous quittez la procédure Sub. Les appels suivants à la procédure ne requièrent pas de réinitialisation ou d'affectation, bien que vous souhaitiez "mettre à zéro" une ou plusieurs valeurs mémorisées.

Celles-ci sont particulièrement utiles lors de la liaison tardive d'un objet dans un sous-ensemble appelé à plusieurs reprises.

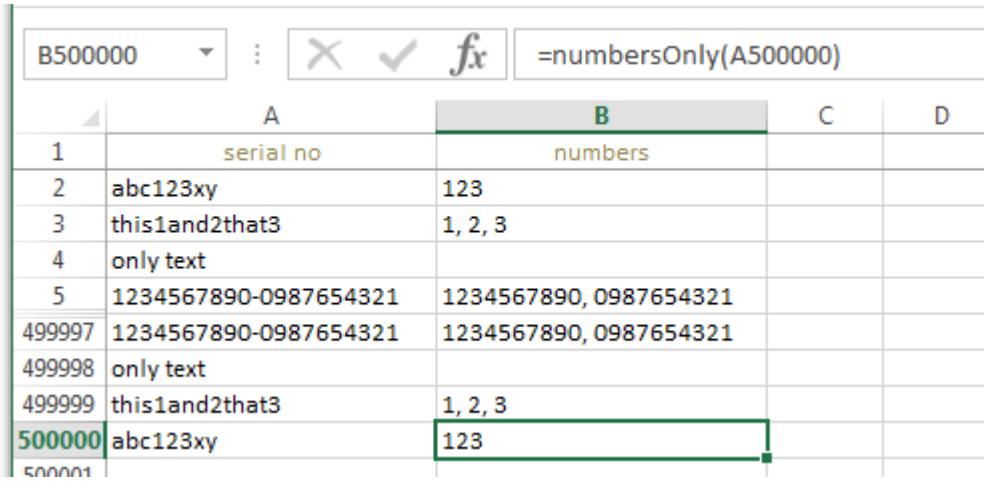
Snippet 1: réutilise un objet Scripting.Dictionary sur plusieurs feuilles de calcul

```
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
```

Extrait de code 2: créez un UDF de feuille de calcul qui lie l'objet VBScript.RegExp en retard

```
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
```



Exemple de fichier UDF avec objet statique rempli par un demi-million de lignes

* Temps écoulés pour remplir des lignes 500K avec UDF:
- avec **Dim rgx As Object** : 148,74 secondes
- avec **rgx statique comme objet** : 26,07 secondes

* Ceux-ci doivent être pris en compte pour la comparaison relative uniquement. Vos propres résultats varieront selon la complexité et
étendue des opérations effectuées.

N'oubliez pas qu'un fichier UDF n'est pas calculé une fois dans la durée de vie d'un classeur. Même un UDF non volatile sera recalculé chaque fois que les valeurs comprises dans la ou les plages auxquelles il fait référence sont susceptibles d'être modifiées. Chaque événement de recalcul ultérieur augmente uniquement les avantages d'une variable déclarée de manière statique.

- Une variable statique est disponible pour la durée de vie du module, et non pour la procédure ou la fonction dans laquelle elle a été déclarée et affectée.
- Les variables statiques ne peuvent être déclarées que localement.
- La variable statique contient un grand nombre des mêmes propriétés d'une variable de niveau de module privé, mais avec une portée plus restreinte.

Référence associée: Statique (Visual Basic)

Lire Déclaration des variables en ligne: https://riptutorial.com/fr/vba/topic/877/declaration-des-variables

Chapitre 20: Déclarer et assigner des chaînes

Remarques

Les chaînes sont un type de référence et sont au centre de la plupart des tâches de programmation. Du texte est assigné aux chaînes, même si le texte est numérique. Les chaînes peuvent être de longueur zéro ou de toute longueur jusqu'à 2 Go. Les versions modernes de VBA stockent les chaînes en interne à l'aide d'un tableau d'octets contenant des octets de jeu de caractères multi-octets (une alternative à Unicode).

Examples

Déclarer une chaîne constante

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

Déclarez une variable de chaîne de largeur variable

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

Déclarez et attribuez une chaîne de largeur fixe

```
'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"
```

Déclarez et attribuez un tableau de chaînes

```
'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"
```

Attribuer des caractères spécifiques dans une chaîne à l'aide de l'instruction Mid

VBA offre une fonction Mid pour *renvoyer des* sous-chaînes dans une chaîne, mais elle offre également la *déclaration* intermédiaire qui peut être utilisée pour attribuer des sous-chaînes ou des caractères individuels à une chaîne.

La fonction `Mid` apparaîtra généralement à droite d'une instruction d'affectation ou dans une condition, mais l'instruction `Mid` apparaît généralement à gauche d'une instruction d'affectation.

```
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
```

Remarque: Si vous devez affecter des *octets* individuels dans une chaîne plutôt que des *caractères* individuels dans une chaîne (voir les remarques ci-dessous concernant le jeu de caractères multi-octets), l'instruction `MidB` peut être utilisée. Dans ce cas, le deuxième argument de l'instruction `MidB` est la position basée sur 1 de l'octet où le remplacement commencera. La ligne équivalente à l'exemple ci-dessus serait `MidB(surname, 5, 2) = "y"`.

Affectation à et depuis un tableau d'octets

Les chaînes peuvent être affectées directement aux tableaux d'octets et inversement. Rappelez-vous que les chaînes sont stockées dans un jeu de caractères multi-octets (voir Remarques ci-dessous).

```
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."
```

Lire Déclarer et assigner des chaînes en ligne: https://riptutorial.com/fr/vba/topic/3446/declarer-et-assigner-des-chaines

Chapitre 21: Erreurs d'exécution VBA

Introduction

Le code qui compile peut encore présenter des erreurs lors de l'exécution. Cette rubrique répertorie les plus courantes, leurs causes et comment les éviter.

Examples

Erreur d'exécution '3': Retour sans GoSub

Code incorrect

```
Sub DoSomething()
    GoSub DoThis
DoThis:
    Debug.Print "Hi!"
    Return
End Sub
```

Pourquoi cela ne marche pas?

L'exécution entre dans la procédure `DoSomething` , passe à l'étiquette `DoThis` , imprime "Hi!" à la sortie de débogage, *retourne* à l'instruction immédiatement après l'appel `GoSub` , imprime "Hi!" encore une fois, et rencontre ensuite une déclaration de `Return` , mais il n'y a nulle part où *revenir* maintenant, car nous ne sommes pas `GoSub` ici avec une déclaration `GoSub` .

Code correct

```
Sub DoSomething()
    GoSub DoThis
    Exit Sub
DoThis:
    Debug.Print "Hi!"
    Return
End Sub
```

Pourquoi ça marche?

En introduisant une instruction `Exit Sub` *avant* le `DoThis` ligne `DoThis` , nous avons séparé la sous-routine `DoThis` du reste du corps de la procédure. La seule manière d'exécuter la sous-routine `DoThis` est le saut `GoSub` .

Autres notes

`GoSub` / `Return` est obsolète et devrait être évité en faveur des appels de procédure réels. Une procédure ne doit pas contenir de sous-routines autres que les gestionnaires d'erreurs.

Ceci est très similaire à l' erreur d'exécution '20': reprendre sans erreur ; dans les deux cas, la solution consiste à s'assurer que le *chemin d'exécution normal* ne peut pas entrer dans une sous-routine (identifiée par une étiquette de ligne) sans saut explicite (en supposant que `On Error GoTo` soit considéré comme un *saut explicite*).

Erreur d'exécution '6': débordement

code incorrect

```
Sub DoSomething()
    Dim row As Integer
    For row = 1 To 100000
        'do stuff
    Next
End Sub
```

Pourquoi cela ne marche pas?

Le type de données `Integer` est un entier signé 16 bits avec une valeur maximale de 32 767; l'affecter à quelque chose de plus grand que celui qui *débordera* le type et provoquera cette erreur.

Code correct

```
Sub DoSomething()
    Dim row As Long
    For row = 1 To 100000
        'do stuff
    Next
End Sub
```

Pourquoi ça marche?

En utilisant un entier `Long` (32 bits) à la place, on peut maintenant faire une boucle qui itère plus de 32 767 fois sans déborder le type de la variable du compteur.

Autres notes

Voir Types de données et limites pour plus d'informations.

Erreur d'exécution '9': indice hors limites

code incorrect

```
Sub DoSomething()
    Dim foo(1 To 10)
    Dim i As Long
    For i = 1 To 100
        foo(i) = i
    Next
End Sub
```

Pourquoi cela ne marche pas?

`foo` est un tableau qui contient 10 éléments. Lorsque le compteur de boucles `i` atteint la valeur 11, `foo(i)` est *hors limites* . Cette erreur se produit chaque fois qu'un tableau ou une collection est accessible avec un index qui n'existe pas dans ce tableau ou cette collection.

Code correct

```
Sub DoSomething()
    Dim foo(1 To 10)
    Dim i As Long
    For i = LBound(foo) To UBound(foo)
        foo(i) = i
    Next
End Sub
```

Pourquoi ça marche?

Utilisez les fonctions `LBound` et `UBound` pour déterminer respectivement les limites inférieure et supérieure d'un tableau.

Autres notes

Lorsque l'index est une chaîne, par exemple `ThisWorkbook.Worksheets("I don't exist")` , cette erreur signifie que le nom fourni n'existe pas dans la collection interrogée.

L'erreur réelle est spécifique à l'implémentation bien que; `Collection` déclenchera l'erreur d'exécution 5 "Appel ou argument de procédure non valide" à la place:

```
Sub RaisesRunTimeError5()
    Dim foo As New Collection
    foo.Add "foo", "foo"
    Debug.Print foo("bar")
End Sub
```

Erreur d'exécution '13': incompatibilité de type

code incorrect

```
Public Sub DoSomething()
```

```
    DoSomethingElse "42?"
End Sub

Private Sub DoSomethingElse(foo As Date)
'    Debug.Print MonthName(Month(foo))
End Sub
```

Pourquoi cela ne marche pas?

VBA essaie vraiment de convertir le `"42?"` argument dans une valeur de `Date` . En cas d'échec, l'appel de `DoSomethingElse` ne peut pas être exécuté, car VBA ne sait pas quelle date doit être transmise, ce qui provoque une *incompatibilité de type* erreur d'exécution 13, car le type de l'argument ne correspond pas au type attendu (et peut ne soit pas implicitement converti non plus).

Code correct

```
Public Sub DoSomething()
    DoSomethingElse Now
End Sub

Private Sub DoSomethingElse(foo As Date)
'    Debug.Print MonthName(Month(foo))
End Sub
```

Pourquoi ça marche?

En transmettant un argument `Date` à une procédure qui attend un paramètre `Date` , l'appel peut aboutir.

Erreur d'exécution '91': variable d'objet ou variable de bloc With non définie

code incorrect

```
Sub DoSomething()
    Dim foo As Collection
    With foo
        .Add "ABC"
        .Add "XYZ"
    End With
End Sub
```

Pourquoi cela ne marche pas?

Les variables d'objet contiennent une *référence* et les références doivent être *définies* à l'aide du mot-clé `Set` . Cette erreur se produit chaque fois qu'un appel de membre est effectué sur un objet dont la référence est `Nothing` . Dans ce cas, `foo` est une référence de `Collection` , mais elle n'est pas initialisée. La référence contient donc `Nothing` - et nous ne pouvons pas appeler `.Add` sur `Nothing` .

Code correct

```
Sub DoSomething()
    Dim foo As Collection
    Set foo = New Collection
    With foo
        .Add "ABC"
        .Add "XYZ"
    End With
End Sub
```

Pourquoi ça marche?

En attribuant à la variable d'objet une référence valide à l'aide du mot clé `Set` , les appels `.Add` réussissent.

Autres notes

Souvent, une fonction ou une propriété peut renvoyer une référence d'objet - un exemple courant est la méthode `Range.Find` d'Excel, qui renvoie un objet `Range` :

```
Dim resultRow As Long
resultRow = SomeSheet.Cells.Find("Something").Row
```

Cependant, la fonction peut très bien retourner `Nothing` (si le terme de recherche n'est pas trouvé), il est donc probable que l'appel de membre chaîné `.Row` échoue.

Avant d'appeler des membres d'objet, vérifiez que la référence est définie avec la condition `If Not xxxx Is Nothing` :

```
Dim result As Range
Set result = SomeSheet.Cells.Find("Something")

Dim resultRow As Long
If Not result Is Nothing Then resultRow = result.Row
```

Erreur d'exécution '20': reprendre sans erreur

code incorrect

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

CleanFail:
    Debug.Print Err.Number
    Resume Next
End Sub
```

Pourquoi cela ne marche pas?

Si la procédure `DoSomethingElse` une erreur, l'exécution passe à l' `CleanFail` ligne `CleanFail` , imprime le numéro d'erreur et l'instruction `Resume Next` retourne à l'instruction qui suit immédiatement la ligne où l'erreur s'est produite, à savoir `Debug.Print` instruction: le sous-programme de traitement des erreurs s'exécute sans contexte d'erreur et lorsque l'instruction `Resume Next` est atteinte, l'erreur d'exécution 20 est déclenchée car il n'y a aucun endroit où reprendre.

Code correct

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

    Exit Sub
CleanFail:
    Debug.Print Err.Number
    Resume Next
End Sub
```

Pourquoi ça marche?

En introduisant une instruction `Exit Sub` avant le `CleanFail` ligne `CleanFail` , nous avons séparé le `CleanFail` -programme de gestion des erreurs `CleanFail` du reste du corps de la procédure. Le seul moyen d'exécuter le sous-programme de traitement des erreurs est un saut d' `On Error` ; par conséquent, aucun chemin d'exécution n'atteint l'instruction `Resume` dehors d'un contexte d'erreur, ce qui évite l'erreur d'exécution 20.

Autres notes

Ceci est très similaire à l' erreur d'exécution "3": Retour sans GoSub ; dans les deux cas, la solution consiste à s'assurer que le *chemin d'exécution normal* ne peut pas entrer dans une sous-routine (identifiée par une étiquette de ligne) sans saut explicite (en supposant que `On Error GoTo` soit considéré comme un *saut explicite*).

Lire Erreurs d'exécution VBA en ligne: https://riptutorial.com/fr/vba/topic/8917/erreurs-d-execution-vba

Chapitre 22: Événements

Syntaxe

- **Module source** : `[Public] Event [identifier]([argument_list])`
- **Handler Module** : `Dim|Private|Public WithEvents [identifier] As [type]`

Remarques

- Un événement ne peut être que `Public` . Le modificateur est facultatif car les membres du module de classe (y compris les événements) sont implicitement `Public` par défaut.
- Une variable `WithEvents` peut être `Private` ou `Public` , mais pas `Friend` . Le modificateur est obligatoire car `WithEvents` n'est pas un mot clé qui déclare une variable, mais une partie de mot clé modificateur de la syntaxe de déclaration de variable. Par conséquent, le mot-clé `Dim` doit être utilisé si un modificateur d'accès n'est pas présent.

Examples

Sources et gestionnaires

Quels sont les événements?

VBA est *piloté par les événements* : le code VBA s'exécute en réponse aux événements déclenchés par l'application hôte ou par le document hôte - la compréhension des événements est fondamentale pour comprendre VBA.

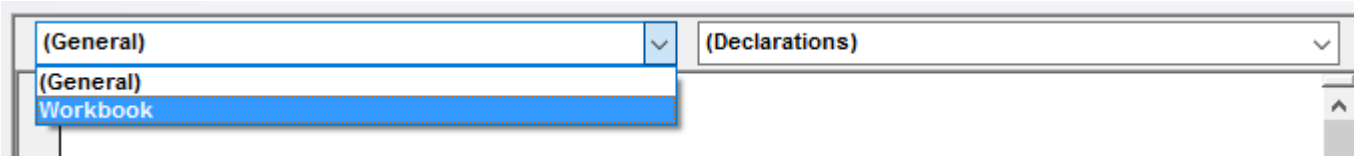
Les API exposent souvent des objets qui déclenchent un certain nombre d' *événements* en réponse à divers états. Par exemple, un objet `Excel.Application` déclenche un événement chaque fois qu'un nouveau classeur est créé, ouvert, activé ou fermé. Ou chaque fois qu'une feuille de calcul est calculée. Ou juste avant qu'un fichier soit enregistré. Ou immédiatement après. Un bouton sur un formulaire déclenche un événement `Click` lorsque l'utilisateur clique dessus, le formulaire utilisateur génère lui-même un événement juste après son activation et un autre juste avant sa fermeture.

Du point de vue de l'API, les événements sont des *points d'extension* : le code client peut choisir d'implémenter du code pour *gérer* ces événements et exécuter du code personnalisé chaque fois que ces événements sont déclenchés. - en gérant l'événement qui est déclenché lorsque la sélection change sur une feuille de calcul.

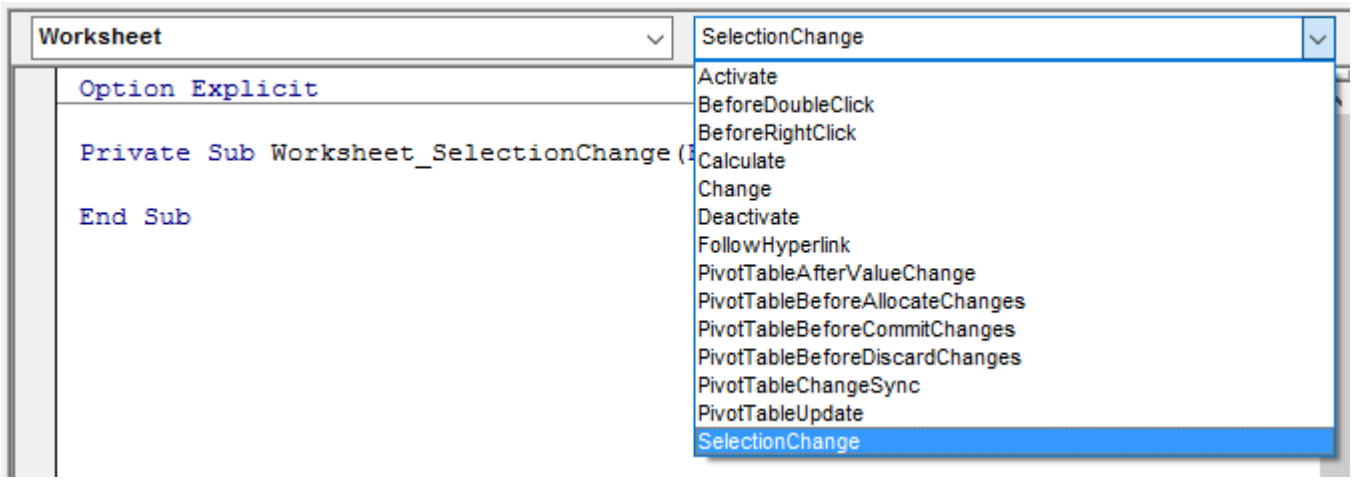
Un objet qui expose des événements est une *source d'événement* . Une méthode qui gère un événement est un *gestionnaire* .

Manieurs

Les modules de document VBA (par exemple `ThisDocument` , `ThisWorkbook` , `Sheet1` , etc.) et les modules `UserForm` sont *des modules de classe* qui *implémentent* des interfaces spéciales qui exposent un certain nombre d' *événements* . Vous pouvez parcourir ces interfaces dans la liste déroulante de gauche en haut du volet de code:



La liste déroulante de droite répertorie les membres de l'interface sélectionnés dans la liste déroulante de gauche:



Le VBE génère automatiquement un stub de gestionnaire d'événements lorsqu'un élément est sélectionné dans la liste de droite ou y navigue s'il existe.

Vous pouvez définir une variable `WithEvents` portée de `WithEvents` dans n'importe quel module:

```
Private WithEvents Foo As Workbook
Private WithEvents Bar As Worksheet
```

Chaque déclaration `WithEvents` peut être sélectionnée dans la liste déroulante de gauche. Lorsqu'un événement est sélectionné dans la liste déroulante de droite, le VBE génère un stub de gestionnaire d'événements nommé après l'objet `WithEvents` et le nom de l'événement, associé à un trait de soulignement:

```
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
```

Seuls les types qui exposent au moins un événement peuvent être utilisés avec `WithEvents` , et les déclarations `WithEvents` ne peuvent pas être `WithEvents` une référence `WithEvents` avec le mot `New` clé `New` . Ce code est illégal:

```
Private WithEvents Foo As New Workbook 'illegal
```

La référence d'objet doit être `Set` explicitement; Dans un module de classe, il est souvent `Class_Initialize` le faire dans le gestionnaire `Class_Initialize` , car la classe gère alors les événements de cet objet tant que son instance existe.

Sources

Tout module de classe (ou module de document ou formulaire utilisateur) peut être une source d'événement. Utilisez le mot-clé `Event` pour définir la *signature* de l'événement dans la *section déclarations* du module:

```
Public Event SomethingHappened(ByVal something As String)
```

La signature de l'événement détermine la façon dont l'événement est déclenché et à quoi les gestionnaires d'événements ressembleront.

Les événements ne peuvent être *déclenchés que* dans la classe dans laquelle ils sont définis - le code client ne peut les *gérer* que Les événements sont `RaiseEvent` avec le mot clé `RaiseEvent` ; les arguments de l'événement sont fournis à ce stade:

```
Public Sub DoSomething()
    RaiseEvent SomethingHappened("hello")
End Sub
```

Sans code qui gère l'événement `SomethingHappened` , l'exécution de la procédure `DoSomething` soulèvera toujours l'événement, mais rien ne se passera. En supposant que la source de l'événement est le code ci-dessus dans une classe nommée `Something` , ce code dans `ThisWorkbook` afficherait une boîte de message disant "bonjour" à chaque fois que `test.DoSomething` est appelé:

```
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
```

Utilisation de paramètres transmis par référence

Un événement peut définir un paramètre `ByRef` destiné à être renvoyé à l'appelant:

```
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
```

Si l'événement `BeforeSomething` a un gestionnaire qui définit son paramètre `cancel` sur `True` , alors lorsque l'exécution retourne du gestionnaire, `cancel` sera `True` et `AfterSomething` ne sera jamais `AfterSomething` .

```
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
```

En supposant que la référence d'objet `foo` soit assignée quelque part, lorsque `foo.DoSomething` s'exécute, une boîte de message vous demande si vous souhaitez annuler, et une seconde boîte de message indique "n'a pas annulé" uniquement lorsque `Non a` été sélectionné.

Utiliser des objets mutables

Vous pouvez également transmettre une copie d'un objet mutable `ByVal` et laisser les gestionnaires modifier les propriétés de cet objet. l'appelant peut alors lire les valeurs de propriété modifiées et agir en conséquence.

```
'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
```

Combiné au type `Variant` , il peut être utilisé pour créer des moyens non évidents de renvoyer une valeur à l'appelant:

```
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
```

Le gestionnaire ressemblerait à ceci:

```
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
```

Lire Événements en ligne: https://riptutorial.com/fr/vba/topic/5278/evenements

Chapitre 23: Formulaires utilisateur

Examples

Les meilleures pratiques

Un objet `UserForm` est un module de classe avec un concepteur et une **instance par défaut** . Vous pouvez accéder au *concepteur* en appuyant sur `Maj` + `F7` tout en affichant le *code-behind* , et vous pouvez accéder au *code-behind* en appuyant sur `F7` tout en affichant le *concepteur* .

Travaillez avec une nouvelle instance à chaque fois.

En tant que *module de classe* , un formulaire est donc un *modèle* pour un *objet* . Un formulaire pouvant contenir des états et des données, il est préférable de travailler avec une nouvelle *instance* de la classe, plutôt qu'avec une *instance* par défaut / globale:

```
With New UserForm1
    .Show vbModal
    If Not .IsCancelled Then
        '...
    End If
End With
```

Au lieu de:

```
UserForm1.Show vbModal
If Not UserForm1.IsCancelled Then
    '...
End If
```

Travailler avec l'instance par défaut peut conduire à des bogues subtils lorsque le formulaire est fermé avec le bouton rouge "X" et / ou lorsque `Unload Me` est utilisé dans le code-behind.

Implémenter la logique ailleurs.

Un formulaire ne doit concerner que la *présentation* : un bouton Le gestionnaire de `Click` qui se connecte à une base de données et exécute une requête paramétrée en fonction des entrées de l'utilisateur **fait trop de choses** .

Au lieu de cela, implémentez la *logique applicative* dans le code responsable de l'affichage du formulaire, ou même mieux, dans les modules et les procédures dédiés.

Écrivez le code de manière à ce que l'objet UserForm ne soit responsable que de savoir comment afficher et collecter des données: la provenance des données ou les événements ultérieurs ne sont pas concernés.

L'appelant ne devrait pas être dérangé par les contrôles.

Créez un *modèle* bien défini pour le formulaire avec lequel travailler, soit dans son propre module de classe dédié, soit encapsulé dans le code-behind du formulaire - exposez le *modèle* avec les procédures `Property Get` , et faites en sorte que le code client fonctionne avec ceci: la forme une *abstraction* sur les contrôles et leurs détails les plus concrets, exposant uniquement les données pertinentes au code client.

