



EBook Gratuito

APPENDIMENTO

excel-vba

Free unaffiliated eBook created from
Stack Overflow contributors.

#excel-vba

Sommario

Di.....	1
Capitolo 1: Iniziare con excel-vba.....	2
Osservazioni.....	2
Versioni.....	2
VB.....	2
Eccellere.....	3
Examples.....	3
Dichiarazione delle variabili.....	3
Altri modi di dichiarare le variabili sono:.....	4
Apertura di Visual Basic Editor (VBE).....	5
Aggiunta di un nuovo riferimento alla libreria degli oggetti.....	6
Ciao mondo.....	11
Introduzione al modello a oggetti di Excel.....	13
Capitolo 2: Array.....	17
Examples.....	17
Compilazione di matrici (aggiunta di valori).....	17
Direttamente.....	17
Usando la funzione Array ().....	17
Dalla gamma.....	17
2D con Evaluate ().....	18
Usando la funzione Dividi ().....	18
Array dinamici (ridimensionamento della matrice e gestione dinamica).....	18
Array frastagliati (array di array).....	18
Controlla se la matrice è inizializzata (se contiene elementi o no).....	19
Array dinamici [Dichiarazione array, ridimensionamento].....	19
Capitolo 3: autofilter; Usi e buone pratiche.....	20
introduzione.....	20
Osservazioni.....	20
Examples.....	20
SmartFilter!.....	20

Capitolo 4: Best practice VBA	26
Osservazioni.....	26
Examples.....	26
Utilizzare SEMPRE "Option Explicit".....	26
Lavora con le matrici, non con le gamme.....	28
Utilizzare le costanti VB quando disponibili.....	29
Usa la denominazione delle variabili descrittive.....	30
Gestione degli errori.....	31
On Error GoTo 0	31
In caso di errore, riprendi	31
On Error GoTo <line>	32
Documenta il tuo lavoro.....	33
Disattiva le proprietà durante l'esecuzione della macro.....	34
Evitare l'uso di ActiveCell o ActiveSheet in Excel.....	36
Non assumere mai il foglio di lavoro.....	36
Evitare l'uso di SELECT o ACTIVATE.....	37
Definisci sempre e imposta i riferimenti a tutte le cartelle di lavoro e fogli.....	38
L'oggetto Worksheet Function viene eseguito più velocemente di un equivalente UDF.....	39
Evita di riproporre i nomi di Proprietà o Metodi come variabili.....	41
Capitolo 5: Celle / gamme unite	43
Examples.....	43
Pensaci due volte prima di utilizzare celle / intervalli uniti.....	43
Dove sono i dati in un intervallo unito?.....	43
Capitolo 6: Come registrare una macro	44
Examples.....	44
Come registrare una macro.....	44
Capitolo 7: Creazione di un menu a discesa nel foglio di lavoro attivo con una casella com	47
introduzione.....	47
Examples.....	47
Menu di Jimi Hendrix.....	47
Esempio 2: Opzioni non incluse.....	48

Capitolo 8: CustomDocumentProperties in pratica	51
introduzione	51
Examples	51
Organizzazione di nuovi numeri di fattura	51
Capitolo 9: Debug e risoluzione dei problemi	54
Sintassi	54
Examples	54
Debug.Print	54
Stop	54
Finestra immediata	54
Utilizzare il timer per individuare i colli di bottiglia nelle prestazioni	55
Aggiunta di un punto di interruzione al codice	56
Finestra dei debugger locali	56
Capitolo 10: Dichiarazioni condizionali	59
Examples	59
La dichiarazione di If	59
Capitolo 11: Errori comuni	61
Examples	61
Riferimenti qualificanti	61
Eliminazione di righe o colonne in un ciclo	62
ActiveWorkbook vs. ThisWorkbook	62
Interfaccia a singolo documento e interfacce a più documenti	63
Capitolo 12: File System Object	66
Examples	66
File, cartella, unità esistente	66
Il file esiste:	66
Cartella esiste:	66
L'unità esiste:	66
Operazioni di file di base	66
Copia:	66
Mossa:	67

Elimina:	67
Operazioni di cartella di base	67
Creare:	67
Copia:	67
Mossa:	67
Elimina:	68
Altre operazioni	68
Ottieni il nome del file:	68
Ottieni il nome base:	68
Ottieni il nome dell'estensione:	68
Ottieni il nome del drive:	69
Capitolo 13: Formattazione condizionale tramite VBA	70
Osservazioni	70
Examples	70
FormatConditions.Add	70
Sintassi:	70
parametri:	70
XIFormatConditionType enumeration	70
Formattazione per valore della cella:	71
operatori	71
La formattazione per testo contiene:	72
operatori	72
Formattazione per periodo	72
operatori	72
Rimuovi il formato condizionale	73
Rimuovi tutti i formati condizionali nell'intervallo	73
Rimuovi tutto il formato condizionale nel foglio di lavoro	73
FormatConditions.AddUniqueValues	73
Evidenziare i valori duplicati	73
Evidenziando i valori unici	73

FormatConditions.AddTop10.....	74
Evidenziando i 5 valori principali.....	74
FormatConditions.AddAboveAverage.....	74
operatori:.....	74
FormatConditions.AddIconSetCondition.....	74
IconSet:.....	75
Genere:.....	76
Operatore:.....	77
Valore:.....	77
Capitolo 14: Funzioni definite dall'utente (UDF).....	78
Sintassi.....	78
Osservazioni.....	78
Examples.....	78
UDF - Hello World.....	78
Consenti riferimenti a colonne complete senza penalità.....	80
Contare i valori unici nell'intervallo.....	81
Capitolo 15: Gamme e celle.....	83
Sintassi.....	83
Osservazioni.....	83
Examples.....	83
Creazione di un intervallo.....	83
Modi per fare riferimento a una singola cella.....	85
Salvataggio di un riferimento a una cella in una variabile.....	85
Proprietà offset.....	86
Come trasporre gli intervalli (da orizzontale a verticale e viceversa).....	86
Capitolo 16: Gamme nominate.....	87
introduzione.....	87
Examples.....	87
Definisci un intervallo con nome.....	87
Utilizzo di intervalli denominati in VBA.....	87
Gestisci intervalli denominati utilizzando Name Manager.....	88
Array gamma nominati.....	90

Capitolo 17: Grafici e grafici	92
Examples.....	92
Creazione di un grafico con intervalli e un nome fisso.....	92
Creare un grafico vuoto.....	93
Creare un grafico modificando la formula SERIE.....	95
Organizzazione di grafici in una griglia.....	97
Capitolo 18: Individuazione di valori duplicati in un intervallo	101
introduzione.....	101
Examples.....	101
Trova duplicati in un intervallo.....	101
Capitolo 19: Integrazione di PowerPoint tramite VBA	103
Osservazioni.....	103
Examples.....	103
Nozioni di base: avvio di PowerPoint da VBA.....	103
Capitolo 20: Lavorare con le tabelle di Excel in VBA	105
introduzione.....	105
Examples.....	105
Istanziare un oggetto ListObject.....	105
Lavorare con ListRows / ListColumns.....	105
Conversione di una tabella di Excel in un intervallo normale.....	106
Capitolo 21: Le cartelle di lavoro	107
Examples.....	107
Cartelle di lavoro dell'applicazione.....	107
Quando utilizzare ActiveWorkbook e ThisWorkbook.....	107
Aprire una (nuova) cartella di lavoro, anche se è già aperta.....	108
Salvataggio di una cartella di lavoro senza chiedere l'utente.....	109
Modifica del numero predefinito di fogli di lavoro in una nuova cartella di lavoro.....	109
Capitolo 22: Metodi per trovare l'ultima riga o colonna usata in un foglio di lavoro	110
Osservazioni.....	110
Examples.....	110
Trova l'ultima cella non vuota in una colonna.....	110
Trova l'ultima riga usando l'intervallo con nome.....	111

Prendi la riga dell'ultima cella in un intervallo.....	111
Trova l'ultima colonna non vuota nel foglio di lavoro.....	112
Ultima cella in Range.CurrentRegion.....	112
Trova l'ultima riga non vuota nel foglio di lavoro.....	113
Trova l'ultima cella non vuota in una riga.....	113
Trova l'ultima cella non vuota nel foglio di lavoro - Prestazioni (matrice).....	114
Capitolo 23: Oggetto dell'applicazione.....	116
Osservazioni.....	116
Examples.....	116
Esempio di oggetto applicazione semplice: ridurre a icona la finestra di Excel.....	116
Esempio di oggetto applicazione semplice: visualizzazione di Excel e versione VBE.....	116
Capitolo 24: Ottimizzazione Excel-VBA.....	117
introduzione.....	117
Osservazioni.....	117
Examples.....	117
Disabilitare l'aggiornamento del foglio di lavoro.....	117
Controllo del tempo di esecuzione.....	117
Utilizzo dei blocchi.....	118
Cancellazione riga - Prestazioni.....	119
Disabilitazione di tutte le funzionalità di Excel Prima di eseguire macro di grandi dimens.....	120
Ottimizzazione della ricerca errori tramite debug esteso.....	121
Capitolo 25: Passa in rassegna tutti i fogli in Active Workbook.....	124
Examples.....	124
Recupera tutti i nomi dei fogli di lavoro nella cartella di lavoro attiva.....	124
Passa attraverso tutti i fogli in tutti i file in una cartella.....	124
Capitolo 26: Rilegatura.....	126
Examples.....	126
Early Binding vs Late Binding.....	126
Capitolo 27: Sicurezza VBA.....	128
Examples.....	128
Password Proteggi il tuo VBA.....	128
Capitolo 28: SQL in Excel VBA: best practice.....	129

Examples.....	129
Come utilizzare ADODB.Connection in VBA?.....	129
Requisiti:.....	129
Dichiarare le variabili.....	129
Crea una connessione.....	129
un. con autenticazione di Windows.....	129
b. con autenticazione di SQL Server.....	130
Esegui il comando sql.....	130
Leggi i dati dal set di record.....	130
Chiudere la connessione.....	130
Come usarlo?.....	130
Risultato.....	131
Capitolo 29: Suggerimenti e trucchi Excel VBA.....	132
Osservazioni.....	132
Examples.....	132
Usando i fogli xlVeryHidden.....	132
Foglio di lavoro. Nome, .Index o .CodeName.....	133
Utilizzo di stringhe con delimitatori al posto di matrici dinamiche.....	135
Evento doppio clic per forme Excel.....	136
Finestra di dialogo Apri file - Più file.....	136
Capitolo 30: Tabelle pivot.....	138
Osservazioni.....	138
Examples.....	138
Creazione di una tabella pivot.....	138
Intervalli di tabella pivot.....	140
Aggiunta di campi a una tabella pivot.....	140
Formattazione dei dati della tabella pivot.....	140
Capitolo 31: Usa oggetto foglio di lavoro e non oggetto Foglio.....	142
introduzione.....	142
Examples.....	142
Stampa il nome del primo oggetto.....	142

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.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

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

Capitolo 1: Iniziare con excel-vba

Osservazioni

Microsoft Excel include un linguaggio di programmazione macro completo chiamato VBA. Questo linguaggio di programmazione ti fornisce almeno tre risorse aggiuntive:

1. Esegui automaticamente l'estrazione di Excel dal codice utilizzando Macro. Per la maggior parte, tutto ciò che l'utente può fare manipolando Excel dall'interfaccia utente può essere fatto scrivendo il codice in Excel VBA.
2. Crea nuove funzioni di foglio di lavoro personalizzate.
3. Interagisci Excel con altre applicazioni come Microsoft Word, PowerPoint, Internet Explorer, Blocco note, ecc.

VBA è l'acronimo di Visual Basic, Applications Edition. È una versione personalizzata del venerabile linguaggio di programmazione Visual Basic che ha alimentato i macro di Microsoft Excel dalla metà degli anni '90.

IMPORTANTE

Assicurati che tutti gli esempi o gli argomenti creati all'interno del tag excel-vba siano **specifici** e **pertinenti** all'uso di VBA con Microsoft Excel. Qualsiasi argomento o esempio suggerito che sia generico alla lingua VBA dovrebbe essere rifiutato al fine di evitare la duplicazione degli sforzi.

- esempi in tema:
 - ✓ *Creazione e interazione con oggetti del foglio di lavoro*
 - ✓ *La classe `WorksheetFunction` e i rispettivi metodi*
 - ✓ *Usare l'enumerazione `xlDirection` per navigare in un intervallo*
- esempi fuori tema:
 - ✗ *Come creare un ciclo "per ogni"*
 - ✗ *Classe `MsgBox` e come visualizzare un messaggio*
 - ✗ *Utilizzo di WinAPI in VBA*

Versionsi

VB

Versione	Data di rilascio
VB6	1998/10/01
VB7	2001-06-06

Versione	Data di rilascio
WIN32	1998/10/01
Win64	2001-06-06
MAC	1998/10/01

Eccellere

Versione	Data di rilascio
16	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

Dichiarazione delle variabili

Per dichiarare esplicitamente le variabili in VBA, utilizzare l'istruzione `Dim`, seguita dal nome e dal tipo della variabile. Se una variabile viene utilizzata senza essere dichiarata, o se non viene specificato alcun tipo, verrà assegnato il tipo `Variant`.

Utilizzare l'istruzione `Option Explicit` sulla prima riga di un modulo per forzare tutte le variabili da dichiarare prima dell'utilizzo (vedere [SEMPRE Utilizzare "Option Explicit"](#)).

