eBook Gratuit

APPRENEZ excel-vba

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

#excel-vba

Table des matières

À propos

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

It is an unofficial and free excel-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 excel-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 excel-vba

Remarques

Microsoft Excel inclut un langage de programmation macro complet appelé VBA. Ce langage de programmation vous fournit au moins trois ressources supplémentaires:

1. Piloter automatiquement Excel à partir du code en utilisant des macros. Pour l'essentiel, tout ce que l'utilisateur peut faire en manipulant Excel depuis l'interface utilisateur peut être fait en écrivant du code dans Excel VBA.
2. Créez de nouvelles fonctions de feuille de calcul personnalisées.
3. Interagir avec Excel avec d'autres applications telles que Microsoft Word, PowerPoint, Internet Explorer, le Bloc-notes, etc.

VBA signifie Visual Basic pour Applications. Il s'agit d'une version personnalisée du vénérable langage de programmation Visual Basic qui alimente les macros de Microsoft Excel depuis le milieu des années 1990.

IMPORTANT
Veuillez vous assurer que tous les exemples ou sujets créés dans la balise excel-vba sont **spécifiques** et **pertinents** pour l'utilisation de VBA avec Microsoft Excel. Tout sujet ou exemple suggéré qui est générique au langage VBA devrait être refusé afin d'éviter la duplication des efforts.

- exemples sur le sujet:

  ✓ Créer et interagir avec des objets de feuille de travail
  ✓ La classe `WorksheetFunction` *et les méthodes respectives*
  ✓ Utilisation de l'énumération `xlDirection` *pour parcourir une plage*

- exemples hors sujet:

  ✗ Comment créer une boucle 'pour chaque'
  Class *Classe* `MsgBox` *et comment afficher un message*
  ✗ Utilisation de WinAPI dans VBA

Versions

VB

Version	Date de sortie
VB6	1998-10-01

Version	Date de sortie
VB7	2001-06-06
WIN32	1998-10-01
WIN64	2001-06-06
MAC	1998-10-01

Exceller

Version	Date de sortie
16	2016-01-01
15	2013-01-01
14	2010-01-01
12	2007-01-01
11	2003-01-01
dix	2001-01-01
9	1999-01-01
8	1997-01-01
7	1995-01-01
5	1993-01-01
2	1987-01-01

Examples

Déclaration des variables

Pour déclarer explicitement des variables dans VBA, utilisez l'instruction `Dim` , suivie du nom et du type de la variable. Si une variable est utilisée sans être déclarée ou si aucun type n'est spécifié, le type `Variant` lui sera affecté.

Utilisez l'instruction `Option Explicit` sur la première ligne d'un module pour forcer toutes les variables à être déclarées avant utilisation (voir TOUJOURS utiliser "Option Explicit").

Il est fortement recommandé d'utiliser toujours `Option Explicit` car cela permet d'éviter les fautes

de frappe et les fautes d'orthographe et garantit que les variables / objets resteront conformes au type souhaité.

```
Option Explicit

Sub Example()
    Dim a As Integer
    a = 2
    Debug.Print a
    'Outputs: 2

    Dim b As Long
    b = a + 2
    Debug.Print b
    'Outputs: 4

    Dim c As String
    c = "Hello, world!"
    Debug.Print c
    'Outputs: Hello, world!
End Sub
```

Plusieurs variables peuvent être déclarées sur une seule ligne en utilisant des virgules en tant que délimiteurs, mais **chaque type doit être déclaré individuellement** ou par défaut dans le type `Variant` .

```
Dim Str As String, IntOne, IntTwo As Integer, Lng As Long
Debug.Print TypeName(Str)    'Output: String
Debug.Print TypeName(IntOne) 'Output: Variant <--- !!!
Debug.Print TypeName(IntTwo) 'Output: Integer
Debug.Print TypeName(Lng)    'Output: Long
```

Les variables peuvent également être déclarées à l'aide des suffixes de caractères de type de données ($% &! # @), Mais leur utilisation est de plus en plus déconseillée.

```
 Dim this$  'String
 Dim this%  'Integer
 Dim this&  'Long
 Dim this!  'Single
 Dim this#  'Double
 Dim this@  'Currency
```

Les autres façons de déclarer des variables sont les suivantes:

- `Static` comme: `Static CounterVariable as Integer`

  Lorsque vous utilisez l'instruction Static au lieu d'une instruction Dim, la variable déclarée conservera sa valeur entre les appels.

- `Public` like: `Public CounterVariable as Integer`

Les variables publiques peuvent être utilisées dans toutes les procédures du projet. Si une variable publique est déclarée dans un module standard ou un module de classe, elle peut également être utilisée dans tous les projets faisant référence au projet dans lequel la variable publique est déclarée.

- `Private` comme: `Private CounterVariable as Integer`

  Les variables privées ne peuvent être utilisées que par les procédures du même module.

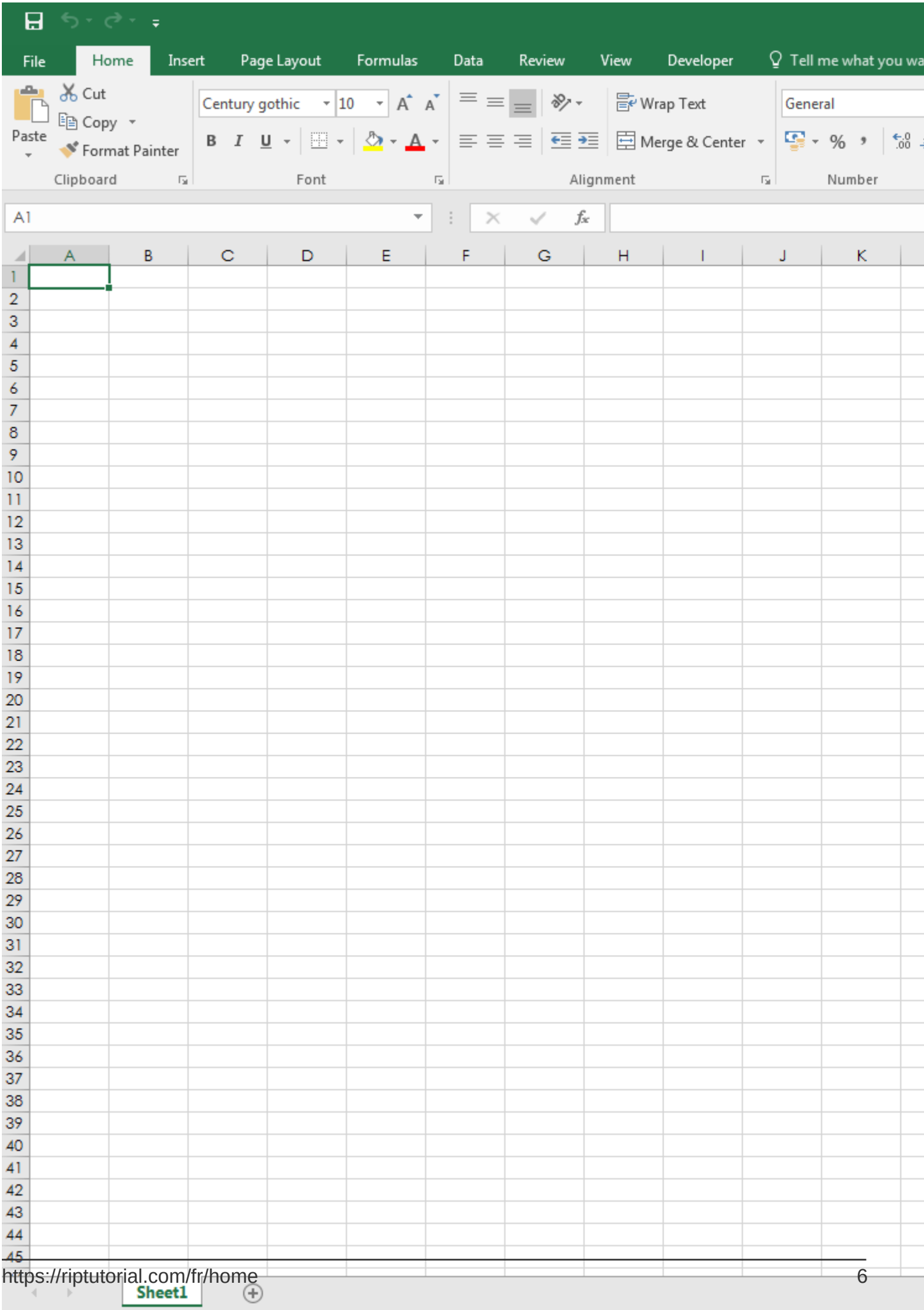
Source et plus d'infos:

Variables déclarant MSDN

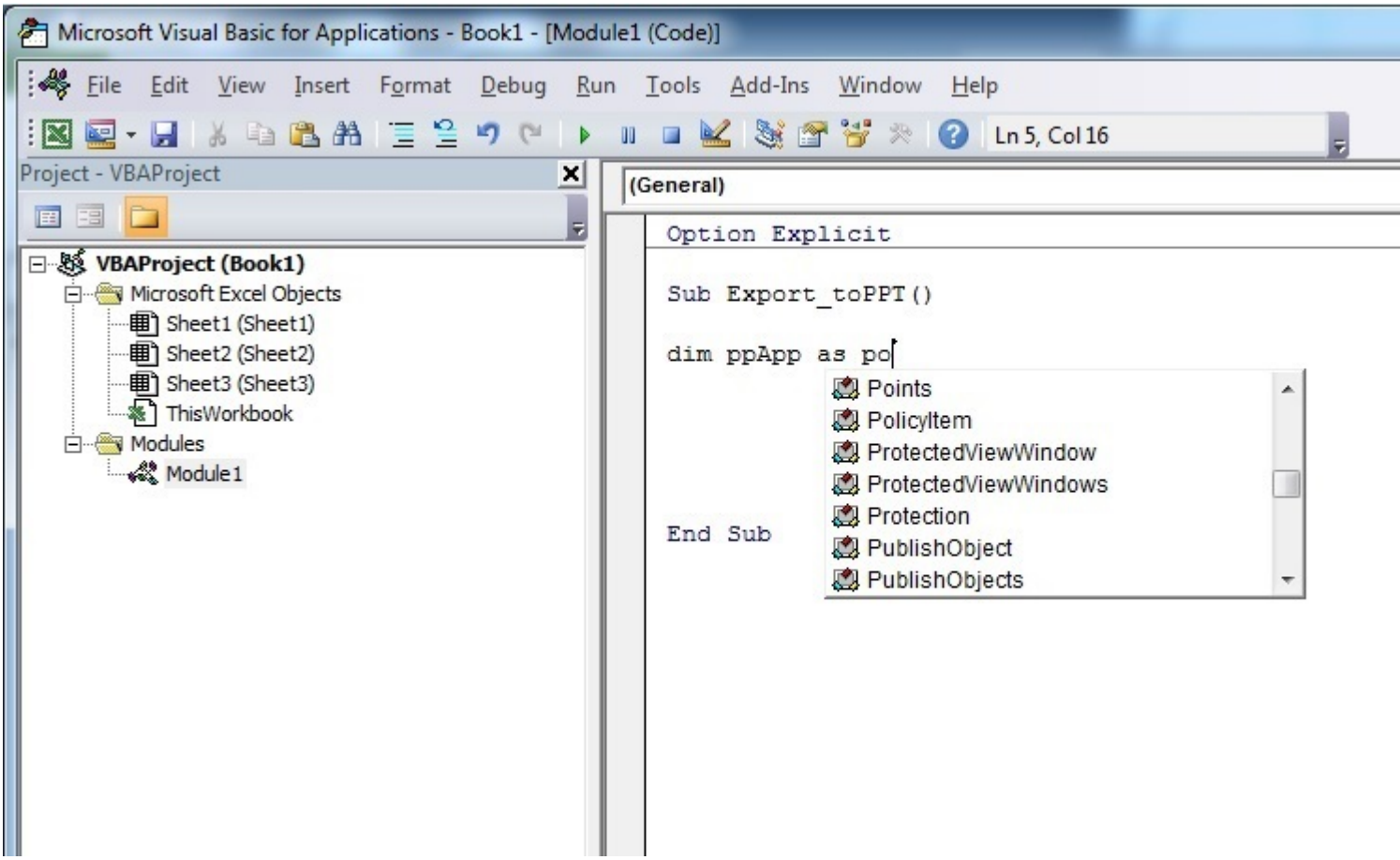
Caractères de type (Visual Basic)

Ouverture de l'éditeur Visual Basic (VBE)

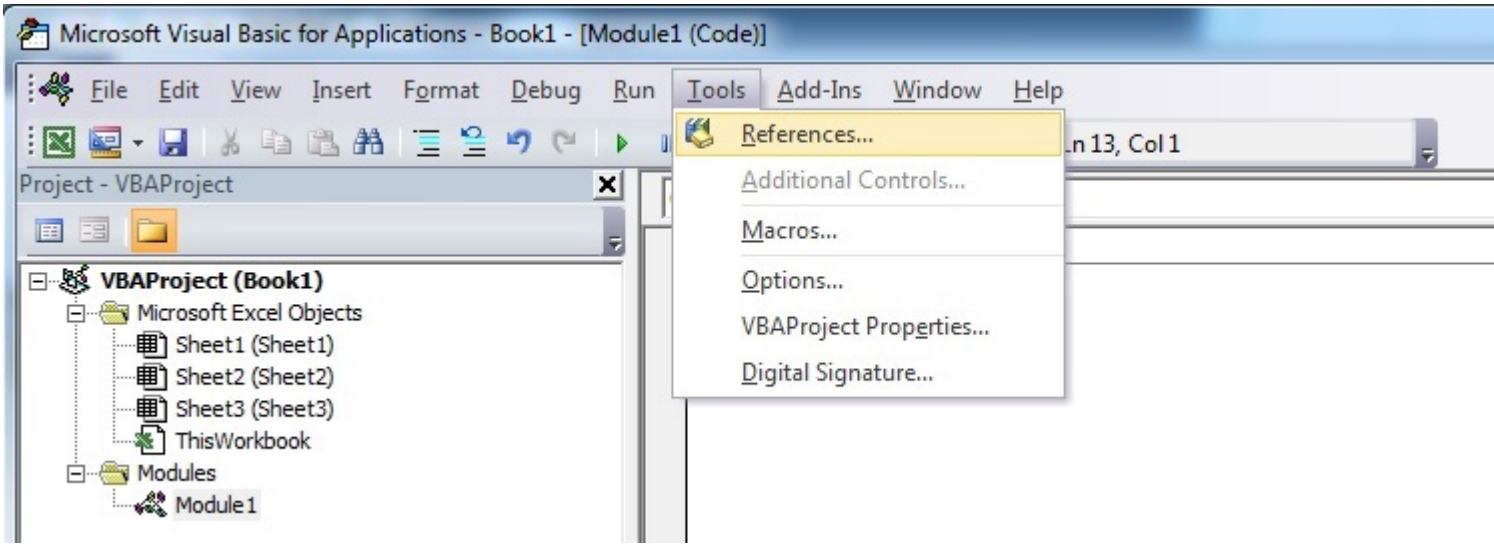
Étape 1: Ouvrir un classeur


File
Home
Insert
Page Layout
Formulas
Data
Review
View
Developer
Tell me what you wa
Cut
Copy
Format Painter
Paste
Century gothic
10
Wrap Text
General
Merge & Center
Clipboard
Font
Alignment
Number
A1
Sheet1

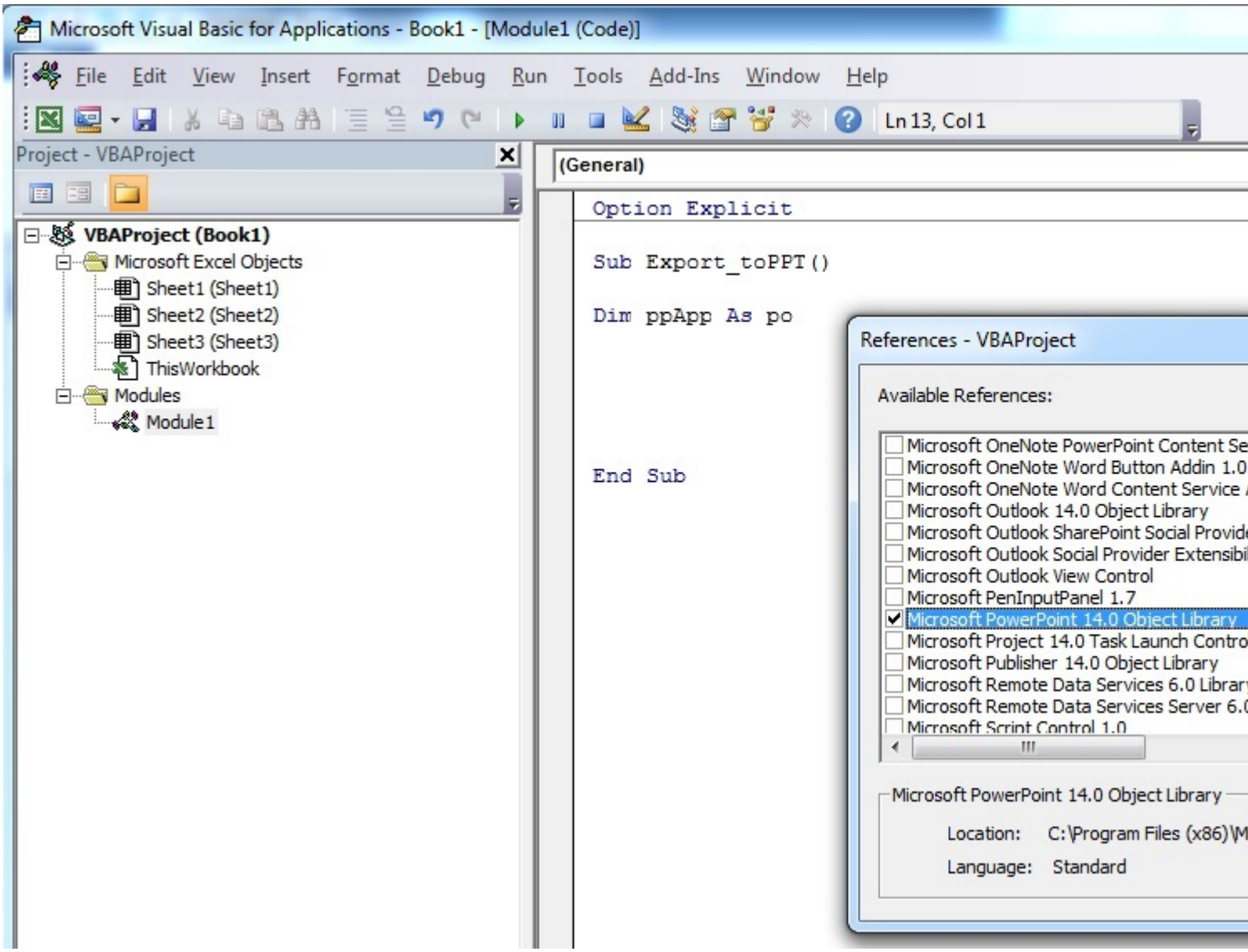
au projet VB existant. Comme on peut le voir, la bibliothèque d'objets PowerPoint n'est actuellement pas disponible.



Étape 1 : Sélectionnez ***Outils de*** menu ***-> Références…***

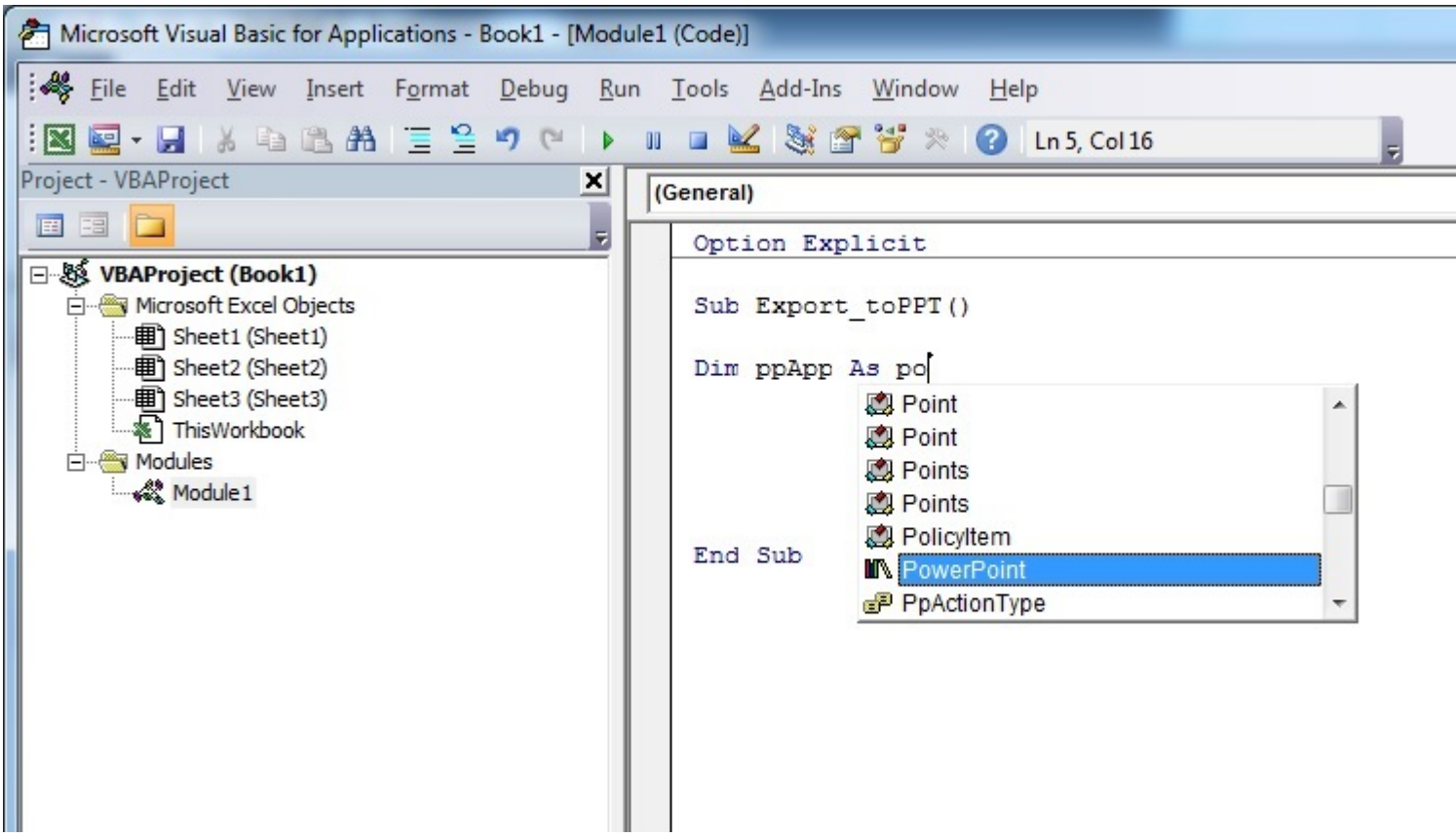


Étape 2 : Sélectionnez la référence à ajouter. Cet exemple, nous faisons défiler pour trouver « ***Bibliothèque d'objets Microsoft PowerPoint 14.0*** », puis appuyez sur « **OK** ».

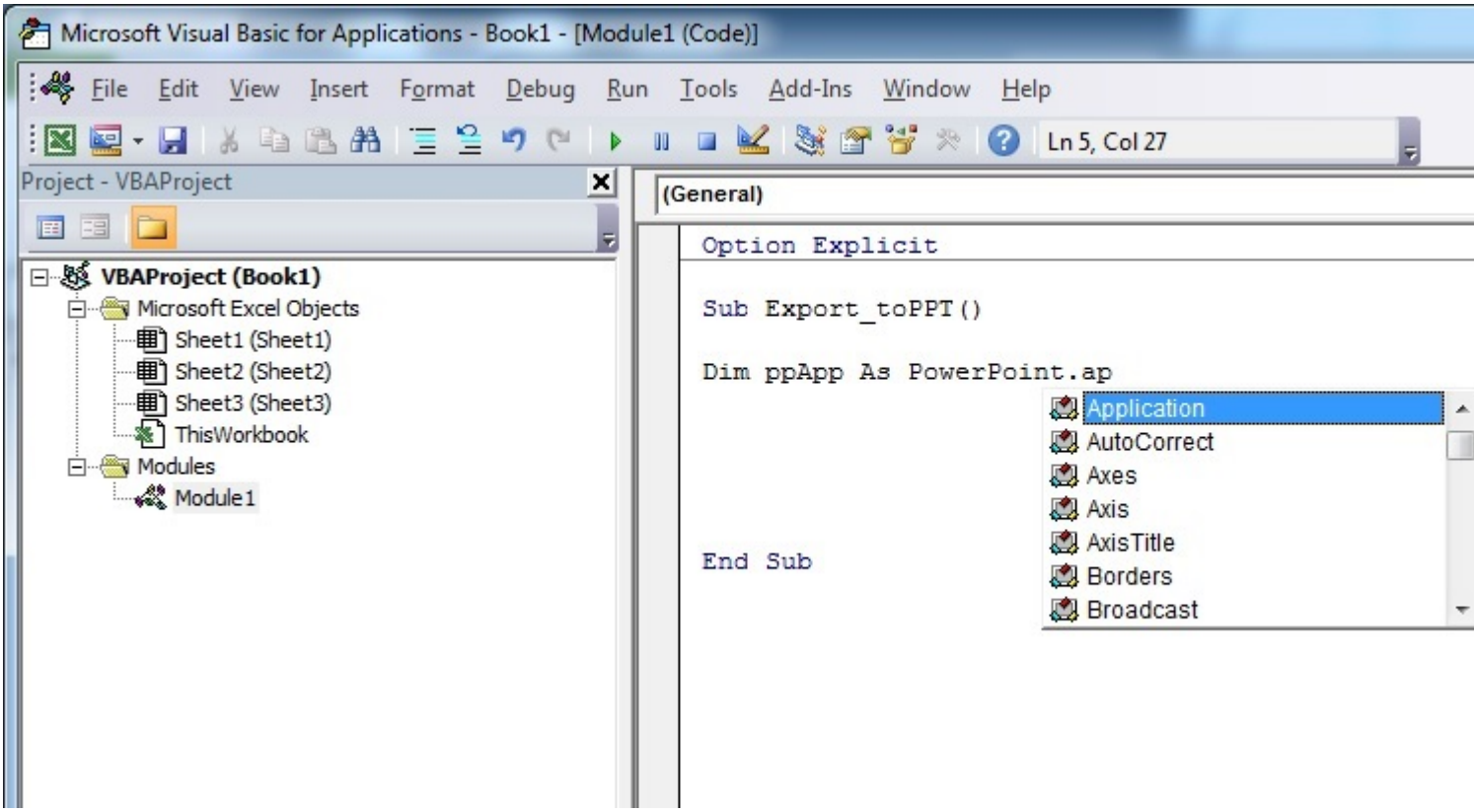


Remarque: PowerPoint 14.0 signifie que la version d'Office 2010 est installée sur le PC.

Étape 3 : dans l'éditeur VB, une fois que vous appuyez sur **Ctrl + Espace** , vous obtenez l'option de saisie semi-automatique de PowerPoint.

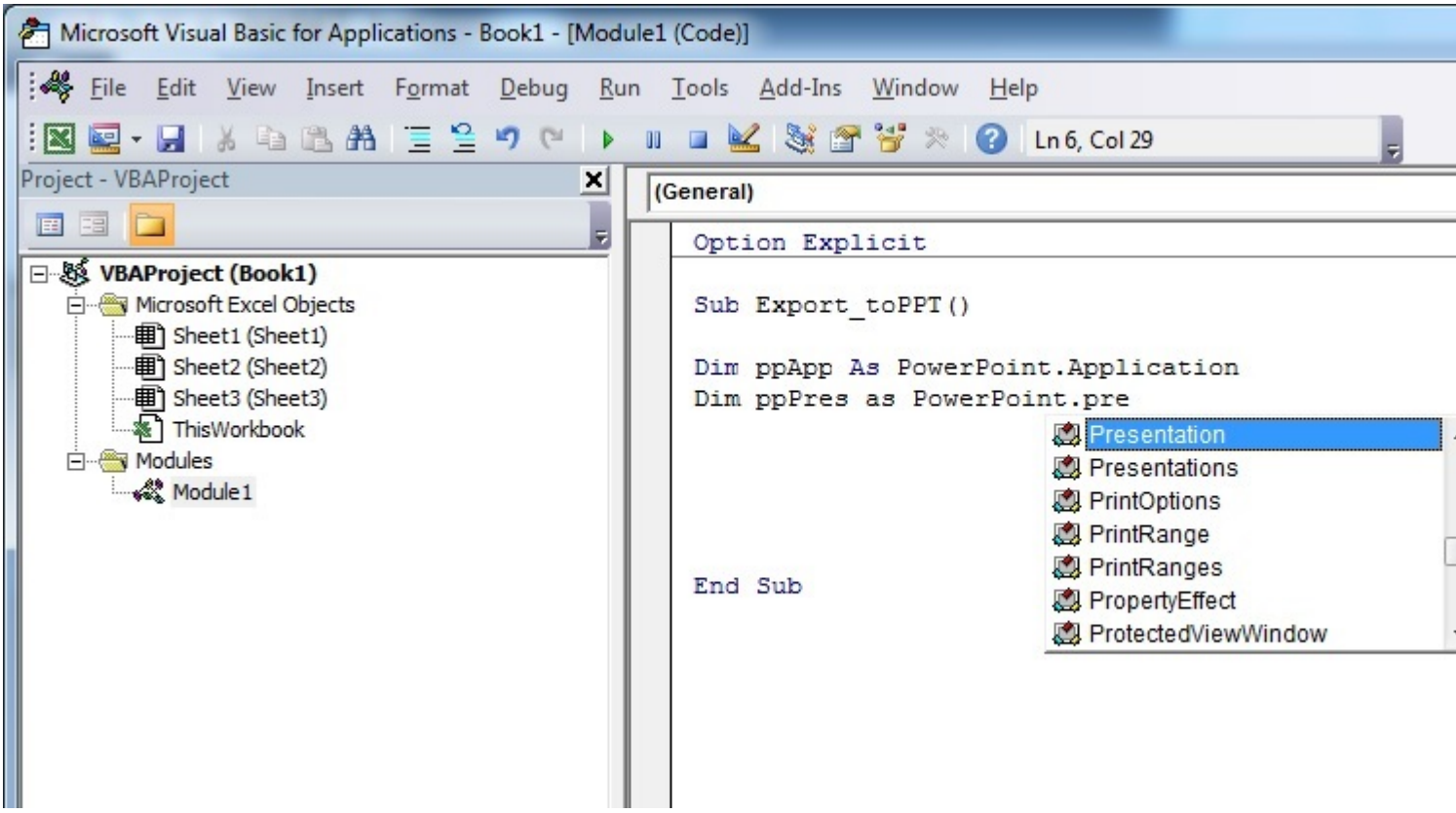


Après avoir sélectionné `PowerPoint` et appuyé sur `.` , un autre menu apparaît avec toutes les options d'objets liées à la bibliothèque d'objets PowerPoint. Cet exemple montre comment sélectionner l'objet `Application` PowerPoint.

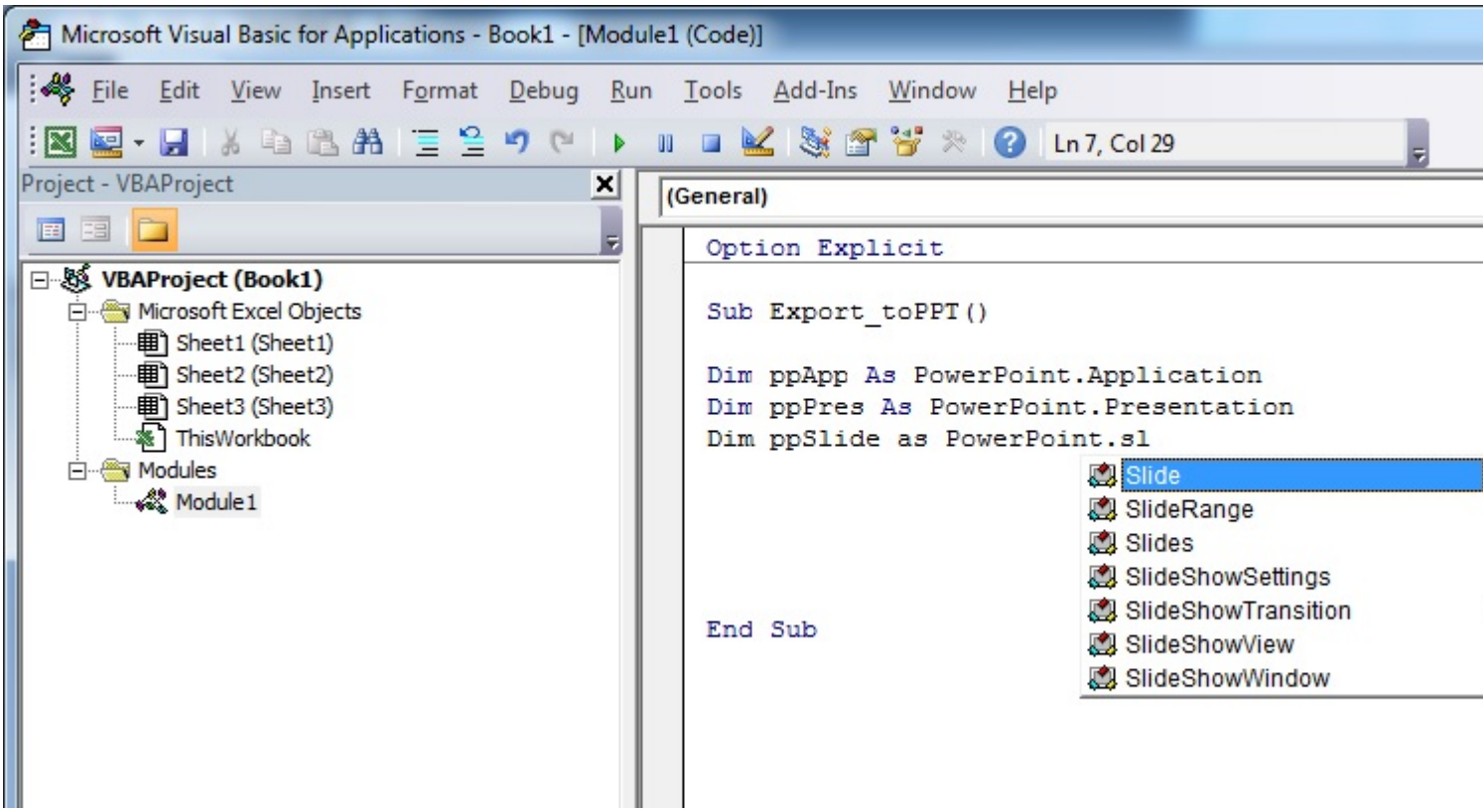


Étape 4 : L'utilisateur peut maintenant déclarer plus de variables à l'aide de la bibliothèque d'objets PowerPoint.

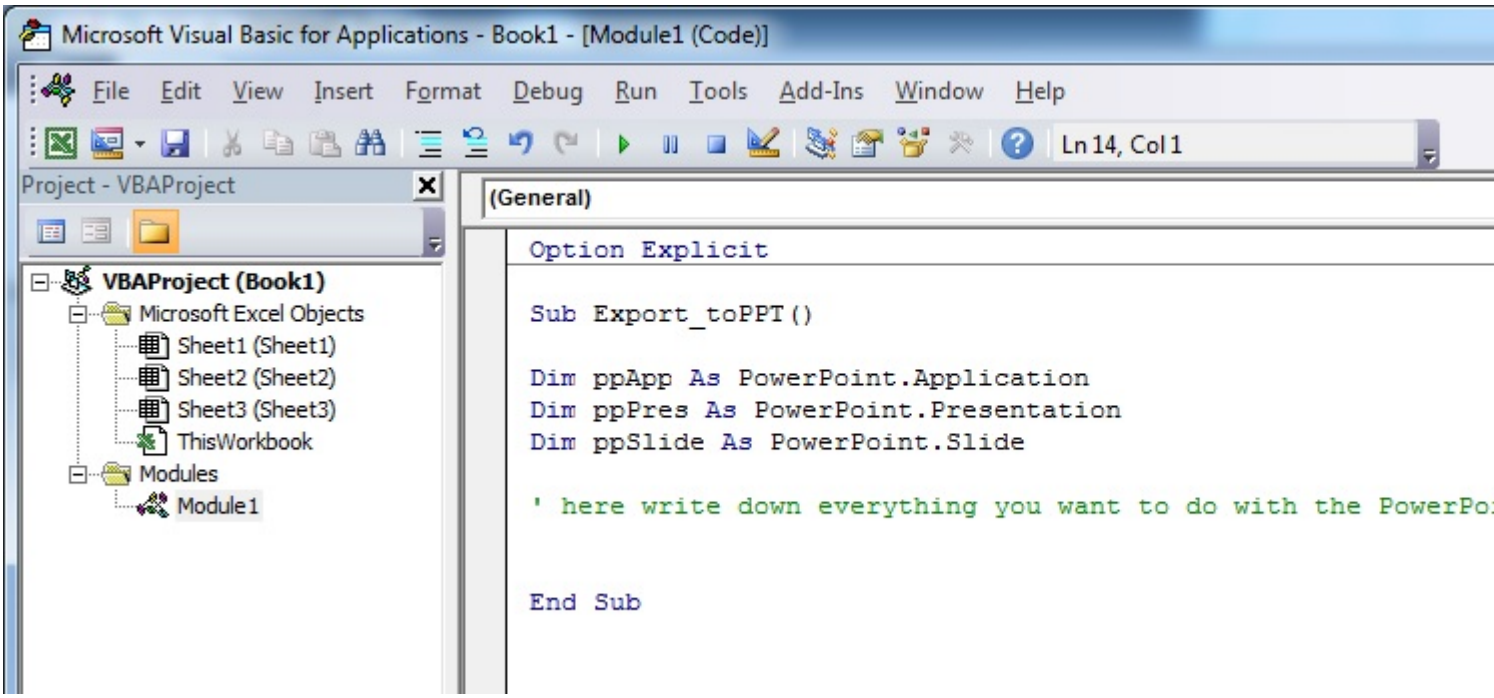
Déclarez une variable faisant référence à l'objet `Presentation` de la bibliothèque d'objets PowerPoint.



Déclarez une autre variable faisant référence à l'objet `Slide` de la bibliothèque d'objets PowerPoint.



Maintenant, la section de déclaration des variables ressemble à la capture d'écran ci-dessous, et l'utilisateur peut commencer à utiliser ces variables dans son code.