Cela signifie un code qui ressemble à ceci:

```
With New UserForm1
    .Show vbModal
    If Not .IsCancelled Then
        MsgBox .Message, vbInformation
    End If
End With
```

Au lieu de cela:

```
With New UserForm1
    .Show vbModal
    If Not .IsCancelled Then
        MsgBox .txtMessage.Text, vbInformation
    End If
End With
```

Gérez l'événement QueryClose.

Les formulaires ont généralement un bouton `Fermer` et les invites / boîtes de dialogue ont les boutons `Ok` et `Annuler` ; l'utilisateur peut fermer le formulaire en utilisant la *boîte de contrôle* du formulaire (le bouton rouge "X"), qui détruit l'instance de formulaire par défaut (une autre bonne raison de *travailler avec une nouvelle instance à chaque fois*).

```
With New UserForm1
    .Show vbModal
    If Not .IsCancelled Then 'if QueryClose isn't handled, this can raise a runtime error.
        '...
    End With
End With
```

Le moyen le plus simple de gérer l'événement `QueryClose` consiste à définir le paramètre `Cancel` sur `True` , puis à *masquer* le formulaire au lieu de le *fermer* :

```
Private Sub UserForm_QueryClose(Cancel As Integer, CloseMode As Integer)
    Cancel = True
    Me.Hide
End Sub
```

De cette façon, le bouton "X" ne détruira jamais l'instance et l'appelant peut accéder en toute sécurité à tous les membres publics.

Cachez, ne fermez pas.

Le code qui crée un objet doit être responsable de sa destruction: ce n'est pas la responsabilité du formulaire de se décharger et de se terminer.

Évitez d'utiliser `Unload Me` dans le code d'un formulaire. Appelez `Me.Hide` place, afin que le code appelant puisse toujours utiliser l'objet créé lors de la fermeture du formulaire.

Nommez les choses.

Utilisez les *propriétés* `(F4 de)` de nommer soigneusement chaque contrôle sur un formulaire. Le nom d'un contrôle est utilisé dans le code-behind, à moins que vous n'utilisiez un outil de refactoring capable de gérer cela, **renommer un contrôle va casser le code** - il est donc beaucoup plus facile de faire les choses en premier lieu. pour définir exactement laquelle des 20 zones de `TextBox12` représente.

Traditionnellement, les contrôles UserForm sont nommés avec des préfixes de style hongrois:

- `lblUserName` pour un contrôle `Label` qui indique un nom d'utilisateur.
- `txtUserName` pour un contrôle `TextBox` où l'utilisateur peut entrer un nom d'utilisateur.
- `cboUserName` pour un contrôle `ComboBox` lequel l'utilisateur peut entrer ou sélectionner un nom d'utilisateur.
- `lstUserName` pour un contrôle `ListBox` où l'utilisateur peut choisir un nom d'utilisateur.
- `btnOk` ou `cmdOk` pour un contrôle `Button` libellé "Ok".

Le problème est que lorsque par exemple l'interface utilisateur obtient redessinée et un `ComboBox` des modifications à un `ListBox` , le nom doit changer pour refléter le nouveau type de contrôle: il est préférable de contrôles de nom pour ce qu'ils représentent, plutôt qu'après leur type de contrôle - à *découpler* la code de l'interface utilisateur autant que possible.

- `UserNameLabel` pour une étiquette en lecture seule qui indique un nom d'utilisateur.
- `UserNameInput` pour un contrôle où l'utilisateur peut entrer ou choisir un nom d'utilisateur.
- `OkButton` pour un bouton de commande intitulé "Ok".

Quel que soit le style choisi, tout est meilleur que de laisser tous les contrôles à leurs noms par défaut. La cohérence dans le style de nommage est également idéale.

Gestion de la requêteClose

L'événement `QueryClose` est `QueryClose` chaque fois qu'un formulaire est sur le point d'être fermé, qu'il s'agisse d'une action utilisateur ou d'un programme. Le paramètre `CloseMode` contient une valeur enum `VbQueryClose` qui indique comment le formulaire a été fermé:

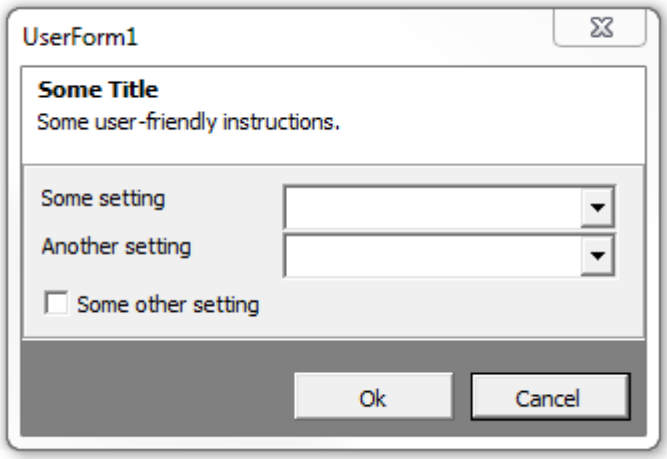
Constant	La description	Valeur
`vbFormControlMenu`	Le formulaire se ferme en réponse à l'action de l'utilisateur	0

Constant	La description	Valeur
`vbFormCode`	Le formulaire se ferme en réponse à une instruction `Unload`	1
`vbAppWindows`	La session Windows se termine	2
`vbAppTaskManager`	Le gestionnaire de tâches Windows ferme l'application hôte	3
`vbFormMDIForm`	Non pris en charge dans VBA	4

Pour une meilleure lisibilité, il est préférable d'utiliser ces constantes au lieu d'utiliser directement leur valeur.

Un objet utilisateur annulable

Étant donné un formulaire avec un bouton `Annuler`



Le code-behind du formulaire pourrait ressembler à ceci:

```
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
```

Le code d'appel peut alors afficher le formulaire et savoir s'il a été annulé:

```
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
```

La propriété `IsCancelled` renvoie `True` lorsque le bouton `Annuler` est cliqué ou lorsque l'utilisateur ferme le formulaire à l'aide de la *boîte de contrôle* .

Lire Formulaires utilisateur en ligne: https://riptutorial.com/fr/vba/topic/5351/formulaires-utilisateur

Chapitre 24: Interfaces

Introduction

Une **interface** est un moyen de définir un ensemble de comportements qu'une classe doit exécuter. La définition d'une interface est une liste de signatures de méthode (nom, paramètres et type de retour). On dit qu'une classe ayant toutes les méthodes "implémente" cette interface.

En VBA, l'utilisation d'interfaces permet au compilateur de vérifier qu'un module implémente toutes ses méthodes. Une variable ou un paramètre peut être défini en termes d'interface au lieu d'une classe spécifique.

Examples

Interface simple - Flyable

L'interface `Flyable` est un module de classe avec le code suivant:

```
Public Sub Fly()
    ' No code.
End Sub

Public Function GetAltitude() As Long
    ' No code.
End Function
```

Un module de classe, `Airplane` , utilise le mot-clé `Implements` pour indiquer au compilateur de générer une erreur à moins qu'il ne dispose de deux méthodes: un sous-objet `Flyable_Fly()` et une fonction `Flyable_GetAltitude()` qui renvoie un `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
```

Un module de deuxième classe, `Duck` , implémente également `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
```

Nous pouvons écrire une routine qui accepte toute valeur `Flyable` , sachant qu'elle répondra à une commande de `Fly` ou `GetAltitude` :

```
Public Sub FlyAndCheckAltitude(F As Flyable)
    F.Fly
    Debug.Print F.GetAltitude
End Sub
```

Comme l'interface est définie, la fenêtre contextuelle IntelliSense affiche `Fly` et `GetAltitude` pour `F`

Lorsque nous exécutons le code suivant:

```
Dim MyDuck As New Duck
Dim MyAirplane As New Airplane

FlyAndCheckAltitude MyDuck
FlyAndCheckAltitude MyAirplane
```

La sortie est la suivante:

```
Flying With Wings!
30
Flying With Jet Engines!
10000
```

Notez que même si la sous-routine est nommée `Flyable_Fly` à la fois dans `Airplane` et `Duck` , elle peut être appelée `Fly` lorsque la variable ou le paramètre est défini comme `Flyable` . Si la variable est définie spécifiquement comme un `Duck` , elle devrait être appelée `Flyable_Fly` .

Plusieurs interfaces dans une classe - Flyable et Swimable

En utilisant l'exemple `Flyable` comme point de départ, nous pouvons ajouter une seconde interface, `Swimmable` , avec le code suivant:

```
Sub Swim()
    ' No code
End Sub
```

L'objet `Duck` peut `Implement` fois voler et nager:

```
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
```

Une classe `Fish` peut également implémenter `Swimmable` :

```
Implements Swimmable

Public Sub Swimmable_Swim()
    Debug.Print "Swimming under the water"
End Sub
```

Maintenant, nous pouvons voir que l'objet `Duck` peut être transmis à un Sub en tant que `Flyable` d'une part, et `Swimmable` de l'autre:

```
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
```

La sortie de ce code est la suivante:

> Fly Check ...
>
> Voler avec des ailes!
>
> 30
>
> Voler avec des moteurs à réaction!
>
> 10000

Swim Check ...

Flottant sur l'eau

Nager sous l'eau

Lire Interfaces en ligne: https://riptutorial.com/fr/vba/topic/8784/interfaces

Chapitre 25: La gestion des erreurs

Examples

Éviter les conditions d'erreur

Lorsqu'une erreur d'exécution se produit, un code correct doit le gérer. La meilleure stratégie de gestion des erreurs consiste à écrire du code qui vérifie les conditions d'erreur et évite simplement d'exécuter du code entraînant une erreur d'exécution.

Un élément clé dans la réduction des erreurs d'exécution est la rédaction de petites procédures qui *font une chose* . Moins il y a de raisons pour lesquelles les procédures doivent échouer, plus le code dans son ensemble est facile à déboguer.

Éviter l'erreur d'exécution 91 - La variable Object ou With block n'est pas définie:

Cette erreur sera déclenchée lorsqu'un objet est utilisé avant que sa référence soit affectée. On peut avoir une procédure qui reçoit un paramètre d'objet:

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

Si la référence n'est pas affectée à une `target` , le code ci-dessus générera une erreur facilement évitée en vérifiant si l'objet contient une référence d'objet réelle:

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

Si la `target` n'est pas attribuée à une référence, la référence non attribuée n'est jamais utilisée et aucune erreur ne se produit.

Cette méthode de sortie anticipée d'une procédure lorsqu'un ou plusieurs paramètres n'est pas valide s'appelle une *clause de garde* .

Éviter l'erreur d'exécution 9 - Indice en dehors des limites:

Cette erreur est déclenchée lorsqu'un tableau est accessible en dehors de ses limites.

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

Étant donné un index supérieur au nombre de feuilles de calcul dans `ActiveWorkbook` , le code ci-

dessus génère une erreur d'exécution. Une simple clause de garde peut éviter que:

```
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
```

La plupart des erreurs d'exécution peuvent être évitées en vérifiant soigneusement les valeurs que nous utilisons *avant de les* utiliser, et en les ramenant en conséquence à l'aide de simples clauses `If` statement - in guard qui ne font aucune hypothèse et valident les paramètres d'une procédure. corps de procédures plus grandes.

En cas d'erreur

Même avec les *clauses de garde* , on ne peut pas *toujours* rendre compte de manière réaliste de toutes les conditions d'erreur possibles pouvant être soulevées dans le corps d'une procédure. L'instruction `On Error GoTo` indique à VBA de passer à un *libellé de ligne* et de passer en "mode de traitement des erreurs" chaque fois qu'une erreur inattendue se produit au moment de l'exécution. Après avoir traité une erreur, le code peut *reprendre* une exécution "normale" à l'aide du mot-clé `Resume` .

Les étiquettes de ligne indiquent des *sous - routines* : comme les sous-programmes proviennent du code BASIC hérité et utilisent des sauts `GoTo` et `GoSub` et des instructions `Return` pour revenir à la routine "principale", il est assez facile d'écrire du *code spaghetti* difficile à suivre . Pour cette raison, il est préférable que:

- une procédure a **un et un seul** sous-programme de gestion des erreurs
- le sous-programme de gestion des erreurs **ne s'exécute jamais que dans un état d'erreur**

Cela signifie qu'une procédure qui gère ses erreurs doit être structurée comme suit:

```
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
```

Stratégies de gestion des erreurs

Parfois, vous voulez gérer des erreurs différentes avec des actions différentes. Dans ce cas, vous

inspecterez l'objet `Err` global, qui contiendra des informations sur l'erreur qui a été générée - et agissez en conséquence:

```
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
```

En règle générale, envisagez d'activer la gestion des erreurs pour l'ensemble de la sous-routine ou de la fonction, et gérez toutes les erreurs pouvant survenir dans sa portée. Si vous ne devez gérer que les erreurs dans la petite section du code - activez et désactivez la gestion des erreurs au même niveau:

```
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
```

Numéros de ligne

VBA prend en charge les numéros de ligne de style ancien (par exemple QBASIC). La propriété masquée `Erl` peut être utilisée pour identifier le numéro de ligne qui a généré la dernière erreur. Si vous n'utilisez pas de numéros de ligne, `Erl` ne retournera jamais que 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
```

Si vous utilisez les numéros de ligne, mais pas toujours, alors `Erl` retournera *le dernier numéro de* la *ligne avant* que *l'instruction qui* a *soulevé l'erreur.*

```
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
```

Gardez à l'esprit `Erl` ne dispose que d'une précision `Integer` et débordera silencieusement. Cela signifie que les numéros de ligne en dehors de la plage entière donneront des résultats incorrects:

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

Le numéro de ligne n'est pas tout à fait aussi pertinent que l'affirmation qui a provoqué l'erreur, et les lignes de numérotation deviennent rapidement fastidieuses et peu conviviales.

Reprendre le mot clé

Un sous-programme de gestion des erreurs sera:

- exécuter jusqu'à la fin de la procédure, auquel cas l'exécution reprend dans la procédure d'appel.
- ou, utilisez le mot-clé `Resume` pour *reprendre l'* exécution dans la même procédure.

Le mot-clé `Resume` ne doit être utilisé que dans un sous-programme de gestion des erreurs, car si VBA rencontre `Resume` sans être dans un état d'erreur, l'erreur d'exécution 20 "Reprendre sans erreur" est générée.

Un sous-programme de gestion des erreurs peut utiliser le mot-clé `Resume` plusieurs manières:

- `Resume` utilisé seul, l'exécution continue **sur l'instruction qui a provoqué l'erreur** . Si l'erreur n'est pas *réellement* gérée avant cela, la même erreur sera à nouveau déclenchée et l'exécution pourra entrer dans une boucle infinie.
- `Resume Next` continue l'exécution **sur l'instruction immédiatement après** l'instruction qui a provoqué l'erreur. Si l'erreur n'est pas *réellement* gérée avant cela, l'exécution peut continuer avec des données potentiellement invalides, ce qui peut entraîner des erreurs logiques et un comportement inattendu.

- `Resume [line label]` continue l'exécution **à l'étiquette de ligne spécifiée** (ou au numéro de ligne, si vous utilisez des numéros de ligne hérités du style). Cela permet généralement d'exécuter du code de nettoyage avant de quitter la procédure proprement, par exemple en veillant à ce qu'une connexion à la base de données soit fermée avant de retourner à l'appelant.

On Error Resume Next

L'instruction `On Error` elle-même peut utiliser le mot clé `Resume` pour indiquer à l'environnement d'exécution VBA d' **ignorer** efficacement **toutes les erreurs** .

> *Si l'erreur n'est pas **réellement gérée** avant cela, l'exécution peut continuer avec des données potentiellement invalides, ce qui peut entraîner **des erreurs logiques et un comportement inattendu** .*

L'emphase ci-dessus ne peut pas être assez soulignée. ***On Error Resume Next* ignore effectivement toutes les erreurs et les place sous le tapis** . Un programme qui explose avec une erreur d'exécution à cause d'une entrée invalide est un meilleur programme que celui qui continue à fonctionner avec des données inconnues / involontaires - que ce soit uniquement parce que le bogue est beaucoup plus facilement identifiable. `On Error Resume Next` peut facilement **masquer les bogues** .

Le `On Error` déclaration est scope procédure - c'est pourquoi il devrait *normalement* y avoir qu'un **seul,** unique comme `On Error` instruction dans une procédure donnée.

Cependant, *parfois,* une condition d'erreur ne peut pas être complètement évitée, et passer à un sous-programme de gestion des erreurs uniquement pour `Resume Next` ne semble pas correct. Dans ce cas spécifique, l'instruction known-to-event-fail peut être **encapsulée** entre deux instructions `On Error` :

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

L'instruction `On Error GoTo 0` réinitialise la gestion des erreurs dans la procédure en cours, de sorte que toute instruction supplémentaire provoquant une erreur d'exécution *ne serait pas gérée dans cette procédure* et passait la pile d'appels jusqu'à ce qu'elle soit interceptée par un gestionnaire d'erreur actif. S'il n'y a pas de gestionnaire d'erreurs actif dans la pile d'appels, il sera traité comme une exception non gérée.

```
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
```

Erreurs personnalisées

Souvent, lors de l'écriture d'une classe spécialisée, vous souhaiterez qu'il génère des erreurs spécifiques et que vous souhaitiez un moyen propre pour l'utilisateur / le code d'appel de gérer ces erreurs personnalisées. Pour ce faire, vous devez définir un type `Enum` dédié:

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

L'utilisation de la `vbObjectError` intégrée `vbObjectError` garantit que les codes d'erreur personnalisés ne chevauchent pas les codes d'erreur réservés / existants. Seule la première valeur d'énumération doit être explicitement spécifiée, car la valeur sous-jacente de chaque membre `Enum` est supérieure de `1` au membre précédent. La valeur sous-jacente de `Err_BarNotInitialized` est donc implicitement `vbObjectError + 1025` .

Augmenter vos propres erreurs d'exécution

Une erreur d'exécution peut être `Err.Raise` l'aide de l'instruction `Err.Raise` , afin que l'erreur `Err_FooWasNotBarred` personnalisée puisse être `Err_FooWasNotBarred` comme suit:

```
Err.Raise Err_FooWasNotBarred
```

La méthode `Err.Raise` peut également prendre les paramètres `Description` et `Source` personnalisés. Pour cette raison, il est `Err.Raise` définir également des constantes pour contenir la description de chaque erreur personnalisée:

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

Et ensuite, créez une méthode privée dédiée pour générer chaque erreur:

```
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
```

L'implémentation de la classe peut alors simplement appeler ces procédures spécialisées pour générer l'erreur:

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

Le code client peut alors gérer `Err_BarNotInitialized` comme toute autre erreur, dans son propre sous-programme de gestion des erreurs.

Remarque: le mot clé `Error` hérité peut également être utilisé à la place de `Err.Raise` , mais il est obsolète / obsolète.

Lire La gestion des erreurs en ligne: https://riptutorial.com/fr/vba/topic/3211/la-gestion-des-erreurs

Chapitre 26: Lecture de 2 Go + fichiers en binaire dans VBA et File Hashes

Introduction

Il existe un moyen simple de lire les fichiers en binaire dans VBA, mais il est limité à 2 Go (2 147 483 647 octets - max de type de données long). À mesure que la technologie évolue, cette limite de 2 Go est facilement atteinte. Par exemple, une image ISO du système d'exploitation installe le disque DVD. Microsoft fournit un moyen de surmonter cela via une API Windows de bas niveau et voici une sauvegarde de celle-ci.

Démontrez également (Lire la partie) le calcul des hachages de fichiers sans programme externe tel que `fciv.exe` de Microsoft.

Remarques

METHODES POUR LA CLASSE PAR MICROSOFT

Nom de la méthode	La description
Est ouvert	Renvoie un booléen pour indiquer si le fichier est ouvert.
OpenFile (*sFileNam* e As String)	Ouvre le fichier spécifié par l'argument sFileName.
Fermer le fichier	Ferme le fichier actuellement ouvert.
ReadBytes (*ByteCount* As Long)	Lit les octets ByteCount et les retourne dans un tableau d'octets Variant et déplace le pointeur.
WriteBytes (*DataBytes ()* As Byte)	Ecrit le contenu du tableau d'octets dans la position actuelle du fichier et déplace le pointeur.
Affleurer	Force Windows à vider le cache en écriture.
SeekAbsolute (*HighPos* As Long, *LowPos* As Long)	Déplace le pointeur de fichier vers la position désignée depuis le début du fichier. Bien que VBA traite les DWORDS comme des valeurs signées, l'API les considère comme non signées. Faites en sorte que l'argument de poids fort non nul dépasse 4 Go. Le faible DWORD sera négatif pour les valeurs entre 2 Go et 4 Go.
SeekRelative (*Offset* As Long)	Déplace le pointeur de fichier jusqu'à +/- 2 Go à partir de l'emplacement actuel. Vous pouvez réécrire cette méthode pour permettre des décalages supérieurs à 2 Go en convertissant un

Nom de la méthode	La description
	décalage signé de 64 bits en deux valeurs de 32 bits.

PROPRIÉTÉS DE LA CLASSE PAR MICROSOFT

Propriété	La description
FileHandle	Le descripteur de fichier du fichier actuellement ouvert. Ceci n'est pas compatible avec les descripteurs de fichiers VBA.
Nom de fichier	Le nom du fichier actuellement ouvert.
AutoFlush	Définit / indique si WriteBytes appellera automatiquement la méthode Flush.

Module normal

Fonction	Remarques
GetFileHash (*sFile* As String, *uBlockSize* As Double, *sHashType* As String)	Il suffit de lancer le chemin complet à hacher, la taille de bloc à utiliser (nombre d'octets) et le type de hachage à utiliser - une des constantes privées: **HashTypeMD5** , **HashTypeSHA1** , **HashTypeSHA256** , **HashTypeSHA384** , **HashTypeSHA512** . Ceci a été conçu pour être aussi générique que possible.

Vous devez **annuler** / commenter le **uFileSize As Double en** conséquence. J'ai testé MD5 et SHA1.

Examples

Cela doit être dans un module de classe, les exemples plus tard appelés "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
```

Code de calcul du hachage de fichier dans un module standard

```
Private Const HashTypeMD5 As String = "MD5" ' https://msdn.microsoft.com/en-
us/library/system.security.cryptography.md5cryptoserviceprovider(v=vs.110).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=vs.110).aspx
Private Const HashTypeSHA384 As String = "SHA384" ' https://msdn.microsoft.com/en-
us/library/system.security.cryptography.sha384cryptoserviceprovider(v=vs.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
```

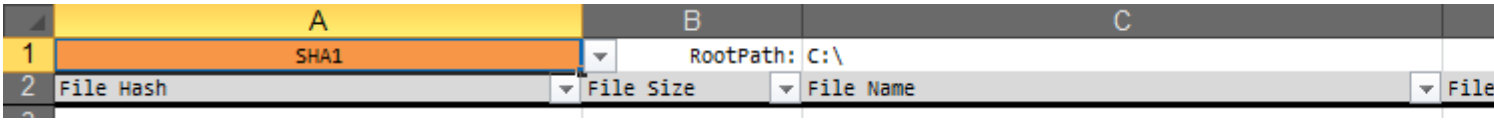
La sortie est assez intéressante, mes fichiers de test indiquent que **`BLOCKSIZE = 131071` (2 ^ 17-1)** donne les meilleures performances globales avec 32bit Office 2010 sur Windows 7 x64, puis *2 ^ 16-1 (65535)* . Note `2^27-1` cède la *mémoire* .

Taille du fichier (octets)	Nom de fichier
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

Calculer tous les fichiers Hash depuis un dossier racine

Une autre variante du code ci-dessus vous donne plus de performances lorsque vous souhaitez obtenir des codes de hachage de tous les fichiers à partir d'un dossier racine, y compris tous les sous-dossiers.

Exemple de feuille de travail:



Code

```
Option Explicit

Private Const HashTypeMD5 As String = "MD5" ' https://msdn.microsoft.com/en-
us/library/system.security.cryptography.md5cryptoserviceprovider(v=vs.110).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=vs.110).aspx
Private Const HashTypeSHA384 As String = "SHA384" ' https://msdn.microsoft.com/en-
us/library/system.security.cryptography.sha384cryptoserviceprovider(v=vs.110).aspx
Private Const HashTypeSHA512 As String = "SHA512" ' https://msdn.microsoft.com/en-
us/library/system.security.cryptography.sha512cryptoserviceprovider(v=vs.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
```

Lire Lecture de 2 Go + fichiers en binaire dans VBA et File Hashes en ligne: https://riptutorial.com/fr/vba/topic/8786/lecture-de-2-go-plus-fichiers-en-binaire-dans-vba-et-file-hashes

Chapitre 27: Les attributs

Syntaxe

- Attribut VB_Name = "ClassOrModuleName"
- Attribut VB_GlobalNameSpace = Faux 'Ignoré
- Attribut VB_Creatable = Faux 'Ignoré
- Attribut VB_PredeclaredId = {True | Faux}
- Attribut VB_Exposed = {True | Faux}
- Attribut variableName.VB_VarUserMemId = 0 'Zéro indique qu'il s'agit du membre par défaut de la classe.
- Attribut variableName.VB_VarDescription = "une chaîne" 'Ajoute le texte aux informations de l'explorateur d'objets pour cette variable.
- Attribut procName.VB_Description = "some string" 'Ajoute le texte aux informations de l'explorateur d'objets pour la procédure.
- Attribut procName.VB_UserMemId = {0 | -4}
    - '0: fait de la fonction le membre par défaut de la classe.
    - '-4: Spécifie que la fonction retourne un énumérateur.