Si consiglia vivamente di utilizzare sempre `Option Explicit` poiché aiuta a prevenire errori di battitura / ortografia e garantisce che le variabili / oggetti rimangano il tipo previsto.

```
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
```

È possibile dichiarare più variabili su una singola riga utilizzando le virgole come delimitatori, ma **ogni tipo deve essere dichiarato singolarmente** oppure verrà impostato di default sul 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
```

Le variabili possono anche essere dichiarate usando i suffissi di tipo di carattere Data (\$% &! # @), Tuttavia l'uso di questi è sempre più scoraggiato.

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

Altri modi di dichiarare le variabili sono:

- **Static come:** `Static CounterVariable as Integer`

Quando si utilizza l'istruzione statica anziché un'istruzione Dim, la variabile dichiarata manterrà il suo valore tra le chiamate.

- **Public come:** `Public CounterVariable as Integer`

Le variabili pubbliche possono essere utilizzate in qualsiasi procedura del progetto. Se una variabile pubblica viene dichiarata in un modulo standard o in un modulo di classe, può essere utilizzata anche in tutti i progetti che fanno riferimento al progetto in cui viene dichiarata la variabile pubblica.

- Private **come:** Private CounterVariable as Integer

Le variabili private possono essere utilizzate solo dalle procedure nello stesso modulo.

Fonte e maggiori informazioni:

[MSDN-dichiarare le variabili](#)

[Digitare caratteri \(Visual Basic\)](#)

Apertura di Visual Basic Editor (VBE)

Passaggio 1: aprire una cartella di lavoro

File Home Insert Page Layout Formulas Data Review View Developer Tell me what you want to do

Cut Copy Paste Format Painter Clipboard

Century gothic 10 A A B I U Font

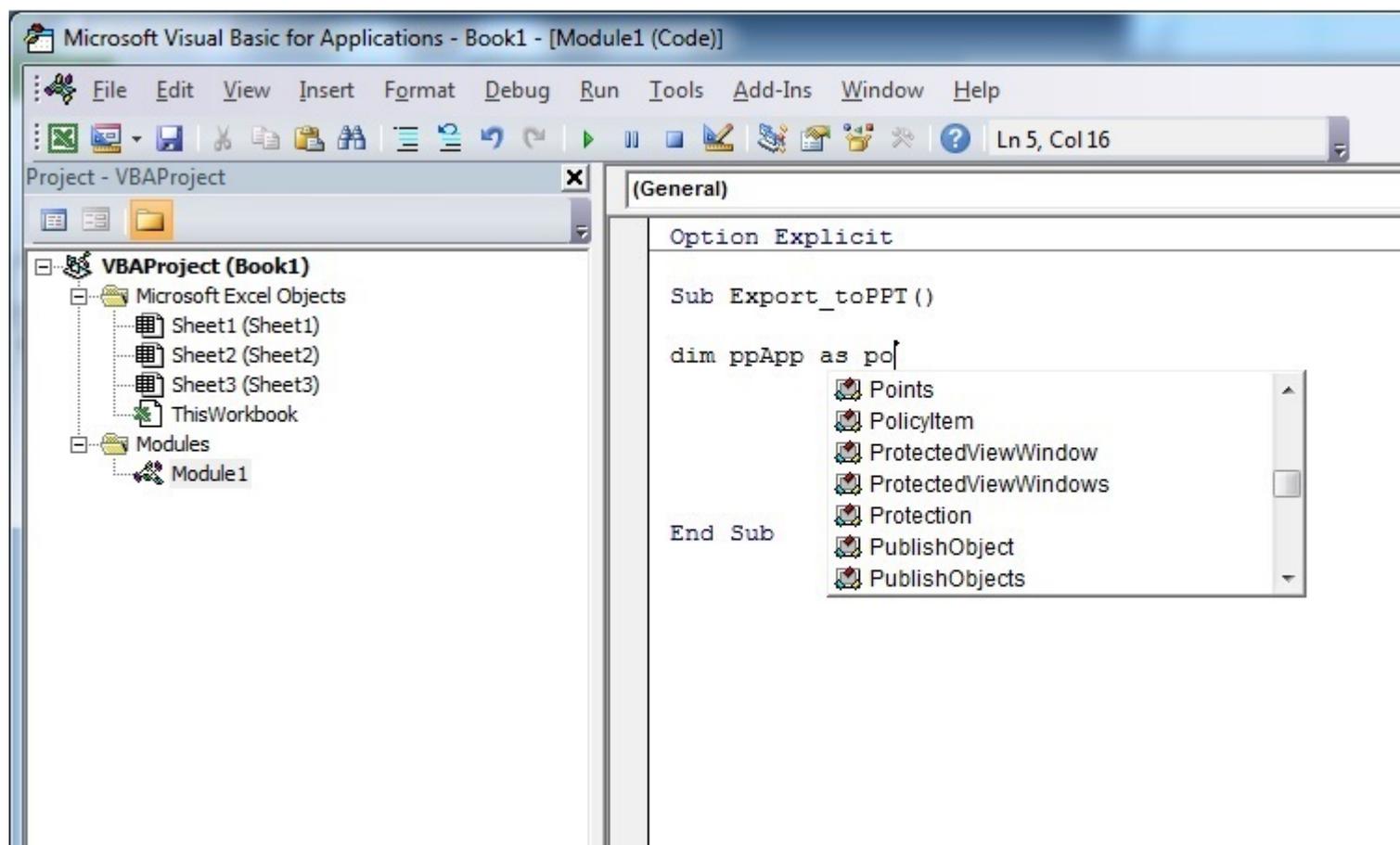
Wrap Text Merge & Center Alignment

General Number

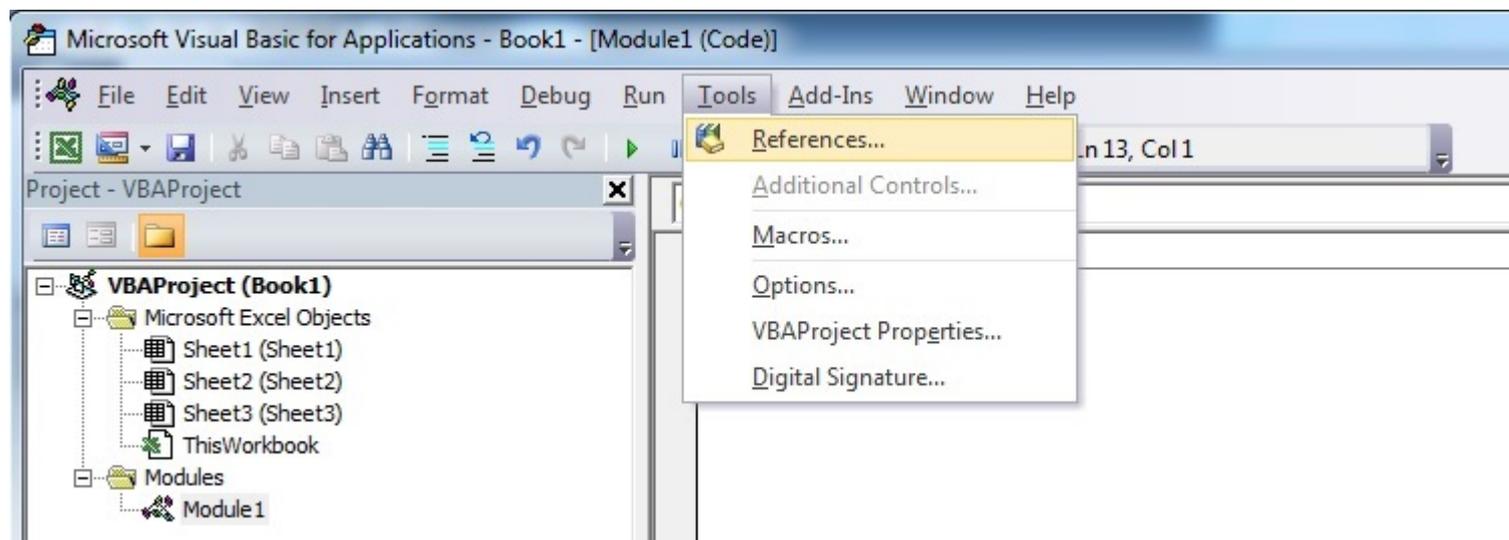
A1

	A	B	C	D	E	F	G	H	I	J	K
1											
2											
3											
4											
5											
6											
7											
8											
9											
10											
11											
