EBook Gratis

APRENDIZAJE

excel-vba

Free unaffiliated eBook created from
Stack Overflow contributors.

#excel-vba

Tabla de contenido

Acerca de

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

Capítulo 1: Empezando con excel-vba

Observaciones

Microsoft Excel incluye un completo lenguaje de programación de macros llamado VBA. Este lenguaje de programación le proporciona al menos tres recursos adicionales:

1. Conducir automáticamente Excel desde el código utilizando macros. En su mayor parte, cualquier cosa que el usuario pueda hacer manipulando Excel desde la interfaz de usuario puede hacerlo escribiendo código en Excel VBA.
2. Crear nuevas funciones de hojas de trabajo personalizadas.
3. Interactúa con otras aplicaciones como Microsoft Word, PowerPoint, Internet Explorer, Bloc de notas, etc.

VBA significa Visual Basic para aplicaciones. Es una versión personalizada del venerable lenguaje de programación Visual Basic que ha impulsado las macros de Microsoft Excel desde mediados de los años noventa.

IMPORTANTE
Asegúrese de que los ejemplos o temas creados dentro de la etiqueta excel-vba sean **específicos** y **relevantes** para el uso de VBA con Microsoft Excel. Cualquier tema sugerido o ejemplos provistos que sean genéricos para el lenguaje de VBA deben ser rechazados para evitar la duplicación de esfuerzos.

- ejemplos sobre el tema:

  ✓ Crear e interactuar con objetos de la hoja de trabajo.
  ✓ La clase `WorksheetFunction` y los métodos respectivos.
  ✓ Usar la enumeración `xlDirection` para navegar por un rango

- ejemplos fuera de tema:

  ✗ Cómo crear un bucle 'para cada'
  Class Clase `MsgBox` y cómo mostrar un mensaje
  ✗ Usando WinAPI en VBA

Versiones

VB

Versión	Fecha de lanzamiento
VB6	1998-10-01

Versión	Fecha de lanzamiento
VB7	2001-06-06
WIN32	1998-10-01
WIN64	2001-06-06
MAC	1998-10-01

Sobresalir

Versión	Fecha de lanzamiento
dieciséis	2016-01-01
15	2013-01-01
14	2010-01-01
12	2007-01-01
11	2003-01-01
10	2001-01-01
9	1999-01-01
8	1997-01-01
7	1995-01-01
5	1993-01-01
2	1987-01-01

Examples

Declarando variables

Para declarar explícitamente las variables en VBA, use la declaración `Dim` , seguida del nombre y tipo de la variable. Si se utiliza una variable sin ser declarada, o si no se especifica ningún tipo, se le asignará el tipo `Variant` .

Use la declaración `Option Explicit` en la primera línea de un módulo para forzar que todas las variables se declaren antes de su uso (vea SIEMPRE use "Option Explicit").

Siempre se recomienda utilizar `Option Explicit` porque ayuda a prevenir errores tipográficos /

ortográficos y garantiza que las variables / objetos sigan siendo su tipo deseado.

```
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
```

Se pueden declarar múltiples variables en una sola línea usando comas como delimitadores, pero **cada tipo debe declararse individualmente** , o se establecerán de forma predeterminada al tipo `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
```

Las variables también se pueden declarar utilizando los sufijos de caracteres del tipo de datos ($% &! # @), Sin embargo, su uso está cada vez más desaconsejado.

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

Otras formas de declarar variables son:

- `Static` como: `Static CounterVariable as Integer`

  Cuando utiliza la instrucción Estática en lugar de una instrucción Dim, la variable declarada conservará su valor entre las llamadas.

- `Public` como: `Public CounterVariable as Integer`

  Las variables públicas se pueden utilizar en cualquier procedimiento en el proyecto. Si una variable pública se declara en un módulo estándar o un módulo de clase, también se puede utilizar en cualquier proyecto que haga referencia al proyecto donde se

declara la variable pública.

- `Private` como: `Private CounterVariable as Integer`

  Las variables privadas solo pueden ser utilizadas por procedimientos en el mismo módulo.

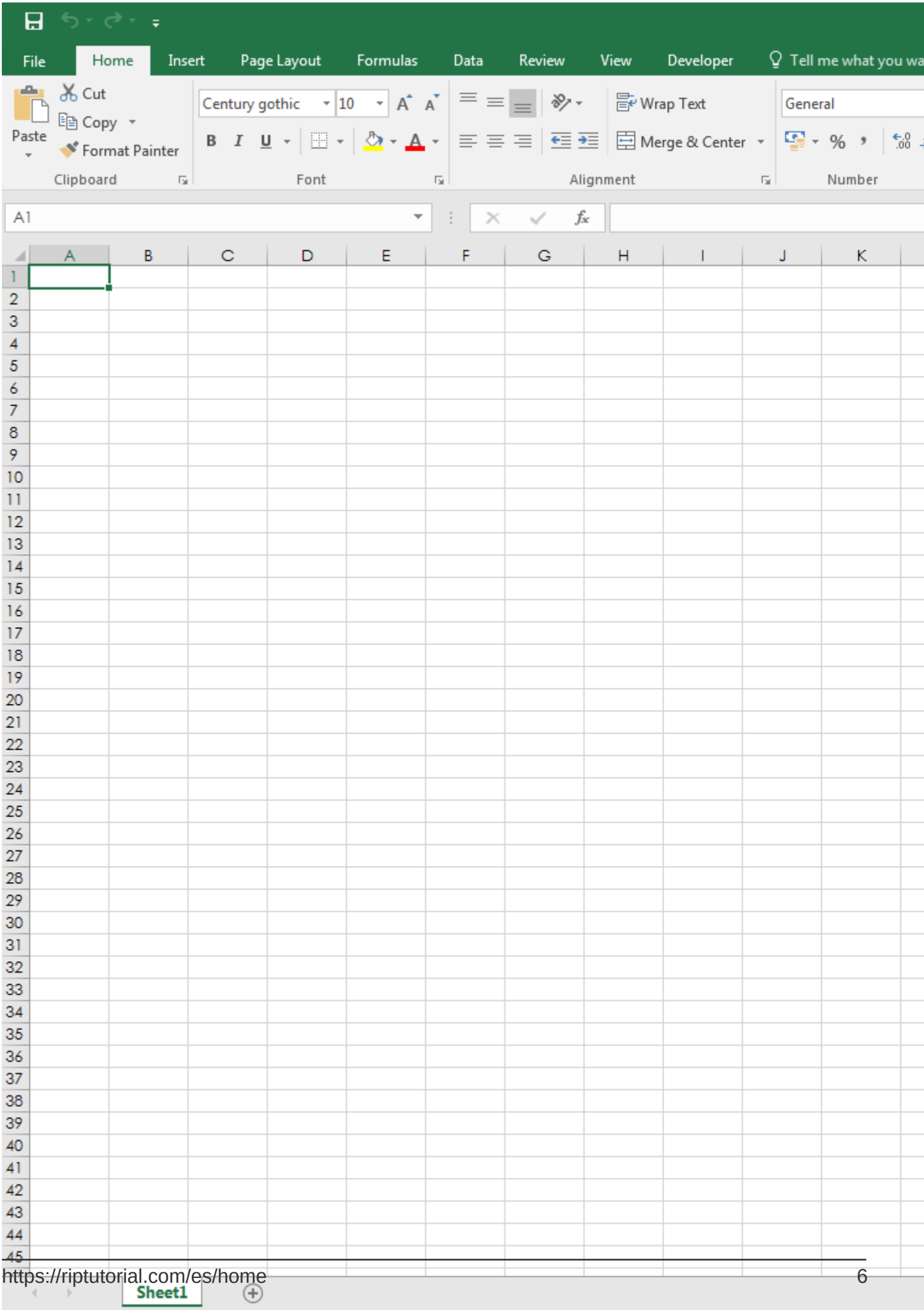
Fuente y más información:

Variables de declaración de MSDN

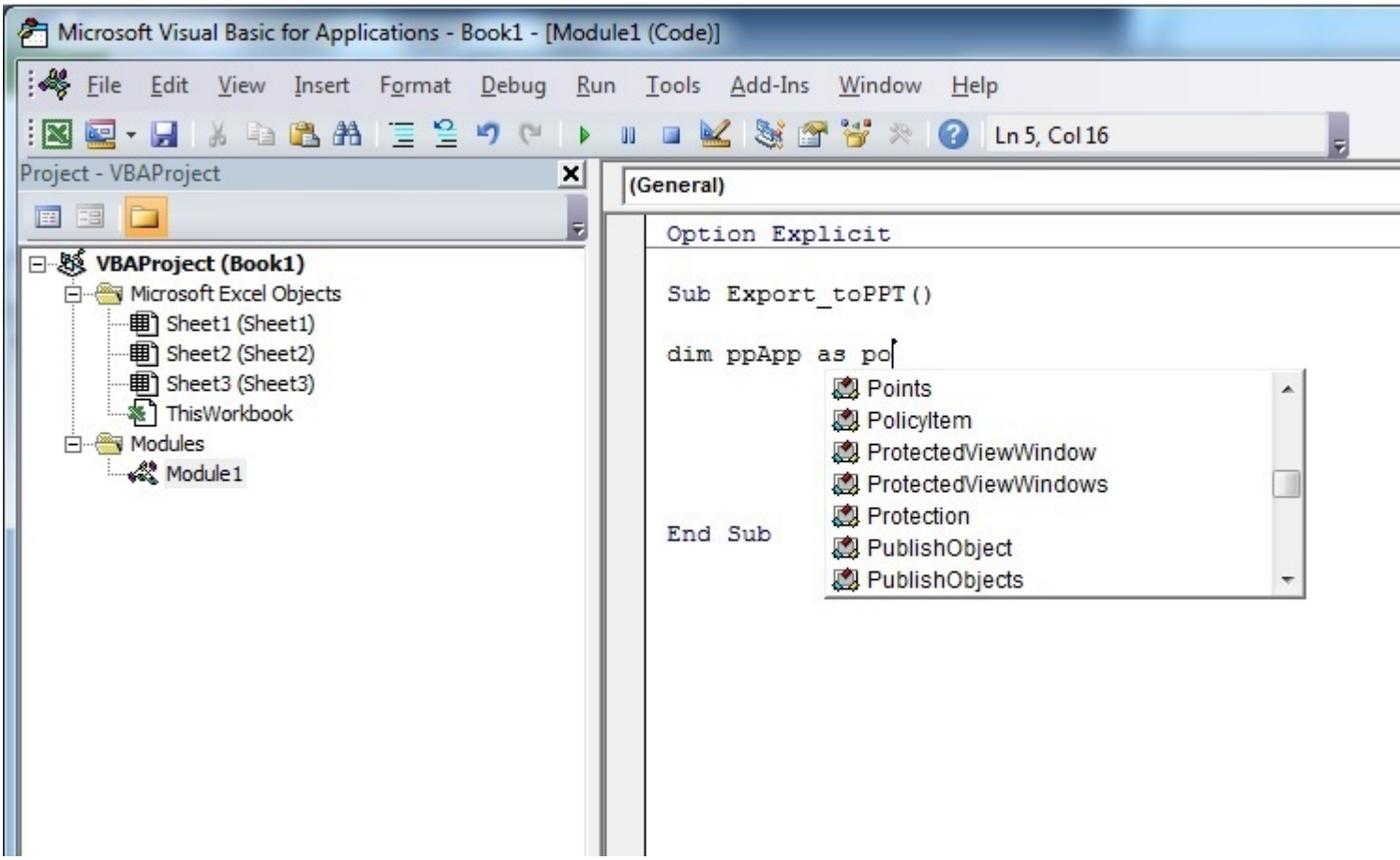
Tipo de caracteres (Visual Basic)

Abriendo el Editor de Visual Basic (VBE)

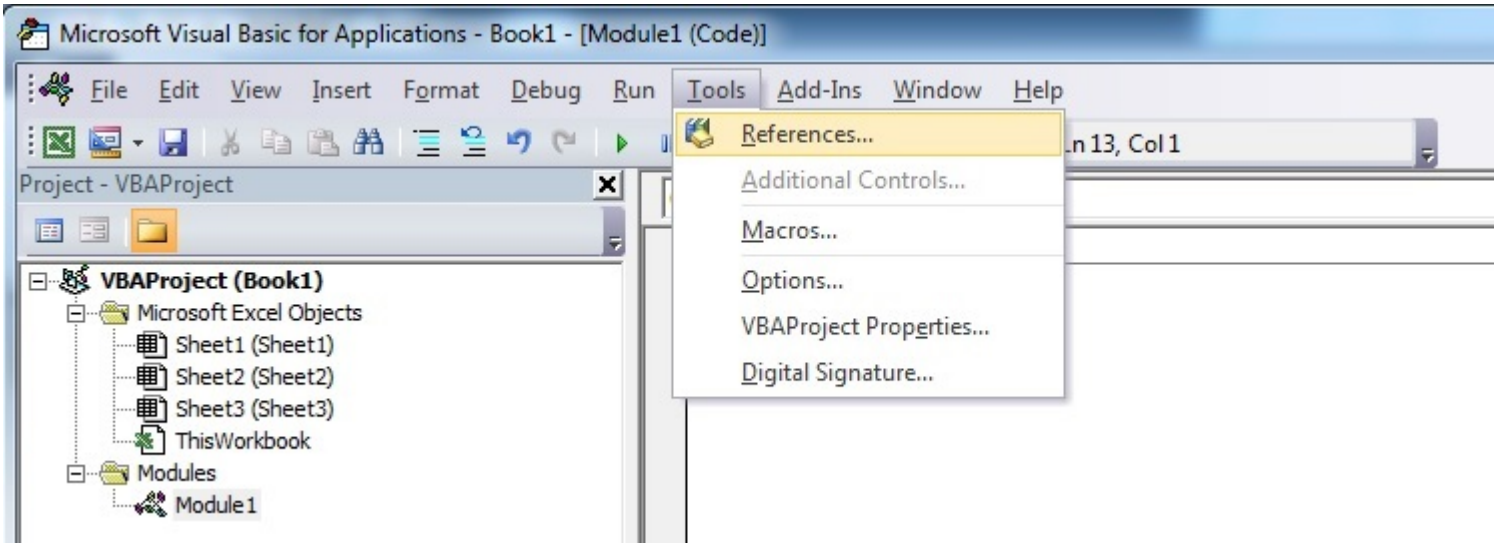
Paso 1: abrir un libro de trabajo


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

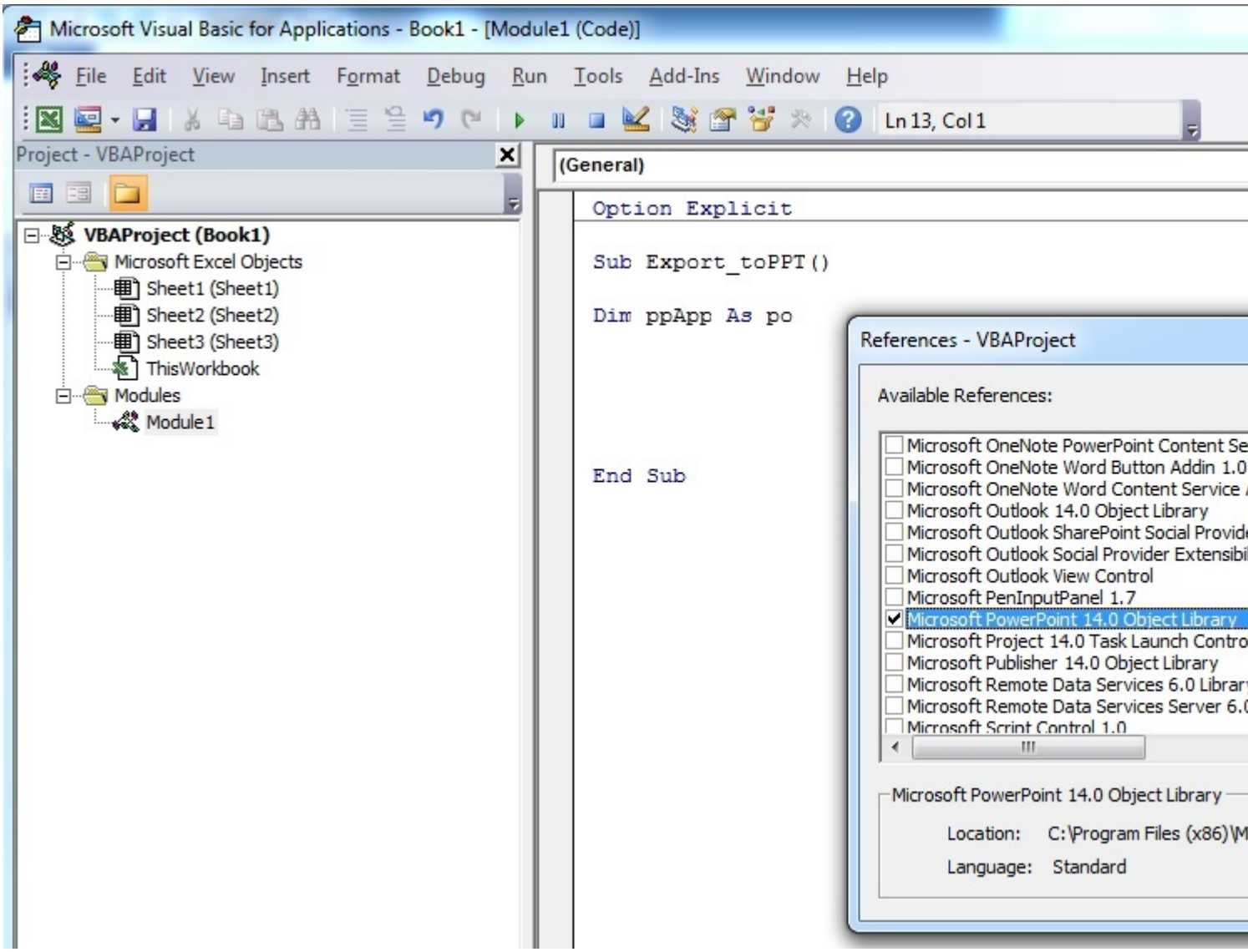
al proyecto VB existente. Como se puede ver, actualmente la biblioteca de objetos de PowerPoint no está disponible.



Paso 1 : Seleccione ***Herramientas de*** menú ***-> Referencias ...***

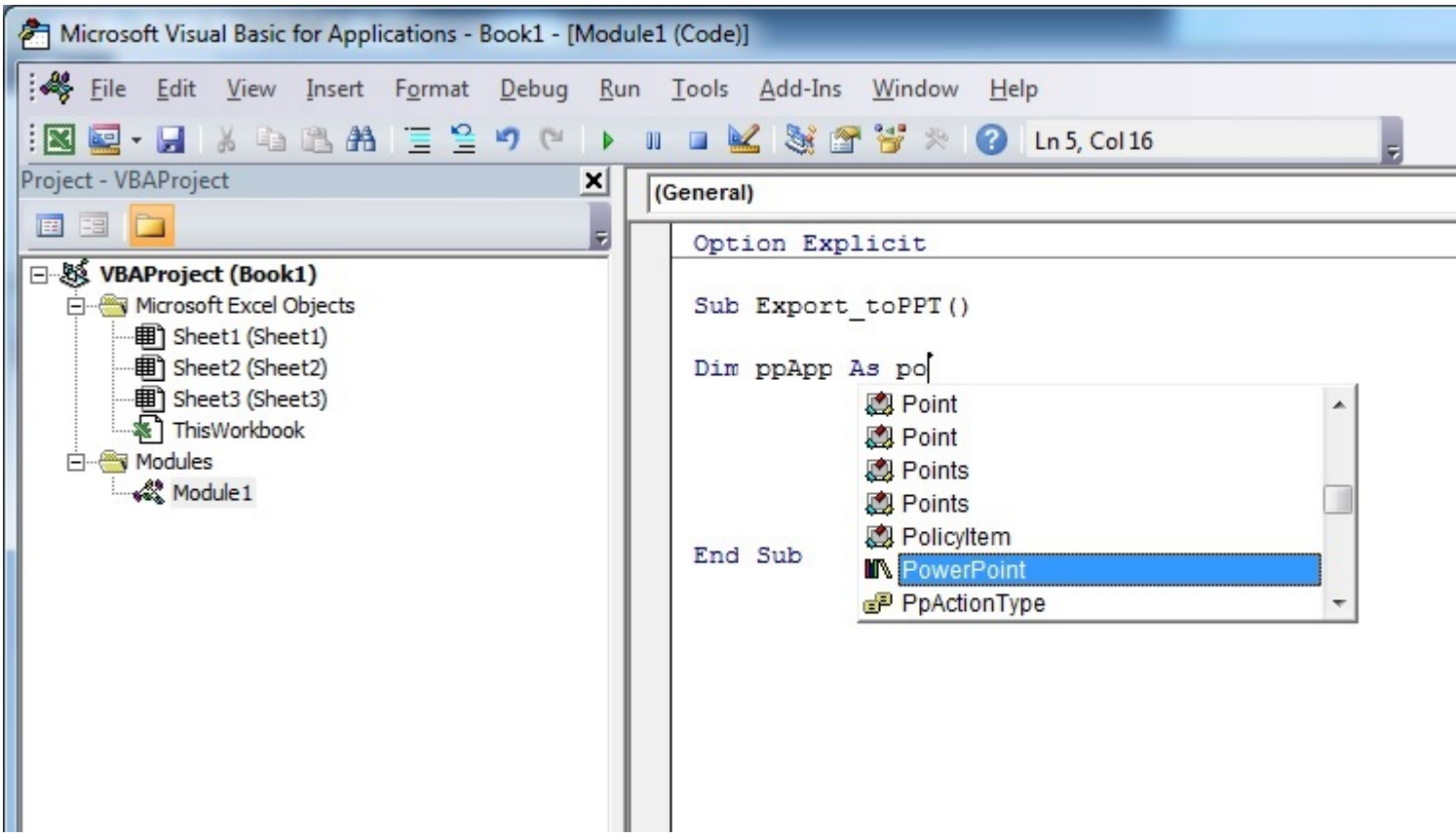


Paso 2 : Seleccione la referencia que desea agregar. En este ejemplo, nos desplazamos hacia abajo para encontrar " ***Microsoft PowerPoint 14.0 Object Library*** ", y luego presionamos " **OK** ".

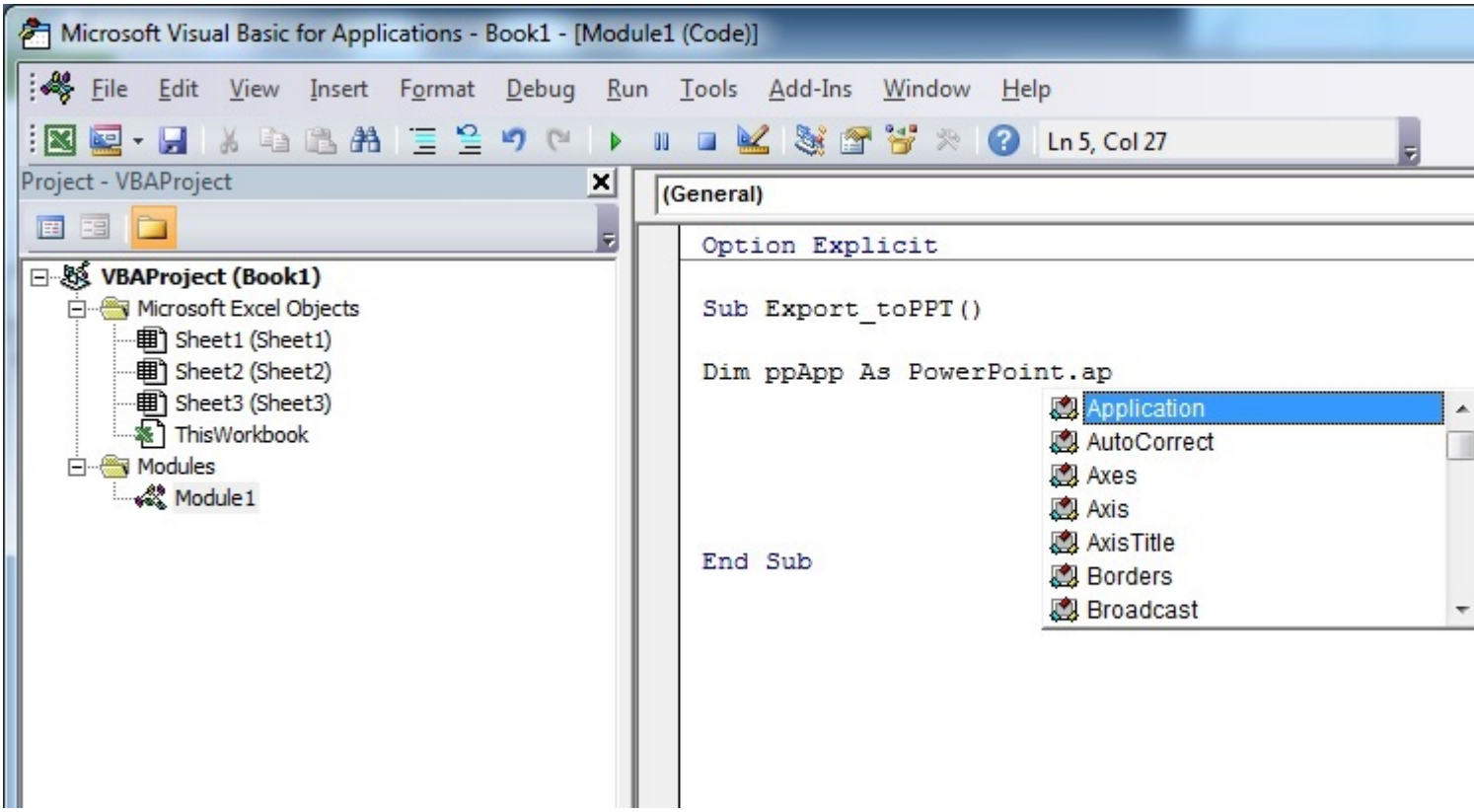


Nota: PowerPoint 14.0 significa que la versión de Office 2010 está instalada en la PC.

Paso 3 : en el Editor VB, una vez que presiona **Ctrl + Espacio** juntos, obtiene la opción de autocompletar de PowerPoint.

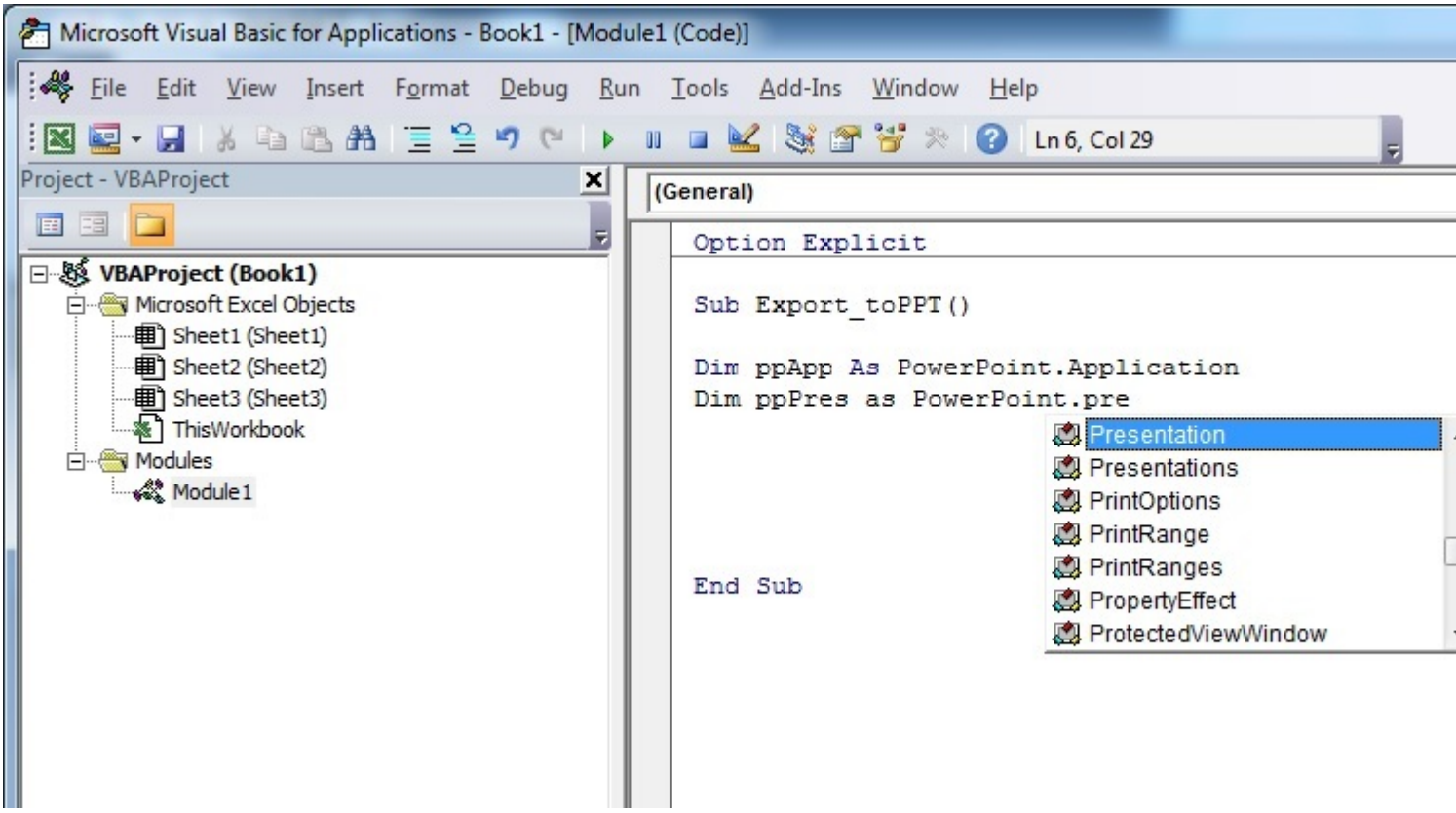


Después de seleccionar `PowerPoint` y presionar `.` , aparece otro menú con todas las opciones de objetos relacionadas con la Biblioteca de objetos de PowerPoint. Este ejemplo muestra cómo seleccionar la `Application` objeto de PowerPoint.

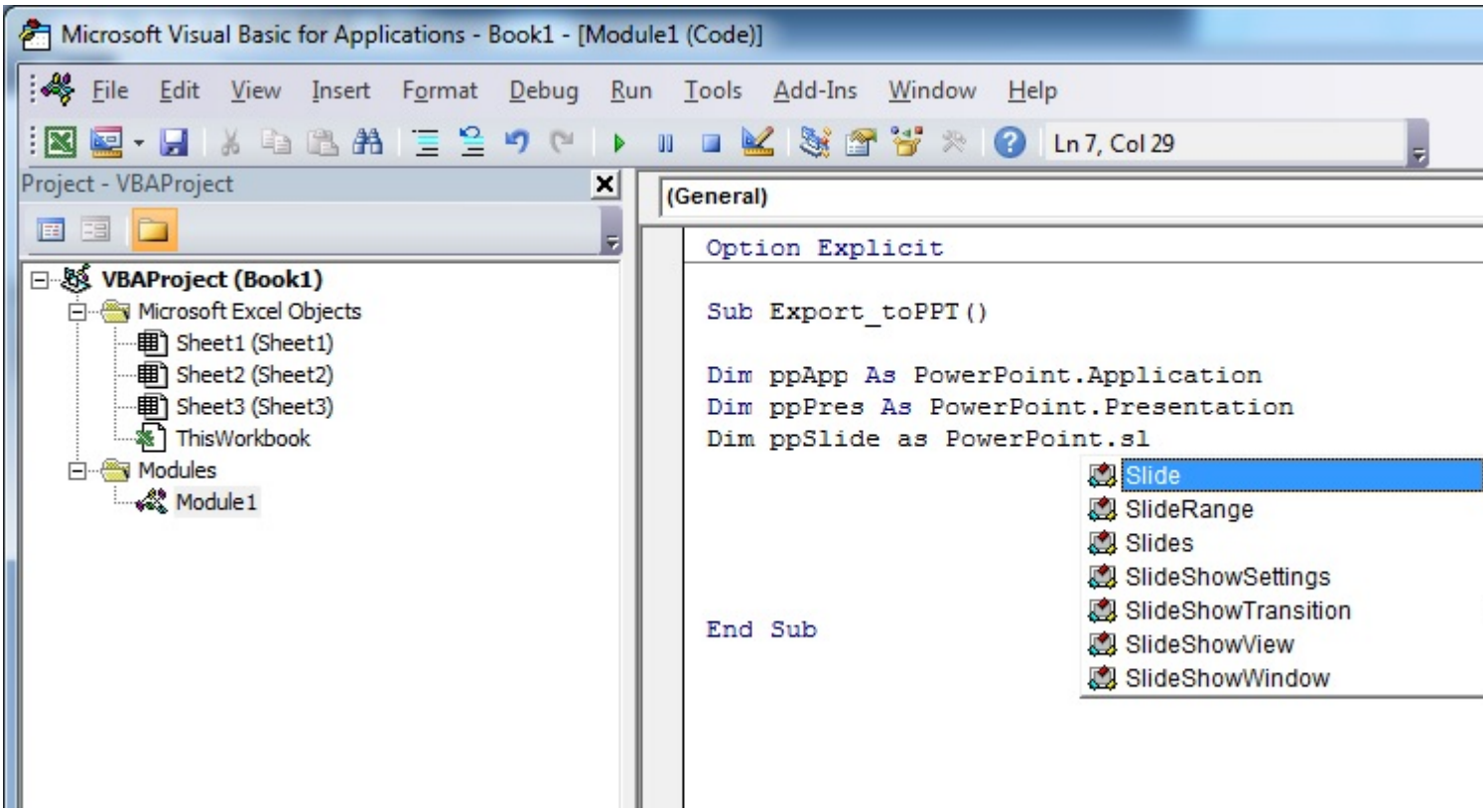


Paso 4 : Ahora el usuario puede declarar más variables utilizando la biblioteca de objetos de PowerPoint.

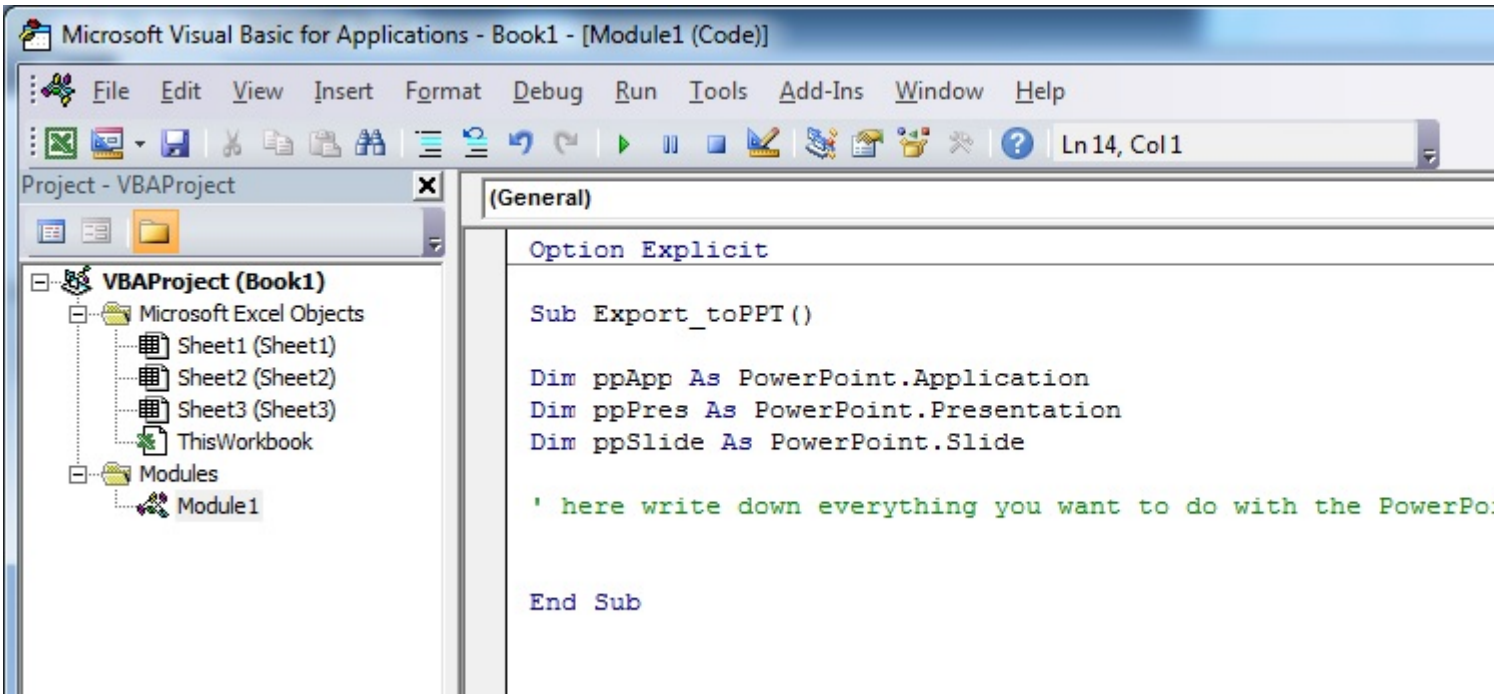
Declare una variable que hace referencia al objeto de `Presentation` de la biblioteca de objetos de PowerPoint.



Declare otra variable que haga referencia al objeto `Slide` de la biblioteca de objetos de PowerPoint.



Ahora la sección de declaración de variables se ve como en la captura de pantalla a continuación, y el usuario puede comenzar a usar estas variables en su código.

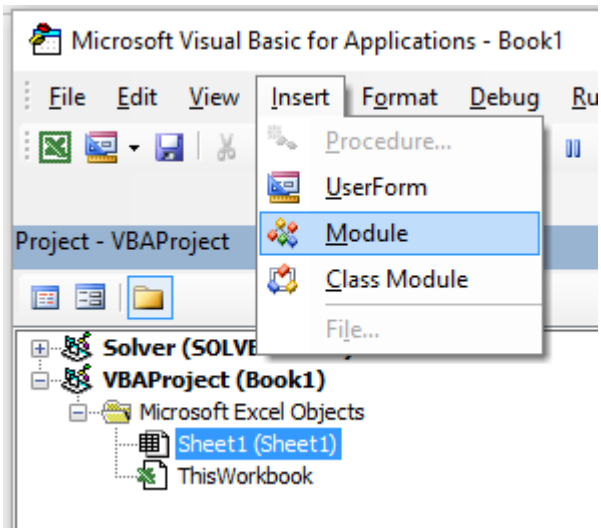


Versión en código de este tutorial:

```
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
```

Hola Mundo

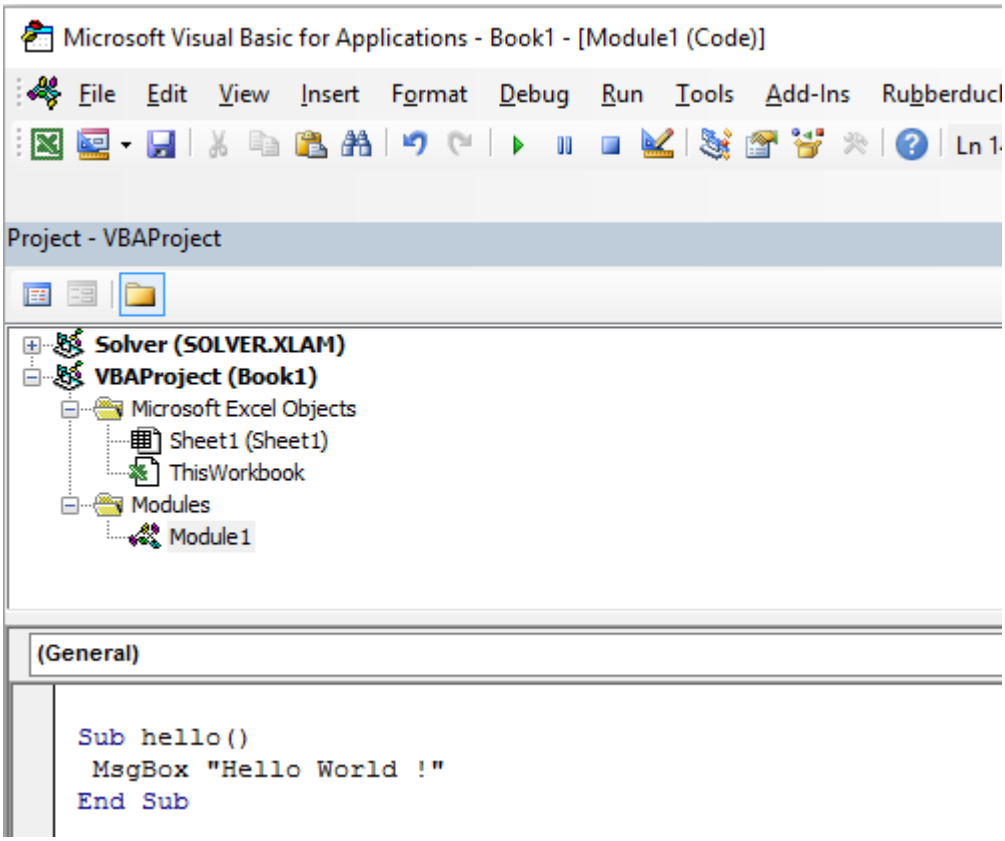
1. Abra el Editor de Visual Basic (vea Abrir el Editor de Visual Basic)
2. Haga clic en Insertar -> Módulo para agregar un nuevo módulo:



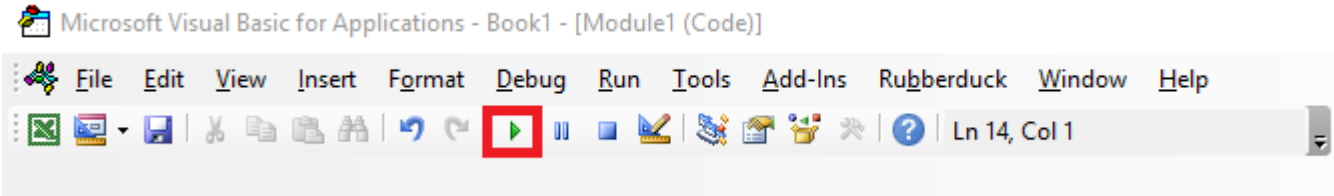
3. Copie y pegue el siguiente código en el nuevo módulo:

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

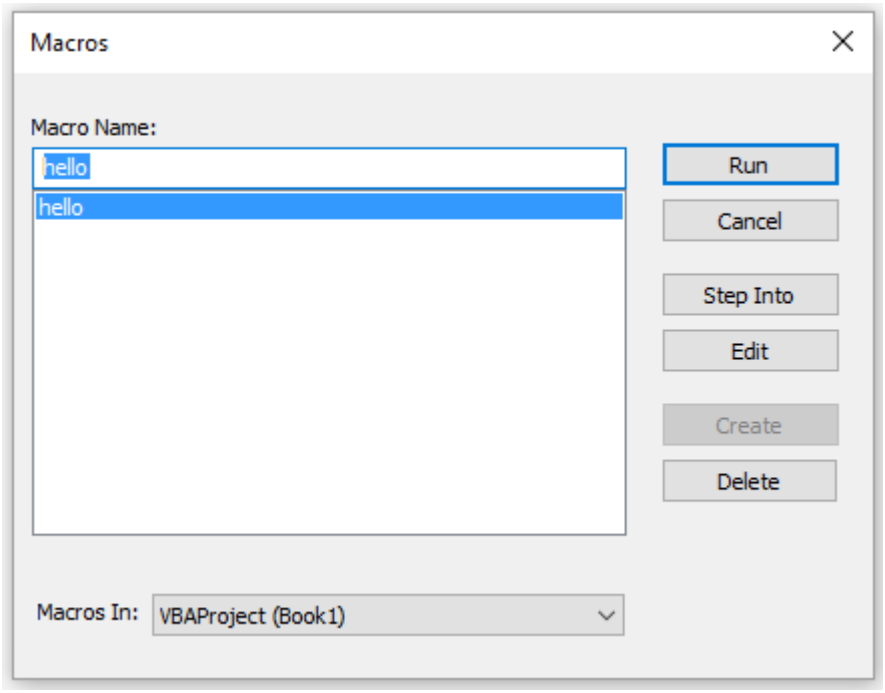
Para obtener :



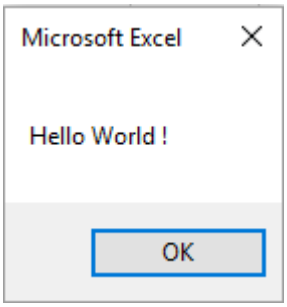
4. Haga clic en la flecha verde "jugar" (o presione F5) en la barra de herramientas de Visual Basic para ejecutar el programa:



5. Seleccione el nuevo sub creado "hola" y haga clic en `Run` :



6. Hecho, deberías ver la siguiente ventana:

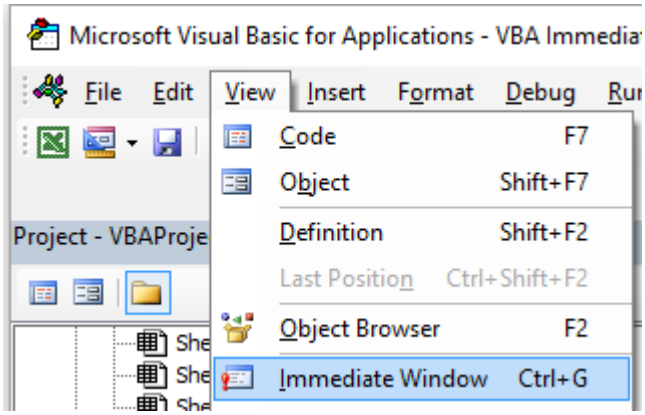


Comenzando con el modelo de objetos de Excel

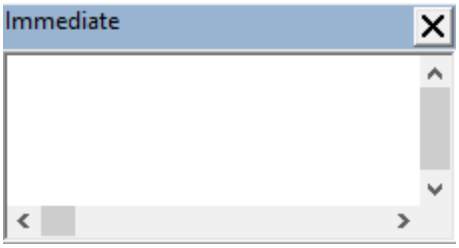
Este ejemplo pretende ser una introducción suave al Modelo de objetos de Excel **para principiantes** .

---

1. Abra el Editor de Visual Basic (VBE)
2. Haga clic en Ver -> Ventana Inmediata para abrir la ventana Inmediata (o `ctrl` + `G`):



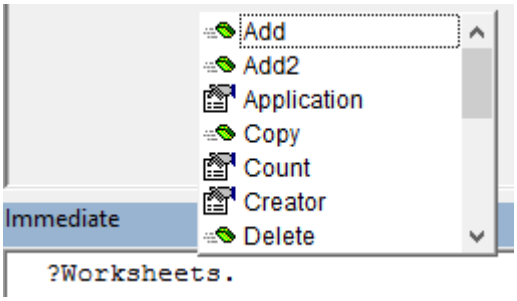
3. Debería ver la siguiente ventana inmediata en la parte inferior de VBE:



Esta ventana le permite probar directamente algunos códigos VBA. Así que vamos a empezar, escriba en esta consola:

```
?Worksheets.
```

VBE tiene intellisense y luego debería abrir una información sobre herramientas como en la siguiente figura:



Seleccione .Contar en la lista o escriba directamente `.Cout` para obtener:

```
?Worksheets.Count
```

4. Luego presione Enter. La expresión se evalúa y debe devolver 1. Esto indica el número de Hoja de trabajo actualmente presente en el libro de trabajo. El signo de interrogación (`?`) Es un alias para Debug.Print.

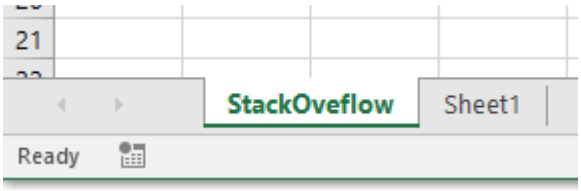
Las hojas de trabajo son un **objeto** y el conteo es un **método** . Excel tiene varios Objetos (`Workbook` `Worksheet` , `Worksheet` , `Range` , `Chart` ...) y cada uno contiene métodos y propiedades específicos. Puede encontrar la lista completa de Objeto en la referencia de Excel VBA . Hojas de trabajo Objeto se presenta aquí .

> Esta referencia de Excel VBA debe convertirse en su principal fuente de información con respecto al Modelo de objetos de Excel.

5. Ahora probemos otra expresión, escriba (sin el carácter `?`):

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

6. Presione Enter. Esto debería crear una nueva hoja de cálculo llamada `StackOverflow.` :



Para comprender esta expresión, debe leer la función Agregar en la referencia de Excel mencionada anteriormente. Encontrarás lo siguiente:

```
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.
```

Entonces, las `Worksheets.Add()` crean una nueva hoja de trabajo y la devuelven. La hoja de trabajo (**sin s**) es en sí misma un Objeto que se puede encontrar en la documentación y `Name` es una de sus **propiedades** (consulte aquí). Se define como:

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

Entonces, al investigar las diferentes definiciones de objetos, podemos entender este código `Worksheets.Add().Name = "StackOveflow"` .

`Add()` crea y agrega una nueva hoja de trabajo y le devuelve una **referencia** , luego establecemos su **propiedad** Name en "StackOverflow"

Ahora seamos más formales, Excel contiene varios objetos. Estos objetos pueden estar compuestos de una o varias colecciones de objetos Excel de la misma clase. Es el caso de `WorkSheets` que es una colección de objetos de `Worksheet` de `Worksheet` . Cada objeto tiene algunas propiedades y métodos con los que el programador puede interactuar.

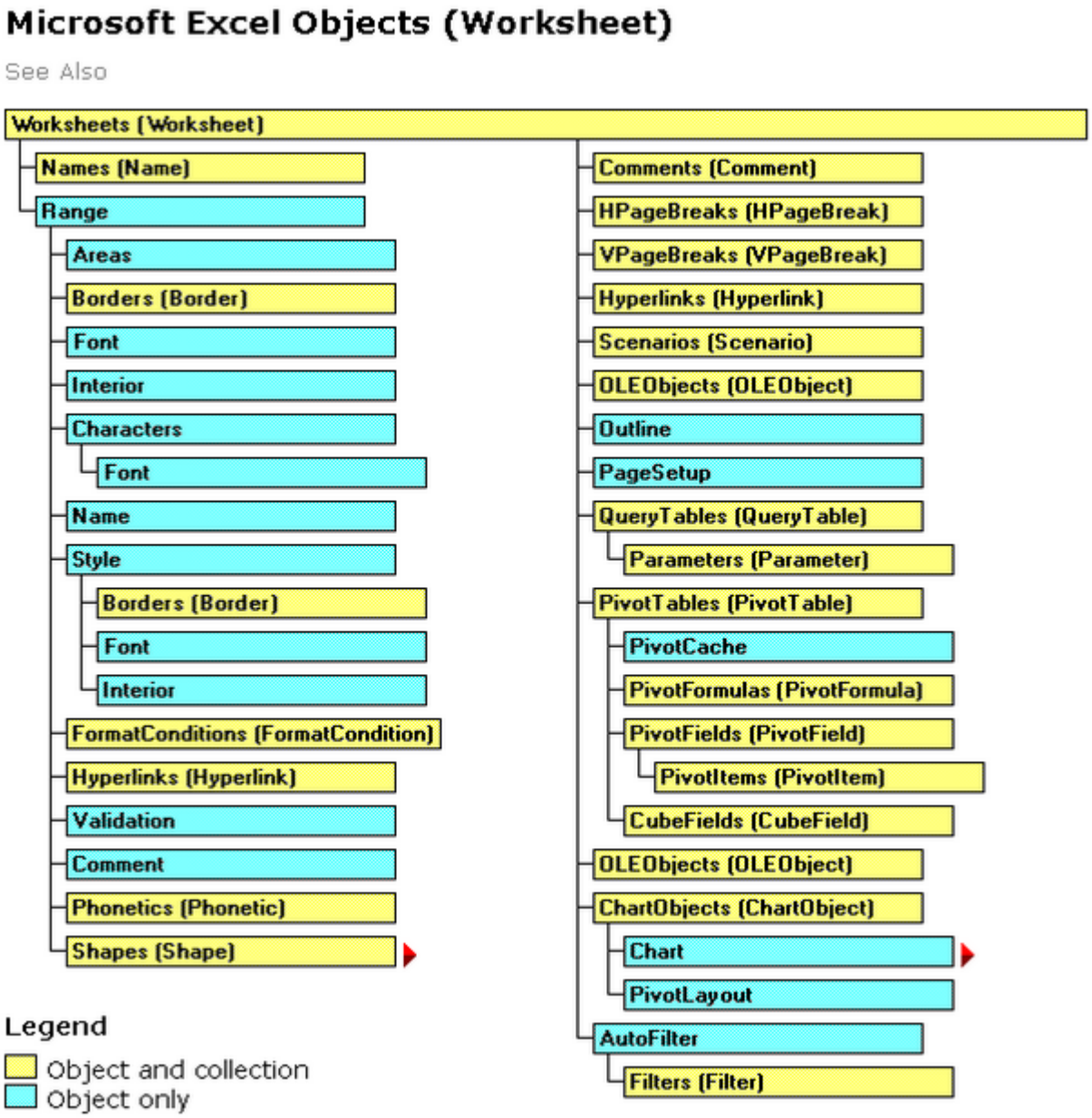
> El modelo de objetos de Excel se refiere a la **jerarquía de objetos de** Excel

En la parte superior de todos los objetos se encuentra el objeto `Application` , que representa la propia instancia de Excel. La programación en VBA requiere una buena comprensión de esta jerarquía porque siempre necesitamos una referencia a un objeto para poder llamar a un método o para establecer / obtener una propiedad.

El modelo de objetos de Excel (muy simplificado) se puede representar como,

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

A continuación se muestra una versión más detallada del objeto de la hoja de trabajo (como está en Excel 2007)



El modelo completo de objetos de Excel se puede encontrar aquí .

Finalmente, algunos objetos pueden tener `events` (por ejemplo, `Workbook.WindowActivate`) que también forman parte del Modelo de objetos de Excel.

Lea Empezando con excel-vba en línea: https://riptutorial.com/es/excel-vba/topic/777/empezando-con-excel-vba

Capítulo 2: Arrays

Examples

Poblando matrices (agregando valores)

Hay varias formas de rellenar una matriz.

Directamente

```
'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"
'...
```

Usando la función 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 rango

```
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 con Evaluar ()

```
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)
```

Usando la función Split ()

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

Arreglos dinámicos (cambio de tamaño de matriz y manejo dinámico)

Debido a que no son contenidos exclusivos de Excel-VBA, este Ejemplo se ha movido a la documentación de VBA.

Enlace: Arrays dinámicos (Array Resizing y Dynamic Handling)

Arreglos irregulares (Arrays of Arrays)

Debido a que no son contenidos exclusivos de Excel-VBA, este Ejemplo se ha movido a la documentación de VBA.

Enlace: Arreglos irregulares (Arrays of Arrays)

Compruebe si la matriz está inicializada (si contiene elementos o no).

Un problema común podría ser intentar iterar sobre Array, que no tiene valores. Por ejemplo:

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

Para evitar este problema, y para comprobar si una matriz contiene elementos, use este *oneliner* :

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

Arreglos dinámicos [Declaración de arreglo, cambio de tamaño]

```
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
```

Lea Arrays en línea: https://riptutorial.com/es/excel-vba/topic/2027/arrays

Capítulo 3: autofiltro Usos y mejores prácticas.

Introducción

El objetivo final del ***filtro automático*** es proporcionar de la manera más rápida posible la extracción de datos de cientos o miles de datos de filas para llamar la atención en los elementos en los que queremos centrarnos. Puede recibir parámetros como "texto / valores / colores" y se pueden apilar entre columnas. Puede conectar hasta 2 criterios por columna basados en conectores lógicos y conjuntos de reglas. Nota: Autofiltro funciona al filtrar filas, no hay Autofiltro para filtrar columnas (al menos no de forma nativa).