Examples

VB_Name

VB_Name spécifie le nom de la classe ou du module.

```
Attribute VB_Name = "Class1"
```

Une nouvelle instance de cette classe serait créée avec

```
Dim myClass As Class1
myClass = new Class1
```

VB_GlobalNameSpace

Dans VBA, cet attribut est ignoré. Il n'a pas été transféré de VB6.

Dans VB6, il crée une instance globale par défaut de la classe (un "raccourci") pour que les membres de la classe puissent être accédés sans utiliser le nom de la classe. Par exemple, `DateTime` (comme dans `DateTime.Now`) fait en réalité partie de la classe `VBA.Conversion` .

```
Debug.Print VBA.Conversion.DateTime.Now
Debug.Print DateTime.Now
```

VB_Createable

Cet attribut est ignoré. Il n'a pas été transféré de VB6.

Dans VB6, il était utilisé en combinaison avec l'attribut `VB_Exposed` pour contrôler l'accessibilité des classes en dehors du projet en cours.

```
VB_Exposed=True
VB_Creatable=True
```

Vous obtiendrez une `Public Class` accessible à partir d'autres projets, mais cette fonctionnalité n'existe pas dans VBA.

VB_PredeclaredId

Crée une instance par défaut globale d'une classe. L'instance par défaut est accessible via le nom de la classe.

Déclaration

```
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
```

Appel

```
Debug.Print Class1.GiveMeATwo
```

À certains égards, cela simule le comportement des classes statiques dans d'autres langages, mais contrairement à d'autres langages, vous pouvez toujours créer une instance de la classe.

```
Dim cls As Class1
Set cls = New Class1
Debug.Print cls.GiveMeATwo
```

VB_Exposed

Contrôle les caractéristiques d'instanciation d'une classe.

```
Attribute VB_Exposed = False
```

Rend la classe `Private` . Il est impossible d'y accéder en dehors du projet en cours.

```
Attribute VB_Exposed = True
```

Expose la classe en `Public` , en dehors du projet. Cependant, `VB_Createable` étant ignoré dans VBA, les instances de la classe ne peuvent pas être créées directement. Cela équivaut à la classe VB.Net suivante.

```
Public Class Foo
    Friend Sub New()
    End Sub
End Class
```

Pour obtenir une instance en dehors du projet, vous devez exposer une fabrique pour créer des instances. Pour ce faire, vous pouvez `Public` module `Public` standard.

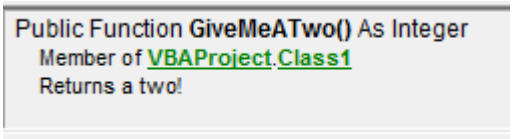
```
Public Function CreateFoo() As Foo
    CreateFoo = New Foo
End Function
```

Les modules publics étant accessibles à partir d'autres projets, cela nous permet de créer de nouvelles instances de nos classes `Public - Not Createable` .

VB_Description

Ajoute une description textuelle à un membre de la classe ou du module qui devient visible dans l'Explorateur d'objets. Idéalement, tous les membres publics d'une interface / API publique devraient avoir une description.

```
Public Function GiveMeATwo() As Integer
    Attribute GiveMeATwo.VB_Description = "Returns a two!"
    GiveMeATwo = 2
End Property
```



Remarque: tous les membres accesseurs d'une propriété (`Get` , `Let` , `Set`) utilisent la même description.

VB_ [Var] UserMemId

`VB_VarUserMemId` (pour les variables de portée de module) et `VB_UserMemId` (pour les procédures) sont utilisés dans VBA principalement pour deux choses.

Spécifier le membre par défaut d'une classe

Une classe de `List` qui encapsulerait une `Collection` voudrait avoir une propriété `Item` , donc le code client peut le faire:

```
For i = 1 To myList.Count 'VBA Collection Objects are 1-based
    Debug.Print myList.Item(i)
Next
```

Mais avec un attribut `VB_UserMemId` défini sur 0 dans la propriété `Item` , le code client peut le faire:

```
For i = 1 To myList.Count 'VBA Collection Objects are 1-based
    Debug.Print myList(i)
Next
```

Un seul membre peut légalement avoir `VB_UserMemId = 0` dans une classe donnée. Pour les propriétés, spécifiez l'attribut dans l'accesseur `Get` :

```
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
```

Rendre une classe itérable avec une construction de boucle `For Each`

Avec la valeur magique `-4` , l'attribut `VB_UserMemId` indique à VBA que ce membre génère un énumérateur - ce qui permet au code client de le faire:

```
Dim item As Variant
For Each item In myList
    Debug.Print item
Next
```

La meilleure façon de mettre en œuvre cette méthode est en appelant le caché `[_NewEnum]` propriété getter sur une interne / encapsulé `Collection` ; l'identifiant doit être placé entre crochets à cause du trait de soulignement principal qui en fait un identifiant VBA illégal:

```
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
```

Lire Les attributs en ligne: https://riptutorial.com/fr/vba/topic/5321/les-attributs

Chapitre 28: Les opérateurs

Remarques

Les opérateurs sont évalués dans l'ordre suivant:

- Opérateurs mathématiques
- Opérateurs binaires
- Opérateurs de concaténation
- Opérateurs de comparaison
- Opérateurs logiques

Les opérateurs avec une priorité correspondante sont évalués de gauche à droite. L'ordre par défaut peut être remplacé en utilisant des parenthèses `(` et `)` pour grouper des expressions.

Examples

Opérateurs Mathématiques

En ordre de priorité:

Jeton	prénom	La description
`^`	Exponentiation	Renvoie le résultat de l'élévation de l'opérande gauche à la puissance de l'opérande droit. Notez que la valeur renvoyée par exponentiation est *toujours* un `Double` , quels que soient les types de valeur divisés. Toute coercition du résultat dans un type de variable a lieu ***après*** le calcul.
`/`	Division [1]	Renvoie le résultat de la division de l'opérande de gauche par l'opérande de droite. Notez que la valeur renvoyée par division est *toujours* un `Double` , indépendamment des types de valeur divisés. Toute contrainte du résultat dans un type de variable a lieu ***après*** le calcul.
`*`	Multiplication [1]	Renvoie le produit de 2 opérandes.
`\`	Division entière	Retourne le résultat entier de la division de l'opérande de gauche par l'opérande de droite ***après avoir*** arrondi les deux côtés avec un arrondi de 0,5 inférieur. Tout reste de la division est ignoré. Si l'opérande droit (le diviseur) est `0` , il en résultera une erreur d'exécution 11: division par zéro. Notez que c'est ***après que*** tout l'arrondi soit effectué - des expressions telles que `3 \ 0.4` entraîneront également une division par erreur zéro.
`Mod`	Modulo	Retourne le nombre entier restant de la division de l'opérande de

Jeton	prénom	La description
		gauche par l'opérande de droite. L'opérande de chaque côté est arrondi à un entier *avant* la division, avec 0,5 arrondi à la valeur inférieure. Par exemple, `8.6 Mod 3` et `12 Mod 2.6` donnent `0` . Si l'opérande droit (le diviseur) est `0` , il en résultera une erreur d'exécution 11: division par zéro. Notez que c'est ***après que*** tout l'arrondi soit effectué - des expressions telles que `3 Mod 0.4` entraîneront également une division par erreur zéro.
`-`	Soustraction [2]	Renvoie le résultat de la soustraction de l'opérande de droite de l'opérande de gauche.
`+`	Ajout [2]	Renvoie la somme de 2 opérandes. Notez que ce jeton est également traité comme un opérateur de concaténation lorsqu'il est appliqué à une `String` . Voir **Opérateurs de concaténation** .

[1] La multiplication et la division sont traitées comme ayant la même priorité.

[2] L' addition et la soustraction sont traitées comme ayant la même priorité.

Opérateurs de concaténation

VBA prend en charge 2 opérateurs de concaténation différents, `+` et `&` et tous deux exécutent exactement la même fonction lorsqu'ils sont utilisés avec les types `String` - la `String` droite est ajoutée à la fin de la `String` gauche.

Si l'opérateur `&` est utilisé avec un type de variable autre qu'une `String` , il est implicitement converti en une `String` avant d'être concaténé.

Notez que l'opérateur de concaténation `+` est une surcharge de l'opérateur `+` addition. Le comportement de `+` est déterminé par les types de variables des opérandes et la priorité des types d'opérateurs. Si les deux opérandes sont typés en tant que `String` ou `Variant` avec un sous-type de `String` , ils sont concaténés:

```
Public Sub Example()
    Dim left As String
    Dim right As String

    left = "5"
    right = "5"

    Debug.Print left + right    'Prints "55"
End Sub
```

Si l' *un des* côtés est un type numérique et que l'autre côté est une `String` pouvant être contenue dans un nombre, la priorité de type des opérateurs mathématiques entraîne le traitement de l'opérateur comme opérateur d'addition et les valeurs numériques sont ajoutées:

```
Public Sub Example()
```

```
    Dim left As Variant
    Dim right As String

    left = 5
    right = "5"

    Debug.Print left + right    'Prints 10
End Sub
```

Ce comportement peut conduire à des erreurs subtiles et difficiles à déboguer, en particulier si des types `Variant` sont utilisés, de sorte que seul l'opérateur `&` devrait être utilisé pour la concaténation.

Opérateurs de comparaison

Jeton	prénom	La description
`=`	Égal à	Renvoie `True` si les opérandes gauche et droit sont égaux. Notez qu'il s'agit d'une surcharge de l'opérateur d'affectation.
`<>`	Pas égal à	Renvoie `True` si les opérandes gauche et droite ne sont pas égaux.
`>`	Plus grand que	Renvoie `True` si l'opérande de gauche est supérieur à l'opérande de droite.
`<`	Moins que	Renvoie `True` si l'opérande de gauche est inférieur à l'opérande de droite.
`>=`	Meilleur que ou égal	Renvoie `True` si the si l'opérande de gauche est supérieur ou égal à l'opérande de droite.
`<=`	Inférieur ou égal	Renvoie `True` si the si l'opérande de gauche est inférieur ou égal à l'opérande de droite.
`Is`	Équité de référence	Renvoie `True` si la référence d'objet de gauche est la même instance que la référence d'objet de droite. Il peut également être utilisé avec `Nothing` (la référence d'objet null) de chaque côté. **Remarque:** L'opérateur `Is` tentera de contraindre les deux opérandes dans un `Object` avant d'effectuer la comparaison. Si l'un des côtés est un type primitif *ou* un `Variant` ne contenant pas d'objet (sous-type non-objet ou `vtEmpty`), la comparaison entraînera une erreur d'exécution 424 - "Objet requis". Si l'un des opérandes appartient à une *interface* différente du même objet, la comparaison renverra `True` . Si vous devez tester l'équité de l'instance *et de* l'interface, utilisez `ObjPtr(left) = ObjPtr(right)` .

Remarques

La syntaxe VBA autorise les "chaînes" d'opérateurs de comparaison, mais ces constructions doivent généralement être évitées. Les comparaisons sont toujours effectuées de gauche à droite sur seulement 2 opérandes à la fois, et chaque comparaison donne un `Boolean` . Par exemple, l'expression ...

```
a = 2: b = 1: c = 0
expr = a > b > c
```

... peut être lu dans certains contextes comme un test pour savoir si `b` est compris entre `a` et `c` . Dans VBA, cela se présente comme suit:

```
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
```

Tout opérateur de comparaison autre que `Is` utilisé avec un `Object` en tant qu'opérande sera exécuté sur la valeur de retour du membre par défaut de l' `Object` . Si l'objet n'a pas de membre par défaut, la comparaison entraînera une erreur d'exécution 438 - "L'objet ne prend pas en charge sa propriété ou sa méthode".

Si l' `Object` n'est pas initialisé, la comparaison entraînera une erreur d'exécution 91 - "Variable d'objet ou Variable de bloc non définie".

Si le littéral `Nothing` est utilisé avec un opérateur de comparaison autre que `Is` , cela entraînera une erreur de compilation - "Utilisation non valide de l'objet".

Si le membre par défaut de l' `Object` est *un autre* *`Object`* , VBA appellera continuellement le membre par défaut de chaque valeur de retour successive jusqu'à ce qu'un type primitif soit renvoyé ou qu'une erreur soit générée. Par exemple, supposons que `SomeClass` a un membre par défaut de `Value` , qui est une instance de `ChildClass` avec un membre par défaut de `ChildValue` . La comparaison...

```
Set x = New SomeClass
Debug.Print x > 42
```

... sera évalué comme:

```
Set x = New SomeClass
Debug.Print x.Value.ChildValue > 42
```

Si un opérande est un type numérique et que l' *autre* opérande est une `String` ou un `Variant` de `String` de sous-type, une comparaison numérique sera effectuée. Dans ce cas, si la `String` ne peut pas être convertie en un nombre, une erreur d'exécution 13 - "Incompatibilité de type" résultera de la comparaison.

Si les ***deux*** opérandes sont une `String` ou une `Variant` de `String` de sous-type, une comparaison de chaîne sera effectuée en fonction du paramètre Option Compare du module de code. Ces comparaisons sont effectuées caractère par caractère. Notez que la *représentation* des *caractères* d'une `String` contenant un nombre n'est ***pas*** la même que la comparaison des valeurs numériques:

```
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
```

Pour cette raison, assurez-vous que les variables `String` ou `Variant` sont converties en nombres avant d'effectuer des comparaisons numériques sur les inégalités.

Si un opérande est une `Date` , une comparaison numérique sur la valeur double sous-jacente sera effectuée si l'autre opérande est numérique ou peut être convertie en un type numérique.

Si l'autre opérande est une `String` ou une `Variant` de `String` de sous-type pouvant être convertie en `Date` utilisant les paramètres régionaux actuels, la `String` sera convertie en `Date` . S'il ne peut pas être converti en une `Date` dans les paramètres régionaux en cours, une erreur d'exécution 13 - "Incompatibilité de type" résultera de la comparaison.

Des précautions doivent être prises lors des comparaisons entre les valeurs `Double` ou `Single` et les valeurs booléennes . Contrairement à d'autres types numériques, les valeurs non nulles ne peuvent pas être considérées comme `True` raison du comportement de VBA consistant à promouvoir le type de données d'une comparaison impliquant un nombre à virgule flottante sur `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
```

Opérateurs binaires \ logiques

Tous les opérateurs logiques de VBA peuvent être considérés comme des "substitutions" des opérateurs binaires du même nom. Techniquement, ils sont *toujours* traités comme des opérateurs binaires. Tous les opérateurs de comparaison dans VBA renvoient un booléen , dont

aucun des bits ne sera défini (`False`) ou *tous* ses bits définis (`True`). Mais il traitera une valeur avec *n'importe quel* bit défini comme `True` . Cela signifie que le résultat de la conversion du résultat binaire d'une expression en un `Boolean` (voir Opérateurs de comparaison) sera toujours le même que le traiter comme une expression logique.

L'affectation du résultat d'une expression à l'aide de l'un de ces opérateurs donnera le résultat binaire. Notez que dans les tables de vérité ci-dessous, `0` est équivalent à `False` et `1` est équivalent à `True` .

`And`

Renvoie `True` si les expressions des deux côtés ont la valeur `True` .

Opérande de gauche	Opérande de droite	Résultat
0	0	0
0	1	0
1	0	0
1	1	1

`Or`

Renvoie `True` si l'un des côtés de l'expression correspond à `True` .

Opérande de gauche	Opérande de droite	Résultat
0	0	0
0	1	1
1	0	1
1	1	1

`Not`

Renvoie `True` si l'expression est `False` et `False` si l'expression est évaluée à `True` .

Opérande de droite	Résultat
0	1
1	0

`Not` est le seul opérande sans opérande de gauche. Visual Basic Editor simplifiera

automatiquement les expressions avec un argument de gauche. Si vous tapez ...

```
Debug.Print x Not y
```

... le VBE changera la ligne pour:

```
Debug.Print Not x
```

Des simplifications similaires seront apportées à toute expression contenant un opérande gauche (y compris les expressions) pour `Not` .

`Xor`

Aussi appelé "exclusif ou". Renvoie `True` si les deux expressions donnent des résultats différents.

Opérande de gauche	Opérande de droite	Résultat
0	0	0
0	1	1
1	0	1
1	1	0

Notez que bien que l'opérateur `Xor` puisse être *utilisé* comme un opérateur logique, il n'y a absolument aucune raison de le faire car il donne le même résultat que l'opérateur de comparaison `<>` .

`Eqv`

Aussi appelé "équivalence". Renvoie `True` lorsque les deux expressions ont le même résultat.

Opérande de gauche	Opérande de droite	Résultat
0	0	1
0	1	0
1	0	0
1	1	1

Notez que la fonction `Eqv` est *très* rarement utilisée, car `x Eqv y` est équivalent au `Not (x Xor y)` beaucoup plus lisible.

`Imp`

Aussi appelé "implication". Renvoie `True` si les deux opérandes sont identiques *ou que* le deuxième opérande est `True` .

Opérande de gauche	Opérande de droite	Résultat
0	0	1
0	1	1
1	0	0
1	1	1

Notez que la fonction `Imp` est très rarement utilisée. Une bonne règle de base est que si vous ne pouvez pas expliquer ce que cela signifie, vous devriez utiliser une autre construction.

Lire Les opérateurs en ligne: https://riptutorial.com/fr/vba/topic/5813/les-operateurs

Chapitre 29: Littéraux de chaîne - Fuites, caractères non imprimables et continuations de ligne

Remarques

L'attribution de littéraux de chaîne dans VBA est limitée par les limitations de l'EDI et la page de codes des paramètres de langue de l'utilisateur actuel. Les exemples ci-dessus illustrent les cas particuliers de chaînes d'échappement, de chaînes spéciales non imprimables et de chaînes longues.

Lors de l'attribution de chaînes littérales contenant des caractères spécifiques à une page de codes, vous devrez peut-être prendre en compte les problèmes d'internationalisation en attribuant une chaîne à partir d'un fichier de ressources Unicode distinct.

Examples

Fuyant le "personnage"

La syntaxe VBA exige qu'un littéral de chaîne apparaisse dans " marks, donc lorsque votre chaîne doit *contenir des* guillemets, vous devez échapper / ajouter le caractère " avec un " supplémentaire pour que VBA comprenne que vous souhaitez que le "" soit interprété comme une " chaîne " .

```
'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"
```

Affectation de littéraux de chaîne longue

L'éditeur VBA n'autorise que 1023 caractères par ligne, mais seuls les 100-150 premiers caractères sont visibles sans défilement. Si vous devez attribuer des chaînes de caractères longues, mais que vous souhaitez garder votre code lisible, vous devrez utiliser des lignes continues et une concaténation pour attribuer votre chaîne.

```
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 vous permettra d'utiliser un nombre limité de lignes continues (le nombre réel varie selon la longueur de chaque ligne dans le bloc continu). Par conséquent, si vous avez de très longues chaînes, vous devrez attribuer et réattribuer avec une concaténation. .

```
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.
```

Utilisation de constantes de chaîne VBA

VBA définit un nombre de constantes de chaîne pour les caractères spéciaux tels que:

- vbCr: Carriage-Return 'Identique à "\ r" dans les langages de style C.
- vbLf: saut de ligne 'Identique à "\ n" dans les langages de style C.
- vbCrLf: retour chariot et saut de ligne (une nouvelle ligne sous Windows)
- vbTab: Caractère de tabulation
- vbNullString: une chaîne vide, comme ""

Vous pouvez utiliser ces constantes avec une concaténation et d'autres fonctions de chaîne pour créer des chaînes de caractères avec des caractères spéciaux.

```
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
```

Utiliser `vbNullString` est considéré comme une meilleure pratique que la valeur équivalente de `""` raison des différences dans la façon dont le code est compilé. Les chaînes sont accessibles via un pointeur vers une zone de mémoire allouée et le compilateur VBA est suffisamment intelligent

pour utiliser un pointeur nul pour représenter `vbNullString` . Le littéral `""` se voit attribuer de la mémoire comme s'il s'agissait d'une variante de type String, rendant l'utilisation de la constante beaucoup plus efficace:

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

Lire Littéraux de chaîne - Fuites, caractères non imprimables et continuations de ligne en ligne: https://riptutorial.com/fr/vba/topic/3445/litteraux-de-chaine---fuites--caracteres-non-imprimables-et-continuations-de-ligne

Chapitre 30: Manipulation de chaîne fréquemment utilisée

Introduction

Exemples rapides de fonctions de chaîne MID LEFT et RIGHT en utilisant INSTR FIND et LEN.

Comment trouvez-vous le texte entre deux termes de recherche (Say: après deux points et avant une virgule)? Comment obtenez-vous le reste d'un mot (en utilisant MID ou en utilisant RIGHT)? Laquelle de ces fonctions utilise des paramètres basés sur zéro et des codes de retour vs One-based? Que se passe-t-il quand les choses tournent mal? Comment traitent-ils les chaînes vides, les résultats non fondés et les nombres négatifs?

Examples

Manipulation de chaînes exemples fréquemment utilisés

Better MID () et d'autres exemples d'extraction de chaînes, actuellement absents du Web. S'il vous plaît, aidez-moi à faire un bon exemple ou complétez celui-ci ici. Quelque chose comme ça:

```
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) '...
```

N'hésitez pas à modifier cet exemple et à l'améliorer. Tant que cela reste clair et comporte des pratiques d'utilisation courantes.

Lire Manipulation de chaîne fréquemment utilisée en ligne:
https://riptutorial.com/fr/vba/topic/8890/manipulation-de-chaine-frequemment-utilisee

Chapitre 31: Mesurer la longueur des cordes

Remarques

La longueur d'une chaîne peut être mesurée de deux manières: La mesure de longueur la plus fréquemment utilisée est le nombre de caractères utilisant les fonctions `Len` , mais VBA peut également révéler le nombre d'octets utilisant des fonctions `LenB` . Un caractère double octet ou Unicode est long de plus d'un octet.

Examples

Utilisez la fonction Len pour déterminer le nombre de caractères d'une chaîne

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

Dim charLength As Long

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

Utilisez la fonction LenB pour déterminer le nombre d'octets d'une chaîne

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

Dim byteLength As Long

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

Prefer `If Len (myString) = 0 Alors` over `If myString =" "Then`

Lorsque vous vérifiez si une chaîne est de longueur zéro, il est préférable et plus efficace d'inspecter la longueur de la chaîne plutôt que de comparer la chaîne à une chaîne vide.

```
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
```

Lire Mesurer la longueur des cordes en ligne: https://riptutorial.com/fr/vba/topic/3576/mesurer-la-longueur-des-cordes

Chapitre 32: Mot-clé VBA Option

Syntaxe

- Option optionName [valeur]
- Option explicite
- Option Comparer {Texte | Binaire | Base de données}
- Module privé d'option
- Option Base {0 | 1}

Paramètres

Option	Détail
Explicite	*Exiger une déclaration de variable* dans le module dans lequel il est spécifié (idéalement, tous); Avec cette option spécifiée, l'utilisation d'une variable non déclarée (/ mal orthographiée) devient une erreur de compilation.
Comparer le texte	Rend les comparaisons de chaînes du module insensibles à la casse, en fonction des paramètres régionaux du système, en hiérarchisant les équivalences alphabétiques (par exemple, "a" = "A").
Comparer les binaires	Mode de comparaison de chaîne par défaut. Rend les comparaisons de chaînes du module sensibles à la casse, en comparant les chaînes à l'aide de la représentation numérique / valeur numérique de chaque caractère (par exemple, ASCII).
Comparer la base de données	(MS-Access uniquement) Permet aux comparaisons de chaînes du module de fonctionner comme elles le feraient dans une instruction SQL.
Module privé	Empêche l'accès au membre `Public` du module en dehors du projet dans lequel le module réside, masquant efficacement les procédures de l'application hôte (c'est-à-dire non disponibles pour être utilisées comme macros ou fonctions définies par l'utilisateur).
Option Base 0	Paramètres par défaut. Définit le tableau implicite inférieur lié à `0` dans un module. Lorsqu'un tableau est déclaré sans valeur de limite inférieure explicite, `0` sera utilisé.
Option Base 1	Définit la limite inférieure du tableau implicite à `1` dans un module. Lorsqu'un tableau est déclaré sans valeur de limite inférieure explicite, `1` sera utilisé.