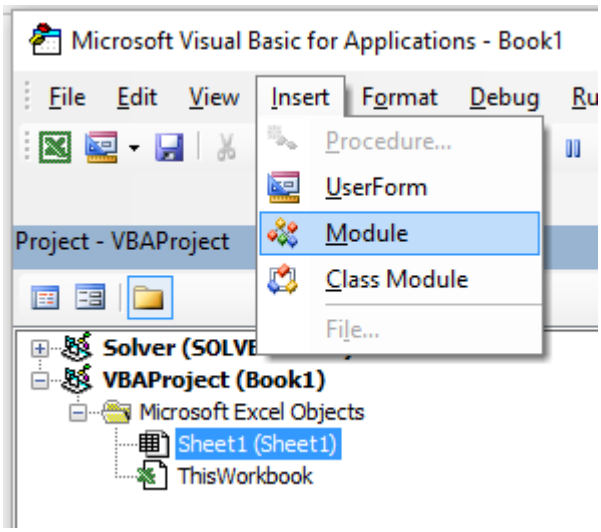


Version de code de ce tutoriel:

```
Option Explicit

Sub Export_toPPT()

Dim ppApp As PowerPoint.Application
Dim ppPres As PowerPoint.Presentation
Dim ppSlide As PowerPoint.Slide

' here write down everything you want to do with the PowerPoint Class and objects


End Sub
```

Bonjour le monde

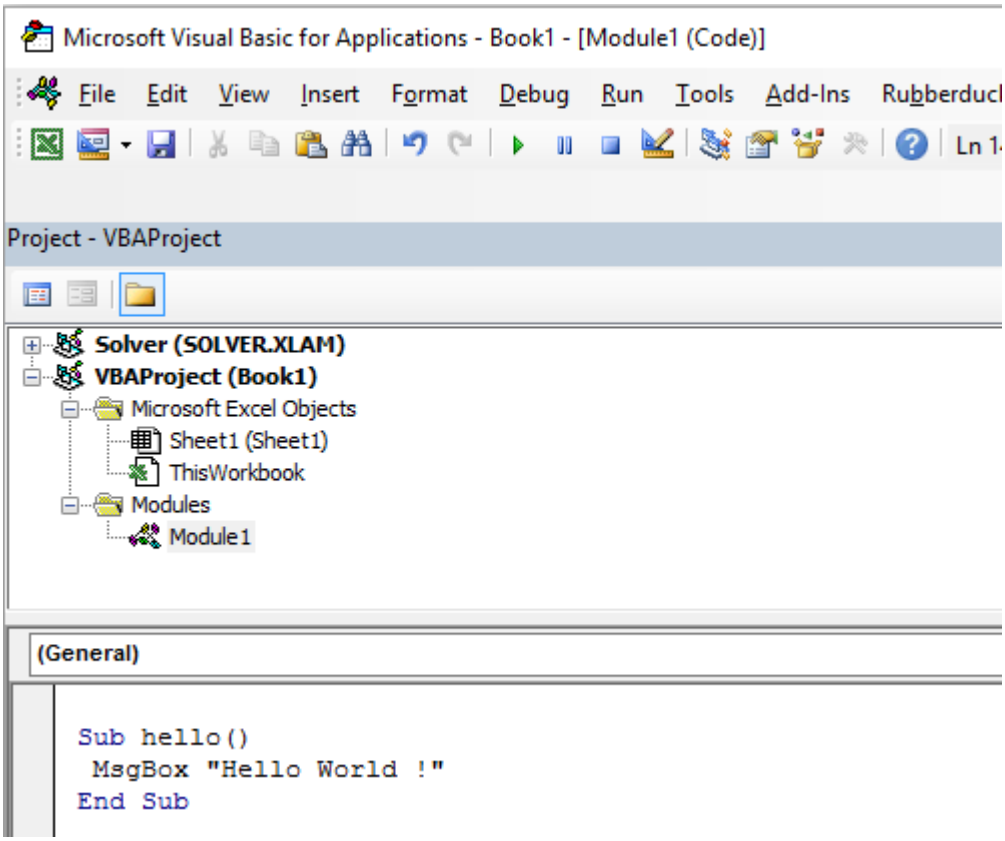
1. Ouvrez Visual Basic Editor (voir Ouverture de Visual Basic Editor)
2. Cliquez sur Insérer -> Module pour ajouter un nouveau module:



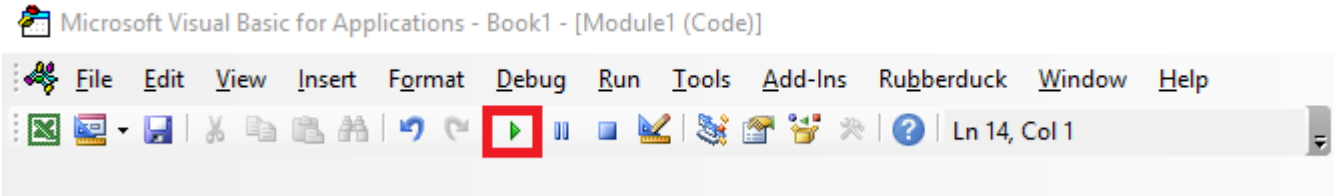
3. Copiez et collez le code suivant dans le nouveau module:

```
Sub hello()
  MsgBox "Hello World !"
End Sub
```

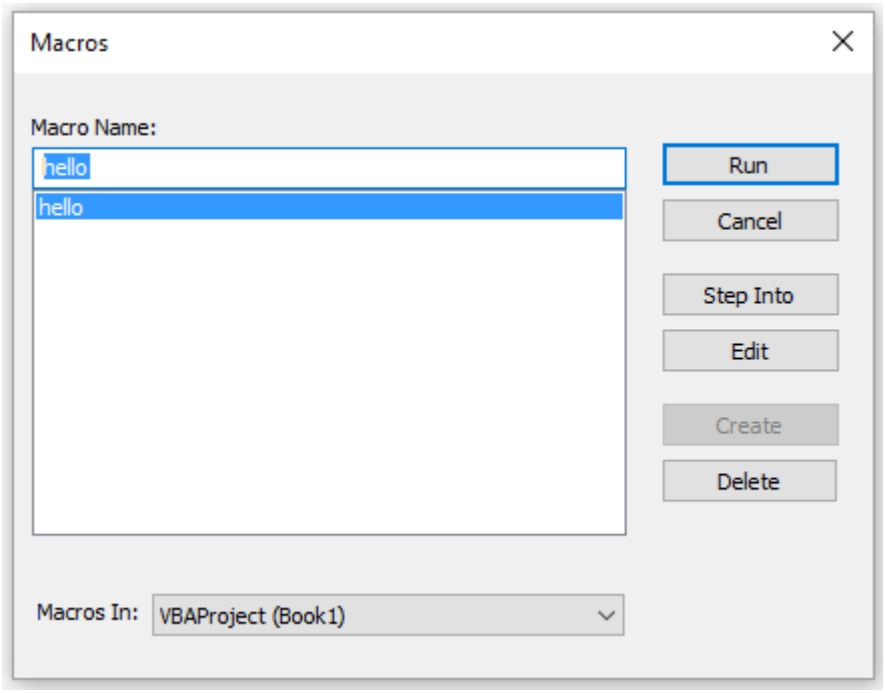
Obtenir :



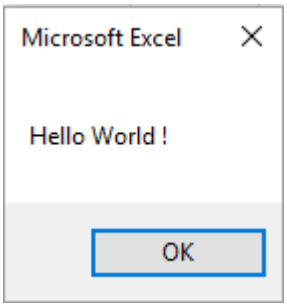
4. Cliquez sur la flèche verte "play" (ou appuyez sur F5) dans la barre d'outils Visual Basic pour exécuter le programme:



5. Sélectionnez le nouveau sous créé "hello" et cliquez sur `Run` :



6. Fait, vous devriez voir la fenêtre suivante:

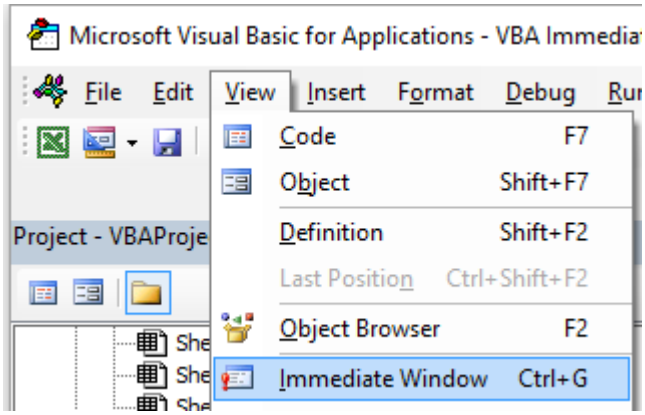


Démarrer avec le modèle d'objet Excel

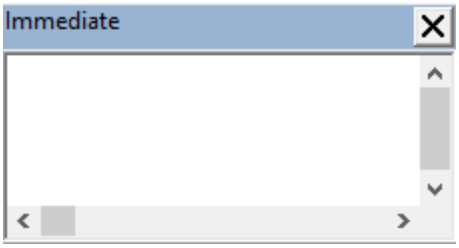
Cet exemple se veut une introduction douce au modèle objet Excel **pour les débutants** .

---

1. Ouvrez l'éditeur Visual Basic (VBE)
2. Cliquez sur Afficher -> Fenêtre immédiate pour ouvrir la fenêtre Immédiat (ou `Ctrl` + `G`):



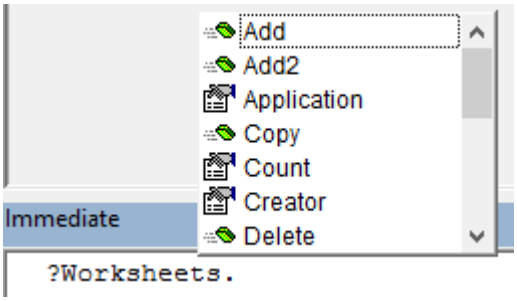
3. Vous devriez voir la fenêtre suivante en bas sur VBE:



Cette fenêtre vous permet de tester directement du code VBA. Commençons donc, tapez cette console:

```
?Worksheets.
```

VBE a intellisense et ensuite il devrait ouvrir une info-bulle comme dans la figure suivante:



Sélectionnez .Count dans la liste ou tapez directement `.Cout` pour obtenir:

```
?Worksheets.Count
```

4. Puis appuyez sur Entrée. L'expression est évaluée et doit renvoyer 1. Cela indique le nombre de feuilles de calcul actuellement présentes dans le classeur. Le point d'interrogation (`?`) Est un alias pour Debug.Print.

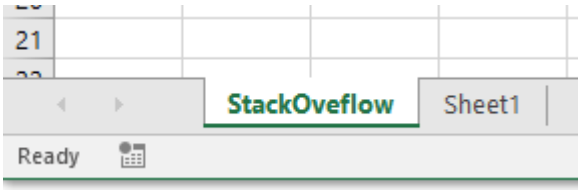
Worksheets est un **objet** et Count est une **méthode** . Excel a plusieurs objets (`Workbook` , `Worksheet` , `Range` , `Chart` ..) et chacun contient des méthodes et des propriétés spécifiques. Vous pouvez trouver la liste complète des objets dans la référence Excel VBA . Feuilles de travail L'objet est présenté ici .

> Cette référence Excel VBA devrait devenir votre principale source d'informations concernant le modèle d'objet Excel.

5. Essayons maintenant une autre expression, tapez (sans le caractère `?`):

```
Worksheets.Add().Name = "StackOveflow"
```

6. Appuyez sur Entrée. Cela devrait créer une nouvelle feuille de calcul appelée `StackOverflow.` :



Pour comprendre cette expression, vous devez lire la fonction Ajouter dans la référence Excel susmentionnée. Vous trouverez ce qui suit:

```
Add:  Creates a new worksheet, chart, or macro sheet.
The new worksheet becomes the active sheet.
Return Value: An Object value that represents the new worksheet, chart,
 or macro sheet.
```

Donc, les `Worksheets.Add()` créent une nouvelle feuille de calcul et la renvoient. La feuille de travail (**sans s**) est elle-même un objet qui peut être trouvé dans la documentation et `Name` est l'une de ses **propriétés** (voir ici). Il est défini comme suit:

```
Worksheet.Name Property:  Returns or sets a String value that
 represents the object name.
```

Ainsi, en étudiant les différentes définitions d'objets, nous pouvons comprendre ce code `Worksheets.Add().Name = "StackOveflow"` .

`Add()` crée et ajoute une nouvelle feuille de calcul et y renvoie une **référence** , puis nous définissons sa **propriété** Name sur "StackOverflow"

Soyons plus formels, Excel contient plusieurs objets. Ces objets peuvent être composés d'une ou plusieurs collection (s) d'objets Excel de la même classe. C'est le cas pour `WorkSheets` qui est une collection d'objets `Worksheet` . Chaque objet possède des propriétés et des méthodes avec lesquelles le programmeur peut interagir.

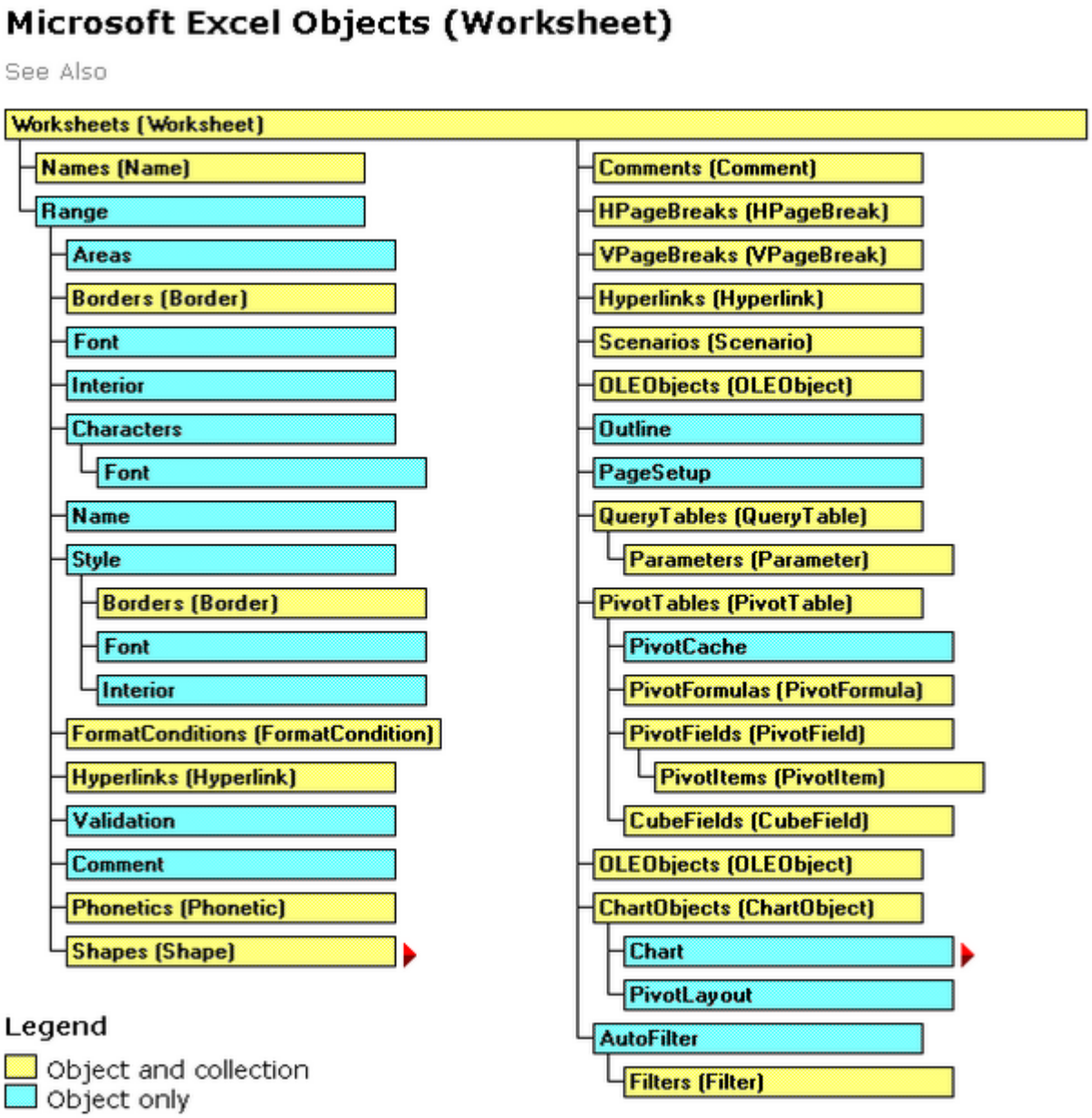
> Le modèle d'objet Excel fait référence à la **hiérarchie d'objet** Excel

En haut de tous les objets se trouve l'objet `Application` , il représente l'instance Excel elle-même. La programmation en VBA nécessite une bonne compréhension de cette hiérarchie car nous avons toujours besoin d'une référence à un objet pour pouvoir appeler une méthode ou définir / obtenir une propriété.

Le modèle d'objet Excel (très simplifié) peut être représenté comme suit:

```
Application
 Workbooks
 Workbook
Worksheets
 Worksheet
  Range
```

Une version plus détaillée de l'objet Worksheet (comme dans Excel 2007) est présentée ci-dessous.



Le modèle d'objet Excel complet peut être trouvé ici .

Enfin, certains objets peuvent avoir des `events` (ex: `Workbook.WindowActivate`) qui font également partie du modèle d'objet Excel.

Lire Démarrer avec excel-vba en ligne: https://riptutorial.com/fr/excel-vba/topic/777/demarrer-avec-excel-vba

Chapitre 2: Cellules / plages fusionnées

Examples

Réfléchissez à deux fois avant d'utiliser les cellules / plages fusionnées

Tout d'abord, les cellules fusionnées ne sont là que pour améliorer l'apparence de vos feuilles.

Donc, c'est littéralement la dernière chose à faire, une fois que votre feuille et votre cahier d'exercices sont totalement fonctionnels!

Où se trouvent les données dans une plage fusionnée?

Lorsque vous fusionnez une plage, vous n'afficherez qu'un seul bloc.

Les données seront dans la toute **première cellule de cette plage** , et les **autres seront des cellules vides** !

Un bon point à ce sujet: pas besoin de remplir toutes les cellules ou la plage une fois fusionnées, il suffit de remplir la première cellule! ;)

Les autres aspects de cette gamme fusionnée sont globalement négatifs:

- Si vous utilisez une méthode pour trouver la dernière ligne ou colonne , vous risquez de rencontrer des erreurs
- Si vous parcourez des lignes et que vous avez fusionné des plages pour une meilleure lisibilité, vous rencontrerez des cellules vides et non la valeur affichée par la plage fusionnée.

Lire Cellules / plages fusionnées en ligne: https://riptutorial.com/fr/excel-vba/topic/7308/cellules---plages-fusionnees

Chapitre 3: Classeurs

Examples

Cahiers de travail

Dans de nombreuses applications Excel, le code VBA effectue des actions sur le classeur dans lequel il est contenu. Vous enregistrez ce classeur avec une extension ".xlsm" et les macros VBA se concentrent uniquement sur les feuilles de calcul et les données à l'intérieur. Toutefois, vous devez souvent combiner ou fusionner des données provenant d'autres classeurs ou écrire certaines de vos données dans un classeur distinct. L'ouverture, la fermeture, la sauvegarde, la création et la suppression d'autres classeurs est un besoin commun pour de nombreuses applications VBA.

À tout moment dans l'éditeur VBA, vous pouvez afficher et accéder à tous les classeurs actuellement ouverts par cette instance d'Excel à l'aide de la propriété `Workbooks` de l'objet `Application` . La documentation MSDN l' explique par des références.

Quand utiliser ActiveWorkbook et ThisWorkbook

Il est recommandé de toujours spécifier le classeur auquel votre code VBA fait référence. Si cette spécification est omise, VBA suppose que le code est dirigé vers le classeur actuellement actif (`ActiveWorkbook`).

```
'--- the currently active workbook (and worksheet) is implied
Range("A1").value = 3.1415
Cells(1, 1).value = 3.1415
```

Toutefois, lorsque plusieurs classeurs sont ouverts simultanément, en particulier lorsque le code VBA s'exécute depuis un complément Excel, les références à `ActiveWorkbook` peuvent être confuses ou mal dirigées. Par exemple, un complément avec une fonction définie par l'utilisateur qui vérifie l'heure et le compare à une valeur stockée dans l'une des feuilles de calcul du complément (qui ne sont généralement pas facilement visibles par l'utilisateur) devra identifier explicitement le classeur être référencé. Dans notre exemple, notre classeur ouvert (et actif) a une formule dans la cellule A1 `=EarlyOrLate()` et n'a PAS de VBA écrit pour ce classeur actif. Dans notre complément, nous avons la fonction définie par l'utilisateur suivante:

```
Public Function EarlyOrLate() As String
    If Hour(Now) > ThisWorkbook.Sheets("WatchTime").Range("A1") Then
        EarlyOrLate = "It's Late!"
    Else
        EarlyOrLate = "It's Early!"
    End If
End Function
```

Le code pour le fichier UDF est écrit et stocké dans le complément Excel installé. Il utilise des données stockées sur une feuille de calcul dans le complément appelé "WatchTime". Si l'UDF

avait utilisé `ActiveWorkbook` au lieu de `ThisWorkbook` , il ne serait jamais en mesure de garantir la destination du classeur.

Ouvrir un (nouveau) classeur, même s'il est déjà ouvert

Si vous souhaitez accéder à un classeur déjà ouvert, vous pouvez accéder directement à l'affectation de la collection `Workbooks` :

```
dim myWB as Workbook
Set myWB = Workbooks("UsuallyFullPathnameOfWorkbook.xlsx")
```

Si vous souhaitez créer un nouveau classeur, utilisez l'objet de collection `Workbooks` pour `Add` une nouvelle entrée.

```
Dim myNewWB as Workbook
Set myNewWB = Workbooks.Add
```

Il y a des moments où vous pouvez ne pas ou (ou vous en soucier) si le classeur dont vous avez besoin est déjà ouvert ou non, ou si cela n'existe pas. La fonction exemple montre comment toujours retourner un objet classeur valide.

```
Option Explicit
Function GetWorkbook(ByVal wbFilename As String) As Workbook
    '--- returns a workbook object for the given filename, including checks
    '    for when the workbook is already open, exists but not open, or
    '    does not yet exist (and must be created)
    '    ***  wbFilename must be a fully specified pathname
    Dim folderFile As String
    Dim returnedWB As Workbook

    '--- check if the file exists in the directory location
    folderFile = File(wbFilename)
    If folderFile = "" Then
        '--- the workbook doesn't exist, so create it
        Dim pos1 As Integer
        Dim fileExt As String
        Dim fileFormatNum As Long
        '--- in order to save the workbook correctly, we need to infer which workbook
        '    type the user intended from the file extension
        pos1 = InStrRev(sFullName, ".", , vbTextCompare)
        fileExt = Right(sFullName, Len(sFullName) - pos1)
        Select Case fileExt
            Case "xlsx"
                fileFormatNum = 51
            Case "xlsm"
                fileFormatNum = 52
            Case "xls"
                fileFormatNum = 56
            Case "xlsb"
                fileFormatNum = 50
            Case Else
                Err.Raise vbObjectError + 1000, "GetWorkbook function", _
                        "The file type you've requested (file extension) is not recognized. "
& _
                        "Please use a known extension: xlsx, xlsm, xls, or xlsb."
```

```
        End Select
        Set returnedWB = Workbooks.Add
        Application.DisplayAlerts = False
        returnedWB.SaveAs filename:=wbFilename, FileFormat:=fileFormatNum
        Application.DisplayAlerts = True
        Set GetWorkbook = returnedWB
    Else
        '--- the workbook exists in the directory, so check to see if
        '    it's already open or not
        On Error Resume Next
        Set returnedWB = Workbooks(sFile)
        If returnedWB Is Nothing Then
            Set returnedWB = Workbooks.Open(sFullName)
        End If
    End If
End Function
```

Enregistrement d'un classeur sans demander à l'utilisateur

L'enregistrement de nouvelles données dans un classeur existant à l'aide de VBA entraîne souvent une question contextuelle indiquant que le fichier existe déjà.

Pour éviter cette question, vous devez supprimer ces types d'alertes.

```
Application.DisplayAlerts = False        'disable user prompt to overwrite file
myWB.SaveAs FileName:="NewOrExistingFilename.xlsx"
Application.DisplayAlerts = True         're-enable user prompt to overwrite file
```

Modification du nombre de feuilles de calcul par défaut dans un nouveau classeur

Le nombre "d'usine par défaut" de feuilles de calcul créées dans un nouveau classeur Excel est généralement défini sur trois. Votre code VBA peut définir explicitement le nombre de feuilles de calcul dans un nouveau classeur.

```
'--- save the current Excel global setting
With Application
    Dim oldSheetsCount As Integer
    oldSheetsCount = .SheetsInNewWorkbook
    Dim myNewWB As Workbook
    .SheetsInNewWorkbook = 1
    Set myNewWB = .Workbooks.Add
    '--- restore the previous setting
    .SheetsInNewWorkbook = oldsheetcount
End With
```

Lire Classeurs en ligne: https://riptutorial.com/fr/excel-vba/topic/2969/classeurs

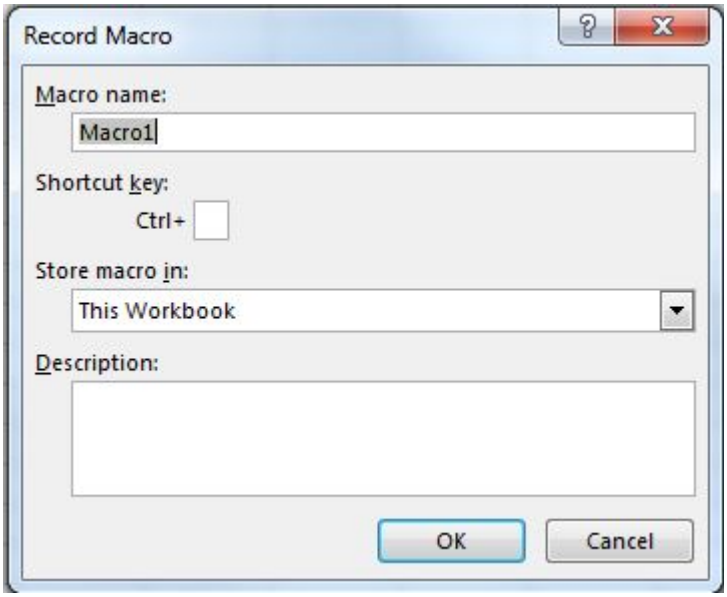
Chapitre 4: Comment enregistrer une macro

Examples

Comment enregistrer une macro

Le moyen le plus simple d'enregistrer une macro est le bouton situé dans le coin inférieur gauche d'Excel: 

Lorsque vous cliquez dessus, vous obtenez une fenêtre pop-up vous demandant de nommer la macro et de décider si vous souhaitez avoir une touche de raccourci. En outre, demande où stocker la macro et pour une description. Vous pouvez choisir n'importe quel nom, aucun espace n'est autorisé.



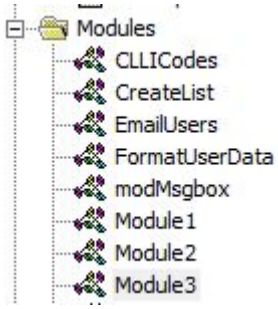
Si vous souhaitez qu'un raccourci soit affecté à votre macro pour une utilisation rapide, choisissez une lettre dont vous vous souviendrez pour pouvoir utiliser rapidement et facilement la macro.

Vous pouvez stocker la macro dans «Ce classeur», «Nouveau classeur» ou «classeur de macros personnelles». Si vous souhaitez que la macro que vous êtes sur le point d'enregistrer soit uniquement disponible dans le classeur actuel, choisissez «Ce classeur». Si vous voulez qu'il soit enregistré dans un nouveau classeur, choisissez «Nouveau classeur». Et si vous souhaitez que la macro soit disponible dans tous les classeurs que vous ouvrez, choisissez «Classeur de macros personnelles».

Après avoir rempli ce pop-up, cliquez sur "Ok".

Effectuez ensuite les actions que vous souhaitez répéter avec la macro. Lorsque vous avez terminé, cliquez sur le même bouton pour arrêter l'enregistrement. Il ressemble maintenant à ceci:

Vous pouvez maintenant accéder à l'onglet Developer et ouvrir Visual Basic. (ou utilisez Alt + F11)



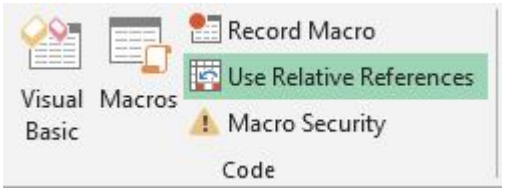
Vous allez maintenant avoir un nouveau module sous le dossier Modules.

Le module le plus récent contiendra la macro que vous venez d'enregistrer. Double-cliquez dessus pour le faire apparaître.

J'ai fait un simple copier-coller:

```
Sub Macro1()
'
' Macro1 Macro
'

'
    Selection.Copy
    Range("A12").Select
    ActiveSheet.Paste
End Sub
```

Si vous ne voulez pas qu'il colle toujours dans "A12", vous pouvez utiliser les références relatives en cochant la case "Utiliser les références relatives" de l'onglet Developer:



En suivant les mêmes étapes que précédemment, vous allez maintenant transformer la macro en ceci:

```
Sub Macro2()
'
' Macro2 Macro
'

'
    Selection.Copy
    ActiveCell.Offset(11, 0).Range("A1").Select
    ActiveSheet.Paste
End Sub
```

Toujours copier la valeur de "A1" dans une cellule de 11 lignes, mais maintenant vous pouvez effectuer la même macro avec n'importe quelle cellule de départ et la valeur de cette cellule sera copiée dans les 11 lignes de la cellule.

Lire Comment enregistrer une macro en ligne: https://riptutorial.com/fr/excel-vba/topic/8204/comment-enregistrer-une-macro

Chapitre 5: Contraignant

Examples

Reliure précoce vs liaison tardive

La liaison est le processus d'affectation d'un objet à un identifiant ou à un nom de variable. La liaison anticipée (également appelée liaison statique) se produit lorsqu'un objet déclaré dans Excel est d'un type d'objet spécifique, tel qu'une feuille de calcul ou un classeur. La liaison tardive se produit lorsque des associations d'objets générales sont créées, telles que les types de déclaration Object et Variant.

La liaison anticipée des références présente certains avantages par rapport à la liaison tardive.

- La liaison anticipée est opérationnellement plus rapide que la liaison tardive pendant l'exécution. La création de l'objet avec une liaison tardive au moment de l'exécution prend un certain temps avant que la liaison anticipée ne soit terminée lorsque le projet VBA est initialement chargé.
- La liaison anticipée offre des fonctionnalités supplémentaires grâce à l'identification des paires clé / article par leur position ordinale.
- Selon la structure du code, une liaison anticipée peut offrir un niveau supplémentaire de vérification de type et réduire les erreurs.
- La correction de la capitalisation du VBE lors de la saisie des propriétés et des méthodes d'un objet lié est active avec une liaison anticipée mais indisponible avec une liaison tardive.

  Remarque: Vous devez ajouter la référence appropriée au projet VBA via la commande Outils → Références de VBE afin de mettre en œuvre la liaison anticipée.

  Cette référence de bibliothèque est ensuite 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.

```
'Looping through a dictionary that was created with late binding[1]
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

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

End Sub
```

```
'Looping through a dictionary that was created with early binding[1]
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

    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
```

Toutefois, si vous utilisez une liaison anticipée et que le document est exécuté sur un système dépourvu de l'une des bibliothèques référencées, vous rencontrerez des problèmes. Non seulement les routines qui utilisent la bibliothèque manquante ne fonctionneront pas correctement, mais le comportement de tout le code dans le document deviendra erratique. Il est probable qu'aucun code du document ne fonctionnera sur cet ordinateur.

C'est là que la liaison tardive est avantageuse. Lorsque vous utilisez une liaison tardive, vous n'avez pas besoin d'ajouter la référence dans le menu Outils> Références. Sur les machines disposant de la bibliothèque appropriée, le code fonctionnera toujours. Sur les machines sans cette bibliothèque, les commandes faisant référence à la bibliothèque ne fonctionneront pas, mais tous les autres codes de votre document continueront à fonctionner.

Si vous ne connaissez pas bien la bibliothèque à laquelle vous faites référence, il peut être utile d'utiliser une liaison anticipée lors de l'écriture du code, puis de passer à la liaison tardive avant le déploiement. De cette façon, vous pouvez tirer parti du navigateur IntelliSense et Object Browser de VBE pendant le développement.

Lire Contraignant en ligne: https://riptutorial.com/fr/excel-vba/topic/3811/contraignant

Chapitre 6: Création d'un menu déroulant dans la feuille de travail active avec une zone de liste déroulante

Introduction

Voici un exemple simple montrant comment créer un menu déroulant dans la feuille active de votre classeur en insérant un objet ActiveX dans la feuille. Vous pourrez insérer l'une des cinq chansons de Jimi Hendrix dans n'importe quelle cellule activée de la feuille et pouvoir la vider en conséquence.

Examples

Menu Jimi Hendrix

En général, le code est placé dans le module d'une feuille.

Il s'agit de l'événement Worksheet_SelectionChange, qui se déclenche chaque fois qu'une cellule différente est sélectionnée dans la feuille active. Vous pouvez sélectionner "Feuille de calcul" dans le premier menu déroulant au-dessus de la fenêtre de code et "Selection_Change" dans le menu déroulant situé à côté. Dans ce cas, chaque fois que vous activez une cellule, le code est redirigé vers le code de la zone de liste déroulante.

```
Private Sub Worksheet_SelectionChange(ByVal Target As Range)

   ComboBox1_Change

End Sub
```

Ici, la routine dédiée à la ComboBox est codée par défaut à l'événement Change. Dans ce document, il y a un tableau fixe, rempli de toutes les options. Pas l'option CLEAR dans la dernière position, qui sera utilisée pour effacer le contenu d'une cellule. Le tableau est ensuite remis à la Combo Box et passé à la routine qui fait le travail.

```
Private Sub ComboBox1_Change()

Dim myarray(0 To 5)
    myarray(0) = "Hey Joe"
    myarray(1) = "Little Wing"
    myarray(2) = "Voodoo Child"
    myarray(3) = "Purple Haze"
    myarray(4) = "The Wind Cries Mary"
    myarray(5) = "CLEAR"

    With ComboBox1
        .List = myarray()
```

```
    End With

    FillACell myarray()

End Sub
```

Le tableau est transmis à la routine qui remplit les cellules avec le nom de la chanson ou la valeur null pour les vider. Tout d'abord, une variable entière reçoit la valeur de la position du choix que l'utilisateur effectue. Ensuite, la zone de liste déroulante est déplacée dans le coin supérieur gauche de la cellule que l'utilisateur active et ses dimensions sont ajustées pour rendre l'expérience plus fluide. La cellule active se voit alors attribuer la valeur dans la position dans la variable entière, ce qui permet de suivre le choix de l'utilisateur. Si l'utilisateur sélectionne CLEAR parmi les options, la cellule est vidée.

La routine entière se répète pour chaque cellule sélectionnée.

```
Sub FillACell(MyArray As Variant)

Dim n As Integer

n = ComboBox1.ListIndex

ComboBox1.Left = ActiveCell.Left
ComboBox1.Top = ActiveCell.Top
Columns(ActiveCell.Column).ColumnWidth = ComboBox1.Width * 0.18

ActiveCell = MyArray(n)

If ComboBox1 = "CLEAR" Then
    Range(ActiveCell.Address) = ""
End If

End Sub
```

Exemple 2: Options non incluses

Cet exemple est utilisé pour spécifier des options qui pourraient ne pas être incluses dans une base de données de logements disponibles et de ses équipements.

Il s'appuie sur l'exemple précédent, avec quelques différences:

1. Deux procédures ne sont plus nécessaires pour une seule combo, en combinant le code en une seule procédure.
2. L'utilisation de la propriété LinkedCell pour permettre la saisie correcte de la sélection de l'utilisateur à chaque fois
3. L'inclusion d'une fonctionnalité de sauvegarde pour garantir que la cellule active se trouve dans la colonne correcte et un code de prévention des erreurs, basé sur l'expérience précédente, où les valeurs numériques seraient mises en forme en tant que chaînes lorsqu'elles étaient renseignées dans la cellule active.

```
Private Sub cboNotIncl_Change()
```

```
Dim n As Long
Dim notincl_array(1 To 9) As String

n = myTarget.Row

    If n >= 3 And n < 10000 Then

        If myTarget.Address = "$G$" & n Then

            'set up the array elements for the not included services
            notincl_array(1) = "Central Air"
            notincl_array(2) = "Hot Water"
            notincl_array(3) = "Heater Rental"
            notincl_array(4) = "Utilities"
            notincl_array(5) = "Parking"
            notincl_array(6) = "Internet"
            notincl_array(7) = "Hydro"
            notincl_array(8) = "Hydro/Hot Water/Heater Rental"
            notincl_array(9) = "Hydro and Utilities"

            cboNotIncl.List = notincl_array()

        Else

            Exit Sub

        End If

        With cboNotIncl

            'make sure the combo box moves to the target cell
            .Left = myTarget.Left
            .Top = myTarget.Top

            'adjust the size of the cell to fit the combo box
            myTarget.ColumnWidth = .Width * 0.18

            'make it look nice by editing some of the font attributes
            .Font.Size = 11
            .Font.Bold = False

            'populate the cell with the user choice, with a backup guarantee that it's in
column G

            If myTarget.Address = "$G$" & n Then

                    .LinkedCell = myTarget.Address

                    'prevent an error where a numerical value is formatted as text
                    myTarget.EntireColumn.TextToColumns

            End If

        End With

    End If 'ensure that the active cell is only between rows 3 and 1000

End Sub
```

La macro ci-dessus est lancée chaque fois qu'une cellule est activée avec l'événement

SelectionChange dans le module de feuille de calcul:

```
Public myTarget As Range

Private Sub Worksheet_SelectionChange(ByVal Target As Range)

    Set myTarget = Target

    'switch for Not Included
    If Target.Column = 7 And Target.Cells.Count = 1 Then

        Application.Run "Module1.cboNotIncl_Change"

    End If

End Sub
```

Lire Création d'un menu déroulant dans la feuille de travail active avec une zone de liste déroulante en ligne: https://riptutorial.com/fr/excel-vba/topic/8929/creation-d-un-menu-deroulant-dans-la-feuille-de-travail-active-avec-une-zone-de-liste-deroulante

Chapitre 7: CustomDocumentProperties dans la pratique

Introduction

L'utilisation de CustomDocumentProperties (CDP) est une bonne méthode pour stocker les valeurs définies par l'utilisateur de manière relativement sûre dans le même classeur, mais en évitant d'afficher simplement les valeurs de cellule associées dans une feuille de travail non protégée *).

Remarque: les CDP représentent une collection distincte comparable à BuiltInDocumentProperties, mais permettent de créer vos propres noms de propriété définis par l'utilisateur au lieu d'une collection fixe.

*) Vous pouvez également saisir des valeurs dans un classeur caché ou "très caché".

Examples

Organisation de nouveaux numéros de facture

L'incrémentation d'un numéro de facture et la sauvegarde de sa valeur sont des tâches fréquentes. L'utilisation de CustomDocumentProperties (CDP) est une bonne méthode pour stocker ces nombres de manière relativement sûre dans le même classeur, tout en évitant d'afficher simplement les valeurs de cellule associées dans une feuille de travail non protégée.

Conseil supplémentaire:

Vous pouvez également saisir des valeurs dans une feuille de calcul masquée ou même dans une feuille de calcul dite "très cachée" (voir Utilisation de feuilles xlVeryHidden . Bien sûr, il est également possible de sauvegarder des données dans des fichiers externes (fichier ini, csv ou tout autre type). ou le registre.