Observaciones

'Para usar Autofiltro dentro de VBA necesitamos llamar con al menos los siguientes parámetros:

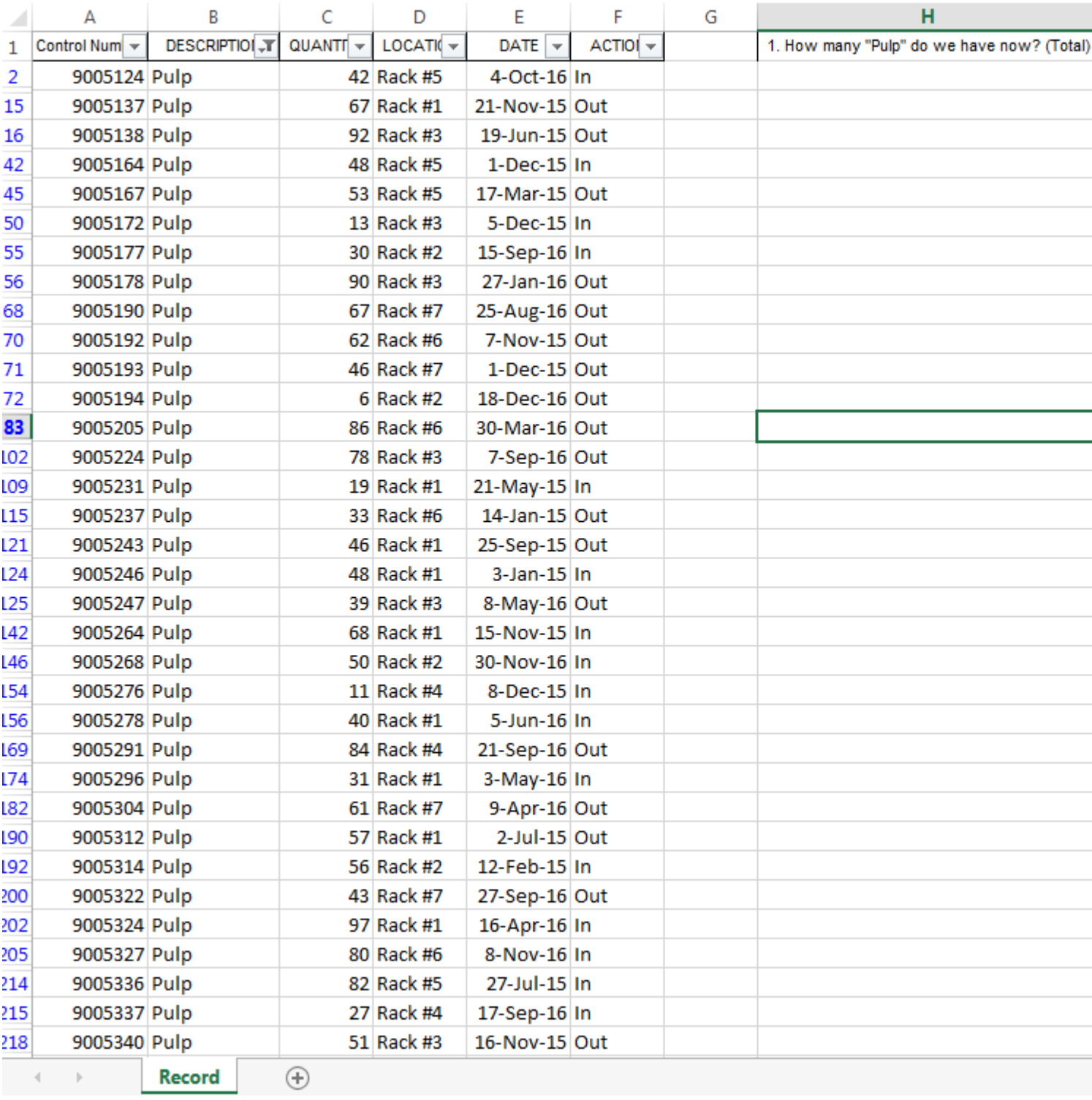
Hoja ("MySheet"). Rango ("MyRange"). Autofilter Field = (ColumnNumberWithin "MyRange" ToBeFilteredInNumericValue) Criteria1: = "WhatIWantToFilter"

'Hay muchos ejemplos en la web o aquí en stackoverflow

Examples

Smartfilter!

Situación problemática
El administrador del almacén tiene una hoja ("Registro") donde se almacena cada movimiento logístico realizado por la instalación, puede filtrar según sea necesario, aunque esto requiere mucho tiempo y le gustaría mejorar el proceso para calcular las consultas más rápidamente, por ejemplo. Ejemplo: ¿Cuántos "pulpa" tenemos ahora (en todos los racks)? ¿Cuánta pulpa tenemos ahora (en el estante # 5)? Los filtros son una gran herramienta pero, de alguna manera, se limitan a responder este tipo de pregunta en cuestión de segundos.



	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

Solución macro:

El programador sabe que los ***autofiltros son la mejor solución, más rápida y más confiable*** en este tipo de escenarios, ya que ***los datos ya existen en la hoja de trabajo*** y la ***entrada para ellos se puede obtener fácilmente,*** en este caso, mediante la entrada del usuario.

El enfoque utilizado es crear una hoja llamada "SmartFilter" donde el administrador puede filtrar fácilmente múltiples datos según sea necesario y el cálculo se realizará de forma instantánea también.

Él usa 2 módulos y el evento `Worksheet_Change` para este asunto

Código para la hoja de cálculo 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
```

Código para el módulo 1, llamado "General_Functions"

```
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
```

Código para el módulo 2, llamado "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
```

Pruebas y resultados:

	A	B	C	D	E	F	G	H	I	J
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

Como vimos en la imagen anterior, esta tarea se ha logrado fácilmente. Mediante el uso de ***filtros automáticos*** , se proporcionó una solución que solo ***demora unos segundos en calcularse, es fácil de explicar al usuario (*** ya que él / ella está familiarizado con este comando) y ***tomó algunas líneas hacia el codificador.***

Lea autofiltro Usos y mejores prácticas. en línea: https://riptutorial.com/es/excel-vba/topic/8645/autofiltro-usos-y-mejores-practicas-

Capítulo 4: Celdas / Rangos Combinados

Examples

Piensa dos veces antes de usar celdas combinadas / rangos

En primer lugar, las celdas combinadas están ahí solo para mejorar el aspecto de sus hojas.

¡Así que, literalmente, es lo último que debe hacer, una vez que su hoja y su libro de trabajo son totalmente funcionales!

¿Dónde están los datos en un rango fusionado?

Cuando fusionas un Rango, solo mostrarás un bloque.

Los datos estarán en la **primera celda de ese rango** , ¡y los **otros serán celdas vacías** !

Un buen punto al respecto: no es necesario llenar todas las celdas o el rango una vez fusionado, ¡simplemente llene la primera celda! ;)

Los otros aspectos de este rango fusionado son globalmente negativos:

- Si usa un método para encontrar la última fila o columna , se arriesgará a algunos errores
- Si recorres las filas y has fusionado algunos rangos para una mejor legibilidad, encontrarás celdas vacías y no el valor mostrado por el rango fusionado

Lea Celdas / Rangos Combinados en línea: https://riptutorial.com/es/excel-vba/topic/7308/celdas---rangos-combinados

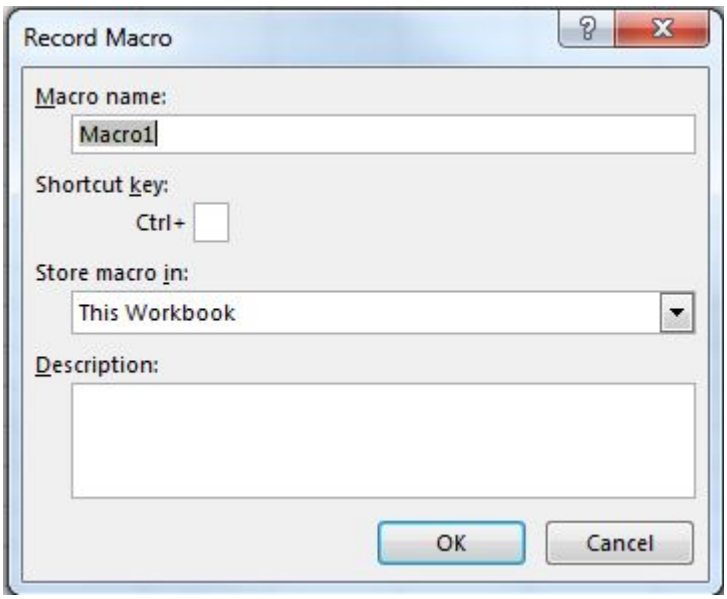
Capítulo 5: Cómo grabar una macro

Examples

Cómo grabar una macro

La forma más fácil de grabar una macro es que el botón en la esquina inferior izquierda de Excel tenga este aspecto: 

Cuando hagas clic en esto, aparecerá una ventana emergente que te pedirá que le asignes un nombre a la Macro y que decidas si quieres tener una tecla de acceso directo. Además, pregunta dónde almacenar la macro y para una descripción. Puede elegir el nombre que desee, no se permiten espacios.



Si desea tener un acceso directo asignado a su macro para un uso rápido, elija una letra que recordará para poder usar la macro rápida y fácilmente una y otra vez.

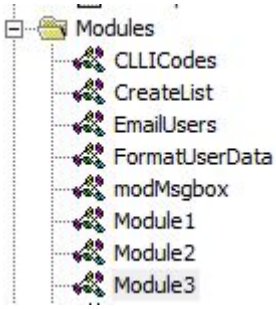
Puede almacenar la macro en "Este libro de trabajo", "Nuevo libro de trabajo" o "Libro de macros personal". Si desea que la macro que está a punto de grabar esté disponible solo en el libro de trabajo actual, elija "Este libro de trabajo". Si desea guardarlo en un libro nuevo, elija "Nuevo libro". Y si desea que la macro esté disponible para cualquier libro que abra, elija "Libro de macros personal".

Una vez que haya completado esta ventana emergente, haga clic en "Aceptar".

A continuación, realice las acciones que desee repetir con la macro. Cuando haya terminado, haga clic en el mismo botón para detener la grabación. Ahora se ve así:

Ahora puedes ir a la pestaña Desarrollador y abrir Visual Basic. (o use Alt + F11)

Ahora tendrá un nuevo módulo en la carpeta Módulos.

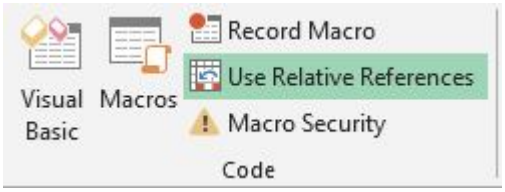


El módulo más nuevo contendrá la macro que acaba de grabar. Haga doble clic en él para que aparezca.

Hice un sencillo copiar y pegar:

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

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

Si no desea que siempre se pegue en "A12", puede usar Referencias Relativas marcando la casilla "Usar Referencias Relativas" en la pestaña Desarrollador:



Siguiendo los mismos pasos que antes, ahora la Macro se convertirá en esto:

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

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

Aún copiando el valor de "A1" en una celda 11 filas hacia abajo, pero ahora puede realizar la

misma macro con cualquier celda inicial y el valor de esa celda se copiará en las celdas 11 filas hacia abajo.

Lea Cómo grabar una macro en línea: https://riptutorial.com/es/excel-vba/topic/8204/como-grabar-una-macro

Capítulo 6: Creación de un menú desplegable en la hoja de cálculo activa con un cuadro combinado

Introducción

Este es un ejemplo sencillo que muestra cómo crear un menú desplegable en la Hoja activa de su libro de trabajo al insertar un objeto Activex Combo Box en la hoja. Podrás insertar una de las cinco canciones de Jimi Hendrix en cualquier celda activada de la hoja y poder borrarla, según corresponda.

Examples

Jimi Hendrix Menu

En general, el código se coloca en el módulo de una hoja.

Este es el evento Worksheet_SelectionChange, que se activa cada vez que se selecciona una celda diferente en la hoja activa. Puede seleccionar "Hoja de trabajo" en el primer menú desplegable sobre la ventana de código y "Selección_Cambiar" en el menú desplegable que se encuentra al lado. En este caso, cada vez que activa una celda, el código se redirige al código del Combo Box.

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

   ComboBox1_Change

End Sub
```

Aquí, la rutina dedicada al ComboBox se codifica para el evento Change por defecto. En él, hay una matriz fija, rellenada con todas las opciones. No la opción BORRAR en la última posición, que se utilizará para borrar el contenido de una celda. La matriz se entrega al Combo Box y se pasa a la rutina que hace el trabajo.

```
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
```

La matriz se pasa a la rutina que llena las celdas con el nombre de la canción o el valor nulo para vaciarlas. Primero, a una variable entera se le asigna el valor de la posición de la elección que el usuario hace. Luego, el cuadro combinado se mueve a la esquina SUPERIOR IZQUIERDA de la celda que el usuario activa y sus dimensiones se ajustan para que la experiencia sea más fluida. A la celda activa se le asigna el valor en la posición en la variable entera, que sigue la elección del usuario. En caso de que el usuario seleccione BORRAR de las opciones, la celda se vacía.

Toda la rutina se repite para cada celda seleccionada.

```
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
```

Ejemplo 2: Opciones no incluidas

Este ejemplo se usa para especificar opciones que podrían no estar incluidas en una base de datos de viviendas disponibles y en las comodidades de los asistentes.

Se basa en el ejemplo anterior, con algunas diferencias:

1. Ya no son necesarios dos procedimientos para un solo cuadro combinado, que se realiza combinando el código en un solo procedimiento.
2. El uso de la propiedad LinkedCell para permitir la entrada correcta de la selección del usuario cada vez
3. La inclusión de una función de copia de seguridad para garantizar que la celda activa se encuentre en la columna correcta y un código de prevención de errores, basado en la experiencia anterior, donde los valores numéricos se formatearían como cadenas cuando se rellenaran con la celda activa.

```
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 anterior se inicia cada vez que se activa una celda con el evento SelectionChange en el módulo de la hoja de trabajo:

```
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
```

Lea Creación de un menú desplegable en la hoja de cálculo activa con un cuadro combinado en línea: https://riptutorial.com/es/excel-vba/topic/8929/creacion-de-un-menu-desplegable-en-la-hoja-de-calculo-activa-con-un-cuadro-combinado

Capítulo 7: Cuadernos de ejercicios

Examples

Libros de aplicación

En muchas aplicaciones de Excel, el código VBA realiza acciones dirigidas al libro de trabajo en el que está contenido. Guarda ese libro de trabajo con una extensión ".xlsm" y las macros de VBA solo se centran en las hojas de cálculo y los datos que contiene. Sin embargo, a menudo hay ocasiones en las que necesita combinar o combinar datos de otros libros, o escribir algunos de sus datos en un libro separado. Abrir, cerrar, guardar, crear y eliminar otros libros de trabajo es una necesidad común para muchas aplicaciones VBA.

En cualquier momento en el Editor de VBA, puede ver y acceder a todos y cada uno de los libros de trabajo abiertos actualmente por esa instancia de Excel utilizando la propiedad `Workbooks` de trabajo del objeto `Application` . La documentación de MSDN lo explica con referencias.

Cuándo usar ActiveWorkbook y ThisWorkbook

Es una buena práctica de VBA especificar siempre a qué libro de trabajo se refiere su código de VBA. Si se omite esta especificación, entonces VBA asume que el código está dirigido al libro activo actual (`ActiveWorkbook`).

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

Sin embargo, cuando se abren varios libros al mismo tiempo, especialmente cuando se ejecuta el código VBA desde un complemento de Excel, las referencias al `ActiveWorkbook` pueden ser confusas o mal dirigidas. Por ejemplo, un complemento con un UDF que verifica la hora del día y lo compara con un valor almacenado en una de las hojas de trabajo del complemento (que normalmente no son fácilmente visibles para el usuario) tendrá que identificar explícitamente qué libro es siendo referenciado. En nuestro ejemplo, nuestro libro abierto (y activo) tiene una fórmula en la celda A1 `=EarlyOrLate()` y NO tiene ningún VBA escrito para ese libro activo. En nuestro complemento, tenemos la siguiente función definida por el usuario (UDF):

```
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
```

El código para el UDF se escribe y almacena en el complemento de Excel instalado. Utiliza datos almacenados en una hoja de cálculo en el complemento llamado "WatchTime". Si el UDF hubiera utilizado `ActiveWorkbook` lugar de `ThisWorkbook` , entonces nunca podría garantizar qué libro estaba

destinado.

Abriendo un libro de trabajo (nuevo), incluso si ya está abierto

Si desea acceder a un libro de trabajo que ya está abierto, obtener la asignación de la colección de `Workbooks` es sencillo:

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

Si desea crear un nuevo libro, a continuación, utilizar el `Workbooks` objeto de la colección de `Add` una nueva entrada.

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

Hay ocasiones en las que no puede o (o le importa) si el libro que necesita ya está abierto o no, o es posible que no exista. La función de ejemplo muestra cómo devolver siempre un objeto de libro válido.

```
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
```

Guardando un libro de ejercicios sin preguntar al usuario

A menudo, guardar nuevos datos en un libro de trabajo usando VBA causará una pregunta emergente que indica que el archivo ya existe.

Para evitar esta pregunta emergente, debe suprimir este tipo de alertas.

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

Cambiar el número predeterminado de hojas de trabajo en un nuevo libro de trabajo

El número "predeterminado de fábrica" de hojas de trabajo creadas en un nuevo libro de Excel generalmente se establece en tres. Su código VBA puede establecer explícitamente la cantidad de hojas de trabajo en un nuevo libro.

```
'--- 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
```

Lea Cuadernos de ejercicios en línea: https://riptutorial.com/es/excel-vba/topic/2969/cuadernos-de-ejercicios

Capítulo 8: CustomDocumentProperties en la práctica

Introducción

El uso de CustomDocumentProperties (CDP) es un buen método para almacenar valores definidos por el usuario de una manera relativamente segura dentro del mismo libro de trabajo, pero evitando mostrar valores de celdas relacionadas simplemente en una hoja de trabajo desprotegida *).

Nota: los CDP representan una colección separada comparable a BuiltInDocumentProperties, pero permiten crear sus propios nombres de propiedad definidos por el usuario en lugar de una colección fija.

*) Alternativamente, puede ingresar valores también en un libro de trabajo oculto o "muy oculto".

Examples

Organizando nuevos números de factura.

Incrementar un número de factura y guardar su valor es una tarea frecuente. El uso de CustomDocumentProperties (CDP) es un buen método para almacenar dichos números de una forma relativamente segura dentro del mismo libro de trabajo, pero evitando mostrar valores de celdas relacionadas simplemente en una hoja de trabajo desprotegida.

Sugerencia adicional:

Alternativamente, puede ingresar valores también en una hoja de cálculo oculta o incluso en una hoja de cálculo "muy oculta" (consulte Uso de las hojas xlVeryHidden . Por supuesto, también es posible guardar datos en archivos externos (por ejemplo, archivo ini, csv o cualquier otro tipo) o el registro.

Contenido de ejemplo :

El siguiente ejemplo muestra

- una función NextInvoiceNo que establece y devuelve el siguiente número de factura,
- un procedimiento DeleteInvoiceNo, que elimina completamente la factura CDP, así como
- un procedimiento showAllCDPs que enumera la colección completa de CDP con todos los nombres. Si no utiliza VBA, también puede enumerarlos a través de la información del libro: Información | Propiedades [DropDown:] | Propiedades Avanzadas | Personalizado

Puede obtener y configurar el siguiente número de factura (último no más uno) simplemente llamando a la función mencionada anteriormente, devolviendo un valor de cadena para facilitar la adición de prefijos. "InvoiceNo" se usa implícitamente como nombre de CDP en todos los

procedimientos.

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

Código de ejemplo:

```
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
```

Lea CustomDocumentProperties en la práctica en línea: https://riptutorial.com/es/excel-vba/topic/10932/customdocumentproperties-en-la-practica

Capítulo 9: Declaraciones condicionales

Examples

La afirmacion if

La instrucción de control `If` permite que se ejecute un código diferente dependiendo de la evaluación de una instrucción condicional (booleana). Una declaración condicional es aquella que se evalúa como `True` o `False` , por ejemplo, `x > 2` .

Hay tres patrones que se pueden usar al implementar una instrucción `If` , que se describen a continuación. Tenga en cuenta que una evaluación condicional `If` siempre va seguida de un `Then` .

1. La evaluación de uno `If` sentencia condicional y hacer algo si es `True`

Una sola línea `If` comunicado

Esta es la forma más corta de usar un `If` y es útil cuando solo se necesita realizar una declaración en una evaluación `True` . Al usar esta sintaxis, todo el código debe estar en una sola línea. No incluya un `End If` al final de la línea.

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

`If` bloque

Si es necesario ejecutar varias líneas de código en una evaluación `True` , se puede usar un bloque `If` .

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

Tenga en cuenta que, si se utiliza un bloque multilínea `If` , se requiere un `End If` correspondiente.

2. Evaluar una sentencia condicional `If` , hacer una cosa si es `True` y hacer otra cosa si es `False`

Línea única `If` , declaración `Else`

Esto se puede usar si una declaración se va a realizar en una evaluación `True` y una declaración diferente se debe llevar a cabo en una evaluación `False` . Tenga cuidado al usar esta sintaxis, ya que a menudo es menos claro para los lectores que hay una declaración `Else` . Al usar esta sintaxis, todo el código debe estar en una sola línea. No incluya un `End If` al final de la línea.

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

`If` , `Else` block

Use un bloque `If` , `Else` para agregar claridad a su código, o si es necesario ejecutar varias líneas de código bajo una evaluación `True` o `False` .

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

Tenga en cuenta que, si se utiliza un bloque multilínea `If` , se requiere un `End If` correspondiente.

3. Evaluar muchas declaraciones condicionales, cuando las declaraciones anteriores son todas `False` y hacer algo diferente para cada una.

Este patrón es el uso más general de `If` y se usaría cuando hay muchas condiciones que no se superponen que requieren un tratamiento diferente. A diferencia de los dos primeros patrones, este caso requiere el uso de un bloque `If` , incluso si solo se ejecutará una línea de código para cada condición.

`If` , `ElseIf` , `...` , `Else` block

En lugar de tener que crear muchos bloques `If` uno debajo del otro, se puede usar un `ElseIf` para evaluar una condición adicional. El `ElseIf` solo se evalúa si alguno de los anteriores `If` evaluación es `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
```

Se pueden incluir tantas `ElseIf` control `ElseIf` entre un `If` y un `End If` según se requiera. No se requiere una declaración de control de `Else` cuando se usa `ElseIf` (aunque se recomienda), pero si se incluye, debe ser la declaración de control final antes de `End If` .

Lea Declaraciones condicionales en línea: https://riptutorial.com/es/excel-vba/topic/9632/declaraciones-condicionales

Capítulo 10: Depuración y solución de problemas

Sintaxis

- Debug.Print (cadena)
- Para para

Examples

Debug.Print

Para imprimir una lista de las descripciones de los códigos de error en la ventana Inmediata, `Debug.Print` a la función `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
```

Puede mostrar la ventana Inmediato por:

- Selección de **V** er | **Me imagino una** ventana desde la barra de menú
- Usando el atajo de teclado **Ctrl-G**

Detener

El comando Detener pausará la ejecución cuando se llame. A partir de ahí, el proceso se puede reanudar o ejecutar paso a paso.

```
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
```

Ventana inmediata

Si desea probar una línea de código de macro sin necesidad de ejecutar un subcomité completo, puede escribir comandos directamente en la ventana Inmediato y presionar `ENTER` para ejecutar la línea.

¿Para probar la salida de una línea, puede precederla con un signo de interrogación `?` para imprimir directamente en la ventana Inmediata. Alternativamente, también puede usar el comando de `print` para `print` la salida.

Mientras esté en el Editor de Visual Basic, presione `CTRL + G` para abrir la ventana Inmediato. Para cambiar el nombre de la hoja seleccionada actualmente a "Hoja de ejemplo", escriba lo siguiente en la ventana Inmediato y presione `ENTER`

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

Para imprimir el nombre de la hoja seleccionada actualmente directamente en la ventana Inmediata

```
? ActiveSheet.Name
ExampleSheet
```

Este método puede ser muy útil para probar la funcionalidad de las funciones integradas o definidas por el usuario antes de implementarlas en el código. El siguiente ejemplo muestra cómo se puede usar la ventana Inmediato para probar la salida de una función o serie de funciones para confirmar un esperado.

```
'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 ventana Inmediato también se puede usar para configurar o restablecer la aplicación, el libro de trabajo u otras propiedades necesarias. Esto puede ser útil si tiene `Application.EnableEvents = False` en una subrutina que inesperadamente lanza un error, causando que se cierre sin restablecer el valor a `True` (lo que puede causar una funcionalidad frustrante e inesperada. En ese caso, los comandos se pueden escribir directamente en la ventana inmediata y ejecute:

```
? 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
```

Para técnicas de depuración más avanzadas, se pueden usar dos puntos `:` como separador de línea. Esto se puede usar para expresiones de varias líneas, como en bucle en el siguiente ejemplo.

```
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
```

Utilice el temporizador para encontrar cuellos de botella en el rendimiento

El primer paso para optimizar la velocidad es encontrar las secciones de código más lentas. La función `Timer` VBA devuelve el número de segundos transcurridos desde la medianoche con una precisión de 1/255 de segundo (3.90625 milisegundos) en PC con Windows. Las funciones VBA `Now` y `Time` son solo precisas a un segundo.

```
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"
```

Agregando un punto de interrupción a su código

Puede agregar fácilmente un punto de interrupción a su código haciendo clic en la columna gris a la izquierda de la línea de su código VBA donde desea que se detenga la ejecución. Aparece un punto rojo en la columna y el código del punto de interrupción también se resalta en rojo.

Puede agregar múltiples puntos de interrupción a lo largo de su código y reanudar la ejecución presionando el icono "jugar" en la barra de menú. No todo el código puede ser un punto de interrupción como líneas de definición variable, la primera o la última línea de un procedimiento y las líneas de comentarios no se pueden seleccionar como un punto de interrupción.

Ventana Locales del Depurador

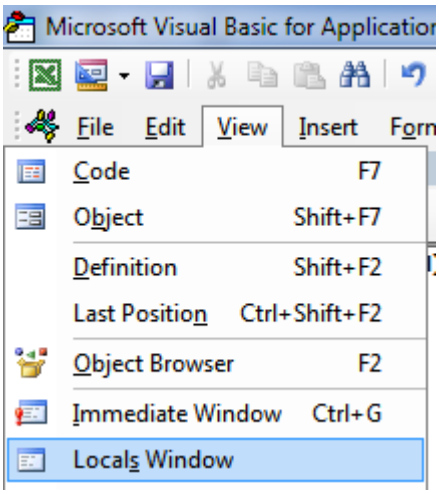
La ventana Locales proporciona un acceso fácil al valor actual de las variables y objetos dentro del alcance de la función o subrutina que está ejecutando. Es una herramienta esencial para

depurar su código y pasar por los cambios para encontrar problemas. También te permite explorar propiedades que quizás no sabías que existían.

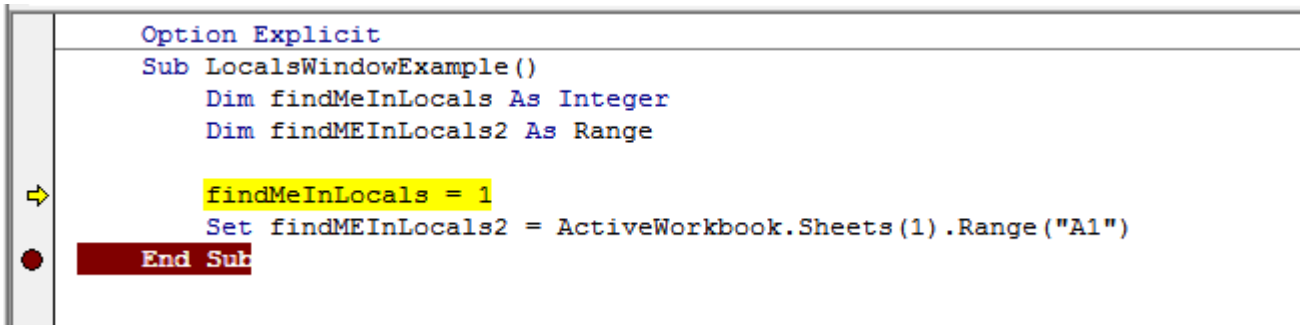
Tomemos el siguiente ejemplo,

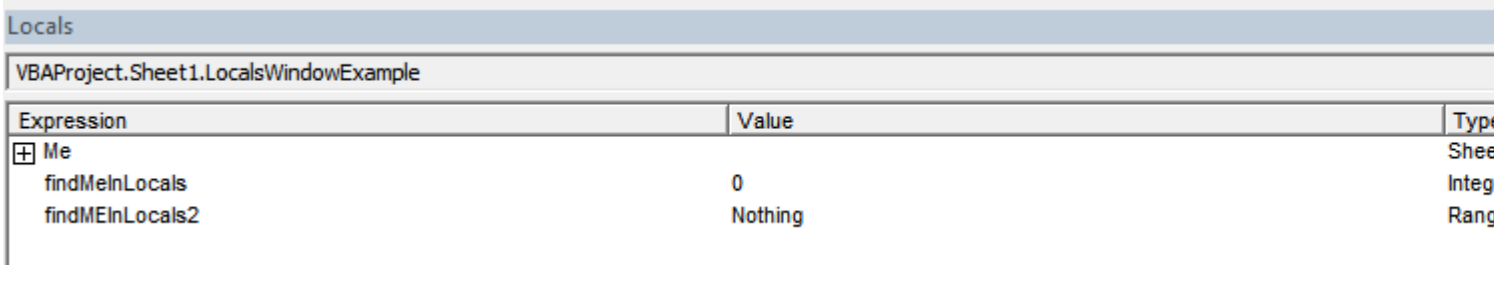
```
Option Explicit
Sub LocalsWindowExample()
    Dim findMeInLocals As Integer
    Dim findMEInLocals2 As Range

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

En el Editor de VBA, haga clic en Ver -> Ventana Locales



Luego, al pasar por el código con F8 después de hacer clic dentro de la subrutina, nos hemos detenido antes de asignar findMeinLocals. Abajo puede ver que el valor es 0 --- y esto es lo que se usaría si nunca le asignara un valor. El objeto de rango es 'Nada'.