Remarques

Il est beaucoup plus facile de contrôler les limites des tableaux en déclarant explicitement les limites plutôt que de laisser le compilateur se rabattre sur une déclaration `Option Base {0|1}` . Cela peut être fait comme ça:

```
Dim myStringsA(0 To 5) As String '// This has 6 elements (0 - 5)
Dim myStringsB(1 To 5) As String '// This has 5 elements (1 - 5)
Dim myStringsC(6 To 9) As String '// This has 3 elements (6 - 9)
```

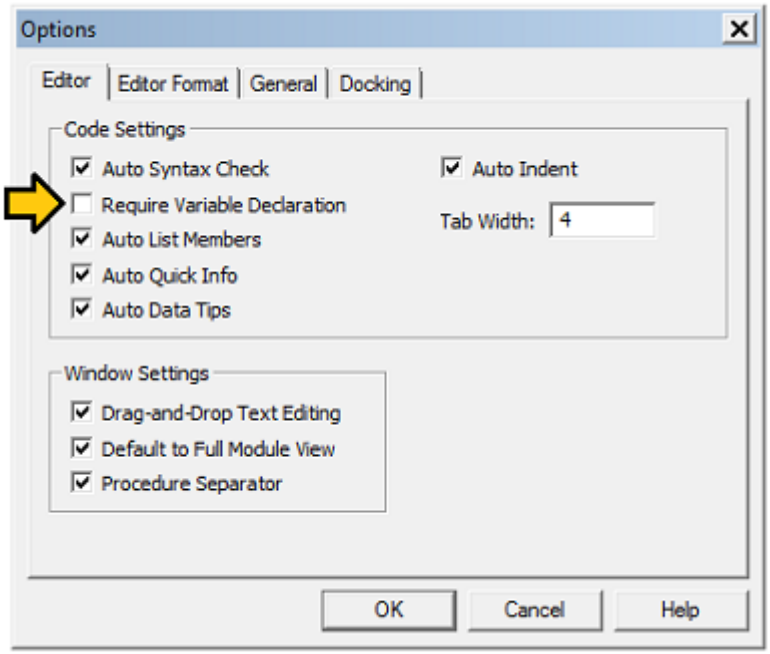
Examples

Option explicite

Il est recommandé de toujours utiliser `Option Explicit` dans VBA, car cela oblige le développeur à déclarer toutes ses variables avant utilisation. Cela présente également d'autres avantages, tels que la mise en majuscules automatique pour les noms de variables déclarés et IntelliSense.

```
Option Explicit

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

Si vous définissez Exiger une déclaration de variable dans la page Outils ► `Options` ► `Éditeur de VBE, l'instruction` **Option Explicit** apparaîtra en haut de chaque nouvelle feuille de code.



Cela évitera les erreurs de codage idiotes comme les fautes d'orthographe et vous incitera à utiliser le type de variable correct dans la déclaration de variable. (Quelques exemples supplémentaires sont donnés à TOUJOURS utiliser "Option Explicit" .)

Option Comparer les binaires

La comparaison binaire rend toutes les vérifications d'égalité de chaîne dans un module / classe *sensibles à la casse* . Techniquement, avec cette option, les comparaisons de chaînes sont effectuées en utilisant l'ordre de tri des représentations binaires de chaque caractère.

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

Si aucune comparaison d'options n'est spécifiée dans un module, Binary est utilisé par défaut.

```
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 Comparer du texte

Option Compare Text fait que toutes les comparaisons de chaînes dans un module / une classe utilisent une comparaison *insensible à la casse* .

(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 Comparer la base de données

Option Compare Database est uniquement disponible dans MS Access. Il définit le module / classe pour utiliser les paramètres de base de données en cours pour déterminer s'il faut utiliser le mode texte ou binaire.

Remarque: L'utilisation de ce paramètre est déconseillée, sauf si le module est utilisé pour écrire des UDF (fonctions définies par l'utilisateur) Access personnalisées devant traiter les comparaisons de texte de la même manière que les requêtes SQL dans cette base de données.

Option Base {0 | 1}

`Option Base` est utilisée pour déclarer la limite inférieure par défaut des éléments du **tableau** . Il est déclaré au niveau du module et n'est valide que pour le module actuel.

Par défaut (et donc si aucune Base Option n'est spécifiée), la Base est 0. Ce qui signifie que le premier élément de tout tableau déclaré dans le module a un index de 0.

Si `Option Base 1` est spécifié, le premier élément du tableau a l'index 1

Exemple en 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
```

Même exemple avec 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
```

Le deuxième exemple a généré un indice hors plage (erreur 9) au premier stade de la boucle car une tentative d'accès à l'index 0 du tableau a été effectuée et cet index n'existe pas lorsque le module est déclaré avec `Base 1`

Le code correct avec Base 1 est:

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

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

    Next i
```

Il convient de noter que la fonction Split crée **toujours** un tableau avec un index d'élément de base zéro, quel que soit le paramètre `Option Base` . Vous trouverez ici des exemples d'utilisation de la fonction **Split** .

Fonction Split

Retourne un tableau à une dimension, basé sur zéro, contenant un nombre spécifié de sous-chaînes.

Dans Excel, les propriétés `Range.Value` et `Range.Formula` d'une plage à plusieurs cellules renvoient *toujours* un tableau à variantes 2D basé sur 1.

De même, dans ADO, la méthode `Recordset.GetRows` renvoie *toujours* un tableau 2D basé sur 1.

L'une des meilleures pratiques recommandées consiste à toujours utiliser les fonctions LBound et UBound pour déterminer les étendues d'un tableau.

```
'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
```

L' `Option Base 1` doit être située en haut de chaque module de code où un tableau est créé ou redimensionné si les tableaux doivent être créés systématiquement avec une limite inférieure de 1.

Lire Mot-clé VBA Option en ligne: https://riptutorial.com/fr/vba/topic/3992/mot-cle-vba-option

Chapitre 33: Objet Scripting.Dictionary

Remarques

Vous devez ajouter Microsoft Scripting Runtime au projet VBA via la commande Outils → Références de VBE afin de mettre en œuvre une liaison anticipée de l'objet Scripting Dictionary. Cette référence de bibliothèque est portée avec le projet; il n'est pas nécessaire de le référencer à nouveau lorsque le projet VBA est distribué et exécuté sur un autre ordinateur.

Examples

Propriétés et méthodes

Un objet Scripting Dictionary stocke des informations dans des paires Key / Item. Les clés doivent être uniques et non un tableau, mais les éléments associés peuvent être répétés (leur caractère unique est détenu par la clé associée) et peuvent être de n'importe quel type de variante ou d'objet.

Un dictionnaire peut être considéré comme une base de données en mémoire à deux champs avec un index unique primaire sur le premier «champ» (la *clé*). Cet index unique sur la propriété Keys permet des recherches très rapides pour récupérer la valeur d'un élément associé à une clé.

Propriétés

prénom	lire écrire	type	la description
CompareMode	*lire écrire*	Constante CompareMode	La définition de CompareMode ne peut être effectuée que sur un dictionnaire vide. Les valeurs acceptées sont 0 (vbBinaryCompare), 1 (vbTextCompare), 2 (vbDatabaseCompare).
Compter	*lecture seulement*	entier long non signé	Un compte à base unique des paires clé / article dans l'objet du dictionnaire de script.
Clé	*lire écrire*	variante non-tableau	Chaque clé unique dans le dictionnaire.
Article (*clé*)	*lire écrire*	toute variante	Propriété par défaut Chaque élément individuel associé à une clé dans le dictionnaire. Notez que tenter de récupérer un élément avec une clé qui n'existe pas dans le dictionnaire *ajoutera implicitement* la clé transmise.

Les méthodes

prénom	la description
Ajouter (*clé* , *article*)	Ajoute une nouvelle clé et un nouvel élément au dictionnaire. La nouvelle clé ne doit pas exister dans la collection Keys actuelle du dictionnaire, mais un élément peut être répété parmi de nombreuses clés uniques.
Existe (*clé*)	Test booléen pour déterminer si une clé existe déjà dans le dictionnaire.
Clés	Renvoie le tableau ou la collection de clés uniques.
Articles	Renvoie le tableau ou la collection d'éléments associés.
Supprimer (*clé*)	Supprime une clé de dictionnaire individuelle et son élément associé.
Enlever tout	Efface toutes les clés et les éléments d'un objet du dictionnaire.

Exemple de 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
```

Agrégation des données avec Scripting.Dictionary (Maximum, Count)

Les dictionnaires sont parfaits pour gérer les informations lorsque plusieurs entrées se produisent, mais vous ne vous préoccupez que d'une seule valeur pour chaque ensemble d'entrées - la première ou la dernière valeur, la valeur minimale ou maximale, une moyenne, une somme, etc.

Considérez un classeur contenant un journal d'activité de l'utilisateur, avec un script qui insère le nom d'utilisateur et la date de modification chaque fois que quelqu'un modifie le classeur:

`Log` travail de `Log`

UNE	B
bob	10/12/2016 9:00
Alice	10/13/2016 13:00
bob	10/13/2016 13:30

UNE	B
Alice	10/13/2016 14:00
Alice	10/14/2016 13:00

Supposons que vous souhaitiez afficher la dernière heure de modification pour chaque utilisateur dans une feuille de calcul appelée `Summary` .

Remarques:

1. Les données sont supposées être dans `ActiveWorkbook` .
2. Nous utilisons un tableau pour extraire les valeurs de la feuille de calcul; c'est plus efficace que l'itération sur chaque cellule.
3. Le `Dictionary` est créé en utilisant la liaison anticipée.

```
Sub LastEdit()
Dim vLog as Variant, vKey as Variant
Dim dict as New Scripting.Dictionary
Dim lastRow As Integer, lastColumn As Integer
Dim i as Long
Dim anchor As Range

With ActiveWorkbook
    With .Sheets("Log")
        'Pull entries in "log" into a variant array
        lastRow = .Range("a" & .Rows.Count).End(xlUp).Row
        vlog = .Range("a1", .Cells(lastRow, 2)).Value2

        'Loop through array
        For i = 1 to lastRow
            Dim username As String
            username = vlog(i, 1)
            Dim editDate As Date
            editDate = vlog(i, 2)

            'If the username is not yet in the dictionary:
            If Not dict.Exists(username) Then
                dict(username) = editDate
            ElseIf dict(username) < editDate Then
                dict(username) = editDate
            End If
        Next
    End With

    With .Sheets("Summary")
        'Loop through keys
        For Each vKey in dict.Keys
            'Add the key and value at the next available row
            Anchor = .Range("A" & .Rows.Count).End(xlUp).Offset(1,0)
            Anchor = vKey
            Anchor.Offset(0,1) = dict(vKey)
        Next vKey
    End With
End With
End Sub
```

et la sortie ressemblera à ceci:

Summary travail Summary

UNE	B
bob	10/13/2016 13:30
Alice	10/14/2016 13:00

Si, d'autre part, vous souhaitez afficher le nombre de fois que chaque utilisateur a modifié le classeur, le corps de la boucle `For` doit ressembler à ceci:

```
        'Loop through array
        For i = 1 to lastRow
            Dim username As String
            username = vlog(i, 1)

            'If the username is not yet in the dictionary:
            If Not dict.Exists(username) Then
                dict(username) = 1
            Else
                dict(username) = dict(username) + 1
            End If
        Next
```

et la sortie ressemblera à ceci:

Summary travail Summary

UNE	B
bob	2
Alice	3

Obtenir des valeurs uniques avec Scripting.Dictionary

Le `Dictionary` permet d'obtenir un ensemble unique de valeurs très simplement. Considérons la fonction suivante:

```
Function Unique(values As Variant) As Variant()
    'Put all the values as keys into a dictionary
    Dim dict As New Scripting.Dictionary
    Dim val As Variant
    For Each val In values
        dict(val) = 1 'The value doesn't matter here
    Next
    Unique = dict.Keys
End Function
```

que vous pourriez alors appeler comme ceci:

```
Dim duplicates() As Variant
duplicates = Array(1, 2, 3, 1, 2, 3)
Dim uniqueVals() As Variant
uniqueVals = Unique(duplicates)
```

et `uniqueVals` ne contiendrait que `{1,2,3}` .

Remarque: Cette fonction peut être utilisée avec n'importe quel objet énumérable.

Lire Objet Scripting.Dictionary en ligne: https://riptutorial.com/fr/vba/topic/3667/objet-scripting-dictionary

Chapitre 34: Rechercher dans les chaînes la présence de sous-chaînes

Remarques

Lorsque vous devez vérifier la présence ou la position d'une sous-chaîne dans une chaîne, VBA propose les fonctions `InStr` et `InStrRev` qui renvoient la position de la chaîne dans la chaîne, si elle est présente.

Examples

Utiliser InStr pour déterminer si une chaîne contient une sous-chaîne

```
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
```

Utiliser InStr pour rechercher la position de la première instance d'une sous-chaîne

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

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

Utiliser InStrRev pour rechercher la position de la dernière instance d'une sous-chaîne

```
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 paramters for InStrRev
posX = InStrRev(baseString, "X", -1, vbBinaryCompare)
'posX = 0
```

Lire Rechercher dans les chaînes la présence de sous-chaînes en ligne:

https://riptutorial.com/fr/vba/topic/3480/rechercher-dans-les-chaines-la-presence-de-sous-chaines

Chapitre 35: Récursivité

Introduction

Une fonction qui s'appelle elle-même est dite *récursive* . La logique récursive peut souvent être implémentée comme une boucle. La récursivité doit être contrôlée avec un paramètre, de sorte que la fonction sache quand arrêter de récurer et d'approfondir la pile d'appels. *Une récursion infinie* entraîne éventuellement une erreur d'exécution "28": "Espace hors pile".

Voir récursivité .

Remarques

La récursivité permet des appels répétés et auto-référencés d'une procédure.

Examples

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
```

Récursivité des dossiers

Early Bound (avec une référence à `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
```

Lire Récursivité en ligne: https://riptutorial.com/fr/vba/topic/3236/recursivite

Chapitre 36: Scripting.FileSystemObject

Examples

Créer un objet 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
```

Lecture d'un fichier texte à l'aide d'un objet 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
```

Création d'un fichier texte avec 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
```

Ecrire dans un fichier existant avec 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
```

Énumérer les fichiers dans un répertoire à l'aide de FileSystemObject

Relié tôt (nécessite une référence à 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
```

Retardé:

```
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
```

Énumérer récursivement les dossiers et les fichiers

Early Bound (avec une référence à `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
```

Sortie lorsqu'elle est appelée avec des arguments tels que: `EnumerateFilesAndFolders "C:\Test"`

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

Sortie lorsqu'elle est appelée avec des arguments comme: `EnumerateFilesAndFolders "C:\Test", 0`

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

Sortie lorsqu'elle est appelée avec des arguments comme: `EnumerateFilesAndFolders "C:\Test", 1, 4`

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

Supprimer l'extension de fichier à partir d'un nom de fichier

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

Imprime `MyFile.something`

Notez que la méthode `GetBaseName()` gère déjà plusieurs périodes dans un nom de fichier.

Récupère uniquement l'extension d'un nom de fichier

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

Prints `txt` Notez que la méthode `GetExtensionName()` gère déjà plusieurs périodes dans un nom de fichier.

Récupérer uniquement le chemin depuis un chemin de fichier

La méthode GetParentFolderName renvoie le dossier parent pour tout chemin d'accès. Bien que cela puisse également être utilisé avec des dossiers, il est sans doute plus utile pour extraire le chemin d'un chemin de fichier absolu:

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

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

Notez que le séparateur de chemin de fin n'est pas inclus dans la chaîne renvoyée.

Utilisation de FSO.BuildPath pour créer un chemin complet à partir du chemin du dossier et du nom du fichier

Si vous acceptez une entrée utilisateur pour les chemins de dossier, vous devrez peut-être vérifier les barres obliques inverses (`\`) avant de créer un chemin de fichier. La méthode `FSO.BuildPath` rend cela plus simple:

```
  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)
```

Sortie:

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

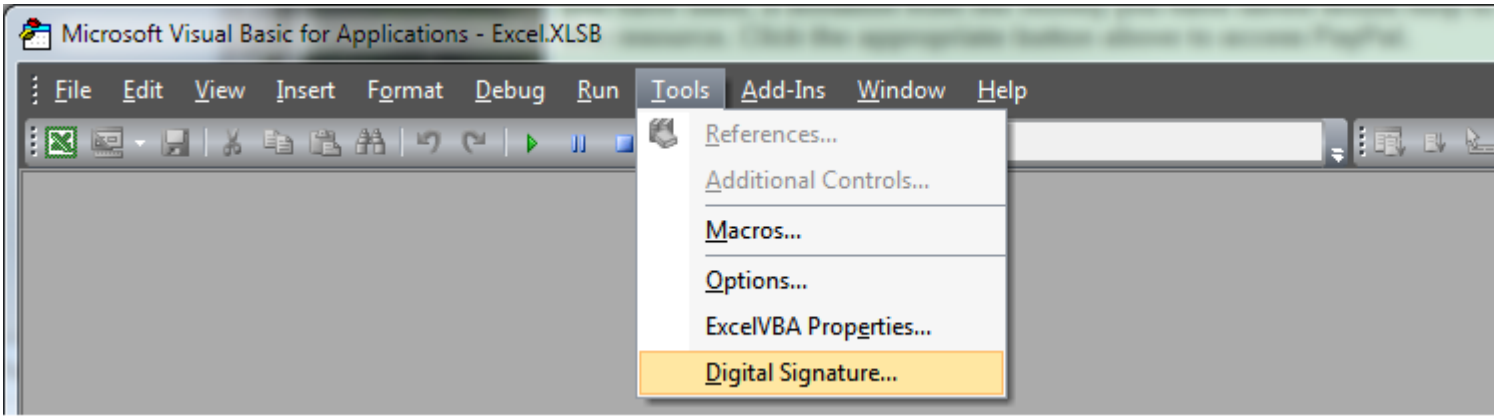
Lire Scripting.FileSystemObject en ligne: https://riptutorial.com/fr/vba/topic/990/scripting-filesystemobject

Chapitre 37: Sécurité des macros et signature des projets / modules VBA

Examples

Créez un certificat auto-signé numérique valide SELFCERT.EXE

Pour exécuter des macros et gérer la sécurité fournie par les applications Office contre le code malveillant, il est nécessaire de signer numériquement VBAProject.OTM à partir de l' *éditeur VBA> Outils> Signature numérique* .

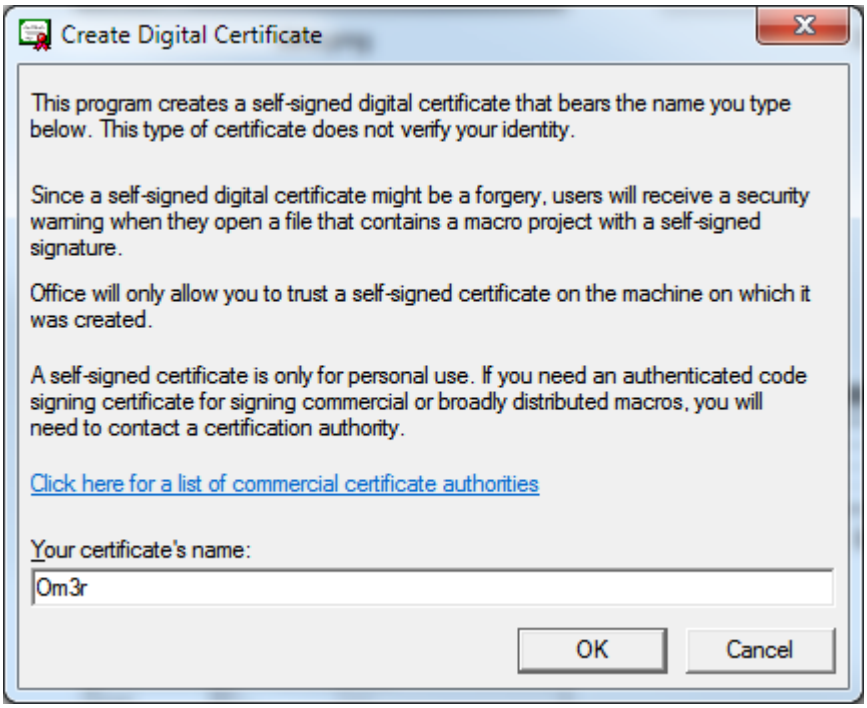


Office est fourni avec un utilitaire permettant de créer un certificat numérique auto-signé que vous pouvez utiliser sur le PC pour signer vos projets.

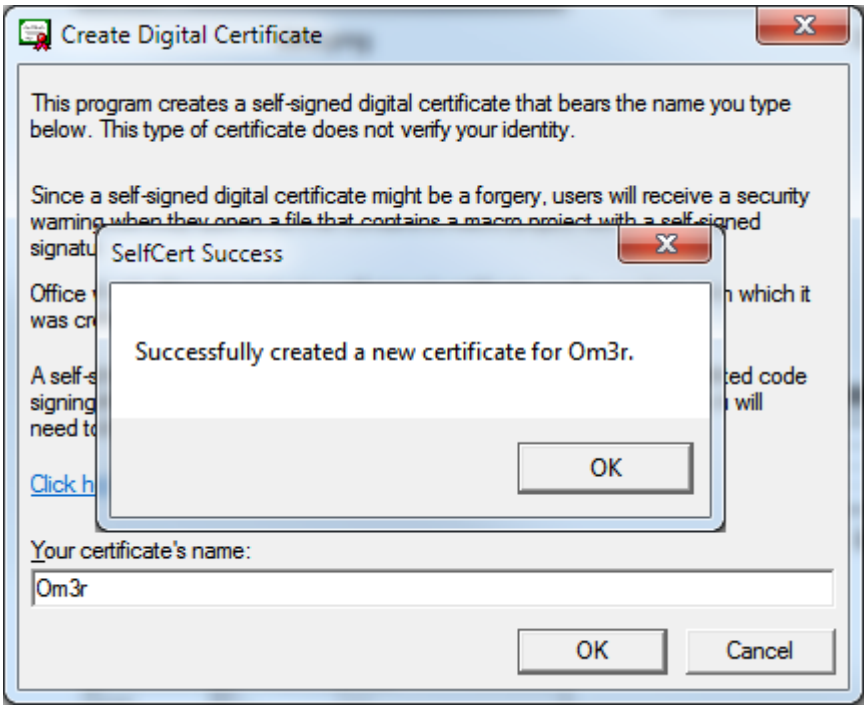
Cet utilitaire **SELFCERT.EXE se** trouve dans le dossier du programme Office,

Cliquez sur Certificat numérique pour les projets VBA pour ouvrir l' *assistant de* certificat.

Dans la boîte de dialogue, entrez un nom approprié pour le certificat et cliquez sur OK.

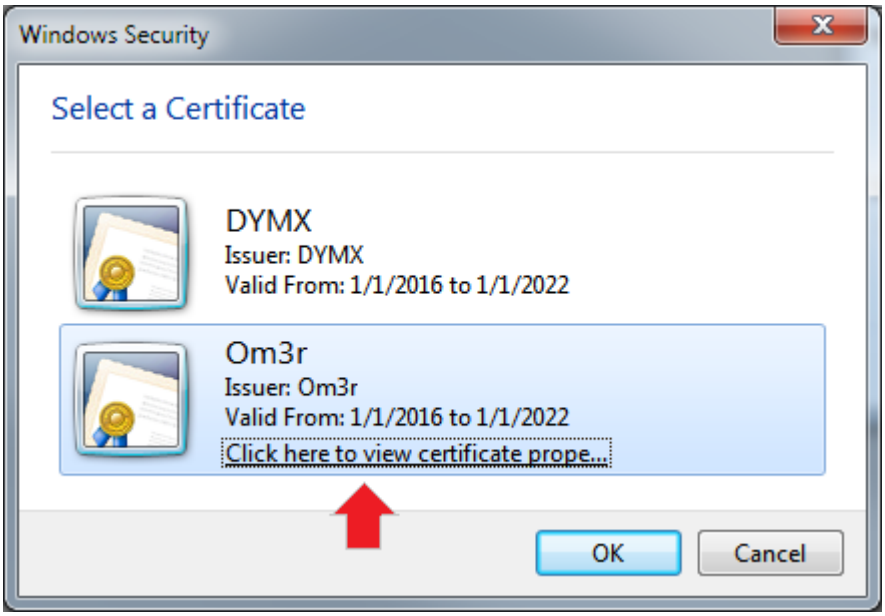


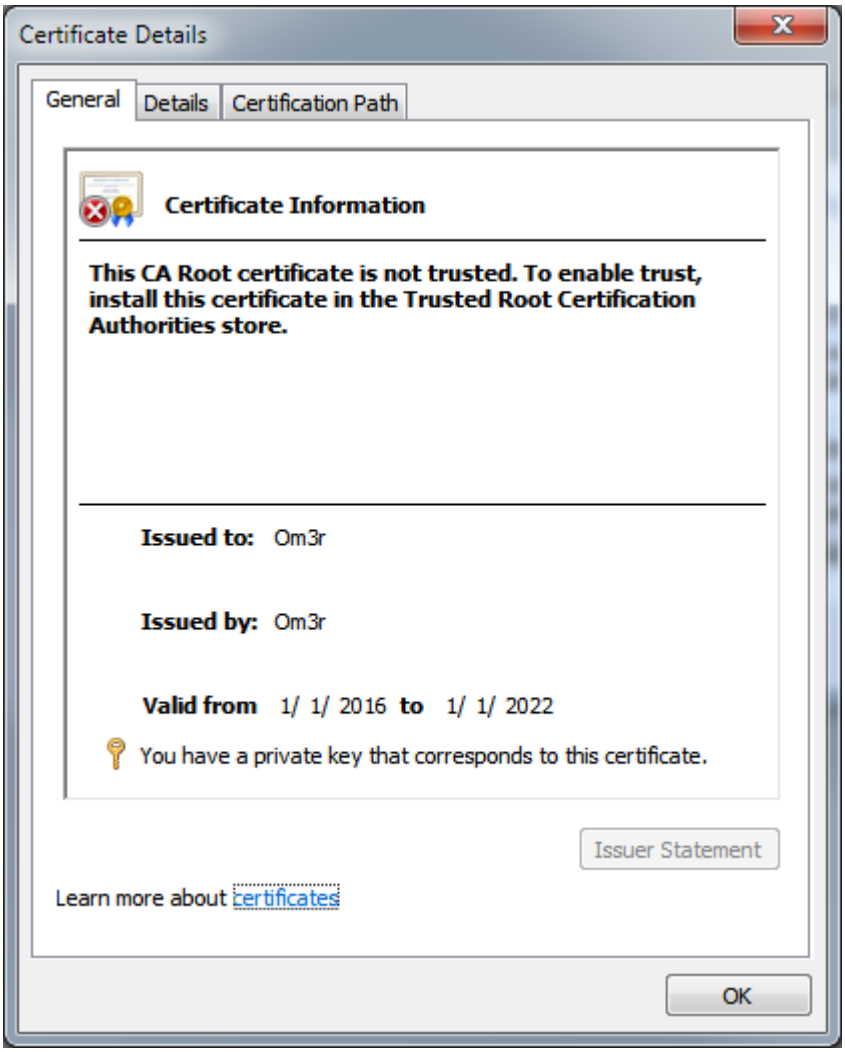
Si tout se passe bien, vous verrez une confirmation:



Vous pouvez maintenant fermer l'assistant **SELFCERT** et **porter** votre attention sur le certificat que vous avez créé.