Exemple de contenu :

L'exemple ci-dessous montre

- une fonction NextInvoiceNo qui définit et renvoie le numéro de facture suivant,
- une procédure DeleteInvoiceNo, qui supprime complètement la facture CDP, ainsi que
- une procédure showAllCDPs répertoriant la collection CDPs complète avec tous les noms. Sans VBA, vous pouvez également les lister via les informations du classeur: Info | Propriétés [DropDown:] | Propriétés avancées | Douane

Vous pouvez obtenir et définir le numéro de facture suivant (dernier pas plus un) simplement en appelant la fonction mentionnée ci-dessus, en renvoyant une valeur de chaîne afin de faciliter l'ajout de préfixes. "FactureNo" est implicitement utilisé comme nom CDP dans toutes les

procédures.

```
Dim sNumber As String
sNumber = NextInvoiceNo ()
```

Exemple de code:

```
Option Explicit

Sub Test()
  Dim sNumber As String
  sNumber = NextInvoiceNo()
  MsgBox "New Invoice No: " & sNumber, vbInformation, "New Invoice Number"
End Sub

Function NextInvoiceNo() As String
' Purpose: a) Set Custom Document Property (CDP) "InvoiceNo" if not yet existing
'          b) Increment CDP value and return new value as string
' Declarations
  Dim prop As Object
  Dim ret  As String
  Dim wb   As Workbook
' Set workbook and CDPs
  Set wb = ThisWorkbook
  Set prop = wb.CustomDocumentProperties

  ' ------------------------------------------------------
  ' Generate new CDP "InvoiceNo" if not yet existing
  ' ------------------------------------------------------
    If Not CDPExists("InvoiceNo") Then
    '  set temporary starting value "0"
       prop.Add "InvoiceNo", False, msoPropertyTypeString, "0"
    End If

  ' ------------------------------------------------------
  ' Increment invoice no and return function value as string
  ' ------------------------------------------------------
       ret = Format(Val(prop("InvoiceNo")) + 1, "0")
  ' a) Set CDP "InvoiceNo" = ret
       prop("InvoiceNo").value = ret
  ' b) Return function value
       NextInvoiceNo = ret
End Function

Private Function CDPExists(sCDPName As String) As Boolean
' Purpose: return True if custom document property (CDP) exists
' Method: loop thru CustomDocumentProperties collection and check if name parameter exists
' Site: cf. http://stackoverflow.com/questions/23917977/alternatives-to-public-variables-in-
vba/23918236#23918236
' vgl.: https://answers.microsoft.com/en-us/msoffice/forum/msoffice_word-mso_other/using-
customdocumentproperties-with-vba/91ef15eb-b089-4c9b-a8a7-1685d073fb9f
' Declarations
  Dim cdp As Variant      ' element of CustomDocumentProperties Collection
  Dim boo As Boolean      ' boolean value showing element exists
  For Each cdp In ThisWorkbook.CustomDocumentProperties
    If LCase(cdp.Name) = LCase(sCDPName) Then
       boo = True      ' heureka
       Exit For        ' exit loop
    End If
```

```
  Next
  CDPExists = boo          ' return value to function
End Function


Sub DeleteInvoiceNo()
' Declarations
  Dim wb     As Workbook
  Dim prop   As Object
' Set workbook and CDPs
  Set wb = ThisWorkbook
  Set prop = wb.CustomDocumentProperties

' ----------------------
' Delete CDP "InvoiceNo"
' ----------------------
 If CDPExists("InvoiceNo") Then
    prop("InvoiceNo").Delete
 End If
```

End Sub

```
Sub showAllCDPs()
' Purpose: Show all CustomDocumentProperties (CDP) and values (if set)
' Declarations
  Dim wb      As Workbook
  Dim cdp     As Object

  Dim i       As Integer
  Dim maxi    As Integer
  Dim s       As String
' Set workbook and CDPs
  Set wb = ThisWorkbook
  Set cdp = wb.CustomDocumentProperties
' Loop thru CDP getting name and value
  maxi = cdp.Count
  For i = 1 To maxi
    On Error Resume Next    ' necessary in case of unset value
    s = s & Chr(i + 96) & ") " & _
            cdp(i).Name & "=" & cdp(i).value & vbCr
  Next i
' Show result string
  Debug.Print s
End Sub
```

Lire CustomDocumentProperties dans la pratique en ligne: https://riptutorial.com/fr/excel-vba/topic/10932/customdocumentproperties-dans-la-pratique

Chapitre 8: Débogage et dépannage

Syntaxe

- Debug.Print (chaîne)
- Stop STOP

Examples

Debug.Print

Pour imprimer une liste des descriptions de code d'erreur dans la fenêtre immédiate, transmettez-la à la fonction `Debug.Print` :

```
Private Sub ListErrCodes()
    Debug.Print "List Error Code Descriptions"
    For i = 0 To 65535
        e = Error(i)
        If e <> "Application-defined or object-defined error" Then Debug.Print i & ": " & e
    Next i
End Sub
```

Vous pouvez afficher la fenêtre immédiate par:

- Sélection de **V** iew | **Je** mmediate fenêtre dans la barre de menu
- Utiliser le raccourci clavier **Ctrl-G**

Arrêtez

La commande Stop met en pause l'exécution lorsqu'elle est appelée. De là, le processus peut être repris ou exécuté pas à pas.

```
Sub Test()
    Dim TestVar as String
    TestVar = "Hello World"
    Stop                    'Sub will be executed to this point and then wait for the user
    MsgBox TestVar
End Sub
```

Fenêtre Immédiate

Si vous souhaitez tester une ligne de code macro sans avoir à exécuter un sous-ensemble, vous pouvez taper des commandes directement dans la fenêtre Immédiat et `ENTER` sur `ENTER` pour exécuter la ligne.

Pour tester la sortie d'une ligne, vous pouvez la faire précéder d'un point d'interrogation `?` pour imprimer directement dans la fenêtre immédiate. Vous pouvez également utiliser la commande

`print` pour `print` la sortie.

Dans Visual Basic Editor, appuyez sur `CTRL + G` pour ouvrir la fenêtre immédiate. Pour renommer votre feuille actuellement sélectionnée en "ExampleSheet", tapez ce qui suit dans la fenêtre Immédiat et `ENTER` sur `ENTER`

```
   ActiveSheet.Name = "ExampleSheet"
```

Pour imprimer le nom de la feuille actuellement sélectionnée directement dans la fenêtre immédiate

```
? ActiveSheet.Name
ExampleSheet
```

Cette méthode peut être très utile pour tester les fonctionnalités des fonctions intégrées ou définies par l'utilisateur avant de les implémenter dans du code. L'exemple ci-dessous montre comment utiliser la fenêtre Immédiat pour tester la sortie d'une fonction ou d'une série de fonctions afin de confirmer un résultat attendu.

```
'In this example, the Immediate Window was used to confirm that a series of Left and Right
'string methods would return the desired string

'expected output: "value"
print Left(Right("1111value1111",9),5) ' <---- written code here, ENTER pressed
value                                  ' <---- output
```

La fenêtre Immédiat peut également être utilisée pour définir ou réinitialiser une application, un classeur ou d'autres propriétés nécessaires. Cela peut être utile si vous avez `Application.EnableEvents = False` dans un sous-programme qui génère une erreur de manière inattendue, ce qui provoque sa fermeture sans réinitialiser la valeur sur `True` (ce qui peut provoquer des fonctionnalités frustrantes et inattendues. Dans ce cas, les commandes peuvent être saisies directement). dans la fenêtre immédiate et exécutez:

```
? Application.EnableEvents       ' <---- Testing the current state of "EnableEvents"
False                            ' <---- Output
Application.EnableEvents = True  ' <---- Resetting the property value to True
? Application.EnableEvents       ' <---- Testing the current state of "EnableEvents"
True                             ' <---- Output
```

Pour les techniques de débogage plus avancées, les deux points `:` peuvent être utilisés comme séparateur de ligne. Cela peut être utilisé pour des expressions multi-lignes telles que la mise en boucle dans l'exemple ci-dessous.

```
x = Split("a,b,c",","): For i = LBound(x,1) to UBound(x,1): Debug.Print x(i): Next i '<----
Input this and press enter
a '<----Output
b '<----Output
c '<----Output
```

Utiliser la minuterie pour trouver des goulots d'étranglement dans les

performances

La première étape pour optimiser la vitesse consiste à trouver les sections de code les plus lentes. La fonction `Timer` VBA renvoie le nombre de secondes écoulées depuis minuit avec une précision de 1 / 256ème de seconde (3,90625 millisecondes) sur les PC Windows. Les fonctions VBA `Now` et `Time` ne sont précises qu'à une seconde.

```
Dim start As Double       ' Timer returns Single, but converting to Double to avoid
start = Timer             ' scientific notation like 3.90625E-03 in the Immediate window
' ... part of the code
Debug.Print Timer - start; "seconds in part 1"

start = Timer
' ... another part of the code
Debug.Print Timer - start; "seconds in part 2"
```

Ajouter un point d'arrêt à votre code

Vous pouvez facilement ajouter un point d'arrêt à votre code en cliquant sur la colonne grise située à gauche de la ligne de votre code VBA où vous souhaitez arrêter l'exécution. Un point rouge apparaît dans la colonne et le code du point d'arrêt est également surligné en rouge.

Vous pouvez ajouter plusieurs points d'arrêt dans votre code et reprendre l'exécution en appuyant sur l'icône "play" dans la barre de menus. Tout le code ne peut pas être un point d'arrêt en tant que lignes de définition de variable, la première ou la dernière ligne d'une procédure et les lignes de commentaires ne peuvent pas être sélectionnées en tant que point d'arrêt.

Fenêtre locale de débogueur

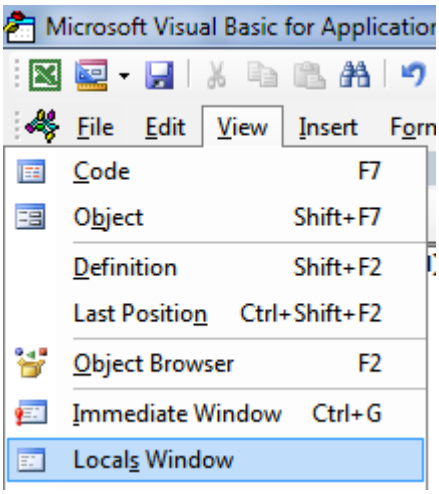
La fenêtre Locaux permet d'accéder facilement à la valeur actuelle des variables et des objets dans la portée de la fonction ou du sous-programme que vous exécutez. C'est un outil essentiel

pour déboguer votre code et passer en revue les modifications afin de trouver des problèmes. Cela vous permet également d'explorer les propriétés que vous ne connaissiez peut-être pas.

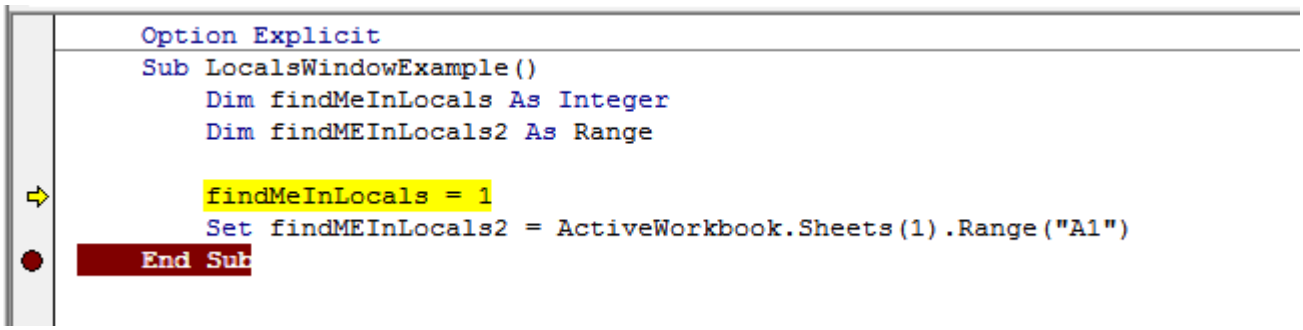
Prenons l'exemple suivant:

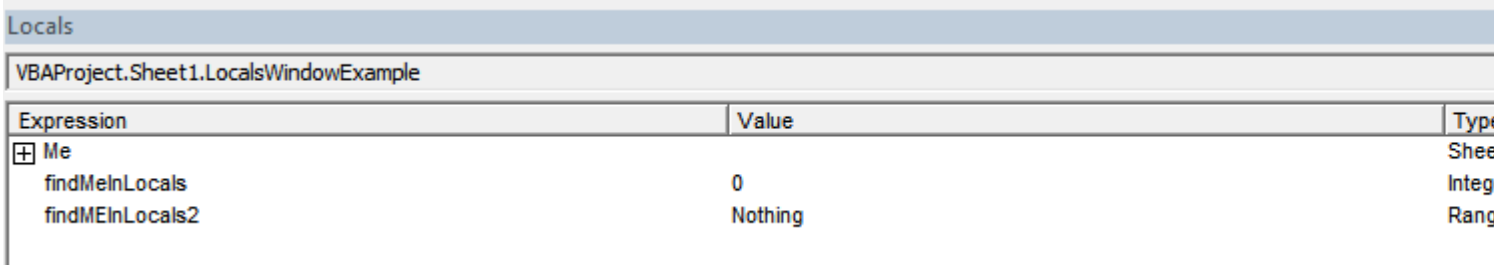
```
Option Explicit
Sub LocalsWindowExample()
    Dim findMeInLocals As Integer
    Dim findMEInLocals2 As Range

    findMeInLocals = 1
    Set findMEInLocals2 = ActiveWorkbook.Sheets(1).Range("A1")
End Sub
```

Dans l'éditeur VBA, cliquez sur Afficher -> Fenêtre locale



Ensuite, en parcourant le code en utilisant F8 après avoir cliqué dans le sous-programme, nous nous sommes arrêtés avant d'avoir assigné findMeinLocals. Vous pouvez voir ci-dessous que la valeur est 0 --- et c'est ce qui serait utilisé si vous ne lui avez jamais attribué de valeur. L'objet Range est "Nothing".





Si nous nous arrêtons juste avant la fin du sous-programme, nous pouvons voir les valeurs finales des variables.

```
Option Explicit
Sub LocalsWindowExample()
    Dim findMeInLocals As Integer
    Dim findMEInLocals2 As Range

    findMeInLocals = 1
    Set findMEInLocals2 = ActiveWorkbook.Sheets(1).Range("A1")
End Sub
```

Nous pouvons voir findMeInLocals avec la valeur 1 et le type Integer, et FindMeInLocals2 avec un type Range / Range. Si nous cliquons sur le signe +, nous pouvons développer l'objet et voir ses propriétés, telles que le nombre ou la colonne.

Locals

VBAProject.Sheet1.LocalsWindowExample

Expression	Value	Typ
⊞ Me		Shee
findMeInLocals	1	Integ
⊟ findMEInLocals2		Rang
AddIndent	False	Varia
AllowEdit	True	Bool
⊞ Application		Appl
⊞ Areas		Area
⊞ Borders		Bord
⊞ Cells		Rang
Column	1	Long
ColumnWidth	8.43	Varia
Comment	Nothing	Com
Count	1	Long
CountLarge	1	Varia
Creator	xlCreatorCode	XlCr
CurrentArray	<No cells were found.>	Rang
⊞ CurrentRegion		Rang
Dependents	<No cells were found.>	Rang
DirectDependents	<No cells were found.>	Rang
DirectPrecedents	<No cells were found.>	Rang
⊞ DisplayFormat		Displ

Lire Débogage et dépannage en ligne: https://riptutorial.com/fr/excel-vba/topic/861/debogage-et-depannage

Chapitre 9: Erreurs courantes

Examples

Références éligibles

En faisant référence à une `worksheet` , à une `range` ou à des `cells` individuelles, il est important de qualifier complètement la référence.

Par exemple:

```
ThisWorkbook.Worksheets("Sheet1").Range(Cells(1, 2), Cells(2, 3)).Copy
```

N'est pas entièrement qualifié: les références de `Cells` ne sont pas associées à un classeur et à une feuille de calcul. Sans référence explicite, Cells fait référence à `ActiveSheet` par défaut. Ce code échouera donc (produira des résultats incorrects) si une feuille de calcul autre que `Sheet1` est l'actuelle `ActiveSheet` .

La manière la plus simple de corriger cela est d'utiliser une instruction `With` comme suit:

```
With ThisWorkbook.Worksheets("Sheet1")
    .Range(.Cells(1, 2), .Cells(2, 3)).Copy
End With
```

Vous pouvez également utiliser une variable de feuille de calcul. (Cela sera probablement la méthode préférée si votre code doit référencer plusieurs feuilles de calcul, comme la copie de données d'une feuille à une autre.)

```
Dim ws1 As Worksheet
Set ws1 = ThisWorkbook.Worksheets("Sheet1")
ws1.Range(ws1.Cells(1, 2), ws1.Cells(2, 3)).Copy
```

Un autre problème fréquent est le référencement de la collection Worksheets sans qualifier le classeur. Par exemple:

```
Worksheets("Sheet1").Copy
```

La feuille de calcul `Sheet1` n'est pas entièrement qualifiée et ne contient pas de classeur. Cela peut échouer si plusieurs classeurs sont référencés dans le code. Utilisez plutôt l'une des options suivantes:

```
ThisWorkbook.Worksheets("Sheet1")       '<--ThisWorkbook refers to the workbook containing
                                        'the running VBA code
Workbooks("Book1").Worksheets("Sheet1") '<--Where Book1 is the workbook containing Sheet1
```

Cependant, évitez d'utiliser les éléments suivants:

```
ActiveWorkbook.Worksheets("Sheet1")     '<--Valid, but if another workbook is activated
                                        'the reference will be changed
```

De même pour les objets de `range` , s'ils ne sont pas explicitement qualifiés, la `range` se réfère à la feuille actuellement active:

```
Range("a1")
```

Est le même que:

```
ActiveSheet.Range("a1")
```

Suppression de lignes ou de colonnes dans une boucle

Si vous souhaitez supprimer des lignes (ou des colonnes) dans une boucle, vous devez toujours effectuer une boucle à partir de la fin de la plage et revenir à chaque étape. En cas d'utilisation du code:

```
Dim i As Long
With Workbooks("Book1").Worksheets("Sheet1")
    For i = 1 To 4
        If IsEmpty(.Cells(i, 1)) Then .Rows(i).Delete
    Next i
End With
```

Vous allez manquer certaines lignes. Par exemple, si le code supprime la ligne 3, la ligne 4 devient la ligne 3. Cependant, la variable `i` passera à 4. Ainsi, dans ce cas, le code manquera une ligne et en vérifiera une autre, qui n'était pas dans la plage précédente.

Le bon code serait

```
Dim i As Long
With Workbooks("Book1").Worksheets("Sheet1")
    For i = 4 To 1 Step -1
        If IsEmpty(.Cells(i, 1)) Then .Rows(i).Delete
    Next i
End With
```

ActiveWorkbook vs. ThisWorkbook

`ActiveWorkbook` et `ThisWorkbook` parfois utilisés indifféremment par les nouveaux utilisateurs de VBA sans comprendre parfaitement à quoi chaque objet se rapporte, cela peut entraîner un comportement indésirable au moment de l'exécution. Ces deux objets appartiennent à l' objet d'application

---

L'objet `ActiveWorkbook` fait référence au classeur qui se trouve actuellement dans la vue la plus haute de l'objet d'application Excel au moment de l'exécution. *(Par exemple, le classeur avec lequel vous pouvez voir et interagir au moment où cet objet est référencé)*

```
Sub ActiveWorkbookExample()

'// Let's assume that 'Other Workbook.xlsx' has "Bar" written in A1.

    ActiveWorkbook.ActiveSheet.Range("A1").Value = "Foo"
    Debug.Print ActiveWorkbook.ActiveSheet.Range("A1").Value '// Prints "Foo"

    Workbooks.Open("C:\Users\BloggsJ\Other Workbook.xlsx")
    Debug.Print ActiveWorkbook.ActiveSheet.Range("A1").Value '// Prints "Bar"

    Workbooks.Add 1
    Debug.Print ActiveWorkbook.ActiveSheet.Range("A1").Value '// Prints nothing

End Sub
```

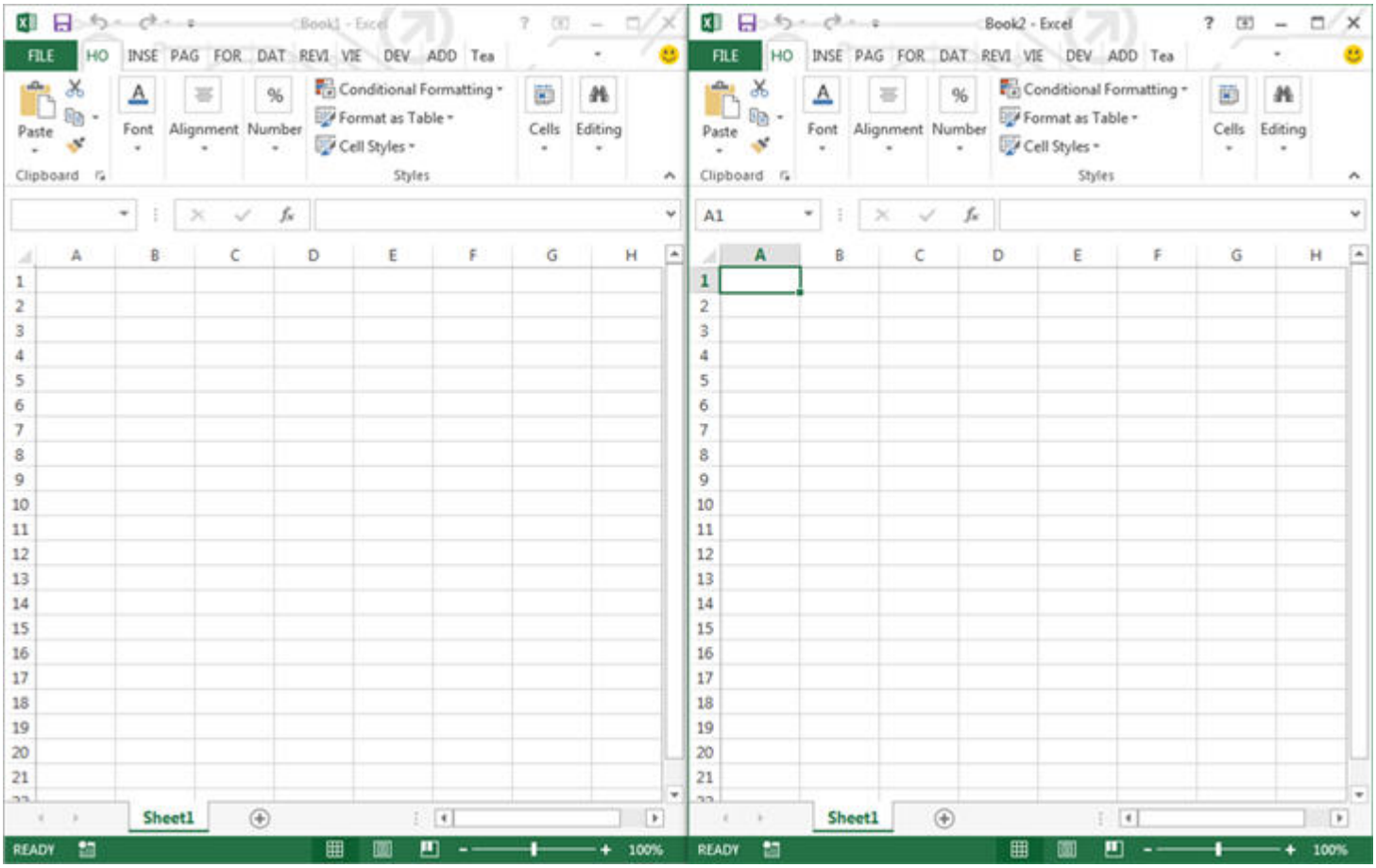
L'objet `ThisWorkbook` fait référence au classeur auquel appartient le code au moment de son exécution.

```
Sub ThisWorkbookExample()

'// Let's assume to begin that this code is in the same workbook that is currently active

    ActiveWorkbook.Sheet1.Range("A1").Value = "Foo"
    Workbooks.Add 1
    ActiveWorkbook.ActiveSheet.Range("A1").Value = "Bar"

    Debug.Print ActiveWorkbook.ActiveSheet.Range("A1").Value '// Prints "Bar"
    Debug.Print ThisWorkbook.Sheet1.Range("A1").Value '// Prints "Foo"

End Sub
```

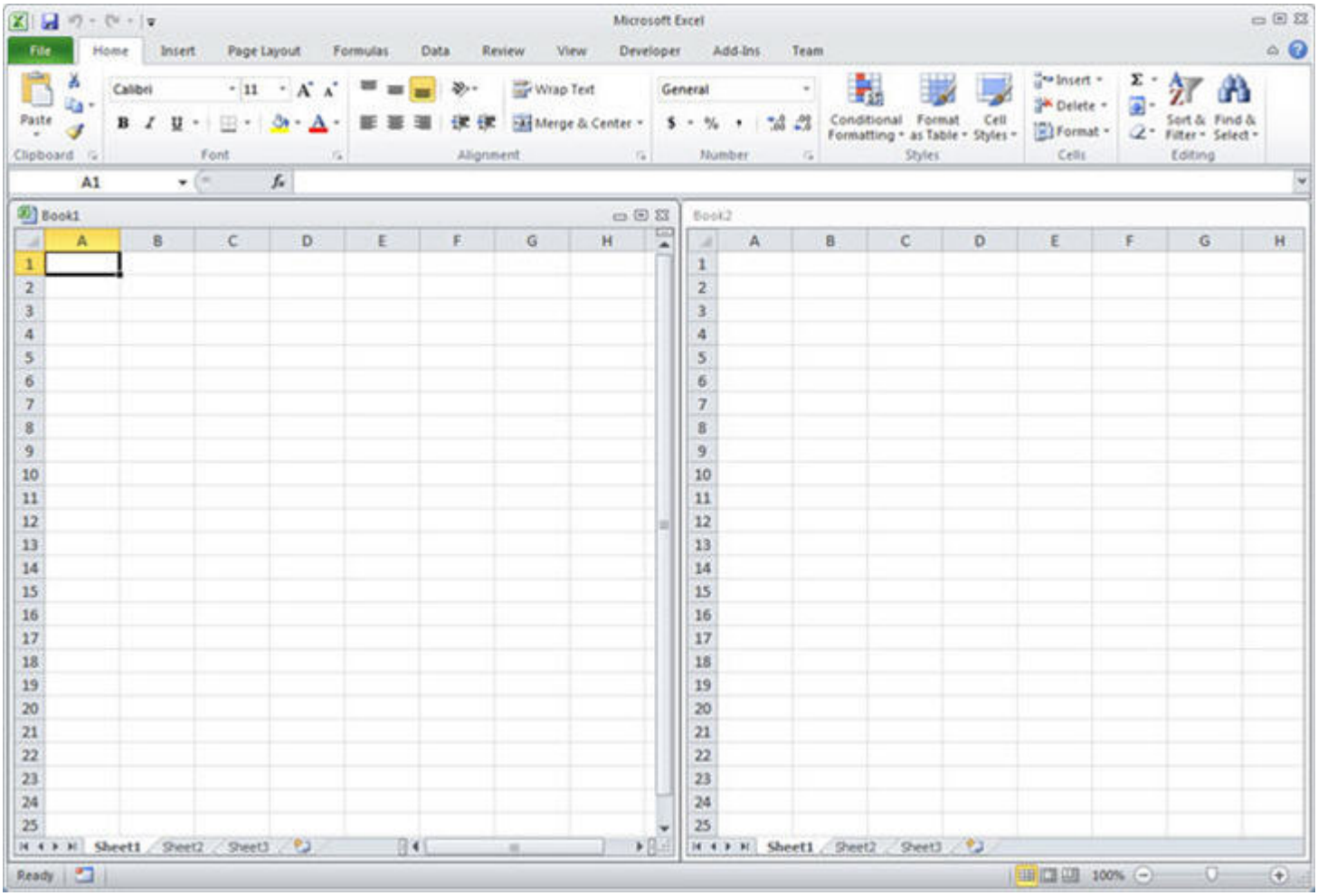
Interface de document unique et interfaces de documents multiples

> Sachez que Microsoft Excel 2013 (et versions ultérieures) utilise l'interface SDI (Single Document Interface) et qu'Excel 2010 (et ci-dessous) utilise plusieurs interfaces de document (MDI).

Cela implique que pour Excel 2013 (SDI), chaque classeur d'une seule instance d'Excel contient sa **propre** interface utilisateur de ruban:



A l' inverse pour Excel 2010, chaque classeur dans une seule instance d'Excel utilise une interface de ruban **commun** (MDI):



Cela pose des problèmes importants si vous souhaitez migrer un code VBA (2010 <-> 2013) qui interagit avec le ruban.

> Une procédure doit être créée pour mettre à jour les contrôles de l'interface utilisateur du ruban dans le même état sur tous les classeurs pour Excel 2013 et versions ultérieures.

Notez que :

1. Toutes les méthodes, événements et propriétés de fenêtre au niveau de l'application Excel restent inchangés. (`Application.ActiveWindow` , `Application.Windows` ...)
2. Dans Excel 2013 et versions ultérieures (SDI), toutes les méthodes, tous les événements et toutes les propriétés de la fenêtre au niveau du classeur fonctionnent désormais dans la fenêtre de niveau supérieur. Il est possible de récupérer le descripteur de cette fenêtre de niveau supérieur avec `Application.Hwnd`

Pour plus de détails, voir la source de cet exemple: MSDN .

Cela provoque également des problèmes avec les formes utilisateur sans modèle. Voir ici pour une solution.

Lire Erreurs courantes en ligne: https://riptutorial.com/fr/excel-vba/topic/1576/erreurs-courantes

Chapitre 10: Expressions conditionnelles

Examples

La déclaration If

L'instruction `If` control permet d'exécuter du code différent en fonction de l'évaluation d'une instruction conditionnelle (booléenne). Une instruction conditionnelle est une instruction qui a la valeur `True` ou `False` , par exemple `x > 2` .

Trois modèles peuvent être utilisés lors de l'implémentation d'une instruction `If` , décrits ci-dessous. Notez qu'une évaluation conditionnelle `If` est toujours suivie par une évaluation `Then` .

1. Evaluer un énoncé conditionnel `If` et faire quelque chose si c'est `True`

Une seule ligne `If` déclaration

C'est le moyen le plus court d'utiliser un `If` et il est utile quand une seule instruction doit être effectuée sur une évaluation `True` . Lorsque vous utilisez cette syntaxe, tout le code doit être sur une seule ligne. N'incluez pas de `End If` à la fin de la ligne.

```
If [Some condition is True] Then [Do something]
```

`If` bloquer

Si plusieurs lignes de code doivent être exécutées lors d'une évaluation `True` , un bloc `If` peut être utilisé.

```
If [Some condition is True] Then
  [Do some things]
End If
```

Notez que si un bloc `If` multi-lignes est utilisé, une `End If` correspondante est requise.

2. Évaluer une instruction `If` conditionnelle, faire une chose si elle est `True` et faire autre chose si elle est `False`

Ligne unique `If` , déclaration `Else`

Cela peut être utilisé si une déclaration doit être effectuée lors d'une évaluation `True` et une déclaration différente doit être effectuée sur une évaluation `False` . Soyez prudent en utilisant cette syntaxe, car il est souvent moins clair pour les lecteurs qu'il existe une déclaration `Else` . Lorsque vous utilisez cette syntaxe, tout le code doit être sur une seule ligne. N'incluez pas de `End If` à la fin de la ligne.

```
If [Some condition is True] Then [Do something] Else [Do something else]
```

`If` , `Else` bloc

Utilisez un bloc `If` , `Else` pour ajouter de la clarté à votre code ou si plusieurs lignes de code doivent être exécutées sous une évaluation `True` ou `False` .

```
If [Some condition is True] Then
   [Do some things]
Else
   [Do some other things]
End If
```

Notez que si un bloc `If` multi-lignes est utilisé, une `End If` correspondante est requise.

3. Évaluer de nombreuses instructions conditionnelles, lorsque les instructions précédentes sont toutes `False` , et faire quelque chose de différent pour chacune

Ce modèle est l'utilisation la plus générale de `If` et serait utilisé lorsqu'il existe de nombreuses conditions qui ne se chevauchent pas et qui nécessitent un traitement différent. Contrairement aux deux premiers modèles, ce cas nécessite l'utilisation d'un bloc `If` , même si une seule ligne de code sera exécutée pour chaque condition.

`If` , `ElseIf` , `...` , d' `Else` bloc

Au lieu de devoir créer plusieurs blocs `If` uns sous les autres, un `ElseIf` peut être utilisé pour évaluer une condition supplémentaire. `ElseIf` n'est évalué que s'il y a un précédent `If` évaluation est `False` .

```
If [Some condition is True] Then
   [Do some thing(s)]
ElseIf [Some other condition is True] Then
   [Do some different thing(s)]
Else    'Everything above has evaluated to False
   [Do some other thing(s)]
End If
```

Autant d' `ElseIf` contrôle `ElseIf` peuvent être incluses entre un `If` et un `End If` si nécessaire. Une instruction de contrôle `Else` n'est pas requise lors de l'utilisation d' `ElseIf` (même si elle est recommandée), mais si elle est incluse, elle doit être l'instruction de contrôle finale avant `End If` .

Lire Expressions conditionnelles en ligne: https://riptutorial.com/fr/excel-vba/topic/9632/expressions-conditionnelles

Chapitre 11: filtre automatique; Utilisations et meilleures pratiques

Introduction

Le but ultime du ***filtre automatique*** est de fournir le plus rapidement possible l'extraction de données à partir de centaines ou de milliers de données de lignes afin d'attirer l'attention sur les éléments sur lesquels nous voulons nous concentrer. Il peut recevoir des paramètres tels que "text / values / colors" et ils peuvent être empilés entre des colonnes. Vous pouvez connecter jusqu'à 2 critères par colonne en fonction des connecteurs logiques et des ensembles de règles. Remarque: le filtrage automatique fonctionne en filtrant les lignes, il n'ya pas de filtre automatique pour filtrer les colonnes (du moins pas de manière native).

Remarques

'Pour utiliser Autofilter dans VBA, nous devons appeler avec au moins les paramètres suivants:

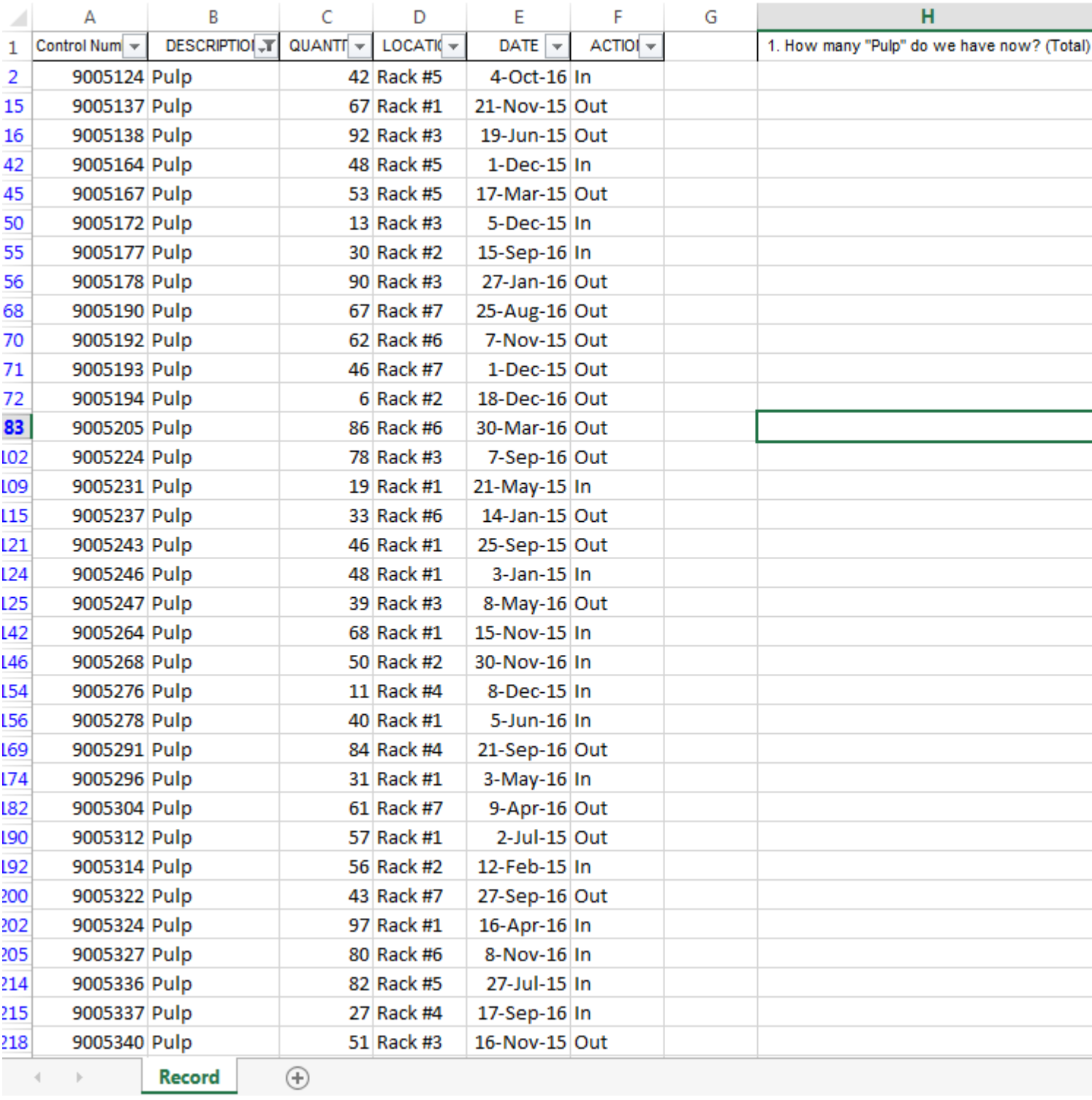
Feuille ("MySheet"). Plage ("MyRange"). Champ Autofilter = (ColumnNumberWithin "MyRange" ToBeFilteredInNumericValue) Criteria1: = "WhatIWantToFilter"

«Il existe de nombreux exemples sur le Web ou ici à stackoverflow

Examples

Smartfilter!

Problème situation
L'administrateur de l'entrepôt a une feuille ("Enregistrement") dans laquelle chaque mouvement logistique effectué par l'installation est stocké, il peut filtrer au besoin, mais cela prend beaucoup de temps et il aimerait améliorer le processus afin de calculer les demandes plus rapidement. exemple: Combien de "pulpes" avons-nous maintenant (dans tous les racks)? Combien de pâte avons-nous maintenant (dans le rack # 5)? Les filtres sont un excellent outil, mais ils sont quelque peu limités pour répondre à ce genre de questions en quelques secondes.



	A	B	C	D	E	F	G	H
1	Control Num	DESCRIPTIO	QUANTI	LOCATI	DATE	ACTIO		1. How many "Pulp" do we have now? (Total)
2	9005124	Pulp	42	Rack #5	4-Oct-16	In		
15	9005137	Pulp	67	Rack #1	21-Nov-15	Out		
16	9005138	Pulp	92	Rack #3	19-Jun-15	Out		
42	9005164	Pulp	48	Rack #5	1-Dec-15	In		
45	9005167	Pulp	53	Rack #5	17-Mar-15	Out		
50	9005172	Pulp	13	Rack #3	5-Dec-15	In		
55	9005177	Pulp	30	Rack #2	15-Sep-16	In		
56	9005178	Pulp	90	Rack #3	27-Jan-16	Out		
68	9005190	Pulp	67	Rack #7	25-Aug-16	Out		
70	9005192	Pulp	62	Rack #6	7-Nov-15	Out		
71	9005193	Pulp	46	Rack #7	1-Dec-15	Out		
72	9005194	Pulp	6	Rack #2	18-Dec-16	Out		
83	9005205	Pulp	86	Rack #6	30-Mar-16	Out		
102	9005224	Pulp	78	Rack #3	7-Sep-16	Out		
109	9005231	Pulp	19	Rack #1	21-May-15	In		
115	9005237	Pulp	33	Rack #6	14-Jan-15	Out		
121	9005243	Pulp	46	Rack #1	25-Sep-15	Out		
124	9005246	Pulp	48	Rack #1	3-Jan-15	In		
125	9005247	Pulp	39	Rack #3	8-May-16	Out		
142	9005264	Pulp	68	Rack #1	15-Nov-15	In		
146	9005268	Pulp	50	Rack #2	30-Nov-16	In		
154	9005276	Pulp	11	Rack #4	8-Dec-15	In		
156	9005278	Pulp	40	Rack #1	5-Jun-16	In		
169	9005291	Pulp	84	Rack #4	21-Sep-16	Out		
174	9005296	Pulp	31	Rack #1	3-May-16	In		
182	9005304	Pulp	61	Rack #7	9-Apr-16	Out		
190	9005312	Pulp	57	Rack #1	2-Jul-15	Out		
192	9005314	Pulp	56	Rack #2	12-Feb-15	In		
200	9005322	Pulp	43	Rack #7	27-Sep-16	Out		
202	9005324	Pulp	97	Rack #1	16-Apr-16	In		
205	9005327	Pulp	80	Rack #6	8-Nov-16	In		
214	9005336	Pulp	82	Rack #5	27-Jul-15	In		
215	9005337	Pulp	27	Rack #4	17-Sep-16	In		
218	9005340	Pulp	51	Rack #3	16-Nov-15	Out		

Record

Solution macro:

Le codeur sait que les ***filtres automatiques sont la solution la meilleure, la plus rapide et la plus fiable*** dans ce type de scénario car ***les données existent déjà dans la feuille de calcul*** et leur ***saisie peut être facilement obtenue*** - dans ce cas, par ***saisie*** utilisateur.

L'approche utilisée consiste à créer une feuille appelée "SmartFilter" où l'administrateur peut facilement filtrer plusieurs données selon les besoins et le calcul sera également effectué instantanément.