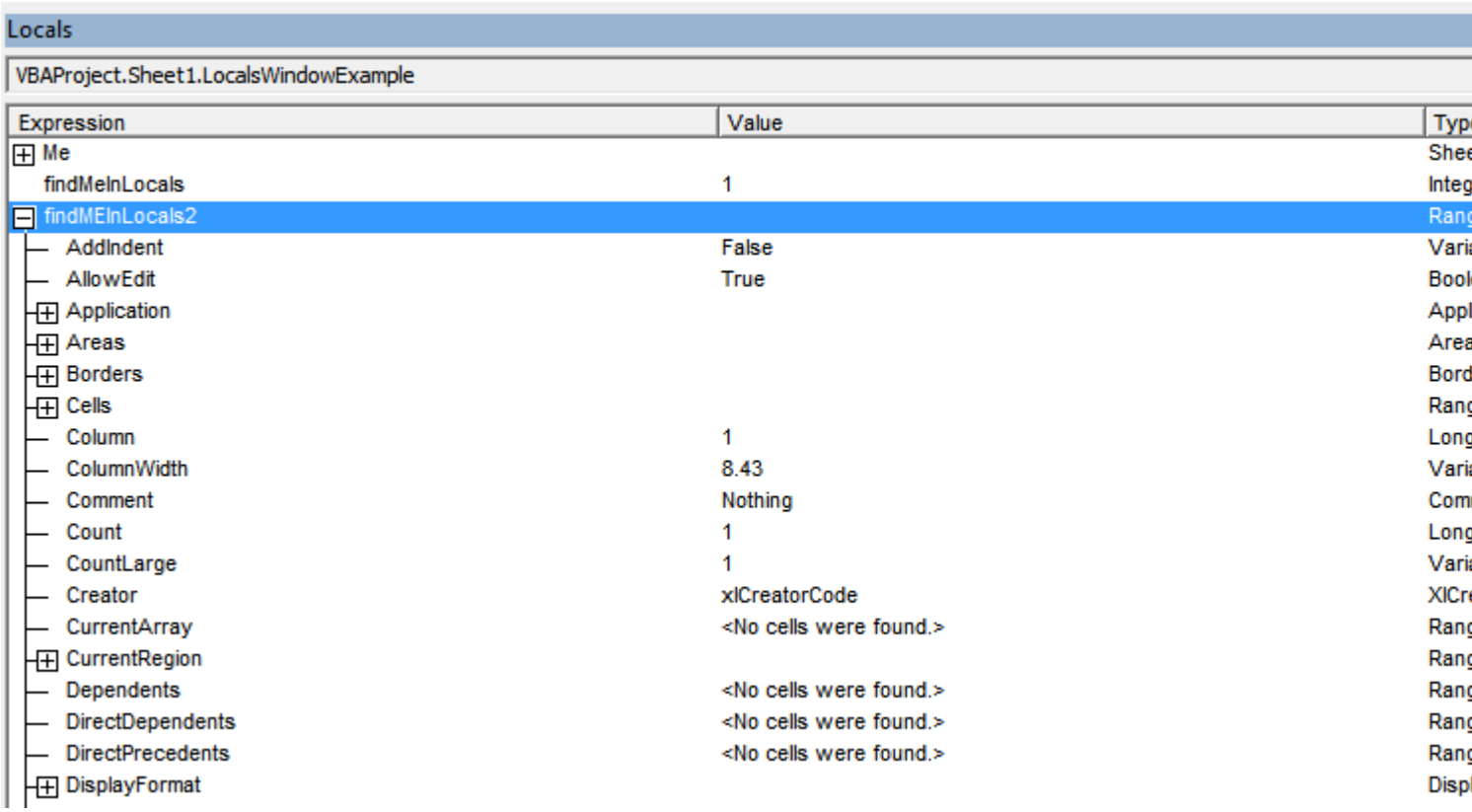


Si nos detenemos justo antes de que finalice la subrutina, podemos ver los valores finales de las

variables.

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

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

Podemos ver findMeInLocals con un valor de 1 y tipo de Integer, y FindMeInLocals2 con un tipo de Rango / Rango. Si hacemos clic en el signo + podemos expandir el objeto y ver sus propiedades, como conteo o columna.



Lea Depuración y solución de problemas en línea: https://riptutorial.com/es/excel-vba/topic/861/depuracion-y-solucion-de-problemas

Capítulo 11: Errores comunes

Examples

Referencias de calificación

Cuando se hace referencia a una `worksheet` , un `range` o `cells` individuales, es importante calificar completamente la referencia.

Por ejemplo:

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

No está completamente calificado: las referencias de `Cells` no tienen un libro de trabajo y una hoja de cálculo asociados. Sin una referencia explícita, las celdas se refieren a `ActiveSheet` por defecto. Por lo tanto, este código fallará (producirá resultados incorrectos) si una hoja de cálculo que no sea `Sheet1` es el `ActiveSheet` actual.

La forma más fácil de corregir esto es usar una instrucción `With` siguiente manera:

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

Alternativamente, puede usar una variable de la hoja de trabajo. (Probablemente este sea el método preferido si su código necesita hacer referencia a varias Hojas de trabajo, como copiar datos de una hoja a otra).

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

Otro problema frecuente es hacer referencia a la colección de hojas de trabajo sin calificar el libro de trabajo. Por ejemplo:

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

La hoja de trabajo `Sheet1` no está completamente calificada y carece de un libro de trabajo. Esto podría fallar si se hace referencia a varios libros en el código. En su lugar, utilice uno de los siguientes:

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

Sin embargo, evite usar lo siguiente:

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

De manera similar, para los objetos de `range` , si no están calificados explícitamente, el `range` se referirá a la hoja actualmente activa:

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

Es lo mismo que:

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

Eliminar filas o columnas en un bucle

Si desea eliminar filas (o columnas) en un bucle, siempre debe hacerlo desde el final del rango y retroceder en cada paso. En caso de utilizar el código:

```
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
```

Te perderás algunas filas. Por ejemplo, si el código elimina la fila 3, entonces la fila 4 se convierte en la fila 3. Sin embargo, la variable `i` cambiará a 4. Entonces, en este caso, el código perderá una fila y marcará otra, que no estaba en el rango anterior.

El código correcto sería

```
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` y `ThisWorkbook` veces se usan indistintamente por los nuevos usuarios de VBA sin entender completamente a qué se refiere cada objeto, esto puede causar un comportamiento no deseado en tiempo de ejecución. Ambos objetos pertenecen al objeto de aplicación.

El objeto `ActiveWorkbook` refiere al libro de trabajo que se encuentra actualmente en la vista superior del objeto de aplicación de Excel en el momento de la ejecución. *(por ejemplo, el libro de trabajo que puede ver e interactuar en el momento en que se hace referencia a este objeto)*

```
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
```

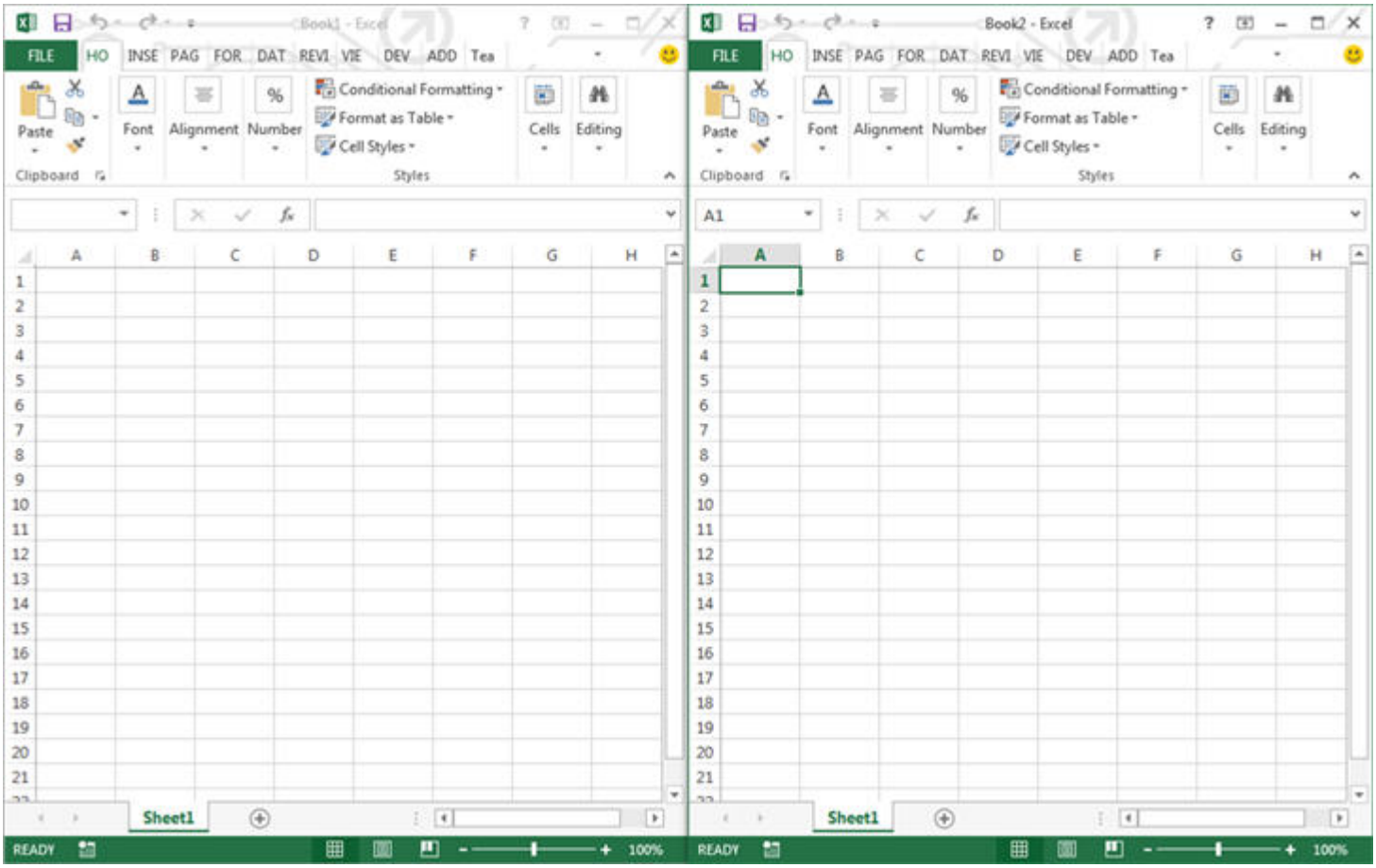
El objeto `ThisWorkbook` refiere al libro de trabajo al que pertenece el código en el momento en que se está ejecutando.

```
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
```

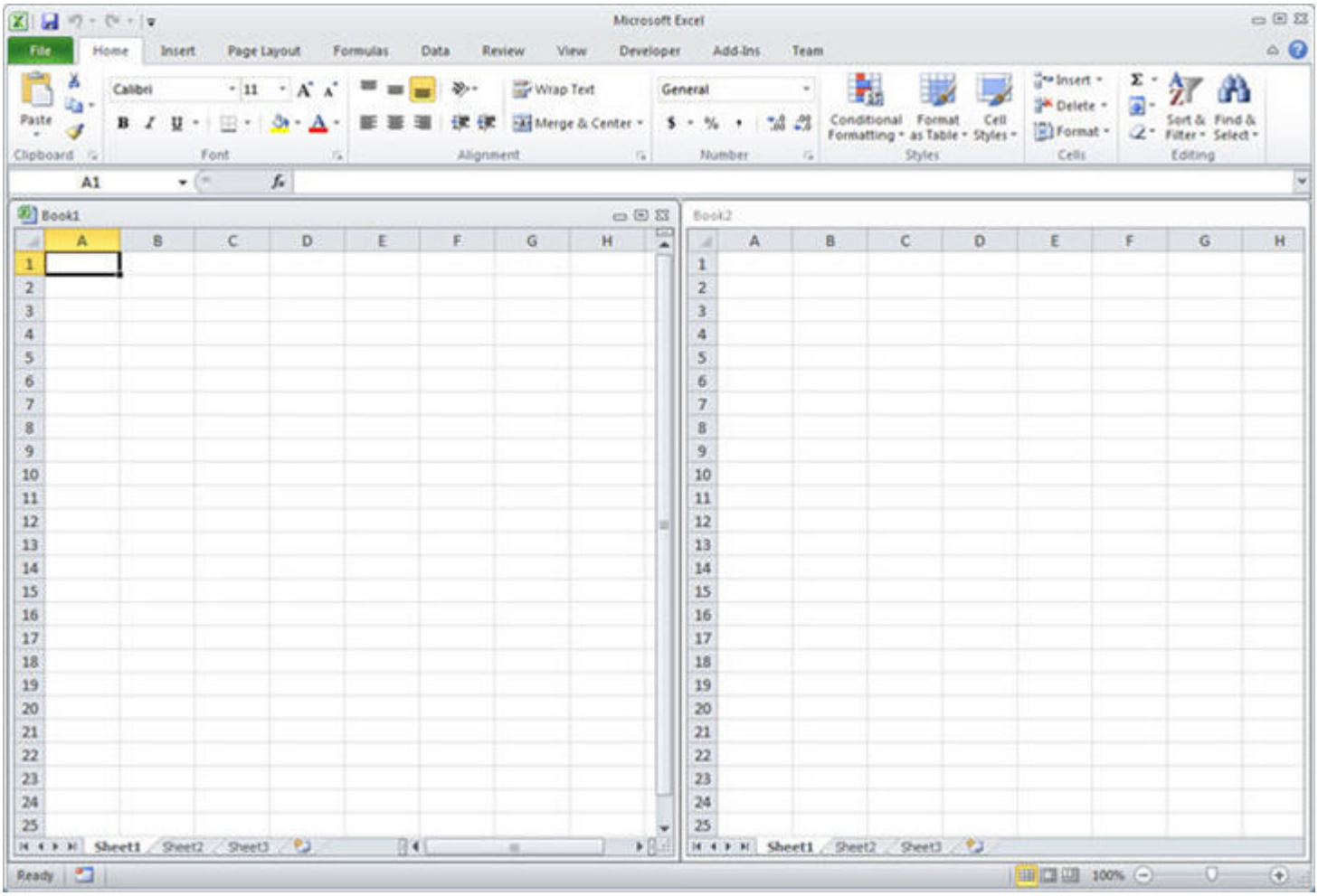
Interfaz de documento único frente a interfaces de documento múltiples

> Tenga en cuenta que Microsoft Excel 2013 (y superior) utiliza la Interfaz de documento único (SDI) y que Excel 2010 (y más abajo) utiliza Interfaces de documentos múltiples (MDI).

Esto implica que para Excel 2013 (SDI), cada libro en una sola instancia de Excel contiene su **propia** interfaz de usuario de cinta:



A la inversa, para Excel 2010, cada libro en una sola instancia de Excel utilizó una IU de cinta **común** (MDI):



Esto plantea algunos problemas importantes si desea migrar un código VBA (2010 <-> 2013) que interactúa con la cinta de opciones.

> Se debe crear un procedimiento para actualizar los controles de la IU de la cinta en el mismo estado en todos los libros de trabajo para Excel 2013 y superior.

Tenga en cuenta que :

1. Todos los métodos, eventos y propiedades de la ventana de nivel de aplicación de Excel no se verán afectados. (`Application.ActiveWindow` , `Application.Windows` ...)
2. En Excel 2013 y superior (SDI), todos los métodos, eventos y propiedades de la ventana de nivel de libro de trabajo ahora operan en la ventana de nivel superior. Es posible recuperar el identificador de esta ventana de nivel superior con `Application.Hwnd`

Para obtener más detalles, consulte la fuente de este ejemplo: MSDN .

Esto también causa algunos problemas con las formas de usuario sin modelo. Vea aquí para una solución.

Lea Errores comunes en línea: https://riptutorial.com/es/excel-vba/topic/1576/errores-comunes

Capítulo 12: Excel VBA consejos y trucos

Observaciones

Este tema consiste en una amplia variedad de consejos y trucos útiles descubiertos por los usuarios de SO a través de su experiencia en la codificación. A menudo, estos son ejemplos de formas de sortear frustraciones comunes o formas de usar Excel de una manera más "inteligente".

Examples

Usando las hojas xlVeryHidden

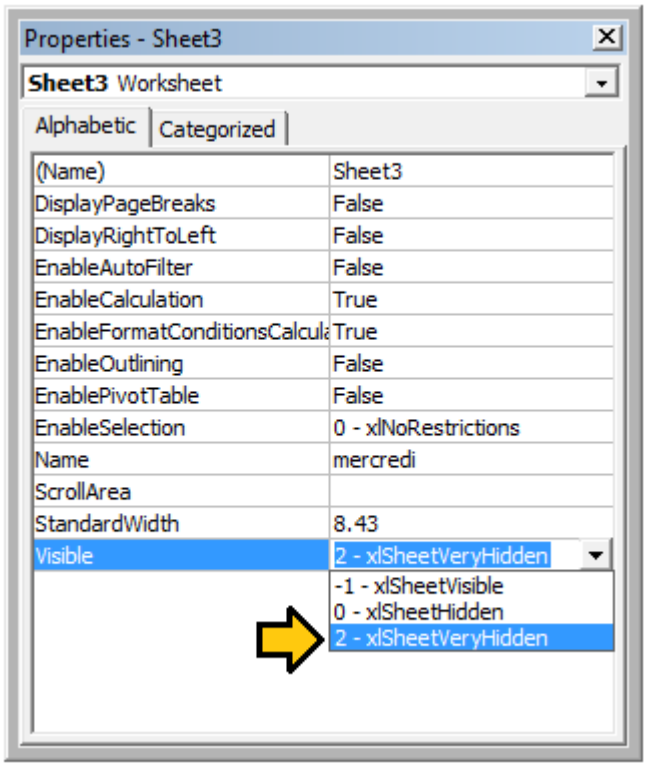
Las hojas de trabajo en Excel tienen tres opciones para la propiedad `Visible` . Estas opciones están representadas por constantes en la enumeración `xlSheetVisibility` y son las siguientes:

1. `xlVisible` o `xlSheetVisible` : `-1` (el valor predeterminado para las hojas nuevas)
2. `xlHidden` o `xlSheetHidden` : `0`
3. `xlVeryHidden` `xlSheetVeryHidden` valor: `2`

Las hojas visibles representan la visibilidad predeterminada para las hojas. Se pueden ver en la barra de pestañas de la hoja y se pueden seleccionar y ver libremente. Las hojas ocultas se ocultan de la barra de pestañas de la hoja y, por lo tanto, no se pueden seleccionar. Sin embargo, las hojas ocultas se pueden ocultar de la ventana de Excel haciendo clic derecho en las pestañas de la hoja y seleccionando "No ocultar"

Las hojas muy ocultas, por otro lado, *solo* son accesibles a través del Editor de Visual Basic. Esto los convierte en una herramienta increíblemente útil para almacenar datos en instancias de Excel, así como para almacenar datos que deben ocultarse a los usuarios finales. Se puede acceder a las hojas mediante una referencia con nombre dentro del código VBA, lo que permite un uso fácil de los datos almacenados.

Para cambiar manualmente la propiedad .Visible de una hoja de trabajo a xlSheetVeryHidden, abra la ventana de Propiedades de VBE (`F4`), seleccione la hoja de trabajo que desea cambiar y use el menú desplegable en la fila trece para hacer su selección.



Para cambiar la propiedad .Visible de una hoja de cálculo a xlSheetVeryHidden[1] en el código, acceda de manera similar a la propiedad .Visible y asigne un nuevo valor.

```
with Sheet3
    .Visible = xlSheetVeryHidden
end with
```

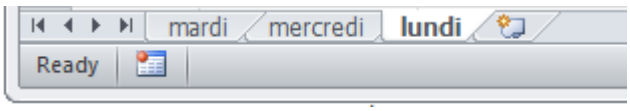
---

[1] Tanto **xlVeryHidden** como **xlSheetVeryHidden** devuelven un valor numérico de **2** (son intercambiables).

Hoja de trabajo .Nombre, .Index o .CódigoNombre

Sabemos que las "mejores prácticas" dictan que un objeto de rango debe tener su hoja de cálculo principal referenciada explícitamente. Se puede hacer referencia a una hoja de cálculo por su propiedad .Name, su propiedad numérica .Index o su propiedad .CodeName, pero un usuario puede reordenar la cola de la hoja de trabajo simplemente arrastrando una pestaña de nombre o renombrando la hoja de cálculo con un doble clic en la misma pestaña y algunos escribiendo en un libro de trabajo desprotegido.

Considere una hoja de trabajo estándar de tres. Ha cambiado el nombre de las tres hojas de trabajo el lunes, martes y miércoles en ese orden y ha codificado los sub procedimientos de VBA que hacen referencia a estos. Ahora considere que un usuario aparece y decide que el lunes corresponde al final de la cola de la hoja de trabajo, luego aparece otro y decide que los nombres de la hoja de trabajo se ven mejor en francés. Ahora tiene un libro de trabajo con una cola de pestañas con el nombre de la hoja de trabajo que se parece a lo siguiente.



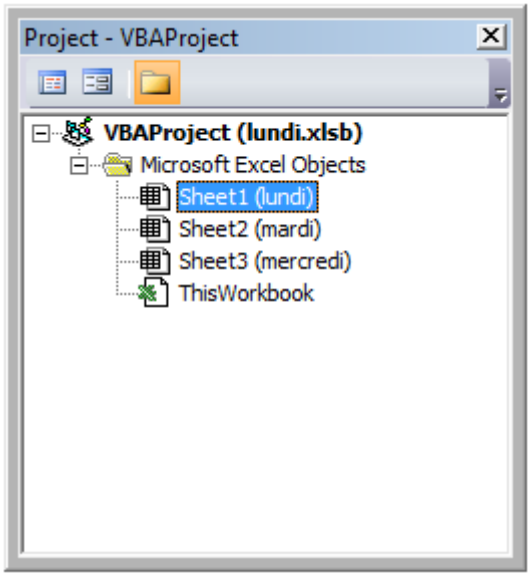
Si ha utilizado alguno de los siguientes métodos de referencia de la hoja de trabajo, su código ahora estaría roto.

```
'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
```

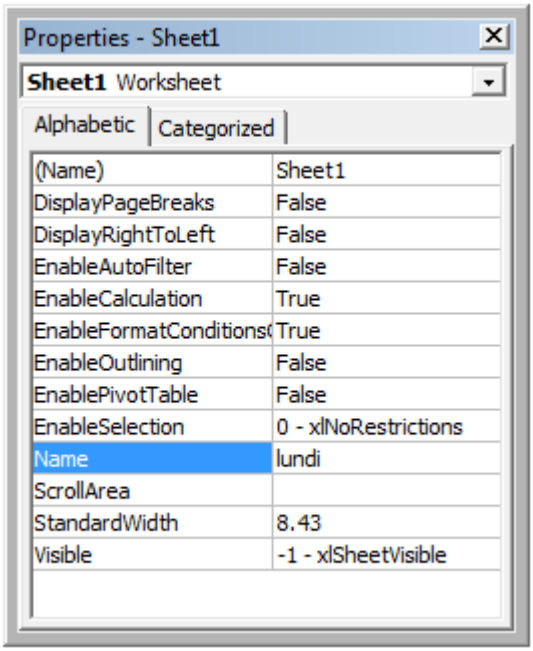
Tanto el pedido original como el nombre original de la hoja de trabajo se han comprometido. Sin embargo, si ha usado la propiedad .CodeName de la hoja de trabajo, su sub procedimiento todavía estaría operativo

```
with Sheet1
    'operation code here; for example:
    .Range(.Cells(2, "A"), .Cells(.Rows.Count, "A").End(xlUp)) = 1
end with
```

La siguiente imagen muestra la ventana del proyecto VBA ([Ctrl] + R) que enumera las hojas de trabajo por .CodeName luego por .Name (entre paréntesis). El orden en que se muestran no cambia; el .Index ordinal se toma según el orden en que se muestran en la cola de la pestaña de nombre en la ventana de la hoja de trabajo.



Si bien es poco común cambiar el nombre de un .CodeName, no es imposible. Simplemente abra la ventana Propiedades de VBE ([F4]).



La hoja de trabajo .CodeName está en la primera fila. El nombre de la hoja de cálculo está en el décimo. Ambos son editables.

Uso de cadenas con delimitadores en lugar de matrices dinámicas

El uso de arreglos dinámicos en VBA puede ser bastante complejo y requiere mucho tiempo en conjuntos de datos muy grandes. Al almacenar tipos de datos simples en una matriz dinámica (Cadenas, Números, Booleanos, etc.), se pueden evitar las declaraciones `ReDim Preserve` de las matrices dinámicas en VBA utilizando la función `Split()` con algunos procedimientos de cadena inteligentes. Por ejemplo, veremos un bucle que agrega una serie de valores de un rango a una cadena en función de algunas condiciones, y luego usamos esa cadena para completar los valores de 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 propia cadena `Delimiter` se puede establecer en cualquier valor, pero es prudente elegir un valor que no se producirá naturalmente dentro del conjunto. Digamos, por ejemplo, que estaban procesando una columna de fechas. En ese caso, utilizando `.` , `-` , o `/` sería imprudente como delimitadores, ya que las fechas se podrían formatear para usar cualquiera de estos, generando más puntos de datos de los que anticipó.

Nota: existen limitaciones para usar este método (es decir, la longitud máxima de las cadenas), por lo que debe usarse con precaución en casos de conjuntos de datos muy grandes. Este no es necesariamente el método más rápido o más eficaz para crear arreglos dinámicos en VBA, pero es una alternativa viable.

Haga doble clic en el evento para formas de Excel

De forma predeterminada, las Formas en Excel no tienen una forma específica de manejar clics simples o dobles, que contienen solo la propiedad "OnAction" para permitirle manejar clics. Sin embargo, puede haber casos en que su código requiera que actúe de manera diferente (o exclusivamente) con un doble clic. La siguiente subrutina se puede agregar a su proyecto de VBA y, cuando se configura como la rutina de `OnAction` para su forma, le permite actuar en doble clic.

```
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
```

Esta rutina hará que la forma ignore funcionalmente el primer clic, solo ejecutando el código deseado en el segundo clic dentro del intervalo de tiempo especificado.

Cuadro de diálogo Abrir archivo - Varios archivos

Esta subrutina es un ejemplo rápido de cómo permitir que un usuario seleccione varios archivos y luego haga algo con esas rutas de archivos, como obtener los nombres de los archivos y enviarlos a la consola a través de 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
```

Lea Excel VBA consejos y trucos en línea: https://riptutorial.com/es/excel-vba/topic/2240/excel-vba-consejos-y-trucos

Capítulo 13: Formato condicional utilizando VBA

Observaciones

No puede definir más de tres formatos condicionales para un rango. Use el método Modificar para modificar un formato condicional existente, o use el método Eliminar para eliminar un formato existente antes de agregar uno nuevo.

Examples

FormatoCondiciones.Agregar

Sintaxis:

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

Parámetros:

Nombre	Requerido / Opcional	Tipo de datos
Tipo	Necesario	XlFormatConditionType
Operador	Opcional	Variante
Fórmula 1	Opcional	Variante
Formula2	Opcional	Variante

Enumeración XlFormatConditionType:

Nombre	Descripción
xlAboveAverageCondition	Condición por encima de la media
xlBlanksCondition	Condición de espacios en blanco
xlCellValue	Valor de celda
xlColorScale	Escala de color

Nombre	Descripción
xlDatabar	Barra de datos
Condición de error	Condición de errores
expresión xl	Expresión
XlIconSet	Conjunto de iconos
xlNoBlanksCondition	Sin espacios en blanco
xlNoErrorsCondition	Condición sin errores.
xlTextString	Cadena de texto
xlTimePeriod	Periodo de tiempo
xlTop10	Los 10 mejores valores
xlUniqueValues	Valores únicos

Formato por valor de celda:

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

Operadores:

Nombre
xlEntre
xlEqual
xlgrande
xlGreaterEqual
xless
xlLessEqual
xlNotBetween

Nombre
xlNotEqual

Si Type es xlExpression, el argumento Operator se ignora.

El formateo por texto contiene:

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

Operadores:

Nombre	Descripción
xlBeginsCon	Comienza con un valor especificado.
xlContains	Contiene un valor especificado.
xlDoesNotContain	No contiene el valor especificado.
xlEndsWith	Termine con el valor especificado.