12											
13											
14											
15											
16											
17											
18											
19											
20											
21											
22											
23											
24											
25											
26											
27											
28											
29											
30											
31											
32											
33											
34											
35											
36											
37											
38											
39											
40											
41											
42											
43											
44											
45											

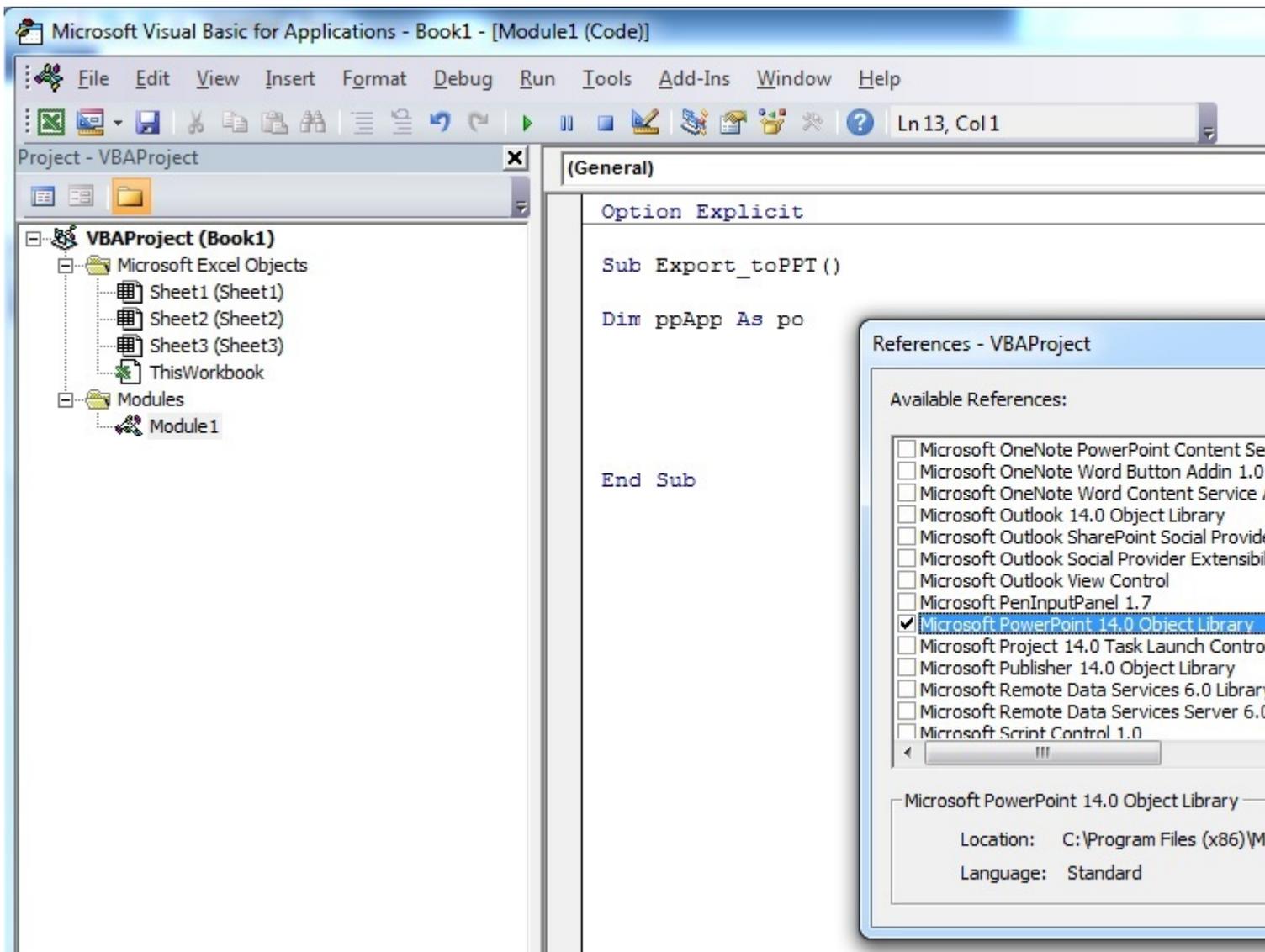
al progetto VB esistente. Come si può vedere, attualmente la libreria degli oggetti di PowerPoint non è disponibile.



Passaggio 1 : selezionare **Strumenti** menu -> **Riferimenti ...**

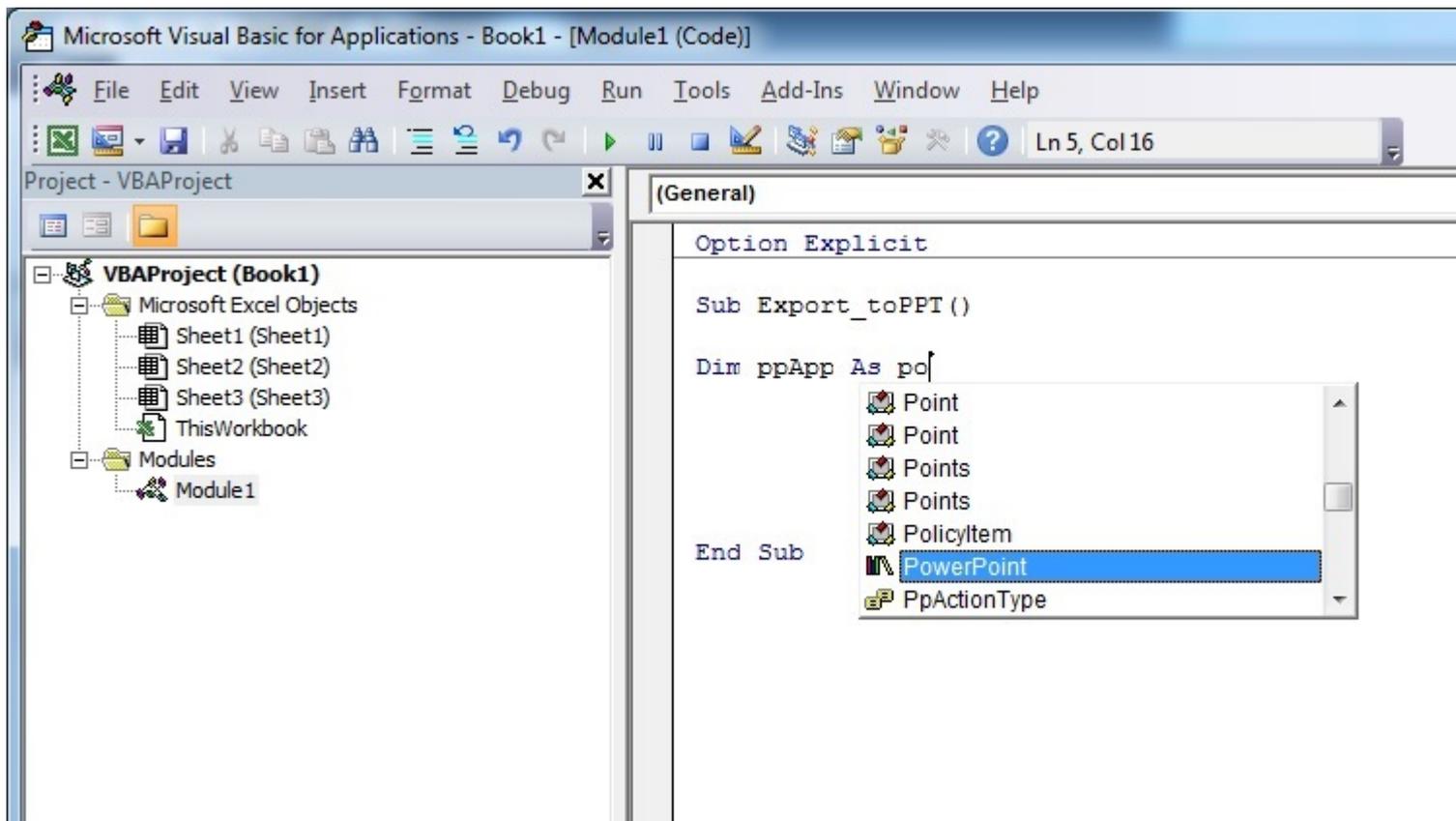


Passaggio 2 : selezionare il riferimento che si desidera aggiungere. Questo esempio si scorre verso il basso per trovare " **Microsoft PowerPoint 14.0 Object Library** ", quindi premere " **OK** ".

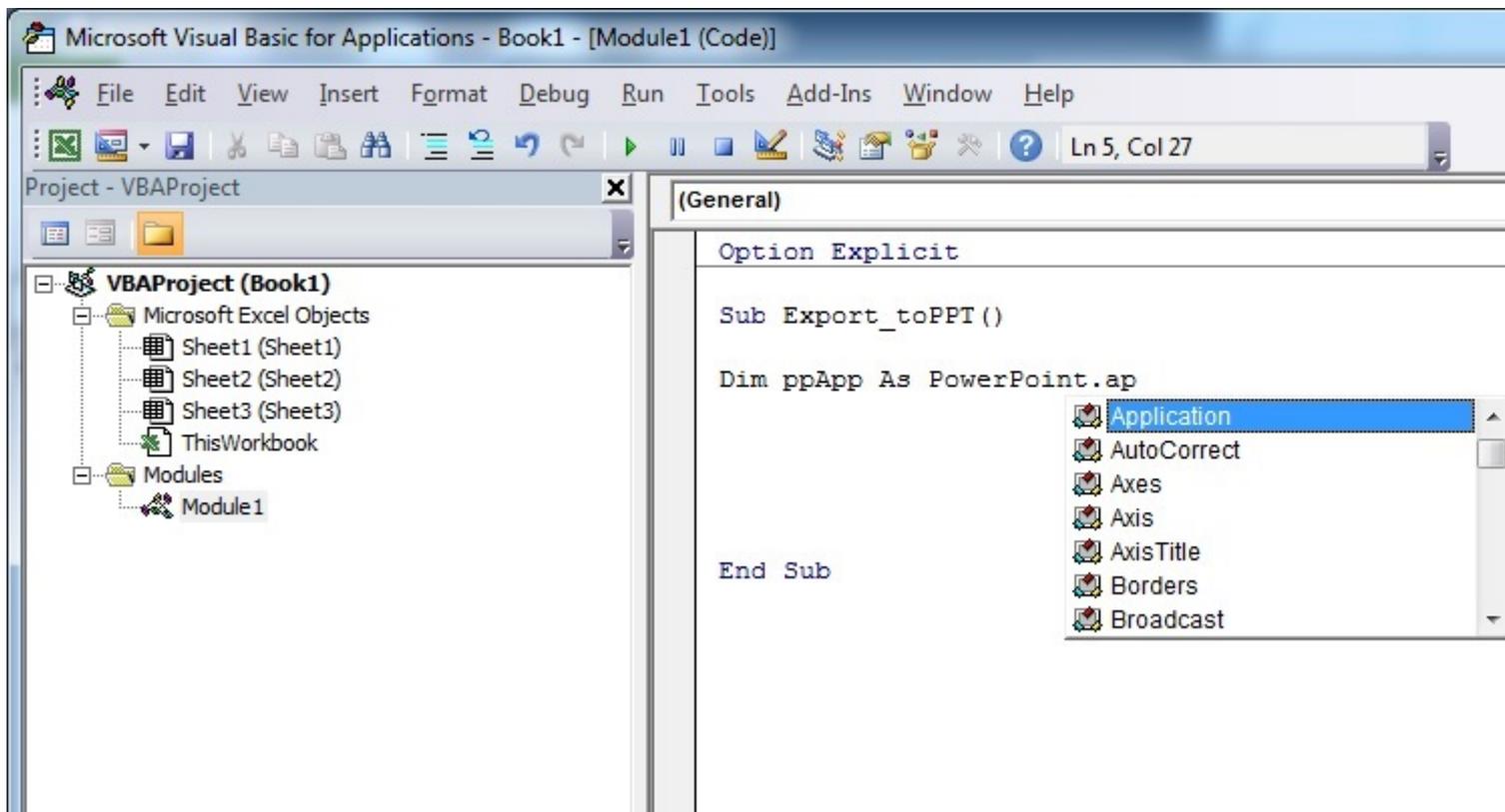


Nota: PowerPoint 14.0 indica che la versione di Office 2010 è installata sul PC.

Passo 3 : nell'Editor VB, una volta premuto **Ctrl + Spazio** insieme, si ottiene l'opzione di completamento automatico di PowerPoint.

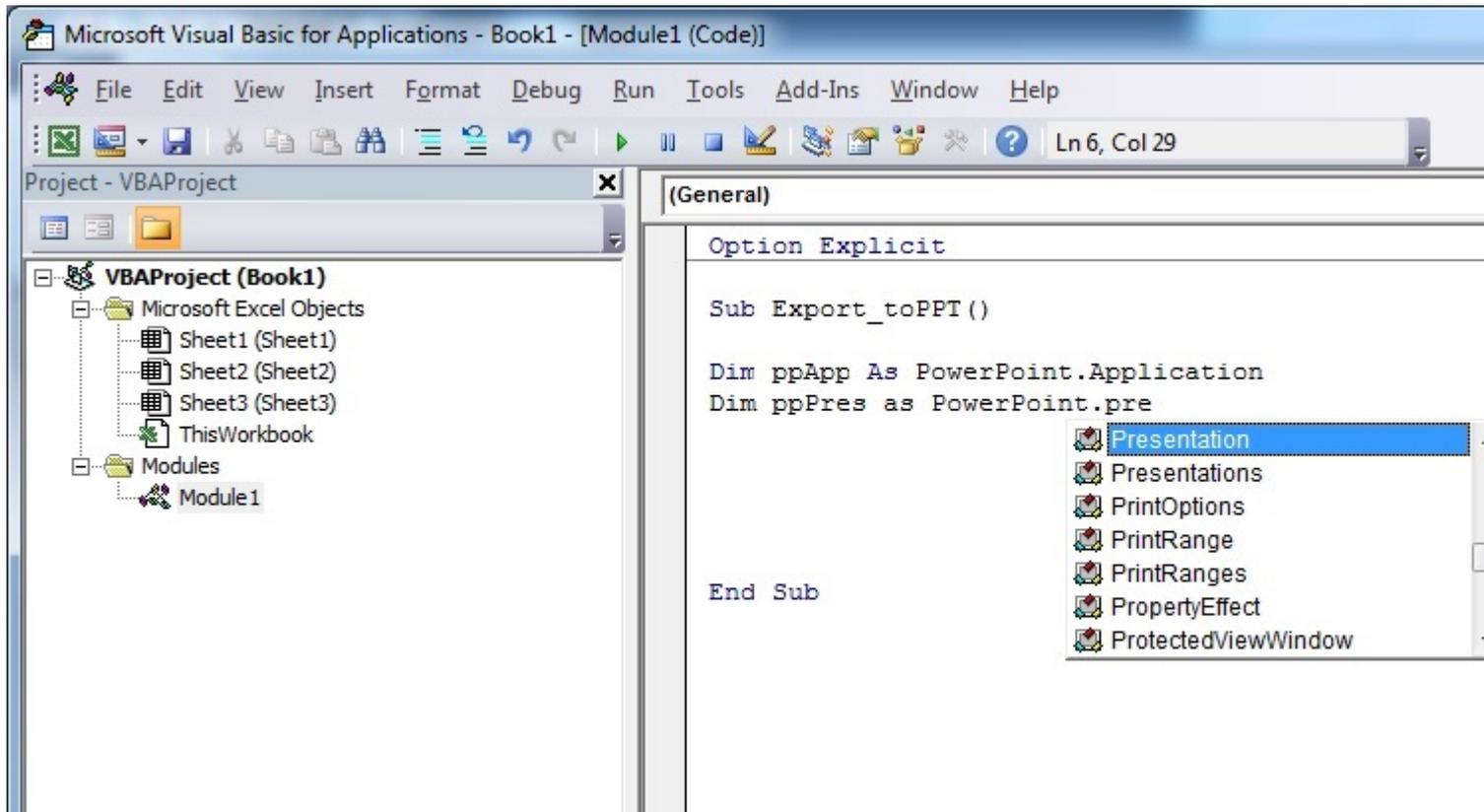


Dopo aver selezionato `PowerPoint` e premuto `.`, viene visualizzato un altro menu con tutte le opzioni relative agli oggetti nella libreria di oggetti di PowerPoint. Questo esempio mostra come selezionare l'`Application` dell'oggetto di PowerPoint.

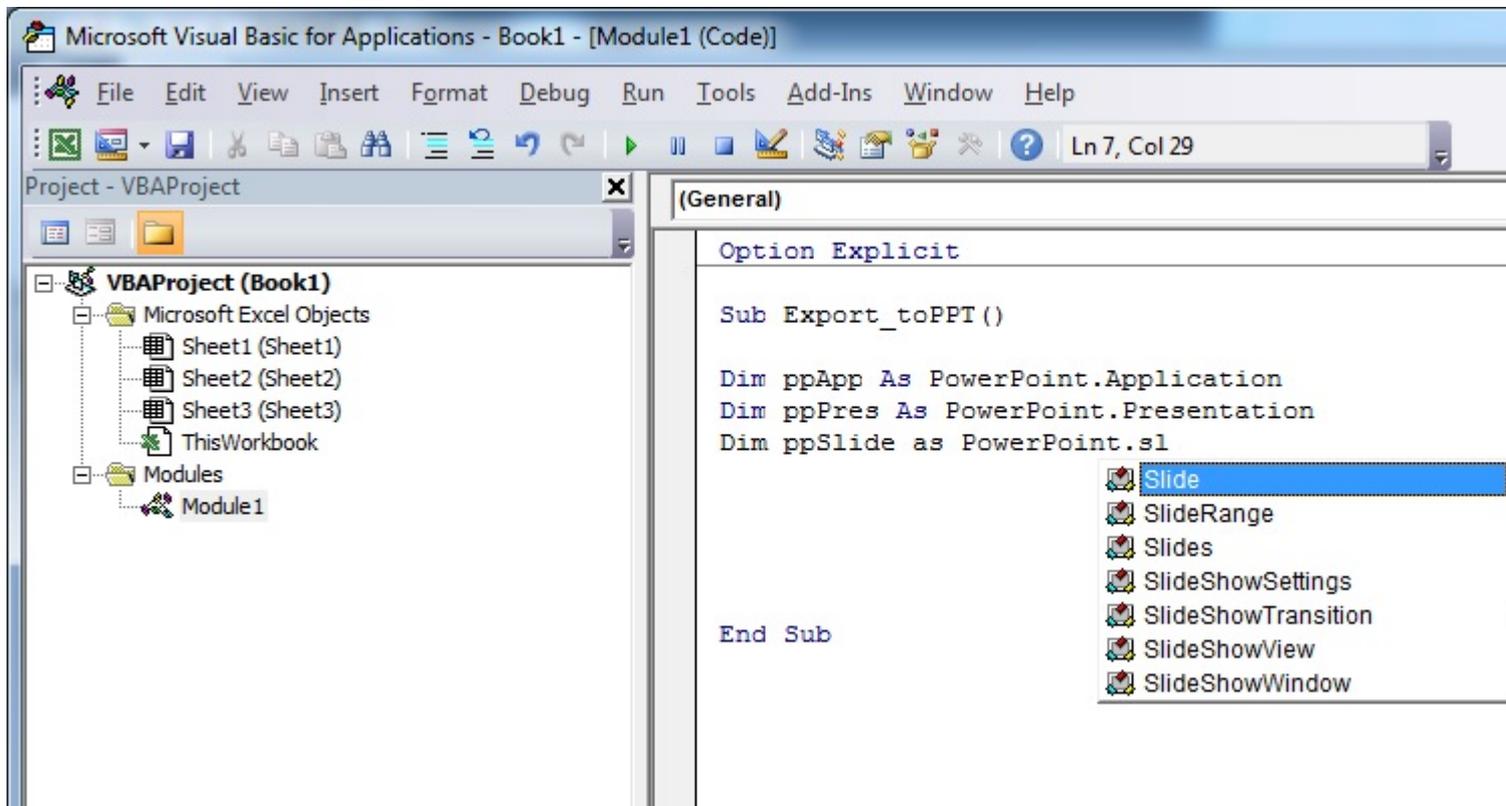


Passaggio 4 : ora l'utente può dichiarare più variabili utilizzando la libreria di oggetti di PowerPoint.

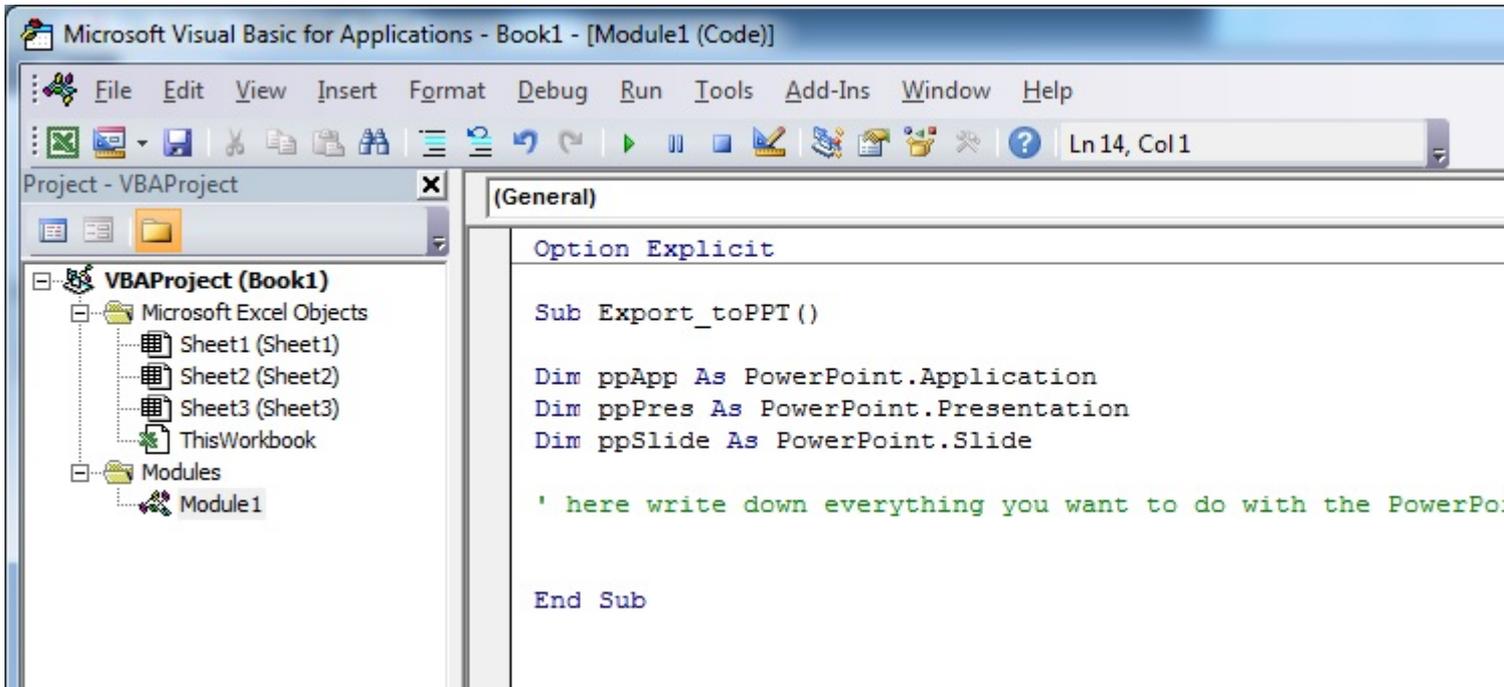
Dichiarare una variabile che fa riferimento all'oggetto `Presentation` della libreria di oggetti di PowerPoint.



Dichiarare un'altra variabile che fa riferimento all'oggetto `Slide` della libreria di oggetti di PowerPoint.



Ora la sezione di dichiarazione delle variabili appare come nella schermata qui sotto, e l'utente può iniziare a usare queste variabili nel suo codice.



Versione del codice di questo tutorial:

```

Option Explicit

Sub Export_toPPT ()

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

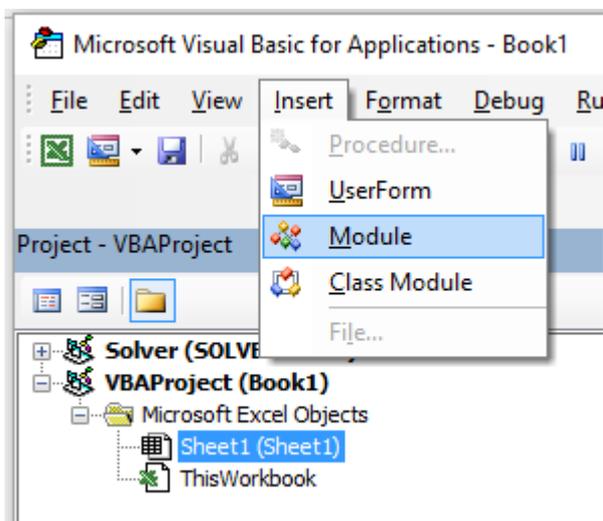
' here write down everything you want to do with the PowerPoint Class and objects

End Sub

```

Ciao mondo

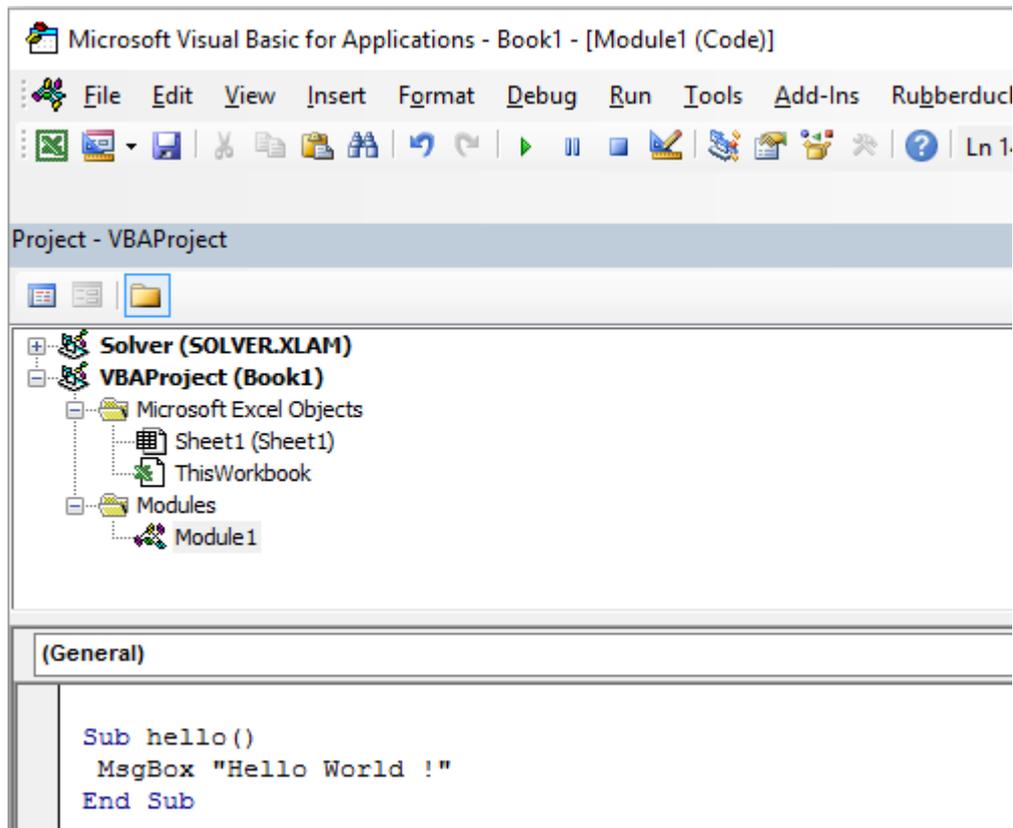
1. Aprire l'editor di Visual Basic (vedere [Apertura dell'editor di Visual Basic](#))
2. Fai clic su Inserisci -> Modulo per aggiungere un nuovo modulo:



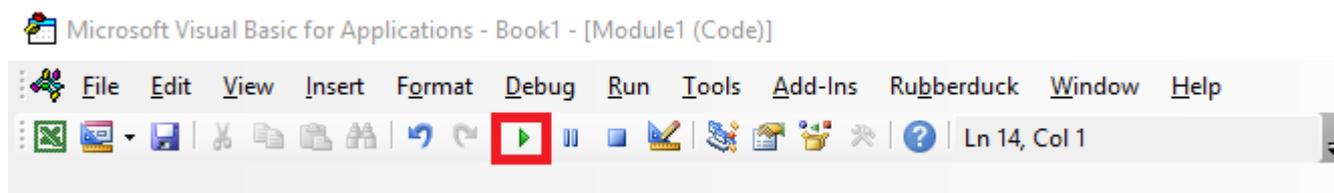
3. Copia e incolla il codice seguente nel nuovo modulo:

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

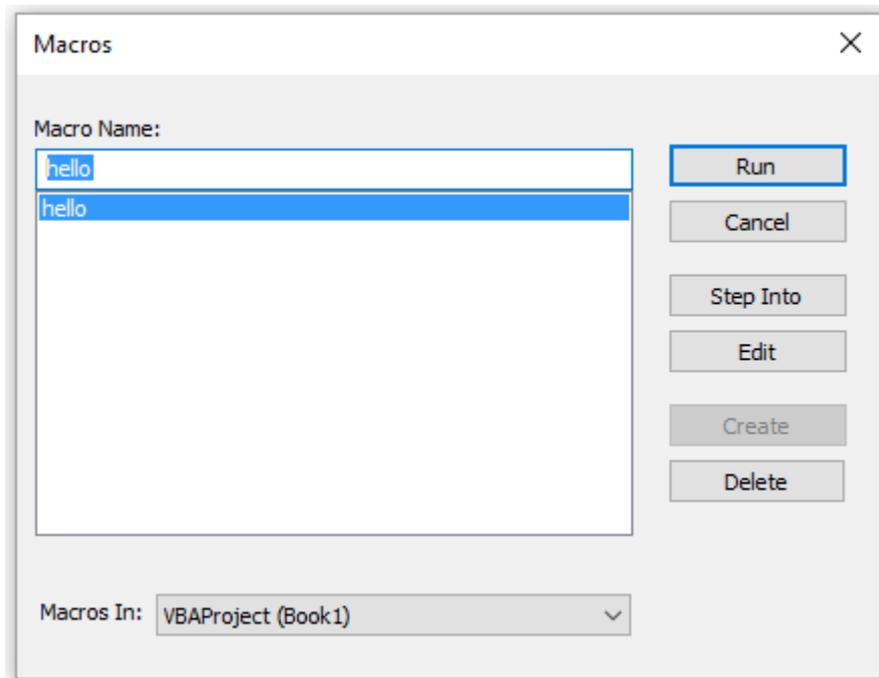
Ottenere :



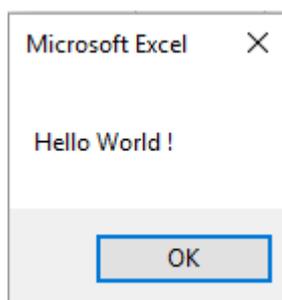
4. Fare clic sulla freccia verde "play" (o premere F5) nella barra degli strumenti di Visual Basic per eseguire il programma:



5. Seleziona il nuovo sottotitolo creato "ciao" e fai clic su `Run` :



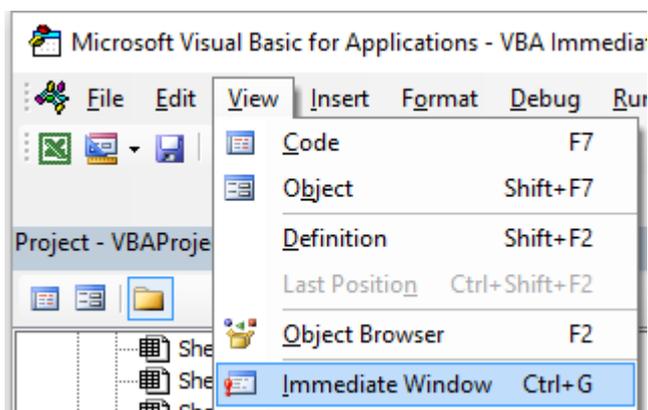
6. Fatto, dovresti vedere la seguente finestra:



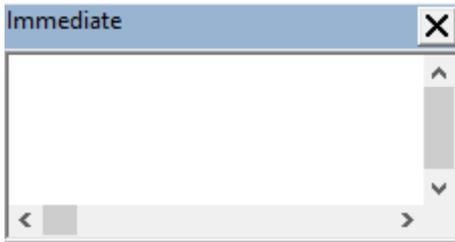
Introduzione al modello a oggetti di Excel

Questo esempio intende essere un'introduzione delicata al modello a oggetti di Excel **per i principianti**.

1. Aprire Visual Basic Editor (VBE)
2. Fai clic su Visualizza -> Finestra immediata per aprire la finestra immediata (o `ctrl + G`):



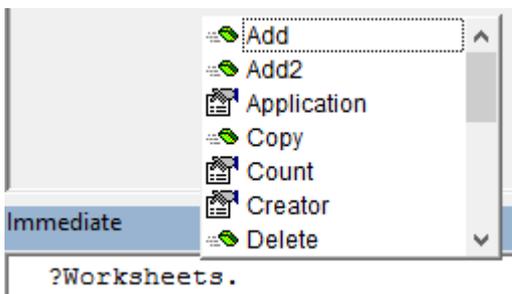
3. Dovresti vedere la seguente finestra immediata in fondo su VBE:



Questa finestra ti consente di testare direttamente alcuni codici VBA. Quindi, iniziamo, digitare questa console:

```
?Worksheets.
```

VBE ha intellisense e quindi dovrebbe aprire un tooltip come nella seguente figura:



Selezionare `.Count` nella lista o digitare direttamente `.Count` per ottenere:

```
?Worksheets.Count
```

4. Quindi premere Invio. L'espressione viene valutata e deve restituire 1. Indica il numero di fogli di lavoro attualmente presenti nella cartella di lavoro. Il punto interrogativo (?) È un alias per `Debug.Print`.

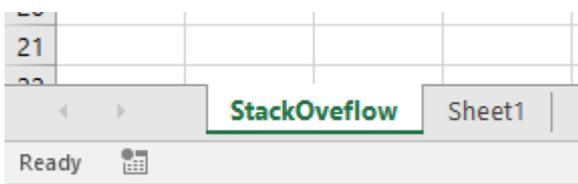
I fogli di lavoro è un **oggetto** e il conteggio è un **metodo** . Excel ha diversi oggetti (`Workbook` , `Worksheet` , `Range` , `Chart` ...) e ognuno di essi contiene metodi e proprietà specifici. È possibile trovare l'elenco completo di oggetti nel [riferimento VBA di Excel](#) . Fogli di lavoro L'oggetto è presentato [qui](#) .

Questo riferimento VBA di Excel dovrebbe diventare la principale fonte di informazioni per quanto riguarda il modello a oggetti di Excel.

5. Ora proviamo un'altra espressione, scrivi (senza il carattere ?):

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

6. Premere Invio. Questo dovrebbe creare un nuovo foglio di lavoro chiamato `StackOverflow` . :



Per comprendere questa espressione è necessario leggere la funzione Aggiungi nel già citato riferimento Excel. Troverete quanto segue:

```
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.
```

Quindi il `Worksheets.Add()` crea un nuovo foglio di lavoro e lo restituisce. Il foglio di lavoro (**senza s**) è di per sé un oggetto che **può essere trovato** nella documentazione e il `Name` è una delle sue **proprietà** (vedi [qui](#)). È definito come:

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

Quindi, esaminando le diverse definizioni di oggetti, siamo in grado di comprendere questo codice `Worksheets.Add().Name = "StackOveflow" .`

`Add()` crea e aggiunge un nuovo foglio di lavoro e restituisce un **riferimento** , quindi impostiamo la **proprietà** `Name` su "StackOverflow"

Ora cerchiamo di essere più formale, Excel contiene diversi oggetti. Questi oggetti possono essere composti da una o più raccolte di oggetti Excel della stessa classe. È il caso dei `WorkSheets` di `WorkSheets` che è una raccolta di oggetti del `Worksheet` di `Worksheet` . Ogni oggetto ha alcune proprietà e metodi con cui il programmatore può interagire.

Il modello di oggetto di Excel fa riferimento alla **gerarchia di oggetti di Excel**

Nella parte superiore di tutti gli oggetti è l'oggetto `Application` , rappresenta l'istanza di Excel stessa. La programmazione in VBA richiede una buona comprensione di questa gerarchia perché abbiamo sempre bisogno di un riferimento a un oggetto per poter chiamare un metodo o impostare / ottenere una proprietà.

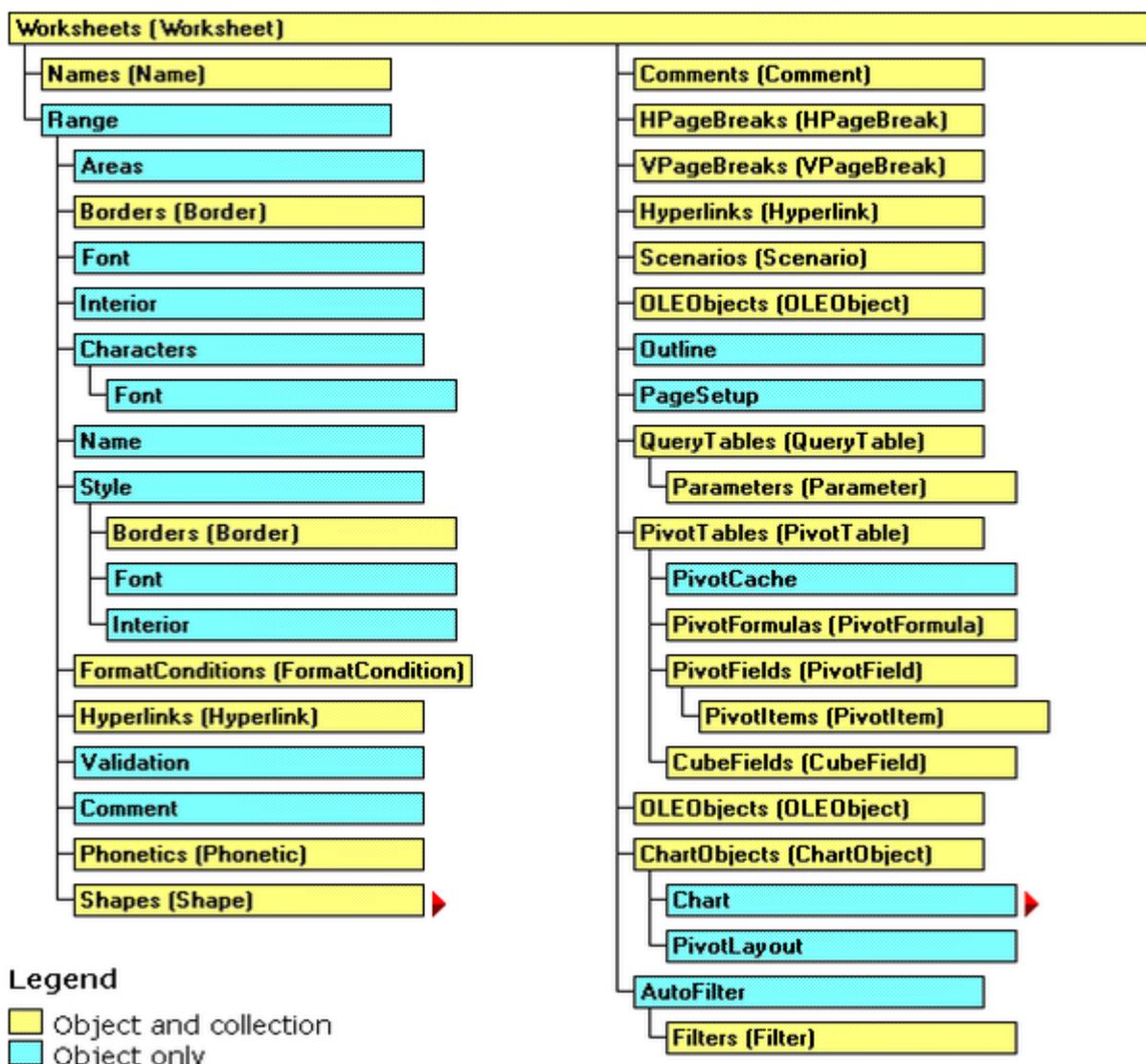
Il modello di oggetti Excel (molto semplificato) può essere rappresentato come,

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

Di seguito è riportata una versione più dettagliata per l'oggetto foglio di lavoro (come in Excel 2007),

Microsoft Excel Objects (Worksheet)

See Also



Il modello di oggetti Excel completo può essere trovato [qui](#) .

Infine alcuni oggetti potrebbero avere `events` (es: `Workbook.WindowActivate`) che fanno anche parte del modello a oggetti di Excel.

Leggi [Iniziare con excel-vba online](https://riptutorial.com/it/excel-vba/topic/777/iniziare-con-excel-vba): <https://riptutorial.com/it/excel-vba/topic/777/iniziare-con-excel-vba>

Capitolo 2: Array

Examples

Compilazione di matrici (aggiunta di valori)

Esistono diversi modi per popolare una matrice.

Direttamente

```
'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 funzione 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"
```

Dalla gamma

```
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
'...

'You 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 Evaluate ()

```
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 funzione Dividi ()

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

Array dinamici (ridimensionamento della matrice e gestione dinamica)

Dato che non si tratta di contenuti esclusivi di Excel-VBA, questo esempio è stato spostato nella documentazione VBA.

Collegamento: [matrici dinamiche \(ridimensionamento della matrice e gestione dinamica\)](#)

Array frastagliati (array di array)

Dato che non si tratta di contenuti esclusivi di Excel-VBA, questo esempio è stato spostato nella documentazione VBA.

Link: [Array frastagliati \(array di array\)](#)

Controlla se la matrice è inizializzata (se contiene elementi o no).

Un problema comune potrebbe essere tentare di ripetere su Array che non contiene valori. Per esempio:

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

Per evitare questo problema e per verificare se una matrice contiene elementi, utilizzare questo *oneliner*:

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

Array dinamici [Dichiarazione array, ridimensionamento]

```
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
```

Leggi Array online: <https://riptutorial.com/it/excel-vba/topic/2027/array>

Capitolo 3: autofilter; Usi e buone pratiche

introduzione

L'obiettivo finale di **Autofilter** è quello di fornire nel modo più rapido possibile il data mining da centinaia o migliaia di dati di righe per attirare l'attenzione sugli articoli su cui vogliamo concentrarci. Può ricevere parametri come "testo / valori / colori" e possono essere impilati tra le colonne. È possibile connettere fino a 2 criteri per colonna in base a connettori logici e insiemi di regole. Osservazione: il filtro automatico funziona filtrando le righe, non c'è alcun filtro automatico per filtrare le colonne (almeno non in modo nativo).

Osservazioni

"Per utilizzare l'Autofiltro all'interno di VBA, è necessario chiamare almeno i seguenti parametri:

Foglio ("MySheet"). Intervallo ("MyRange"). Campo filtro automatico = (ColumnNumberWithin "MyRange" ToBeFilteredInNumericValue) Criteria1: = "WhatIWantToFilter"

"Ci sono molti esempi sul web o qui [su StackOverflow](#)

Examples

SmartFilter!

Situazione problema

L'amministratore del magazzino ha un foglio ("Record") in cui è memorizzato ogni movimento logistico eseguito dalla struttura, può filtrare secondo necessità, anche se questo richiede molto tempo e vorrebbe migliorare il processo per calcolare più rapidamente le richieste, per esempio: quanti "pulp" abbiamo ora (in tutti gli scaffali)? Quanti polpa abbiamo ora (nel rack n. 5)? I filtri sono un ottimo strumento ma, in qualche modo, sono limitati a rispondere a questo tipo di domande in pochi secondi.

	A	B	C	D	E	F	G	H
1	Control Num	DESCRITIO	QUANTT	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		

Soluzione macro:

Il codificatore sa che gli **autofilters sono la soluzione migliore, veloce e più affidabile** in questo tipo di scenari poiché **i dati esistono già nel foglio di lavoro** e **l'input per essi può essere ottenuto facilmente**, in questo caso, tramite l'input dell'utente.

L'approccio utilizzato è quello di creare un foglio chiamato "SmartFilter" in cui l'amministratore può facilmente filtrare più dati secondo necessità e il calcolo verrà eseguito immediatamente.

Usa 2 moduli e l'evento `Worksheet_Change` per questa materia

Codice per foglio di lavoro 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

```

Codice per il modulo 1, chiamato "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: Title '" &
TitleToFind & "' was not found in sheet '" & InSheet & "'", vbCritical: Call ExcelNormal: On
Error GoTo -1: End
    End If '99. If error
End Function

```

Codice per il modulo 2, chiamato "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, Criterial:=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, Criterial:=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, Criterial:=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

```

Test e risultati:

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

Come abbiamo visto nell'immagine precedente, questo compito è stato raggiunto facilmente. Usando i **filtri automatici** è stata fornita una soluzione che **richiede** solo **pochi secondi per essere calcolata, è facile da spiegare all'utente**, dal momento che ha familiarità con questo comando, e ha **preso poche righe dal codificatore**.

Leggi autofilter; Usi e buone pratiche online: <https://riptutorial.com/it/excel-vba/topic/8645/autofilter--usi-e-buone-pratiche>

Capitolo 4: Best practice VBA

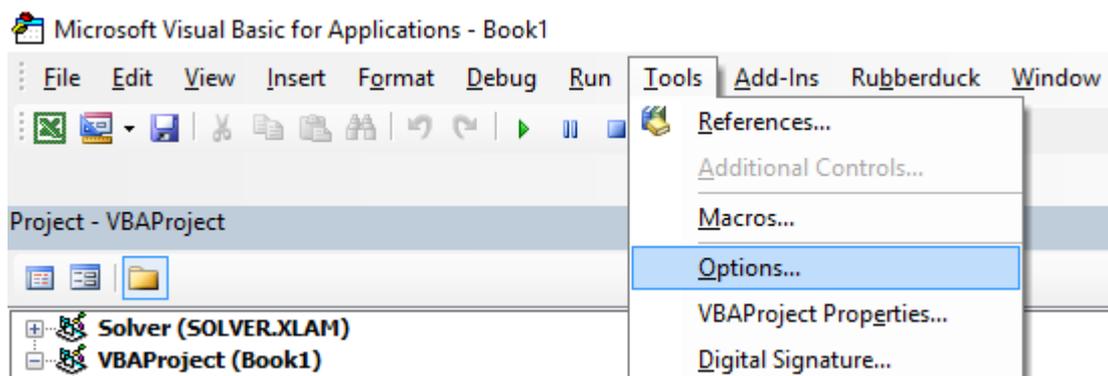
Osservazioni

Li conosciamo tutti, ma queste pratiche sono molto meno ovvie a chi inizia a programmare in VBA.