Il utilise 2 modules et l'événement `Worksheet_Change` pour cette question

Code pour feuille de calcul SmartFilter:

```
Private Sub Worksheet_Change(ByVal Target As Range)
Dim ItemInRange As Range
Const CellsFilters As String = "C2,E2,G2"
    Call ExcelBusy
    For Each ItemInRange In Target
    If Not Intersect(ItemInRange, Range(CellsFilters)) Is Nothing Then Call Inventory_Filter
    Next ItemInRange
    Call ExcelNormal
End Sub
```

Code pour le module 1, appelé "Fonctions générales"

```
Sub ExcelNormal()
        With Excel.Application
        .EnableEvents = True
        .Cursor = xlDefault
        .ScreenUpdating = True
        .DisplayAlerts = True
        .StatusBar = False
        .CopyObjectsWithCells = True
        End With
End Sub
Sub ExcelBusy()
        With Excel.Application
        .EnableEvents = False
        .Cursor = xlWait
        .ScreenUpdating = False
        .DisplayAlerts = False
        .StatusBar = False
        .CopyObjectsWithCells = True
        End With
End Sub
Sub Select_Sheet(NameSheet As String, Optional VerifyExistanceOnly As Boolean)
    On Error GoTo Err01Select_Sheet
    Sheets(NameSheet).Visible = True
    If VerifyExistanceOnly = False Then ' 1. If VerifyExistanceOnly = False
    Sheets(NameSheet).Select
    Sheets(NameSheet).AutoFilterMode = False
    Sheets(NameSheet).Cells.EntireRow.Hidden = False
    Sheets(NameSheet).Cells.EntireColumn.Hidden = False
    End If ' 1. If VerifyExistanceOnly = False
    If 1 = 2 Then '99. If error
Err01Select_Sheet:
    MsgBox "Err01Select_Sheet: Sheet " & NameSheet & " doesn't exist!", vbCritical: Call
ExcelNormal: On Error GoTo -1: End
    End If '99. If error
End Sub
Function General_Functions_Find_Title(InSheet As String, TitleToFind As String, Optional
InRange As Range, Optional IsNeededToExist As Boolean, Optional IsWhole As Boolean) As Range
Dim DummyRange As Range
    On Error GoTo Err01General_Functions_Find_Title
    If InRange Is Nothing Then ' 1. If InRange Is Nothing
    Set DummyRange = IIf(IsWhole = True, Sheets(InSheet).Cells.Find(TitleToFind,
LookAt:=xlWhole), Sheets(InSheet).Cells.Find(TitleToFind, LookAt:=xlPart))
    Else ' 1. If InRange Is Nothing
    Set DummyRange = IIf(IsWhole = True,
Sheets(InSheet).Range(InRange.Address).Find(TitleToFind, LookAt:=xlWhole),
```

```
Sheets(InSheet).Range(InRange.Address).Find(TitleToFind, LookAt:=xlPart))
    End If ' 1. If InRange Is Nothing
    Set General_Functions_Find_Title = DummyRange
    If 1 = 2 Or DummyRange Is Nothing Then '99. If error
Err01General_Functions_Find_Title:
    If IsNeededToExist = True Then MsgBox "Err01General_Functions_Find_Title: Ttile '" &
TitleToFind & "' was not found in sheet '" & InSheet & "'", vbCritical: Call ExcelNormal: On
Error GoTo -1: End
    End If '99. If error
End Function
```

Code pour le module 2, appelé "Inventory_Handling"

```
Const TitleDesc As String = "DESCRIPTION"
Const TitleLocation As String = "LOCATION"
Const TitleActn As String = "ACTION"
Const TitleQty As String = "QUANTITY"
Const SheetRecords As String = "Record"
Const SheetSmartFilter As String = "SmartFilter"
Const RowFilter As Long = 2
Const ColDataToPaste As Long = 2
Const RowDataToPaste As Long = 7
Const RangeInResult As String = "K1"
Const RangeOutResult As String = "K2"
Sub Inventory_Filter()
Dim ColDesc As Long: ColDesc = General_Functions_Find_Title(SheetSmartFilter, TitleDesc,
IsNeededToExist:=True, IsWhole:=True).Column
Dim ColLocation As Long: ColLocation = General_Functions_Find_Title(SheetSmartFilter,
TitleLocation, IsNeededToExist:=True, IsWhole:=True).Column
Dim ColActn As Long: ColActn = General_Functions_Find_Title(SheetSmartFilter, TitleActn,
IsNeededToExist:=True, IsWhole:=True).Column
Dim ColQty As Long: ColQty = General_Functions_Find_Title(SheetSmartFilter, TitleQty,
IsNeededToExist:=True, IsWhole:=True).Column
Dim CounterQty As Long
Dim TotalQty As Long
Dim TotalIn As Long
Dim TotalOut As Long
Dim RangeFiltered As Range
    Call Select_Sheet(SheetSmartFilter)
    If Cells(Rows.Count, ColDataToPaste).End(xlUp).Row > RowDataToPaste - 1 Then
Rows(RowDataToPaste & ":" & Cells(Rows.Count, "B").End(xlUp).Row).Delete
    Sheets(SheetRecords).AutoFilterMode = False
    If Cells(RowFilter, ColDesc).Value <> "" Or Cells(RowFilter, ColLocation).Value <> "" Or
Cells(RowFilter, ColActn).Value <> "" Then ' 1. If Cells(RowFilter, ColDesc).Value <> "" Or
Cells(RowFilter, ColLocation).Value <> "" Or Cells(RowFilter, ColActn).Value <> ""
    With Sheets(SheetRecords).UsedRange
    If Sheets(SheetSmartFilter).Cells(RowFilter, ColDesc).Value <> "" Then .AutoFilter
Field:=General_Functions_Find_Title(SheetRecords, TitleDesc, IsNeededToExist:=True,
IsWhole:=True).Column, Criteria1:=Sheets(SheetSmartFilter).Cells(RowFilter, ColDesc).Value
    If Sheets(SheetSmartFilter).Cells(RowFilter, ColLocation).Value <> "" Then .AutoFilter
Field:=General_Functions_Find_Title(SheetRecords, TitleLocation, IsNeededToExist:=True,
IsWhole:=True).Column, Criteria1:=Sheets(SheetSmartFilter).Cells(RowFilter, ColLocation).Value
    If Sheets(SheetSmartFilter).Cells(RowFilter, ColActn).Value <> "" Then .AutoFilter
Field:=General_Functions_Find_Title(SheetRecords, TitleActn, IsNeededToExist:=True,
IsWhole:=True).Column, Criteria1:=Sheets(SheetSmartFilter).Cells(RowFilter, ColActn).Value
    'If we don't use a filter we would need to use a cycle For/to or For/Each Cell in range
    'to determine whether or not the row meets the criteria that we are looking and then
    'save it on an array, collection, dictionary, etc
    'IG: For CounterRow = 2 To TotalRows
    'If Sheets(SheetSmartFilter).Cells(RowFilter, ColDesc).Value <> "" and
```

```
Sheets(SheetRecords).cells(CounterRow,ColDescInRecords).Value=
Sheets(SheetSmartFilter).Cells(RowFilter, ColDesc).Value then
    'Redim Preserve MyUnecessaryArray(UnecessaryNumber) ''Save to array:
(UnecessaryNumber)=MyUnecessaryArray. Or in a dictionary, etc. At the end, we would transpose
this values into the sheet, at the end
    'both are the same, but, just try to see the time invested on each logic.
    If .Cells(1, 1).End(xlDown).Value <> "" Then Set RangeFiltered = .Rows("2:" &
Sheets(SheetRecords).Cells(Rows.Count, "A").End(xlUp).Row).SpecialCells(xlCellTypeVisible)
    'If it is not <>"" means that there was not filtered data!
    If RangeFiltered Is Nothing Then MsgBox "Err01Inventory_Filter: No data was found with the
given criteria!", vbCritical: Call ExcelNormal: End
    RangeFiltered.Copy Destination:=Cells(RowDataToPaste, ColDataToPaste)
    TotalQty = Cells(Rows.Count, ColQty).End(xlUp).Row
    For CounterQty = RowDataToPaste + 1 To TotalQty
    If Cells(CounterQty, ColActn).Value = "In" Then ' 2. If Cells(CounterQty, ColActn).Value =
"In"
    TotalIn = Cells(CounterQty, ColQty).Value + TotalIn
    ElseIf Cells(CounterQty, ColActn).Value = "Out" Then ' 2. If Cells(CounterQty,
ColActn).Value = "In"
    TotalOut = Cells(CounterQty, ColQty).Value + TotalOut
    End If ' 2. If Cells(CounterQty, ColActn).Value = "In"
    Next CounterQty
    Range(RangeInResult).Value = TotalIn
    Range(RangeOutResult).Value = -(TotalOut)
    End With
    End If ' 1. If Cells(RowFilter, ColDesc).Value <> "" Or Cells(RowFilter,
ColLocation).Value <> "" Or Cells(RowFilter, ColActn).Value <> ""
End Sub
```

Tests et résultats:

	A	B	C	D	E	F	G	H	I	J	K
912	9013034	Batch weight	21	Rack #1	9-Jun-16	Out					
913	9013035	Pectin	72	Rack #7	22-Jun-16	In					
914	9013036	Sugar	28	Rack #1	5-Aug-15	In					
915	9013037	Solids content	51	Rack #7	11-Sep-16	In					
916	9013038	Pulp	45	Rack #3	9-Apr-16	Out					
917	9013039	Batch weight	19	Rack #4	6-Apr-15	Out					
918	9013040	Citric Acid	98	Rack #4	17-Jun-16	Out					
919	9013041	Citric Acid	97	Rack #1	29-Feb-16	In					
920	9013042	Pulp	57	Rack #5	25-Nov-16	Out					
921	9013043	Citric Acid	42	Rack #2	27-Feb-16	In					
922	9013044	Batch weight	54	Rack #1	16-Sep-15	Out					
923	9013045	Solids content	12	Rack #4	13-Jul-15	In					
924	9013046	Pulp	79	Rack #4	13-Jul-15	Out					
925	9013047	Citric Acid	36	Rack #4	15-Nov-16	Out					
926	9013048	Sugar	35	Rack #3	5-Feb-16	Out					
927	9013049	Pulp	63	Rack #6	16-Dec-16	Out					
928	9013050	Solids content	48	Rack #4	1-Mar-15	In					
929	9013051	Pulp	39	Rack #4	31-May-16	Out					
930	9013052	Pulp	47	Rack #6	26-Feb-16	In					
931	9013053	Sugar	6	Rack #6	3-Mar-16	Out					
932	9013054	Pulp	53	Rack #2	11-Sep-15	Out					
933	9013055	Solids content	87	Rack #4	19-Jan-15	Out					
934	9013056	Sugar	48	Rack #7	23-Nov-16	In					
935	9013057	Solids content	62	Rack #6	15-May-16	Out					
936	9013058	Batch weight	61	Rack #3	3-Dec-16	Out					
937	9013059	Citric Acid	64	Rack #7	7-Feb-16	Out					
938	9013060	Sugar	91	Rack #7	23-Sep-15	Out					
939	9013061	Citric Acid	29	Rack #1	7-Jul-16	Out					
940	9013062	Citric Acid	31	Rack #6	17-Feb-16	In					
941	9013063	Batch weight	53	Rack #1	5-Apr-15	Out					
942	9013064	Citric Acid	25	Rack #6	30-Jul-15	Out					
943	9013065	Citric Acid	68	Rack #4	22-Mar-16	Out					
944	9013066	Boiling time	22	Rack #6	17-Jun-15	In					
945	9013067	Pectin	99	Rack #2	2-Nov-16	Out					
946	9013068	Solids content	79	Rack #2	17-Nov-16	Out					

SmartFilter | Record

Comme nous l'avons vu dans l'image précédente, cette tâche a été réalisée facilement. En utilisant des ***filtres automatiques,*** une solution a été fournie qui ne ***prend*** que quelques ***secondes à calculer, est facile à expliquer à l'utilisateur -*** depuis qu'il / elle est familiarisé avec cette commande - et a ***pris quelques lignes pour le codeur.***

Lire filtre automatique; Utilisations et meilleures pratiques en ligne: https://riptutorial.com/fr/excel-vba/topic/8645/filtre-automatique--utilisations-et-meilleures-pratiques

Chapitre 12: Fonctions définies par l'utilisateur (UDF)

Syntaxe

1. **Fonction functionName (argumentVariable As dataType, argumentVariable2 As dataType, Argument optionnelVariable3 As dataType) As functionReturnDataType**
   Déclaration de base d'une fonction. Chaque fonction a besoin d'un nom, mais il n'est pas nécessaire qu'elle prenne des arguments. Il peut prendre 0 argument ou prendre un nombre donné d'arguments. Vous pouvez également déclarer un argument comme facultatif (ce qui signifie que peu importe si vous le fournissez lors de l'appel de la fonction). Il est recommandé de fournir le type de variable pour chaque argument et, de même, de renvoyer le type de données que la fonction elle-même va retourner.

2. **functionName = theVariableOrValueBeingReturned**
   Si vous venez d'autres langages de programmation, vous pouvez être habitué au mot-clé `Return` . Ceci n'est pas utilisé dans VBA - au lieu de cela, nous utilisons le nom de la fonction. Vous pouvez le définir sur le contenu d'une variable ou sur une valeur directement fournie. Notez que si vous avez défini un type de données pour le retour de la fonction, la variable ou les données que vous fournissez cette fois doivent être de ce type de données.

3. **Fonction de fin**
   Obligatoire. Signifie la fin du bloc de code `Function` et doit donc être à la fin. Le VBE le fournit généralement automatiquement lorsque vous créez une nouvelle fonction.

Remarques

Une fonction définie par l'utilisateur (ou UDF) fait référence à une fonction spécifique à une tâche créée par l'utilisateur. Il peut être appelé comme une fonction de feuille de calcul (ex: `=SUM(...)`) ou utilisé pour renvoyer une valeur à un processus en cours d'exécution dans une procédure Sub. Un fichier UDF renvoie une valeur, généralement à partir d'informations transmises sous la forme d'un ou de plusieurs paramètres.

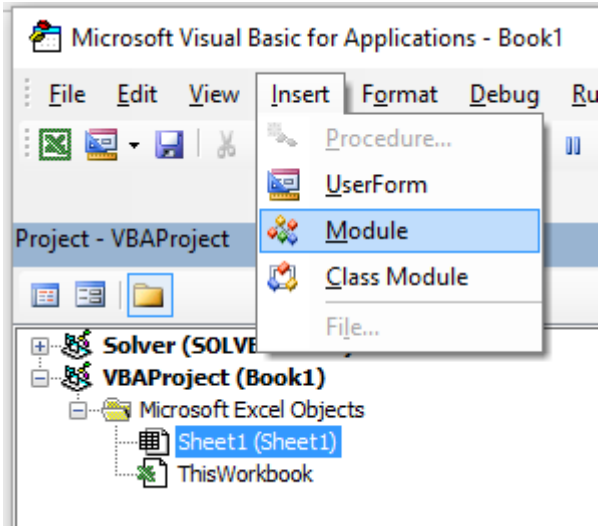
Il peut être créé par:

1. en utilisant VBA.
2. en utilisant l'API Excel C - En créant un fichier XLL qui exporte les fonctions compilées vers Excel.
3. en utilisant l'interface COM.

Examples

UDF - Hello World

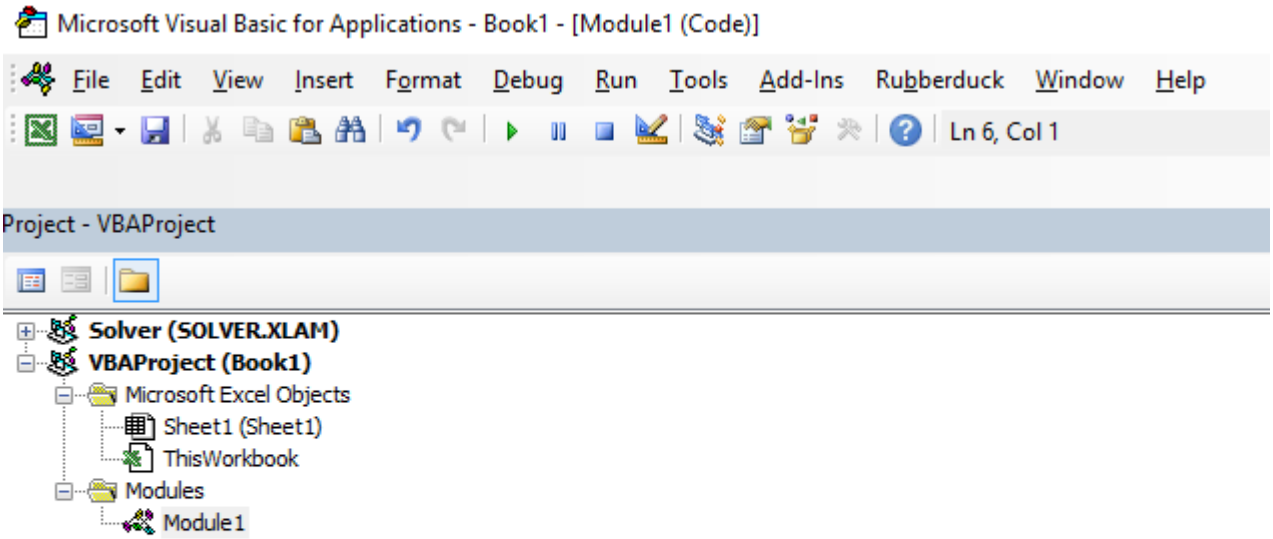
1. Ouvrir Excel
2. Ouvrez Visual Basic Editor (voir Ouverture de Visual Basic Editor)
3. Ajoutez un nouveau module en cliquant sur Insérer -> Module:

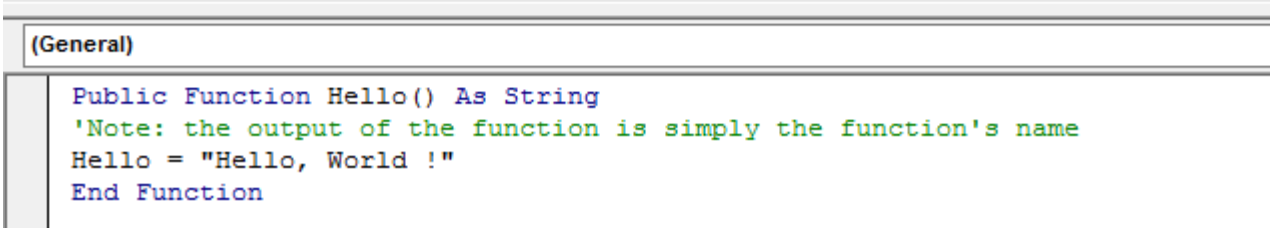


4. Copiez et collez le code suivant dans le nouveau module:

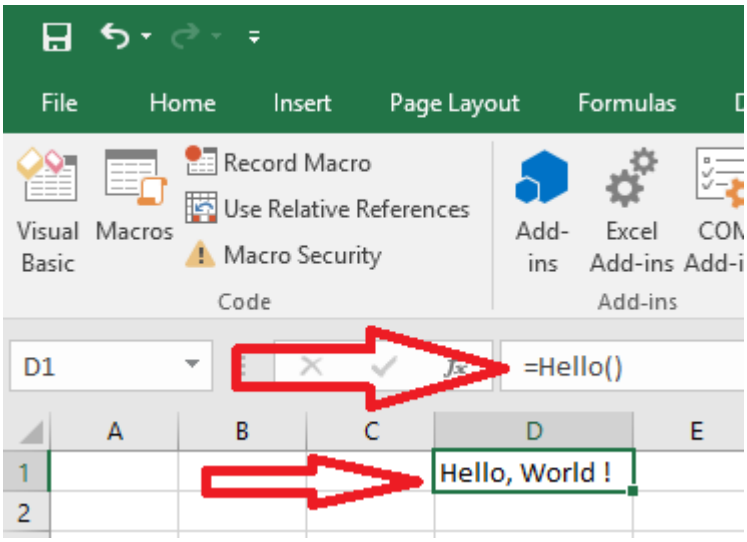
```
Public Function Hello() As String
'Note: the output of the function is simply the function's name
Hello = "Hello, World !"
End Function
```

Obtenir :





5. Retournez dans votre classeur et tapez "= Hello ()" dans une cellule pour voir le "Hello World".



Autoriser les références de colonne complètes sans pénalité

Il est plus facile d'implémenter des UDF sur la feuille de calcul si des références de colonne complètes peuvent être transmises en tant que paramètres. Cependant, en raison de la nature explicite du codage, toute boucle impliquant ces plages peut traiter des centaines de milliers de cellules complètement vides. Cela réduit votre projet VBA (et votre classeur) à un désordre gelé pendant que les non-valeurs inutiles sont traitées.

Traverser les cellules d'une feuille de calcul est l'une des méthodes les plus lentes pour accomplir une tâche, mais elle est parfois inévitable. Découper le travail effectué jusqu'à ce qui est réellement requis est parfaitement logique.

La solution consiste à tronquer les références complètes de colonne ou de ligne complète à la propriété Worksheet.UsedRange avec la méthode Intersect . L'exemple suivant va répliquer de manière lâche fonction SOMME.SI native d'une feuille de calcul afin que le *criteria_range* sera également redimensionnée en fonction de la *somme_plage* puisque chaque valeur dans le *somme_plage* doit être accompagnée d'une valeur dans la *criteria_range.*

Application.Caller pour une fonction définie par l'utilisateur utilisée sur une feuille de calcul est la cellule dans laquelle elle réside. La propriété .Parent de la cellule est la feuille de calcul. Ceci sera utilisé pour définir le .UsedRange.

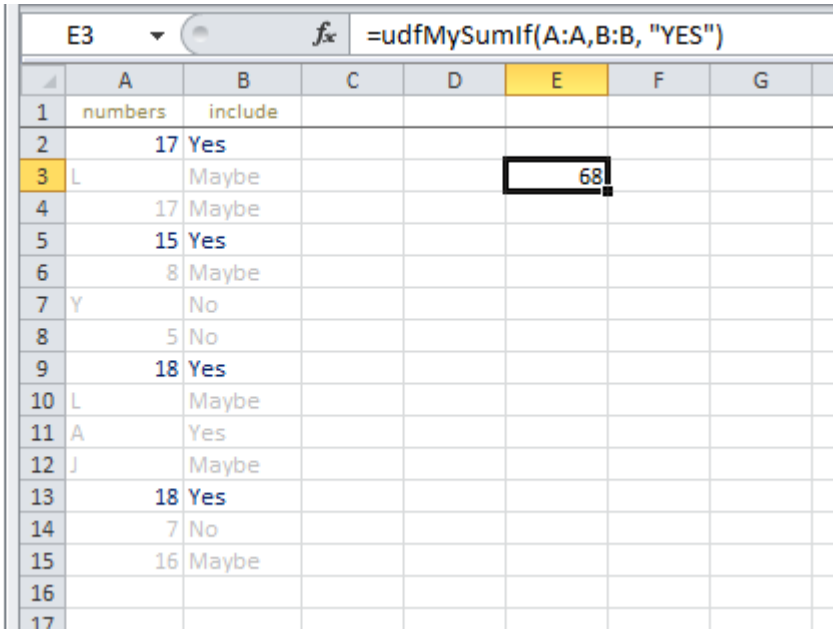
Dans une feuille de code du module:

```
Option Explicit

Function udfMySumIf(rngA As Range, rngB As Range, _
                    Optional crit As Variant = "yes")
    Dim c As Long, ttl As Double

    With Application.Caller.Parent
        Set rngA = Intersect(rngA, .UsedRange)
        Set rngB = rngB.Resize(rngA.Rows.Count, rngA.Columns.Count)
    End With

    For c = 1 To rngA.Cells.Count
        If IsNumeric(rngA.Cells(c).Value2) Then
```

```
            If LCase(rngB(c).Value2) = LCase(crit) Then
                ttl = ttl + rngA.Cells(c).Value2
            End If
        End If
    Next c

    udfMySumIf = ttl

End Function
```

Syntaxe:

```
=udfMySumIf(*sum_range*, *criteria_range*, [*criteria*])
```



	A	B	C	D	E	F	G
1	numbers	include					
2	17	Yes					
3	L	Maybe			68		
4	17	Maybe					
5	15	Yes					
6	8	Maybe					
7	Y	No					
8	5	No					
9	18	Yes					
10	L	Maybe					
11	A	Yes					
12	J	Maybe					
13	18	Yes					
14	7	No					
15	16	Maybe					
16							

Bien qu'il s'agisse d'un exemple assez simpliste, il montre de manière adéquate le passage de deux références de colonne complètes (1 048 576 lignes chacune), mais uniquement le traitement de 15 lignes de données et de critères.

Documentation MSDN officielle liée aux méthodes et propriétés individuelles de Microsoft ™.

Compter les valeurs uniques dans Range

```
Function countUnique(r As range) As Long
    'Application.Volatile False ' optional
    Set r = Intersect(r, r.Worksheet.UsedRange) ' optional if you pass entire rows or columns
to the function
    Dim c As New Collection, v
    On Error Resume Next   ' to ignore the Run-time error 457: "This key is already associated
with an element of this collection".
    For Each v In r.Value  ' remove .Value for ranges with more than one Areas
        c.Add 0, v & ""
    Next
    c.Remove "" ' optional to exclude blank values from the count
    countUnique = c.Count
End Function
```

Collections

Lire Fonctions définies par l'utilisateur (UDF) en ligne: https://riptutorial.com/fr/excel-vba/topic/1070/fonctions-definies-par-l-utilisateur--udf-

Chapitre 13: Gammes et cellules

Syntaxe

- **Set** - Opérateur utilisé pour définir une référence à un objet, tel qu'une plage
- **For Each** - L'opérateur a l'habitude de parcourir tous les éléments d'une collection

Remarques

Notez que les noms de variables `r` , `cell` et others peuvent être nommés comme vous le souhaitez mais doivent être nommés de manière à ce que le code soit plus facile à comprendre pour vous et pour les autres.

Examples

Créer une plage

Une plage ne peut pas être créée ou remplie de la même manière qu'une chaîne:

```
Sub RangeTest()
    Dim s As String
    Dim r As Range 'Specific Type of Object, with members like Address, WrapText, AutoFill,
etc.

    ' This is how we fill a String:
    s = "Hello World!"

    ' But we cannot do this for a Range:
    r = Range("A1") '//Run. Err.:  91 Object variable or With block variable not set//

    ' We have to use the Object approach, using keyword Set:
    Set r = Range("A1")
End Sub
```

La qualification de vos références est considérée comme la meilleure pratique. Nous utiliserons donc désormais la même approche.
En savoir plus sur la création de variables d'objet (par exemple, plage) sur MSDN . En savoir plus sur Set Statement sur MSDN .

Il existe différentes manières de créer la même gamme:

```
Sub SetRangeVariable()
    Dim ws As Worksheet
    Dim r As Range

    Set ws = ThisWorkbook.Worksheets(1) ' The first Worksheet in Workbook with this code in it

    ' These are all equivalent:
    Set r = ws.Range("A2")
```

```
    Set r = ws.Range("A" & 2)
    Set r = ws.Cells(2, 1) ' The cell in row number 2, column number 1
    Set r = ws.[A2] 'Shorthand notation of Range.
    Set r = Range("NamedRangeInA2") 'If the cell A2 is named NamedRangeInA2. Note, that this
is Sheet independent.
    Set r = ws.Range("A1").Offset(1, 0) ' The cell that is 1 row and 0 columns away from A1
    Set r = ws.Range("A1").Cells(2,1) ' Similar to Offset. You can "go outside" the original
Range.

    Set r = ws.Range("A1:A5").Cells(2) 'Second cell in bigger Range.
    Set r = ws.Range("A1:A5").Item(2) 'Second cell in bigger Range.
    Set r = ws.Range("A1:A5")(2) 'Second cell in bigger Range.
End Sub
```

Notez dans l'exemple que Cells (2, 1) est équivalent à Range ("A2"). Cela est dû au fait que Cells renvoie un objet Range.
Quelques sources: Chip Pearson-Cells Within Ranges ; MSDN-Range Object ; John Walkenback-Se référant aux gammes dans votre code VBA .

Notez également que dans toute instance où un nombre est utilisé dans la déclaration de la plage et que le nombre lui-même est en dehors des guillemets, tels que Range ("A" & 2), vous pouvez échanger ce nombre contre une variable contenant un entier / long. Par exemple:

```
Sub RangeIteration()
    Dim wb As Workbook, ws As Worksheet
    Dim r As Range

    Set wb = ThisWorkbook
    Set ws = wb.Worksheets(1)

    For i = 1 To 10
        Set r = ws.Range("A" & i)
        ' When i = 1, the result will be Range("A1")
        ' When i = 2, the result will be Range("A2")
        ' etc.
        ' Proof:
        Debug.Print r.Address
    Next i
End Sub
```

Si vous utilisez des doubles boucles, Cells est mieux:

```
Sub RangeIteration2()
    Dim wb As Workbook, ws As Worksheet
    Dim r As Range

    Set wb = ThisWorkbook
    Set ws = wb.Worksheets(1)

    For i = 1 To 10
        For j = 1 To 10
            Set r = ws.Cells(i, j)
            ' When i = 1 and j = 1, the result will be Range("A1")
            ' When i = 2 and j = 1, the result will be Range("A2")
            ' When i = 1 and j = 2, the result will be Range("B1")
            ' etc.
            ' Proof:
```

```
            Debug.Print r.Address
        Next j
    Next i
End Sub
```

Façons de se référer à une seule cellule

La manière la plus simple de faire référence à une seule cellule de la feuille de calcul Excel actuelle consiste simplement à inclure la forme A1 de sa référence entre crochets:

```
[a3] = "Hello!"
```

Notez que les crochets ne sont que du sucre syntaxique pratique pour la méthode `Evaluate` de l'objet `Application` , ce qui est techniquement identique au code suivant:

```
Application.Evaluate("a3") = "Hello!"
```

Vous pouvez également appeler la méthode `Cells` qui prend une ligne et une colonne et retourne une référence de cellule.

```
Cells(3, 1).Formula = "=A1+A2"
```

Rappelez-vous que chaque fois que vous passez une ligne et une colonne à Excel à partir de VBA, la ligne est toujours la première, suivie de la colonne, ce qui est déroutant car elle est opposée à la notation `A1` commune.

Dans ces deux exemples, nous n'avons pas spécifié de feuille de calcul. Par conséquent, Excel utilisera la feuille active (la feuille qui se trouve dans l'interface utilisateur). Vous pouvez spécifier explicitement la feuille active:

```
ActiveSheet.Cells(3, 1).Formula = "=SUM(A1:A2)"
```

Ou vous pouvez fournir le nom d'une feuille particulière:

```
Sheets("Sheet2").Cells(3, 1).Formula = "=SUM(A1:A2)"
```

Il existe une grande variété de méthodes qui peuvent être utilisées pour aller d'une gamme à une autre. Par exemple, la méthode `Rows` peut être utilisée pour accéder aux lignes individuelles de n'importe quelle plage et la méthode `Cells` peut être utilisée pour accéder aux cellules individuelles d'une ligne ou d'une colonne. Le code suivant fait référence à la cellule C1:

```
ActiveSheet.Rows(1).Cells(3).Formula = "hi!"
```

Enregistrement d'une référence à une cellule dans une variable

Pour enregistrer une référence à une cellule dans une variable, vous devez utiliser la syntaxe `Set` , par exemple:

```
Dim R as Range
Set R = ActiveSheet.Cells(3, 1)
```

plus tard...

```
R.Font.Color = RGB(255, 0, 0)
```

Pourquoi le mot-clé `Set` -il requis? `Set` indique à Visual Basic que la valeur du côté droit de `=` est censée être un objet.

Propriété décalée

- **Décalage (lignes, colonnes)** - L'opérateur a utilisé pour référencer statiquement un autre point de la cellule en cours. Souvent utilisé dans les boucles. Il faut comprendre que les nombres positifs dans la section des lignes se déplacent à droite, à mesure que les négatifs se déplacent vers la gauche. Avec la section des colonnes, les points positifs se déplacent vers le bas et les négatifs montent.

c'est à dire

```
Private Sub this()
    ThisWorkbook.Sheets("Sheet1").Range("A1").Offset(1, 1).Select
    ThisWorkbook.Sheets("Sheet1").Range("A1").Offset(1, 1).Value = "New Value"
    ActiveCell.Offset(-1, -1).Value = ActiveCell.Value
    ActiveCell.Value = vbNullString
End Sub
```

Ce code sélectionne B2, y place une nouvelle chaîne, puis déplace cette chaîne vers A1 après avoir effacé B2.

Comment transposer des plages (horizontales à verticales et vice versa)

```
Sub TransposeRangeValues()
    Dim TmpArray() As Variant, FromRange as Range, ToRange as Range

    set FromRange = Sheets("Sheet1").Range("a1:a12")         'Worksheets(1).Range("a1:p1")
    set ToRange = ThisWorkbook.Sheets("Sheet1").Range("a1")
'ThisWorkbook.Sheets("Sheet1").Range("a1")

    TmpArray = Application.Transpose(FromRange.Value)
    FromRange.Clear
    ToRange.Resize(FromRange.Columns.Count,FromRange.Rows.Count).Value2 = TmpArray
End Sub
```

Remarque: Copy / PasteSpecial a également une option Paste Transpose qui met également à jour les formules des cellules transposées.

Lire Gammes et cellules en ligne: https://riptutorial.com/fr/excel-vba/topic/1503/gammes-et-cellules

Chapitre 14: Gammes Nommées

Introduction

Topic doit inclure des informations spécifiquement liées aux plages nommées dans Excel, y compris des méthodes pour créer, modifier, supprimer et accéder aux plages nommées définies.

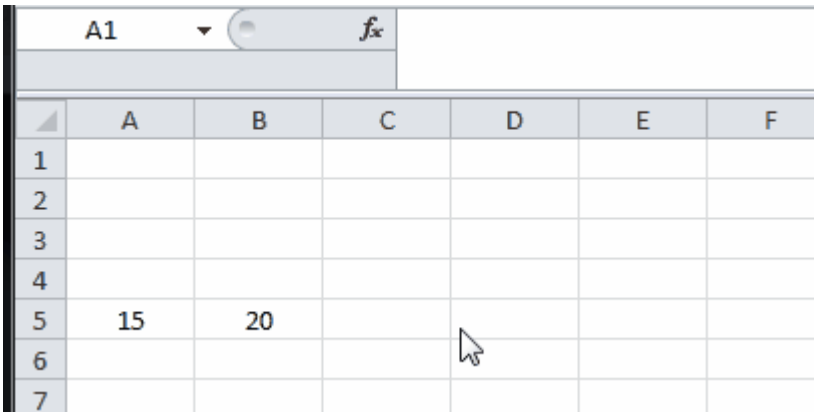
Examples

Définir une plage nommée

L'utilisation de plages nommées vous permet de décrire la signification d'un contenu de cellule (s) et d'utiliser ce nom défini à la place d'une adresse de cellule réelle.

Par exemple, la formule `=A5*B5` peut être remplacée par `=Width*Height` pour faciliter la lecture et la compréhension de la formule.

Pour définir une nouvelle plage nommée, sélectionnez la ou les cellules à nommer, puis tapez un nouveau nom dans la zone Nom en regard de la barre de formule.



> Remarque: les plages nommées sont définies par défaut sur une portée globale, ce qui signifie qu'elles sont accessibles depuis n'importe où dans le classeur. Les anciennes versions d'Excel prennent en charge les noms en double, il faut donc veiller à éviter les noms en double de portée mondiale, sinon les résultats seront imprévisibles. Utilisez le gestionnaire de noms de l'onglet Formules pour modifier l'étendue.

Utilisation de plages nommées dans VBA

Créer une nouvelle plage nommée appelée 'MyRange' affectée à la cellule `A1`

```
ThisWorkbook.Names.Add Name:="MyRange", _
    RefersTo:=Worksheets("Sheet1").Range("A1")
```

Supprimer la plage nommée nommée par nom

```
ThisWorkbook.Names("MyRange").Delete
```

Accès à la plage nommée par nom

```
Dim rng As Range
Set rng = ThisWorkbook.Worksheets("Sheet1").Range("MyRange")
Call MsgBox("Width = " & rng.Value)
```

Accéder à une plage nommée avec un raccourci

Comme pour toute autre plage , vous pouvez accéder directement aux plages nommées à l'aide d'une notation de raccourci ne nécessitant pas la création d'un objet `Range` . Les trois lignes de l'extrait de code ci-dessus peuvent être remplacées par une seule ligne:

```
Call MsgBox("Width = " & [MyRange])
```

> Remarque: La propriété par défaut pour une plage est sa valeur, donc `[MyRange]` est identique à `[MyRange].Value`

Vous pouvez également appeler des méthodes sur la plage. Ce qui suit sélectionne `MyRange` :

```
[MyRange].Select
```

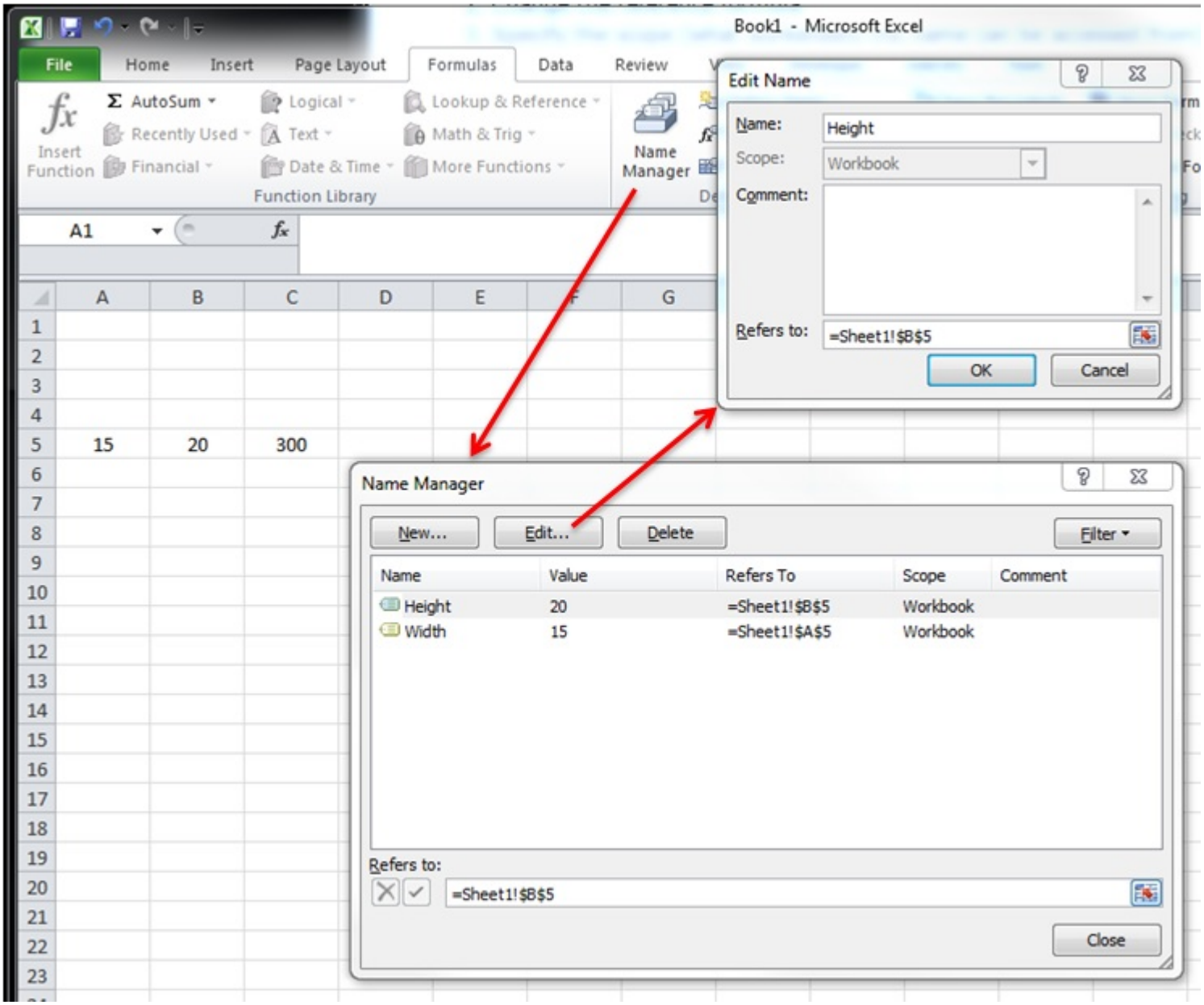
> Remarque: une mise en garde est que la notation de raccourci ne fonctionne pas avec les mots utilisés ailleurs dans la bibliothèque VBA. Par exemple, une plage nommée `Width` ne serait pas accessible en tant que `[Width]` mais fonctionnerait comme prévu si elle était accessible via `ThisWorkbook.Worksheets("Sheet1").Range("Width")`

Gérer les plages nommées à l'aide du gestionnaire de noms

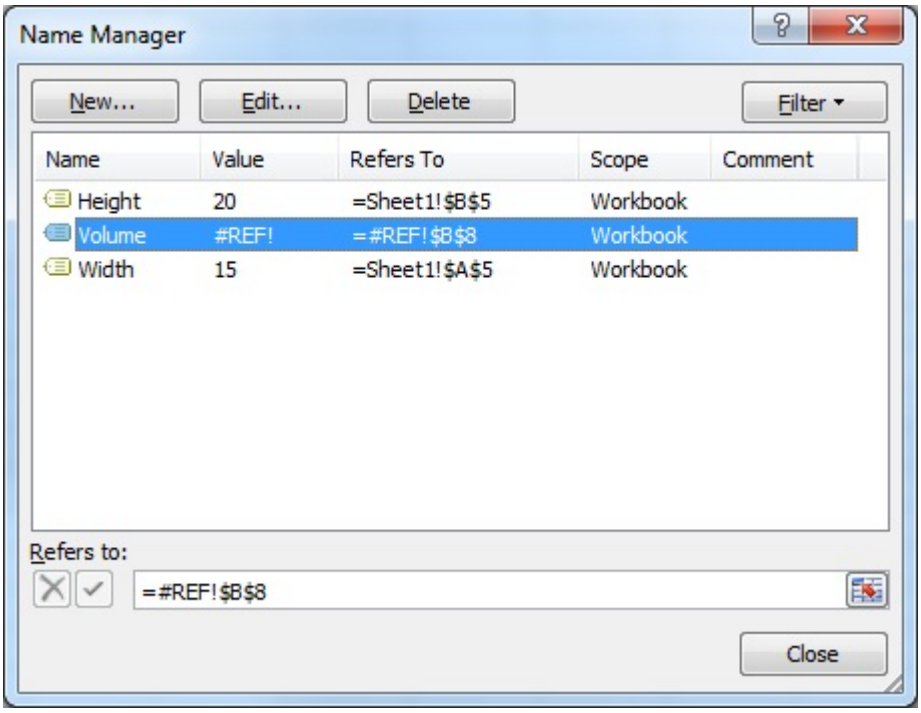
Onglet Formules> Groupe Noms définis> Bouton Gestionnaire de noms

Le gestionnaire nommé vous permet de:

1. Créer ou changer le nom
2. Créer ou modifier la référence de cellule
3. Créer ou modifier l'étendue
4. Supprimer une plage nommée existante

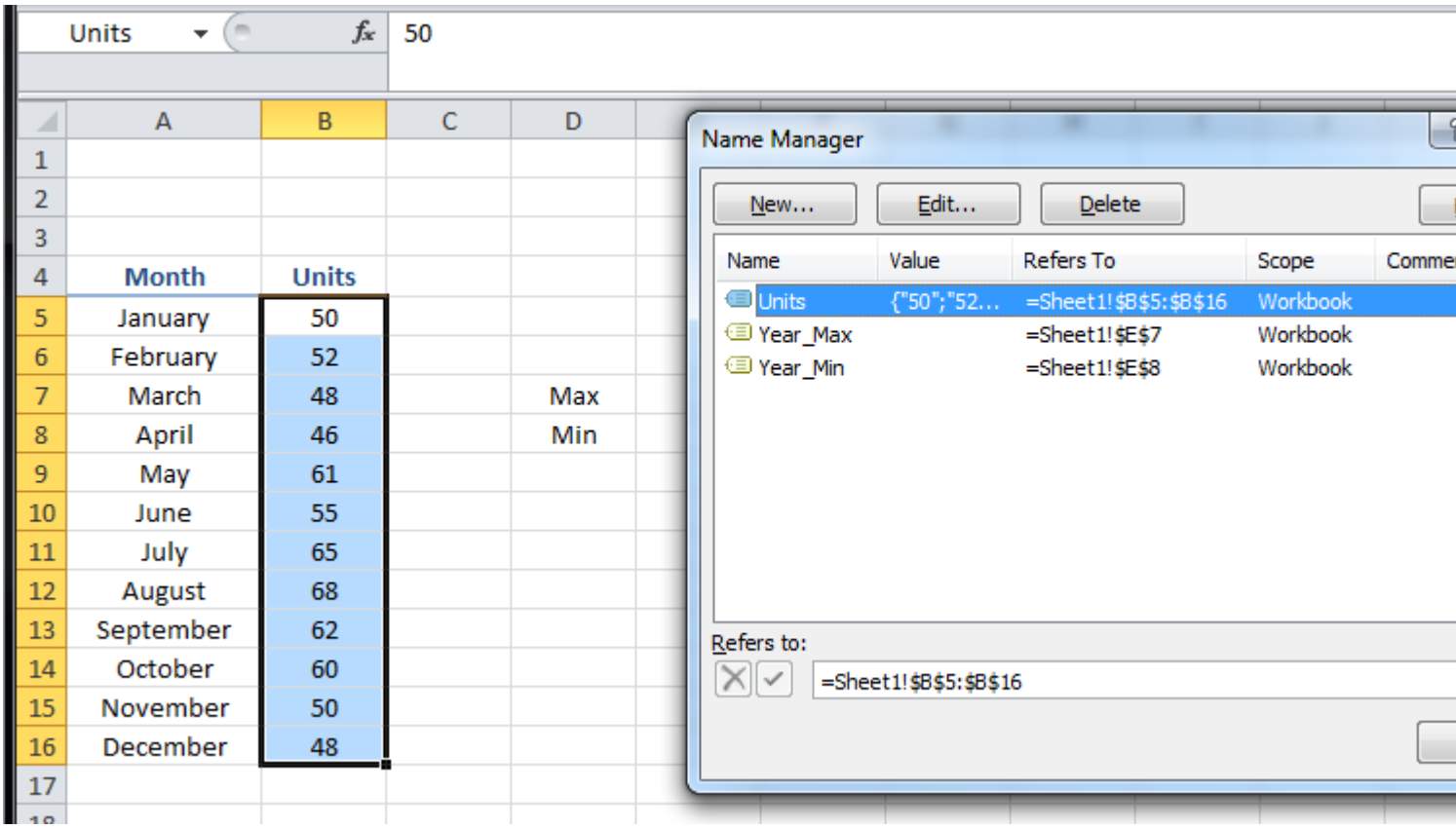


Named Manager fournit un aperçu rapide des liens rompus.