Formato por periodo de tiempo

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

Operadores:

Nombre
ayer
xlmañana
xlLast7Days

Nombre
xlLastWeek
xl esta semana
xlnextWeek
xlLastMonth
xl este mes
xlNextMonth

Eliminar formato condicional

Eliminar todo el formato condicional en el rango:

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

Eliminar todo el formato condicional en la hoja de trabajo:

```
Cells.FormatConditions.Delete
```

FormatConditions.AddUniqueValues

Resaltando valores duplicados

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

Destacando los valores únicos

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

FormatConditions.AddTop10

Destacando los 5 mejores valores

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

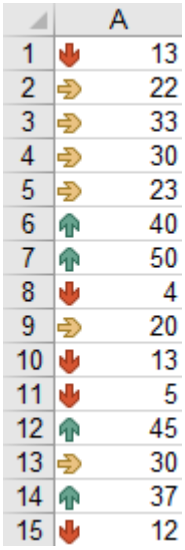
FormatConditions.AddAboveAverage

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

Operadores:

Nombre	Descripción
XlAboveAverage	Por encima de la media
XlAboveStdDev	Por encima de la desviación estándar
XlBelowAverage	Por debajo de la media
XlBelowStdDev	Por debajo de la desviación estándar
XlEqualAboveAverage	Igual por encima de la media
XlEqualBelowAverage	Igual por debajo de la media

FormatConditions.AddIconSetCondition



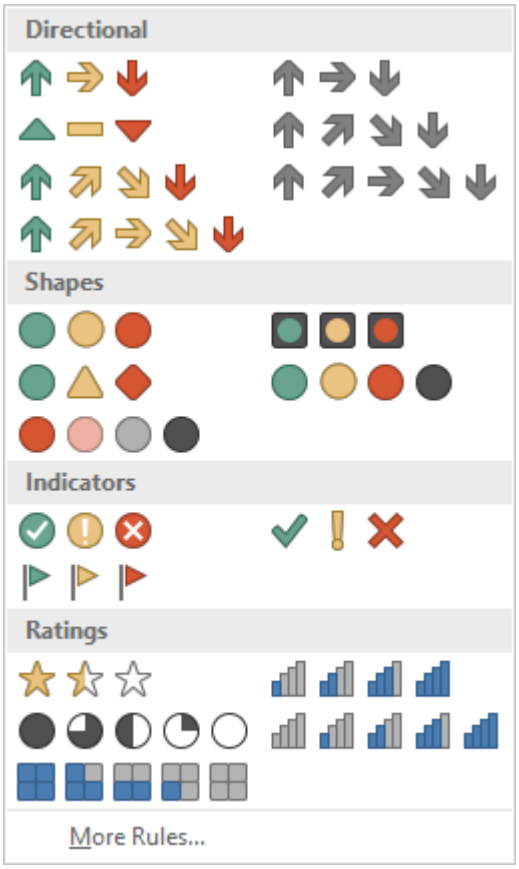
	A
1	13
2	22
3	33
4	30
5	23
6	40
7	50
8	4
9	20
10	13
11	5
12	45
13	30
14	37
15	12

```
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:

Nombre
xl3rrows
xl3ArrowsGray
xl3Flags
xl3Signs
xl3stars
simbolos xl3
xl3symbols2
xl3TrafficLights1

Nombre
xl3TrafficLights2
triángulos xl3
xl4rrows
xl4ArrowsGray
xl4CRV
xl4RedToBlack
xl4TrafficLights
xl5rrows
xl5ArrowsGray
xl5 casillas
xl5CRV
xl5 quarters



Tipo:

Nombre
xlConditionValuePercent
xlConditionValueNumber
xlConditionValuePercentile
xlConditionValueFormula

Operador:

Nombre	Valor
xlgrande	5
xlGreaterEqual	7

Valor:

Devuelve o establece el valor de umbral para un icono en un formato condicional.

Lea Formato condicional utilizando VBA en línea: https://riptutorial.com/es/excel-vba/topic/9912/formato-condicional-utilizando-vba

Capítulo 14: Funciones definidas por el usuario (UDF)

Sintaxis

1. **Function functionName (argumentVariable As dataType, argumentVariable2 As dataType, opcional argumentVariable3 As dataType) As functionReturnDataType**
   Declaración básica de una función. Cada función necesita un nombre, pero no tiene que tomar ningún argumento. Puede tomar 0 argumentos, o puede tomar un número dado de argumentos. También puede declarar un argumento como opcional (lo que significa que no importa si lo proporciona cuando llama a la función). Es una práctica recomendada proporcionar el tipo de variable para cada argumento, y de la misma manera, devolver el tipo de datos que la función misma devolverá.

2. **functionName = theVariableOrValueBeingReturned**
   Si viene de otros lenguajes de programación, puede estar acostumbrado a la palabra clave `Return` . Esto no se usa en VBA; en cambio, usamos el nombre de la función. Puede establecerlo en el contenido de una variable o en algún valor suministrado directamente. Tenga en cuenta que si estableció un tipo de datos para el retorno de la función, la variable o los datos que está proporcionando en este momento deben ser de ese tipo de datos.

3. **Función final**
   Obligatorio. Significa el final del Bloque de código de la `Function` y, por lo tanto, debe estar al final. El VBE normalmente proporciona esto automáticamente cuando creas una nueva función.

Observaciones

Una función definida por el usuario (también conocida como UDF) se refiere a una función específica de la tarea que ha sido creada por el usuario. Puede llamarse como una función de hoja de trabajo (por ejemplo, `=SUM(...)`) o usarse para devolver un valor a un proceso en ejecución en un procedimiento Sub. Una UDF devuelve un valor, normalmente a partir de la información que se le pasa como uno o más parámetros.

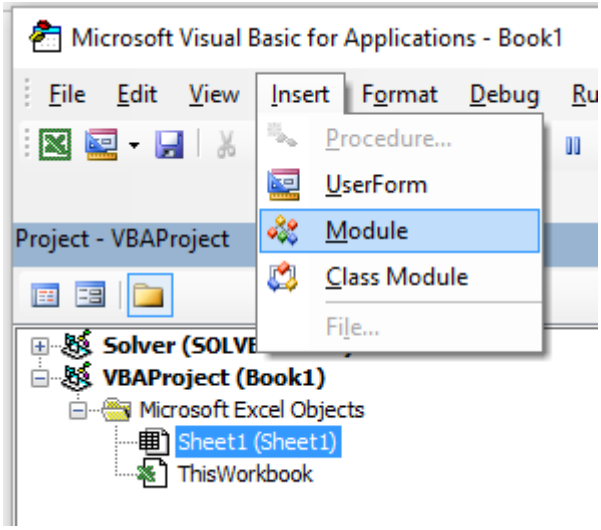
Puede ser creado por:

1. utilizando VBA.
2. utilizando Excel C API: creando un XLL que exporta funciones compiladas a Excel.
3. utilizando la interfaz COM.

Examples

UDF - Hola Mundo

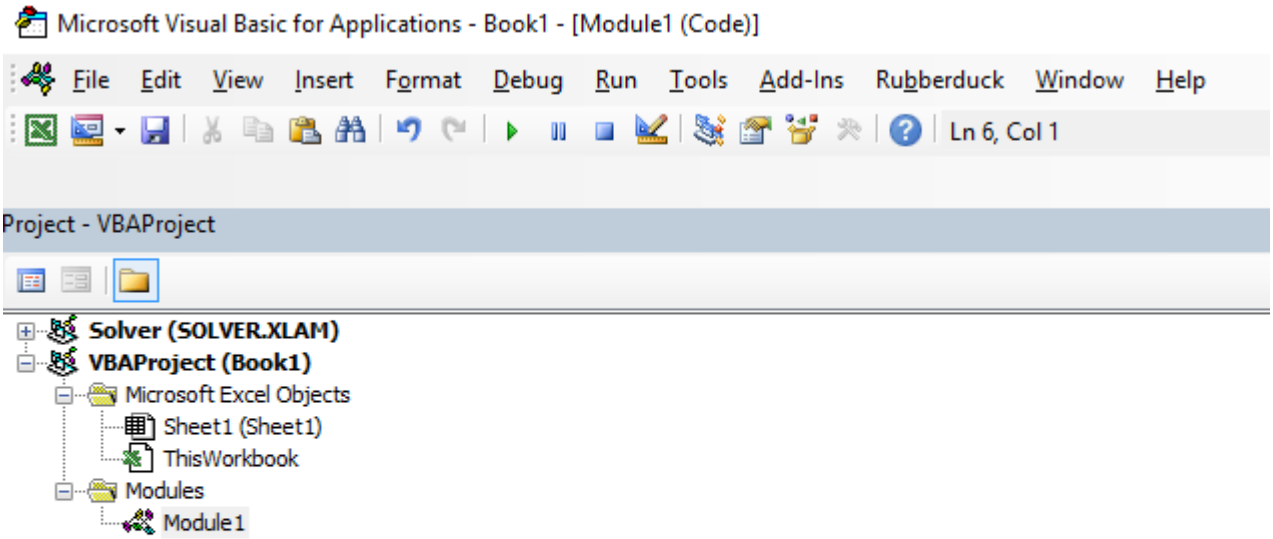
1. Abrir Excel
2. Abra el Editor de Visual Basic (vea Abrir el Editor de Visual Basic)
3. Agregue un nuevo módulo haciendo clic en Insertar -> Módulo:

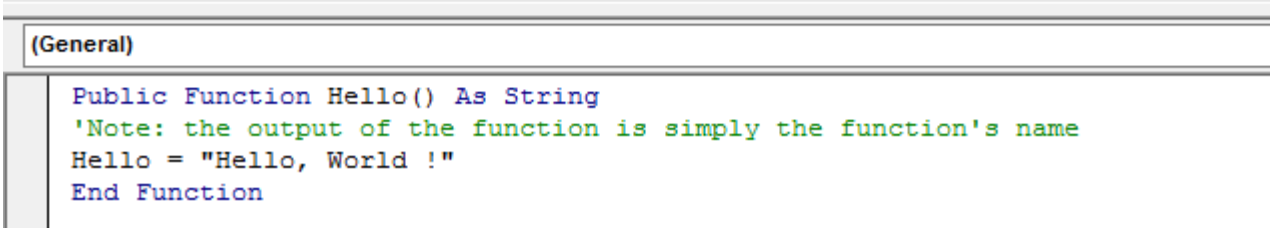


4. Copie y pegue el siguiente código en el nuevo módulo:

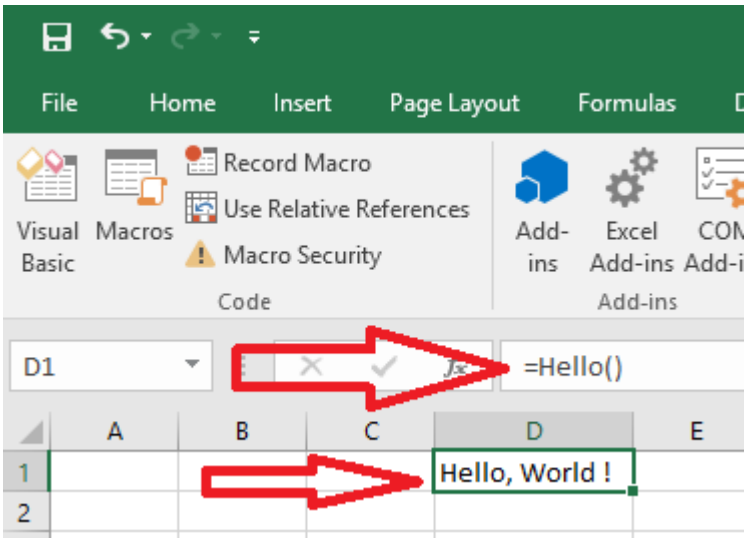
```
Public Function Hello() As String
'Note: the output of the function is simply the function's name
Hello = "Hello, World !"
End Function
```

Para obtener :





5. Vuelva a su libro de trabajo y escriba "= Hello ()" en una celda para ver "Hello World".



Permitir referencias de columnas completas sin penalización

Es más fácil implementar algunas UDF en la hoja de trabajo si se pueden pasar referencias de columnas completas como parámetros. Sin embargo, debido a la naturaleza explícita de la codificación, cualquier bucle que incluya estos rangos puede estar procesando cientos de miles de celdas que están completamente vacías. Esto reduce su proyecto de VBA (y libro de trabajo) a un desastre congelado mientras se procesan valores sin valor innecesarios.

Pasar por las celdas de una hoja de trabajo es uno de los métodos más lentos para realizar una tarea, pero a veces es inevitable. Cortar el trabajo realizado a lo que realmente se requiere tiene mucho sentido.

La solución es truncar las referencias de columna completa o fila completa a la propiedad Worksheet.UsedRange con el método Intersect . La siguiente muestra replicará de forma holgada la función SUMIF nativa de una hoja de trabajo, por lo que el *criterio de* rango de *criterios* también cambiará de tamaño para adaptarse al *sum_range,* ya que cada valor en el *sum_range* debe ir acompañado de un valor en el *criterio* .

El Application.Caller para un UDF utilizado en una hoja de cálculo es la celda en la que reside. La propiedad .Parent de la celda es la hoja de trabajo. Esto se utilizará para definir el .UsedRange.

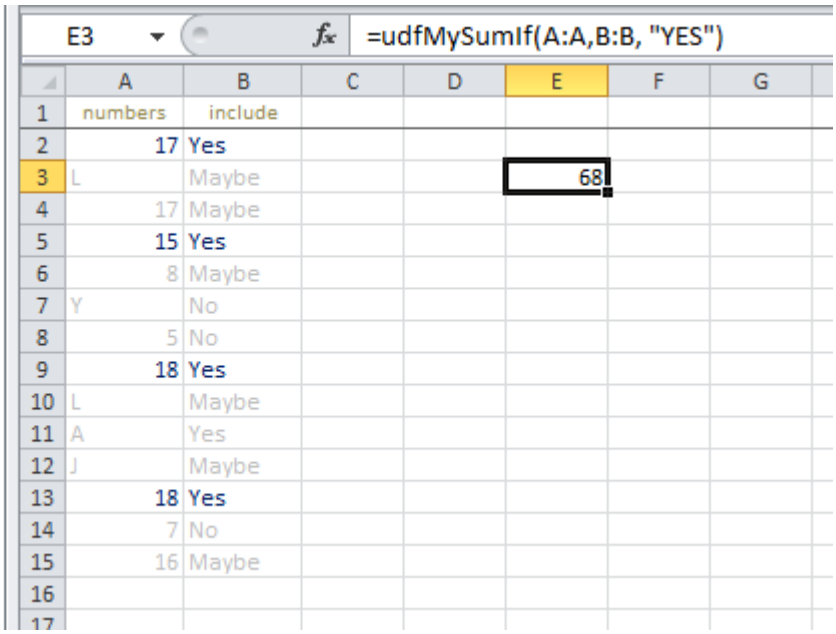
En una hoja de código de módulo:

```
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
```

Sintaxis:

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



Si bien este es un ejemplo bastante simplista, demuestra adecuadamente pasar dos referencias de columna completas (1,048,576 filas cada una) pero solo procesa 15 filas de datos y criterios.

Documentación oficial vinculada de MSDN de métodos y propiedades individuales, cortesía de Microsoft ™.

Contar valores únicos en rango

```
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
```

Colecciones

Lea Funciones definidas por el usuario (UDF) en línea: https://riptutorial.com/es/excel-

vba/topic/1070/funciones-definidas-por-el-usuario--udf-

Capítulo 15: Gamas nombradas

Introducción

El tema debe incluir información específicamente relacionada con los rangos con nombre en Excel, incluidos los métodos para crear, modificar, eliminar y acceder a los rangos con nombre definidos.

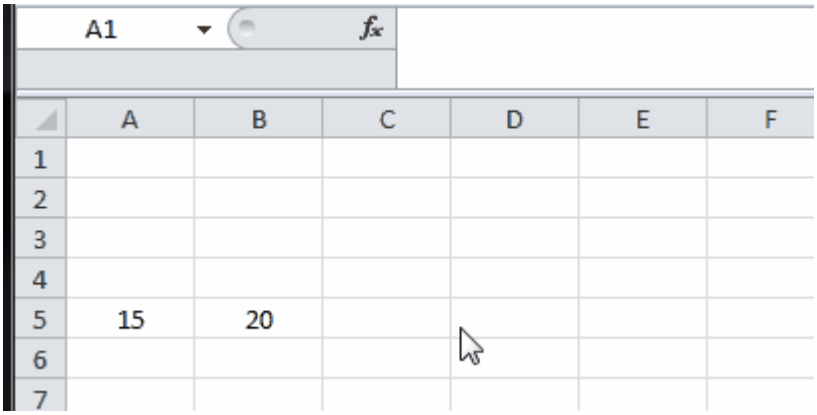
Examples

Definir un rango con nombre

El uso de rangos con nombre le permite describir el significado de los contenidos de una celda y usar este nombre definido en lugar de una dirección de celda real.

Por ejemplo, la fórmula `=A5*B5` se puede reemplazar con `=Width*Height` para que la fórmula sea mucho más fácil de leer y entender.

Para definir un nuevo rango con nombre, seleccione la celda o celdas para nombrar y luego escriba el nuevo nombre en el Cuadro de nombre junto a la barra de fórmulas.



> Nota: Los rangos con nombre están predeterminados en el ámbito global, lo que significa que se puede acceder a ellos desde cualquier lugar dentro del libro de trabajo. Las versiones anteriores de Excel permiten nombres duplicados, por lo que se debe tener cuidado para evitar nombres duplicados de alcance global, de lo contrario los resultados serán impredecibles. Use el Administrador de nombres de la pestaña Fórmulas para cambiar el alcance.

Usando gamas nombradas en VBA

Crear un nuevo rango denominado 'MyRange' asignado a la celda `A1`

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

Eliminar el rango definido por nombre

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

Acceso Rango Nombrado por nombre

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

Acceder a un rango con nombre con un acceso directo

Al igual que cualquier otro rango , se puede acceder directamente a los rangos con nombre mediante una notación de acceso directo que no requiere la creación de un objeto `Range` . Las tres líneas del extracto del código anterior se pueden reemplazar por una sola línea:

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

> Nota: la propiedad predeterminada para un rango es su valor, por lo que `[MyRange]` es el mismo que `[MyRange].Value`

También puede llamar a los métodos en el rango. Lo siguiente selecciona `MyRange` :

```
[MyRange].Select
```

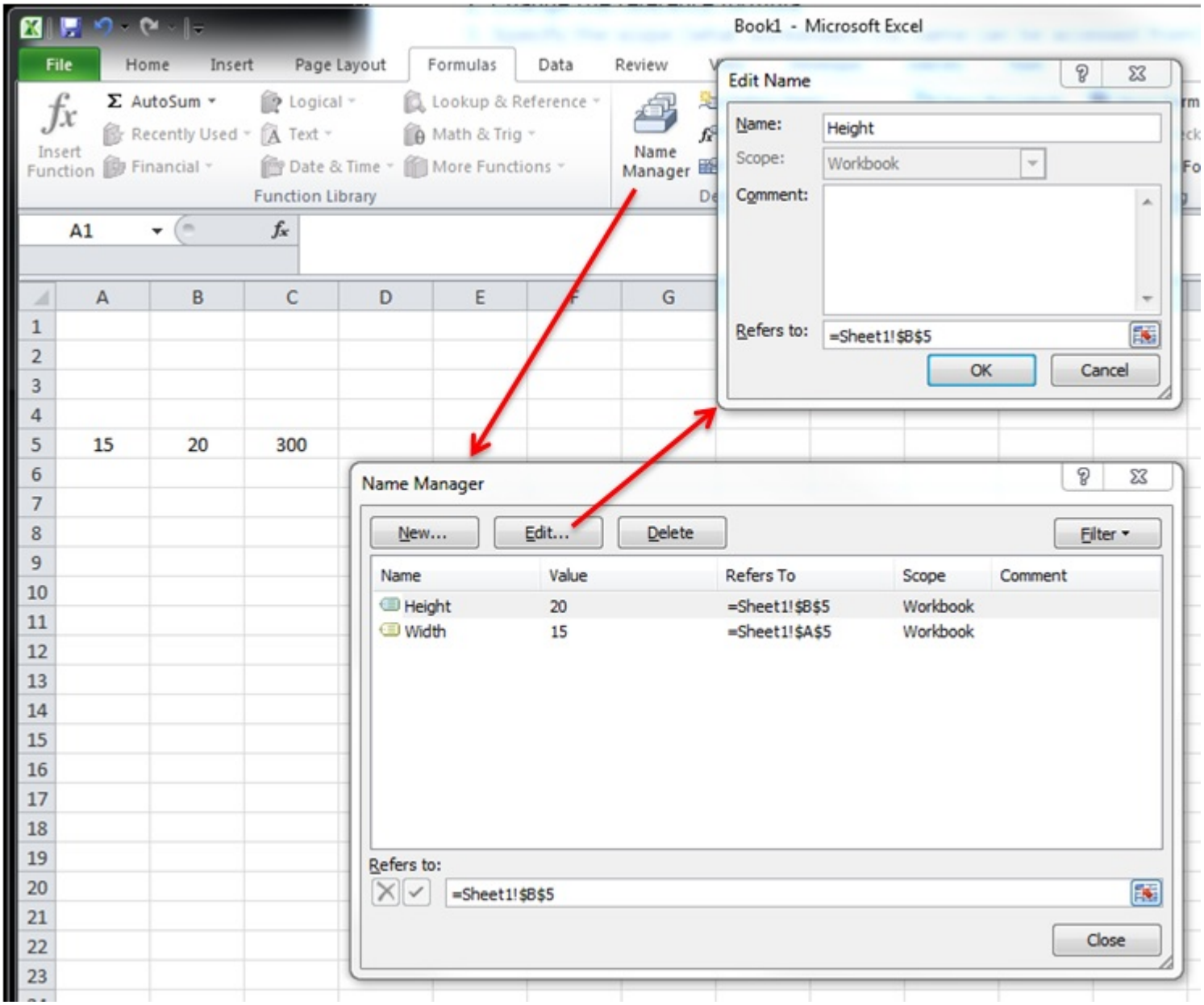
> Nota: una advertencia es que la notación de acceso directo no funciona con palabras que se usan en otras partes de la biblioteca de VBA. Por ejemplo, un rango llamado `Width` no sería accesible como `[Width]` pero funcionaría como se espera si se accede a través de `ThisWorkbook.Worksheets("Sheet1").Range("Width")`

Administrar rango (s) con nombre usando el administrador de nombres

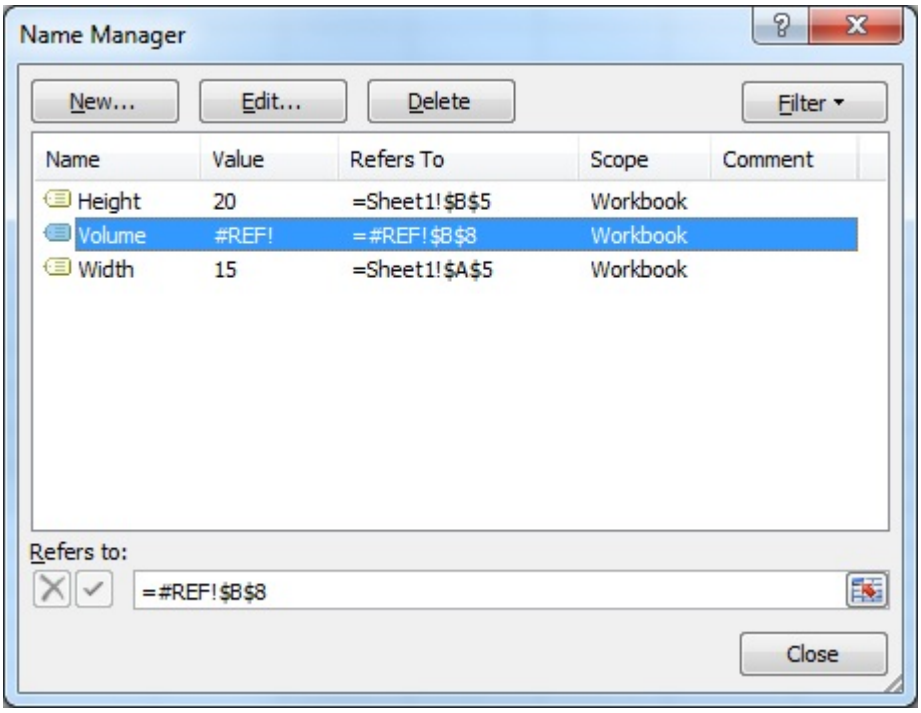
Pestaña Fórmulas> grupo Nombres definidos> botón Administrador de nombres

Named Manager le permite:

1. Crear o cambiar nombre
2. Crear o cambiar referencia de celda.
3. Crear o cambiar alcance
4. Eliminar el rango con nombre existente

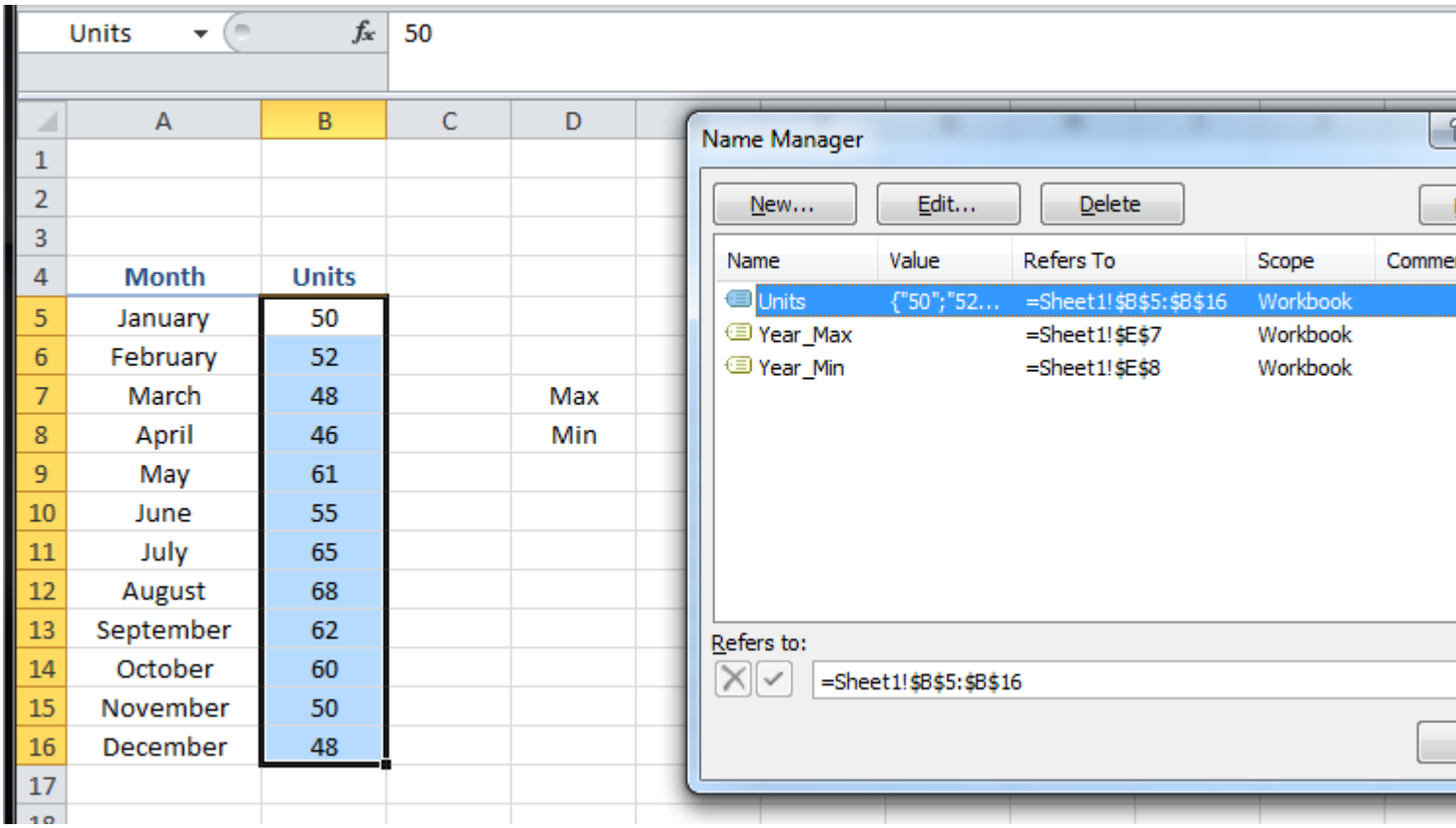


Named Manager proporciona una mirada rápida útil para enlaces rotos.



Arreglos de rango con nombre

Hoja de ejemplo



Código

```
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
```

Resultado

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			

Lea Gamas nombradas en línea: https://riptutorial.com/es/excel-vba/topic/8360/gamas-nombradas

Capítulo 16: Gamas y celulas

Sintaxis

- **Establecer** : el operador utilizado para establecer una referencia a un objeto, como un rango
- **Para cada** - El operador solía recorrer cada elemento de una colección.

Observaciones

Tenga en cuenta que los nombres de las variables `r` , `cell` y otros pueden nombrarse como desee, pero deben nombrarse adecuadamente para que el código sea más fácil de entender para usted y los demás.

Examples

Creando un rango

Un Rango no puede ser creado o poblado de la misma manera que una cadena:

```
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
```

Se considera una buena práctica calificar sus referencias , por lo que a partir de ahora usaremos el mismo enfoque aquí.
Más información sobre la creación de variables de objeto (por ejemplo, rango) en MSDN . Más sobre Set Statement en MSDN .

Hay diferentes maneras de crear el mismo rango:

```
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
```

Observe en el ejemplo que las celdas (2, 1) son equivalentes al rango ("A2"). Esto se debe a que las celdas devuelven un objeto Range.
Algunas fuentes: Chip Pearson-Cells Within Ranges ; Objeto de rango MSDN ; John Walkenback-Referente a rangos en su código de VBA .

También tenga en cuenta que en cualquier caso en que se use un número en la declaración del rango, y el número en sí esté fuera de comillas, como el rango ("A" y 2), puede intercambiar ese número por una variable que contenga un entero / largo Por ejemplo:

```
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 está utilizando bucles dobles, las celdas son mejores:

```
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
```

Maneras de referirse a una sola celda

La forma más sencilla de referirse a una sola celda en la hoja de cálculo de Excel actual es simplemente encerrar la forma A1 de su referencia entre corchetes:

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

Tenga en cuenta que los corchetes son solo azúcar sintáctica conveniente para el método `Evaluate` del objeto `Application` , por lo que técnicamente, esto es idéntico al siguiente código:

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

También puede llamar al método de `Cells` que toma una fila y una columna y devuelve una referencia de celda.