Si vous essayez d'utiliser le certificat que vous venez de créer et que vous vérifiez ses propriétés

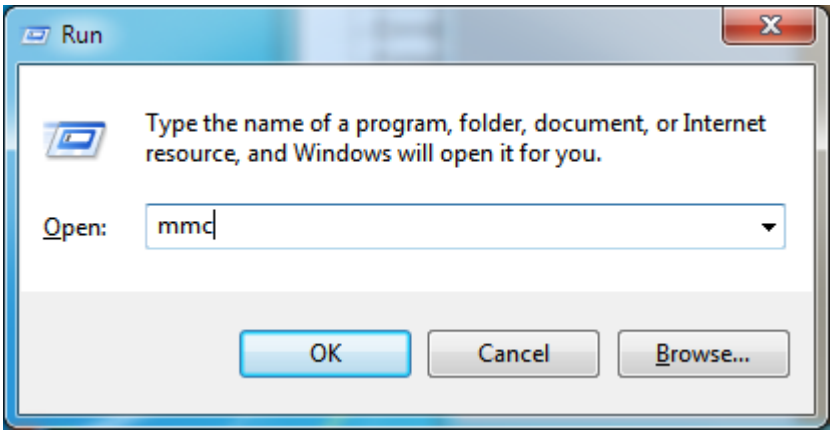




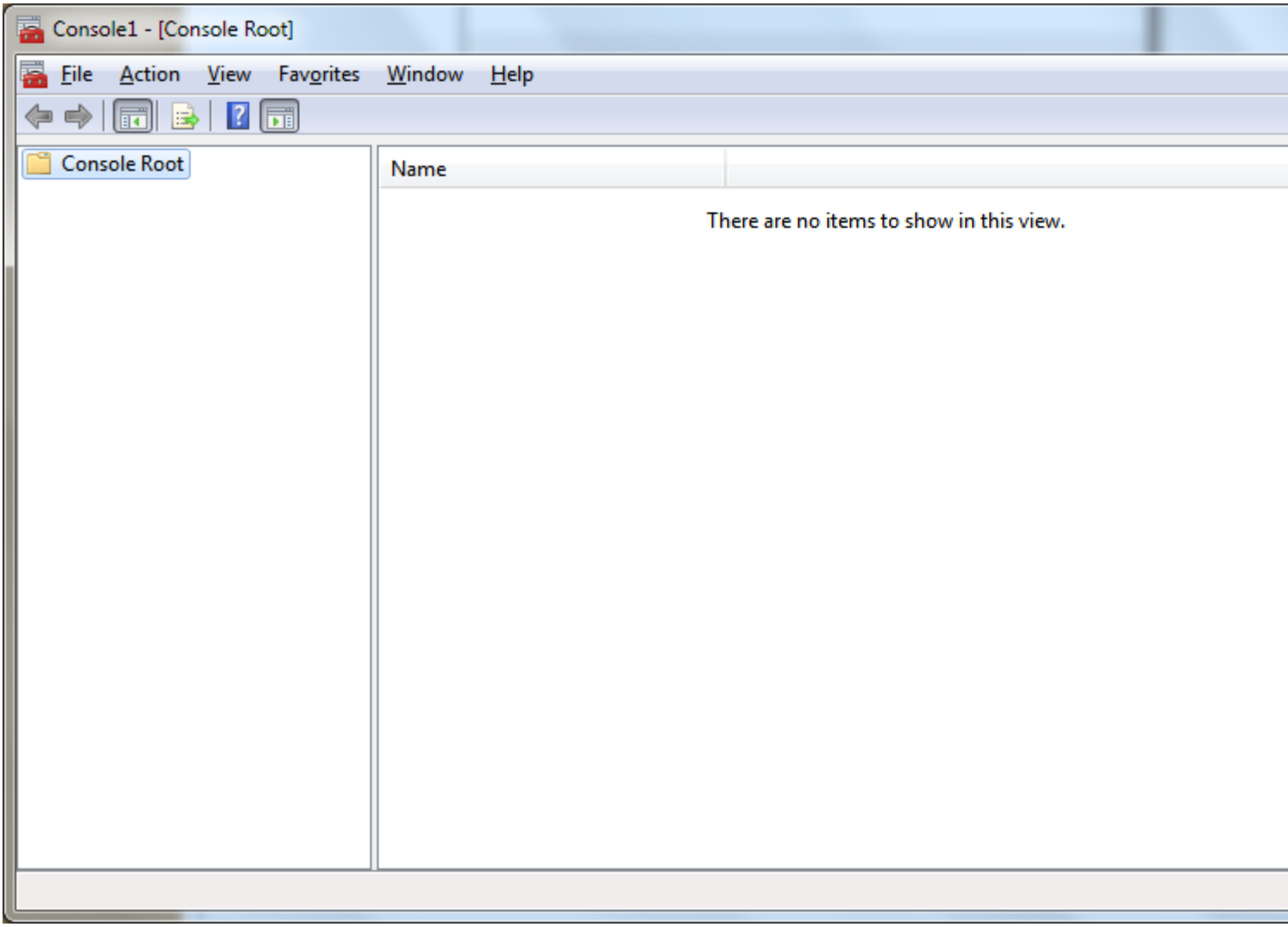
Vous verrez que le certificat n'est pas approuvé et que la raison est indiquée dans la boîte de dialogue.

Le certificat a été créé dans le magasin Utilisateur actuel> Personnel> Certificats. Il doit se trouver dans Ordinateur local> Autorités de certification racine de confiance> Banque de certificats, vous devez donc exporter à partir de la première et importer dans cette dernière.

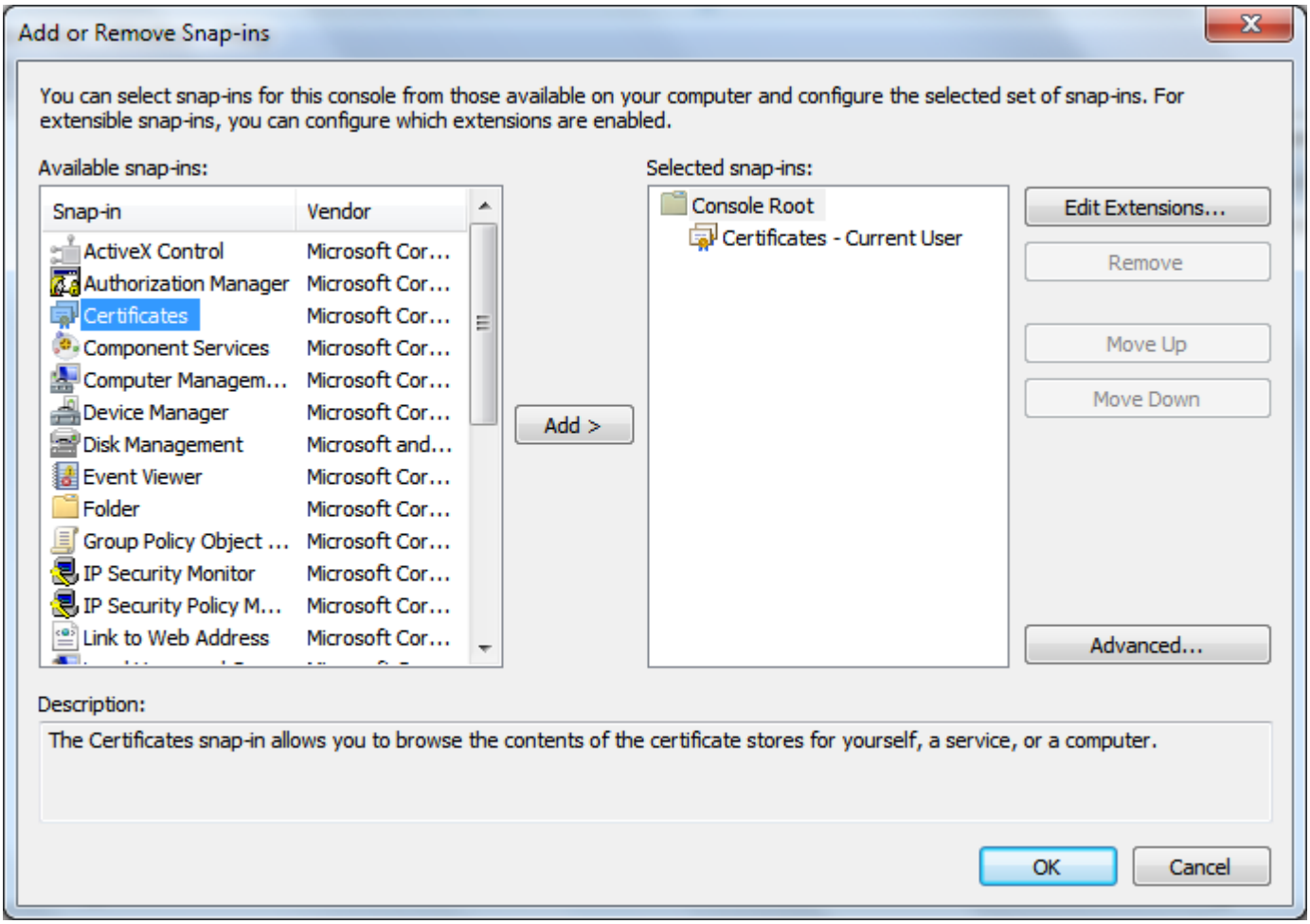
Appuyez sur la touche Windows + R pour ouvrir la fenêtre "Exécuter". Entrez ensuite «mmc» dans la fenêtre comme indiqué ci-dessous et cliquez sur «OK».



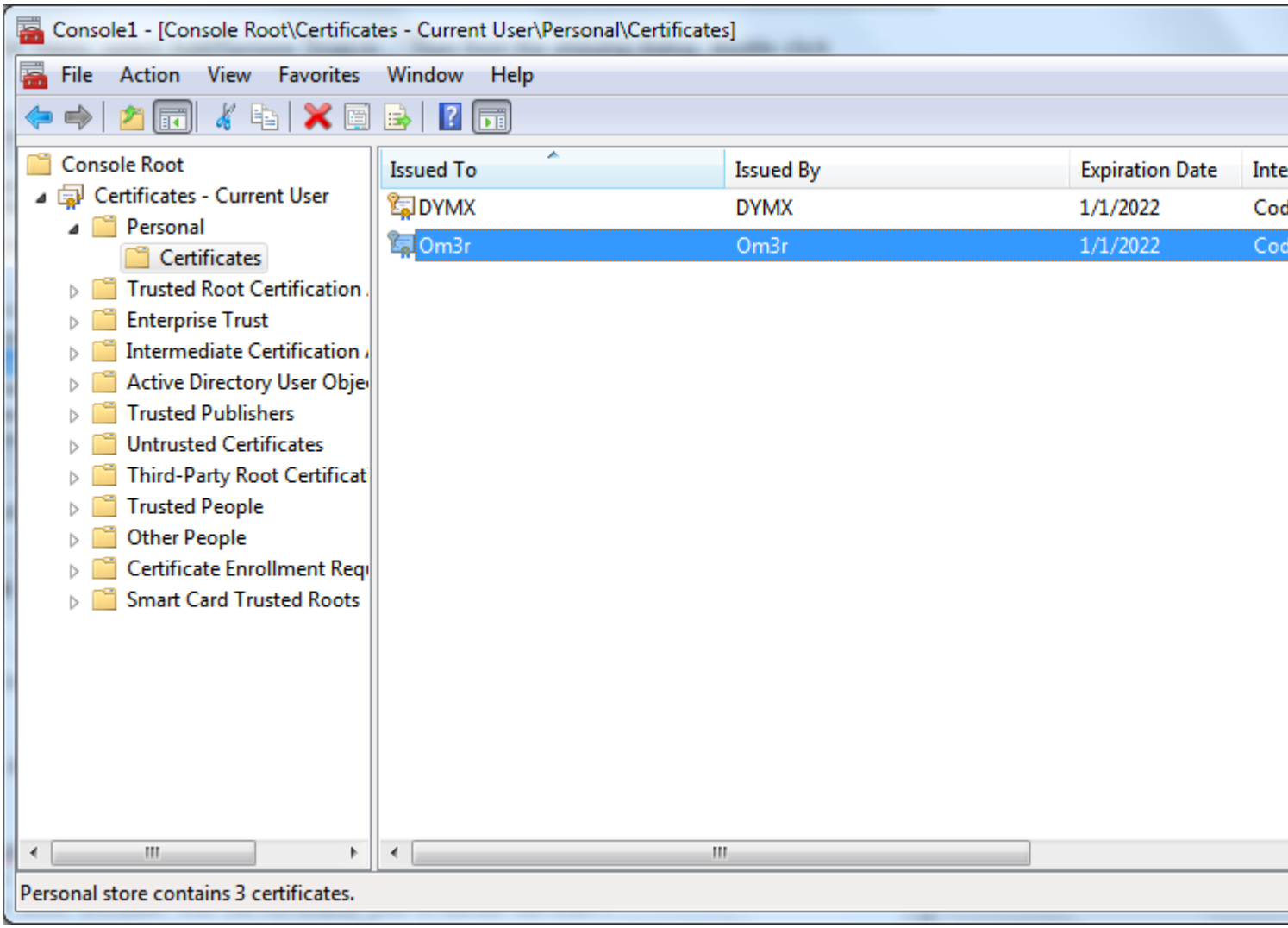
La console de gestion Microsoft s'ouvre et ressemble à ce qui suit.



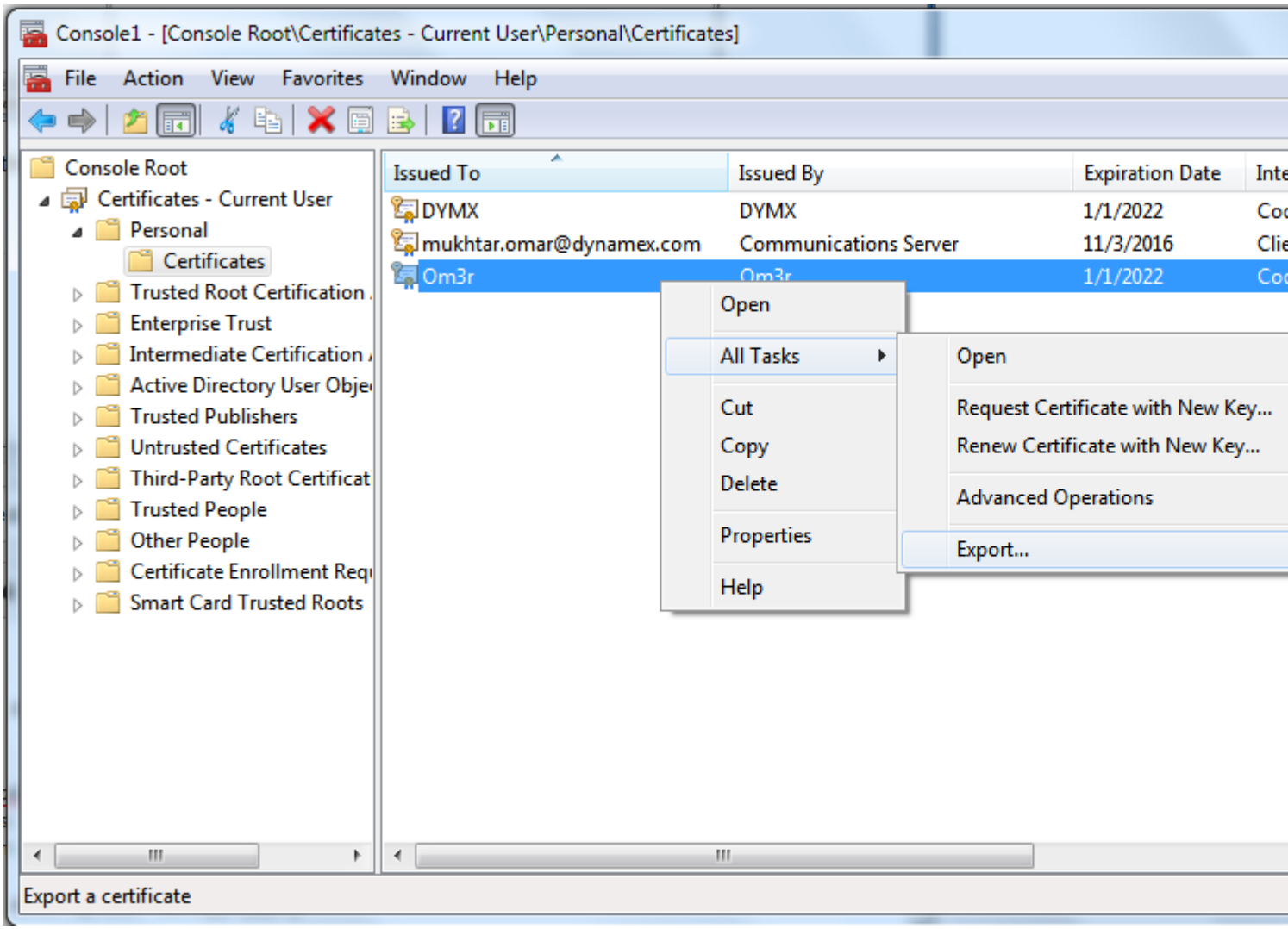
Dans le menu Fichier, sélectionnez Ajouter / Supprimer un composant logiciel enfichable ... Dans la boîte de dialogue suivante, double-cliquez sur Certificats, puis cliquez sur OK.



Développez la liste déroulante dans la fenêtre de gauche pour *Certificats - Utilisateur actuel* »et sélectionnez les certificats comme indiqué ci-dessous. Le panneau central affichera alors les certificats à cet emplacement, qui incluront le certificat que vous avez créé précédemment:

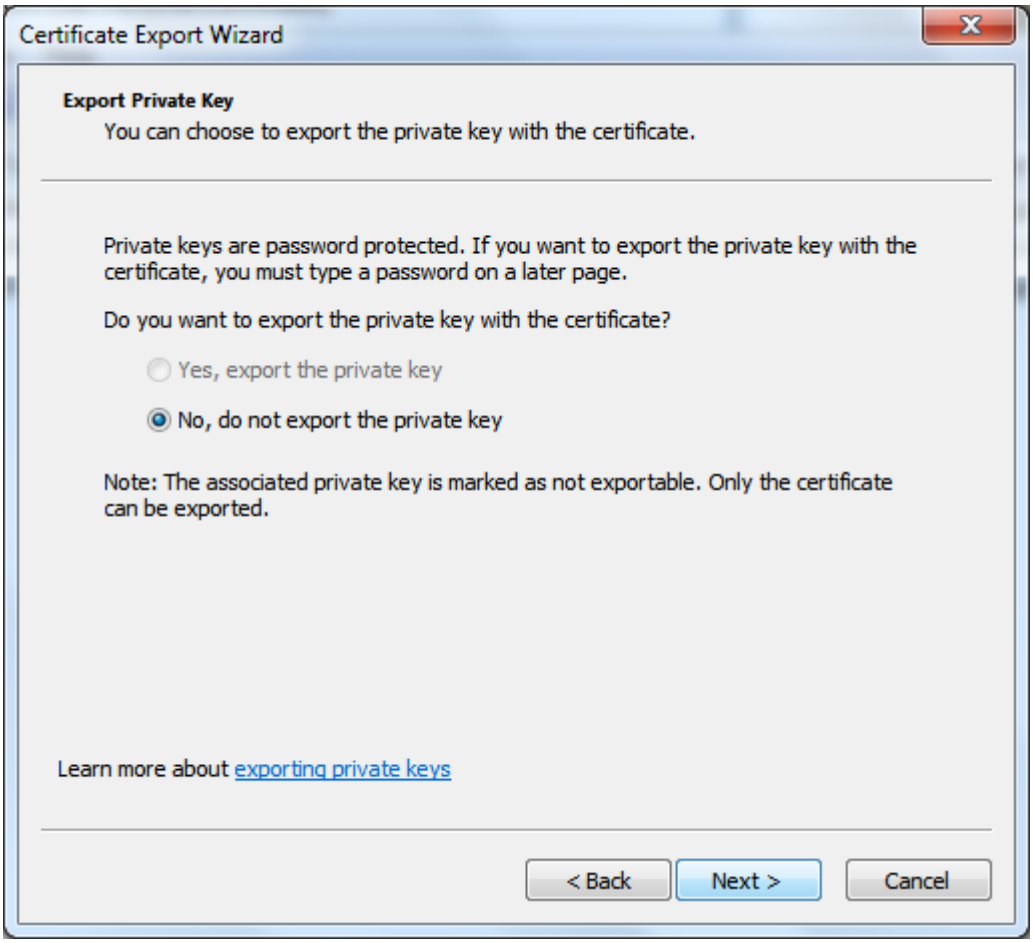


Cliquez avec le bouton droit sur le certificat et sélectionnez Toutes les tâches> Exporter:

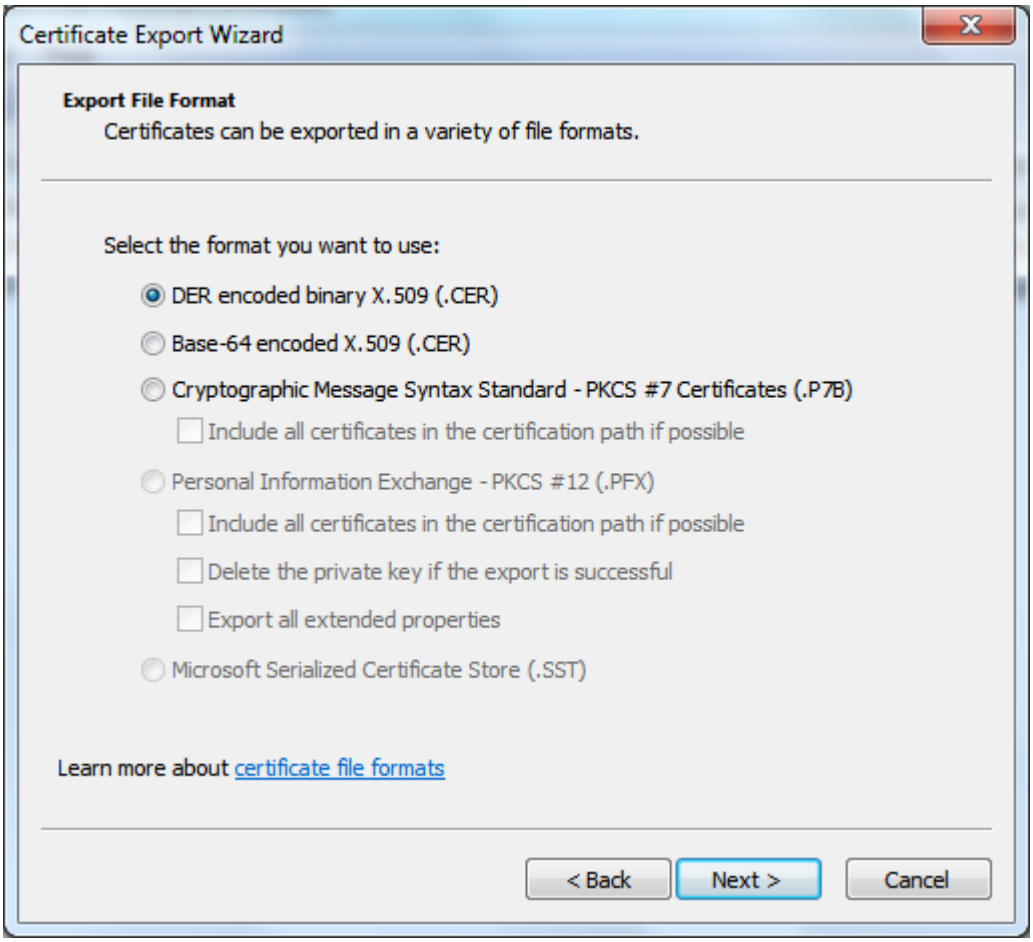


Assistant d'exportation

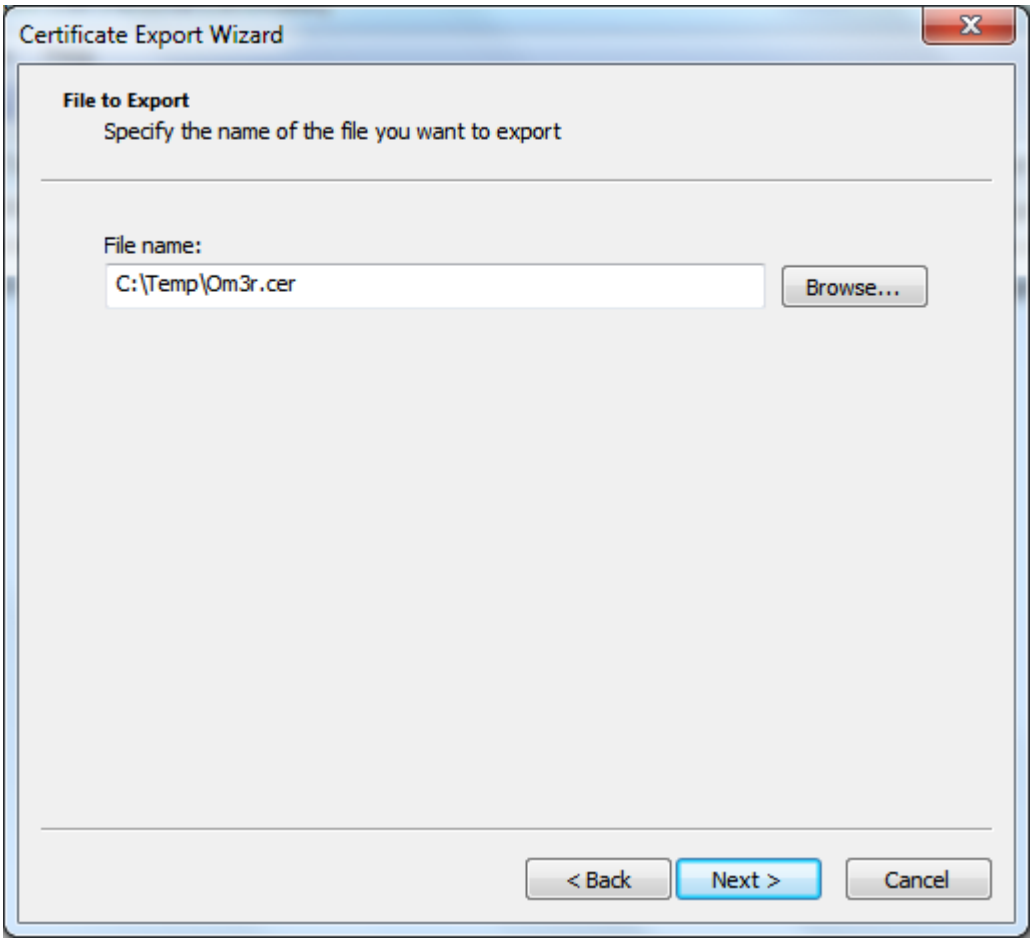
Cliquez sur Suivant



la seule option pré-sélectionnée sera disponible, alors cliquez à nouveau sur «Suivant»:



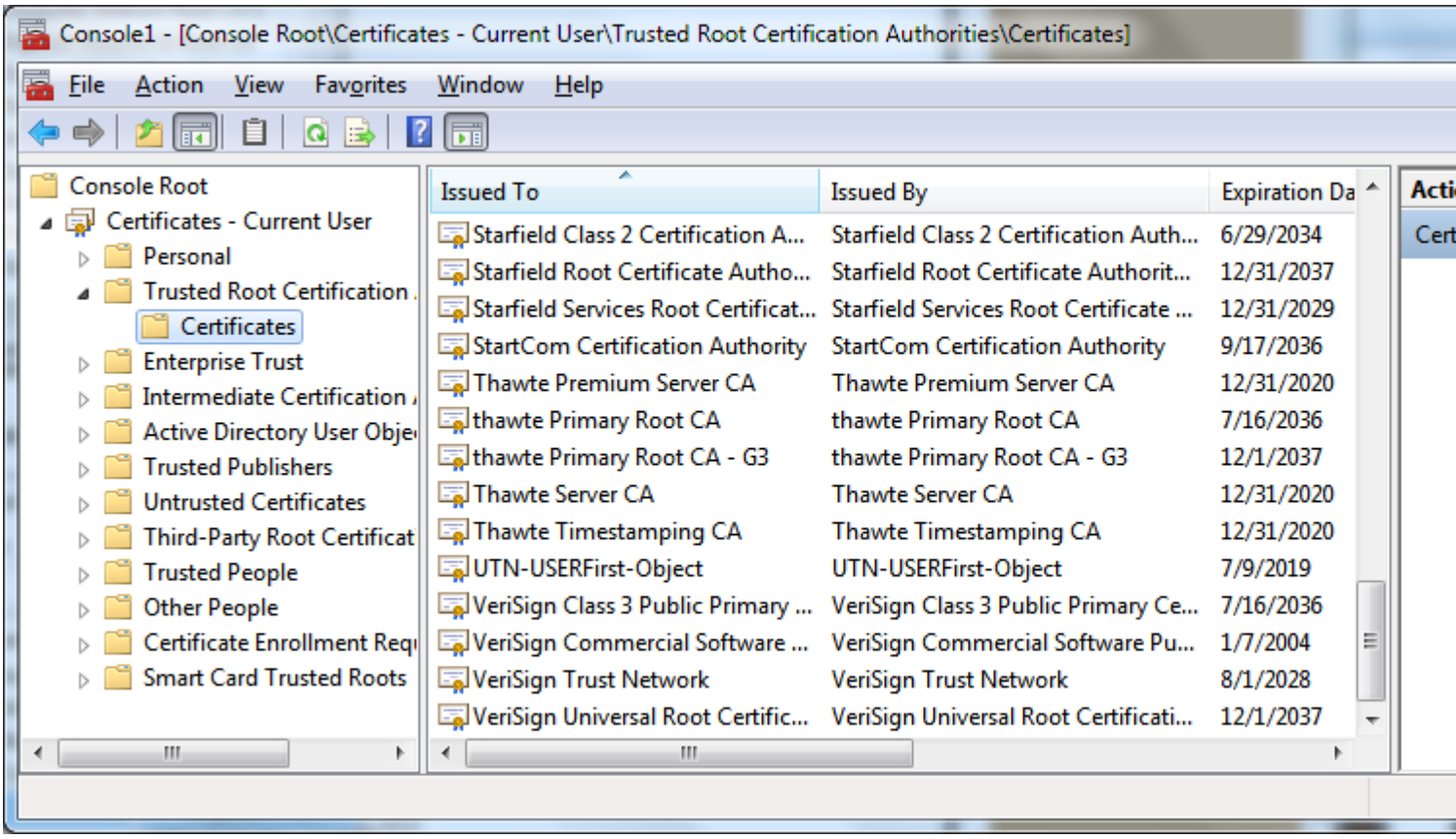
Le premier élément sera déjà présélectionné. Cliquez à nouveau sur Suivant et choisissez un nom et un emplacement pour enregistrer le certificat exporté.



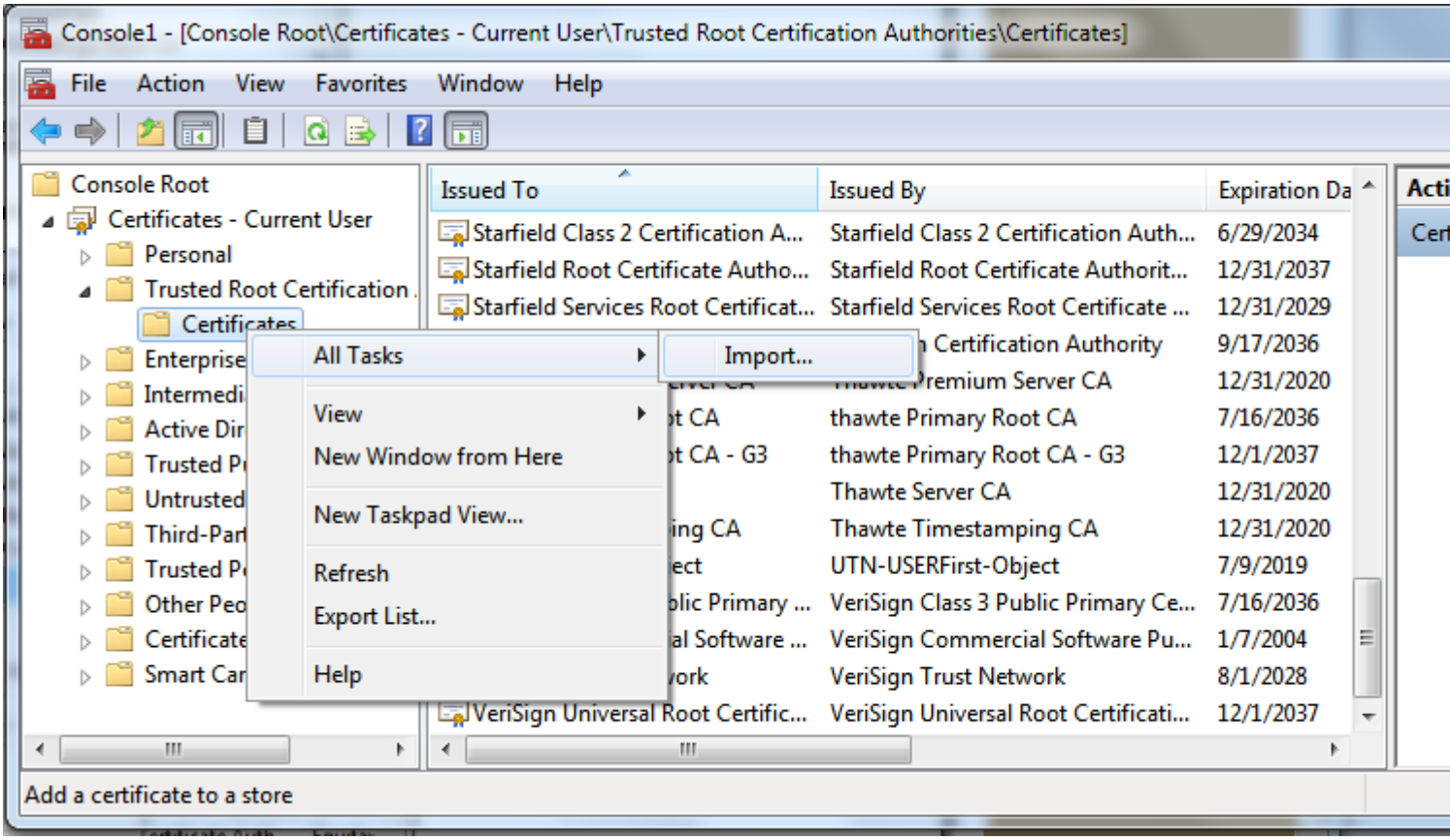
Cliquez à nouveau sur Suivant pour enregistrer le certificat

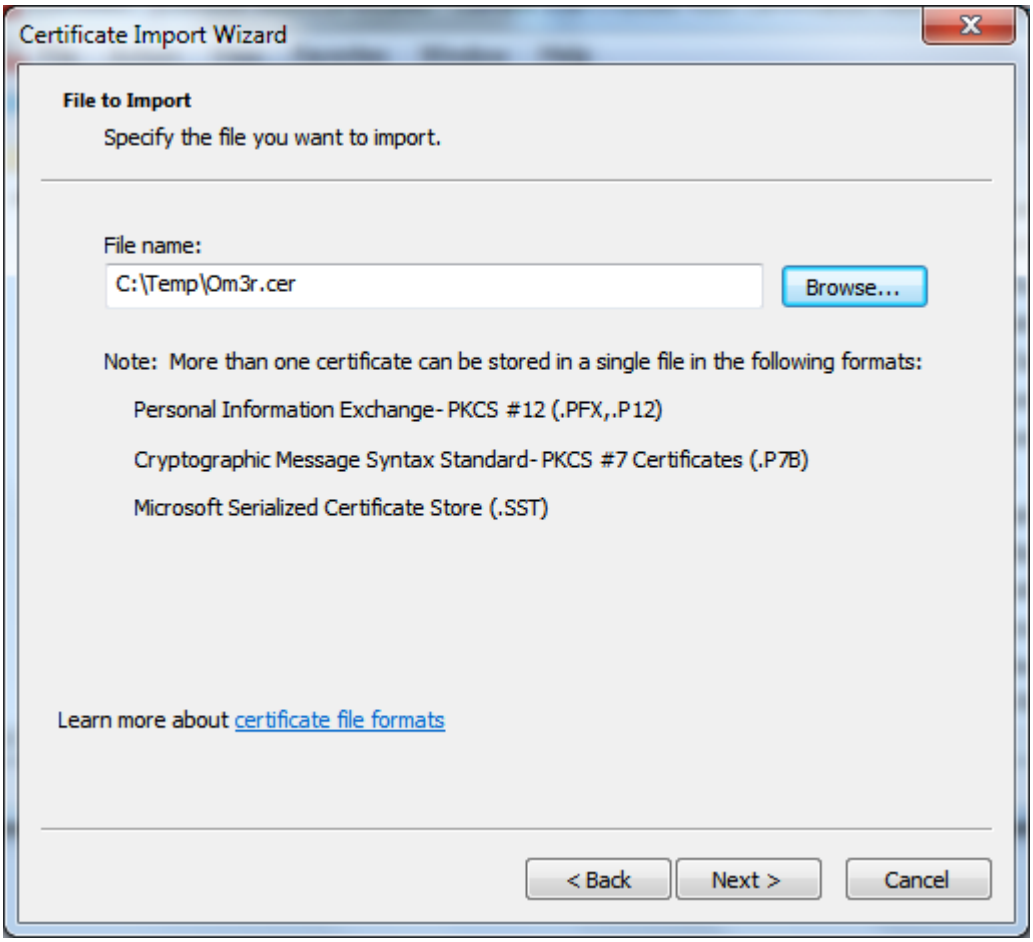
Une fois que le focus est renvoyé à la console de gestion.

Développez le menu *Certificats* et, dans le menu Autorités de certification racines de confiance, sélectionnez *Certificats* .

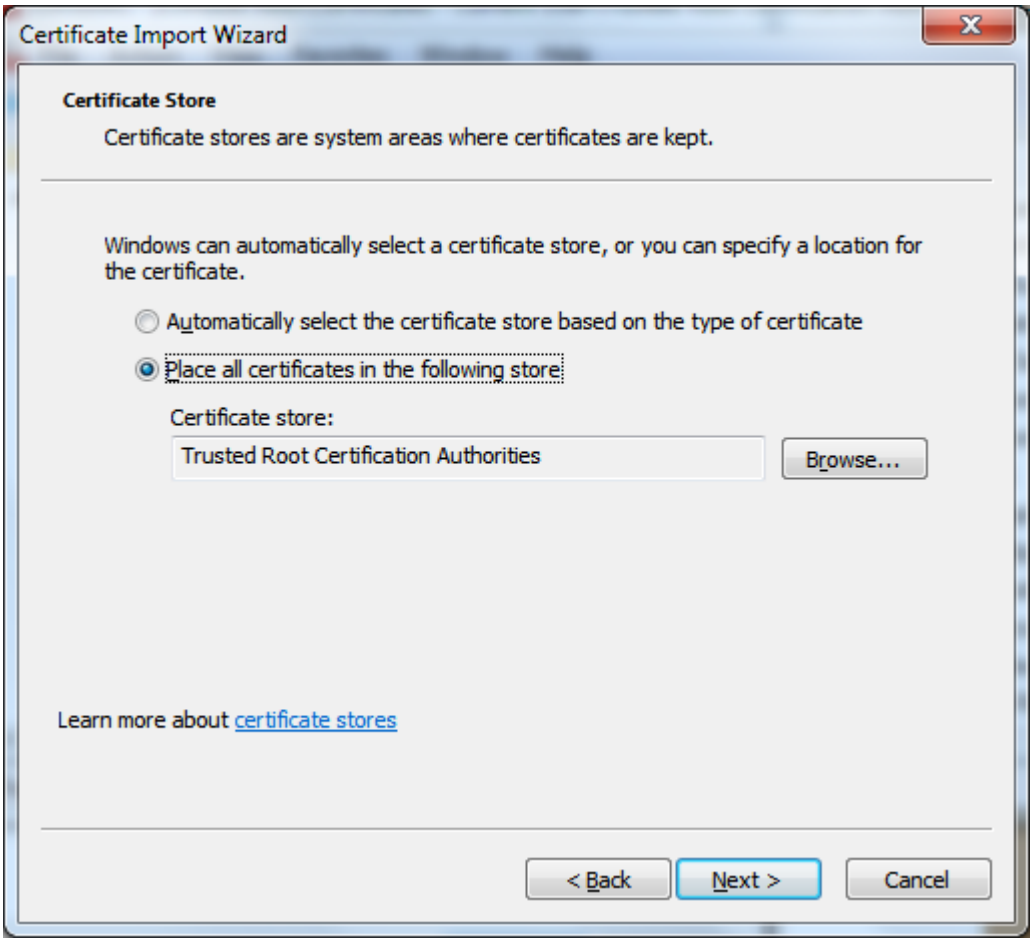


Clic-droit. Sélectionner *toutes les tâches* et *importer*



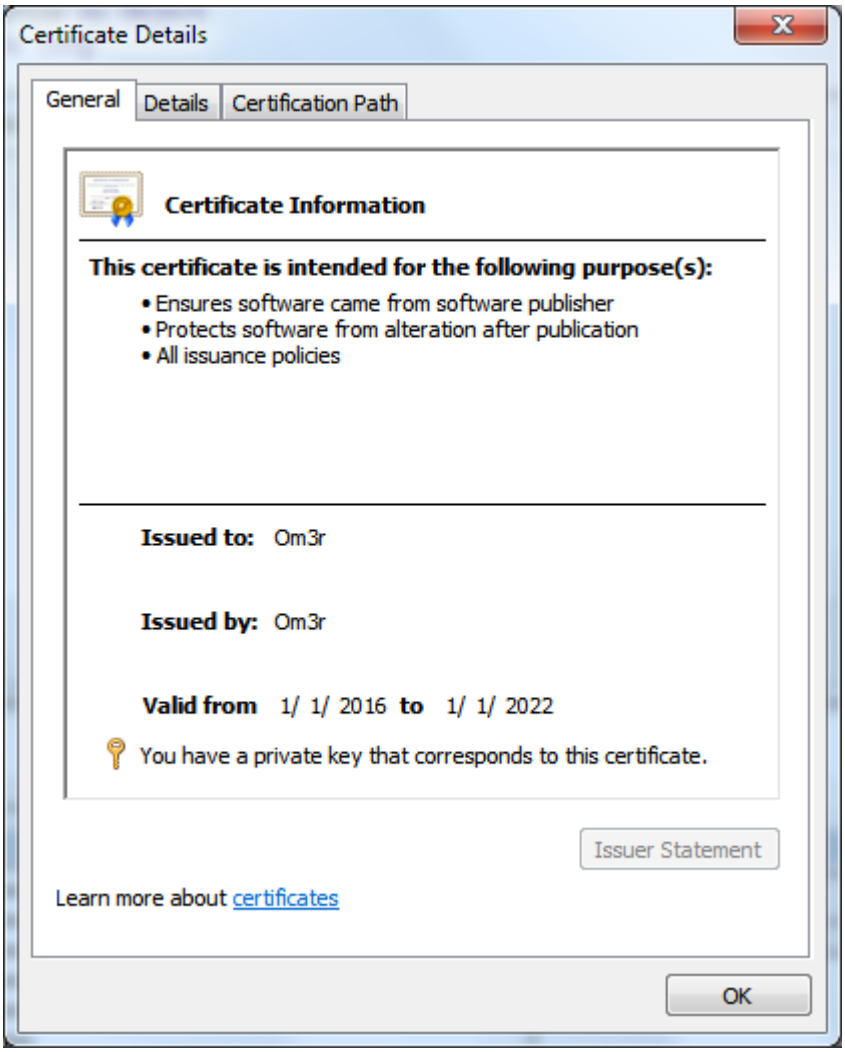


Cliquez sur suivant et sur Enregistrer dans le *magasin des autorités de certification racine de confiance* :



Puis sur Suivant> Terminer, fermez maintenant la console.

Si vous utilisez maintenant le certificat et vérifiez ses propriétés, vous verrez qu'il s'agit d'un certificat de confiance et que vous pouvez l'utiliser pour signer votre projet:



Lire Sécurité des macros et signature des projets / modules VBA en ligne: https://riptutorial.com/fr/vba/topic/7733/securite-des-macros-et-signature-des-projets---modules-vba

Chapitre 38: Sous-supports

Remarques

VBA possède des fonctions intégrées pour extraire des parties spécifiques de chaînes, notamment:

- `Left` / `Left$`
- `Right` / `Right$`
- `Mid` / `Mid$`
- `Trim` / `Trim$`

Pour éviter la conversion de type implicite onverhead (et donc pour de meilleures performances), utilisez la version $ -suffixed de la fonction lorsqu'une variable de chaîne est transmise à la fonction et / ou si le résultat de la fonction est affecté à une variable de chaîne.

Passer une valeur de paramètre `Null` à une fonction $ -suffixed génère une erreur d'exécution ("utilisation non valide de null") - ceci est particulièrement pertinent pour le code impliquant une base de données.

Examples

Utilisez Left ou Left $ pour obtenir les 3 caractères les plus à gauche dans une chaîne

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

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

Utilisez Right ou Right $ pour obtenir les 3 caractères les plus à droite d'une chaîne

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

Utilisez Mid ou Mid $ pour obtenir des caractères spécifiques dans une chaîne

```
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"
```

Utilisez Trim pour obtenir une copie de la chaîne sans espaces de début ou de fin

```
'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"
```

Lire Sous-supports en ligne: https://riptutorial.com/fr/vba/topic/3481/sous-supports

Chapitre 39: Structures de contrôle de flux

Examples

Sélectionner un cas

`Select Case` peut être utilisé lorsque plusieurs conditions différentes sont possibles. Les conditions sont vérifiées de haut en bas et seul le premier cas correspondant sera exécuté.

```
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
```

La logique du bloc `Select Case` peut être inversée pour prendre en charge également le test de différentes variables. Dans ce type de scénario, nous pouvons également utiliser des opérateurs logiques:

```
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
```

Les instructions de cas peuvent également utiliser des opérateurs arithmétiques. Lorsqu'un opérateur arithmétique est utilisé avec la valeur `Select Case` , il doit être précédé du mot `Is` clé `Is` :

```
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
```

Pour chaque boucle

La construction de boucle `For Each` est idéale pour itérer tous les éléments d'une 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
```

Utilisez `For Each` lors de l'itération de collections d'objets:

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

Eviter `For Each` lors de itérations de tableaux; une boucle `For` offrira des performances nettement meilleures avec les tableaux. À l'inverse, une boucle `For Each` offrira de meilleures performances lors de l'itération d'une `Collection` .

Syntaxe

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

Le mot clé `Next` peut éventuellement être suivi de la variable iterator; Cela peut aider à clarifier les boucles imbriquées, bien qu'il existe de meilleurs moyens de clarifier le code imbriqué, tel que l'extraction de la boucle interne dans sa propre procédure.

```
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
```

Faire une boucle

```
 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
```

En boucle

```
'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
```

Pour la boucle

La boucle `For` est utilisée pour répéter la section de code incluse un nombre de fois donné. L'exemple simple suivant illustre la syntaxe de base:

```
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
```

Le code ci-dessus déclare un entier `i` . La boucle `For` attribue à `i` toutes les valeurs comprises entre 1 et 10, puis exécute `Debug.Print i` , c'est-à-dire que le code imprime les numéros 1 à 10 dans la fenêtre immédiate. Notez que la variable de boucle est incrémentée par l'instruction `Next` , c'est-à-dire après l'exécution du code inclus par opposition à avant son exécution.

Par défaut, le compteur sera incrémenté de 1 à chaque exécution de la boucle. Cependant, une `Step` peut être spécifiée pour modifier le montant de l'incrément comme un littéral ou la valeur de retour d'une fonction. Si la valeur de départ, la valeur finale ou la valeur de `Step` est un nombre à virgule flottante, elle sera arrondie à la valeur entière la plus proche. `Step` peut être une valeur positive ou négative.

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

En général, une boucle `For` serait utilisée dans les situations où elle est connue avant que la boucle ne commence à exécuter le code joint (sinon, une boucle `Do` ou `While` peut être plus appropriée). En effet, la condition de sortie est corrigée après la première entrée en boucle,

comme le montre ce code:

```
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
```

Une boucle `For` peut être sortie plus tôt avec l'instruction `Exit For` :

```
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
```

Lire Structures de contrôle de flux en ligne: https://riptutorial.com/fr/vba/topic/1873/structures-de-controle-de-flux

Chapitre 40: Structures de données

Introduction

[TODO: Cette rubrique devrait être un exemple de toutes les structures de données de base du CS 101, accompagnée d'explications expliquant comment les structures de données peuvent être implémentées dans VBA. Ce serait une bonne occasion de lier et de renforcer les concepts introduits dans les rubriques relatives aux classes dans la documentation VBA.]

Examples

Liste liée

Cet exemple de liste chaînée implémente les opérations de type de données abstraites définies .

Classe **SinglyLinkedNode**

```
Option Explicit

Private Value As Variant
Private NextNode As SinglyLinkedNode '"Next" is a keyword in VBA and therefore is not a valid
variable name
```

Classe **LinkedList**

```
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
```

Arbre binaire

Voici un exemple d' arbre de recherche binaire déséquilibré. Un arbre binaire est structuré conceptuellement comme une hiérarchie de nœuds descendant depuis une racine commune, où chaque nœud a deux enfants: gauche et droite. Par exemple, supposons que les nombres 7, 5, 9, 3, 11, 6, 12, 14 et 15 ont été insérés dans un BinaryTree. La structure serait comme ci-dessous. Notez que cet arbre binaire n'est pas équilibré , ce qui peut être une caractéristique souhaitable pour garantir les performances des recherches - voir les arborescences AVL pour un exemple d'arbre de recherche binaire à équilibrage automatique.