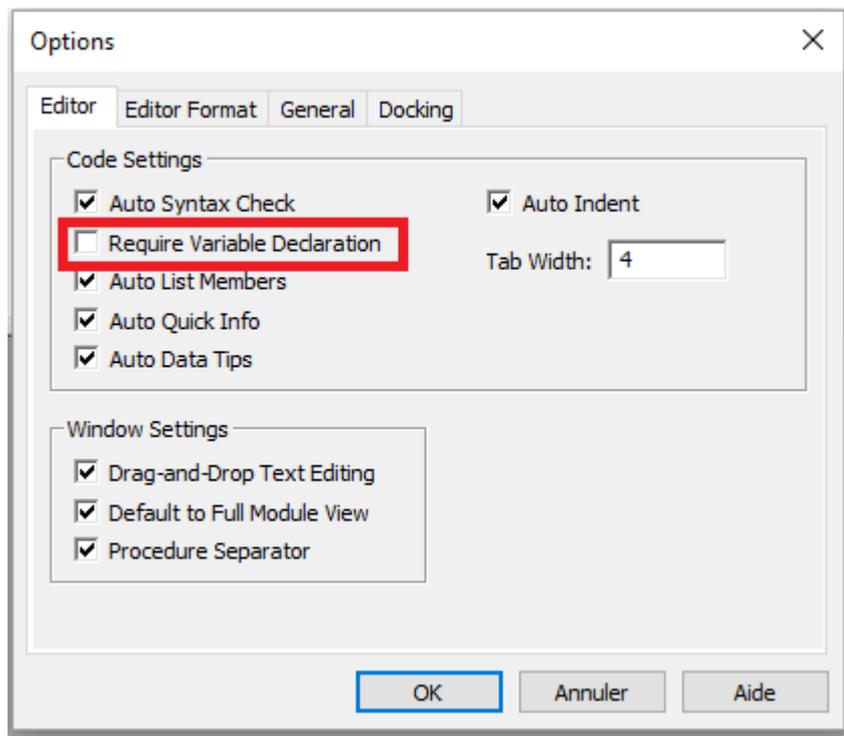
Examples

Utilizzare SEMPRE "Option Explicit"

Nella finestra Editor VBA, dal menu Strumenti seleziona "Opzioni":



Quindi nella scheda "Editor", assicurati che "Richiedi dichiarazione di variabili" sia spuntato:



Selezionando questa opzione verrà automaticamente aggiunta l' `Option Explicit` nella parte superiore di ogni modulo VBA.

Piccola nota: questo è vero per i moduli, i moduli di classe, ecc. Che non sono stati aperti finora. Quindi, se hai già dato un'occhiata ad esempio al codice di `Sheet1` prima di attivare l'opzione "Require Variable Declaration", `Option Explicit` non verrà aggiunta!

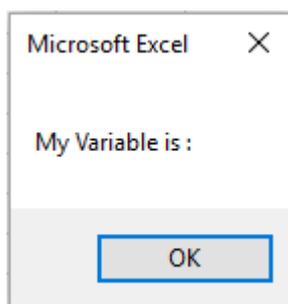
`Option Explicit` richiede che ogni variabile debba essere definita prima dell'uso, ad esempio con un'istruzione `Dim`. Senza `Option Explicit` abilitato, il compilatore VBA assumerà qualsiasi parola non riconosciuta come una nuova variabile del tipo `Variant`, causando bug estremamente difficili da individuare in relazione agli errori tipografici. Con `Option Explicit` abilitata, qualsiasi parola non riconosciuta causerà un errore di compilazione, che indica la linea offendente.

Esempio :

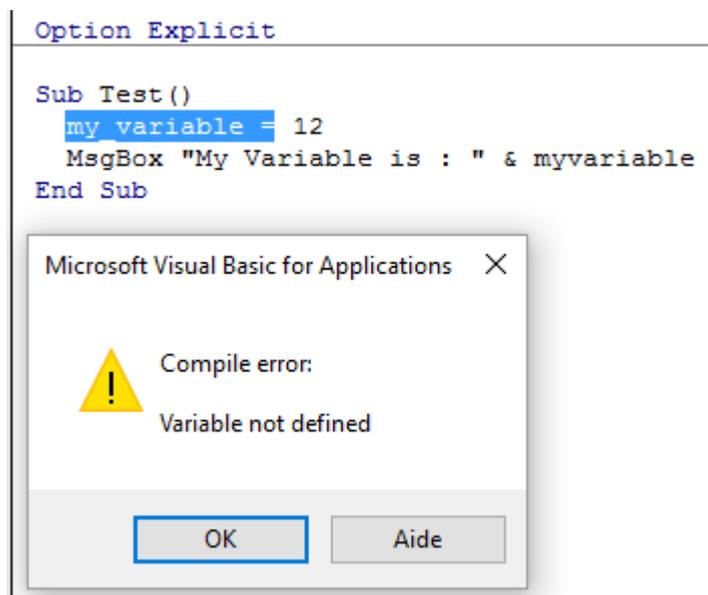
Se si esegue il seguente codice:

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

Riceverai il seguente messaggio:



Hai fatto un errore scrivendo `myvariable` anziché `my_variable`, quindi la finestra del messaggio mostra una variabile vuota. Se si utilizza `Option Explicit`, questo errore non è possibile perché si otterrà un messaggio di errore di compilazione che indica il problema.



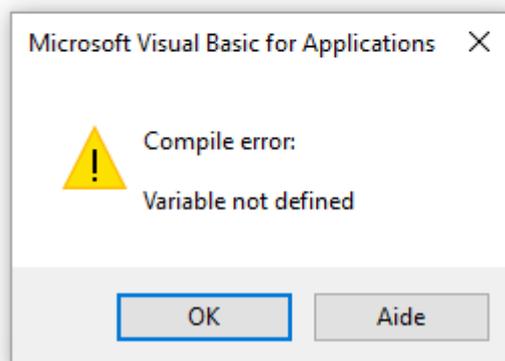
Ora se aggiungi la dichiarazione corretta:

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

myvariable un messaggio di errore che indica precisamente l'errore con myvariable :

Option Explicit

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



Nota su Option Explicit e Arrays ([Dichiarazione di una matrice dinamica](#)):

È possibile utilizzare l'istruzione ReDim per dichiarare implicitamente un array all'interno di una procedura.

- Fare attenzione a non digitare errori ortografici nel nome dell'array quando si utilizza l'istruzione ReDim
- Anche se l'istruzione Option Explicit è inclusa nel modulo, verrà creato un nuovo array

```
Dim arr() as Long
```

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

Lavora con le matrici, non con le gamme

[Blog di Office: best practice per la codifica delle prestazioni di Excel VBA](#)

Spesso, le migliori prestazioni si ottengono evitando l'uso di `Range` il più possibile. In questo esempio si legge in un intero `Range` oggetto in una matrice quadrata ciascun numero nella matrice, e quindi restituire l'array alla `Range`. Questo accede a `Range` solo due volte, mentre un loop accederà 20 volte per la lettura / scrittura.

```
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

```

Ulteriori suggerimenti e informazioni con esempi temporizzati possono essere trovati negli [UDF di VBA efficienti di Charles Williams \(parte 1\)](#) e in [altri articoli della serie](#) .

Utilizzare le costanti VB quando disponibili

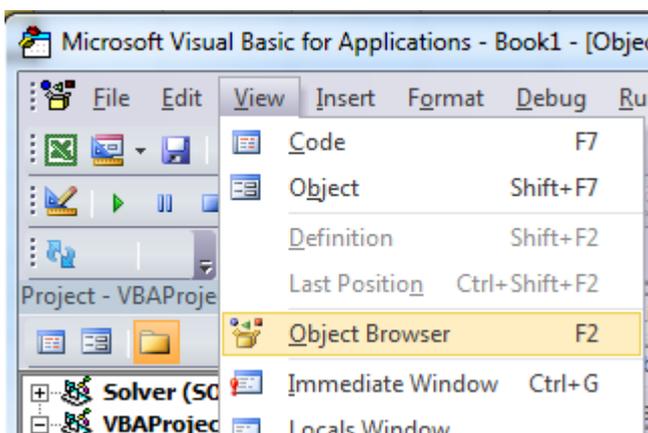
```
If MsgBox("Click OK") = vbOK Then
```

può essere usato al posto di

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

al fine di migliorare la leggibilità.

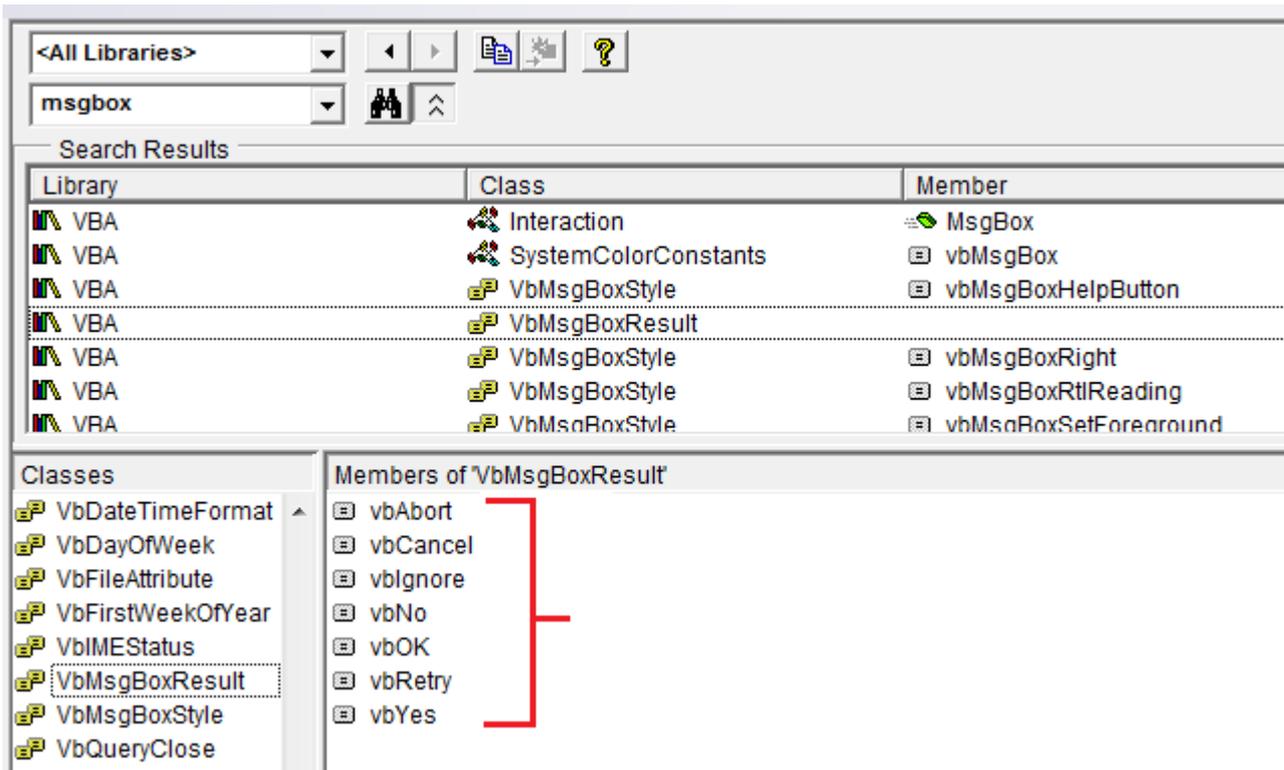
Utilizzare il Visualizzatore *oggetti* per trovare le costanti VB disponibili. *Visualizza* → *Browser oggetti* o F2 da VB Editor.



Inserisci la classe per la ricerca



Visualizza i membri disponibili



Usa la denominazione delle variabili descrittive

I nomi e la struttura descrittivi nel codice aiutano a rendere inutili i commenti

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

ductArea = ductWidth * ductHeight
```

è meglio di

```
Dim a, w, h

a = w * h
```

Ciò è particolarmente utile quando si copiano dati da un luogo a un altro, che si tratti di una cella, un intervallo, un foglio di lavoro o una cartella di lavoro. Aiutati usando nomi come questi:

```
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
```

Se dichiari più variabili in una riga, assicurati di specificare un tipo per *ogni* variabile come:

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

Quanto segue dichiarerà solo l'ultima variabile e le prime rimarranno `Variant` :

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

Gestione degli errori

Una buona gestione degli errori impedisce agli utenti finali di visualizzare errori di runtime di VBA e aiuta lo sviluppatore a diagnosticare e correggere facilmente gli errori.

Esistono tre metodi principali di gestione degli errori in VBA, due dei quali dovrebbero essere evitati per i programmi distribuiti, a meno che non siano specificatamente richiesti nel codice.

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

o

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

Preferisci usare:

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

On Error GoTo 0

Se nel tuo codice non è impostata la gestione degli errori, `On Error GoTo 0` è il gestore degli errori predefinito. In questa modalità, qualsiasi errore di runtime avvierà il tipico messaggio di errore VBA, consentendo di terminare il codice o accedere alla modalità di `debug`, identificando la sorgente. Durante la scrittura del codice, questo metodo è il più semplice e utile, ma dovrebbe essere sempre evitato per il codice distribuito agli utenti finali, poiché questo metodo è molto sgradevole e difficile da comprendere per gli utenti finali.

In caso di errore, riprendi

`On Error Resume Next` farà in modo che VBA ignori gli eventuali errori generati in fase di esecuzione per tutte le righe successive alla chiamata di errore fino a quando il gestore degli errori non è stato modificato. In casi molto specifici, questa linea può essere utile, ma dovrebbe essere evitata al di

fuori di questi casi. Ad esempio, quando si avvia un programma separato da una macro di Excel, la chiamata `On Error Resume Next` può essere utile se non si è certi che il programma sia già aperto o meno:

```
'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
```

Se non avessimo utilizzato la chiamata `On Error Resume Next` e l'applicazione Powerpoint non fosse già aperta, il metodo `GetObject` genererebbe un errore. Pertanto, `On Error Resume Next` stato necessario per evitare di creare due istanze dell'applicazione.

Nota: è anche consigliabile ripristinare *immediatamente* il gestore degli errori non appena non è più necessaria la chiamata `On Error Resume Next`

On Error GoTo <line>

Questo metodo di gestione degli errori è consigliato per tutto il codice che viene distribuito ad altri utenti. Ciò consente al programmatore di controllare esattamente come VBA gestisce un errore inviando il codice alla linea specificata. Il tag può essere riempito con qualsiasi stringa (comprese le stringhe numeriche) e invierà il codice alla stringa corrispondente seguita da due punti. È possibile utilizzare più blocchi di gestione degli errori effettuando chiamate diverse di `On Error GoTo <line>`. La subroutine di seguito mostra la sintassi di una chiamata `On Error GoTo <line>`.

Nota: è essenziale che la riga `Exit Sub` sia posizionata sopra il primo gestore di errori e prima di ogni gestore di errori successivo per impedire al codice di avanzare naturalmente nel blocco *senza che* venga chiamato un errore. Pertanto, è buona pratica per la funzione e la leggibilità posizionare i gestori degli errori alla fine di un blocco di codice.