Tableaux de plage nommés

Exemple de fiche



Code

```
Sub Example()
    Dim wks As Worksheet
```

```
    Set wks = ThisWorkbook.Worksheets("Sheet1")

    Dim units As Range
    Set units = ThisWorkbook.Names("Units").RefersToRange

    Worksheets("Sheet1").Range("Year_Max").Value = WorksheetFunction.Max(units)
    Worksheets("Sheet1").Range("Year_Min").Value = WorksheetFunction.Min(units)
End Sub
```

Résultat

Month	Units			
January	50			
February	52			
March	48		Max	68
April	46		Min	46
May	61			
June	55			
July	65			
August	68			
September	62			
October	60			
November	50			
December	48			

Lire Gammes Nommées en ligne: https://riptutorial.com/fr/excel-vba/topic/8360/gammes-nommees

Chapitre 15: Graphiques et graphiques

Examples

Créer un graphique avec des plages et un nom fixe

Vous pouvez créer des graphiques en travaillant directement avec l'objet `Series` qui définit les données du graphique. Pour accéder à la `Series` sans graphique existant, vous créez un objet `ChartObject` sur une `Worksheet` donnée, puis vous obtenez l'objet `Chart` . L'avantage de travailler avec la `Series` objet est que vous pouvez définir les `Values` et `XValues` en se référant à `Range` objets. Ces propriétés de données définiront correctement la `Series` avec des références à ces plages. L'inconvénient de cette approche est que la même conversion n'est pas gérée lors de la définition du `Name` ; c'est une valeur fixe. Il ne s'ajustera pas avec les données sous-jacentes dans la `Range` origine. En cochant la formule `SERIES` , il est évident que le nom est fixe. Cela doit être géré en créant directement la formule `SERIES` .

Code utilisé pour créer un graphique

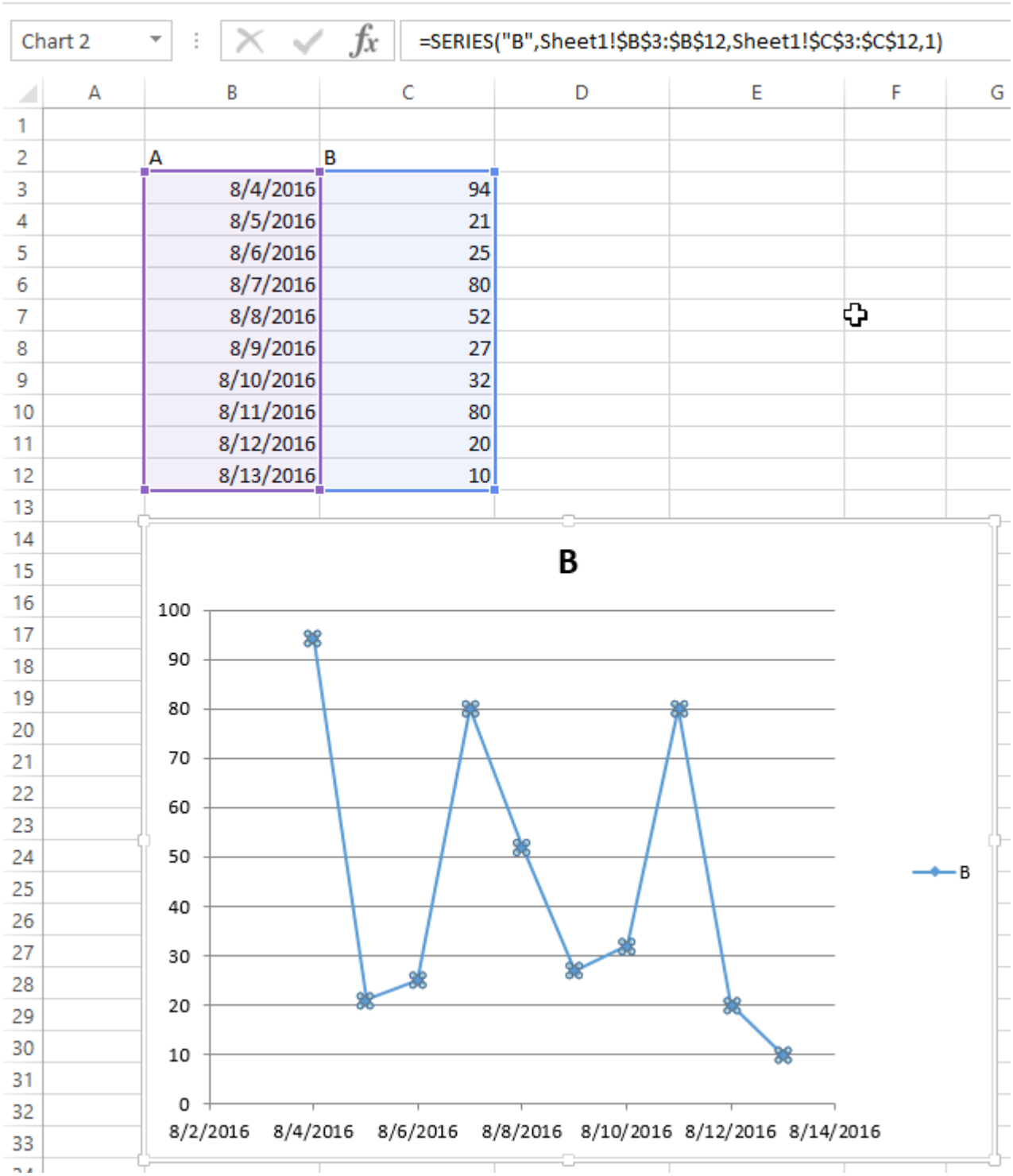
Notez que ce code contient des déclarations de variables supplémentaires pour le `Chart` et la `Worksheet` . Ceux-ci peuvent être omis s'ils ne sont pas utilisés. Ils peuvent cependant être utiles si vous modifiez le style ou d'autres propriétés du graphique.

```
Sub CreateChartWithRangesAndFixedName()

    Dim xData As Range
    Dim yData As Range
    Dim serName As Range

    'set the ranges to get the data and y value label
    Set xData = Range("B3:B12")
    Set yData = Range("C3:C12")
    Set serName = Range("C2")

    'get reference to ActiveSheet
    Dim sht As Worksheet
    Set sht = ActiveSheet

    'create a new ChartObject at position (48, 195) with width 400 and height 300
    Dim chtObj As ChartObject
    Set chtObj = sht.ChartObjects.Add(48, 195, 400, 300)

    'get reference to chart object
    Dim cht As Chart
    Set cht = chtObj.Chart

    'create the new series
    Dim ser As Series
    Set ser = cht.SeriesCollection.NewSeries

    ser.Values = yData
    ser.XValues = xData
    ser.Name = serName
```

```
    ser.ChartType = xlXYScatterLines

End Sub
```

Données / plages d'origine et `Chart` résultant après l'exécution du code

Notez que la formule `SERIES` inclut un `"B"` pour le nom de la série au lieu d'une référence à la `Range` qui l'a créé.



Créer un graphique vide

Le point de départ de la grande majorité des codes graphiques est de créer un `Chart` vide. Notez que ce `Chart` est soumis au modèle de graphique par défaut actif et peut ne pas être vide (si le modèle a été modifié).

La clé de `ChartObject` détermine son emplacement. La syntaxe de l'appel est `ChartObjects.Add(Left, Top, Width, Height)` . Une fois l'objet `ChartObject` créé, vous pouvez utiliser son objet `Chart` pour modifier le graphique. Le `ChartObject` se comporte plus comme une `Shape` pour positionner le graphique sur la feuille.

Code pour créer un graphique vide

```
Sub CreateEmptyChart()

    'get reference to ActiveSheet
    Dim sht As Worksheet
    Set sht = ActiveSheet

    'create a new ChartObject at position (0, 0) with width 400 and height 300
    Dim chtObj As ChartObject
    Set chtObj = sht.ChartObjects.Add(0, 0, 400, 300)

    'get refernce to chart object
    Dim cht As Chart
    Set cht = chtObj.Chart

    'additional code to modify the empty chart
    '...

End Sub
```

Graphique résultant

Créer un graphique en modifiant la formule SERIES

Pour un contrôle complet sur un nouvel objet `Chart` et `Series` (en particulier pour un nom de `Series` dynamique), vous devez modifier directement la formule `SERIES` . Le processus de configuration des objets `Range` est simple et le principal obstacle est simplement la création de chaînes pour la formule `SERIES` .

La formule `SERIES` prend la syntaxe suivante:

```
=SERIES(Name,XValues,Values,Order)
```

Ces contenus peuvent être fournis sous forme de références ou de valeurs de tableau pour les éléments de données. `Order` représente la position de la série dans le graphique. Notez que les références aux données ne fonctionneront que si elles sont pleinement qualifiées avec le nom de la feuille. Pour obtenir un exemple de formule de travail, cliquez sur une série existante et vérifiez la barre de formule.

Code pour créer un graphique et configurer les données à l'aide de la formule `SERIES`

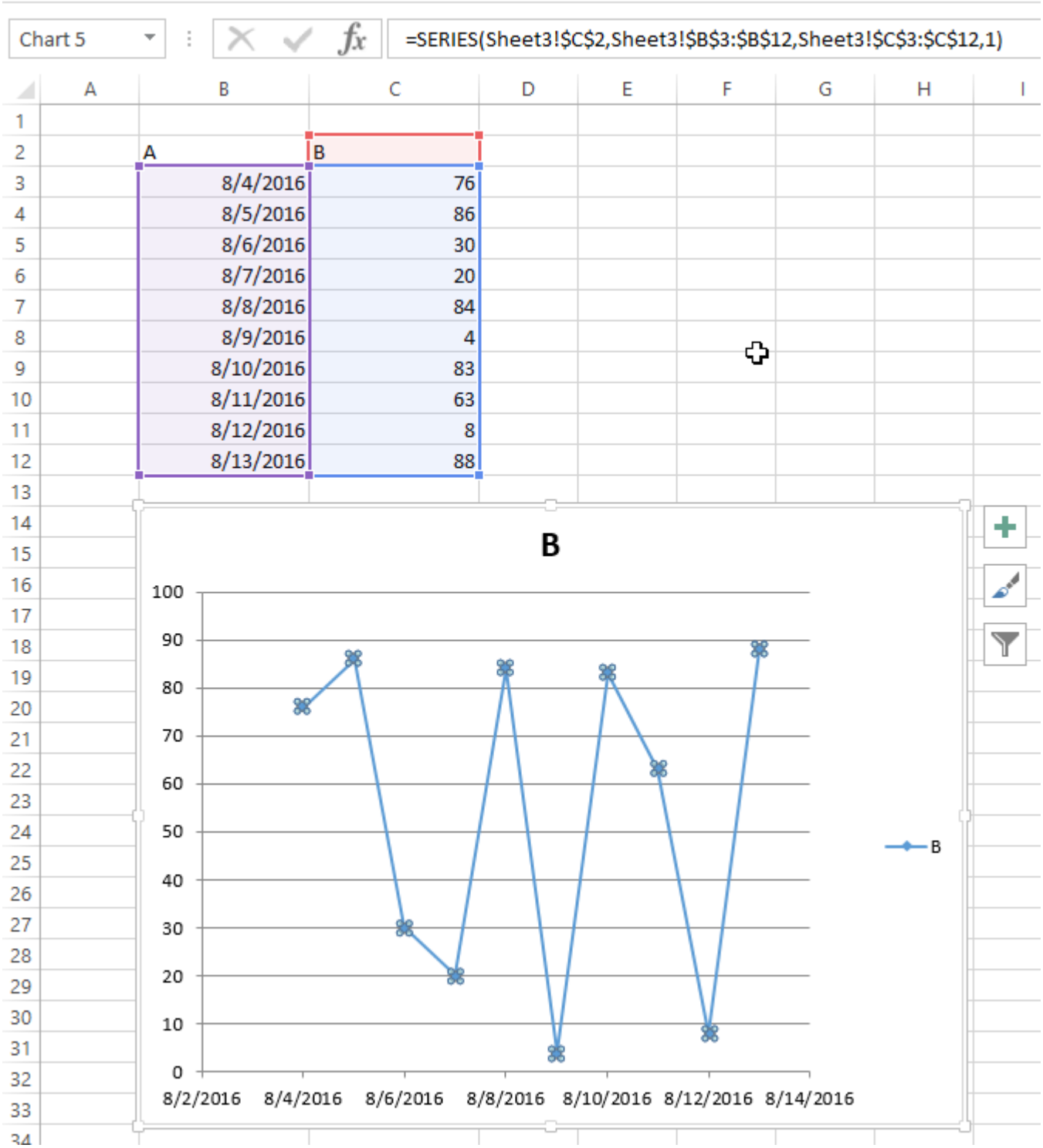
Notez que la construction de la chaîne pour créer la formule `SERIES` utilise `.Address(,,,True)` . Cela garantit que la référence de plage *externe* est utilisée pour inclure une adresse complète avec le nom de la feuille. Vous **obtiendrez une erreur si le nom de la feuille est exclu** .

```
Sub CreateChartUsingSeriesFormula()

    Dim xData As Range
```

```
    Dim yData As Range
    Dim serName As Range

    'set the ranges to get the data and y value label
    Set xData = Range("B3:B12")
    Set yData = Range("C3:C12")
    Set serName = Range("C2")

    'get reference to ActiveSheet
    Dim sht As Worksheet
    Set sht = ActiveSheet

    'create a new ChartObject at position (48, 195) with width 400 and height 300
    Dim chtObj As ChartObject
    Set chtObj = sht.ChartObjects.Add(48, 195, 400, 300)

    'get refernce to chart object
    Dim cht As Chart
    Set cht = chtObj.Chart

    'create the new series
    Dim ser As Series
    Set ser = cht.SeriesCollection.NewSeries

    'set the SERIES formula
    '=SERIES(name, xData, yData, plotOrder)

    Dim formulaValue As String
    formulaValue = "=SERIES(" & _
        serName.Address(, , , True) & "," & _
        xData.Address(, , , True) & "," & _
        yData.Address(, , , True) & ",1)"

    ser.Formula = formulaValue
    ser.ChartType = xlXYScatterLines

End Sub
```

Données d'origine et graphique résultant

Notez que pour ce graphique, le nom de la série est correctement défini avec une plage dans la cellule souhaitée. Cela signifie que les mises à jour se propageront au `Chart` .



Organiser des graphiques dans une grille

Une tâche courante avec les graphiques dans Excel est la standardisation de la taille et de la disposition de plusieurs graphiques sur une seule feuille. Si c'est fait manuellement, vous pouvez maintenir `ALT` tout en redimensionnant ou en déplaçant le graphique pour "coller" aux limites de la cellule. Cela fonctionne pour quelques graphiques, mais une approche VBA est beaucoup plus simple.

Code pour créer une grille

Ce code créera une grille de graphiques commençant à une position donnée (en haut, à gauche),

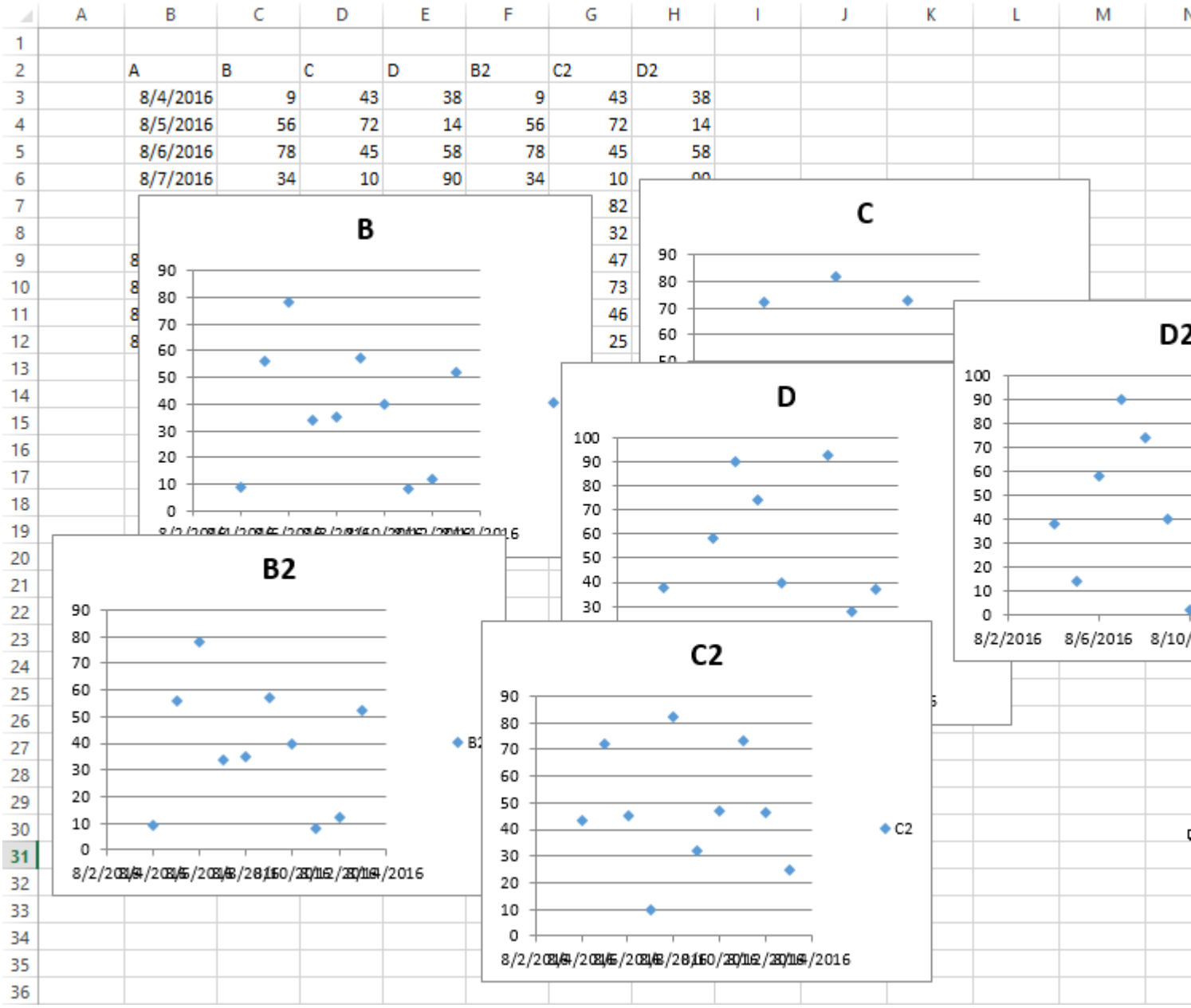
avec un nombre défini de colonnes et une taille de graphique commune définie. Les graphiques seront placés dans l'ordre dans lequel ils ont été créés et entoureront l'arête pour former une nouvelle ligne.

```
Sub CreateGridOfCharts()

    Dim int_cols As Integer
    int_cols = 3

    Dim cht_width As Double
    cht_width = 250

    Dim cht_height As Double
    cht_height = 200

    Dim offset_vertical As Double
    offset_vertical = 195

    Dim offset_horz As Double
    offset_horz = 40

    Dim sht As Worksheet
    Set sht = ActiveSheet

    Dim count As Integer
    count = 0

    'iterate through ChartObjects on current sheet
    Dim cht_obj As ChartObject
    For Each cht_obj In sht.ChartObjects

        'use integer division and Mod to get position in grid
        cht_obj.Top = (count \ int_cols) * cht_height + offset_vertical
        cht_obj.Left = (count Mod int_cols) * cht_width + offset_horz
        cht_obj.Width = cht_width
        cht_obj.Height = cht_height

        count = count + 1

    Next cht_obj
End Sub
```

Résultat avec plusieurs graphiques

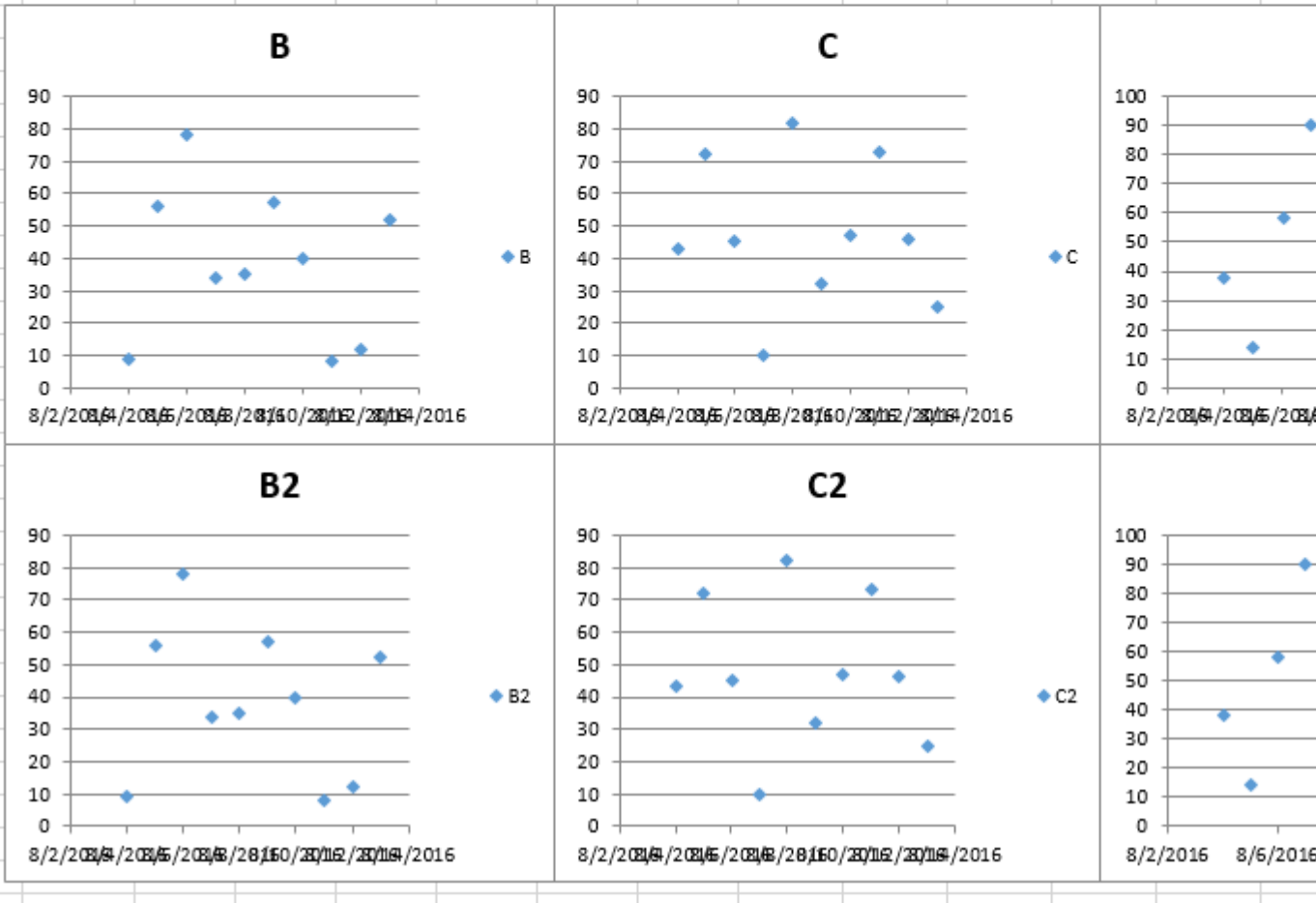
Ces images montrent la disposition aléatoire originale des graphiques et la grille résultante d'exécuter le code ci-dessus.

Avant


A B C D E F G H
A B C D B2 C2 D2
8/4/2016 9 43 38 9 43 38
8/5/2016 56 72 14 56 72 14
8/6/2016 78 45 58 78 45 58
8/7/2016 34 10 90 34 10
B
C
D
D2
B2
C2
8/2/2016
8/6/2016

Après

	A	B	C	D	E	F	G	H
1								
2		A	B	C	D	B2	C2	D2
3		8/4/2016	9	43	38	9	43	38
4		8/5/2016	56	72	14	56	72	14
5		8/6/2016	78	45	58	78	45	58
6		8/7/2016	34	10	90	34	10	90
7		8/8/2016	35	82	74	35	82	74
8		8/9/2016	57	32	40	57	32	40
9		8/10/2016	40	47	2	40	47	2
10		8/11/2016	8	73	93	8	73	93
11		8/12/2016	12	46	28	12	46	28
12		8/13/2016	52	25	37	52	25	37



Lire Graphiques et graphiques en ligne: https://riptutorial.com/fr/excel-vba/topic/4968/graphiques-et-graphiques

Chapitre 16: Intégration PowerPoint via VBA

Remarques

Cette section présente diverses manières d'interagir avec PowerPoint via VBA. De l'affichage des données sur les diapositives à la création de graphiques, PowerPoint est un outil très puissant lorsqu'il est utilisé avec Excel. Ainsi, cette section cherche à démontrer les différentes façons dont VBA peut être utilisé pour automatiser cette interaction.

Examples

Les bases: Lancer PowerPoint à partir de VBA

Bien que de nombreux paramètres puissent être modifiés et des variantes puissent être ajoutées en fonction de la fonctionnalité souhaitée, cet exemple présente la structure de base pour le lancement de PowerPoint.

> **Remarque:** Ce code nécessite que la référence PowerPoint ait été ajoutée au projet VBA actif. Voir l'entrée Documentation de référence pour savoir comment activer la référence.

Tout d'abord, définissez les variables pour les objets Application, Presentation et Slide. Bien que cela puisse être fait avec une liaison tardive, il est toujours préférable d'utiliser une liaison anticipée, le cas échéant.

```
Dim PPApp As PowerPoint.Application
Dim PPPres As PowerPoint.Presentation
Dim PPSlide As PowerPoint.Slide
```

Ensuite, ouvrez ou créez une nouvelle instance de l'application PowerPoint. Ici, l'appel de `On Error Resume Next` est utilisé pour éviter qu'une erreur ne soit `GetObject` par `GetObject` si PowerPoint n'a pas encore été ouvert. Voir l'exemple de gestion des erreurs de la rubrique Meilleures pratiques pour une explication plus détaillée.

```
'Open PPT if not running, otherwise select active instance
On Error Resume Next
Set PPApp = GetObject(, "PowerPoint.Application")
On Error GoTo ErrHandler
If PPApp Is Nothing Then
    'Open PowerPoint
    Set PPApp = CreateObject("PowerPoint.Application")
    PPApp.Visible = True
End If
```

Une fois l'application lancée, une nouvelle présentation et une diapositive contenue par la suite sont générées pour être utilisées.

```
'Generate new Presentation and slide for graphic creation
Set PPPres = PPApp.Presentations.Add
Set PPSlide = PPPres.Slides.Add(1, ppLayoutBlank)

'Here, the slide type is set to the 4:3 shape with slide numbers enabled and the window
'maximized on the screen.  These properties can, of course, be altered as needed

PPApp.ActiveWindow.ViewType = ppViewSlide
PPPres.PageSetup.SlideOrientation = msoOrientationHorizontal
PPPres.PageSetup.SlideSize = ppSlideSizeOnScreen
PPPres.SlideMaster.HeadersFooters.SlideNumber.Visible = msoTrue
PPApp.ActiveWindow.WindowState = ppWindowMaximized
```

À la fin de ce code, une nouvelle fenêtre PowerPoint avec une diapositive vierge sera ouverte. En utilisant les variables d'objet, des formes, du texte, des graphiques et des gammes Excel peuvent être ajoutés à volonté

Lire Intégration PowerPoint via VBA en ligne: https://riptutorial.com/fr/excel-vba/topic/2327/integration-powerpoint-via-vba

Chapitre 17: Localisation des valeurs en double dans une plage

Introduction

À certains moments, vous évaluerez une série de données et vous devrez y localiser les doublons. Pour les ensembles de données plus volumineux, il existe un certain nombre d'approches utilisant le code VBA ou les fonctions conditionnelles. Cet exemple utilise une simple condition if-then dans deux boucles imbriquées suivantes pour tester si chaque cellule de la plage est égale en valeur à une autre cellule de la plage.

Examples

Trouver des doublons dans une plage

Les tests suivants vont de A2 à A7 pour les valeurs en double. **Remarque:** Cet exemple illustre une solution possible en tant que première approche d'une solution. Il est plus rapide d'utiliser un tableau qu'une plage et on peut utiliser des collections, des dictionnaires ou des méthodes XML pour vérifier les doublons.

```
    Sub find_duplicates()
' Declare variables
  Dim ws     As Worksheet               ' worksheet
  Dim cell   As Range                   ' cell within worksheet range
  Dim n      As Integer                 ' highest row number
  Dim bFound As Boolean                 ' boolean flag, if duplicate is found
  Dim sFound As String: sFound = "|"    ' found duplicates
  Dim s      As String                  ' message string
  Dim s2     As String                  ' partial message string
' Set Sheet to memory
  Set ws = ThisWorkbook.Sheets("Duplicates")

' loop thru FULLY QUALIFIED REFERENCE
  For Each cell In ws.Range("A2:A7")
    bFound = False: s2 = ""             ' start each cell with empty values
 '  Check if first occurrence of this value as duplicate to avoid further searches
    If InStr(sFound, "|" & cell & "|") = 0 Then

      For n = cell.Row + 1 To 7           ' iterate starting point to avoid REDUNDANT SEARCH
        If cell = ws.Range("A" & n).Value Then
           If cell.Row <> n Then        ' only other cells, as same cell cannot be a duplicate
                 bFound = True             ' boolean flag
              '  found duplicates in cell A{n}
                 s2 = s2 & vbNewLine & " -> duplicate in A" & n
           End If
        End If
       Next
     End If
   ' notice all found duplicates
     If bFound Then
```

```
          ' add value to list of all found duplicate values
          ' (could be easily split to an array for further analyze)
            sFound = sFound & cell & "|"
            s = s & cell.Address & " (value=" & cell & ")" & s2 & vbNewLine & vbNewLine
      End If
    Next
' Messagebox with final result
  MsgBox "Duplicate values are " & sFound & vbNewLine & vbNewLine & s, vbInformation, "Found
duplicates"
End Sub
```

Selon vos besoins, l'exemple peut être modifié - par exemple, la limite supérieure de n peut être la valeur de ligne de la dernière cellule contenant des données dans la plage, ou l'action en cas de condition True Si peut être modifiée pour extraire la copie. valeur ailleurs. Cependant, la mécanique de la routine ne changerait pas.

Lire Localisation des valeurs en double dans une plage en ligne: https://riptutorial.com/fr/excel-vba/topic/8295/localisation-des-valeurs-en-double-dans-une-plage

Chapitre 18: Meilleures pratiques de VBA

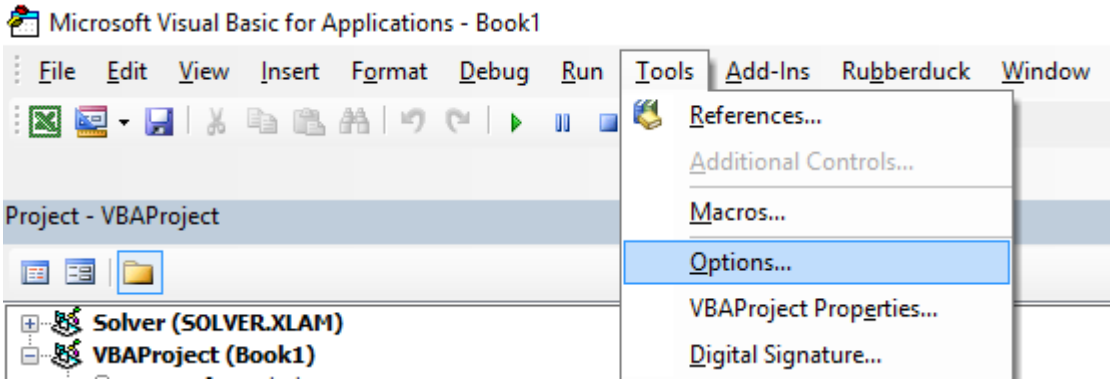
Remarques

Nous les connaissons tous, mais ces pratiques sont beaucoup moins évidentes pour quelqu'un qui commence à programmer en VBA.

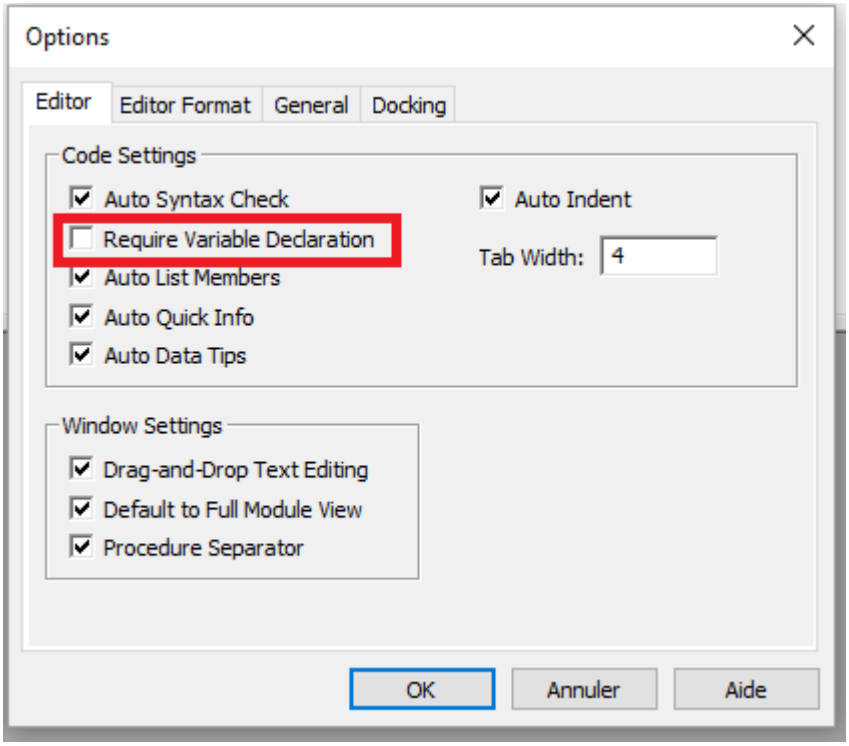
Examples

TOUJOURS utiliser "Option Explicit"

Dans la fenêtre Editeur VBA, dans le menu Outils, sélectionnez "Options":



Ensuite, dans l'onglet "Éditeur", assurez-vous que "Exiger une déclaration de variable" est coché:



La sélection de cette option place automatiquement `Option Explicit` en haut de chaque module

VBA.

> **Petite remarque:** Cela est vrai pour les modules, les modules de classe, etc. qui n'ont pas encore été ouverts. Donc, si vous avez déjà regardé par exemple le code de `Sheet1` avant d'activer l'option "Exiger une déclaration de variable", `Option Explicit` ne sera pas ajoutée!

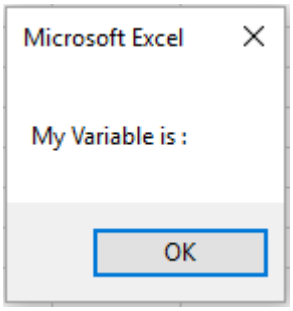
`Option Explicit` exige que chaque variable soit définie avant utilisation, par exemple avec une instruction `Dim` . Sans l' `Option Explicit` , le compilateur VBA supposera que tout mot non reconnu sera une nouvelle variable du type `Variant` , ce qui causera des bogues extrêmement difficiles à détecter liés aux erreurs typographiques. Lorsque l' `Option Explicit` activée, tous les mots non reconnus provoquent une erreur de compilation indiquant la ligne incriminée.