```
          7
         / \
        5   9
       / \   \
      3   6   11
                \
                 12
                   \
                    14
                     \
                      15
```

Classe **BinaryTreeNode**

```
Option Explicit

Public left As BinaryTreeNode
Public right As BinaryTreeNode
Public key As Variant
Public value As Variant
```

Classe **BinaryTree**

[FAIRE]

Lire Structures de données en ligne: https://riptutorial.com/fr/vba/topic/8628/structures-de-donnees

Chapitre 41: Tableaux

Examples

Déclaration d'un tableau dans VBA

Déclarer un tableau est très similaire à déclarer une variable, sauf que vous devez déclarer la dimension du tableau juste après son nom:

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

Par défaut, les tableaux de VBA sont **indexés à partir de ZERO** . Le nombre entre parenthèses ne fait donc pas référence à la taille du tableau, mais plutôt à **l'index du dernier élément.**

Accéder aux éléments

L'accès à un élément du tableau se fait en utilisant le nom du tableau, suivi de l'index de l'élément, entre parenthèses:

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

Indexation de tableau

Vous pouvez modifier l'indexation des tableaux en plaçant cette ligne en haut d'un module:

```
Option Base 1
```

Avec cette ligne, tous les tableaux déclarés dans le module seront **indexés depuis ONE** .

Index spécifique

Vous pouvez également déclarer chaque tableau avec son propre index en utilisant le mot clé `To` et les limites inférieure et supérieure (= 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
```

Déclaration dynamique

Lorsque vous ne connaissez pas la taille de votre tableau avant sa déclaration, vous pouvez utiliser la déclaration dynamique et le mot clé `ReDim` :

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

Notez que l'utilisation du mot-clé `ReDim` effacera tout contenu précédent de votre tableau. Pour éviter cela, vous pouvez utiliser le mot-clé `Preserve` après `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
```

Utilisation de Split pour créer un tableau à partir d'une chaîne

Fonction Split

renvoie un tableau à une dimension, basé sur zéro, contenant un nombre spécifié de sous-chaînes.

Syntaxe

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

Partie	La description
expression	Champs obligatoires. Expression de chaîne contenant des sous-chaînes et des délimiteurs. Si *expression* est une chaîne de longueur nulle ("" ou vbNullString), **Split** renvoie un tableau vide ne contenant aucun élément et aucune donnée. Dans ce cas, le tableau renvoyé aura un LBound de 0 et un UBound de -1.
délimiteur	Optionnel. Caractère de chaîne utilisé pour identifier les limites de sous-chaîne. S'il est omis, le caractère d'espace ("") est supposé être le délimiteur. Si le ***délimiteur*** est une chaîne de longueur nulle, un tableau à un seul élément contenant la chaîne d' ***expression*** complète est renvoyé.
limite	Optionnel. Nombre de sous-chaînes à renvoyer -1 indique que toutes les sous-chaînes sont renvoyées.
comparer	Optionnel. Valeur numérique indiquant le type de comparaison à utiliser lors de l'évaluation des sous-chaînes. Voir la section Paramètres pour les valeurs.

Paramètres

L'argument de ***comparaison*** peut avoir les valeurs suivantes:

Constant	Valeur	La description
La description	-1	Effectue une comparaison en utilisant le paramètre de

Constant	Valeur	La description
		l'instruction **Option Compare** .
vbBinaryCompare	0	Effectue une comparaison binaire.
vbTextCompare	1	Effectue une comparaison textuelle.
vbDatabaseCompare	2	Microsoft Access uniquement. Effectue une comparaison basée sur les informations de votre base de données.

Exemple

Dans cet exemple, il est démontré comment Split fonctionne en affichant plusieurs styles. Les commentaires afficheront le jeu de résultats pour chacune des différentes options de Split effectuées. Enfin, il est démontré comment effectuer une boucle sur le tableau de chaînes renvoyé.

```
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
```

Éléments itératifs d'un tableau

Pour ... Suivant

L'utilisation de la variable itérateur comme numéro d'index est le moyen le plus rapide d'itérer les éléments d'un tableau:

```
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
```

Les boucles imbriquées peuvent être utilisées pour itérer des tableaux multidimensionnels:

```
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
```

Pour chacun ... Suivant

Une boucle `For Each...Next` peut également être utilisée pour itérer des tableaux, si les performances importent peu:

```
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
```

Une boucle `For Each` itère toutes les dimensions de l'extérieur vers l'intérieur (le même ordre que les éléments sont mis en mémoire), donc il n'y a pas besoin de boucles imbriquées:

```
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
```

Notez que les boucles `For Each` sont mieux utilisées pour itérer les objets `Collection` , si les performances sont importantes.

Les 4 extraits ci-dessus produisent la même sortie:

```
 0
 1
 2
 3
```

Tableaux dynamiques (redimensionnement de matrice et traitement dynamique)

Tableaux dynamiques

L'ajout et la réduction dynamiques de variables dans un tableau est un avantage considérable lorsque les informations que vous traitez ne comportent pas un nombre défini de variables.

Ajout dynamique de valeurs

Vous pouvez simplement redimensionner le tableau avec l'instruction `ReDim` , cela redimensionnera le tableau, mais si vous souhaitez conserver les informations déjà stockées dans le tableau, vous aurez besoin du composant `Preserve` .

Dans l'exemple ci-dessous, nous créons un tableau et l'augmentons d'une variable supplémentaire dans chaque itération tout en préservant les valeurs déjà présentes dans le tableau.

```
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
```

Suppression dynamique des valeurs

Nous pouvons utiliser la même logique pour diminuer le tableau. Dans l'exemple, la valeur "last" sera supprimée du tableau.

```
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
```

Réinitialisation d'un tableau et réutilisation dynamique

Nous pouvons également réutiliser les tableaux que nous créons pour ne pas en avoir beaucoup en mémoire, ce qui ralentirait le temps d'exécution. Ceci est utile pour les tableaux de différentes tailles. Un extrait que vous pourriez utiliser pour réutiliser le tableau est de `ReDim` le tableau sur `(0)` , d'attribuer une variable au tableau et d'augmenter à nouveau librement le tableau.

Dans l'extrait ci-dessous, je construis un tableau avec les valeurs 1 à 40, vide le tableau et remplit le tableau avec les valeurs 40 à 100, tout cela de manière dynamique.

```
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
```

Tableaux dentelés (tableaux de tableaux)

Tableaux dentelés PAS de tableaux multidimensionnels

Les tableaux de tableaux (Jagged Arrays) ne sont pas les mêmes que les tableaux multidimensionnels si vous les considérez visuellement. Les tableaux multidimensionnels ressemblent à des matrices (rectangulaires) avec un nombre défini d'éléments sur leurs dimensions (tableaux internes). calendrier avec les tableaux intérieurs ayant un nombre différent d'éléments, comme des jours dans des mois différents.

Bien que les baies Jagged soient assez compliquées et délicates à utiliser en raison de leurs niveaux imbriqués et qu'elles n'ont pas beaucoup de sécurité, mais elles sont très flexibles, elles vous permettent de manipuler assez facilement différents types de données et vous n'avez pas besoin de éléments vides.

Créer un tableau dentelé

Dans l'exemple ci-dessous, nous allons initialiser un tableau en escalier contenant deux tableaux l'un pour les noms et l'autre pour les nombres, puis en accédant à un élément de chaque

```
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
```

Création dynamique et lecture de tableaux dentelés

Nous pouvons aussi être plus dynamiques dans notre approche de la construction des baies, imaginez que nous ayons une fiche de données client dans Excel et que nous souhaitons construire un tableau pour afficher les détails du client.

```
   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
```

Nous allons construire dynamiquement un tableau d'en-tête et un tableau Customers, l'en-tête contiendra les titres des colonnes et le tableau Customers contiendra les informations de chaque client / ligne en tant que tableaux.

```
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.

Le résultat de l'extrait ci-dessus est un Jagged Array avec deux tableaux l'un de ces tableaux avec 4 éléments, 2 niveaux d'indentation, et l'autre étant lui-même un autre Jagged Array contenant 5 tableaux de 4 éléments chacun et 3 niveaux d'indentation, voir ci-dessous la 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)
```

Pour accéder aux informations, vous devez tenir compte de la structure de Jagged Array que vous créez. Dans l'exemple ci-dessus, vous pouvez voir que le `Main Array` contient un tableau d'en-`Headers` et un tableau de tableaux (`Customers`). accéder aux éléments.

Nous allons maintenant lire les informations du `Main Array` et imprimer chacune des informations du client en tant que `Info Type: Info` .

```
For n = 0 To UBound(Main_Array(1))
    'n to iterate from fisrt 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
```

N'OUBLIEZ PAS de garder une trace de la structure de votre tableau Jagged, dans l'exemple ci-dessus pour accéder au nom d'un client en accédant à `Main_Array -> Customers -> CustomerNumber -> Name` qui `"Person4"` trois niveaux, pour renvoyer `"Person4"` vous aurez besoin. l'emplacement des clients dans Main_Array, puis l'emplacement du client quatre sur le tableau Customers Jagged et enfin l'emplacement de l'élément dont vous avez besoin, dans ce cas `Main_Array(1)(3)(0)` qui est `Main_Array(Customers)(CustomerNumber)(Name)` .

Tableaux multidimensionnels

Tableaux multidimensionnels

Comme leur nom l'indique, les tableaux multidimensionnels sont des tableaux contenant plus d'une dimension, généralement deux ou trois, mais peuvent comporter jusqu'à 32 dimensions.

Un multi-tableau fonctionne comme une matrice avec différents niveaux, par exemple une comparaison entre une, deux et trois dimensions.

One Dimension est votre tableau typique, il ressemble à une liste d'éléments.

```
Dim 1D(3) as Variant

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

```
(1)
(2)
```

Two Dimensions ressemblerait à une grille de Sudoku ou à une feuille Excel. Lors de l'initialisation du tableau, vous définiriez le nombre de lignes et de colonnes que le tableau aurait.

```
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 commencerait à ressembler à Rubik's Cube. Lors de l'initialisation du tableau, vous définiriez des lignes, des colonnes et des couches / profondeurs.

```
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)
```

D'autres dimensions pourraient être considérées comme la multiplication de la 3D, de sorte qu'un 4D (1,3,3,3) serait deux tableaux 3D côte à côte.

Tableau à deux dimensions

La création

L'exemple ci-dessous sera une compilation d'une liste d'employés, chaque employé aura un ensemble d'informations sur la liste (Prénom, Nom, Adresse, Email, Téléphone ...), l'exemple sera essentiellement stocké sur le tableau (employé, information) étant le (0,0) est le prénom du premier employé.

```
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
```

Redimensionnement

Redimensionnement ou `ReDim Preserve` un multi-tableau comme la norme pour un tableau One-Dimension obtiendrait une erreur, au lieu de cela, les informations doivent être transférées dans un tableau temporaire de la même taille que l'original plus le nombre de lignes / colonnes à ajouter. Dans l'exemple ci-dessous, nous verrons comment initialiser un tableau temporaire, transférer les informations du tableau d'origine, remplir les éléments vides restants et remplacer le tableau temporaire par le tableau d'origine.

```
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
```

Modification des valeurs d'élément

Pour modifier / modifier les valeurs d'un élément donné, il suffit d'appeler la coordonnée pour la modifier et de lui donner une nouvelle valeur: `Employees(0, 0) = "NewValue"`

Vous pouvez également parcourir les conditions d'utilisation des coordonnées pour faire correspondre les valeurs correspondant aux paramètres nécessaires:

```
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
```

En train de lire

L'accès aux éléments du tableau peut être effectué à l'aide d'une boucle imbriquée (itération de chaque élément), d'une boucle et d'une coordonnée (itération des lignes et accès direct aux colonnes) ou d'un accès direct aux deux coordonnées.

```
'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)
```

***Rappelez - vous** , il est toujours utile de garder une carte de tableau lors de l'utilisation de tableaux multidimensionnels, ils peuvent facilement devenir une confusion.*

Tableau à trois dimensions

Pour le tableau 3D, nous utiliserons les mêmes prémisses que le tableau 2D, en plus de stocker l'employé et les informations, mais aussi de construire dans lequel ils travaillent.

Le tableau 3D aura les employés (peuvent être considérés comme des lignes), les informations (colonnes) et le bâtiment qui peuvent être considérés comme des feuilles différentes sur un document Excel, ils ont la même taille entre eux, mais chaque feuille a un ensemble d'informations différent dans ses cellules / éléments. Le tableau 3D contiendra un nombre ***n*** de tableaux 2D.

La création

Un tableau 3D a besoin de 3 coordonnées pour être initialisé `Dim 3Darray(2,5,5) As Variant` la première coordonnée du tableau sera le nombre de feuilles de construction (différents ensembles de lignes et de colonnes), la deuxième coordonnée définira les lignes et la troisième Colonnes. Le `Dim` ci-dessus donnera un tableau 3D avec 108 éléments (`3*6*6`), avec 3 ensembles de tableaux 2D différents.

```
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
```

Redimensionnement

Le redimensionnement d'un tableau 3D est similaire au redimensionnement d'un 2D, créez d'abord un tableau Temporary avec la même taille que l'original en ajoutant un dans la coordonnée du paramètre à augmenter, le premier augmentera le nombre d'ensembles dans le tableau, le second et les troisièmes coordonnées augmenteront le nombre de lignes ou de colonnes dans chaque ensemble.

L'exemple ci-dessous augmente le nombre de lignes dans chaque ensemble par un et remplit ces éléments récemment ajoutés avec de nouvelles informations.

```
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
```

Modification des valeurs des éléments et lecture

La lecture et la modification des éléments du tableau 3D peuvent se faire de la même façon que nous le faisons avec le tableau 2D, il suffit de régler le niveau supplémentaire dans les boucles et les coordonnées.

```
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)
```

Lire Tableaux en ligne: https://riptutorial.com/fr/vba/topic/3064/tableaux

Chapitre 42: Travailler avec ADO

Remarques

Les exemples présentés dans cette rubrique utilisent une liaison anticipée pour plus de clarté et nécessitent une référence à la bibliothèque Microsoft ActiveX Data Object xx. Ils peuvent être convertis en liaison tardive en remplaçant les références fortement typées par `Object` et en remplaçant la création d'objets à l'aide de `New` par `CreateObject` cas échéant.

Examples

Connexion à une source de données

La première étape pour accéder à une source de données via ADO consiste à créer un objet ADO `Connection` . Cela se fait généralement en utilisant une chaîne de connexion pour spécifier les paramètres de la source de données, bien qu'il soit également possible d'ouvrir une connexion DSN en transmettant le DSN, l'ID utilisateur et le mot de passe à la méthode `.Open` .

Notez qu'un DSN n'est pas obligé de se connecter à une source de données via ADO - toute source de données disposant d'un fournisseur ODBC peut être connectée à l'aide de la chaîne de connexion appropriée. Bien que des chaînes de connexion spécifiques à différents fournisseurs soient hors de la portée de cette rubrique, ConnectionStrings.com est une excellente référence pour trouver la chaîne appropriée à votre fournisseur.

```
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
```

Notez que le mot de passe de la base de données n'est inclus dans la chaîne de connexion de l'exemple ci-dessus que pour des raisons de clarté. Les meilleures pratiques exigent de ***ne pas*** stocker les mots de passe de base de données dans le code. Cela peut être accompli en prenant le mot de passe via l'entrée utilisateur ou en utilisant l'authentification Windows.

Récupérer des enregistrements avec une requête

Les requêtes peuvent être exécutées de deux manières, les deux `Recordset` objet ADO `Recordset` qui est une collection de lignes renvoyées. Notez que les deux exemples ci-dessous utilisent la fonction `OpenDatabaseConnection` partir de l'exemple Créer une connexion à une source de données pour des raisons de brièveté. Rappelez-vous que la syntaxe du SQL transmis à la source de données est spécifique au fournisseur.

La première méthode consiste à transmettre l'instruction SQL directement à l'objet Connection et constitue la méthode la plus simple pour exécuter des requêtes simples:

```
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
```

La deuxième méthode consiste à créer un objet de `Command` ADO pour la requête que vous souhaitez exécuter. Cela nécessite un peu plus de code, mais est nécessaire pour pouvoir utiliser des requêtes paramétrées:

```
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
```

Notez que les commandes envoyées à la source de données sont **vulnérables à l'injection SQL** , intentionnelle ou non. En général, les requêtes ne doivent pas être créées en concaténant une entrée utilisateur quelconque. Au lieu de cela, ils doivent être paramétrés (voir Création de commandes paramétrées).

Exécution de fonctions non scalaires

Les connexions ADO peuvent être utilisées pour effectuer à peu près toutes les fonctions de base de données prises en charge par le fournisseur via SQL. Dans ce cas, il n'est pas toujours nécessaire d'utiliser le jeu d' `Recordset` renvoyé par la fonction `Execute` , bien qu'il puisse être utile pour obtenir des affectations de clés après des instructions INSERT avec @@ Identity ou des commandes SQL similaires. Notez que l'exemple ci-dessous utilise la fonction `OpenDatabaseConnection` partir de l'exemple Créer une connexion à une source de données pour des raisons de brièveté.

```
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
```

Notez que les commandes envoyées à la source de données sont **vulnérables à l'injection SQL** , intentionnelle ou non. En général, les instructions SQL ne doivent pas être créées en concaténant une entrée utilisateur quelconque. Au lieu de cela, ils doivent être paramétrés (voir Création de commandes paramétrées).

Création de commandes paramétrées

Chaque fois que SQL exécuté via une connexion ADO doit contenir une entrée utilisateur, il est conseillé de le paramétrer afin de minimiser les risques d'injection SQL. Cette méthode est également plus lisible que les longues concaténations et facilite un code plus robuste et maintenable (en utilisant une fonction qui renvoie un tableau de `Parameter`).

Dans la syntaxe ODBC standard, les paramètres sont donnés `?` "espaces réservés" dans le texte de la requête, puis les paramètres sont ajoutés à la `Command` dans l'ordre dans lequel ils apparaissent dans la requête.

Notez que l'exemple ci-dessous utilise la fonction `OpenDatabaseConnection` partir de la création d'une connexion à une source de données pour plus de concision.

```
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
```

Remarque: L'exemple ci-dessus illustre une instruction UPDATE paramétrée, mais toute instruction SQL peut recevoir des paramètres.

Lire Travailler avec ADO en ligne: https://riptutorial.com/fr/vba/topic/3578/travailler-avec-ado

Chapitre 43: Travailler avec des fichiers et des répertoires sans utiliser FileSystemObject

Remarques

`Scripting.FileSystemObject` est beaucoup plus robuste que les méthodes héritées de cette rubrique. Cela devrait être préféré dans presque tous les cas.

Examples

Déterminer si les dossiers et les fichiers existent

Des dossiers:

Pour déterminer si un fichier existe, transmettez simplement le nom de fichier à la fonction `Dir$` et testez pour voir s'il renvoie un résultat. Notez que `Dir$` prend en charge les caractères génériques, donc pour tester un fichier *spécifique* , le nom de `pathName` transmis doit être testé pour s'assurer qu'il ne les contient pas. L'exemple ci-dessous génère une erreur - si ce n'est pas le comportement souhaité, la fonction peut être modifiée pour renvoyer simplement `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
```

Dossiers (méthode Dir $):

La fonction `Dir$()` peut également être utilisée pour déterminer si un dossier existe en spécifiant la transmission de `vbDirectory` pour le paramètre d' `attributes` facultatif. Dans ce cas, la valeur `pathName` transmise doit se terminer par un séparateur de chemin (`\`), car les *noms de fichiers* correspondants provoqueront des faux positifs. Gardez à l'esprit que les caractères génériques ne sont autorisés qu'après le dernier séparateur de chemin. La fonction exemple ci-dessous génère une erreur d'exécution 52 - "Nom ou numéro de fichier incorrect" si l'entrée contient un caractère générique. Si ce n'est pas le comportement souhaité, décommentez le message d' `On Error Resume Next` en haut de la fonction. Rappelez-vous également que `Dir$` prend en charge les chemins de fichiers relatifs (c.-à `..\Foo\Bar` d `..\Foo\Bar`), ainsi la validité des résultats est garantie tant que le répertoire de travail actuel n'est pas modifié.

```
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
```

Dossiers (méthode ChDir):

L'instruction `ChDir` peut également être utilisée pour tester si un dossier existe. Notez que cette méthode modifie temporairement l'environnement dans lequel s'exécute VBA. Par conséquent, la méthode `Dir$` doit être utilisée à la place. Il a l'avantage d'être beaucoup moins tolérant avec son paramètre. Cette méthode prend également en charge les chemins d'accès relatifs aux fichiers, de même que la méthode `Dir$` .

```
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
```

Création et suppression de dossiers de fichiers

Remarque: par souci de concision, les exemples ci-dessous utilisent la fonction FolderExists de l'exemple **Determining If Folders and Files Exist** de cette rubrique.

L'instruction `MkDir` peut être utilisée pour créer un nouveau dossier. Il accepte les chemins contenant des lettres de lecteur (`C:\Foo`), des noms UNC (`\\Server\Foo`), des chemins relatifs (`..\Foo`) ou le répertoire de travail actuel (`Foo`).

Si le lecteur ou le nom UNC est omis (par exemple, `\Foo`), le dossier est créé sur le lecteur en cours. Cela peut être ou ne pas être le même lecteur que le répertoire de travail en cours.

```
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
```

L'instruction `RmDir` peut être utilisée pour supprimer des dossiers existants. Il accepte les chemins

sous les mêmes formes que `MkDir` et utilise la même relation avec le répertoire de travail et le lecteur en cours. Notez que cette instruction est similaire à la commande shell Windows `rd` , elle lancera donc une erreur d'exécution 75: "Erreur d'accès au fichier / chemin" si le répertoire cible n'est pas vide.

```
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
```

Lire Travailler avec des fichiers et des répertoires sans utiliser FileSystemObject en ligne: https://riptutorial.com/fr/vba/topic/5706/travailler-avec-des-fichiers-et-des-repertoires-sans-utiliser-filesystemobject

Chapitre 44: Tri

Introduction

Contrairement à la structure .NET, la bibliothèque Visual Basic pour Applications n'inclut pas de routines pour trier les tableaux.

Il existe deux types de solutions: 1) implémenter un algorithme de tri à partir de zéro ou 2) utiliser des routines de tri dans d'autres bibliothèques couramment disponibles.

Examples

Implémentation d'algorithme - Tri rapide sur un tableau monodimensionnel

De la fonction de tri du tableau VBA?

```
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
```

Utilisation de la bibliothèque Excel pour trier un tableau à une dimension

Ce code tire parti de la classe `Sort` dans la bibliothèque d'objets Microsoft Excel.

Pour plus de lecture, voir:

- Copier une plage dans une plage virtuelle
- Comment copier la plage sélectionnée dans un tableau donné?

```
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
```

Lire Tri en ligne: https://riptutorial.com/fr/vba/topic/8836/tri

Chapitre 45: Types de données et limites

Examples

Octet

```
Dim Value As Byte
```

Un octet est un type de données non signé de 8 bits. Il peut représenter des nombres entiers compris entre 0 et 255 et tenter de stocker une valeur en dehors de cette plage entraîne l' erreur d'exécution 6: `Overflow` . Byte est le seul type non signé intrinsèque disponible dans VBA.

La fonction de conversion en un octet est `CByte()` . Pour les moulages à partir de types à virgule flottante, le résultat est arrondi à la valeur entière la plus proche, avec un arrondi de 0,5.

Byte Arrays et Strings

Les chaînes et les tableaux d'octets peuvent être remplacés par une simple affectation (aucune fonction de conversion nécessaire).

Par exemple:

```
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
```

Pour pouvoir encoder des caractères Unicode , chaque caractère de la chaîne occupe deux octets dans le tableau, l'octet le moins significatif étant le premier. Par exemple:

```
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
```

Entier

```
Dim Value As Integer
```

Un entier est un type de données signé 16 bits. Il peut stocker des nombres entiers compris entre -32 768 et 32 767 et toute tentative de stockage d'une valeur en dehors de cette plage entraîne l'erreur d'exécution 6: dépassement de capacité.

Les entiers sont stockés en mémoire sous forme de valeurs little-endian avec des négatifs représentés par un complément à deux .

Notez qu'en général, il est préférable d'utiliser un long plutôt qu'un entier à moins que le plus petit type soit membre d'un type ou qu'il soit requis (par une convention d'appel API ou pour toute autre raison) pour qu'il soit 2 octets. Dans la plupart des cas, VBA traite les entiers en 32 bits en interne, ce qui signifie qu'il n'ya généralement aucun avantage à utiliser un type plus petit. De plus, une pénalité de performance est encourue chaque fois qu'un type Integer est utilisé car il est lancé silencieusement en tant que Long.