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

Recuerde que siempre que pase una fila y una columna a Excel desde VBA, la fila siempre es la primera, seguida de la columna, lo cual es confuso porque es lo opuesto a la notación `A1` común donde la columna aparece primero.

En ambos ejemplos, no especificamos una hoja de trabajo, por lo que Excel usará la hoja activa (la hoja que está al frente en la interfaz de usuario). Puede especificar la hoja activa explícitamente:

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

O puede proporcionar el nombre de una hoja en particular:

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

Hay una amplia variedad de métodos que se pueden utilizar para ir de un rango a otro. Por ejemplo, el método `Rows` se puede usar para llegar a las filas individuales de cualquier rango, y el método `Cells` se puede usar para llegar a las celdas individuales de una fila o columna, por lo que el siguiente código se refiere a la celda C1:

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

Guardar una referencia a una celda en una variable

Para guardar una referencia a una celda en una variable, debe usar la sintaxis `Set` , por ejemplo:

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

luego...

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

¿Por qué se requiere la palabra clave `Set` ? `Set` le dice a Visual Basic que el valor en el lado derecho de `=` está destinado a ser un objeto.

Propiedad compensada

- **Desplazamiento (filas, columnas)** : el operador utiliza para hacer referencia estáticamente a otro punto de la celda actual. A menudo se utiliza en bucles. Debe entenderse que los números positivos en la sección de filas se mueven a la derecha, mientras que los negativos se mueven a la izquierda. Con la sección de columnas, los positivos se mueven hacia abajo y los negativos se mueven hacia arriba.

es decir

```
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
```

Este código selecciona B2, coloca una nueva cadena allí, luego mueve esa cadena de nuevo a A1 después de eliminar B2.

Cómo Transponer Rangos (Horizontal a Vertical y viceversa)

```
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
```

Nota: Copy / PasteSpecial también tiene una opción Pegar Transposición que actualiza las fórmulas de las celdas transpuestas también.

Lea Gamas y celulas en línea: https://riptutorial.com/es/excel-vba/topic/1503/gamas-y-celulas

Capítulo 17: Gráficos y gráficos

Examples

Creando un gráfico con rangos y un nombre fijo

Los gráficos se pueden crear trabajando directamente con el objeto `Series` que define los datos del gráfico. Para llegar a la `Series` sin un gráfico existente, crea un objeto `ChartObject` en una `ChartObject` de `Worksheet` determinada y luego obtiene el objeto `Chart` de la misma. La ventaja de trabajar con el objeto `Series` es que puede establecer los `Values` y `Values XValues` consultando los objetos de `Range` . Estas propiedades de datos definirán correctamente la `Series` con referencias a esos rangos. La desventaja de este enfoque es que la misma conversión no se maneja al configurar el `Name` ; Es un valor fijo. No se ajustará con los datos subyacentes en el `Range` original. Verificando la fórmula `SERIES` y es obvio que el nombre es fijo. Esto debe ser manejado creando la fórmula `SERIES` directamente.

Código utilizado para crear el gráfico

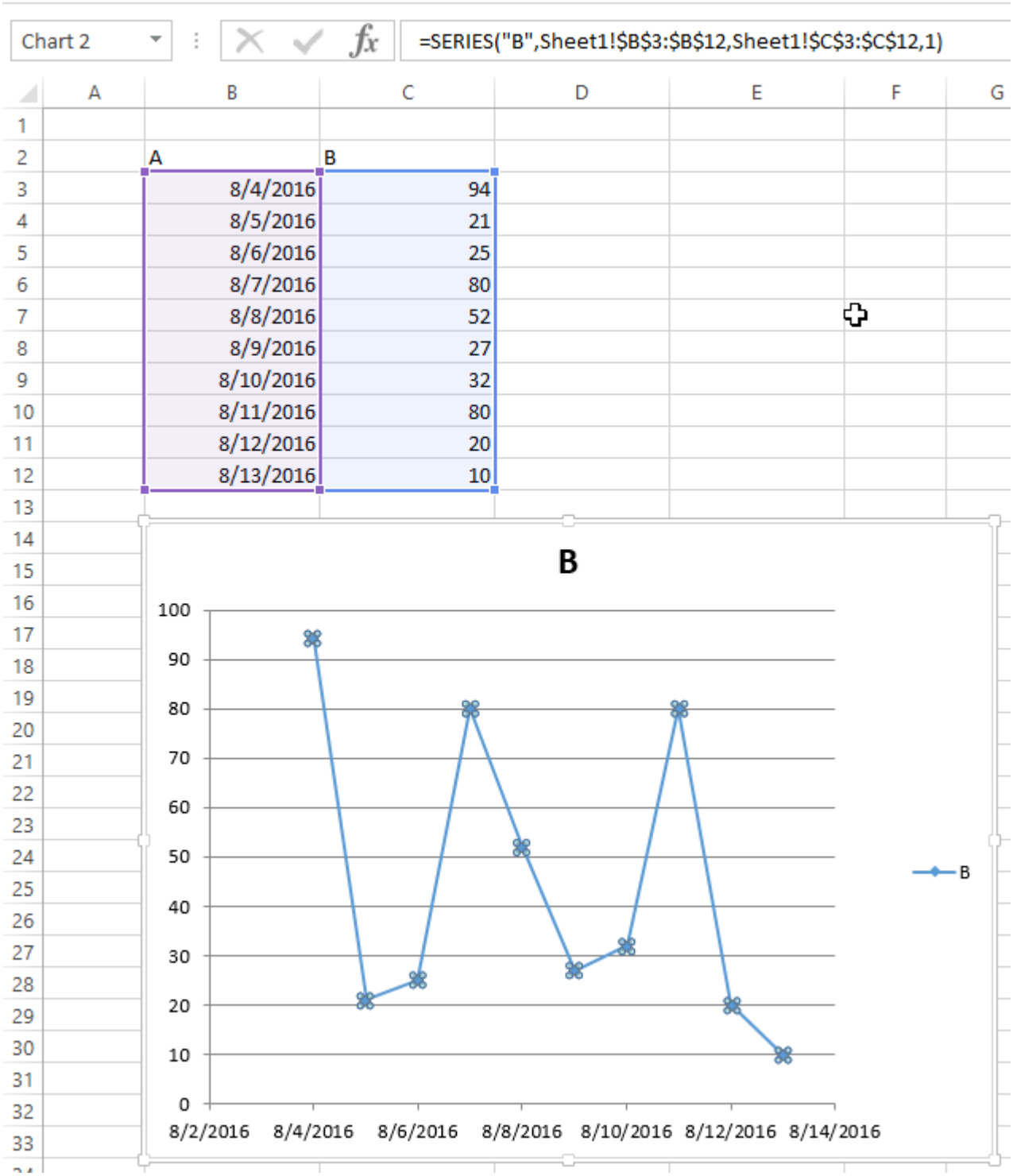
Tenga en cuenta que este código contiene declaraciones de variables adicionales para el `Chart` y la `Worksheet` . Estos pueden omitirse si no se utilizan. Sin embargo, pueden ser útiles si está modificando el estilo o cualquier otra propiedad del gráfico.

```
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
```

Datos / rangos originales y `Chart` resultante después de que se ejecuta el código

Tenga en cuenta que la fórmula `SERIES` incluye una `"B"` para el nombre de la serie en lugar de una referencia al `Range` que la creó.



Creando un gráfico vacío

El punto de partida para la gran mayoría de códigos de gráficos es crear un `Chart` vacío. Tenga en cuenta que esta `Chart` está sujeta a la plantilla de gráfica predeterminada que está activa y puede que no esté realmente vacía (si la plantilla se ha modificado).

La clave para el `ChartObject` es determinar su ubicación. La sintaxis de la llamada es `ChartObjects.Add(Left, Top, Width, Height)` . Una vez que se crea el objeto `ChartObject` , puede utilizar su objeto `Chart` para modificar realmente el gráfico. El objeto `ChartObject` comporta más como una `Shape` para colocar el gráfico en la hoja.

Código para crear un gráfico vacío

```
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
```

Gráfico resultante

Crea un gráfico modificando la fórmula SERIES

Para tener un control completo sobre un nuevo objeto `Chart` y `Series` (especialmente para un nombre de `Series` dinámico), debe recurrir a la modificación de la fórmula `SERIES` directamente. El proceso para configurar los objetos de `Range` es sencillo y el principal obstáculo es simplemente la creación de cadenas para la fórmula `SERIES` .

La fórmula `SERIES` toma la siguiente sintaxis:

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

Estos contenidos pueden suministrarse como referencias o como valores de matriz para los elementos de datos. `Order` representa la posición de la serie dentro del gráfico. Tenga en cuenta que las referencias a los datos no funcionarán a menos que estén completamente calificadas con el nombre de la hoja. Para ver un ejemplo de una fórmula de trabajo, haga clic en cualquier serie existente y verifique la barra de fórmulas.

Código para crear un gráfico y configurar datos usando la fórmula `SERIES`

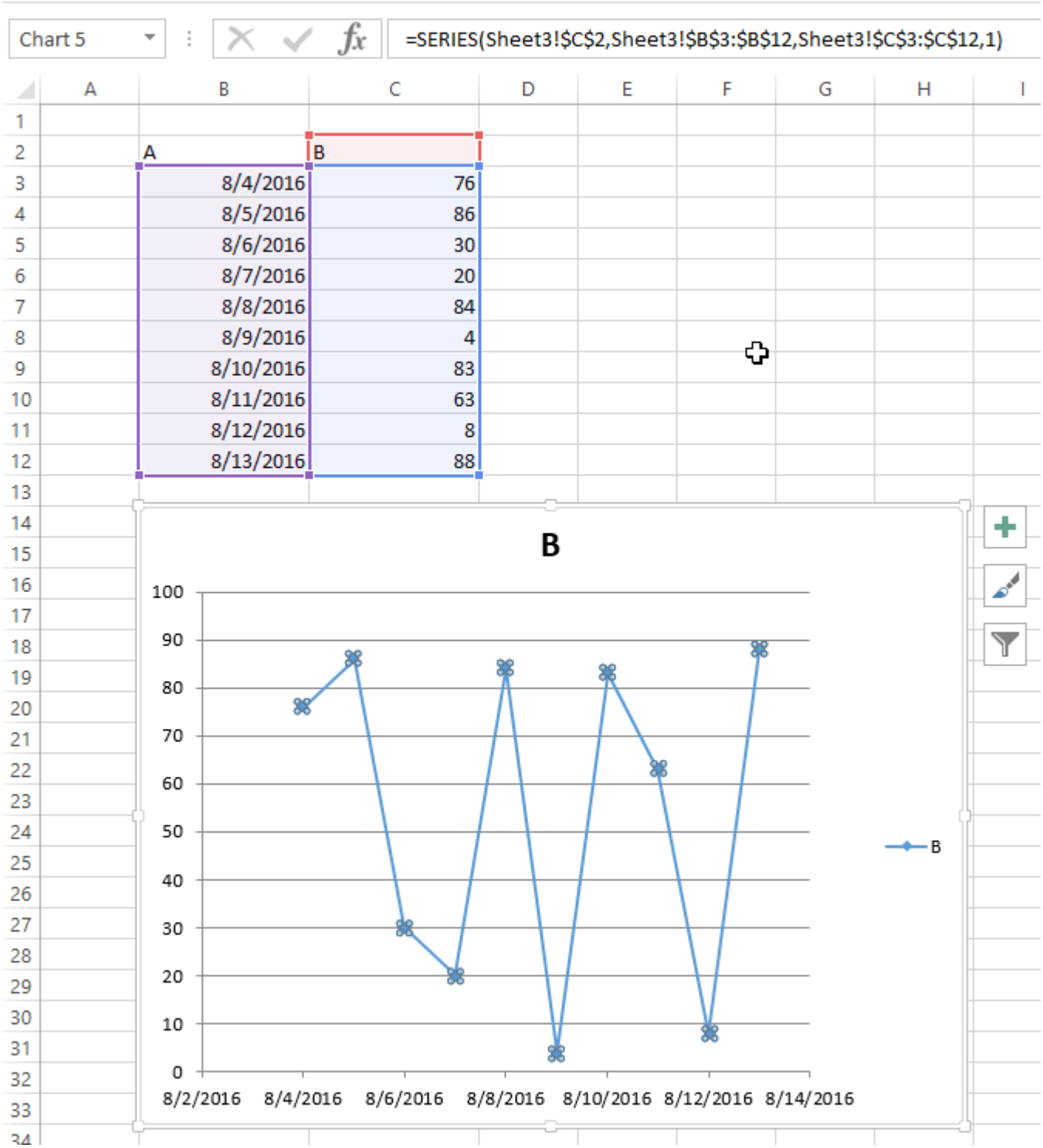
Tenga en cuenta que la creación de cadenas para crear la fórmula `SERIES` utiliza `.Address(,,,True)` . Esto garantiza que se utilice la referencia de rango *externa* para que se incluya una dirección completamente calificada con el nombre de la hoja. **Recibirá un error si se excluye el nombre de la hoja** .

```
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
```

Datos originales y cuadro resultante.

Tenga en cuenta que para este gráfico, el nombre de la serie se establece correctamente con un rango a la celda deseada. Esto significa que las actualizaciones se propagarán al `Chart` .



Organizar gráficos en una cuadrícula

Una tarea común con gráficos en Excel es estandarizar el tamaño y el diseño de múltiples gráficos en una sola hoja. Si lo hace manualmente, puede mantener presionada la `tecla ALT` mientras cambia el tamaño o mueve la tabla para "pegarse" a los límites de las celdas. Esto funciona para un par de gráficos, pero un enfoque de VBA es mucho más simple.

Código para crear una grilla

Este código creará una cuadrícula de gráficos que comienza en una posición dada (Arriba, Izquierda), con un número definido de columnas y un tamaño de gráfico común definido. Los

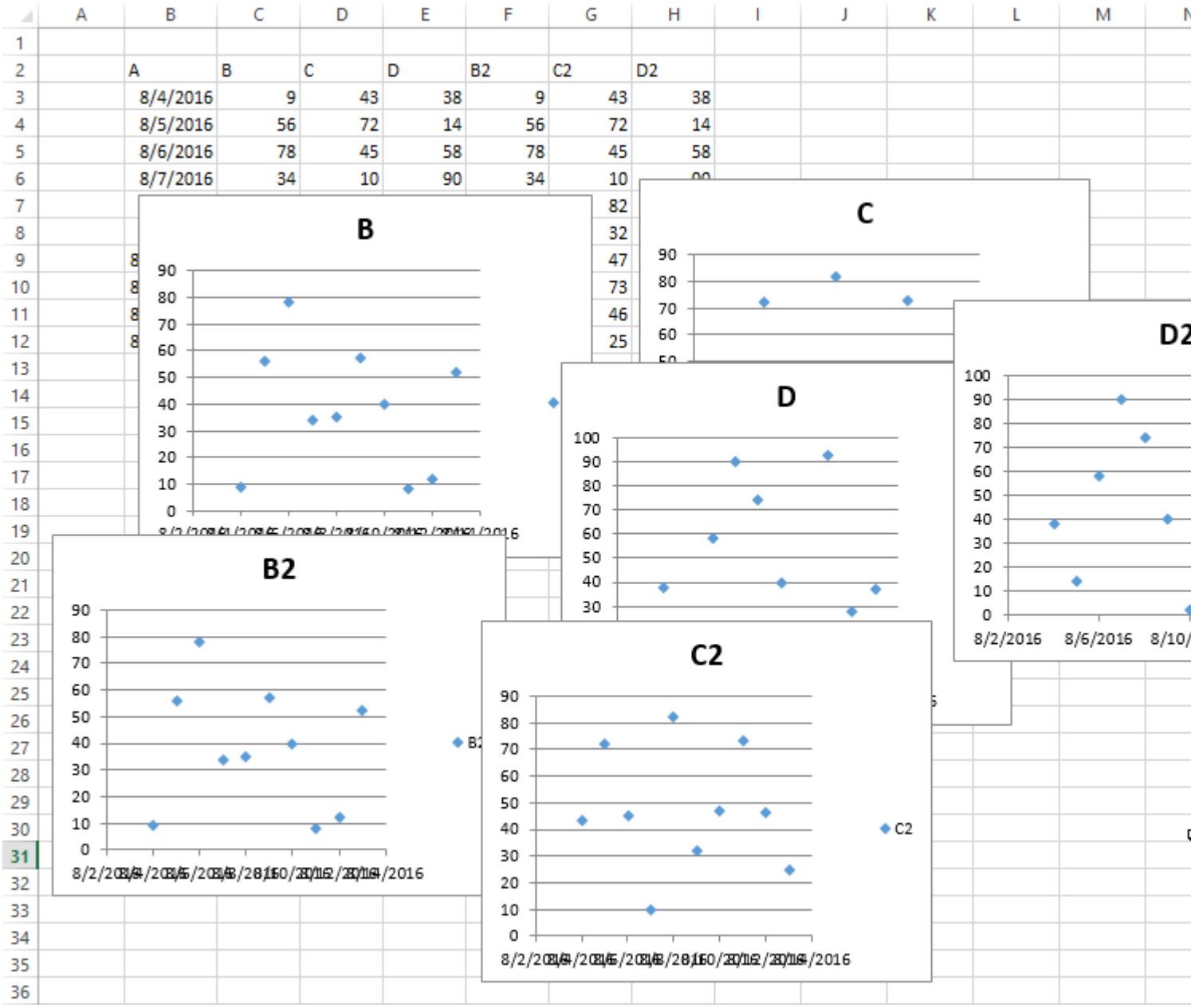
gráficos se colocarán en el orden en que se crearon y se ajustarán alrededor del borde para formar una nueva fila.

```
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
```

Resultado con varios gráficos

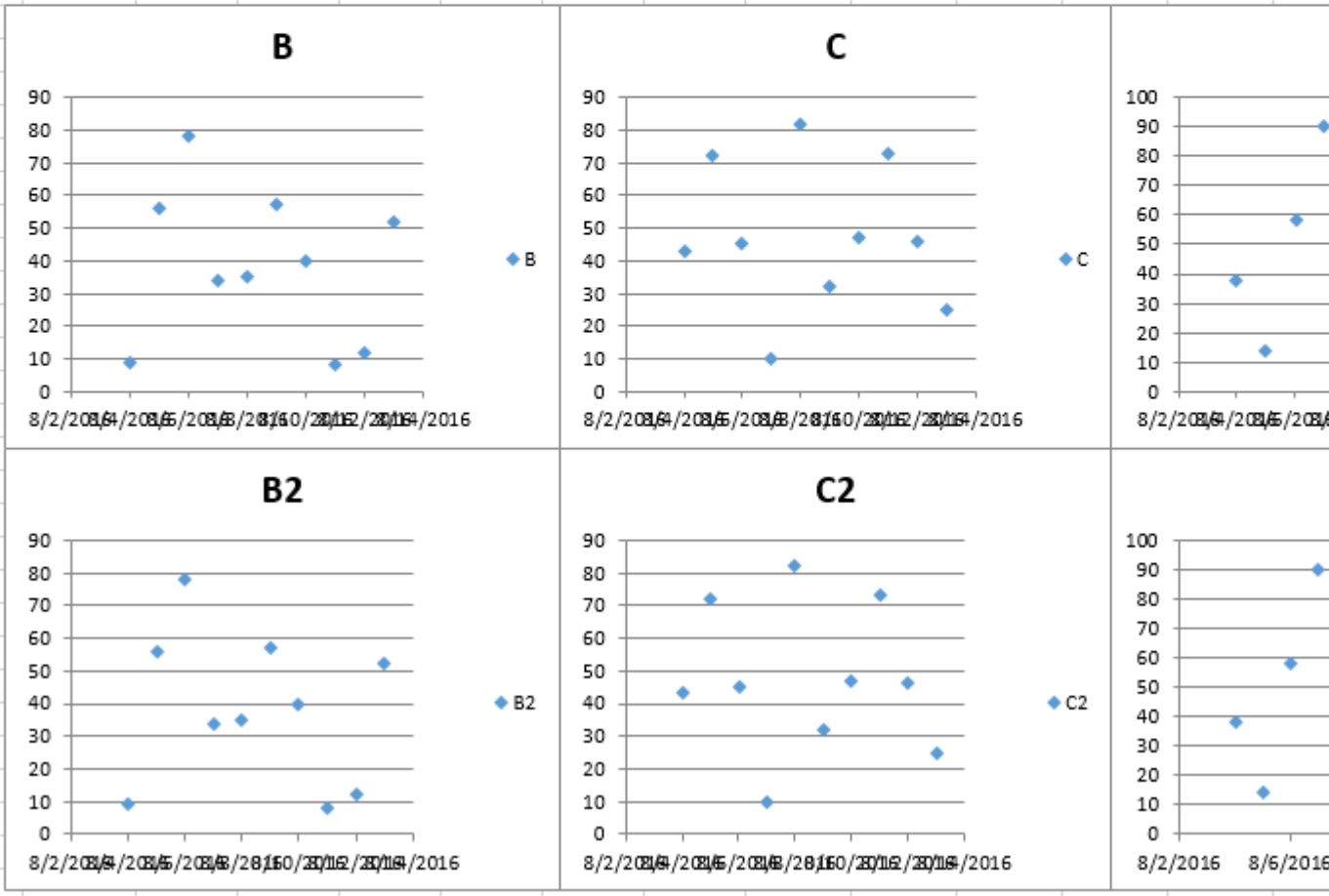
Estas imágenes muestran el diseño aleatorio original de los gráficos y la cuadrícula resultante de ejecutar el código anterior.

antes de


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

Después

	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



Lea Gráficos y gráficos en línea: https://riptutorial.com/es/excel-vba/topic/4968/graficos-y-graficos

Capítulo 18: Integración de PowerPoint a través de VBA

Observaciones

Esta sección muestra una variedad de formas de interactuar con PowerPoint a través de VBA. Desde mostrar datos en diapositivas hasta crear gráficos, PowerPoint es una herramienta muy poderosa cuando se usa junto con Excel. Por lo tanto, esta sección busca demostrar las diversas formas en que se puede utilizar VBA para automatizar esta interacción.

Examples

Lo básico: Lanzar PowerPoint desde VBA

Si bien hay muchos parámetros que se pueden cambiar y variaciones que se pueden agregar según la funcionalidad deseada, este ejemplo presenta el marco básico para lanzar PowerPoint.

> **Nota:** este código requiere que la referencia de PowerPoint se haya agregado al Proyecto VBA activo. Consulte la entrada Documentación de referencias para saber cómo habilitar la referencia.

Primero, defina las variables para la aplicación, presentación y objetos de diapositiva. Si bien esto se puede hacer con la vinculación tardía, siempre es mejor usar la vinculación temprana cuando corresponda.

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

A continuación, abra o cree una nueva instancia de la aplicación de PowerPoint. Aquí, la llamada `On Error Resume Next` se usa para evitar que `GetObject` un error si aún no se ha abierto PowerPoint. Consulte el ejemplo de manejo de errores del tema de Mejores prácticas para obtener una explicación más detallada.

```
'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
```

Una vez iniciada la aplicación, se genera una nueva presentación y una diapositiva contenida para su uso.

```
'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
```

Al completar este código, se abrirá una nueva ventana de PowerPoint con una diapositiva en blanco. Al usar las variables de objetos, se pueden agregar formas, texto, gráficos y rangos de Excel como se desee

Lea Integración de PowerPoint a través de VBA en línea: https://riptutorial.com/es/excel-vba/topic/2327/integracion-de-powerpoint-a-traves-de-vba

Capítulo 19: Localización de valores duplicados en un rango

Introducción

En ciertos puntos, estará evaluando un rango de datos y necesitará ubicar los duplicados en ellos. Para conjuntos de datos más grandes, hay una serie de enfoques que puede utilizar que utilizan el código VBA o funciones condicionales. Este ejemplo utiliza una condición simple if-then dentro de dos bucles anidados para ver si cada celda del rango tiene el mismo valor que cualquier otra celda del rango.

Examples

Encuentra duplicados en un rango

Las siguientes pruebas van del A2 al A7 para valores duplicados. **Observación:** este ejemplo ilustra una posible solución como un primer acercamiento a una solución. Es más rápido usar una matriz que un rango y uno podría usar colecciones o diccionarios o métodos xml para verificar si hay duplicados.

```
    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
```

Dependiendo de sus necesidades, el ejemplo puede modificarse; por ejemplo, el límite superior de n puede ser el valor de la fila de la última celda con datos en el rango, o la acción en caso de una condición True If puede editarse para extraer el duplicado valor en otro lugar. Sin embargo, la mecánica de la rutina no cambiaría.

Lea Localización de valores duplicados en un rango en línea: https://riptutorial.com/es/excel-vba/topic/8295/localizacion-de-valores-duplicados-en-un-rango

Capítulo 20: Mejores Prácticas VBA

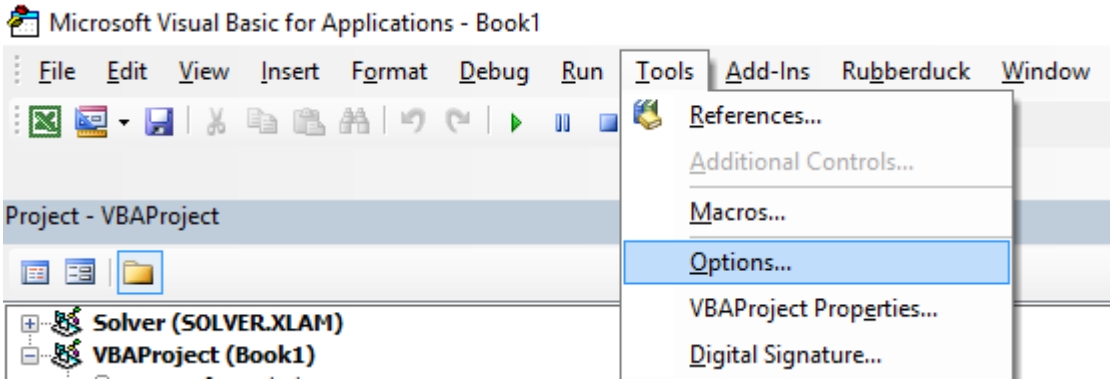
Observaciones

Todos los conocemos, pero estas prácticas son mucho menos obvias para alguien que comienza a programar en VBA.