```
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.Description
        Application.ScreenUpdating = True
        Application.EnableEvents = True
        Exit Sub
    End If
    Exit Sub

End Sub

```

Se esci dal metodo con il codice di gestione degli errori, assicurati di pulire:

- Annulla tutto ciò che è parzialmente completato
- Chiudi i file
- Ripristina l'aggiornamento della schermata
- Ripristina la modalità di calcolo
- Reimposta eventi
- Ripristina il puntatore del mouse
- Chiama il metodo di scaricamento su istanze di oggetti che persistono dopo `End Sub`
- Reimposta la barra di stato

Documenta il tuo lavoro

È buona norma documentare il proprio lavoro per un uso successivo, specialmente se si sta codificando per un carico di lavoro dinamico. Buoni commenti dovrebbero spiegare perché il codice sta facendo qualcosa, non quello che sta facendo il codice.

```

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

```

Se il tuo codice è così oscuro da richiedere commenti per spiegare cosa sta facendo, considera di riscriverlo per essere più chiaro invece di spiegarlo attraverso i commenti. Ad esempio, invece di:

```

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

```

Chiarire il codice per essere più facile da seguire, come ad esempio:

```
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
```

Disattiva le proprietà durante l'esecuzione della macro

È una buona pratica in qualsiasi linguaggio di programmazione per **evitare l'ottimizzazione prematura**. Tuttavia, se i test rivelano che il tuo codice è in esecuzione troppo lentamente, potresti guadagnare un po' di velocità disattivando alcune delle proprietà dell'applicazione mentre è in esecuzione. Aggiungi questo codice a un modulo standard:

```
Public Sub SpeedUp( _
    SpeedUpOn As Boolean, _
    Optional xlCalc as XlCalculation = xlCalculationAutomatic _
)
    With Application
        If SpeedUpOn Then
            .ScreenUpdating = False
            .Calculation = xlCalculationManual
            .EnableEvents = False
            .DisplayStatusBar = False 'in case you are not showing any messages
            ActiveSheet.DisplayPageBreaks = False 'note this is a sheet-level setting
        Else
            .ScreenUpdating = True
            .Calculation = xlCalc
            .EnableEvents = True
            .DisplayStatusBar = True
            ActiveSheet.DisplayPageBreaks = True
        End If
    End With
End Sub
```

Ulteriori informazioni sul [blog di Office - Best practice per la codifica delle prestazioni di Excel VBA](#)

E basta chiamarlo all'inizio e alla fine dei macro:

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

    SpeedUp True

    'code here ...
End Sub
```

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

Mentre questi possono essere in gran parte considerati "miglioramenti" per le normali procedure `Public Sub`, la disabilitazione della gestione degli eventi con `Application.EnableEvents = False` deve essere considerata obbligatoria per le macro di eventi privati `Worksheet_Change` e `Workbook_SheetChange` che modificano i valori su uno o più fogli di lavoro. La mancata disabilitazione dei trigger di evento farà sì che la macro dell'evento ricorra in modo ricorsivo su se stessa quando un valore cambia e può portare a una cartella di lavoro "congelata". Ricordarsi di riattivare gli eventi prima di lasciare la macro dell'evento, possibilmente tramite un gestore degli errori di "uscita sicura".

```
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
```

Un avvertimento: se disabilitare queste impostazioni migliorerà il tempo di esecuzione, potrebbe rendere molto più difficile il debug dell'applicazione. Se il tuo codice *non* funziona correttamente, commenta la chiamata `SpeedUp True` finché non indovina il problema.

Ciò è particolarmente importante se si scrivono le celle in un foglio di lavoro e quindi si leggono i risultati calcolati dalle funzioni del foglio di lavoro poiché `xlCalculationManual` impedisce il calcolo della cartella di lavoro. Per aggirare questo problema senza disabilitare `SpeedUp`, è possibile includere `Application.Calculate` per eseguire un calcolo in punti specifici.

NOTA: poiché queste sono proprietà `Application` stessa, è necessario assicurarsi che siano nuovamente abilitate prima che la macro esca. Ciò rende particolarmente importante l'utilizzo di gestori di errori e per evitare più punti di uscita (ad es. `End 0 Unload Me`).

Con la gestione degli errori:

```
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

```

Evitare l'uso di ActiveCell o ActiveSheet in Excel

L'utilizzo di `ActiveCell` o `ActiveSheet` può essere fonte di errori se (per qualsiasi motivo) il codice viene eseguito nel posto sbagliato.

```

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

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

```

- L'uso di `Active*` può creare problemi nelle macro di lunga durata se l'utente si annoia e fa clic su un altro foglio di lavoro o apre un'altra cartella di lavoro.
- Può creare problemi se il codice si apre o crea un'altra cartella di lavoro.
- Può creare problemi se il tuo codice utilizza `Sheets("MyOtherSheet").Select` e hai dimenticato il foglio in cui ti `Sheets("MyOtherSheet").Select` prima di iniziare a leggere o scrivere su di esso.

Non assumere mai il foglio di lavoro

Anche quando tutto il tuo lavoro è diretto a un singolo foglio di lavoro, è comunque molto utile specificare esplicitamente il foglio di lavoro nel tuo codice. Questa abitudine rende molto più facile espandere il codice in un secondo momento o sollevare parti (o tutto) di un `Sub` o di una `Function` da riutilizzare altrove. Molti sviluppatori stabiliscono l'abitudine di (ri) utilizzare lo stesso nome di variabile locale per un foglio di lavoro nel loro codice, rendendo il riutilizzo di quel codice ancora più semplice.

Ad esempio, il codice seguente è ambiguo, ma funziona! - finché lo sviluppatore non si attiva o non passa a un altro foglio di lavoro:

```

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

```

Se `Sheet1` è attivo, la cella A1 su `Sheet1` verrà riempita con la data e l'ora correnti. Ma se l'utente cambia i fogli di lavoro per qualsiasi motivo, il codice aggiornerà qualunque sia il foglio di lavoro attualmente attivo. Il foglio di lavoro di destinazione è ambiguo.

La migliore pratica è identificare sempre il foglio di lavoro a cui si riferisce il codice:

```
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
```

Il codice sopra riportato è chiaro nell'identificare sia la cartella di lavoro che il foglio di lavoro. Anche se può sembrare eccessivo, la creazione di una buona abitudine riguardante i riferimenti target ti farà risparmiare problemi futuri.

Evitare l'uso di SELECT o ACTIVATE

E **molto** raro che tu abbia mai desidera utilizzare `Select` o `Activate` nel codice, ma alcuni metodi di Excel richiedono un foglio di lavoro o cartella di lavoro da attivare prima che funzioneranno come previsto.

Se stai appena iniziando a imparare VBA, ti verrà spesso suggerito di registrare le tue azioni usando il registratore di macro, quindi vai a guardare il codice. Ad esempio, ho registrato le azioni eseguite per immettere un valore nella cella D3 su Sheet2 e il codice macro ha il seguente aspetto:

```
Option Explicit
Sub Macro1()
    '
    ' Macro1 Macro
    '
    '
    '
    Sheets("Sheet2").Select
    Range("D3").Select
    ActiveCell.FormulaR1C1 = "3.1415" ' (see **note below)
    Range("D4").Select
End Sub
```

Ricorda però che il registratore di macro crea una linea di codice per OGNI delle tue azioni (utente). Ciò include il fare clic sulla scheda del foglio di lavoro per selezionare Foglio2 (`Sheets("Sheet2").Select`), facendo clic sulla cella D3 prima di immettere il valore (`Range("D3").Select`) e utilizzando il tasto Invio (che è effettivamente " selezionando "la cella sotto la cella attualmente selezionata: `Range("D4").Select`).

Ci sono più problemi nell'uso. `.Select` qui:

- **Il foglio di lavoro non è sempre specificato.** Ciò accade se non si cambiano i fogli di lavoro durante la registrazione e ciò significa che il codice produrrà risultati diversi per diversi fogli di lavoro attivi.
- `.select ()` è lento. Anche se `Application.ScreenUpdating` è impostato su `False` , si tratta di

un'operazione non necessaria da elaborare.

- `.select()` è **indisciplinato**. Se `Application.ScreenUpdating` viene lasciato su `True`, Excel selezionerà effettivamente le celle, il foglio di lavoro, il modulo ... con qualunque cosa stai lavorando. Questo è stressante per gli occhi e davvero sgradevole da guardare.
- `.select()` **attiverà gli ascoltatori**. Questo è già un po' avanzato, ma a meno che non funzionino, verranno attivate funzioni come `Worksheet_SelectionChange()`.

Quando si codifica in VBA, tutte le azioni di "digitazione" (cioè le istruzioni `select`) non sono più necessarie. Il tuo codice può essere ridotto a una singola istruzione per inserire il valore nella cella:

```
'--- 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
```

(L'esempio BETTER in alto mostra l'utilizzo di variabili intermedie per separare parti diverse del riferimento di cella. L'esempio GOOD funzionerà sempre bene, ma può essere molto ingombrante in moduli di codice molto più lunghi e più difficile da eseguire il debug se uno dei riferimenti è stato digitato erroneamente.)

**** NOTA:** il registratore di macro fa molte ipotesi sul tipo di dati che stai inserendo, in questo caso inserendo un valore di stringa come formula per creare il valore. Il tuo codice non deve farlo e può semplicemente assegnare un valore numerico direttamente alla cella come mostrato sopra.

**** NOTA2:** la pratica raccomandata è di impostare la variabile della cartella di lavoro locale su `ThisWorkbook` invece di `ActiveWorkbook` (a meno che non ne abbiate esplicitamente bisogno). Il motivo è che la macro in genere richiede / utilizza risorse in qualsiasi cartella di lavoro che il codice VBA ha origine e NON guarderà al di fuori di tale cartella di lavoro, sempre, a meno che non si diriga esplicitamente il proprio codice per lavorare con un'altra cartella di lavoro. Quando in Excel sono aperte più cartelle di lavoro, `ActiveWorkbook` è quello con lo stato attivo *che potrebbe essere diverso dalla cartella di lavoro visualizzata nell'editor VBA*. Quindi pensi di essere eseguito in una cartella di lavoro quando stai davvero citando un altro. `ThisWorkbook` di lavoro si riferisce alla cartella di lavoro contenente il codice in esecuzione.

Definisci sempre e imposta i riferimenti a tutte le cartelle di lavoro e fogli

Quando si lavora con più cartelle di lavoro aperte, ognuna delle quali può avere più fogli, è più sicuro definire e impostare il riferimento a tutte le cartelle di lavoro e fogli di lavoro.

Non fare affidamento su `ActiveWorkbook` o `ActiveSheet` in quanto potrebbero essere modificati

dall'utente.

Il seguente illustra come copiare un range da foglio "RAW_DATA" nella cartella di lavoro "Data.xlsx" alla scheda "Refined_Data" nella cartella di lavoro "Results.xlsx".

La procedura mostra anche come copiare e incollare senza utilizzare il metodo `Select`.

```
Option Explicit

Sub CopyRanges_BetweenShts()

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

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

    ' set reference to all sheets by name, don't rely on ActiveSheet
    Set shtCopy = wbSrc.Sheet1 '// "Raw_Data" sheet
    Set shtPaste = wbDest.Sheet2 '// "Refined_Data" sheet

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

End Sub
```

L'oggetto Worksheet Function viene eseguito più velocemente di un equivalente UDF

VBA è compilato in fase di esecuzione, il che ha un enorme impatto negativo sulle sue prestazioni, tutto il built-in sarà più veloce, provare a usarli.

Come esempio sto confrontando le funzioni SUM e CONTA.SE, ma puoi usare se per qualcosa che puoi risolvere con WorksheetFunctions.

Un primo tentativo per quelli sarebbe quello di scorrere l'intervallo e processarlo cella per cella (utilizzando un intervallo):

```
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
    End For
End Sub
```

```

Next rng
Debug.Print Total & ", " & CountLessThan01
End Sub

```

Un miglioramento può essere quello di memorizzare i valori dell'intervallo in una matrice e elaborare ciò:

```

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

```

Ma invece di scrivere qualsiasi loop puoi usare `Application.WorksheetFunction` che è molto utile per l'esecuzione di formule semplici:

```

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

```

Oppure, per calcoli più complessi puoi persino usare `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

```

Infine, passando sopra i Subs a 25.000 volte ciascuno, ecco il tempo medio (5 test) in millisecondi

(ovviamente sarà diverso per ogni pc, ma confrontati tra loro si comportano allo stesso modo):

1. Funzione UseWorksheet: 2156 ms
2. UseArray: 2219 ms (+ 3%)
3. Usa valutazione: 4693 ms (+ 118%)
4. UseRange: 6530 ms (+ 203%)

Evita di riproporre i nomi di Proprietà o Metodi come variabili

Generalmente non è considerata la "migliore pratica" per riutilizzare i nomi riservati di Proprietà o Metodi come nome / i delle proprie procedure e variabili.

Bad Form - Mentre il seguente è (strettamente parlando) codice legale, funzionante, la ri-proposizione del metodo **Find** così come le proprietà **Row** , **Column** e **Address** possono causare problemi / conflitti con l'ambiguità del nome e sono semplicemente confuse in generale.