Exemple :

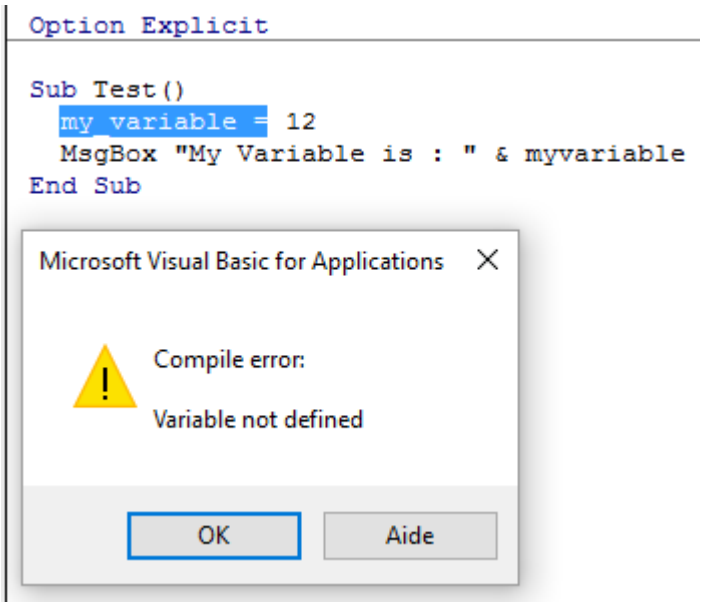
Si vous exécutez le code suivant:

```
Sub Test()
  my_variable = 12
  MsgBox "My Variable is : " & myvariable
End Sub
```

Vous recevrez le message suivant:



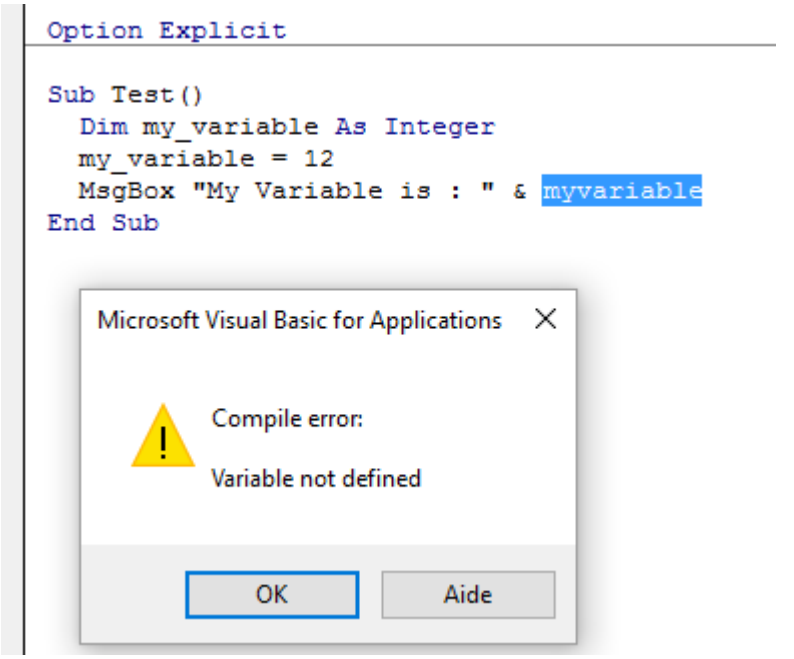
Vous avez fait une erreur en écrivant `myvariable` au lieu de `my_variable` , puis la boîte de message affiche une variable vide. Si vous utilisez `Option Explicit` , cette erreur n'est pas possible car vous obtiendrez un message d'erreur de compilation indiquant le problème.



Maintenant, si vous ajoutez la déclaration correcte:

```
Sub Test()
  Dim my_variable As Integer
  my_variable = 12
  MsgBox "My Variable is : " & myvariable
End Sub
```

Vous obtiendrez un message d'erreur indiquant précisément l'erreur avec `myvariable` :



Remarque sur les options explicites et tableaux (déclaration d'un tableau dynamique):

Vous pouvez utiliser l'instruction ReDim pour déclarer un tableau implicitement dans une procédure.

- Veillez à ne pas mal orthographier le nom du tableau lorsque vous utilisez l'instruction ReDim
- Même si l'instruction Option Explicit est incluse dans le module, un nouveau tableau sera créé

```
Dim arr() as Long

ReDim ar() 'creates new array "ar" - "ReDim ar()" acts like "Dim ar()"
```

Travailler avec des tableaux, pas avec des gammes

Blog Office - Meilleures pratiques pour le codage des performances VBA Excel

Souvent, les meilleures performances sont obtenues en évitant autant que possible l'utilisation de `Range` . Dans cet exemple, nous lisons un objet `Range` entier dans un tableau, mettons en carré chaque nombre du tableau, puis nous renvoyons le tableau à la `Range` . Cela n'accède à `Range` que deux fois, alors qu'une boucle y accède 20 fois pour les lectures / écritures.

```
Option Explicit
Sub WorkWithArrayExample()

Dim DataRange As Variant
Dim Irow As Long
Dim Icol As Integer
DataRange = ActiveSheet.Range("A1:A10").Value ' read all the values at once from the Excel
grid, put into an array

For Irow = LBound(DataRange,1) To UBound(DataRange, 1) ' Get the number of rows.
  For Icol = LBound(DataRange,2) To UBound(DataRange, 2) ' Get the number of columns.
    DataRange(Irow, Icol) = DataRange(Irow, Icol) * DataRange(Irow, Icol) ' cell.value^2
  Next Icol
Next Irow
ActiveSheet.Range("A1:A10").Value = DataRange ' writes all the results back to the range at
once

End Sub
```

Vous trouverez plus de conseils et d'informations avec des exemples temporels dans les FDU de Writing Williams VBA (Part 1) de Charles Williams et d' autres articles de la série .

Utiliser les constantes VB lorsqu'elles sont disponibles

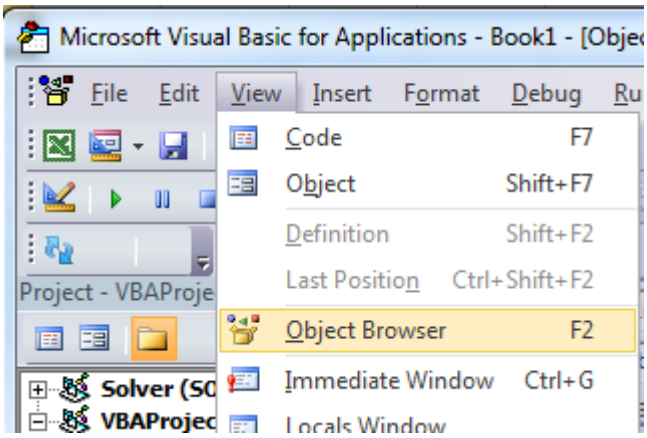
```
If MsgBox("Click OK") = vbOK Then
```

peut être utilisé à la place de

```
If MsgBox("Click OK") = 1 Then
```

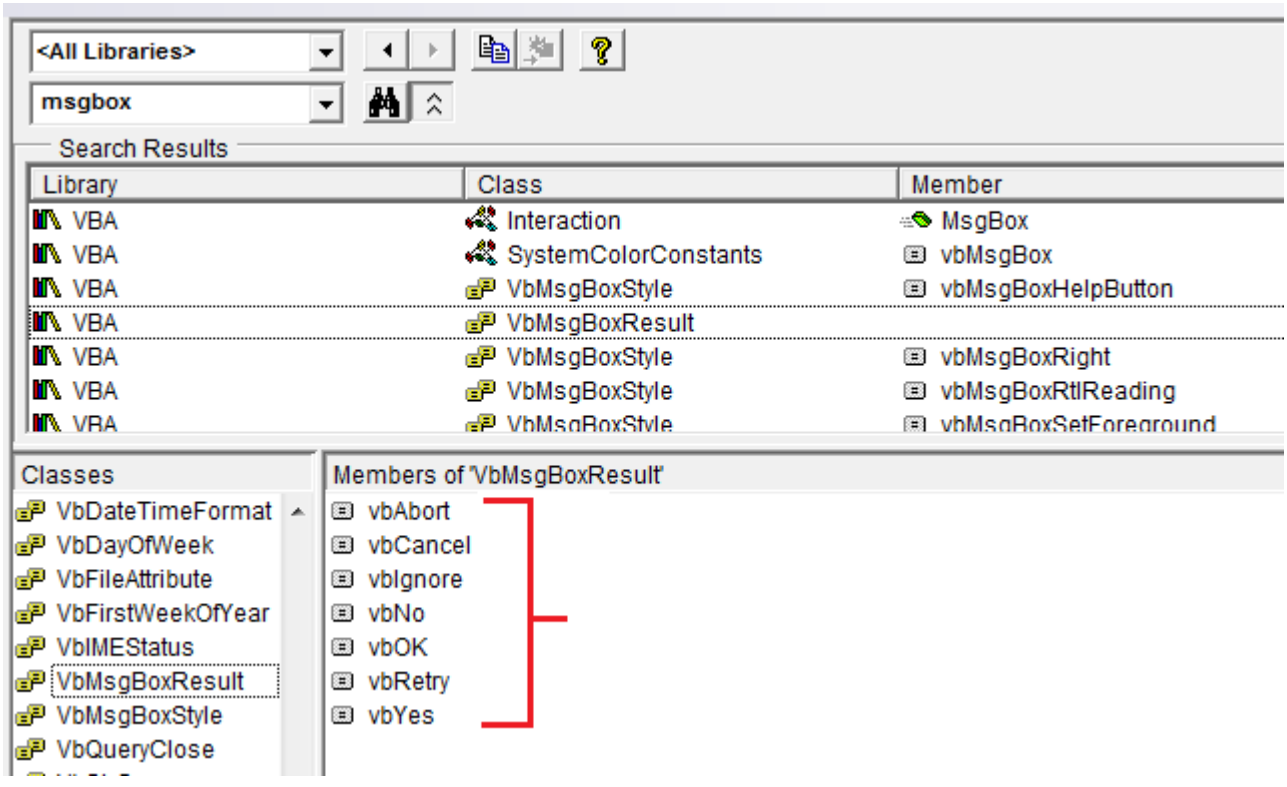
afin d'améliorer la lisibilité.

Utilisez le *navigateur d'objets* pour trouver les constantes VB disponibles. *Afficher* → *Navigateur d'objets* ou `F2` de l'éditeur VB.



Entrez le cours pour rechercher

Afficher les membres disponibles



Utiliser une dénomination de variable descriptive

Les noms descriptifs et la structure dans votre code aident à rendre les commentaires inutiles

```
Dim ductWidth  As Double
Dim ductHeight As Double
Dim ductArea   As Double
```

```
ductArea = ductWidth * ductHeight
```

est mieux que

```
Dim a, w, h

a = w * h
```

Cela est particulièrement utile lorsque vous copiez des données d'un endroit à un autre, qu'il s'agisse d'une cellule, d'une plage, d'une feuille de calcul ou d'un classeur. Aidez-vous en utilisant des noms tels que ceux-ci:

```
Dim myWB As Workbook
Dim srcWS As Worksheet
Dim destWS As Worksheet
Dim srcData As Range
Dim destData As Range

Set myWB = ActiveWorkbook
Set srcWS = myWB.Sheets("Sheet1")
Set destWS = myWB.Sheets("Sheet2")
Set srcData = srcWS.Range("A1:A10")
Set destData = destWS.Range("B11:B20")
destData = srcData
```

Si vous déclarez plusieurs variables dans une ligne, assurez-vous de spécifier un type pour *chaque* variable comme:

```
Dim ductWidth As Double, ductHeight As Double, ductArea As Double
```

Ce qui suit ne déclarera que la dernière variable et les premières resteront `Variant` :

```
Dim ductWidth, ductHeight, ductArea As Double
```

La gestion des erreurs

Une bonne gestion des erreurs empêche les utilisateurs finaux de voir les erreurs d'exécution de VBA et aide le développeur à diagnostiquer et à corriger facilement les erreurs.

Il existe trois méthodes principales de gestion des erreurs dans VBA, dont deux doivent être évitées pour les programmes distribués, sauf si cela est spécifiquement requis dans le code.

```
On Error GoTo 0 'Avoid using
```

ou

```
On Error Resume Next 'Avoid using
```

Préférez utiliser:

```
On Error GoTo <line> 'Prefer using
```

En cas d'erreur GoTo 0

Si aucune gestion des erreurs n'est définie dans votre code, `On Error GoTo 0` est le gestionnaire d'erreurs par défaut. Dans ce mode, toute erreur d'exécution lancera le message d'erreur VBA standard, vous permettant de mettre fin au code ou d'entrer en mode `debug` , en identifiant la source. Lors de l'écriture de code, cette méthode est la plus simple et la plus utile, mais elle devrait toujours être évitée pour le code distribué aux utilisateurs finaux, car cette méthode est très difficile à comprendre pour les utilisateurs finaux.

On Error Resume Next

`On Error Resume Next` fera en sorte que VBA ignore toutes les erreurs générées à l'exécution pour toutes les lignes après l'appel d'erreur jusqu'à ce que le gestionnaire d'erreurs ait été modifié. Dans des cas très spécifiques, cette ligne peut être utile, mais elle doit être évitée en dehors de ces cas. Par exemple, lors du lancement d'un programme distinct à partir d'une macro Excel, l'appel de `On Error Resume Next` peut être utile si vous ne savez pas si le programme est déjà ouvert ou non:

```
'In this example, we open an instance of Powerpoint using the On Error Resume Next call
Dim PPApp As PowerPoint.Application
Dim PPPres As PowerPoint.Presentation
Dim PPSlide As PowerPoint.Slide

'Open PPT if not running, otherwise select active instance
On Error Resume Next
Set PPApp = GetObject(, "PowerPoint.Application")
On Error GoTo ErrHandler
If PPApp Is Nothing Then
    'Open PowerPoint
    Set PPApp = CreateObject("PowerPoint.Application")
    PPApp.Visible = True
End If
```

Si nous n'avions pas utilisé l'appel de reprise `On Error Resume Next` et que l'application Powerpoint n'était pas déjà ouverte, la méthode `GetObject` émettrait une erreur. Ainsi, `On Error Resume Next` était nécessaire pour éviter de créer deux instances de l'application.

Remarque: Il est également recommandé de réinitialiser *immédiatement* le gestionnaire d'erreurs dès que vous n'avez plus besoin de l'appel de la fonction de `On Error Resume Next`

En cas d'erreur GoTo <line>

Cette méthode de gestion des erreurs est recommandée pour tout le code distribué aux autres utilisateurs. Cela permet au programmeur de contrôler exactement comment VBA traite une erreur en envoyant le code à la ligne spécifiée. La balise peut être remplie avec n'importe quelle chaîne (y compris les chaînes numériques) et enverra le code à la chaîne correspondante suivie de deux points. Plusieurs blocs de gestion des erreurs peuvent être utilisés en effectuant différents appels de `On Error GoTo <line>` . La sous-routine ci-dessous illustre la syntaxe d'un appel `On Error GoTo <line>` .

Remarque: Il est essentiel que la ligne de `Exit Sub` soit placée au-dessus du premier gestionnaire d'erreur et avant chaque gestionnaire d'erreur suivant pour empêcher le code de progresser naturellement dans le bloc *sans* qu'une erreur soit appelée. Par conséquent, il est recommandé de placer les gestionnaires d'erreurs à la fin d'un bloc de code dans les fonctions et la lisibilité.

```
Sub YourMethodName()
    On Error GoTo errorHandler
    ' Insert code here
    On Error GoTo secondErrorHandler

    Exit Sub 'The exit sub line is essential, as the code will otherwise
             'continue running into the error handling block, likely causing an error

errorHandler:
    MsgBox "Error " & Err.Number & ": " & Err.Description & " in " & _
        VBE.ActiveCodePane.CodeModule, vbOKOnly, "Error"
    Exit Sub

secondErrorHandler:
    If Err.Number = 424 Then 'Object not found error (purely for illustration)
        Application.ScreenUpdating = True
        Application.EnableEvents = True
        Exit Sub
    Else
        MsgBox "Error " & Err.Number & ": " & Err.Desctription
        Application.ScreenUpdating = True
        Application.EnableEvents = True
        Exit Sub
    End If
    Exit Sub

End Sub
```

Si vous quittez votre méthode avec votre code de gestion des erreurs, assurez-vous de nettoyer:

- Annule tout ce qui est partiellement terminé
- Fermer les fichiers
- Réinitialiser la mise à jour de l'écran
- Réinitialiser le mode de calcul
- Réinitialiser les événements
- Réinitialiser le pointeur de la souris
- Appelez la méthode de déchargement sur les instances d'objets qui persistent après la `End Sub`
- Réinitialiser la barre d'état

Documentez votre travail

Il est recommandé de documenter votre travail pour une utilisation ultérieure, en particulier si vous codez pour une charge de travail dynamique. Les bons commentaires devraient expliquer pourquoi le code fait quelque chose, pas ce que le code fait.

```
Function Bonus(EmployeeTitle as String) as Double
    If EmployeeTitle = "Sales" Then
        Bonus = 0    'Sales representatives receive commission instead of a bonus
    Else
        Bonus = .10
    End If
End Function
```

Si votre code est tellement obscur qu'il nécessite des commentaires pour expliquer ce qu'il fait, envisagez de le réécrire pour qu'il soit plus clair au lieu de l'expliquer par des commentaires. Par exemple, au lieu de:

```
Sub CopySalesNumbers
    Dim IncludeWeekends as Boolean

    'Boolean values can be evaluated as an integer, -1 for True, 0 for False.
    'This is used here to adjust the range from 5 to 7 rows if including weekends.
    Range("A1:A" & 5 - (IncludeWeekends * 2)).Copy
    Range("B1").PasteSpecial
End Sub
```

Clarifier le code pour être plus facile à suivre, tel que:

```
Sub CopySalesNumbers
    Dim IncludeWeekends as Boolean
    Dim DaysinWeek as Integer

    If IncludeWeekends Then
        DaysinWeek = 7
    Else
        DaysinWeek = 5
    End If
    Range("A1:A" & DaysinWeek).Copy
    Range("B1").PasteSpecial
End Sub
```

Désactiver les propriétés lors de l'exécution de la macro

Il est recommandé dans tout langage de programmation d' **éviter toute optimisation prématurée.** Cependant, si les tests révèlent que votre code s'exécute trop lentement, vous pouvez gagner en rapidité en désactivant certaines propriétés de l'application pendant son exécution. Ajoutez ce code à un module standard:

```
Public Sub SpeedUp( _
    SpeedUpOn As Boolean, _
    Optional xlCalc as XlCalculation = xlCalculationAutomatic _
)
```

```
    With Application
        If SpeedUpOn Then
            .ScreenUpdating = False
            .Calculation = xlCalculationManual
            .EnableEvents = False
            .DisplayStatusBar = False 'in case you are not showing any messages
            ActiveSheet.DisplayPageBreaks = False 'note this is a sheet-level setting
        Else
            .ScreenUpdating = True
            .Calculation = xlCalc
            .EnableEvents = True
            .DisplayStatusBar = True
            ActiveSheet.DisplayPageBreaks = True
        End If
    End With
End Sub
```

Plus d'informations sur le blog Office - Meilleures pratiques pour le codage des performances VBA Excel

Et juste l'appeler au début et à la fin des macros:

```
Public Sub SomeMacro
    'store the initial "calculation" state
    Dim xlCalc As XlCalculation
    xlCalc = Application.Calculation

    SpeedUp True

    'code here ...

    'by giving the second argument the initial "calculation" state is restored
    'otherwise it is set to 'xlCalculationAutomatic'
    SpeedUp False, xlCalc
End Sub
```

Bien que ceux - ci peuvent en grande partie être considérés comme des « améliorations » pour régulières `Public Sub` procédures, événement invalidantes manipulation avec `Application.EnableEvents = False` devrait être considérée comme obligatoire pour `Worksheet_Change` et `Workbook_SheetChange` macros d'événement privé qui modifient les valeurs sur une ou plusieurs feuilles de calcul. Si vous ne désactivez pas les déclencheurs d'événement, la macro d'événement s'exécute de manière récursive sur elle-même lorsqu'une valeur change et peut conduire à un classeur "gelé". N'oubliez pas de réactiver les événements avant de quitter la macro d'événements, éventuellement via un gestionnaire d'erreurs "exit sécurisé".

```
Option Explicit

Private Sub Worksheet_Change(ByVal Target As Range)
    If Not Intersect(Target, Range("A:A")) Is Nothing Then
        On Error GoTo bm_Safe_Exit
        Application.EnableEvents = False

        'code that may change a value on the worksheet goes here

    End If
bm_Safe_Exit:
```

```
    Application.EnableEvents = True
End Sub
```

Une mise en garde: bien que la désactivation de ces paramètres améliore le temps d'exécution, ils peuvent rendre le débogage de votre application beaucoup plus difficile. Si votre code *ne* fonctionne *pas* correctement, `SpeedUp True` commentaire l'appel `SpeedUp True` jusqu'à ce que vous trouviez le problème.

Cela est particulièrement important si vous écrivez dans des cellules d'une feuille de calcul et que vous lisez ensuite les résultats calculés à partir des fonctions de la feuille de calcul, car `xlCalculationManual` empêche le calcul du classeur. Pour contourner ce `SpeedUp` sans désactiver `SpeedUp` , vous pouvez inclure `Application.Calculate` pour exécuter un calcul sur des points spécifiques.

REMARQUE: Comme ce sont des propriétés de l' `Application` elle-même, vous devez vous assurer qu'elles sont à nouveau activées avant la fermeture de votre macro. Cela rend particulièrement important d'utiliser des gestionnaires d'erreurs et d'éviter plusieurs points de sortie (par exemple, `End` ou `Unload Me`).

Avec gestion des erreurs:

```
Public Sub SomeMacro()
    'store the initial "calculation" state
    Dim xlCalc As XlCalculation
    xlCalc = Application.Calculation

    On Error GoTo Handler
    SpeedUp True

    'code here ...
    i = 1 / 0
CleanExit:
    SpeedUp False, xlCalc
    Exit Sub
Handler:
    'handle error
    Resume CleanExit
End Sub
```

Évitez d'utiliser ActiveCell ou ActiveSheet dans Excel

Utiliser `ActiveCell` ou `ActiveSheet` peut être source d’erreurs si (pour une raison quelconque) le code est exécuté au mauvais endroit.

```
ActiveCell.Value = "Hello"
'will place "Hello" in the cell that is currently selected
Cells(1, 1).Value = "Hello"
'will always place "Hello" in A1 of the currently selected sheet

ActiveSheet.Cells(1, 1).Value = "Hello"
'will place "Hello" in A1 of the currently selected sheet
Sheets("MySheetName").Cells(1, 1).Value = "Hello"
'will always place "Hello" in A1 of the sheet named "MySheetName"
```

- L'utilisation d' `Active*` peut créer des problèmes lors de longues macros si votre utilisateur s'ennuie et clique sur une autre feuille de calcul ou ouvre un autre classeur.
- Cela peut créer des problèmes si votre code ouvre ou crée un autre classeur.
- Cela peut créer des problèmes si votre code utilise des `Sheets("MyOtherSheet").Select` et vous avez oublié quelle feuille vous étiez avant de commencer à lire ou à écrire.

Ne jamais assumer la feuille de travail

Même lorsque tout votre travail est dirigé vers une seule feuille de calcul, il est toujours recommandé de spécifier explicitement la feuille de calcul dans votre code. Cette habitude facilite considérablement l'élargissement de votre code ultérieurement ou la levée de certaines parties (ou de toutes) d'une `Sub` - `Function` ou d'une `Function` à réutiliser ailleurs. De nombreux développeurs prennent l'habitude de réutiliser le même nom de variable locale pour une feuille de calcul dans leur code, ce qui rend la réutilisation de ce code encore plus simple.

Par exemple, le code suivant est ambigu - mais fonctionne! - tant que le développeur n'active pas ou ne change pas de feuille de calcul:

```
Option Explicit
Sub ShowTheTime()
    '--- displays the current time and date in cell A1 on the worksheet
    Cells(1, 1).Value = Now()  ' don't refer to Cells without a sheet reference!
End Sub
```

Si la `Sheet1` est active, la cellule A1 de la feuille Sheet1 sera remplie avec la date et l'heure actuelles. Mais si l'utilisateur modifie les feuilles de calcul pour une raison quelconque, le code sera mis à jour quelle que soit la feuille de calcul active. La feuille de calcul de la destination est ambiguë.

La meilleure pratique consiste à toujours identifier la feuille de calcul à laquelle votre code fait référence:

```
Option Explicit
Sub ShowTheTime()
    '--- displays the current time and date in cell A1 on the worksheet
    Dim myWB As Workbook
    Set myWB = ThisWorkbook
    Dim timestampSH As Worksheet
    Set timestampSH = myWB.Sheets("Sheet1")
    timestampSH.Cells(1, 1).Value = Now()
End Sub
```

Le code ci-dessus identifie clairement le classeur et la feuille de calcul. Bien que cela puisse sembler exagéré, créer une bonne habitude concernant les références cibles vous évitera de futurs problèmes.

Évitez d'utiliser SELECT ou ACTIVATE

Il est **très** rare que vous souhaitiez utiliser `Select` ou `Activate` dans votre code, mais certaines méthodes Excel nécessitent l'activation d'une feuille de calcul ou d'un classeur avant de pouvoir

fonctionner comme prévu.

Si vous commencez juste à apprendre VBA, vous serez souvent invité à enregistrer vos actions à l'aide de l'enregistreur de macros, puis à regarder le code. Par exemple, j'ai enregistré les actions entreprises pour entrer une valeur dans la cellule D3 de la feuille Sheet2 et le code de la macro ressemble à ceci:

```
Option Explicit
Sub Macro1()
'
' Macro1 Macro
'

'
    Sheets("Sheet2").Select
    Range("D3").Select
    ActiveCell.FormulaR1C1 = "3.1415"   '(see **note below)
    Range("D4").Select
End Sub
```

Rappelez-vous cependant que l'enregistreur de macros crée une ligne de code pour chacune de vos actions (utilisateur). Cela inclut de cliquer sur l'onglet de la feuille de calcul pour sélectionner Sheet2 (`Sheets("Sheet2").Select`), en cliquant sur la cellule D3 avant d'entrer la valeur (`Range("D3").Select` et en utilisant la touche Entrée en sélectionnant "la cellule sous la cellule actuellement sélectionnée: `Range("D4").Select`).

Il existe plusieurs problèmes d'utilisation de `.Select` ici:

- **La feuille de calcul n'est pas toujours spécifiée.** Cela se produit si vous ne changez pas de feuille de calcul pendant l'enregistrement et que le code produira des résultats différents pour les différentes feuilles de calcul actives.
- **`.Select()` est lent.** Même si `Application.ScreenUpdating` est défini sur `False` , il s'agit d'une opération inutile à traiter.
- **`.Select()` est indiscipliné.** Si `Application.ScreenUpdating` est laissé à `True` , Excel sélectionnera les cellules, la feuille de calcul, le formulaire, etc. C'est stressant pour les yeux et vraiment désagréable à regarder.
- **`.Select()` déclenchera les écouteurs.** C'est déjà un peu avancé, mais à moins d'avoir fonctionné, des fonctions comme `Worksheet_SelectionChange()` seront déclenchées.

Lorsque vous codez en VBA, toutes les actions de "typage" (c.-à-d `Select` instructions `Select`) ne sont plus nécessaires. Votre code peut être réduit à une seule instruction pour mettre la valeur dans la cellule:

```
'--- GOOD
ActiveWorkbook.Sheets("Sheet2").Range("D3").Value = 3.1415

'--- BETTER
Dim myWB      As Workbook
Dim myWS      As Worksheet
Dim myCell    As Range

Set myWB = ThisWorkbook             '*** see NOTE2
```

```
Set myWS = myWB.Sheets("Sheet2")
Set myCell = myWS.Range("D3")

myCell.Value = 3.1415
```

(Le meilleur exemple ci-dessus montre l'utilisation de variables intermédiaires pour séparer différentes parties de la référence de cellule. L'exemple GOOD fonctionnera toujours correctement, mais peut être très lourd dans des modules de code beaucoup plus longs et plus difficile à déboguer.)

** REMARQUE: l'enregistreur de macros émet de nombreuses hypothèses sur le type de données que vous entrez, en entrant dans ce cas une valeur de chaîne en tant que formule pour créer la valeur. Votre code n'a pas à le faire et peut simplement attribuer une valeur numérique directement à la cellule, comme indiqué ci-dessus.

** NOTE2: la pratique recommandée est de définir votre variable de classeur local sur `ThisWorkbook` au lieu d' `ActiveWorkbook` (sauf si vous en avez explicitement besoin). La raison en est que votre macro a généralement besoin / utilise des ressources dans le classeur dont provient le code VBA et ne regardera PAS en dehors de ce classeur, à moins que vous ne demandiez explicitement à votre code de travailler avec un autre classeur. Lorsque plusieurs classeurs sont ouverts dans Excel, `ActiveWorkbook` est celui *qui peut être différent du classeur affiché dans votre éditeur VBA* . Donc, vous pensez que vous exécutez dans un seul classeur lorsque vous faites vraiment référence à un autre. `ThisWorkbook` fait référence au classeur contenant le code en cours d'exécution.

Toujours définir et définir des références à tous les classeurs et feuilles

Lorsque vous travaillez avec plusieurs classeurs ouverts, chacun pouvant comporter plusieurs feuilles, il est plus sûr de définir et de définir une référence à tous les classeurs et feuilles.

Ne vous fiez pas à `ActiveWorkbook` ou `ActiveSheet` car ils peuvent être modifiés par l'utilisateur.

L'exemple de code suivant montre comment copier une plage de feuille « *raw_data* » dans le classeur « *Data.xlsx* » à feuille « *Refined_Data* » dans le classeur « *Results.xlsx* ».

La procédure montre également comment copier et coller sans utiliser la méthode `Select` .

```
Option Explicit

Sub CopyRanges_BetweenShts()


    Dim wbSrc                           As Workbook
    Dim wbDest                          As Workbook
    Dim shtCopy                         As Worksheet
    Dim shtPaste                        As Worksheet

    ' set reference to all workbooks by name, don't rely on ActiveWorkbook
    Set wbSrc = Workbooks("Data.xlsx")
    Set wbDest = Workbooks("Results.xlsx")

    ' set reference to all sheets by name, don't rely on ActiveSheet
```

```
    Set shtCopy = wbSrc.Sheet1 '// "Raw_Data" sheet
    Set shtPaste = wbDest.Sheet2 '// "Refined_Data") sheet

    ' copy range from "Data" workbook to "Results" workbook without using Select
    shtCopy.Range("A1:C10").Copy _
    Destination:=shtPaste.Range("A1")

End Sub
```

L'objet WorksheetFunction s'exécute plus rapidement qu'un équivalent UDF

VBA est compilé au moment de l'exécution, ce qui a un impact négatif énorme sur ses performances, tout ce qui est intégré sera plus rapide, essayez de les utiliser.

Par exemple, je compare les fonctions SUM et COUNTIF, mais vous pouvez utiliser si vous pouvez résoudre avec WorkSheetFunctions.

Une première tentative serait de parcourir la plage et de la traiter cellule par cellule (en utilisant une plage):

```
Sub UseRange()
    Dim rng as Range
    Dim Total As Double
    Dim CountLessThan01 As Long

    Total = 0
    CountLessThan01 = 0
    For Each rng in Sheets(1).Range("A1:A100")
        Total = Total + rng.Value2
        If rng.Value < 0.1 Then
            CountLessThan01 = CountLessThan01 + 1
        End If
    Next rng
    Debug.Print Total & ", " & CountLessThan01
End Sub
```

Une amélioration peut être de stocker les valeurs de plage dans un tableau et de traiter les éléments suivants:

```
Sub UseArray()
    Dim DataToSummarize As Variant
    Dim i As Long
    Dim Total As Double
    Dim CountLessThan01 As Long

    DataToSummarize = Sheets(1).Range("A1:A100").Value2 'faster than .Value
    Total = 0
    CountLessThan01 = 0
    For i = 1 To 100
        Total = Total + DataToSummarize(i, 1)
        If DataToSummarize(i, 1) < 0.1 Then
            CountLessThan01 = CountLessThan01 + 1
        End If
    Next i
    Debug.Print Total & ", " & CountLessThan01
End Sub
```

Mais au lieu d'écrire une boucle, vous pouvez utiliser `Application.Worksheetfunction` qui est très pratique pour exécuter des formules simples:

```
Sub UseWorksheetFunction()
    Dim Total As Double
    Dim CountLessThan01 As Long

    With Application.WorksheetFunction
        Total = .Sum(Sheets(1).Range("A1:A100"))
        CountLessThan01 = .CountIf(Sheets(1).Range("A1:A100"), "<0.1")
    End With

    Debug.Print Total & ", " & CountLessThan01
End Sub
```

Ou, pour des calculs plus complexes, vous pouvez même utiliser `Application.Evaluate` :

```
Sub UseEvaluate()
    Dim Total As Double
    Dim CountLessThan01 As Long

    With Application
        Total = .Evaluate("SUM(" & Sheet1.Range("A1:A100").Address( _
            external:=True) & ")")
        CountLessThan01 = .Evaluate("COUNTIF('Sheet1'!A1:A100,""<0.1"")")
    End With

    Debug.Print Total & ", " & CountLessThan01
End Sub
```

Et enfin, en cours d'exécution au-dessus de 25 000 fois chacun, voici la moyenne (5 tests) en millisecondes (bien sûr, ce sera différent sur chaque PC, mais les uns par rapport aux autres, ils se comporteront de la même manière):

1. UseWorksheetFunction: 2156 ms
2. UseArray: 2219 ms (+ 3%)
3. UseEvaluate: 4693 ms (+ 118%)
4. UseRange: 6530 ms (+ 203%)

Évitez de réutiliser les noms de propriétés ou de méthodes comme variables

Il n'est généralement pas considéré comme «meilleure pratique» de réutiliser les noms réservés de propriétés ou de méthodes en tant que noms de vos propres procédures et variables.

Mauvaise forme - Bien que ce qui suit soit (à proprement parler) légal, le code de travail, la réutilisation de la méthode Find ainsi que les propriétés Row , Column et Address peuvent provoquer des problèmes / conflits avec une ambiguïté de nom.

```
Option Explicit

Sub find()
    Dim row As Long, column As Long
    Dim find As String, address As Range
```

```
    find = "something"

    With ThisWorkbook.Worksheets("Sheet1").Cells
        Set address = .SpecialCells(xlCellTypeLastCell)
        row = .find(what:=find, after:=address).row        '< note .row not capitalized
        column = .find(what:=find, after:=address).column  '< note .column not capitalized

        Debug.Print "The first 'something' is in " & .Cells(row, column).address(0, 0)
    End With
End Sub
```

Bonne forme - Tous les mots réservés étant renommés en approximations proches mais uniques des originaux, tous les conflits de noms potentiels ont été évités.

```
Option Explicit

Sub myFind()
    Dim rw As Long, col As Long
    Dim wht As String, lastCell As Range

    wht = "something"

    With ThisWorkbook.Worksheets("Sheet1").Cells
        Set lastCell = .SpecialCells(xlCellTypeLastCell)
        rw = .Find(What:=wht, After:=lastCell).Row         '◂ note .Find and .Row
        col = .Find(What:=wht, After:=lastCell).Column     '◂ .Find and .Column

        Debug.Print "The first 'something' is in " & .Cells(rw, col).Address(0, 0)
    End With
End Sub
```

Bien qu'il puisse arriver que vous souhaitiez réécrire intentionnellement une méthode ou une propriété standard selon vos propres spécifications, ces situations sont rares. Dans la plupart des cas, évitez de réutiliser des noms réservés pour vos propres constructions.

Lire Meilleures pratiques de VBA en ligne: https://riptutorial.com/fr/excel-vba/topic/1107/meilleures-pratiques-de-vba

Chapitre 19: Méthodes de recherche de la dernière ligne ou colonne utilisée dans une feuille de calcul

Remarques

Vous pouvez trouver une bonne explication sur pourquoi les autres méthodes sont découragées / inexactes ici: http://stackoverflow.com/a/11169920/4628637

Examples

Trouver la dernière cellule non vide dans une colonne

Dans cet exemple, nous examinerons une méthode de retour de la dernière ligne non vide dans une colonne pour un ensemble de données.

Cette méthode fonctionnera indépendamment des régions vides dans l'ensemble de données.

Toutefois, ***il** faut être **prudent** si des **cellules fusionnées** sont impliquées* , car la méthode `End` sera "arrêtée" contre une région fusionnée, en renvoyant la première cellule de la région fusionnée.

De plus, les cellules non vides dans les ***lignes masquées*** ne seront pas prises en compte.

```
Sub FindingLastRow()
    Dim wS As Worksheet, LastRow As Long
    Set wS = ThisWorkbook.Worksheets("Sheet1")

    'Here we look in Column A
    LastRow = wS.Cells(wS.Rows.Count, "A").End(xlUp).Row
    Debug.Print LastRow
End Sub
```

Pour répondre aux limitations indiquées ci-dessus, la ligne:
`LastRow = wS.Cells(wS.Rows.Count, "A").End(xlUp).Row`

peut être remplacé par:

1. pour la dernière ligne utilisée de `"Sheet1"` :

   `LastRow = wS.UsedRange.Row - 1 + wS.UsedRange.Rows.Count .`

2. pour la dernière cellule non vide de la colonne `"A"` dans `"Sheet1"` :

   ```
   Dim i As Long
   For i = LastRow To 1 Step -1
       If Not (IsEmpty(Cells(i, 1))) Then Exit For
   ```

```
Next i
LastRow = i
```

Rechercher la dernière ligne à l'aide de la plage nommée

Si vous avez une plage nommée dans votre feuille et que vous souhaitez obtenir de manière dynamique la dernière ligne de cette plage nommée dynamique. Couvre également les cas où la plage nommée ne commence pas à partir de la première ligne.

```
Sub FindingLastRow()

Dim sht As Worksheet
Dim LastRow As Long
Dim FirstRow As Long

Set sht = ThisWorkbook.Worksheets("form")

'Using Named Range "MyNameRange"
FirstRow = sht.Range("MyNameRange").Row

' in case "MyNameRange" doesn't start at Row 1
LastRow = sht.Range("MyNameRange").Rows.count + FirstRow - 1

End Sub
```

Mettre à jour:
Une lacune potentielle a été signalée par @Jeeped pour une plage nommée avec des lignes non contiguës car elle génère un résultat inattendu. Pour résoudre ce problème, le code est révisé comme ci-dessous.
Asumptions: targes sheet = `form` , nommée range = `MyNameRange`

```
Sub FindingLastRow()
    Dim rw As Range, rwMax As Long
    For Each rw In Sheets("form").Range("MyNameRange").Rows
        If rw.Row > rwMax Then rwMax = rw.Row
    Next
    MsgBox "Last row of 'MyNameRange' under Sheets 'form': " & rwMax
End Sub
```

Récupère la ligne de la dernière cellule dans une plage

```
'if only one area (not multiple areas):
With Range("A3:D20")
    Debug.Print .Cells(.Cells.CountLarge).Row
    Debug.Print .Item(.Cells.CountLarge).Row 'using .item is also possible
End With 'Debug prints: 20

'with multiple areas (also works if only one area):
Dim rngArea As Range, LastRow As Long
With Range("A3:D20, E5:I50, H20:R35")
    For Each rngArea In .Areas
        If rngArea(rngArea.Cells.CountLarge).Row > LastRow Then
            LastRow = rngArea(rngArea.Cells.CountLarge).Row
        End If
```

```
    Next
    Debug.Print LastRow 'Debug prints: 50
End With
```

Trouver la dernière colonne non vide dans la feuille de calcul

```
Private Sub Get_Last_Used_Row_Index()
    Dim wS As Worksheet

    Set wS = ThisWorkbook.Sheets("Sheet1")
    Debug.Print LastCol_1(wS)
    Debug.Print LastCol_0(wS)
End Sub
```

Vous pouvez choisir entre 2 options, si vous voulez savoir s'il n'y a pas de données dans la feuille de travail:

- NO: Utilisez LastCol_1: vous pouvez l'utiliser directement dans `wS.Cells(...,LastCol_1(wS))`
- OUI: Utiliser LastCol_0: Vous devez tester si le résultat obtenu de la fonction est 0 ou non avant de l'utiliser

```
Public Function LastCol_1(wS As Worksheet) As Double
    With wS
        If Application.WorksheetFunction.CountA(.Cells) <> 0 Then
            LastCol_1 = .Cells.Find(What:="*", _
                                    After:=.Range("A1"), _
                                    Lookat:=xlPart, _
                                    LookIn:=xlFormulas, _
                                    SearchOrder:=xlByColumns, _
                                    SearchDirection:=xlPrevious, _
                                    MatchCase:=False).Column
        Else
            LastCol_1 = 1
        End If
    End With
End Function
```

Les propriétés de l'objet Err sont automatiquement réinitialisées à la sortie de la fonction.

```
Public Function LastCol_0(wS As Worksheet) As Double
    On Error Resume Next
    LastCol_0 = wS.Cells.Find(What:="*", _
                              After:=ws.Range("A1"), _
                              Lookat:=xlPart, _
                              LookIn:=xlFormulas, _
                              SearchOrder:=xlByColumns, _
                              SearchDirection:=xlPrevious, _
                              MatchCase:=False).Column
End Function
```

Dernière cellule dans Range.CurrentRegion

`Range.CurrentRegion` est une zone de plage rectangulaire entourée de cellules vides. Les cellules vides avec des formules telles que `=""` ou `'` ne sont pas considérées comme vides (même par la

fonction `ISBLANK` Excel).

```
Dim rng As Range, lastCell As Range
Set rng = Range("C3").CurrentRegion       ' or Set rng = Sheet1.UsedRange.CurrentRegion
Set lastCell = rng(rng.Rows.Count, rng.Columns.Count)
```

Trouver la dernière ligne non vide dans la feuille de calcul

```
Private Sub Get_Last_Used_Row_Index()
    Dim wS As Worksheet

    Set wS = ThisWorkbook.Sheets("Sheet1")
    Debug.Print LastRow_1(wS)
    Debug.Print LastRow_0(wS)
End Sub
```

Vous pouvez choisir entre 2 options, si vous voulez savoir s'il n'y a pas de données dans la feuille de travail:

- NO: Utilisez LastRow_1: vous pouvez l'utiliser directement dans `wS.Cells(LastRow_1(wS),...)`
- OUI: Utilisez LastRow_0: vous devez tester si le résultat obtenu de la fonction est 0 ou non avant de l'utiliser

```
Public Function LastRow_1(wS As Worksheet) As Double
    With wS
        If Application.WorksheetFunction.CountA(.Cells) <> 0 Then
            LastRow_1 = .Cells.Find(What:="*", _
                                    After:=.Range("A1"), _
                                    Lookat:=xlPart, _
                                    LookIn:=xlFormulas, _
                                    SearchOrder:=xlByRows, _
                                    SearchDirection:=xlPrevious, _
                                    MatchCase:=False).Row
        Else
            LastRow_1 = 1
        End If
    End With
End Function

Public Function LastRow_0(wS As Worksheet) As Double
    On Error Resume Next
    LastRow_0 = wS.Cells.Find(What:="*", _
                              After:=ws.Range("A1"), _
                              Lookat:=xlPart, _
                              LookIn:=xlFormulas, _
                              SearchOrder:=xlByRows, _
                              SearchDirection:=xlPrevious, _
                              MatchCase:=False).Row
End Function
```

Trouver la dernière cellule non vide dans une ligne

Dans cet exemple, nous allons examiner une méthode pour retourner la dernière colonne non vide dans une ligne.