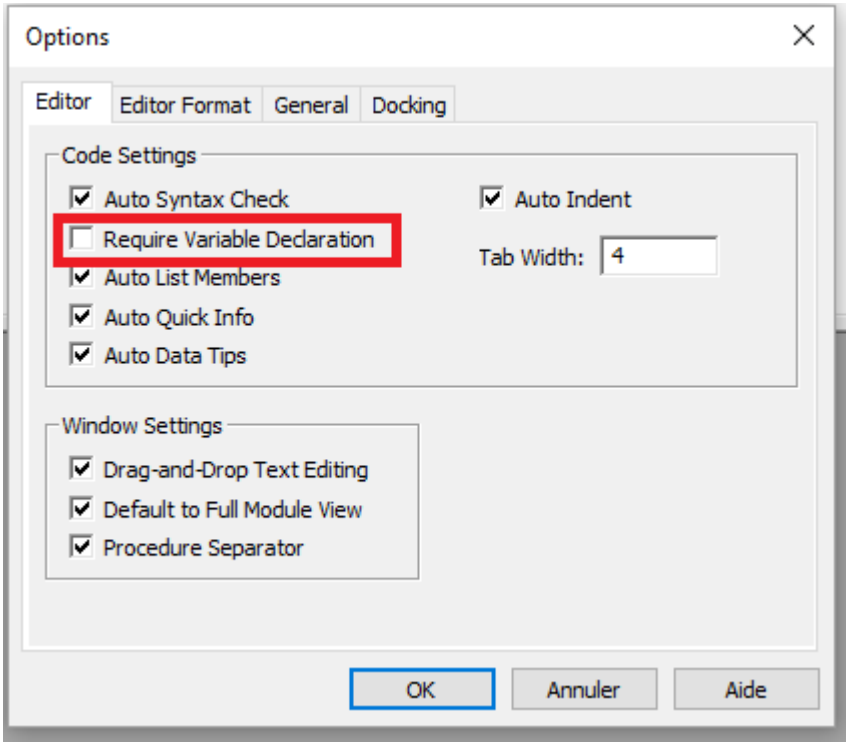
Examples

SIEMPRE use "Opción explícita"

En la ventana del editor de VBA, en el menú Herramientas, seleccione "Opciones":



Luego, en la pestaña "Editor", asegúrese de que la opción "Requerir declaración variable" esté marcada:



La selección de esta opción colocará automáticamente `Option Explicit` en la parte superior de cada módulo VBA.

> **Nota pequeña:** Esto es cierto para los módulos, módulos de clase, etc. que no se han abierto hasta ahora. Por lo tanto, si ya vio el código de la `Sheet1` antes de activar la opción "Requerir Declaración Variable", ¡ `Option Explicit` no se agregará!

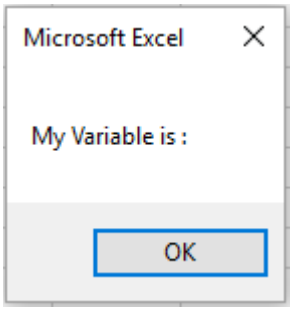
`Option Explicit` requiere que cada variable se defina antes de su uso, por ejemplo, con una instrucción `Dim` . Sin la `Option Explicit` habilitada, el compilador VBA asumirá que cualquier palabra no reconocida es una nueva variable del tipo `Variant` , lo que causa errores extremadamente difíciles de detectar relacionados con errores tipográficos. Con `Option Explicit` habilitado, cualquier palabra no reconocida provocará que se genere un error de compilación, que indica la línea ofensiva.

Ejemplo:

Si ejecuta el siguiente código:

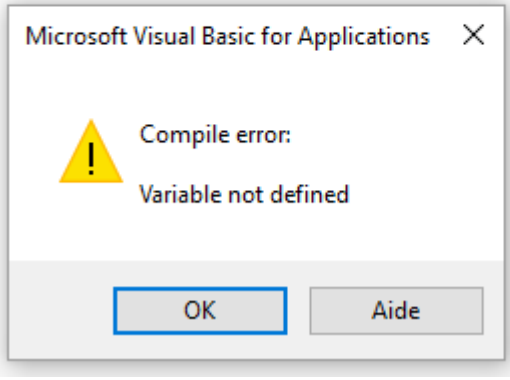
```
Sub Test()
  my_variable = 12
  MsgBox "My Variable is : " & myvariable
End Sub
```

Recibirás el siguiente mensaje:



Ha cometido un error al escribir `myvariable` lugar de `my_variable` , luego el cuadro de mensaje muestra una variable vacía. Si usa `Option Explicit` , este error no es posible porque recibirá un mensaje de error de compilación que indica el problema.

```
Option Explicit

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

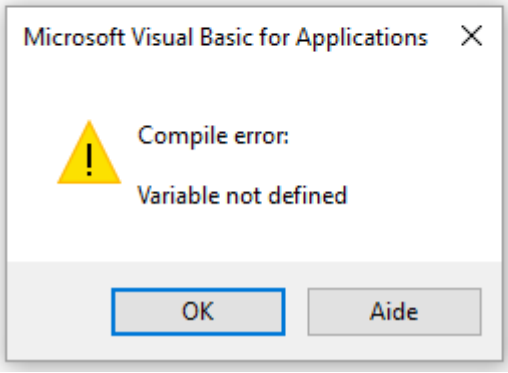


Ahora si agrega la declaración correcta:

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

Obtendrá un mensaje de error que indica precisamente el error con `myvariable` :

```
Option Explicit

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



Nota sobre las opciones explícitas y las matrices (declarar una matriz dinámica):

Puede usar la declaración ReDim para declarar una matriz implícitamente dentro de un procedimiento.

- Tenga cuidado de no escribir mal el nombre de la matriz cuando use la

instrucción ReDim

- Incluso si la instrucción Option Explicit se incluye en el módulo, se creará una nueva matriz

```
Dim arr() as Long

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

Trabajar con matrices, no con rangos

Office Blog - Excel VBA Rendimiento de las mejores prácticas de codificación

A menudo, el mejor rendimiento se logra evitando el uso de `Range` tanto como sea posible. En este ejemplo, leemos en un objeto de `Range` completo en una matriz, cuadramos cada número en la matriz y luego devolvemos la matriz al `Range` . Esto accede al `Range` solo dos veces, mientras que un bucle lo haría 20 veces para las lecturas / escrituras.

```
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
```

Se pueden encontrar más consejos e información con ejemplos cronometrados en UBAs VBA eficientes de escritura de Charles Williams (Parte 1) y otros artículos de la serie .

Use constantes de VB cuando estén disponibles

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

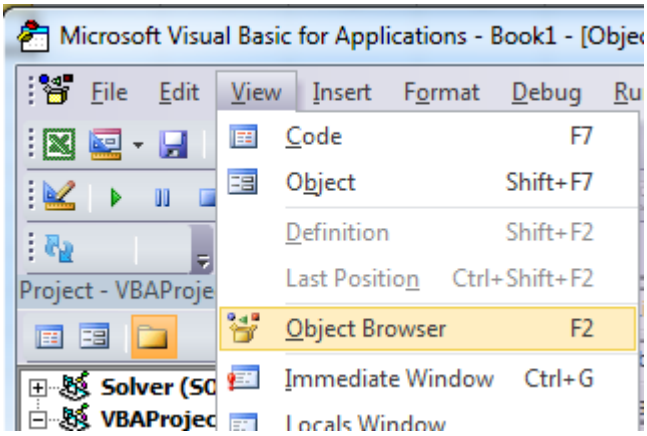
se puede utilizar en lugar de

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

con el fin de mejorar la legibilidad.

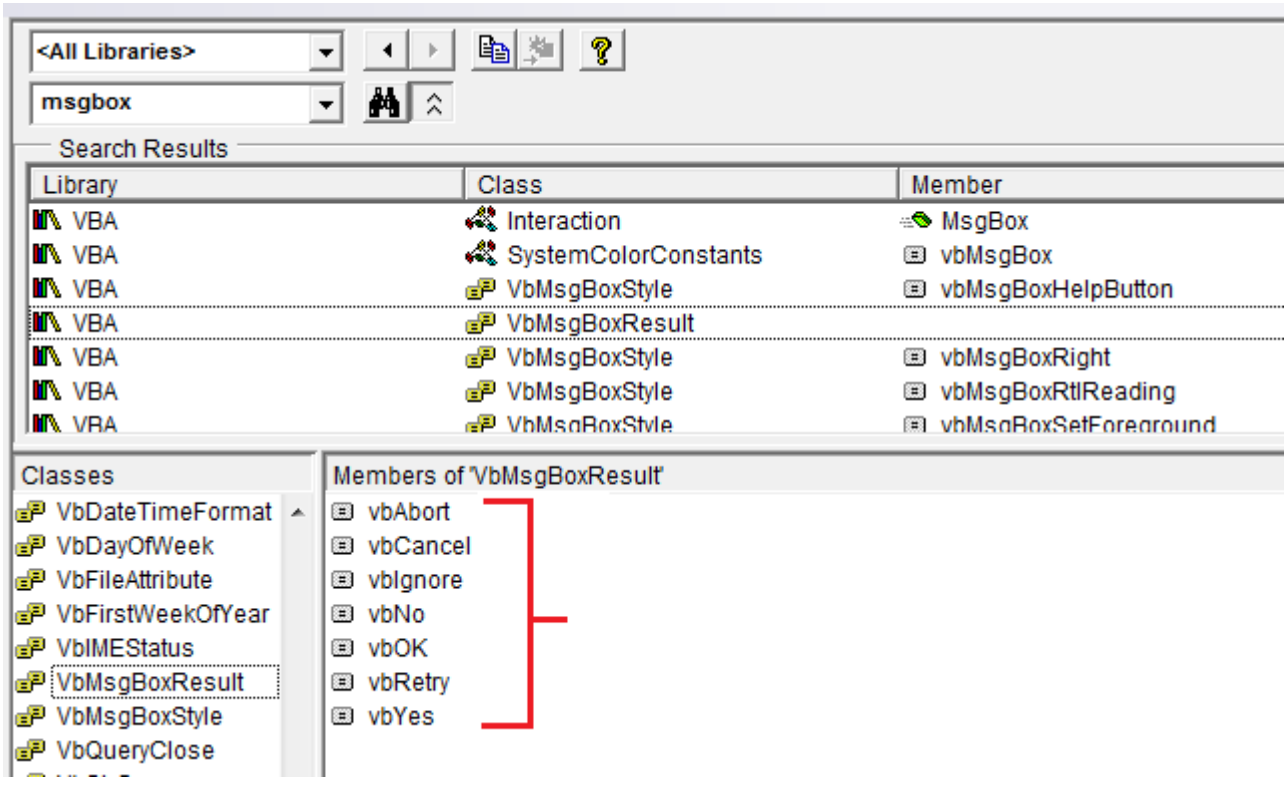
Utilice el *buscador de objetos* para encontrar las constantes de VB disponibles. *Ver → Object*

Browser o `F2` desde VB Editor.



Entrar en clase para buscar

Ver miembros disponibles



Utilizar nombres descriptivos de variables.

Los nombres descriptivos y la estructura en su código ayudan a hacer comentarios innecesarios

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

ductArea = ductWidth * ductHeight
```

es mejor que

```
Dim a, w, h

a = w * h
```

Esto es especialmente útil cuando está copiando datos de un lugar a otro, ya sea una celda, rango, hoja de trabajo o libro de trabajo. Ayúdate usando nombres como estos:

```
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 declara múltiples variables en una línea, asegúrese de especificar un tipo para *cada* variable como:

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

Lo siguiente solo declarará la última variable y las primeras seguirán siendo `Variant` :

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

Manejo de errores

El buen manejo de errores evita que los usuarios finales vean los errores de tiempo de ejecución de VBA y ayuda al desarrollador a diagnosticar y corregir errores fácilmente.

Existen tres métodos principales de manejo de errores en VBA, dos de los cuales deben evitarse para los programas distribuidos a menos que el código lo requiera específicamente.

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

o

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

Prefiero usar:

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

En Error GoTo 0

Si no se establece ningún manejo de errores en su código, `On Error GoTo 0` es el controlador de errores predeterminado. En este modo, cualquier error de tiempo de ejecución iniciará el mensaje de error típico de VBA, permitiéndole finalizar el código o ingresar al modo de `debug` , identificando la fuente. Al escribir código, este método es el más simple y útil, pero siempre debe evitarse para el código que se distribuye a los usuarios finales, ya que este método es muy desagradable y difícil de entender para los usuarios finales.

En error reanudar siguiente

`On Error Resume Next` hará que VBA ignore cualquier error que se lance en el tiempo de ejecución para todas las líneas que siguen a la llamada de error hasta que se haya cambiado el controlador de errores. En casos muy específicos, esta línea puede ser útil, pero debe evitarse fuera de estos casos. Por ejemplo, al iniciar un programa separado desde una macro de Excel, la llamada `On Error Resume Next` puede ser útil si no está seguro de si el programa ya está abierto:

```
'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 no hubiésemos utilizado la llamada `On Error Resume Next` y la aplicación de Powerpoint no estuviera abierta, el método `GetObject` generaría un error. Por lo tanto, `On Error Resume Next` fue necesario para evitar crear dos instancias de la aplicación.

Nota: también es una buena práctica restablecer *inmediatamente* el controlador de errores tan pronto como ya no necesite la llamada `On Error Resume Next`

On Error GoTo <line>

Este método de manejo de errores se recomienda para todo el código que se distribuye a otros usuarios. Esto le permite al programador controlar exactamente cómo VBA maneja un error al enviar el código a la línea especificada. La etiqueta se puede rellenar con cualquier cadena (incluidas las cadenas numéricas) y enviará el código a la cadena correspondiente que va

seguida de dos puntos. Se pueden utilizar múltiples bloques de manejo de errores haciendo diferentes llamadas de `On Error GoTo <line>` . La subrutina a continuación muestra la sintaxis de una llamada `On Error GoTo <line>` .

Nota: es esencial que la línea de `Exit Sub` se coloque sobre el primer controlador de errores y antes de cada controlador de errores posterior para evitar que el código progrese naturalmente en el bloque *sin que* se llame un error. Por lo tanto, es una práctica recomendada para la función y la legibilidad colocar controladores de errores al final de un bloque de código.

```
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 sale de su método con su código de manejo de errores, asegúrese de limpiar:

- Deshacer todo lo que está parcialmente completado
- Cerrar archivos
- Restablecer actualización de pantalla
- Restablecer el modo de cálculo
- Restablecer eventos
- Restablecer el puntero del mouse
- Llame al método de descarga en instancias de objetos, que persisten después del `End Sub`
- Restablecer barra de estado

Documenta tu trabajo

Es una buena práctica documentar su trabajo para su uso posterior, especialmente si está codificando para una carga de trabajo dinámica. Los buenos comentarios deben explicar por qué el código está haciendo algo, no lo que está haciendo el código.

```
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 su código es tan oscuro que requiere comentarios para explicar lo que está haciendo, considere volver a escribirlo para que sea más claro en lugar de explicarlo a través de comentarios. Por ejemplo, en lugar 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
```

Aclare el código para que sea más fácil de seguir, como por ejemplo:

```
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
```

Desactivar propiedades durante la ejecución de macro

Es la mejor práctica en cualquier lenguaje de programación para **evitar una optimización prematura.** Sin embargo, si las pruebas revelan que su código se está ejecutando muy lentamente, puede ganar algo de velocidad desactivando algunas de las propiedades de la aplicación mientras se ejecuta. Agregue este código a un módulo estándar:

```
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
```

Más información en Office Blog: Excel VBA Performance Coding Besting Practices

Y solo llámelo al principio y al final de las 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
```

Si bien estas pueden considerarse en gran parte "mejoras" para los procedimientos regulares de `Public Sub` , deshabilitar el manejo de eventos con `Application.EnableEvents = False` debe considerar obligatorio para las macros de eventos privados `Worksheet_Change` y `Workbook_SheetChange` que cambian los valores en una o más hojas de trabajo. Si no se deshabilitan los activadores de eventos, la macro del evento se ejecutará de forma recursiva sobre sí misma cuando un valor cambie y puede llevar a un libro "congelado". Recuerde volver a activar los eventos antes de abandonar la macro de eventos, posiblemente a través de un controlador de errores de "salida segura".

```
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
```

Una advertencia: aunque la desactivación de esta configuración mejorará el tiempo de ejecución, puede hacer que la depuración de la aplicación sea mucho más difícil. Si su código *no* funciona correctamente, comente la llamada `SpeedUp True` hasta que descubra el problema.

Esto es particularmente importante si está escribiendo en celdas en una hoja de trabajo y luego leyendo los resultados calculados de las funciones de la hoja de trabajo, ya que `xlCalculationManual` evita que se `xlCalculationManual` el libro de trabajo. Para evitar esto sin deshabilitar `SpeedUp` , es posible que desee incluir `Application.Calculate` para ejecutar un cálculo en puntos específicos.

NOTA: ya que estas son propiedades de la `Application` sí misma, debe asegurarse de que estén habilitadas nuevamente antes de que salga su macro. Esto hace que sea particularmente importante usar controladores de errores y evitar múltiples puntos de salida (es decir, `End` o `Unload Me`).

Con manejo de errores:

```
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
```

Evite usar ActiveCell o ActiveSheet en Excel

El uso de `ActiveCell` o `ActiveSheet` puede ser fuente de errores si (por algún motivo) el código se ejecuta en el lugar equivocado.

```
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"
```

- El uso de `Active*` puede crear problemas en macros de larga ejecución si su usuario se aburre y hace clic en otra hoja de trabajo o abre otro libro.
- Puede crear problemas si su código se abre o crea otro libro de trabajo.
- Puede crear problemas si su código utiliza `Sheets("MyOtherSheet").Select` y ha olvidado en qué hoja estaba antes de comenzar a leer o escribir en él.

Nunca asuma la hoja de trabajo

Incluso cuando todo su trabajo se dirige a una sola hoja de trabajo, es una buena práctica especificar explícitamente la hoja de trabajo en su código. Este hábito hace que sea mucho más fácil expandir su código más adelante, o levantar partes (o todas) de una `Sub` o `Function` para ser reutilizadas en otro lugar. Muchos desarrolladores establecen el hábito de (re) usar el mismo nombre de variable local para una hoja de trabajo en su código, lo que hace que la reutilización de ese código sea aún más sencilla.

Como ejemplo, el siguiente código es ambiguo, ¡pero funciona! - mientras el desarrollador no active o cambie a una hoja de trabajo diferente:

```
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` está activa, entonces la celda A1 en la Hoja1 se llenará con la fecha y hora actuales. Pero si el usuario cambia las hojas de trabajo por cualquier motivo, entonces el código actualizará cualquier cosa que la hoja de trabajo esté actualmente activa. La hoja de trabajo de destino es ambigua.

La mejor práctica es identificar siempre a qué hoja de cálculo se refiere su código:

```
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
```

El código anterior es claro para identificar tanto el libro de trabajo como la hoja de trabajo. Si bien puede parecer una exageración, crear un buen hábito con respecto a las referencias de destino lo salvará de problemas futuros.

Evite utilizar SELECT o ACTIVAR

Es **muy** raro que alguna vez quiera usar `Select` o `Activate` en su código, pero algunos métodos de Excel requieren que se active una hoja de cálculo o libro antes de que funcionen como se espera.

Si recién está comenzando a aprender VBA, a menudo se le sugerirá que grabe sus acciones utilizando la grabadora de macros, luego observe el código. Por ejemplo, registré las acciones tomadas para ingresar un valor en la celda D3 en la Hoja2, y el código de macro se ve así:

```
Option Explicit
Sub Macro1()
```

```
'
' Macro1 Macro
'

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

Sin embargo, recuerde que la grabadora de macros crea una línea de código para CADA de sus acciones (de usuario). Esto incluye hacer clic en la pestaña de la hoja de trabajo para seleccionar Hoja2 (`Sheets("Sheet2").Select`), hacer clic en la celda D3 antes de ingresar el valor (`Range("D3").Select`), y usar la tecla Intro (que es efectivamente " seleccionando "la celda debajo de la celda seleccionada actualmente: `Range("D4").Select`).

Hay varios problemas con el uso de `.Select` aquí:

- **La hoja de trabajo no siempre se especifica.** Esto sucede si no cambia las hojas de trabajo durante la grabación, y significa que el código producirá resultados diferentes para diferentes hojas de trabajo activas.
- **`.Select()` es lento.** Incluso si `Application.ScreenUpdating` se establece en `False` , esta es una operación innecesaria para ser procesada.
- **`.Select()` es ingobernable.** Si `Application.ScreenUpdating` se deja en `True` , Excel seleccionará realmente las celdas, la hoja de trabajo, el formulario ... sea lo que sea con lo que esté trabajando. Esto es estresante para los ojos y realmente desagradable de ver.
- **`.Select()` activará oyentes.** Esto ya es un poco avanzado, pero a menos que se trabaje alrededor, se activarán funciones como `Worksheet_SelectionChange()` .

El momento de codificar en VBA, todas las acciones de "escribir" (es decir `Select` estados) ya no son necesarios. Su código puede reducirse a una sola declaración para poner el valor en la celda:

```
'--- 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
```

(El ejemplo MEJOR anterior muestra el uso de variables intermedias para separar diferentes partes de la referencia de celda. El ejemplo BUENO siempre funcionará bien, pero puede ser muy engorroso en módulos de código mucho más largos y más difícil de depurar si una de las referencias está mal escrita).)

** NOTA: la grabadora de macros hace muchas suposiciones sobre el tipo de datos que está ingresando, en este caso ingresando un valor de cadena como una fórmula para crear el valor. Su código no tiene que hacer esto y simplemente puede asignar un valor numérico directamente a la celda como se muestra arriba.

** NOTA2: la práctica recomendada es establecer la variable del libro de trabajo local en `ThisWorkbook` lugar de `ActiveWorkbook` (a menos que lo necesite explícitamente). La razón es que su macro generalmente necesitará / usará recursos en el libro de trabajo en que se origina el código VBA y NO mirará fuera de ese libro, de nuevo, a menos que usted indique explícitamente que su código funcione con otro libro. Cuando tiene varios libros abiertos en Excel, `ActiveWorkbook` es el que tiene un enfoque *diferente al que se está viendo en su Editor de VBA* . Entonces, crees que estás ejecutando en un libro de trabajo cuando realmente estás haciendo referencia a otro. `ThisWorkbook` refiere al libro de trabajo que contiene el código que se está ejecutando.

Siempre defina y establezca referencias a todos los libros de trabajo y hojas

Cuando trabaje con varios libros de trabajo abiertos, cada uno de los cuales puede tener múltiples hojas, es más seguro definir y establecer referencias a todos los libros de trabajo y hojas.

No confíe en `ActiveWorkbook` o `ActiveSheet` ya que pueden ser modificados por el usuario.

El ejemplo de código siguiente muestra cómo copiar un rango de hoja *“Raw_Data”* en el libro *“Data.xlsx”* a la hoja de *“Refined_Data”* en el libro *“Results.xlsx”.*

El procedimiento también se muestra cómo copiar y pegar sin utilizar la `Select` método.

```
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
```

El objeto WorksheetFunction se ejecuta más rápido que un equivalente UDF

VBA se compila en tiempo de ejecución, lo que tiene un gran impacto negativo en su rendimiento,

todo lo que se incorpore será más rápido, intente usarlos.

Como ejemplo, estoy comparando las funciones SUM y COUNTIF, pero puede usarlo para cualquier cosa que pueda resolver con WorkSheetFunctions.

Un primer intento para esos sería recorrer el rango y procesarlo celda por celda (usando un rango):

```
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
```

Una mejora puede ser almacenar los valores de rango en una matriz y procesar que:

```
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
```

Pero en lugar de escribir cualquier bucle, puede usar la función `Application.Worksheetfunction` que es muy útil para ejecutar fórmulas 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
```

O, para cálculos más complejos, incluso puedes usar `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
```

Y finalmente, corriendo por encima de los Subs de 25,000 veces cada uno, aquí está el tiempo promedio (5 pruebas) en milisegundos (por supuesto, será diferente en cada PC, pero comparados entre sí, se comportarán de manera similar):

1. UseWorksheetFunction: 2156 ms
2. UseArray: 2219 ms (+ 3%)
3. UsoEvaluar: 4693 ms (+ 118%)
4. Rango de utilización: 6530 ms (+ 203%)

Evite volver a proponer los nombres de Propiedades o Métodos como sus variables

En general, no se considera una "mejor práctica" reutilizar los nombres reservados de Propiedades o Métodos como el nombre de sus propios procedimientos y variables.

Forma incorrecta : si bien lo siguiente es (estrictamente hablando) legal, el código de trabajo, la reutilización del método de Búsqueda , así como las propiedades de Fila , Columna y Dirección , pueden causar problemas / conflictos con la ambigüedad del nombre y es simplemente confuso en general.

```
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
```

Buena forma : con todas las palabras reservadas renombradas en aproximaciones cercanas pero únicas de los originales, se han evitado todos los posibles conflictos de nombres.

```
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
```

Si bien puede llegar un momento en el que desee reescribir intencionalmente un método o una propiedad estándar según sus propias especificaciones, esas situaciones son pocas y distantes entre sí. En su mayor parte, evite reutilizar nombres reservados para sus propias construcciones.

Lea Mejores Prácticas VBA en línea: https://riptutorial.com/es/excel-vba/topic/1107/mejores-practicas-vba

Capítulo 21: Métodos para encontrar la última fila o columna utilizada en una hoja de trabajo

Observaciones

Puede encontrar una buena explicación sobre por qué otros métodos son desalentados / inexactos aquí: http://stackoverflow.com/a/11169920/4628637

Examples

Encuentre la última celda no vacía en una columna

En este ejemplo, veremos un método para devolver la última fila no vacía en una columna para un conjunto de datos.

Este método funcionará independientemente de las regiones vacías dentro del conjunto de datos.

Sin embargo, se *debe tener* ***cuidado*** *si las* ***celdas combinadas*** *están involucradas* , ya que el método `End` se "detendrá" contra una región fusionada, devolviendo la primera celda de la región fusionada.

Además, no se tendrán en cuenta las celdas no vacías en ***filas ocultas*** .

```
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
```

Para abordar las limitaciones indicadas anteriormente, la línea:
`LastRow = wS.Cells(wS.Rows.Count, "A").End(xlUp).Row`

puede ser reemplazado por:

1. Para la última fila usada de `"Sheet1"` :

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

2. para la última celda no vacía de la Columna `"A"` en `"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
```

Encuentra la última fila usando el rango con nombre

En caso de que tenga un rango con nombre en su hoja, y quiera obtener dinámicamente la última fila de ese rango con nombre dinámico. También cubre los casos en los que el rango con nombre no se inicia desde la primera fila.

```
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
```

Actualizar:
@Jeeped señaló una posible laguna para un rango con nombre con filas no contiguas, ya que genera un resultado inesperado. Para resolver ese problema, el código se revisa como se muestra a continuación.
Supuestos: hoja de targes = `form` , rango denominado = `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
```

Obtener la fila de la última celda en un rango

```
'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
```

Encuentre la última columna no vacía en la hoja de trabajo

```
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
```

Puede elegir entre 2 opciones, con respecto a si desea saber si no hay datos en la hoja de trabajo:

- NO: Use LastCol_1: puede usarlo directamente dentro de `wS.Cells(...,LastCol_1(wS))`
- SÍ: use LastCol_0: debe probar si el resultado que obtiene de la función es 0 o no antes de usarlo

```
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
```

Las propiedades del objeto Err se restablecen automáticamente a cero al salir de la función.

```
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
```

Última celda en Range.CurrentRegion

`Range.CurrentRegion` es un área de rango rectangular rodeada por celdas vacías. Las celdas en blanco con fórmulas como `=""` o `'` no se consideran en blanco (incluso por la función `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)
```

Encuentre la última fila no vacía en la hoja de trabajo

```
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
```

Puede elegir entre 2 opciones, con respecto a si desea saber si no hay datos en la hoja de trabajo:

- NO: Use LastRow_1: puede usarlo directamente dentro de `wS.Cells(LastRow_1(wS),...)`
- SÍ: use LastRow_0: debe probar si el resultado que obtiene de la función es 0 o no antes de usarlo

```
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
```

Encuentra la última celda no vacía en una fila

En este ejemplo, veremos un método para devolver la última columna no vacía en una fila.

Este método funcionará independientemente de las regiones vacías dentro del conjunto de datos.

Sin embargo, se *debe tener* ***cuidado*** *si las* ***celdas combinadas*** *están involucradas* , ya que el método `End` se "detendrá" contra una región fusionada, devolviendo la primera celda de la región fusionada.

Además, no se tendrán en cuenta las celdas no vacías en ***columnas ocultas*** .

```
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
```

Buscar la última celda no vacía en la hoja de trabajo - Rendimiento (matriz)

- La primera función, usando una matriz, es **mucho más rápida**
- Si se llama sin el parámetro opcional, se establecerá de forma predeterminada en `.ThisWorkbook.ActiveSheet`
- Si el rango está vacío, se devolverá la `Cell( 1, 1 )` como predeterminada, en lugar de `Nothing`

Velocidad:

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

.Medido con 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
```

Lea Métodos para encontrar la última fila o columna utilizada en una hoja de trabajo en línea: https://riptutorial.com/es/excel-vba/topic/918/metodos-para-encontrar-la-ultima-fila-o-columna-utilizada-en-una-hoja-de-trabajo

Capítulo 22: Objeto de aplicación

Observaciones

Excel VBA viene con un *modelo de objetos* completo que contiene clases y objetos que puede usar para manipular cualquier parte de la aplicación Excel en ejecución. Uno de los objetos más comunes que utilizará es el objeto **Aplicación** . Este es un catchall de nivel superior que representa la instancia actual de ejecución de Excel. Casi todo lo que no está conectado a un libro de Excel en particular está en el objeto **Aplicación** .