La fonction de conversion en un entier est `CInt()` . Pour les moulages à partir de types à virgule flottante, le résultat est arrondi à la valeur entière la plus proche, avec un arrondi de 0,5.

Booléen

```
Dim Value As Boolean
```

Un booléen permet de stocker des valeurs pouvant être représentées par True ou False. En interne, le type de données est stocké sous forme de valeur 16 bits avec 0 représentant la valeur False et toute autre valeur représentant True.

Il convient de noter que lorsqu'un booléen est converti en un type numérique, tous les bits sont définis sur 1. Cela se traduit par une représentation interne de -1 pour les types signés et la valeur maximale pour un type non signé (octet).

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

La fonction de conversion pour convertir en booléen est `CBool()` . Même s'il est représenté en interne sous la forme d'un nombre à 16 bits, la conversion en booléen à partir de valeurs situées en dehors de cette plage est protégée contre le débordement, bien qu'il définisse tous les 16 bits sur 1:

```
Dim Example As Boolean
```

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

Longue

```
Dim Value As Long
```

Un long est un type de données 32 bits signé. Il peut stocker des nombres entiers compris entre -2 147 483 648 et 2 147 483 647, et tenter de stocker une valeur en dehors de cette plage entraînera une erreur d'exécution 6: dépassement de capacité.

Les longues sont stockées en mémoire sous forme de valeurs peu endiennes avec des négatifs représentés par un complément à deux .

Notez que comme un Long correspond à la largeur d'un pointeur dans un système d'exploitation 32 bits, les Longs sont couramment utilisés pour stocker et transmettre des pointeurs vers et depuis des fonctions API.

La fonction de conversion pour convertir en un long est `CLng()` . Pour les moulages à partir de types à virgule flottante, le résultat est arrondi à la valeur entière la plus proche, avec un arrondi de 0,5.

Unique

```
Dim Value As Single
```

Un Single est un type de données à virgule flottante 32 bits signé. Il est stocké en interne en utilisant une disposition de mémoire IEEE 754 peu endian . En tant que tel, il n'y a pas de plage de valeurs fixe pouvant être représentée par le type de données - ce qui est limité est la précision de la valeur stockée. Un seul peut stocker une valeur ***entière de*** valeurs comprise entre -16 777 216 et 16 777 216 sans perte de précision. La précision des nombres à virgule flottante dépend de l'exposant.

Un seul dépassera si une valeur supérieure à environ 2^{128} est attribuée. Il ne débordera pas d'exposants négatifs, bien que la précision utilisable soit discutable avant que la limite supérieure soit atteinte.

Comme avec tous les nombres à virgule flottante, il faut faire attention lors des comparaisons d'égalité. La meilleure pratique consiste à inclure une valeur delta adaptée à la précision requise.

La fonction de conversion en un seul est `CSng()` .

Double

```
Dim Value As Double
```

Un double est un type de données à virgule flottante 64 bits signé. Comme le Single , il est stocké en interne à l'aide d'une mémoire IEEE 754 peu volumineuse et les mêmes précautions concernant la précision doivent être prises. Un double peut stocker ***des*** valeurs ***entières*** comprises entre -9 007 199 et 254 740 992 et 9 007 199 254 740 992 sans perte de précision. La précision des nombres à virgule flottante dépend de l'exposant.

Un Double débordera si une valeur supérieure à environ 2 ^{1024} lui est attribuée. Il ne débordera pas d'exposants négatifs, bien que la précision utilisable soit discutable avant que la limite supérieure soit atteinte.

La fonction de conversion en double est `CDbl()` .

Devise

```
Dim Value As Currency
```

Une devise est un type de données en virgule flottante de 64 bits signé similaire à un double , mais mis à l'échelle de 10 000 pour donner une plus grande précision aux 4 chiffres à droite du séparateur décimal. Une variable de devise peut stocker des valeurs de -922 337 203 685 477,5808 à 922 337 203 685 477,5807, ce qui lui donne la plus grande capacité de tout type intrinsèque dans une application 32 bits. Comme le nom du type de données l'indique, il est recommandé d'utiliser ce type de données lors de la représentation des calculs monétaires, car la mise à l'échelle permet d'éviter les erreurs d'arrondi.

La fonction de conversion en une devise est `CCur()` .

Rendez-vous amoureux

```
Dim Value As Date
```

Un type de date est représentée en interne comme un type de données à virgule flottante signé 64 bits avec la valeur à la gauche de la décimale représentant le nombre de jours à compter de la date de l' époque du 30 Décembre 1899 (bien que voir la note ci - dessous). La valeur à droite de la décimale représente l'heure sous forme de jour fractionnaire. Ainsi, un entier Date aurait un composant temps de 12:00:00 AM et x.5 aurait un composant heure de 12:00:00 PM.

Les valeurs valides pour les dates sont entre le 1 ^{er} 100 et 31 Décembre 9999. Janvier Puisqu'une double a une portée plus grande, il est possible de déborder une date en attribuant des valeurs en dehors de cette plage.

En tant que tel, il peut être utilisé indifféremment avec un calcul Double pour la date:

```
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.
```

La fonction de conversion à convertir en Date est `CDate()` , qui accepte toute représentation par date / heure de type chaîne numérique. Il est important de noter que les représentations sous forme de chaîne des dates seront converties en fonction des paramètres régionaux en cours d'utilisation. Par conséquent, les distributions directes doivent être évitées si le code est censé être portable.

Chaîne

Une chaîne représente une séquence de caractères et se présente sous deux formes:

Longueur variable

```
Dim Value As String
```

Une chaîne de longueur variable permet l'ajout et la troncature et est stockée en mémoire en tant que COM BSTR . Cela consiste en un entier non signé de 4 octets qui stocke la longueur de la chaîne en octets, suivie des données de la chaîne elle-même sous la forme de caractères larges (2 octets par caractère) et se termine par 2 octets nuls. Ainsi, la longueur maximale de chaîne pouvant être gérée par VBA est de 2 147 483 647 caractères.

Le pointeur interne à la structure (récupérable par la fonction `StrPtr()`) pointe sur l'emplacement mémoire des *données* et non sur le préfixe de longueur. Cela signifie qu'une chaîne VBA peut être transmise directement aux fonctions API nécessitant un pointeur sur un tableau de caractères.

Étant donné que la longueur peut changer, VBA réaffecte la mémoire pour une chaîne à *chaque affectation de la variable* , ce qui peut entraîner des pénalités de performance pour les procédures qui les modifient de manière répétée.

Longueur fixe

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

Les chaînes de longueur fixe se voient attribuer 2 octets pour chaque caractère et sont stockées en mémoire sous la forme d'un tableau d'octets simple. Une fois allouée, la longueur de la chaîne est immuable. Ils ***ne*** sont ***pas*** terminés par la valeur NULL en mémoire, de sorte qu'une chaîne remplissant la mémoire allouée avec des caractères non nuls ne convient pas pour être transmise aux fonctions de l'API qui attendent une chaîne terminée par un caractère nul.

Les chaînes de longueur fixe transmettent une limitation d'index 16 bits héritée, elles ne peuvent donc comporter que 65 535 caractères. Tenter d'attribuer une valeur plus longue que l'espace mémoire disponible n'entraînera pas d'erreur d'exécution. La valeur résultante sera simplement tronquée:

```
Dim Foobar As String * 5
Foobar = "Foo" & "bar"
Debug.Print Foobar          'Prints "Fooba"
```

La fonction de conversion pour convertir en une chaîne de l'un ou l'autre type est `CStr()` .

LongLong

```
Dim Value As LongLong
```

Un LongLong est un type de données 64 bits signé et n'est disponible que dans les applications 64 bits. Il n'est **pas** disponible dans les applications 32 bits fonctionnant sur des systèmes d'exploitation 64 bits. Il peut stocker des valeurs entières comprises entre -9,223,372,036,854,775,808 et 9,223,372,036,854,775,807 et tenter de stocker une valeur en dehors de cette plage entraînera une erreur d'exécution 6: dépassement de capacité.

Les LongLongs sont stockés en mémoire sous forme de valeurs little-endian avec des négatifs représentés par un complément à deux .

Le type de données LongLong a été introduit dans le cadre de la prise en charge du système d'exploitation 64 bits de VBA. Dans les applications 64 bits, cette valeur peut être utilisée pour stocker et transmettre des pointeurs à des API 64 bits.

La fonction de conversion pour convertir en LongLong est `CLngLng()` . Pour les moulages à partir de types à virgule flottante, le résultat est arrondi à la valeur entière la plus proche, avec un arrondi de 0,5.

Une variante

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

Un Variant est un type de données COM utilisé pour stocker et échanger des valeurs de types arbitraires, et tout autre type dans VBA peut être affecté à un Variant. Variables déclarées sans type explicite spécifié par `As [Type]` par défaut à Variant.

Les variantes sont stockées en mémoire sous la forme d'une structure VARIANT constituée d'un descripteur de type d'octet (VARTYPE) suivi de 6 octets réservés puis d'une zone de données de 8 octets. Pour les types numériques (y compris Date et Boolean), la valeur sous-jacente est stockée dans le Variant lui-même. Pour tous les autres types, la zone de données contient un pointeur sur la valeur sous-jacente.

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

Le type sous-jacent d'un Variant peut être déterminé avec la fonction `VarType()` qui renvoie la valeur numérique stockée dans le descripteur de type ou la fonction `TypeName()` qui renvoie la représentation sous forme de chaîne:

```
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"
```

Comme les variantes peuvent stocker des valeurs de n'importe quel type, les affectations de littéraux sans indications de type seront implicitement converties en un variant du type approprié, conformément au tableau ci-dessous. Les littéraux avec des indications de type seront convertis en une variante du type suggéré.

Valeur	Type résultant
Valeurs de chaîne	Chaîne
Nombres non flottants dans la plage entière	Entier
Nombres non flottants dans la plage longue	Longue
Nombre non flottant en dehors de Long range	Double
Tous les nombres à virgule flottante	Double

Remarque: Sauf s'il existe une raison spécifique d'utiliser un variant (c'est-à-dire un itérateur dans une boucle For Each ou une exigence API), le type doit généralement être évité pour les tâches de routine pour les raisons suivantes:

- Ils ne sont pas sécurisés, ce qui augmente la possibilité d'erreurs d'exécution. Par exemple, un Variant contenant une valeur Integer changera silencieusement en Long au lieu de déborder.
- Ils introduisent une surcharge de traitement en exigeant au moins une déréférence de pointeur supplémentaire.
- La mémoire requise pour un variant est toujours supérieure d' ***au moins*** 8 octets à celle nécessaire pour stocker le type sous-jacent.

La fonction de conversion en une variante est `CVar()` .

LongPtr

```
Dim Value As LongPtr
```

Le LongPtr a été introduit dans VBA afin de prendre en charge les plates-formes 64 bits. Sur un système 32 bits, il est traité comme un Long et sur les systèmes 64 bits, il est traité comme un LongLong .

Son utilisation principale consiste à fournir un moyen portable de stocker et de transmettre des pointeurs sur les deux architectures (voir Modification du comportement du code au moment de la compilation) .

Bien qu'il soit traité par le système d'exploitation comme une adresse mémoire lorsqu'il est utilisé dans les appels d'API, il convient de noter que VBA le traite comme un type signé (et donc sujet à un débordement signé ou non signé). Pour cette raison, toute arithmétique de pointeur effectuée à l'aide de LongPtrs ne doit pas utiliser `>` ou `<` comparaisons. Cette «bizarrerie» permet également d'ajouter des décalages simples pointant vers des adresses valides en mémoire, ce qui peut entraîner des erreurs de dépassement, il faut donc être prudent lorsque vous utilisez des pointeurs dans VBA.

La fonction de conversion pour convertir en LongPtr est `CLngPtr()` . Pour les conversions à partir de types à virgule flottante, le résultat est arrondi à la valeur entière la plus proche, avec un arrondi supérieur à 0,5 (bien qu'il s'agisse généralement d'une adresse mémoire, son utilisation au mieux est dangereuse).

Décimal

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

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

Le type de données `Decimal` est *uniquement* disponible en tant que sous-type de `Variant` . Vous devez donc déclarer toute variable devant contenir un `Decimal` tant que `Variant` , *puis* attribuer une valeur `Decimal` à l'aide de la fonction `CDec` . Le mot-clé `Decimal` est un mot réservé (ce qui suggère que VBA allait éventuellement ajouter un support de première classe pour le type), de sorte que `Decimal` ne peut pas être utilisé comme nom de variable ou de procédure.

Le type `Decimal` nécessite 14 octets de mémoire (en plus des octets requis par le variant parent) et peut stocker des nombres comportant jusqu'à 28 décimales. Pour les nombres sans aucune décimale, la plage des valeurs autorisées est de -79 228 162 162 514 264 333 593 543 950 335 à +79 228 162 514 514 264 inclus. Pour les nombres ayant au maximum 28 décimales, la plage de valeurs autorisées est comprise entre -7,9228162514264337593543950335 et +7,9228162514264337593543950335 inclus.

Lire Types de données et limites en ligne: https://riptutorial.com/fr/vba/topic/3418/types-de-donnees-et-limites

Chapitre 46: VBA orienté objet

Examples

Abstraction

Les niveaux d'abstraction aident à déterminer quand diviser les choses.

L'abstraction est obtenue en implémentant des fonctionnalités avec un code de plus en plus détaillé. Le point d'entrée d'une macro doit être une petite procédure avec un *niveau d'abstraction élevé* qui facilite la compréhension en un coup d'œil de ce qui se passe:

```
Public Sub DoSomething()
    With New SomeForm
        Set .Model = CreateViewModel
        .Show vbModal
        If .IsCancelled Then Exit Sub
        ProcessUserData .Model
    End With
End Sub
```

La procédure `DoSomething` a un niveau d' *abstraction élevé* : on peut dire qu'elle affiche un formulaire et crée un modèle, et en passant cet objet à une procédure `ProcessUserData` qui sait quoi en faire - comment le modèle est créé est le travail d'une autre procédure:

```
Private Function CreateViewModel() As ISomeModel
    Dim result As ISomeModel
    Set result = SomeModel.Create(Now, Environ$("UserName"))
    result.AvailableItems = GetAvailableItems
    Set CreateViewModel = result
End Function
```

La fonction `CreateViewModel` est uniquement responsable de la création d'une instance `ISomeModel` . Une partie de cette responsabilité consiste à acquérir un tableau d' *éléments disponibles* - la manière dont ces éléments sont acquis est un détail d'implémentation qui est résumé derrière la procédure `GetAvailableItems` :

```
Private Function GetAvailableItems() As Variant
    GetAvailableItems = DataSheet.Names("AvailableItems").RefersToRange
End Function
```

Ici, la procédure lit les valeurs disponibles d'une plage nommée sur une feuille de calcul `DataSheet` . Cela pourrait tout aussi bien être de les lire à partir d'une base de données ou les valeurs pourraient être codées en dur: c'est un *détail d'implémentation* qui ne concerne aucun des niveaux d'abstraction les plus élevés.

Encapsulation

L'encapsulation masque les détails d'implémentation du code client.

L'exemple Handling QueryClose illustre l'encapsulation: le formulaire a un contrôle de case à cocher, mais son code client ne fonctionne pas directement - la case à cocher est un *détail d'implémentation. Le* code client doit savoir si le paramètre est activé ou non.

Lorsque la valeur de la case à cocher change, le gestionnaire affecte un membre de champ privé:

```
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
```

Et lorsque le code client veut lire cette valeur, il n'a pas à se soucier d'une case à cocher - au lieu de cela, il utilise simplement la propriété `SomeOtherSetting` :

```
Public Property Get SomeOtherSetting() As Boolean
    SomeOtherSetting = this.SomeOtherSetting
End Property
```

La propriété `SomeOtherSetting` *encapsule* l'état de la case à cocher; le code client n'a pas besoin de savoir qu'il y a une case à cocher, mais seulement qu'il y a un paramètre avec une valeur booléenne. En *encapsulant* la valeur `Boolean` , nous avons ajouté une *couche d'abstraction* autour de la case à cocher.

Utiliser des interfaces pour renforcer l'immuabilité

Allons plus loin en *encapsulant* le *modèle* du formulaire dans un module de classe dédié. Mais si nous avons fait une `Public Property` pour la `UserName` et `Timestamp` , nous devons exposer la `Property Let` accesseurs, ce qui rend les propriétés mutable, et nous ne voulons pas le code client d'avoir la possibilité de changer ces valeurs après ils sont mis.

La fonction `CreateViewModel` dans l'exemple **Abstraction** renvoie une classe `ISomeModel` : c'est notre *interface* , et elle ressemble à ceci:

```
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
```

Notez que les propriétés `Timestamp` et `UserName` exposent uniquement un accesseur `Property Get` . Maintenant, la classe `SomeModel` peut implémenter cette 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
```

Les membres de l'interface sont tous `Private` , et tous les membres de l'interface doivent être implémentés pour que le code puisse être compilé. Les membres `Public` ne font pas partie de l'interface et ne sont donc pas exposés au code écrit sur l'interface `ISomeModel` .

Utilisation d'une méthode d'usine pour simuler un constructeur

En utilisant un attribut VB_PredeclaredId , nous pouvons faire en sorte que la classe `SomeModel` ait une *instance par défaut* et écrire une fonction qui fonctionne comme un membre de type niveau (`Shared` en VB.NET, `static` en C #) que le code client peut appeler sans avoir à créer au préalable une instance, comme nous l'avons fait ici:

```
Private Function CreateViewModel() As ISomeModel
    Dim result As ISomeModel
    Set result = SomeModel.Create(Now, Environ$("UserName"))
    result.AvailableItems = GetAvailableItems
    Set CreateViewModel = result
End Function
```

Cette *méthode de fabrique* attribue les valeurs de propriété en lecture seule lorsque vous y `ISomeModel` depuis l'interface `ISomeModel` , ici `Timestamp` et `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
```

Et maintenant, nous pouvons coder sur l'interface `ISomeModel` , qui expose `Timestamp` et `UserName` tant que propriétés en lecture seule qui ne peuvent jamais être réaffectées (tant que le code est écrit sur l'interface).

Polymorphisme

Le polymorphisme est la capacité de présenter la même interface pour différentes implémentations sous-jacentes.

La possibilité d'implémenter des interfaces permet de découpler complètement la logique d'application de l'interface utilisateur, de la base de données ou de telle ou telle feuille de travail.

Disons que vous avez une interface `ISomeView` que le formulaire implémente lui-même:

```
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
```

Le code-behind du formulaire pourrait ressembler à ceci:

```
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
```

Mais alors, rien n'interdit de créer un autre module de classe qui implémente l'interface `ISomeView` *sans être un formulaire utilisateur* - cela pourrait être une classe `SomeViewMock` :

```
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
```

Et maintenant, nous pouvons changer le code qui fonctionne avec un objet `UserForm` et le faire fonctionner depuis l'interface `ISomeView` , par exemple en lui donnant le formulaire en tant que paramètre au lieu de l'instancier:

```
Public Sub DoSomething(ByVal view As ISomeView)
    With view
        Set .Model = CreateViewModel
        .Show
        If .IsCancelled Then Exit Sub
        ProcessUserData .Model
    End With
End Sub
```

Parce que la `DoSomething` méthode dépend d'une interface (par exemple une *abstraction)* et non pas une *classe concrète* (par exemple un spécifique `UserForm`), nous pouvons écrire un test unitaire automatisé qui assure `ProcessUserData` n'est pas exécutée lorsque `view.IsCancelled` est `True` , en faisant notre test créer une instance `SomeViewMock` , en définissant sa propriété `IsCancelled` sur `True` , et en la transmettant à `DoSomething` .

Le code testable dépend des abstractions

L'écriture de tests unitaires dans VBA peut être effectuée, il y a des compléments là-bas qui l'intègrent même dans l'EDI. Mais lorsque le code est *étroitement couplé* avec une feuille de calcul, une base de données, une forme, ou le système de fichiers, le test unitaire commence nécessitant une feuille de calcul réelle, base de données, sous forme ou système de fichiers - et ces *dépendances* sont nouvel échec hors de contrôle les points que le code testable doit isoler, de sorte que les tests unitaires *ne* nécessitent *pas* une feuille de calcul, une base de données, un formulaire ou un système de fichiers réel.

En écrivant du code sur des interfaces, de manière à ce que le code de test *injecte* des

implémentations stub / mock (comme l'exemple `SomeViewMock` ci-dessus), vous pouvez écrire des tests dans un "environnement contrôlé" et simuler ce qui se passe lorsque permutations des interactions de l'utilisateur sur les données du formulaire, sans même afficher un formulaire et cliquer manuellement sur un contrôle de formulaire.

Lire VBA orienté objet en ligne: https://riptutorial.com/fr/vba/topic/5357/vba-oriente-objet

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec VBA	0m3r, Andre Terra, Benno Grimm, Bookeater, Comintern, Community, Derpcode, Kaz, lfrandom, litelite, Maarten van Stam, Macro Man, Máté Juhász, Nick Dewitt, PankajKushwaha, RubberDuck, Stefan Pinnow
2	Appels API	paul bica
3	Appels de procédure	Macro Man, Mat's Mug, Neil Mussett, Sam Johnson
4	Arguments de passage ByRef ou ByVal	Branislav Kollár, Comintern, Mat's Mug, R3uK, RamenChef, ZygD
5	Assigner des chaînes avec des caractères répétés	ThunderFrame
6	Automatisation ou utilisation d'autres bibliothèques	Branislav Kollár
7	Caractères non latins	Neil Mussett
8	Chaînes concaténantes	ThunderFrame
9	Collections	Comintern
10	commentaires	Comintern, Hosch250, Johnny C, litelite, Macro Man, Nijin22, Shawn V. Wilson, ThunderFrame
11	Compilation conditionnelle	Macro Man, Mat's Mug, RubberDuck, Steve Rindsberg
12	Conventions de nommage	FreeMan, Kaz, Mat's Mug, Victor Moraes
13	Conversion d'autres types en chaînes	ThunderFrame
14	Copier, retourner et passer des tableaux	Mark.R
15	CreateObject vs. GetObject	Branislav Kollár, Dave, Tim
16	Créer une classe personnalisée	Branislav Kollár, Macro Man, Mat's Mug, Neil Mussett, ThunderFrame

17	Créer une procédure	Comintern, LiamH, Mat's Mug, Sivaprasath Vadivel, Tomas Zubiri
18	Date Manipulation	Comintern, FreeMan, Thomas G
19	Déclaration des variables	Comintern, dadde, Dave, Franck Dernoncourt, Jeeped, Kaz, lfrandom, litelite, Macro Man, Mark.R, Mat's Mug, Neil Mussett, RubberDuck, Shawn V. Wilson, SWa, Thierry Dalon, ThunderFrame, Tom, Victor Moraes, Zaider
20	Déclarer et assigner des chaînes	Comintern, ThunderFrame
21	Erreurs d'exécution VBA	Branislav Kollár, Macro Man, Mat's Mug
22	Événements	Mat's Mug
23	Formulaires utilisateur	Mat's Mug
24	Interfaces	Neil Mussett
25	La gestion des erreurs	Comintern, Logan Reed, Mat's Mug
26	Lecture de 2 Go + fichiers en binaire dans VBA et File Hashes	PatricK
27	Les attributs	hymced, Mat's Mug, RamenChef, RubberDuck
28	Les opérateurs	Comintern, Macro Man
29	Littéraux de chaîne - Fuites, caractères non imprimables et continuations de ligne	Comintern, ThunderFrame
30	Manipulation de chaîne fréquemment utilisée	pashute
31	Mesurer la longueur des cordes	Steve Rindsberg, ThunderFrame
32	Mot-clé VBA Option	Jeeped, Maarten van Stam, Macro Man, Mat's Mug, RamenChef, RubberDuck, Stefan Pinnow, Thomas G, ThunderFrame
33	Objet Scripting.Dictionary	Comintern, Jeeped, Kyle, RamenChef, Tim, Wolf, Zev Spitz
34	Rechercher dans les	ThunderFrame

	chaînes la présence de sous-chaînes	
35	Récursivité	Mat's Mug, ThunderFrame
36	Scripting.FileSystemObject	Comintern, Dave, Macro Man, Mikegrann, RubberDuck, Siva, Steve Rindsberg, ThunderFrame
37	Sécurité des macros et signature des projets / modules VBA	0m3r
38	Sous-supports	Mat's Mug, ThunderFrame
39	Structures de contrôle de flux	Benno Grimm, Comintern, Kelly Tessena Keck, Leviathan, litelite, Macro Man, Martin, Mat's Mug, Roland, Siva, ThunderFrame
40	Structures de données	Blackhawk
41	Tableaux	Comintern, Dave, Hubisan, jamheadart, Josan Iracheta, Maarten van Stam, Mark.R, Mat's Mug, Miguel_Ryu, Tazaf
42	Travailler avec ADO	Comintern, SandPiper, Tazaf
43	Travailler avec des fichiers et des répertoires sans utiliser FileSystemObject	Comintern, Macro Man, SandPiper
44	Tri	Neil Mussett
45	Types de données et limites	Comintern, FreeMan, Neil Mussett, StackzOfZtuff, Stephen Leppik, ThunderFrame
46	VBA orienté objet	IvenBach, Mat's Mug