Cette méthode fonctionnera indépendamment des régions vides dans l'ensemble de données.

Toutefois, ***il** faut être **prudent** si des **cellules fusionnées** sont impliquées* , car la méthode `End` sera "arrêtée" contre une région fusionnée, en renvoyant la première cellule de la région fusionnée.

De plus, les cellules non vides dans les ***colonnes masquées*** ne seront pas prises en compte.

```
Sub FindingLastCol()
    Dim wS As Worksheet, LastCol As Long
    Set wS = ThisWorkbook.Worksheets("Sheet1")

    'Here we look in Row 1
    LastCol = wS.Cells(1, wS.Columns.Count).End(xlToLeft).Column
    Debug.Print LastCol
End Sub
```

Trouver la dernière cellule non vide dans la feuille de calcul - Performances (tableau)

- La première fonction, utilisant un tableau, est **beaucoup plus rapide**
- Si elle est appelée sans le paramètre facultatif, elle passera par défaut à `.ThisWorkbook.ActiveSheet`
- Si la plage est vide, la `Cell( 1, 1 )` remplacée par défaut, au lieu de `Nothing`

La vitesse:

```
GetMaxCell (Array): Duration: 0.0000790063 seconds
GetMaxCell (Find ): Duration: 0.0002903480 seconds
```

.Mesuré avec MicroTimer

```
Public Function GetLastCell(Optional ByVal ws As Worksheet = Nothing) As Range
    Dim uRng As Range, uArr As Variant, r As Long, c As Long
    Dim ubR As Long, ubC As Long, lRow As Long

    If ws Is Nothing Then Set ws = Application.ThisWorkbook.ActiveSheet
    Set uRng = ws.UsedRange
    uArr = uRng
    If IsEmpty(uArr) Then
        Set GetLastCell = ws.Cells(1, 1):   Exit Function
    End If
    If Not IsArray(uArr) Then
        Set GetLastCell = ws.Cells(uRng.Row, uRng.Column):  Exit Function
    End If
    ubR = UBound(uArr, 1):  ubC = UBound(uArr, 2)
    For r = ubR To 1 Step -1    '---------------------------------------------- last row
        For c = ubC To 1 Step -1
            If Not IsError(uArr(r, c)) Then
                If Len(Trim$(uArr(r, c))) > 0 Then
                    lRow = r:   Exit For
                End If
            End If
        Next
```

```
        If lRow > 0 Then Exit For
    Next
    If lRow = 0 Then lRow = ubR
    For c = ubC To 1 Step -1    '---------------------------------------------- last col
        For r = lRow To 1 Step -1
            If Not IsError(uArr(r, c)) Then
                If Len(Trim$(uArr(r, c))) > 0 Then
                    Set GetLastCell = ws.Cells(lRow + uRng.Row - 1, c + uRng.Column - 1)
                    Exit Function
                End If
            End If
        Next
    Next
End Function
```

```
'Returns last cell (max row & max col) using Find

Public Function GetMaxCell2(Optional ByRef rng As Range = Nothing) As Range 'Using Find

    Const NONEMPTY As String = "*"

    Dim lRow As Range, lCol As Range

    If rng Is Nothing Then Set rng = Application.ThisWorkbook.ActiveSheet.UsedRange

    If WorksheetFunction.CountA(rng) = 0 Then
        Set GetMaxCell2 = rng.Parent.Cells(1, 1)
    Else
        With rng
            Set lRow = .Cells.Find(What:=NONEMPTY, LookIn:=xlFormulas, _
                                   After:=.Cells(1, 1), _
                                   SearchDirection:=xlPrevious, _
                                   SearchOrder:=xlByRows)
            If Not lRow Is Nothing Then
                Set lCol = .Cells.Find(What:=NONEMPTY, LookIn:=xlFormulas, _
                                       After:=.Cells(1, 1), _
                                       SearchDirection:=xlPrevious, _
                                       SearchOrder:=xlByColumns)

                Set GetMaxCell2 = .Parent.Cells(lRow.Row, lCol.Column)
            End If
        End With
    End If
End Function
```

.

MicroTimer :

```
Private Declare PtrSafe Function getFrequency Lib "Kernel32" Alias "QueryPerformanceFrequency"
(cyFrequency As Currency) As Long
Private Declare PtrSafe Function getTickCount Lib "Kernel32" Alias "QueryPerformanceCounter"
(cyTickCount As Currency) As Long

Function MicroTimer() As Double
    Dim cyTicks1 As Currency
    Static cyFrequency As Currency
```

```
    MicroTimer = 0
    If cyFrequency = 0 Then getFrequency cyFrequency        'Get frequency
    getTickCount cyTicks1                                   'Get ticks
    If cyFrequency Then MicroTimer = cyTicks1 / cyFrequency 'Returns Seconds
End Function
```

Lire Méthodes de recherche de la dernière ligne ou colonne utilisée dans une feuille de calcul en ligne: https://riptutorial.com/fr/excel-vba/topic/918/methodes-de-recherche-de-la-derniere-ligne-ou-colonne-utilisee-dans-une-feuille-de-calcul

Chapitre 20: Mise en forme conditionnelle à l'aide de VBA

Remarques

Vous ne pouvez pas définir plus de trois formats conditionnels pour une plage. Utilisez la méthode Modify pour modifier un format conditionnel existant ou utilisez la méthode Delete pour supprimer un format existant avant d'en ajouter un nouveau.

Examples

FormatConditions.Ajouter

Syntaxe:

```
FormatConditions.Add(Type, Operator, Formula1, Formula2)
```

Paramètres:

prénom	Obligatoire / Facultatif	Type de données
Type	Champs obligatoires	XlFormatConditionType
Opérateur	Optionnel	Une variante
Formule 1	Optionnel	Une variante
Formule2	Optionnel	Une variante

Énumération XlFormatConditionType:

prénom	La description
xlAboveAverageCondition	Au dessus de la moyenne
xlBlanksCondition	Condition des blancs
xlCellValue	Valeur de cellule
xlColorScale	Échelle de couleur

prénom	La description
xlDatabar	Barre de données
xlErrorsCondition	Condition d'erreur
xlExpression	Expression
XlIconSet	Jeu d'icônes
xlNoBlanksCondition	Aucune condition de blanc
xlNoErrorsCondition	Aucune condition d'erreur
xlTextString	Chaîne de texte
xlTimePeriod	Période de temps
xlTop10	Top 10 des valeurs
xlUniqueValues	Valeurs uniques

Formatage par valeur de cellule:

```
With Range("A1").FormatConditions.Add(xlCellValue, xlGreater, "=100")
    With .Font
        .Bold = True
        .ColorIndex = 3
     End With
End With
```

Les opérateurs:

prénom
xlEntre
xlEqual
xlGreater
xlGreaterEqual
xlLess
xlLessEqual
xlPasB / Entre

prénom
xlNotEqual

Si Type est xlExpression, l'argument Opérateur est ignoré.

Le formatage par texte contient:

```
With Range("a1:a10").FormatConditions.Add(xlTextString, TextOperator:=xlContains,
String:="egg")
    With .Font
        .Bold = True
        .ColorIndex = 3
    End With
End With
```

Les opérateurs:

prénom	La description
xlBeginsAvec	Commence avec une valeur spécifiée.
xlcontient	Contient une valeur spécifiée.
xlDoesNotContain	Ne contient pas la valeur spécifiée.
xlEndsAvec	Se termine avec la valeur spécifiée

Formatage par période

```
With Range("a1:a10").FormatConditions.Add(xlTimePeriod, DateOperator:=xlToday)
    With .Font
        .Bold = True
        .ColorIndex = 3
    End With
End With
```

Les opérateurs:

prénom
xlHier
xl
xlLast7Days

prénom
xlLastWeek
xlheure
xlNextWeek
xlLastMonth
xlThisMonth
xlNextMonth

Supprimer le format conditionnel

Supprimer tous les formats conditionnels dans la plage:

```
Range("A1:A10").FormatConditions.Delete
```

Supprimer tous les formats conditionnels dans la feuille de calcul:

```
Cells.FormatConditions.Delete
```

FormatConditions.AddUniqueValues

Mise en évidence des valeurs en double

```
With Range("E1:E100").FormatConditions.AddUniqueValues
   .DupeUnique = xlDuplicate
   With .Font
       .Bold = True
       .ColorIndex = 3
   End With
End With
```

Mettre en valeur des valeurs uniques

```
With Range("E1:E100").FormatConditions.AddUniqueValues
   With .Font
       .Bold = True
       .ColorIndex = 3
   End With
End With
```

FormatConditions.AddTop10

Mise en évidence des 5 meilleures valeurs

```
With Range("E1:E100").FormatConditions.AddTop10
    .TopBottom = xlTop10Top
    .Rank = 5
    .Percent = False
    With .Font
        .Bold = True
        .ColorIndex = 3
    End With
End With
```

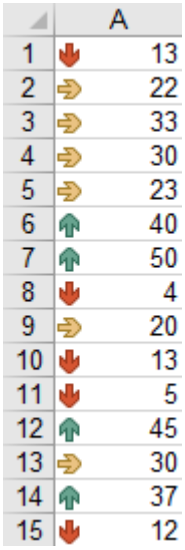
FormatConditions.AddAboveAverage

```
With Range("E1:E100").FormatConditions.AddAboveAverage
    .AboveBelow = xlAboveAverage
    With .Font
        .Bold = True
        .ColorIndex = 3
    End With
End With
```

Les opérateurs:

prénom	La description
XlAboveAverage	Au dessus de la moyenne
XlAboveStdDev	Au-dessus de l'écart type
XlBelowAverage	Sous la moyenne
XlBelowStdDev	En dessous de l'écart type
XlEqualAboveAverage	Égal au dessus de la moyenne
XlEqualBelowAverage	Égale à la moyenne

FormatConditions.AddIconSetCondition



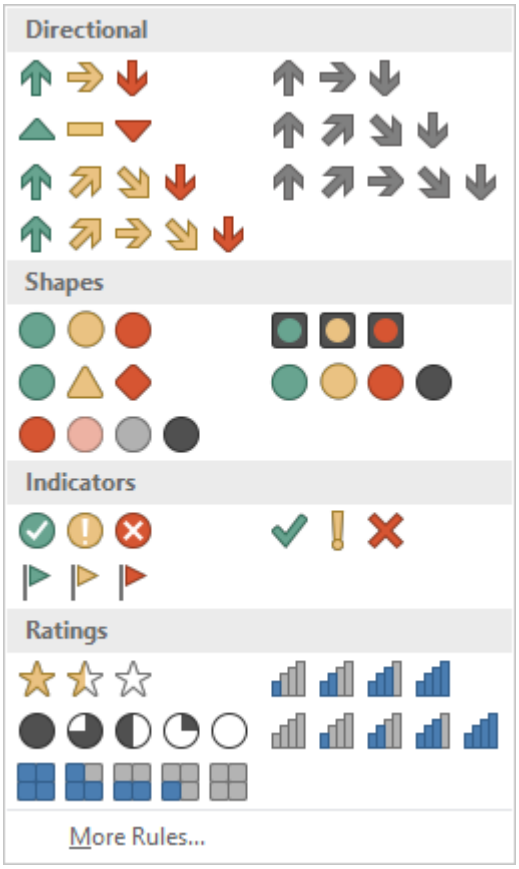
	A
1	13
2	22
3	33
4	30
5	23
6	40
7	50
8	4
9	20
10	13
11	5
12	45
13	30
14	37
15	12

```
Range("a1:a10").FormatConditions.AddIconSetCondition
With Selection.FormatConditions(1)
    .ReverseOrder = False
    .ShowIconOnly = False
    .IconSet = ActiveWorkbook.IconSets(xl3Arrows)
End With

With Selection.FormatConditions(1).IconCriteria(2)
    .Type = xlConditionValuePercent
    .Value = 33
    .Operator = 7
End With

With Selection.FormatConditions(1).IconCriteria(3)
    .Type = xlConditionValuePercent
    .Value = 67
    .Operator = 7
End With
```

IconSet:

prénom
xl3Arrows
xl3ArrowsGray
xl3Flags
xl3Signs
xl3Stars
xl3Symbols
xl3Symbols2
xl3TrafficLights1

prénom
xl3TrafficLights2
xl3Triangles
xl4Arrows
xl4ArrowsGray
xl4CRV
xl4RedToBlack
xl4TrafficLights
xl5Arrows
xl5ArrowsGray
xl5Boxes
xl5CRV
xl5Quarters



Type:

prénom
xlConditionValuePercent
xlConditionValueNumber
xlConditionValuePercentile
xlConditionValueFormula

Opérateur:

prénom	Valeur
xlGreater	5
xlGreaterEqual	7

Valeur:

Renvoie ou définit la valeur de seuil pour une icône dans un format conditionnel.

Lire Mise en forme conditionnelle à l'aide de VBA en ligne: https://riptutorial.com/fr/excel-vba/topic/9912/mise-en-forme-conditionnelle-a-l-aide-de-vba

Chapitre 21: Objet d'application

Remarques

Excel VBA est fourni avec un *modèle d'objet* complet qui contient des classes et des objets que vous pouvez utiliser pour manipuler n'importe quelle partie de l'application Excel en cours d'exécution. L'un des objets les plus courants que vous utiliserez est l'objet **Application** . Ceci est un ensemble de couches de niveau supérieur qui représente l'instance en cours d'exécution d'Excel. Presque tout ce qui n'est pas connecté à un classeur Excel particulier se trouve dans l'objet **Application** .

L'objet *Application* , en tant qu'objet de niveau supérieur, possède littéralement des centaines de propriétés, méthodes et événements pouvant être utilisés pour contrôler tous les aspects d'Excel.

Examples

Exemple d'objet d'application simple: Minimiser la fenêtre Excel

Ce code utilise l'objet **Application de** niveau supérieur pour minimiser la fenêtre Excel principale.

```
Sub MinimizeExcel()

    Application.WindowState = xlMinimized

End Sub
```

Exemple d'application simple: Afficher les versions Excel et VBE

```
Sub DisplayExcelVersions()

    MsgBox "The version of Excel is " & Application.Version
    MsgBox "The version of the VBE is " & Application.VBE.Version

End Sub
```

L'utilisation de la propriété Application.Version est utile pour garantir que le code ne fonctionne que sur une version compatible d'Excel.

Lire Objet d'application en ligne: https://riptutorial.com/fr/excel-vba/topic/5645/objet-d-application

Chapitre 22: Objet du système de fichiers

Examples

Fichier, dossier, lecteur existe

Le fichier existe:

```
Sub FileExists()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    If fso.FileExists("D:\test.txt") = True Then
       MsgBox "The file is exists."
    Else
       MsgBox "The file isn't exists."
    End If
End Sub
```

Le dossier existe:

```
Sub FolderExists()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    If fso.FolderExists("D:\testFolder") = True Then
       MsgBox "The folder is exists."
    Else
       MsgBox "The folder isn't exists."
    End If
End Sub
```

Drive existe:

```
Sub DriveExists()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    If fso.DriveExists("D:\") = True Then
       MsgBox "The drive is exists."
    Else
       MsgBox "The drive isn't exists."
    End If
End Sub
```

Opérations de base sur les fichiers

Copie:

```
Sub CopyFile()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    fso.CopyFile "c:\Documents and Settings\Makro.txt", "c:\Documents and Settings\Macros\"
End Sub
```

Bouge toi:

```
Sub MoveFile()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    fso.MoveFile "c:\*.txt", "c:\Documents and Settings\"
End Sub
```

Effacer:

```
Sub DeleteFile()
    Dim fso
    Set fso = CreateObject("Scripting.FileSystemObject")
    fso.DeleteFile "c:\Documents and Settings\Macros\Makro.txt"
End Sub
```

Opérations de base du dossier

Créer:

```
Sub CreateFolder()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    fso.CreateFolder "c:\Documents and Settings\NewFolder"
End Sub
```

Copie:

```
Sub CopyFolder()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    fso.CopyFolder "C:\Documents and Settings\NewFolder", "C:\"
End Sub
```

Bouge toi:

```
Sub MoveFolder()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    fso.MoveFolder "C:\Documents and Settings\NewFolder", "C:\"
End Sub
```

Effacer:

```
Sub DeleteFolder()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    fso.DeleteFolder "C:\Documents and Settings\NewFolder"
End Sub
```

Autres opérations

Obtenir le nom du fichier:

```
Sub GetFileName()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    MsgBox fso.GetFileName("c:\Documents and Settings\Makro.txt")
End Sub
```

Résultat: *Makro.txt*

Obtenez le nom de base:

```
Sub GetBaseName()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    MsgBox fso.GetBaseName("c:\Documents and Settings\Makro.txt")
End Sub
```

Résultat: *Makro*

Obtenir le nom de l'extension:

```
Sub GetExtensionName()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    MsgBox fso.GetExtensionName("c:\Documents and Settings\Makro.txt")
```

```
End Sub
```

Résultat: *txt*

Obtenir le nom du lecteur:

```
Sub GetDriveName()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    MsgBox fso.GetDriveName("c:\Documents and Settings\Makro.txt")
End Sub
```

Résultat: *c:*

Lire Objet du système de fichiers en ligne: https://riptutorial.com/fr/excel-vba/topic/9933/objet-du-systeme-de-fichiers

Chapitre 23: Optimisation Excel-VBA

Introduction

Excel-VBA Optimization se réfère également à un meilleur traitement des erreurs par la documentation et des détails supplémentaires. Ceci est montré ici.

Remarques

*) Les numéros de ligne représentent des entiers, c'est-à-dire un type de données signé à 16 bits compris entre -32 768 et 32 767, sinon vous produisez un dépassement de capacité. Habituellement, les numéros de ligne sont insérés par pas de 10 sur une partie du code ou sur toutes les procédures d'un module dans son ensemble.

Examples

Désactivation de la mise à jour de la feuille de calcul

La désactivation du calcul de la feuille de calcul peut réduire considérablement le temps d'exécution de la macro. De plus, la désactivation des événements, la mise à jour de l'écran et les sauts de page seraient bénéfiques. La suite `Sub` peut être utilisée dans n'importe quelle macro à cette fin.

```
Sub OptimizeVBA(isOn As Boolean)
    Application.Calculation = IIf(isOn, xlCalculationManual, xlCalculationAutomatic)
    Application.EnableEvents = Not(isOn)
    Application.ScreenUpdating = Not(isOn)
    ActiveSheet.DisplayPageBreaks = Not(isOn)
End Sub
```

Pour l'optimisation, suivez le pseudo-code ci-dessous:

```
Sub MyCode()

    OptimizeVBA True

    'Your code goes here

    OptimizeVBA False

End Sub
```

Vérification de l'heure d'exécution

Des procédures différentes peuvent donner le même résultat, mais elles utiliseraient un temps de traitement différent. Pour vérifier lequel est le plus rapide, un code comme celui-ci peut être utilisé:

```
time1 = Timer

For Each iCell In MyRange
   iCell = "text"
Next iCell

time2 = Timer

For i = 1 To 30
   MyRange.Cells(i) = "text"
Next i

time3 = Timer

debug.print "Proc1 time: " & cStr(time2-time1)
debug.print "Proc2 time: " & cStr(time3-time2)
```

MicroTimer :

```
Private Declare PtrSafe Function getFrequency Lib "Kernel32" Alias "QueryPerformanceFrequency"
(cyFrequency As Currency) As Long
Private Declare PtrSafe Function getTickCount Lib "Kernel32" Alias "QueryPerformanceCounter"
(cyTickCount As Currency) As Long

Function MicroTimer() As Double
    Dim cyTicks1 As Currency
    Static cyFrequency As Currency

    MicroTimer = 0
    If cyFrequency = 0 Then getFrequency cyFrequency        'Get frequency
    getTickCount cyTicks1                                   'Get ticks
    If cyFrequency Then MicroTimer = cyTicks1 / cyFrequency 'Returns Seconds
End Function
```

Utiliser des blocs avec

L'utilisation de blocs peut accélérer le processus d'exécution d'une macro. Au lieu d'écrire une plage, un nom de graphique, une feuille de calcul, etc., vous pouvez utiliser des blocs comme ci-dessous;

```
With ActiveChart
    .Parent.Width = 400
    .Parent.Height = 145
    .Parent.Top = 77.5 + 165 * step - replacer * 15
    .Parent.Left = 5
End With
```

Ce qui est plus rapide que cela:

```
ActiveChart.Parent.Width = 400
ActiveChart.Parent.Height = 145
ActiveChart.Parent.Top = 77.5 + 165 * step - replacer * 15
ActiveChart.Parent.Left = 5
```

Remarques:

- Une fois qu'un bloc With est entré, l'objet ne peut plus être modifié. Par conséquent, vous ne pouvez pas utiliser une seule instruction With pour affecter plusieurs objets différents.
- **Ne sautez pas dans ou hors des blocs With** . Si des instructions dans un bloc With sont exécutées, mais que l'instruction With ou End With n'est pas exécutée, **une variable temporaire contenant une référence à l'objet reste en mémoire jusqu'à ce que vous quittiez la procédure**
- Ne pas boucler l'intérieur avec des instructions, surtout si l'objet mis en cache est utilisé comme itérateur
- Vous pouvez imbriquer des instructions en en plaçant une avec un bloc dans une autre. Cependant, comme les membres des blocs With externes sont masqués dans l'intérieur avec des blocs, vous devez fournir une référence d'objet pleinement qualifiée dans un bloc With interne à tout membre d'un objet dans un bloc With externe.

Exemple d'imbrication:

Cet exemple utilise l'instruction With pour exécuter une série d'instructions sur un seul objet. L'objet et ses propriétés sont des noms génériques utilisés à des fins d'illustration uniquement.

```
With MyObject
    .Height = 100               'Same as MyObject.Height = 100.
    .Caption = "Hello World"    'Same as MyObject.Caption = "Hello World".
    With .Font
        .Color = Red            'Same as MyObject.Font.Color = Red.
        .Bold = True            'Same as MyObject.Font.Bold = True.
        MyObject.Height = 200   'Inner-most With refers to MyObject.Font (must be qualified
    End With
End With
```

Plus d'infos sur MSDN

Suppression de lignes - Performance

- La suppression des lignes est lente, en particulier lors de la lecture en boucle de cellules et de la suppression de lignes, une par une
- Une approche différente consiste à utiliser un filtre automatique pour masquer les lignes à supprimer.
- Copiez la plage visible et collez-la dans une nouvelle feuille de travail
- Retirez entièrement la feuille initiale
- Avec cette méthode, plus il y a de lignes à supprimer, plus vite ce sera

Exemple:

```
Option Explicit

'Deleted rows: 775,153, Total Rows: 1,000,009, Duration: 1.87 sec

Public Sub DeleteRows()
    Dim oldWs As Worksheet, newWs As Worksheet, wsName As String, ur As Range

    Set oldWs = ThisWorkbook.ActiveSheet
    wsName = oldWs.Name
    Set ur = oldWs.Range("F2", oldWs.Cells(oldWs.Rows.Count, "F").End(xlUp))

    Application.ScreenUpdating = False
    Set newWs = Sheets.Add(After:=oldWs)    'Create a new WorkSheet

    With ur    'Copy visible range after Autofilter (modify Criteria1 and 2 accordingly)
        .AutoFilter Field:=1, Criteria1:="<>0", Operator:=xlAnd, Criteria2:="<>"
        oldWs.UsedRange.Copy
    End With
    'Paste all visible data into the new WorkSheet (values and formats)
    With newWs.Range(oldWs.UsedRange.Cells(1).Address)
        .PasteSpecial xlPasteColumnWidths
        .PasteSpecial xlPasteAll
        newWs.Cells(1, 1).Select: newWs.Cells(1, 1).Copy
    End With

    With Application
        .CutCopyMode = False
        .DisplayAlerts = False
            oldWs.Delete
        .DisplayAlerts = True
        .ScreenUpdating = True
    End With
    newWs.Name = wsName
End Sub
```

Désactivation de toutes les fonctionnalités d'Excel Avant d'exécuter de grandes macros

Les procédures ci-dessous désactivent temporairement toutes les fonctionnalités Excel au niveau WorkBook et WorkSheet.

- FastWB () est une bascule qui accepte les indicateurs On ou Off
- FastWS () accepte un objet facultatif WorkSheet, ou aucun
- Si le paramètre ws est manquant, toutes les fonctionnalités seront activées et désactivées pour toutes les feuilles de calcul de la collection.
    - Un type personnalisé peut être utilisé pour capturer tous les paramètres avant de les désactiver
    - À la fin du processus, les paramètres initiaux peuvent être restaurés

---

```
Public Sub FastWB(Optional ByVal opt As Boolean = True)
    With Application
        .Calculation = IIf(opt, xlCalculationManual, xlCalculationAutomatic)
```

```
        If .DisplayAlerts <> Not opt Then .DisplayAlerts = Not opt
        If .DisplayStatusBar <> Not opt Then .DisplayStatusBar = Not opt
        If .EnableAnimations <> Not opt Then .EnableAnimations = Not opt
        If .EnableEvents <> Not opt Then .EnableEvents = Not opt
        If .ScreenUpdating <> Not opt Then .ScreenUpdating = Not opt
    End With
    FastWS , opt
End Sub
```

```
Public Sub FastWS(Optional ByVal ws As Worksheet, Optional ByVal opt As Boolean = True)
    If ws Is Nothing Then
        For Each ws In Application.ThisWorkbook.Sheets
            OptimiseWS ws, opt
        Next
    Else
        OptimiseWS ws, opt
    End If
End Sub
Private Sub OptimiseWS(ByVal ws As Worksheet, ByVal opt As Boolean)
    With ws
        .DisplayPageBreaks = False
        .EnableCalculation = Not opt
        .EnableFormatConditionsCalculation = Not opt
        .EnablePivotTable = Not opt
    End With
End Sub
```

Restaurer tous les paramètres Excel par défaut

```
Public Sub XlResetSettings()    'default Excel settings
    With Application
        .Calculation = xlCalculationAutomatic
        .DisplayAlerts = True
        .DisplayStatusBar = True
        .EnableAnimations = False
        .EnableEvents = True
        .ScreenUpdating = True
        Dim sh As Worksheet
        For Each sh In Application.ThisWorkbook.Sheets
            With sh
                .DisplayPageBreaks = False
                .EnableCalculation = True
                .EnableFormatConditionsCalculation = True
                .EnablePivotTable = True
            End With
        Next
    End With
End Sub
```

Optimisation de la recherche d'erreur par débogage étendu

Utiliser des numéros de ligne ... et les documenter en cas d'erreur ("L'importance de voir Erl")

Détecter quelle ligne soulève une erreur est une partie substantielle de tout débogage et réduit la recherche de la cause. Pour documenter les lignes d'erreur identifiées avec une brève description,

le suivi des erreurs est réussi, au mieux, avec les noms des modules et des procédures. L'exemple ci-dessous enregistre ces données dans un fichier journal.

Contexte

L'objet d'erreur renvoie le numéro d'erreur (Err.Number) et la description de l'erreur (Err.Description), mais ne répond pas explicitement à la question de l'emplacement de l'erreur. La fonction **Erl,** cependant, le fait, mais à la condition que vous ajoutez les *numéros de ligne *)* au code (BTW une de plusieurs autres concessions aux anciens temps de base).

S'il n'y a aucune ligne d'erreur, alors la fonction Erl renvoie 0, si la numérotation est incomplète, vous obtenez le dernier numéro de ligne de la procédure.

```
Option Explicit


Public Sub MyProc1()
Dim i As Integer
Dim j As Integer
On Error GoTo LogErr
10     j = 1 / 0    ' raises an error
okay:
Debug.Print "i=" & i
Exit Sub

LogErr:
MsgBox LogErrors("MyModule", "MyProc1", Err), vbExclamation, "Error " & Err.Number
Stop
Resume Next
End Sub

Public Function LogErrors( _
           ByVal sModule As String, _
           ByVal sProc As String, _
           Err As ErrObject) As String
' Purpose: write error number, description and Erl to log file and return error text
  Dim sLogFile As String: sLogFile = ThisWorkbook.Path & Application.PathSeparator &
"LogErrors.txt"
  Dim sLogTxt  As String
  Dim lFile    As Long

' Create error text
  sLogTxt = sModule & "|" & sProc & "|Erl " & Erl & "|Err " & Err.Number & "|" &
Err.Description

  On Error Resume Next
  lFile = FreeFile

  Open sLogFile For Append As lFile
  Print #lFile, Format$(Now(), "yy.mm.dd hh:mm:ss "); sLogTxt
      Print #lFile,
  Close lFile
' Return error text
  LogErrors = sLogTxt
 End Function
```

' **Code supplémentaire pour afficher le fichier journal**

```
Sub ShowLogFile()
Dim sLogFile As String: sLogFile = ThisWorkbook.Path & Application.PathSeparator &
"LogErrors.txt"

On Error GoTo LogErr
Shell "notepad.exe " & sLogFile, vbNormalFocus

okay:
On Error Resume Next
Exit Sub

LogErr:
MsgBox LogErrors("MyModule", "ShowLogFile", Err), vbExclamation, "Error No " & Err.Number
Resume okay
End Sub
```

Lire Optimisation Excel-VBA en ligne: https://riptutorial.com/fr/excel-vba/topic/9798/optimisation-excel-vba

Chapitre 24: Sécurité VBA

Examples

Mot de passe Protégez votre VBA

Vous avez parfois des informations sensibles dans votre VBA (par exemple, des mots de passe) auxquelles vous ne souhaitez pas que les utilisateurs puissent accéder. Vous pouvez obtenir une sécurité de base sur ces informations en protégeant votre projet VBA par mot de passe.

Suivez ces étapes:

1. Ouvrez votre éditeur Visual Basic (Alt + F11)
2. Accédez à Outils -> Propriétés VBAProject ...
3. Accédez à l'onglet Protection.
4. Cochez la case "Verrouiller le projet pour consultation"
5. Entrez le mot de passe souhaité dans les zones de texte Mot de passe et Confirmer le mot de passe

Maintenant, lorsque quelqu'un veut accéder à votre code dans une application Office, il doit d'abord entrer le mot de passe. Soyez conscient, cependant, que même un mot de passe de projet VBA fort est trivial à briser.

Lire Sécurité VBA en ligne: https://riptutorial.com/fr/excel-vba/topic/7642/securite-vba

Chapitre 25: SQL dans Excel VBA - Meilleures pratiques

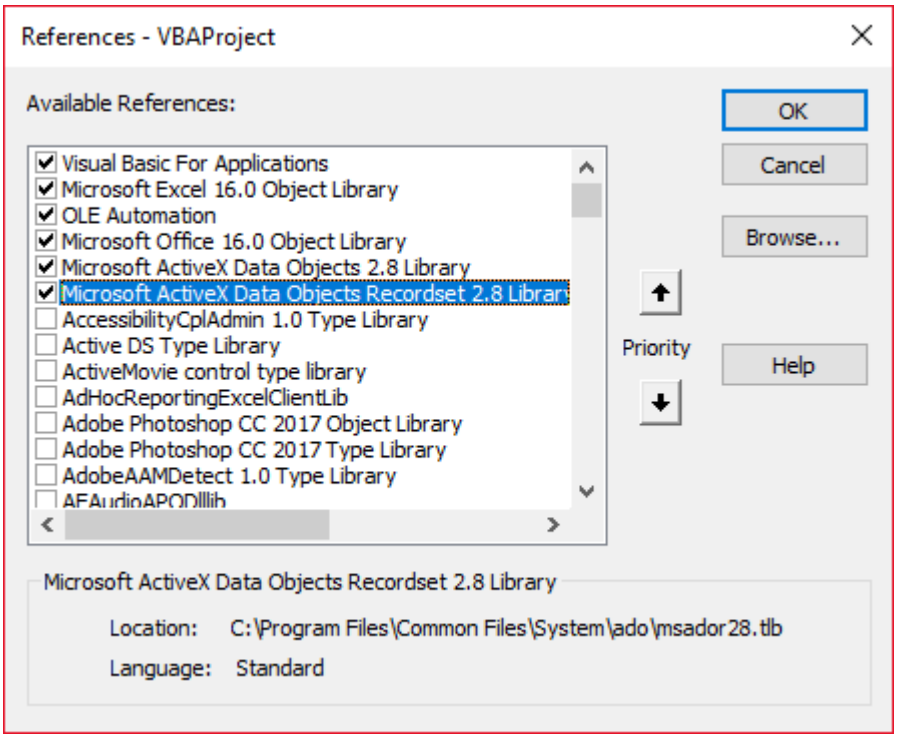
Examples

Comment utiliser ADODB.Connection dans VBA?

Exigences:

Ajouter les références suivantes au projet:

- Bibliothèque Microsoft ActiveX Data Objects 2.8
- Bibliothèque de jeu d'enregistrements Microsoft ActiveX Data Objects 2.8



Déclarer des variables

```
Private mDataBase As New ADODB.Connection
Private mRS As New ADODB.Recordset
Private mCmd As New ADODB.Command
```

Créer une connexion

une. avec l'authentification Windows

```
Private Sub OpenConnection(pServer As String, pCatalog As String)
    Call mDataBase.Open("Provider=SQLOLEDB;Initial Catalog=" & pCatalog & ";Data Source=" &
pServer & ";Integrated Security=SSPI")
    mCmd.ActiveConnection = mDataBase
End Sub
```

b. avec l'authentification SQL Server

```
Private Sub OpenConnection2(pServer As String, pCatalog As String, pUser As String, pPsw As
String)
    Call mDataBase.Open("Provider=SQLOLEDB;Initial Catalog=" & pCatalog & ";Data Source=" &
pServer & ";Integrated Security=SSPI;User ID=" & pUser & ";Password=" & pPsw)
    mCmd.ActiveConnection = mDataBase
End Sub
```

Exécuter la commande sql

```
Private Sub ExecuteCmd(sql As String)
    mCmd.CommandText = sql
    Set mRS = mCmd.Execute
End Sub
```

Lire les données du jeu d'enregistrements

```
Private Sub ReadRS()
    Do While Not (mRS.EOF)
        Debug.Print "ShipperID: " & mRS.Fields("ShipperID").Value & " CompanyName: " &
mRS.Fields("CompanyName").Value & " Phone: " & mRS.Fields("Phone").Value
        Call mRS.MoveNext
    Loop
End Sub
```

Fermer la connexion

```
Private Sub CloseConnection()
    Call mDataBase.Close
    Set mRS = Nothing
    Set mCmd = Nothing
    Set mDataBase = Nothing
End Sub
```

Comment l'utiliser?

```
Public Sub Program()
    Call OpenConnection("ServerName", "NORTHWND")
    Call ExecuteCmd("INSERT INTO [NORTHWND].[dbo].[Shippers]([CompanyName],[Phone]) Values
('speedy shipping','(503) 555-1234')")
    Call ExecuteCmd("SELECT * FROM [NORTHWND].[dbo].[Shippers]")
    Call ReadRS
    Call CloseConnection
End Sub
```

Résultat

Expéditeur: 1 Nom de l'entreprise: Speedy Express Téléphone: (503) 555-9831

ID expéditeur: 2 CompanyName: United Package Téléphone: (503) 555-3199

ShipperID: 3 CompanyName: Federal Shipping Téléphone: (503) 555-9931

Expéditeur: 4 Nom de l'entreprise: expédition rapide Téléphone: (503) 555-1234

Lire SQL dans Excel VBA - Meilleures pratiques en ligne: https://riptutorial.com/fr/excel-vba/topic/9958/sql-dans-excel-vba---meilleures-pratiques

Chapitre 26: Tableaux

Examples

Remplissage des tableaux (ajout de valeurs)

Il existe plusieurs façons de remplir un tableau.

Directement

```
'one-dimensional
Dim arrayDirect1D(2) As String
arrayDirect(0) = "A"
arrayDirect(1) = "B"
arrayDirect(2) = "C"

'multi-dimensional (in this case 3D)
Dim arrayDirectMulti(1, 1, 2)
arrayDirectMulti(0, 0, 0) = "A"
arrayDirectMulti(0, 0, 1) = "B"
arrayDirectMulti(0, 0, 2) = "C"
arrayDirectMulti(0, 1, 0) = "D"
'...
```

Utilisation de la fonction Array ()

```
'one-dimensional only
Dim array1D As Variant 'has to be type variant
array1D = Array(1, 2, "A")
'-> array1D(0) = 1, array1D(1) = 2, array1D(2) = "A"
```

De la gamme

```
Dim arrayRange As Variant 'has to be type variant

'putting ranges in an array always creates a 2D array (even if only 1 row or column)
'starting at 1 and not 0, first dimension is the row and the second the column
arrayRange = Range("A1:C10").Value
'-> arrayRange(1,1) = value in A1
'-> arrayRange(1,2) = value in B1
'-> arrayRange(5,3) = value in C5
'...

'Yoo can get an one-dimensional array from a range (row or column)
'by using the worksheet functions index and transpose:
```

```
'one row from range into 1D-Array:
arrayRange = Application.WorksheetFunction.Index(Range("A1:C10").Value, 3, 0)
'-> row 3 of range into 1D-Array
'-> arrayRange(1) = value in A3, arrayRange(2) = value in B3, arrayRange(3) = value in C3

'one column into 1D-Array:
'limited to 65536 rows in the column, reason: limit of .Transpose
arrayRange = Application.WorksheetFunction.Index( _
Application.WorksheetFunction.Transpose(Range("A1:C10").Value), 2, 0)
'-> column 2 of range into 1D-Array
'-> arrayRange(1) = value in B1, arrayRange(2) = value in B2, arrayRange(3) = value in B3
'...

'By using Evaluate() - shorthand [] - you can transfer the
'range to an array and change the values at the same time.
'This is equivalent to an array formula in the sheet:
arrayRange = [(A1:C10*3)]
arrayRange = [(A1:C10&"_test")]
arrayRange = [(A1:B10*C1:C10)]
'...
```

2D avec Evaluer ()

```
Dim array2D As Variant
'[] ist a shorthand for evaluate()
'Arrays defined with evaluate start at 1 not 0
array2D = [{"1A","1B","1C";"2A","2B","3B"}]
'-> array2D(1,1) = "1A", array2D(1,2) = "1B", array2D(2,1) = "2A" ...

'if you want to use a string to fill the 2D-Array:
Dim strValues As String
strValues = "{""1A"",""1B"",""1C"";""2A"",""2B"",""2C""}"
array2D = Evaluate(strValues)
```

Utiliser la fonction Split ()

```
Dim arraySplit As Variant 'has to be type variant
arraySplit = Split("a,b,c", ",")
'-> arraySplit(0) = "a", arraySplit(1) = "b", arraySplit(2) = "c"
```

Tableaux dynamiques (redimensionnement de matrice et traitement dynamique)

En raison de l'absence de contenu exclusif à Excel-VBA, cet exemple a été déplacé dans la documentation de VBA.