El objeto *Aplicación* , como un objeto de nivel superior, tiene literalmente cientos de propiedades, métodos y eventos que se pueden usar para controlar cada aspecto de Excel.

Examples

Ejemplo de objeto de aplicación simple: minimizar la ventana de Excel

Este código utiliza el objeto de **aplicación de** nivel superior para minimizar la ventana principal de Excel.

```
Sub MinimizeExcel()

    Application.WindowState = xlMinimized

End Sub
```

Ejemplo de objeto de aplicación simple: Mostrar versión Excel y VBE

```
Sub DisplayExcelVersions()

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

End Sub
```

El uso de la propiedad Application.Version es útil para garantizar que el código solo funcione en una versión compatible de Excel.

Lea Objeto de aplicación en línea: https://riptutorial.com/es/excel-vba/topic/5645/objeto-de-aplicacion

Capítulo 23: Objeto del sistema de archivos

Examples

Archivo, carpeta, unidad existe

El archivo 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
```

Carpeta 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
```

La unidad 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
```

Operaciones básicas de archivo

Dupdo:

```
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
```

Movimiento:

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

Borrar:

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

Operaciones básicas de carpeta

Crear:

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

Dupdo:

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

Movimiento:

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

Borrar:

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

Otras operaciones

Obtener nombre de archivo:

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

Resultado: *Makro.txt*

Obtener el nombre de la base:

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

Resultado: *Makro*

Obtener el nombre de la extensión:

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

```
End Sub
```

Resultado: *txt*

Obtener el nombre de la unidad:

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

Resultado: *c:*

Lea Objeto del sistema de archivos en línea: https://riptutorial.com/es/excel-vba/topic/9933/objeto-del-sistema-de-archivos

Capítulo 24: Optimización Excel-VBA

Introducción

La optimización de Excel-VBA se refiere también a la codificación de un mejor manejo de errores mediante documentación y detalles adicionales. Esto se muestra aquí.

Observaciones

*) Los números de línea que representan son enteros, es decir, un tipo de datos de 16 bits con signo en el rango de -32,768 a 32,767; de lo contrario, produce un desbordamiento. Por lo general, los números de línea se insertan en pasos de 10 sobre una parte del código o todos los procedimientos de un módulo en su totalidad.

Examples

Desactivando la actualización de la hoja de trabajo

Deshabilitar el cálculo de la hoja de trabajo puede disminuir significativamente el tiempo de ejecución de la macro. Además, deshabilitar eventos, actualizar la pantalla y saltos de página sería beneficioso. El siguiente `Sub` puede ser usado en cualquier macro para este propósito.

```
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
```

Para la optimización siga el siguiente pseudo-código:

```
Sub MyCode()

    OptimizeVBA True

    'Your code goes here

    OptimizeVBA False

End Sub
```

Comprobación del tiempo de ejecución.

Diferentes procedimientos pueden dar el mismo resultado, pero usarían un tiempo de procesamiento diferente. Para ver cuál es más rápido, se puede usar un código como este:

```
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
```

Utilizando con bloques

Usar con bloques puede acelerar el proceso de ejecutar una macro. En lugar de escribir un rango, un nombre de gráfico, una hoja de trabajo, etc., puede usar with-blocks como abajo;

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

Que es más rápido que esto:

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

Notas:

- Una vez que se ingresa un bloque With, el objeto no se puede cambiar. Como resultado, no puede usar una sola instrucción With para afectar una cantidad de objetos diferentes
- **No saltes dentro o fuera de los bloques** . Si se ejecutan las instrucciones en un bloque With, pero no se ejecuta la instrucción With o End With, **una variable temporal que contiene una referencia al objeto permanece en la memoria hasta que salga del procedimiento.**
- No bucle dentro de las declaraciones, especialmente si el objeto almacenado en caché se usa como un iterador
- Puede anidar con sentencias colocando una con bloque dentro de otra. Sin embargo, debido a que los miembros de los bloques Con externos están enmascarados dentro de los bloques Con internos, debe proporcionar una referencia de objeto completamente calificada en un bloque Con interno a cualquier miembro de un objeto en un bloque Con externo.

Ejemplo de anidación:

Este ejemplo utiliza la instrucción With para ejecutar una serie de instrucciones en un solo objeto. El objeto y sus propiedades son nombres genéricos utilizados solo para fines de ilustración.

```
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
```

Más información en MSDN

Eliminación de filas - Rendimiento

- La eliminación de filas es lenta, especialmente cuando se recorren las celdas y se eliminan filas, una por una
- Un enfoque diferente es usar un Autofiltro para ocultar las filas que se eliminarán
- Copie el rango visible y péguelo en una nueva hoja de trabajo
- Retire la hoja inicial por completo.
- Con este método, cuantas más filas se eliminen, más rápido será

Ejemplo:

```
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
```

Deshabilitar toda la funcionalidad de Excel antes de ejecutar macros grandes

Los siguientes procedimientos deshabilitarán temporalmente todas las funciones de Excel en WorkBook y WorkSheet level.

- FastWB () es un conmutador que acepta indicadores de activación o desactivación
- FastWS () acepta un objeto WorkSheet opcional, o ninguno
- Si falta el parámetro ws, se activarán y desactivarán todas las funciones para todas las hojas de trabajo de la colección.
    - Se puede usar un tipo personalizado para capturar todas las configuraciones antes de apagarlas
    - Al final del proceso, las configuraciones iniciales pueden ser restauradas

---

```
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
```

Restaura todas las configuraciones de Excel a las predeterminadas

```
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
```

Optimizando la búsqueda de errores por depuración extendida

Usando números de línea ... y documentándolos en caso de error ("La importancia de ver Erl")

Detectar qué línea genera un error es una parte sustancial de cualquier depuración y limita la búsqueda de la causa. Para documentar las líneas de error identificadas con una breve descripción, se completa un seguimiento exitoso de los errores, en el mejor de los casos junto con los nombres del módulo y el procedimiento. El siguiente ejemplo guarda estos datos en un archivo de registro.

Fondo

El objeto de error devuelve el número de error (Err.Number) y la descripción del error (Err.Description), pero no responde explícitamente a la pregunta dónde ubicar el error. Sin embargo, la función **Erl** sí lo hace, pero a condición de que agregue * *números de línea)* al código (por cierto, una de varias otras concesiones a los tiempos básicos anteriores).

Si no hay ninguna línea de error, entonces la función Erl devuelve 0, si la numeración está incompleta, obtendrá el último número de línea anterior del procedimiento.

```
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
```

' Código adicional para mostrar el archivo de registro

```
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
```

Lea Optimización Excel-VBA en línea: https://riptutorial.com/es/excel-vba/topic/9798/optimizacion-excel-vba

Capítulo 25: Recorrer todas las hojas en Active Workbook

Examples

Recuperar todos los nombres de hojas de trabajo en 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
```

Recorrer todas las hojas en todos los archivos en una carpeta

```
 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
```

Lea Recorrer todas las hojas en Active Workbook en línea: https://riptutorial.com/es/excel-vba/topic/1144/recorrer-todas-las-hojas-en-active-workbook

Capítulo 26: Seguridad VBA

Examples

Protege con contraseña tu VBA

A veces, tiene información confidencial en su VBA (por ejemplo, contraseñas) a la que no quiere que accedan los usuarios. Puede lograr una seguridad básica en esta información protegiendo con una contraseña su proyecto VBA.

Sigue estos pasos:

1. Abra su Editor de Visual Basic (Alt + F11)
2. Vaya a Herramientas -> Propiedades de VBAProject ...
3. Vaya a la pestaña Protección
4. Marque la casilla de verificación "Bloquear proyecto para ver"
5. Ingrese su contraseña deseada en los cuadros de texto Contraseña y Confirmar contraseña

Ahora, cuando alguien quiere acceder a su código dentro de una aplicación de Office, primero deberá ingresar la contraseña. Tenga en cuenta, sin embargo, que incluso una contraseña de proyecto VBA fuerte es trivial de romper.

Lea Seguridad VBA en línea: https://riptutorial.com/es/excel-vba/topic/7642/seguridad-vba

Capítulo 27: SQL en Excel VBA - Mejores Prácticas

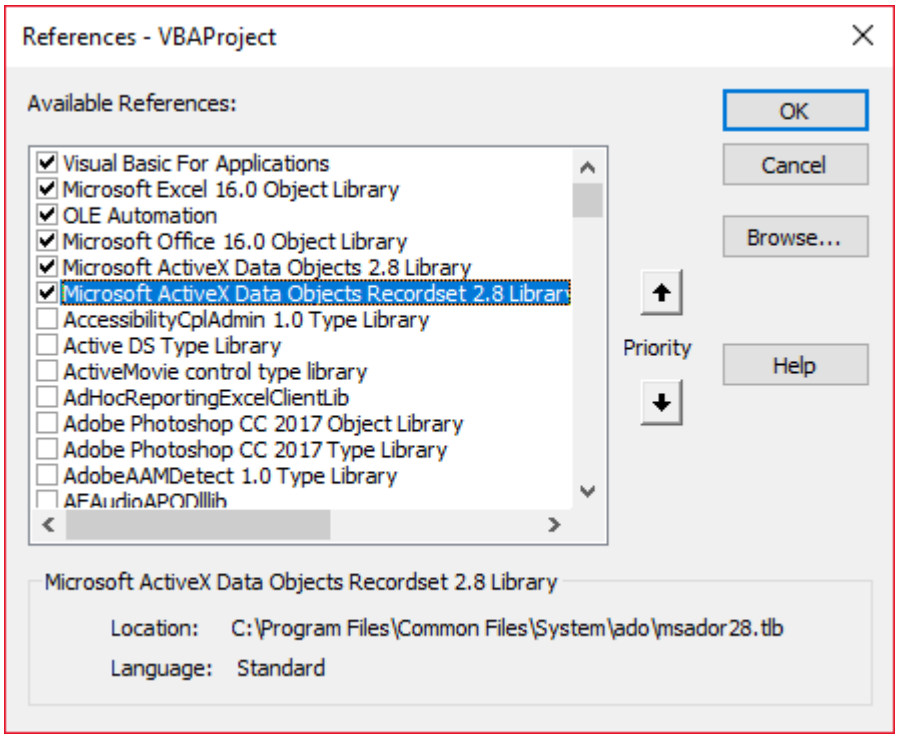
Examples

¿Cómo usar ADODB.Connection en VBA?

Requisitos:

Agregue las siguientes referencias al proyecto:

- Biblioteca Microsoft ActiveX Data Objects 2.8
- Biblioteca de Microsoft ActiveX Data Objects Recordset 2.8



Declarar variables

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

Crear conexion

a. con autenticación de 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
```

segundo. con autenticación de 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
```

Ejecutar comando sql

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

Leer datos del conjunto de registros

```
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
```

Conexión cercana

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

¿Cómo usarlo?

```
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
```

Resultado

ShipperID: 1 CompanyName: Speedy Express Teléfono: (503) 555-9831

ShipperID: 2 CompanyName: United Package Phone: (503) 555-3199

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

ShipperID: 4 CompanyName: speedy shipping Teléfono: (503) 555-1234

Lea SQL en Excel VBA - Mejores Prácticas en línea: https://riptutorial.com/es/excel-vba/topic/9958/sql-en-excel-vba---mejores-practicas

Capítulo 28: Tablas dinamicas

Observaciones

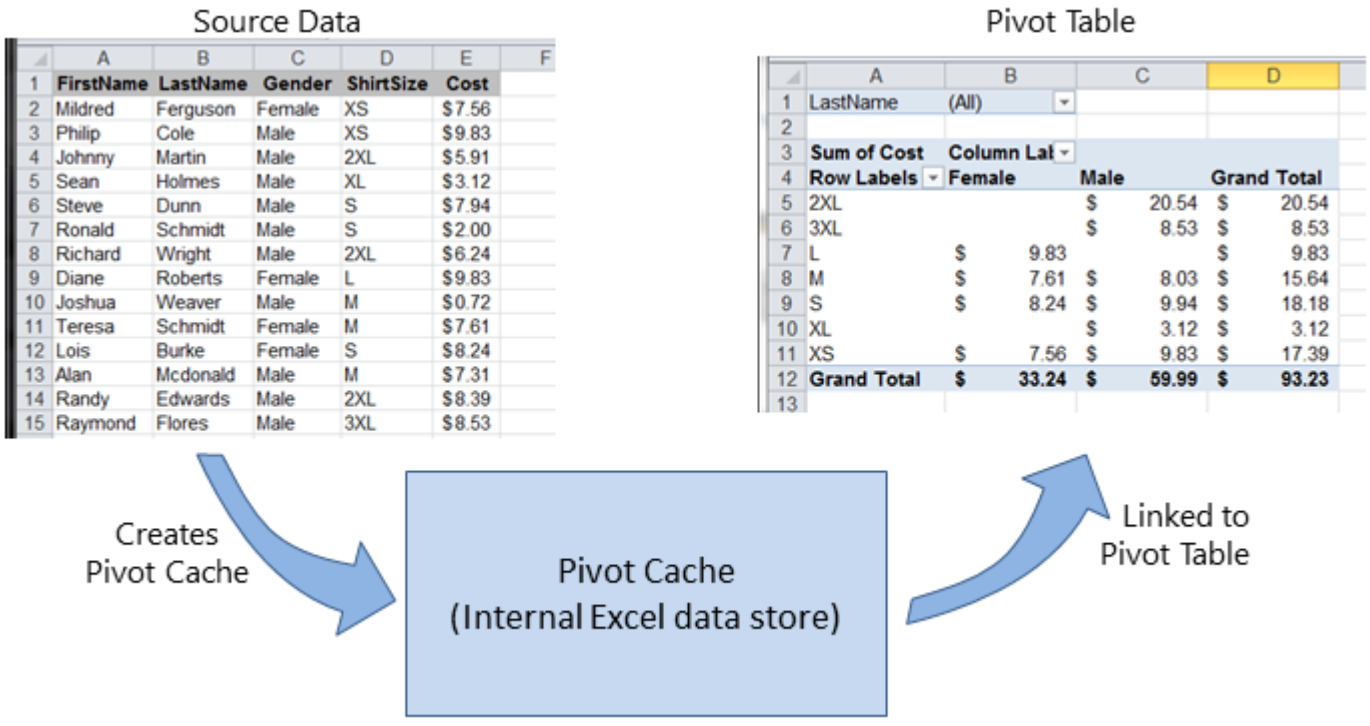
Hay muchas fuentes de referencia y ejemplos excelentes en la Web. Algunos ejemplos y explicaciones se crean aquí como un punto de recolección para respuestas rápidas. Se pueden vincular ilustraciones más detalladas al contenido externo (en lugar de copiar el material original existente).

Examples

Creación de una tabla dinámica

Una de las capacidades más poderosas en Excel es el uso de tablas dinámicas para clasificar y analizar datos. Usar VBA para crear y manipular los Pivots es más fácil si entiende la relación de Pivot Tables con los Pivot Caches y cómo hacer referencia y usar las diferentes partes de las Tablas.

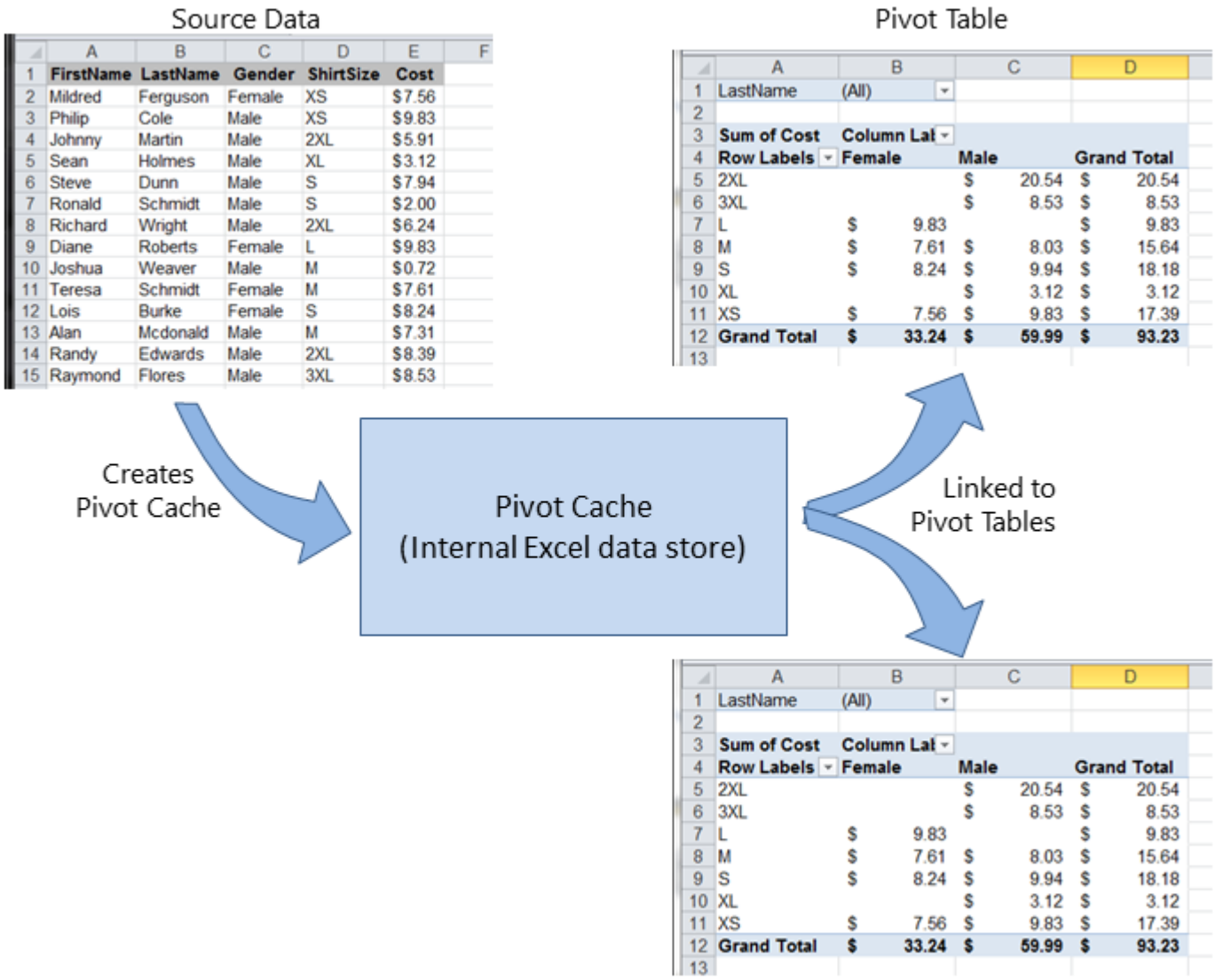
En su forma más básica, sus datos de origen son un área de datos de `Range` en una `Worksheet` . Esta área de datos **DEBE** identificar las columnas de datos con una fila de encabezado como la primera fila en el rango. Una vez que se crea la tabla dinámica, el usuario puede ver y cambiar los datos de origen en cualquier momento. Sin embargo, es posible que los cambios no se reflejen de forma automática o inmediata en la propia Tabla dinámica porque existe una estructura de almacenamiento de datos intermedia denominada Caché de pivote que está directamente conectada a la propia Tabla dinámica.



	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 se necesitan varias tablas dinámicas, basadas en los mismos datos de origen, la memoria caché dinámica puede reutilizarse como almacén de datos interno para cada una de las tablas dinámicas. Esta es una buena práctica porque ahorra memoria y reduce el tamaño del archivo de Excel para el almacenamiento.



Source Data

	A	B	C	D	E
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				

Como ejemplo, para crear una tabla dinámica basada en los datos de origen que se muestran en las figuras anteriores:

```
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
```

Referencias MSDN Pivot Table Object

Rangos de tabla de pivote

Estas excelentes fuentes de referencia proporcionan descripciones e ilustraciones de los diversos rangos en las tablas dinámicas.

Referencias

- Referencia a los rangos de tablas dinámicas en VBA - del blog de tecnología de Jon Peltier
- Referencia a un rango de tabla dinámica de Excel mediante VBA - de globaliconnect Excel VBA

Agregar campos a una tabla dinámica

Dos cosas importantes a tener en cuenta al agregar campos a una tabla dinámica son la orientación y la posición. A veces, un desarrollador puede asumir dónde se coloca un campo, por lo que siempre es más claro definir explícitamente estos parámetros. Estas acciones solo afectan a la tabla dinámica dada, no al caché dinámico.

```
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
```

Formato de los datos de la tabla dinámica

Este ejemplo cambia / establece varios formatos en el área de rango de datos (`DataBodyRange`) de la Tabla de Pivot dada. Todos los parámetros formateables en un `Range` estándar están disponibles. El formateo de los datos solo afecta a la propia tabla dinámica, no a la memoria caché dinámica.

NOTA: la propiedad se llama `TableStyle2` porque la propiedad `TableStyle` no es un miembro de las propiedades de objeto de la `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
```

Lea Tablas dinamicas en línea: https://riptutorial.com/es/excel-vba/topic/3797/tablas-dinamicas

Capítulo 29: Trabajando con tablas de Excel en VBA

Introducción

Este tema trata sobre el trabajo con tablas en VBA y asume el conocimiento de las tablas de Excel. En VBA, o más bien en el modelo de objetos de Excel, las tablas se conocen como ListObjects. Las propiedades utilizadas con más frecuencia de un objeto ListObject son ListRow (s), ListColumn (s), DataBodyRange, Range y HeaderRowRange.

Examples

Creando un objeto de lista

```
Dim lo as ListObject
Dim MyRange as Range

Set lo = Sheet1.ListObjects(1)

'or

Set lo = Sheet1.ListObjects("Table1")

'or

Set lo = MyRange.ListObject
```

Trabajando con 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
```

Convertir una tabla de Excel a un rango normal

```
Dim lo as ListObject

Set lo = Sheet1.ListObjects("Table1")
lo.Unlist
```

Lea Trabajando con tablas de Excel en VBA en línea: https://riptutorial.com/es/excel-vba/topic/9753/trabajando-con-tablas-de-excel-en-vba

Capítulo 30: Unión

Examples

Unión temprana vs Unión tardía

La vinculación es el proceso de asignar un objeto a un identificador o nombre de variable. El enlace temprano (también conocido como enlace estático) es cuando un objeto declarado en Excel es de un tipo de objeto específico, como una hoja de trabajo o un libro de trabajo. La vinculación tardía se produce cuando se realizan asociaciones de objetos generales, como los tipos de declaración Objeto y Variante.

Enlace anticipado de referencias algunas ventajas sobre el enlace tardío.

- La vinculación temprana es operativamente más rápida que la vinculación tardía durante el tiempo de ejecución. Crear el objeto con un enlace tardío en tiempo de ejecución lleva tiempo que el enlace temprano se logra cuando el proyecto VBA se carga inicialmente.
- El enlace temprano ofrece una funcionalidad adicional a través de la identificación de pares de Clave / Artículo por su posición ordinal.
- Dependiendo de la estructura del código, el enlace temprano puede ofrecer un nivel adicional de verificación de tipos y reducir los errores.
- La corrección de capitalización de VBE cuando se escriben las propiedades y los métodos de un objeto enlazado está activa con la vinculación temprana pero no está disponible con la vinculación tardía.

  Nota: debe agregar la referencia apropiada al proyecto de VBA a través del comando Herramientas → Referencias de VBE para implementar el enlace temprano.

  Esta referencia de la biblioteca se lleva entonces con el proyecto; no es necesario volver a referenciarlo cuando el proyecto de VBA se distribuye y ejecuta en otra computadora.

```
'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 binding¹
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
```

Sin embargo, si está utilizando el enlace anticipado y el documento se ejecuta en un sistema que carece de una de las bibliotecas a las que ha hecho referencia, encontrará problemas. Las rutinas que utilizan la biblioteca faltante no solo no funcionarán correctamente, sino que el comportamiento de todo el código dentro del documento se volverá errático. Es probable que ninguno de los códigos del documento funcione en esa computadora.

Aquí es donde la unión tardía es ventajosa. Cuando utilice el enlace tardío, no tiene que agregar la referencia en el menú Herramientas> Referencias. En las máquinas que tienen la biblioteca adecuada, el código seguirá funcionando. En las máquinas sin esa biblioteca, los comandos que hacen referencia a la biblioteca no funcionarán, pero todos los demás códigos de su documento continuarán funcionando.

Si no está completamente familiarizado con la biblioteca a la que hace referencia, puede ser útil utilizar el enlace temprano al escribir el código, y luego cambiar al enlace tardío antes de la implementación. De esa manera, puede aprovechar el IntelliSense y el buscador de objetos de VBE durante el desarrollo.

Lea Unión en línea: https://riptutorial.com/es/excel-vba/topic/3811/union

Capítulo 31: Usar el objeto de la hoja de trabajo y no el objeto de hoja

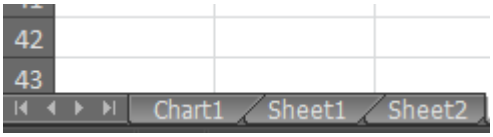
Introducción

Muchos de los usuarios de VBA consideran que las hojas de trabajo y las hojas de objetos son sinónimos. Ellos no son.

El objeto Hojas se compone de hojas de trabajo y gráficos. Por lo tanto, si tenemos gráficos en nuestro Libro de Excel, debemos tener cuidado de no usar `Sheets Worksheets` y `Sheets Worksheets` como sinónimos.

Examples

Imprime el nombre del primer objeto.




```
Option Explicit

Sub CheckWorksheetsDiagram()

    Debug.Print Worksheets(1).Name
    Debug.Print Charts(1).Name
    Debug.Print Sheets(1).Name

End Sub
```

El resultado:

```
Sheet1
Chart1
Chart1
```

Lea Usar el objeto de la hoja de trabajo y no el objeto de hoja en línea: https://riptutorial.com/es/excel-vba/topic/9996/usar-el-objeto-de-la-hoja-de-trabajo-y-no-el-objeto-de-hoja

Creditos

S. No	Capítulos	Contributors
1	Empezando con 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	Arrays	Alon Adler, Hubisan, Miguel_Ryu, Shahin
3	autofiltro Usos y mejores prácticas.	Sgdva
4	Celdas / Rangos Combinados	R3uK
5	Cómo grabar una macro	Mike, Robby
6	Creación de un menú desplegable en la hoja de cálculo activa con un cuadro combinado	Macro Man, quadrature, R3uK
7	Cuadernos de ejercicios	PeterT
8	CustomDocumentProperties en la práctica	T.M.
9	Declaraciones condicionales	SteveES
10	Depuración y solución de problemas	Cody G., Etheur, Gregor y, Julian Kuchlbauer, Kyle, Malick, Michael Russo, RGA, Ron McMahon, Slai, Steven Schroeder, Taylor Ostberg
11	Errores comunes	Egan Wolf, Gordon Bell, Macro Man, Malick, Peh, SWa, Taylor Ostberg
12	Excel VBA consejos y trucos	Andre Terra, Cody G., Jeeped, Kumar Sourav, Macro Man, RGA
13	Formato condicional utilizando VBA	Zsmaster
14	Funciones definidas por el	Jeeped, Malick, Slai, user3561813, Vegard

	usuario (UDF)	
15	Gamas nombradas	Andre Terra, Portland Runner
16	Gamas y celulas	Adam, Branislav Kollár, Doug Coats, Gregor y, Jbjstam, Joel Spolsky, Julian Kuchlbauer, Máté Juhász, Miguel_Ryu, Patrick Wynne, Vegard
17	Gráficos y gráficos	Byron Wall
18	Integración de PowerPoint a través de VBA	mnoronha, RGA
19	Localización de valores duplicados en un rango	quadrature, T.M.
20	Mejores Prácticas 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
21	Métodos para encontrar la última fila o columna utilizada en una hoja de trabajo	curious, Hubisan, Máté Juhász, Michael Russo, Miqi180, paul bica, R3uK, Raystafarian, RGA, Shai Rado, Slai, Thomas Inzina, YowE3K
22	Objeto de aplicación	Captain Grumpy, Joel Spolsky
23	Objeto del sistema de archivos	Zsmaster
24	Optimización Excel-VBA	Masoud, paul bica, T.M.
25	Recorrer todas las hojas en Active Workbook	Doug Coats, Shai Rado
26	Seguridad VBA	Chel, TheGuyThatDoesn'tKnowMuch
27	SQL en Excel VBA - Mejores Prácticas	Zsmaster
28	Tablas dinamicas	PeterT
29	Trabajando con tablas de Excel en VBA	Excel Developers
30	Unión	Captain Grumpy, EEM, Jeeped, jlookup, Malick,

		Raystafarian
31	Usar el objeto de la hoja de trabajo y no el objeto de hoja	Vityata