Lien: tableaux dynamiques (redimensionnement de tableaux et traitement dynamique)

Tableaux dentelés (tableaux de tableaux)

En raison de l'absence de contenu exclusif à Excel-VBA, cet exemple a été déplacé dans la

documentation de VBA.

Lien: tableaux déchiquetés (tableaux de tableaux)

Vérifiez si le tableau est initialisé (s'il contient des éléments ou non).

Un problème courant peut être d'essayer d'itérer sur Array qui ne contient aucune valeur. Par exemple:

```
Dim myArray() As Integer
For i = 0 To UBound(myArray) 'Will result in a "Subscript Out of Range" error
```

Pour éviter ce problème et vérifier si un tableau contient des éléments, utilisez cet *oneliner* :

```
If Not Not myArray Then MsgBox UBound(myArray) Else MsgBox "myArray not initialised"
```

Tableaux dynamiques [Déclaration de tableau, redimensionnement]

```
Sub Array_clarity()

Dim arr() As Variant  'creates an empty array
Dim x As Long
Dim y As Long

x = Range("A1", Range("A1").End(xlDown)).Cells.Count
y = Range("A1", Range("A1").End(xlToRight)).Cells.Count

ReDim arr(0 To x, 0 To y) 'fixing the size of the array

For x = LBound(arr, 1) To UBound(arr, 1)
    For y = LBound(arr, 2) To UBound(arr, 2)
        arr(x, y) = Range("A1").Offset(x, y) 'storing the value of Range("A1:E10") from
activesheet in x and y variables
    Next
Next

'Put it on the same sheet according to the declaration:
Range("A14").Resize(UBound(arr, 1), UBound(arr, 2)).Value = arr

End Sub
```

Lire Tableaux en ligne: https://riptutorial.com/fr/excel-vba/topic/2027/tableaux

Chapitre 27: Tableaux pivotants

Remarques

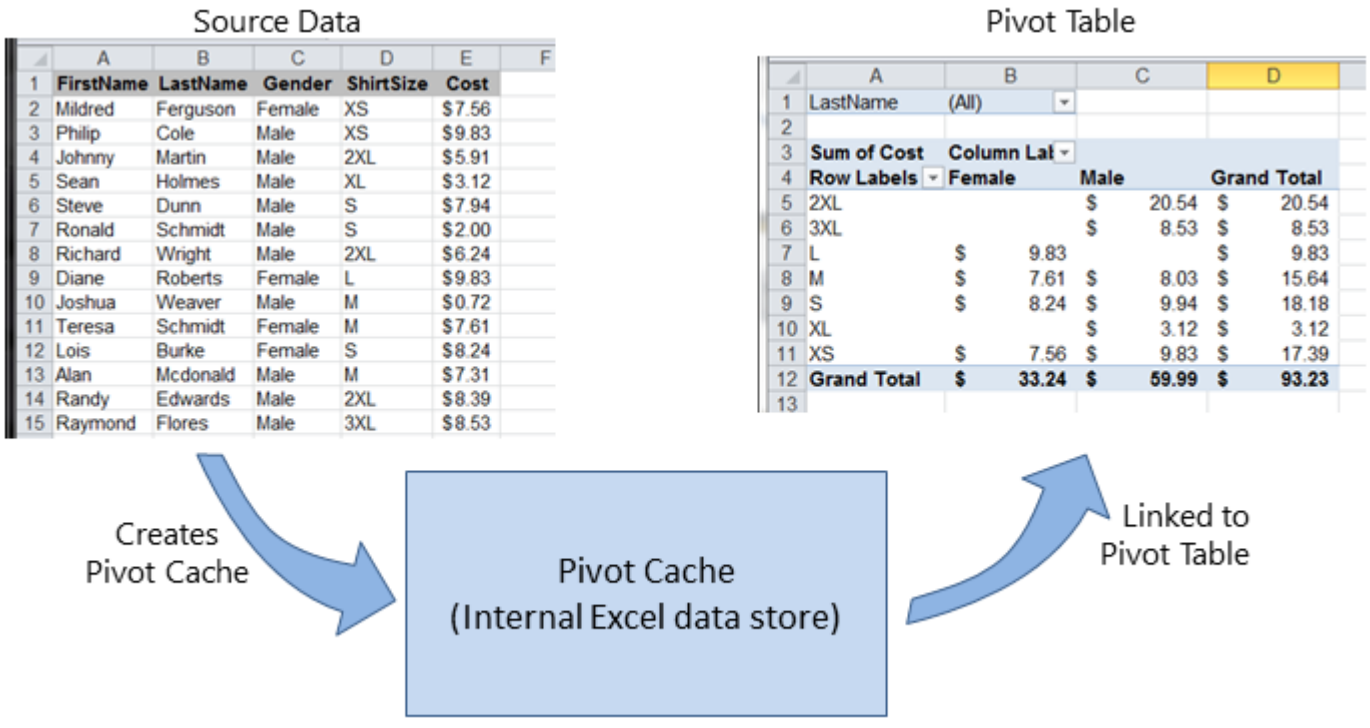
Il y a beaucoup d'excellentes sources de référence et d'exemples sur le Web. Quelques exemples et explications sont créés ici comme point de collecte pour des réponses rapides. Des illustrations plus détaillées peuvent être liées au contenu externe (au lieu de copier le matériel original existant).

Examples

Création d'un tableau croisé dynamique

L'une des fonctionnalités les plus puissantes d'Excel est l'utilisation des tableaux croisés dynamiques pour trier et analyser les données. Utiliser VBA pour créer et manipuler les pivots est plus facile si vous comprenez la relation entre les tableaux croisés dynamiques et les caches pivotants et comment référencer et utiliser les différentes parties des tableaux.

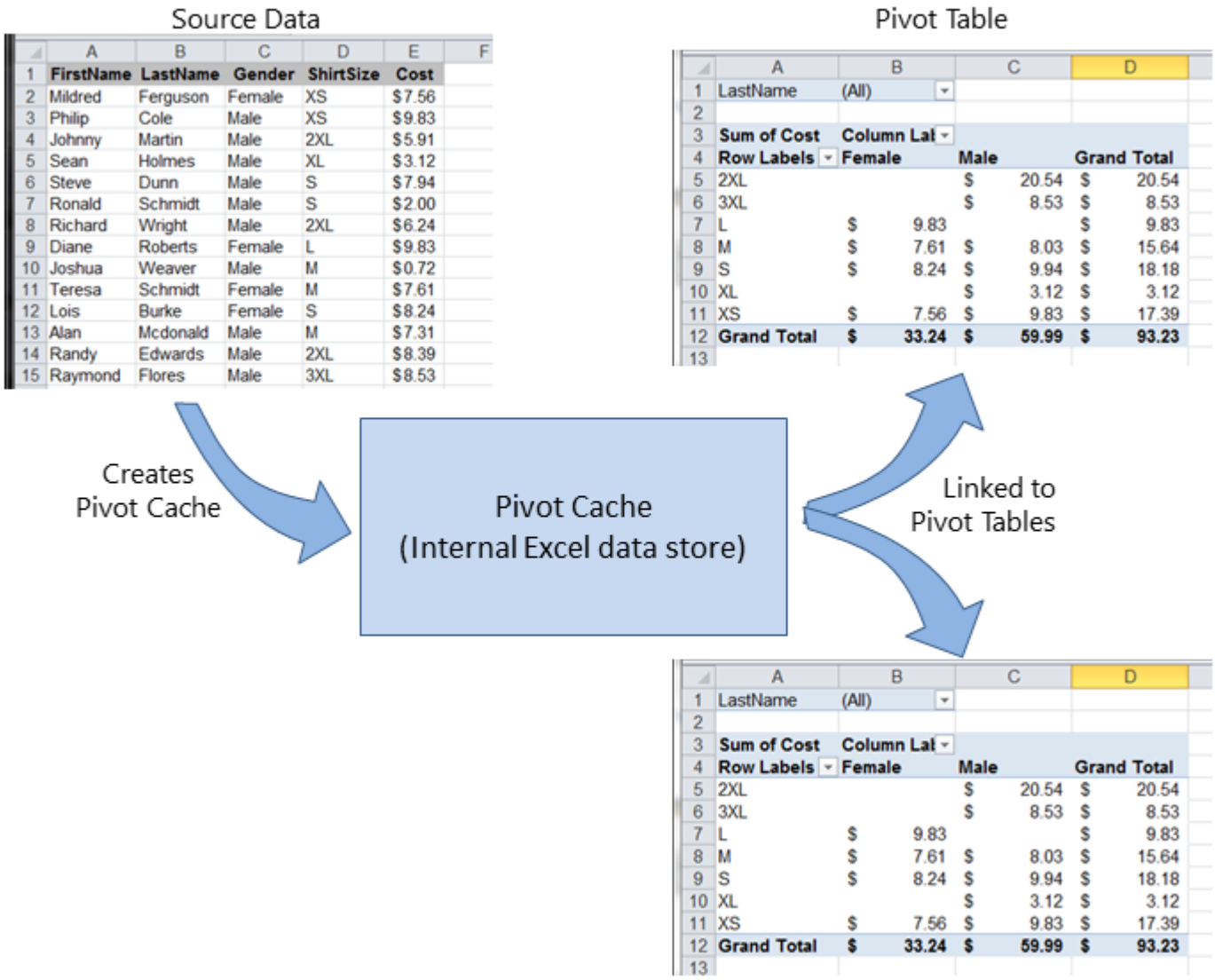
À la base, vos données sources sont une zone de données `Range` sur une `Worksheet` . Cette zone de données **DOIT** identifier les colonnes de données avec une ligne d'en-tête comme première ligne de la plage. Une fois le tableau croisé dynamique créé, l'utilisateur peut afficher et modifier les données source à tout moment. Cependant, les modifications peuvent ne pas être automatiquement ou immédiatement reflétées dans le tableau croisé dynamique lui-même car il existe une structure de stockage de données intermédiaire appelée cache de tableau croisé dynamique qui est directement connectée au tableau croisé dynamique lui-même.



	A	B	C	D	E	F
1	FirstName	LastName	Gender	ShirtSize	Cost	
2	Mildred	Ferguson	Female	XS	$7.56	
3	Philip	Cole	Male	XS	$9.83	
4	Johnny	Martin	Male	2XL	$5.91	
5	Sean	Holmes	Male	XL	$3.12	
6	Steve	Dunn	Male	S	$7.94	
7	Ronald	Schmidt	Male	S	$2.00	
8	Richard	Wright	Male	2XL	$6.24	
9	Diane	Roberts	Female	L	$9.83	
10	Joshua	Weaver	Male	M	$0.72	
11	Teresa	Schmidt	Female	M	$7.61	
12	Lois	Burke	Female	S	$8.24	
13	Alan	Mcdonald	Male	M	$7.31	
14	Randy	Edwards	Male	2XL	$8.39	
15	Raymond	Flores	Male	3XL	$8.53	

	A	B	C	D
1	LastName	(All)		
2				
3	Sum of Cost	Column Lal		
4	Row Labels	Female	Male	Grand Total
5	2XL		$ 20.54	$ 20.54
6	3XL		$ 8.53	$ 8.53
7	L	$ 9.83		$ 9.83
8	M	$ 7.61	$ 8.03	$ 15.64
9	S	$ 8.24	$ 9.94	$ 18.18
10	XL		$ 3.12	$ 3.12
11	XS	$ 7.56	$ 9.83	$ 17.39
12	Grand Total	$ 33.24	$ 59.99	$ 93.23
13				

Si plusieurs tableaux croisés dynamiques sont nécessaires, basés sur les mêmes données source, le cache pivot peut être réutilisé en tant que magasin de données interne pour chacun des tableaux croisés dynamiques. C'est une bonne pratique car cela économise de la mémoire et réduit la taille du fichier Excel pour le stockage.



Source Data

	A	B	C	D	E	F
1	**FirstName**	**LastName**	**Gender**	**ShirtSize**	**Cost**	
2	Mildred	Ferguson	Female	XS	$7.56	
3	Philip	Cole	Male	XS	$9.83	
4	Johnny	Martin	Male	2XL	$5.91	
5	Sean	Holmes	Male	XL	$3.12	
6	Steve	Dunn	Male	S	$7.94	
7	Ronald	Schmidt	Male	S	$2.00	
8	Richard	Wright	Male	2XL	$6.24	
9	Diane	Roberts	Female	L	$9.83	
10	Joshua	Weaver	Male	M	$0.72	
11	Teresa	Schmidt	Female	M	$7.61	
12	Lois	Burke	Female	S	$8.24	
13	Alan	Mcdonald	Male	M	$7.31	
14	Randy	Edwards	Male	2XL	$8.39	
15	Raymond	Flores	Male	3XL	$8.53	

Pivot Table

	A	B	C	D
1	LastName	(All)		
2				
3	**Sum of Cost**	**Column Lal**		
4	**Row Labels**	**Female**	**Male**	**Grand Total**
5	2XL		$ 20.54	$ 20.54
6	3XL		$ 8.53	$ 8.53
7	L	$ 9.83		$ 9.83
8	M	$ 7.61	$ 8.03	$ 15.64
9	S	$ 8.24	$ 9.94	$ 18.18
10	XL		$ 3.12	$ 3.12
11	XS	$ 7.56	$ 9.83	$ 17.39
12	**Grand Total**	**$ 33.24**	**$ 59.99**	**$ 93.23**
13				

Par exemple, pour créer un tableau croisé dynamique basé sur les données source indiquées dans les figures ci-dessus:

```
Sub test()
    Dim pt As PivotTable
    Set pt = CreatePivotTable(ThisWorkbook.Sheets("Sheet1").Range("A1:E15"))
End Sub

Function CreatePivotTable(ByRef srcData As Range) As PivotTable
    '--- creates a Pivot Table from the given source data and
    '    assumes that the first row contains valid header data
    '    for the columns
    Dim thisPivot As PivotTable
    Dim dataSheet As Worksheet
    Dim ptSheet As Worksheet
    Dim ptCache As PivotCache
```

```
    '--- the Pivot Cache must be created first...
    Set ptCache = ThisWorkbook.PivotCaches.Create(SourceType:=xlDatabase, _
                                                  SourceData:=srcData)
    '--- ... then use the Pivot Cache to create the Table
    Set ptSheet = ThisWorkbook.Sheets.Add
    Set thisPivot = ptCache.CreatePivotTable(TableDestination:=ptSheet.Range("A3"))
    Set CreatePivotTable = thisPivot
End Function
```

Références Objet tableau croisé dynamique MSDN

Plates-formes de tableau pivotant

Ces excellentes sources de référence fournissent des descriptions et des illustrations des différentes gammes de tableaux croisés dynamiques.

Les références

- Référencement des plages de tableau croisé dynamique dans VBA - sur le blog Tech de Jon Peltier
- Référencement d'une plage de tableau croisé dynamique Excel à l' aide de VBA - à partir de globaliconnect Excel VBA

Ajout de champs à un tableau croisé dynamique

Deux points importants à noter lors de l'ajout de champs à un tableau croisé dynamique sont l'orientation et la position. Parfois, un développeur peut supposer où un champ est placé, il est donc toujours plus clair de définir explicitement ces paramètres. Ces actions n'affectent que le tableau croisé dynamique donné, pas le cache de tableau croisé dynamique.

```
Dim thisPivot As PivotTable
Dim ptSheet As Worksheet
Dim ptField As PivotField

Set ptSheet = ThisWorkbook.Sheets("SheetNameWithPivotTable")
Set thisPivot = ptSheet.PivotTables(1)

With thisPivot
    Set ptField = .PivotFields("Gender")
    ptField.Orientation = xlRowField
    ptField.Position = 1
    Set ptField = .PivotFields("LastName")
    ptField.Orientation = xlRowField
    ptField.Position = 2
    Set ptField = .PivotFields("ShirtSize")
    ptField.Orientation = xlColumnField
    ptField.Position = 1
    Set ptField = .AddDataField(.PivotFields("Cost"), "Sum of Cost", xlSum)
    .InGridDropZones = True
    .RowAxisLayout xlTabularRow
End With
```

Formatage des données du tableau croisé dynamique

Cet exemple modifie / définit plusieurs formats dans la zone de plage de données (`DataBodyRange`) du tableau `DataBodyRange` donné. Tous les paramètres formatables dans une `Range` standard sont disponibles. Le formatage des données affecte uniquement le tableau croisé dynamique lui-même, pas le cache de tableau croisé dynamique.

REMARQUE: la propriété est nommée `TableStyle2` car la propriété `TableStyle` n'est pas membre des propriétés de l'objet du `PivotTable` .

```
Dim thisPivot As PivotTable
Dim ptSheet As Worksheet
Dim ptField As PivotField

Set ptSheet = ThisWorkbook.Sheets("SheetNameWithPivotTable")
Set thisPivot = ptSheet.PivotTables(1)

With thisPivot
    .DataBodyRange.NumberFormat = "_($* #,##0.00_);_($* (#,##0.00);_($* "-"??_);_(@_)"
    .DataBodyRange.HorizontalAlignment = xlRight
    .ColumnRange.HorizontalAlignment = xlCenter
    .TableStyle2 = "PivotStyleMedium9"
End With
```

Lire Tableaux pivotants en ligne: https://riptutorial.com/fr/excel-vba/topic/3797/tableaux-pivotants

Chapitre 28: Travailler avec des tableaux Excel dans VBA

Introduction

Cette rubrique traite de l'utilisation de tables dans VBA et suppose une connaissance des tableaux Excel. Dans VBA, ou plutôt le modèle d'objet Excel, les tables sont appelées ListObjects. Les propriétés les plus fréquemment utilisées d'un objet ListObject sont ListRow (s), ListColumn (s), DataBodyRange, Range et HeaderRowRange.

Examples

Instancier un objet ListObject

```
Dim lo as ListObject
Dim MyRange as Range

Set lo = Sheet1.ListObjects(1)

'or

Set lo = Sheet1.ListObjects("Table1")

'or

Set lo = MyRange.ListObject
```

Travailler avec ListRows / ListColumns

```
Dim lo as ListObject
Dim lr as ListRow
Dim lc as ListColumn

Set lr = lo.ListRows.Add
Set lr = lo.ListRows(5)

For Each lr in lo.ListRows
    lr.Range.ClearContents
    lr.Range(1, lo.ListColumns("Some Column").Index).Value = 8
Next

Set lc = lo.ListColumns.Add
Set lc = lo.ListColumns(4)
Set lc = lo.ListColumns("Header 3")

For Each lc in lo.ListColumns
    lc.DataBodyRange.ClearContents   'DataBodyRange excludes the header row
    lc.Range(1,1).Value = "New Header Name"    'Range includes the header row
Next
```

Conversion d'une table Excel en une plage normale

```
Dim lo as ListObject

Set lo = Sheet1.ListObjects("Table1")
lo.Unlist
```

Lire Travailler avec des tableaux Excel dans VBA en ligne: https://riptutorial.com/fr/excel-vba/topic/9753/travailler-avec-des-tableaux-excel-dans-vba

Chapitre 29: Traverser toutes les feuilles dans Active Workbook

Examples

Récupérer tous les noms de feuilles de calcul dans Active Workbook

```
Option Explicit

Sub LoopAllSheets()

Dim sht As Excel.Worksheet
' declare an array of type String without committing to maximum number of members
Dim sht_Name() As String
Dim i As Integer

' get the number of worksheets in Active  Workbook , and put it as the maximum number of
members in the array
ReDim sht_Name(1 To ActiveWorkbook.Worksheets.count)

i = 1

' loop through all worksheets in Active Workbook
For Each sht In ActiveWorkbook.Worksheets
    sht_Name(i) = sht.Name ' get the name of each worksheet and save it in the array
    i = i + 1
Next sht

End Sub
```

Boucler toutes les feuilles de tous les fichiers d'un dossier

```
 Sub Theloopofloops()

 Dim wbk As Workbook
 Dim Filename As String
 Dim path As String
 Dim rCell As Range
 Dim rRng As Range
 Dim wsO As Worksheet
 Dim sheet As Worksheet


 path = "pathtofile(s)" & "\"
 Filename = Dir(path & "*.xl??")
 Set wsO = ThisWorkbook.Sheets("Sheet1") 'included in case you need to differentiate_
             between workbooks i.e currently opened workbook vs workbook containing code

 Do While Len(Filename) > 0
     DoEvents
     Set wbk = Workbooks.Open(path & Filename, True, True)
         For Each sheet In ActiveWorkbook.Worksheets  'this needs to be adjusted for
specifiying sheets. Repeat loop for each sheet so thats on a per sheet basis
```

```
                Set rRng = sheet.Range("a1:a1000") 'OBV needs to be changed
                For Each rCell In rRng.Cells
                If rCell <> "" And rCell.Value <> vbNullString And rCell.Value <> 0 Then

                   'code that does stuff

                End If
                Next rCell
        Next sheet
    wbk.Close False
    Filename = Dir
Loop
End Sub
```

Lire Traverser toutes les feuilles dans Active Workbook en ligne: https://riptutorial.com/fr/excel-vba/topic/1144/traverser-toutes-les-feuilles-dans-active-workbook

Chapitre 30: Trucs et astuces Excel VBA

Remarques

Cette rubrique comprend une grande variété de trucs et astuces utiles découverts par les utilisateurs SO grâce à leur expérience en matière de codage. Ce sont souvent des exemples de moyens permettant de contourner les frustrations ou les façons d'utiliser Excel de manière plus "intelligente".

Examples

Utiliser les feuilles xlVeryHidden

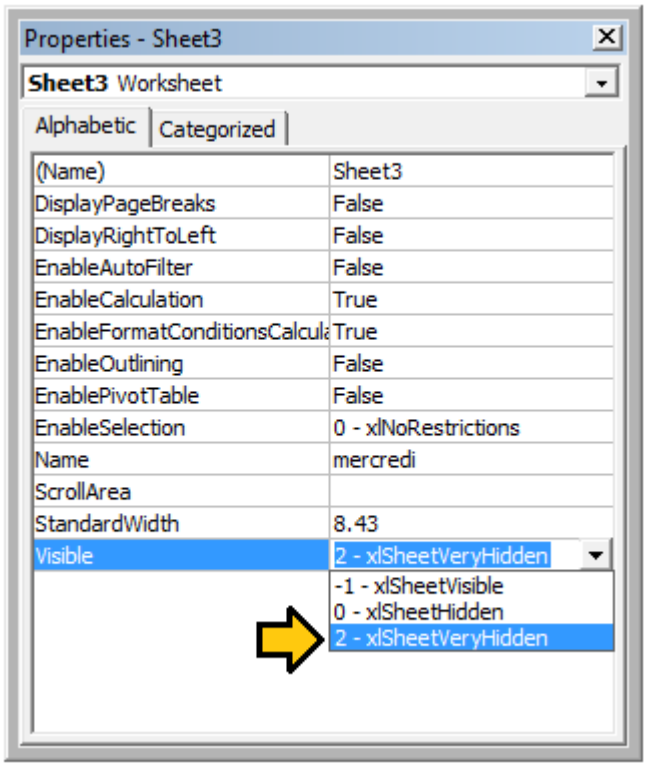
Les feuilles de calcul dans Excel ont trois options pour la propriété `Visible` . Ces options sont représentées par des constantes dans l'énumération `xlSheetVisibility` et sont les suivantes:

1. `xlVisible` ou `xlSheetVisible` : `-1` (valeur par défaut pour les nouvelles feuilles)
2. `xlHidden` ou `xlSheetHidden` : `0`
3. `xlVeryHidden` `xlSheetVeryHidden` : `2`

Les feuilles visibles représentent la visibilité par défaut des feuilles. Ils sont visibles dans la barre des onglets et peuvent être librement sélectionnés et visualisés. Les feuilles masquées sont masquées dans la barre des onglets et ne peuvent donc pas être sélectionnées. Cependant, les feuilles masquées peuvent être masquées depuis la fenêtre Excel en cliquant avec le bouton droit de la souris sur les onglets de la feuille et en sélectionnant "Afficher".

Les feuilles très cachées, par contre, *ne* sont accessibles que via Visual Basic Editor. Cela en fait un outil extrêmement utile pour stocker des données sur des instances d'Excel et stocker des données qui doivent être masquées pour les utilisateurs finaux. Les feuilles peuvent être consultées par référence nommée dans le code VBA, ce qui permet une utilisation facile des données stockées.

Pour modifier manuellement la propriété .Visible d'une feuille de calcul en xlSheetVeryHidden, ouvrez la fenêtre Propriétés de VBE (`F4`), sélectionnez la feuille de calcul à modifier et utilisez la liste déroulante de la treizième ligne pour effectuer votre sélection.



Pour modifier la propriété .Visible d'une feuille de calcul en xlSheetVeryHidden[1] dans le code, accédez de la même manière à la propriété .Visible et affectez une nouvelle valeur.

```
with Sheet3
    .Visible = xlSheetVeryHidden
end with
```

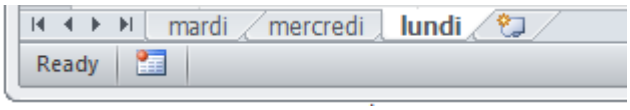
---

[1] **xlVeryHidden** et **xlSheetVeryHidden** renvoient une valeur numérique de **2** (elles sont interchangeables).

Feuille de calcul .Name, .Index ou .CodeName

Nous savons que la «meilleure pratique» dicte qu'un objet de plage doit avoir sa feuille de travail parente explicitement référencée. Une feuille de calcul peut être désignée par sa propriété .Name, sa propriété .Index numérique ou sa propriété .CodeName, mais un utilisateur peut réorganiser la file d'attente de la feuille de calcul en faisant simplement glisser un onglet de nom ou en double-cliquant sur le même onglet et taper dans un classeur non protégé.

Considérons une feuille de travail standard trois. Vous avez renommé les trois feuilles de calcul lundi, mardi et mercredi dans cet ordre et codé les sous-procédures VBA qui font référence à celles-ci. Considérez maintenant qu'un utilisateur arrive et décide que le lundi appartient à la fin de la file d'attente de la feuille de calcul, tandis qu'un autre intervient et décide que les noms des feuilles de calcul ont l'air mieux en français. Vous avez maintenant un classeur avec une file d'attente d'onglets de nom de feuille de calcul qui ressemble à ce qui suit.



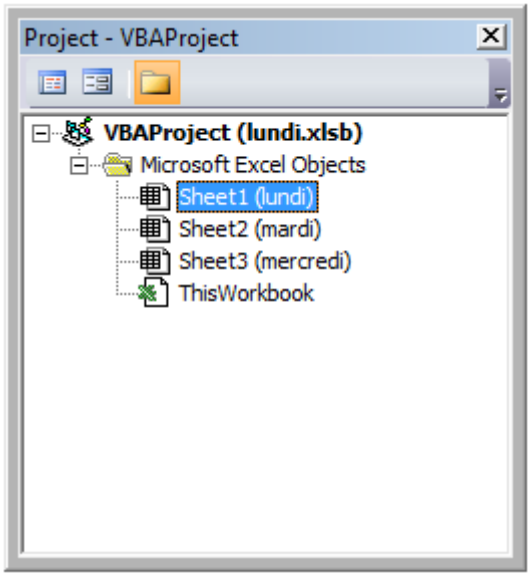
Si vous aviez utilisé l'une des méthodes de référence de feuille de calcul suivantes, votre code serait désormais rompu.

```
'reference worksheet by .Name
with worksheets("Monday")
    'operation code here; for example:
    .Range(.Cells(2, "A"), .Cells(.Rows.Count, "A").End(xlUp)) = 1
end with

'reference worksheet by ordinal .Index
with worksheets(1)
    'operation code here; for example:
    .Range(.Cells(2, "A"), .Cells(.Rows.Count, "A").End(xlUp)) = 1
end with
```

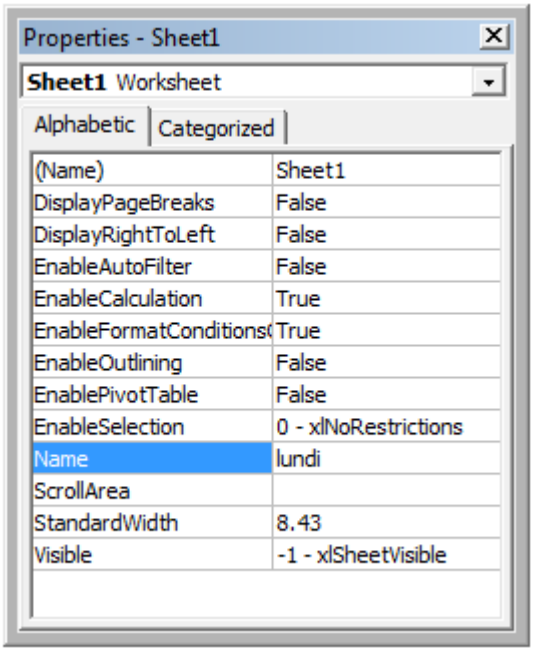
L'ordre d'origine et le nom de la feuille de calcul d'origine ont été compromis. Toutefois, si vous aviez utilisé la propriété .CodeName de la feuille de calcul, votre sous-procédure serait toujours opérationnelle

```
with Sheet1
    'operation code here; for example:
    .Range(.Cells(2, "A"), .Cells(.Rows.Count, "A").End(xlUp)) = 1
end with
```

L'image suivante montre la fenêtre Projet VBA ([Ctrl] + R) qui répertorie les feuilles de calcul par .CodeName puis par .Name (entre parenthèses). L'ordre dans lequel ils sont affichés ne change pas; l'ordinal .Index est pris par l'ordre dans lequel ils sont affichés dans la file d'attente de l'onglet Nom dans la fenêtre de la feuille de calcul.



Bien qu'il soit rare de renommer un nom de code, ce n'est pas impossible. Ouvrez simplement la fenêtre Propriétés de VBE ([F4]).



La feuille de calcul .CodeName est dans la première ligne. La feuille de calcul .Name est dans le dixième. Les deux sont modifiables.

Utilisation de chaînes avec des délimiteurs à la place des tableaux dynamiques

L'utilisation de tableaux dynamiques dans VBA peut s'avérer très fastidieuse et prendre du temps sur des ensembles de données très volumineux. Lorsque vous stockez des types de données simples dans un tableau dynamique (chaînes, nombres, booléens, etc.), vous pouvez éviter les `ReDim Preserve` requises pour les tableaux dynamiques dans VBA en utilisant la fonction `Split()` avec certaines procédures de chaîne intelligentes. Par exemple, nous allons examiner une boucle qui ajoute une série de valeurs d'une plage à une chaîne basée sur certaines conditions, puis utilise cette chaîne pour renseigner les valeurs d'un ListBox.

```
Private Sub UserForm_Initialize()

Dim Count As Long, DataString As String, Delimiter As String

For Count = 1 To ActiveSheet.UsedRows.Count
    If ActiveSheet.Range("A" & Count).Value <> "Your Condition" Then
        RowString = RowString & Delimiter & ActiveSheet.Range("A" & Count).Value
        Delimiter = "><" 'By setting the delimiter here in the loop, you prevent an extra
occurance of the delimiter within the string
    End If
Next Count

ListBox1.List = Split(DataString, Delimiter)

End Sub
```

La chaîne `Delimiter` elle-même peut être définie sur n'importe quelle valeur, mais il est prudent de choisir une valeur qui ne se produira pas naturellement dans l'ensemble. Disons, par exemple, que vous traitiez une colonne de dates. Dans ce cas, utiliser `.` , `-` , ou `/` serait imprudent en tant que délimiteurs, car les dates pourraient être formatées pour utiliser l'une quelconque de celles-ci,

générant plus de points de données que prévu.

Remarque: L' utilisation de cette méthode (à savoir la longueur maximale des chaînes) est limitée, elle doit donc être utilisée avec précaution dans le cas de jeux de données très volumineux. Ce n'est pas nécessairement la méthode la plus rapide ou la plus efficace pour créer des tableaux dynamiques dans VBA, mais c'est une alternative viable.

Evénement Double Click pour les formes Excel

Par défaut, les formes dans Excel n'ont pas de moyen spécifique pour gérer les clics simples et doubles, contenant uniquement la propriété "OnAction" pour vous permettre de gérer les clics. Cependant, il peut arriver que votre code vous oblige à agir différemment (ou exclusivement) en double-cliquant. Le sous-programme suivant peut être ajouté à votre projet VBA et, lorsqu'il est défini comme routine `OnAction` pour votre forme, vous permet d'agir sur les clics doubles.

```
Public Const DOUBLECLICK_WAIT as Double = 0.25 'Modify to adjust click delay
Public LastClickObj As String, LastClickTime As Date

Sub ShapeDoubleClick()

    If LastClickObj = "" Then
        LastClickObj = Application.Caller
        LastClickTime = CDbl(Timer)
    Else
        If CDbl(Timer) - LastClickTime > DOUBLECLICK_WAIT Then
            LastClickObj = Application.Caller
            LastClickTime = CDbl(Timer)
        Else
            If LastClickObj = Application.Caller Then
                'Your desired Double Click code here
                LastClickObj = ""
            Else
                LastClickObj = Application.Caller
                LastClickTime = CDbl(Timer)
            End If
        End If
    End If

End Sub
```

Cette routine fera que la forme ignorera fonctionnellement le premier clic, exécutant uniquement le code souhaité sur le deuxième clic dans la période spécifiée.

Boîte de dialogue Ouvrir un fichier - Fichiers multiples

Ce sous-programme est un exemple rapide sur la manière de permettre à un utilisateur de sélectionner plusieurs fichiers, puis de faire quelque chose avec ces chemins, par exemple obtenir les noms de fichiers et les envoyer à la console via debug.print.

```
Option Explicit

Sub OpenMultipleFiles()
    Dim fd As FileDialog
```

```
    Dim fileChosen As Integer
    Dim i As Integer
    Dim basename As String
    Dim fso As Variant
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set fd = Application.FileDialog(msoFileDialogFilePicker)
    basename = fso.getBaseName(ActiveWorkbook.Name)
    fd.InitialFileName = ActiveWorkbook.Path ' Set Default Location to the Active Workbook
Path
    fd.InitialView = msoFileDialogViewList
    fd.AllowMultiSelect = True

    fileChosen = fd.Show
    If fileChosen = -1 Then
        'open each of the files chosen
        For i = 1 To fd.SelectedItems.Count
            Debug.Print (fd.SelectedItems(i))
            Dim fileName As String
            ' do something with the files.
            fileName = fso.getFileName(fd.SelectedItems(i))
            Debug.Print (fileName)
        Next i
    End If

End Sub
```

Lire Trucs et astuces Excel VBA en ligne: https://riptutorial.com/fr/excel-vba/topic/2240/trucs-et-astuces-excel-vba

Chapitre 31: Utiliser un objet Feuille de calcul et non un objet Feuille

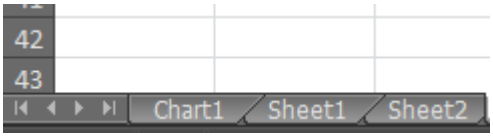
Introduction

De nombreux utilisateurs de VBA considèrent les objets Worksheets et Sheets des synonymes. Ils ne sont pas.

L'objet Sheets se compose de feuilles de calcul et de graphiques. Ainsi, si nous avons des tableaux dans notre classeur Excel, nous devrions faire attention à ne pas utiliser les `Sheets` et les `Worksheets` de `Worksheets` comme synonymes.

Examples

Imprimer le nom du premier objet




```
Option Explicit

Sub CheckWorksheetsDiagram()

    Debug.Print Worksheets(1).Name
    Debug.Print Charts(1).Name
    Debug.Print Sheets(1).Name

End Sub
```

Le résultat:

```
Sheet1
Chart1
Chart1
```

Lire Utiliser un objet Feuille de calcul et non un objet Feuille en ligne: https://riptutorial.com/fr/excel-vba/topic/9996/utiliser-un-objet-feuille-de-calcul-et-non-un-objet-feuille

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec excel-vba	Branislav Kollár, chris neilsen, Cody G., Comintern, Community, Doug Coats, EEM, Gordon Bell, Jeeped, Joel Spolsky, Kaz, Laurel, LucyMarieJ, Macro Man, Malick, Maxime Porté, Regis, RGA, Ron McMahon, SandPiper, Shai Rado, Taylor Ostberg, whytheq
2	Cellules / plages fusionnées	R3uK
3	Classeurs	PeterT
4	Comment enregistrer une macro	Mike, Robby
5	Contraignant	Captain Grumpy, EEM, Jeeped, jlookup, Malick, Raystafarian
6	Création d'un menu déroulant dans la feuille de travail active avec une zone de liste déroulante	Macro Man, quadrature, R3uK
7	CustomDocumentProperties dans la pratique	T.M.
8	Débogage et dépannage	Cody G., Etheur, Gregor y, Julian Kuchlbauer, Kyle, Malick, Michael Russo, RGA, Ron McMahon, Slai, Steven Schroeder, Taylor Ostberg
9	Erreurs courantes	Egan Wolf, Gordon Bell, Macro Man, Malick, Peh, SWa, Taylor Ostberg
10	Expressions conditionnelles	SteveES
11	filtre automatique; Utilisations et meilleures pratiques	Sgdva
12	Fonctions définies par l'utilisateur (UDF)	Jeeped, Malick, Slai, user3561813, Vegard
13	Gammes et cellules	Adam, Branislav Kollár, Doug Coats, Gregor y, Jbjstam, Joel Spolsky, Julian Kuchlbauer, Máté Juhász,

		Miguel_Ryu, Patrick Wynne, Vegard
14	Gammes Nommées	Andre Terra, Portland Runner
15	Graphiques et graphiques	Byron Wall
16	Intégration PowerPoint via VBA	mnoronha, RGA
17	Localisation des valeurs en double dans une plage	quadrature, T.M.
18	Meilleures pratiques de VBA	Alexis Olson, Branislav Kollár, Chel, Cody G., Comintern, EEM, FreeMan, genespos, Hubisan, Huzaifa Essajee, Jeeped, JKAbrams, Kumar Sourav, Kyle, Macro Man, Malick, Máté Juhász, Munkeeface, paul bica, Peh, PeterT, Portland Runner, RGA, Shai Rado, Stefan Pinnow, Steven Schroeder, Taylor Ostberg, ThunderFrame, Verzweifler, Vityata
19	Méthodes de recherche de la dernière ligne ou colonne utilisée dans une feuille de calcul	curious, Hubisan, Máté Juhász, Michael Russo, Miqi180, paul bica, R3uK, Raystafarian, RGA, Shai Rado, Slai, Thomas Inzina, YowE3K
20	Mise en forme conditionnelle à l'aide de VBA	Zsmaster
21	Objet d'application	Captain Grumpy, Joel Spolsky
22	Objet du système de fichiers	Zsmaster
23	Optimisation Excel-VBA	Masoud, paul bica, T.M.
24	Sécurité VBA	Chel, TheGuyThatDoesn'tKnowMuch
25	SQL dans Excel VBA - Meilleures pratiques	Zsmaster
26	Tableaux	Alon Adler, Hubisan, Miguel_Ryu, Shahin
27	Tableaux pivotants	PeterT
28	Travailler avec des tableaux Excel dans VBA	Excel Developers
29	Traverser toutes les feuilles dans Active Workbook	Doug Coats, Shai Rado

30	Trucs et astuces Excel VBA	Andre Terra, Cody G., Jeeped, Kumar Sourav, Macro Man, RGA
31	Utiliser un objet Feuille de calcul et non un objet Feuille	Vityata