```
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
```

Forma buona - Con tutte le parole riservate rinominate in approssimazioni ravvicinate ma univoche degli originali, sono stati evitati potenziali conflitti di denominazione.

```
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
```

Mentre può arrivare un momento in cui vuoi riscrivere intenzionalmente un metodo o una proprietà

standard secondo le tue specifiche, quelle situazioni sono poche e lontane tra loro. Per la maggior parte, evita di riutilizzare i nomi riservati per i tuoi costrutti.

Leggi Best practice VBA online: <https://riptutorial.com/it/excel-vba/topic/1107/best-practice-vba>

Capitolo 5: Celle / gamme unite

Examples

Pensaci due volte prima di utilizzare celle / intervalli uniti

Prima di tutto, le celle unite servono solo a migliorare l'aspetto dei tuoi fogli.

Quindi è letteralmente l'ultima cosa che dovresti fare, una volta che il tuo foglio e la tua cartella di lavoro sono totalmente funzionanti!

Dove sono i dati in un intervallo unito?

Quando unisci un intervallo, visualizzerai solo un blocco.

I dati saranno nella primissima **cella di quell'intervallo** e gli **altri saranno celle vuote** !

Un buon punto a riguardo: non è necessario riempire tutte le celle o l'intervallo una volta uniti, basta riempire la prima cella! ;)

Gli altri aspetti di questo insieme sono negativi a livello globale:

- Se usi un [metodo per trovare l'ultima riga o colonna](#) , rischierai alcuni errori
- Se esegui il looping delle righe e hai unito alcuni intervalli per una migliore leggibilità, incontrerai celle vuote e non il valore visualizzato dall'intervallo unito

Leggi [Celle / gamme unite online](https://riptutorial.com/it/excel-vba/topic/7308/celle---gamme-unite): <https://riptutorial.com/it/excel-vba/topic/7308/celle---gamme-unite>

Capitolo 6: Come registrare una macro

Examples

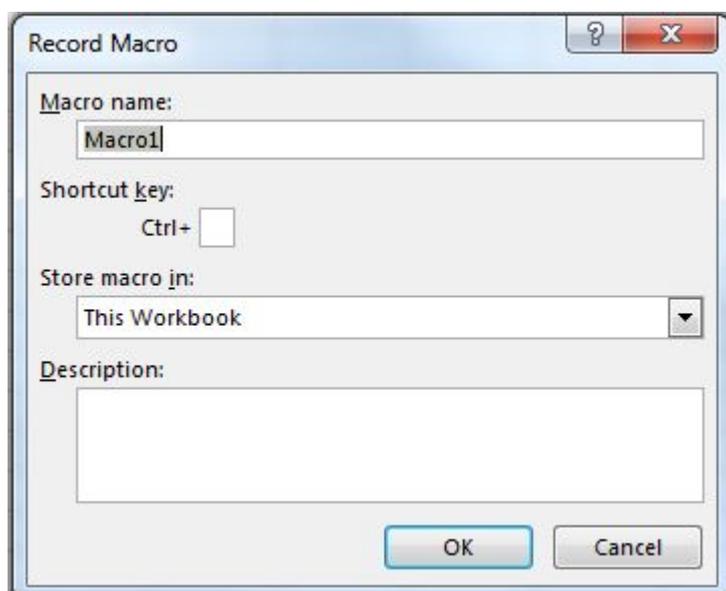
Come registrare una macro

Il modo più semplice per registrare una macro è che il pulsante nell'angolo in basso a sinistra di



Excel assomiglia a questo:

Quando fai clic su questo si aprirà un popup che ti chiede di dare un nome alla Macro e decidere se vuoi avere un tasto di scelta rapida. Inoltre, chiede dove memorizzare la macro e per una descrizione. Puoi scegliere qualsiasi nome tu voglia, nessuno spazio è permesso.



Se vuoi avere una scorciatoia assegnata alla tua macro per un uso rapido, scegli una lettera che ricorderai in modo da poter usare la macro più e più volte.

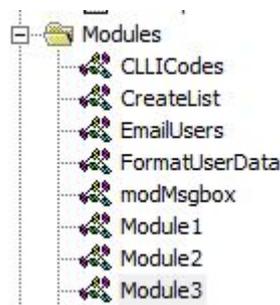
È possibile memorizzare la macro in "Questa cartella di lavoro", "Nuova cartella di lavoro" o "Cartella macro personale". Se vuoi che la macro che stai per registrare sia disponibile solo nella cartella di lavoro corrente, scegli "Questa cartella di lavoro". Se vuoi salvarlo in una nuova cartella di lavoro, scegli "Nuova cartella di lavoro". E se vuoi che la macro sia disponibile per qualsiasi cartella di lavoro che apri, scegli "Cartella macro personale".

Dopo aver compilato questo pop-up, fai clic su "Ok".

Quindi esegui tutte le azioni che desideri ripetere con la macro. Al termine, fai clic sullo stesso pulsante per interrompere la registrazione. Ora sembra così:



Ora puoi andare alla scheda Sviluppatore e aprire Visual Basic. (o usa Alt + F11)



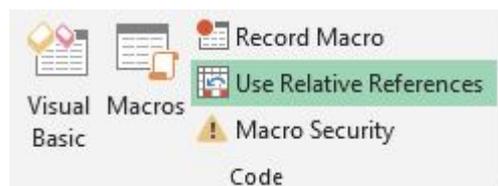
Ora avrai un nuovo modulo sotto la cartella Modules.

Il modulo più recente conterrà la macro che hai appena registrato. Fare doppio clic su di esso per richiamarlo.

Ho fatto un semplice copia e incolla:

```
Sub Macro1 ()  
'  
' Macro1 Macro  
'  
'  
    Selection.Copy  
    Range("A12").Select  
    ActiveSheet.Paste  
End Sub
```

Se non vuoi che venga sempre incollato in "A12", puoi utilizzare i Riferimenti relativi selezionando



la casella "Usa riferimenti relativi" nella scheda Sviluppatore:

Seguendo gli stessi passaggi di prima ora trasformeremo la Macro in questo:

```
Sub Macro2 ()  
'  
' Macro2 Macro  
'  
'  
    Selection.Copy  
    ActiveCell.Offset(11, 0).Range("A1").Select  
    ActiveSheet.Paste  
End Sub
```

Continuo a copiare il valore da "A1" in una cella di 11 righe, ma ora è possibile eseguire la stessa macro con qualsiasi cella iniziale e il valore da quella cella verrà copiato nella cella 11 righe in

basso.

Leggi Come registrare una macro online: <https://riptutorial.com/it/excel-vba/topic/8204/come-registrare-una-macro>

Capitolo 7: Creazione di un menu a discesa nel foglio di lavoro attivo con una casella combinata

introduzione

Questo è un semplice esempio che dimostra come creare un menu a discesa nel foglio attivo della cartella di lavoro inserendo un oggetto ActiveX della casella combinata nel foglio. Sarai in grado di inserire uno dei cinque brani di Jimi Hendrix in qualsiasi cella attivata del foglio ed essere in grado di cancellarlo, di conseguenza.

Examples

Menu di Jimi Hendrix

In generale, il codice viene inserito nel modulo di un foglio.

Questo è l'evento `Worksheet_SelectionChange`, che si attiva ogni volta che viene selezionata una cella diversa nel foglio attivo. Puoi selezionare "Foglio di lavoro" dal primo menu a discesa sopra la finestra del codice e "Selection_Change" dal menu a discesa accanto ad esso. In questo caso, ogni volta che si attiva una cella, il codice viene reindirizzato al codice della casella combinata.

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

    ComboBox1_Change

End Sub
```

Qui, la routine dedicata al `ComboBox` è codificata per l'evento `Change` per impostazione predefinita. In esso, c'è una matrice fissa, popolata con tutte le opzioni. Non l'opzione `CLEAR` nell'ultima posizione, che verrà utilizzata per cancellare il contenuto di una cella. L'array viene quindi passato al `Combo Box` e passato alla routine che esegue il lavoro.

```
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
```

L'array viene passato alla routine che riempie le celle con il nome del brano o il valore null per svuotarle. Innanzitutto, a una variabile intera viene assegnato il valore della posizione della scelta effettuata dall'utente. Quindi, la casella combinata viene spostata nell'angolo in alto a sinistra della cella che l'utente attiva e le sue dimensioni regolate per rendere l'esperienza più fluida. Alla cella attiva viene quindi assegnato il valore nella posizione della variabile intera, che tiene traccia della scelta dell'utente. Nel caso in cui l'utente selezioni CANCELLA dalle opzioni, la cella viene svuotata.

L'intera routine si ripete per ogni cella selezionata.

```
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
```

Esempio 2: Opzioni non incluse

Questo esempio viene utilizzato per specificare opzioni che potrebbero non essere incluse in un database di alloggi disponibili e relativi servizi.

Si basa sull'esempio precedente, con alcune differenze:

1. Due procedure non sono più necessarie per una singola casella combinata, ottenuta combinando il codice in un'unica procedura.
2. L'uso della proprietà `LinkedCell` per consentire l'inserimento corretto della selezione utente ogni volta
3. L'inclusione di una funzionalità di backup per garantire che la cella attiva si trovi nella colonna corretta e un codice di prevenzione degli errori, in base all'esperienza precedente, in cui i valori numerici sarebbero stati formattati come stringhe quando inseriti nella cella attiva.

```
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 precedente viene avviata ogni volta che viene attivata una cella con l'evento SelectionChange nel modulo del foglio di lavoro:

```
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
```

Leggi Creazione di un menu a discesa nel foglio di lavoro attivo con una casella combinata online:
<https://riptutorial.com/it/excel-vba/topic/8929/creazione-di-un-menu-a-discesa-nel-foglio-di-lavoro-attivo-con-una-casella-combinata>

Capitolo 8: CustomDocumentProperties in pratica

introduzione

L'uso di CustomDocumentProperties (CDP) è un buon metodo per archiviare i valori definiti dall'utente in modo relativamente sicuro all'interno della stessa cartella di lavoro, ma evitando di mostrare i valori delle celle correlate semplicemente in un foglio di lavoro non protetto *).

Nota: i CDP rappresentano una raccolta separata comparabile a BuiltInDocumentProperties, ma consentono di creare nomi di proprietà definiti dall'utente invece che una raccolta fissa.

*) In alternativa, puoi inserire valori anche in una cartella di lavoro nascosta o "molto nascosta".

Examples

Organizzazione di nuovi numeri di fattura

Aumentare un numero di fattura e salvarne il valore è un compito frequente. L'uso di CustomDocumentProperties (CDP) è un buon metodo per archiviare tali numeri in modo relativamente sicuro all'interno dello stesso libro di lavoro, ma evitando di mostrare i relativi valori di cella semplicemente in un foglio di lavoro non protetto.

Suggerimento aggiuntivo:

In alternativa, è possibile inserire valori anche in un foglio di lavoro nascosto o anche in un cosiddetto foglio di lavoro "molto nascosto" (vedere [Utilizzo di fogli xlVeryHidden](#) . Naturalmente, è possibile salvare i dati anche su file esterni (ad esempio file ini, csv o qualsiasi altro tipo) o il registro.

Contenuto di esempio :

L'esempio qui sotto mostra

- una funzione NextInvoiceNo che imposta e restituisce il prossimo numero di fattura,
- una procedura DeleteInvoiceNo, che cancella completamente il CDP delle fatture, così come
- una procedura showAllCDP che elenca la raccolta completa dei CDP con tutti i nomi. Non utilizzando VBA, è anche possibile elencarli tramite le informazioni della cartella di lavoro:
Informazioni | Proprietà [DropDown:] | Proprietà avanzate | costume

È possibile ottenere e impostare il numero di fattura successivo (l'ultimo no più uno) semplicemente chiamando la funzione sopra menzionata, restituendo un valore di stringa per facilitare l'aggiunta di prefissi. "InvoiceNo" è implicitamente utilizzato come nome CDP in tutte le procedure.

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

Codice di esempio:

```
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 CDPEExists("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

```

Leggi CustomDocumentProperties in pratica online: <https://riptutorial.com/it/excel-vba/topic/10932/customdocumentproperties-in-pratica>

Capitolo 9: Debug e risoluzione dei problemi

Sintassi

- Debug.Print (stringa)
- Basta basta

Examples

Debug.Print

Per stampare un elenco delle descrizioni dei codici di errore nella finestra immediata, passarlo alla funzione `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
```

Puoi mostrare la finestra immediata di:

- Selezionare **V** isualizza | **I** mmediate Window dalla barra dei menu
- Usando la scorciatoia da tastiera **Ctrl-G**

Stop

Il comando `Stop` interrompe l'esecuzione quando viene chiamato. Da lì, il processo può essere ripreso o essere eseguito passo dopo passo.

```
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
```

Finestra immediata

Se desideri testare una riga di codice macro senza dover eseguire un intero sottotitolo, puoi digitare i comandi direttamente nella finestra immediata e premere `ENTER` per eseguire la riga.

Per testare l'output di una linea, puoi precederla con un punto interrogativo `?` per stampare direttamente nella finestra immediata. In alternativa, è anche possibile utilizzare il comando di `print` per `print` l'output.

Mentre nel Visual Basic Editor, premere `CTRL + G` per aprire la Finestra Immediata. Per rinominare il foglio attualmente selezionato in "Foglio di esempio", digita quanto segue nella Finestra immediata e premi `ENTER`

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

Per stampare il nome del foglio attualmente selezionato direttamente nella Finestra immediata

```
? ActiveSheet.Name  
ExampleSheet
```

Questo metodo può essere molto utile per testare la funzionalità di funzioni incorporate o definite dall'utente prima di implementarle nel codice. L'esempio seguente mostra come la finestra immediata può essere utilizzata per testare l'output di una funzione o una serie di funzioni per confermare un risultato previsto.

```
'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 finestra immediata può anche essere utilizzata per impostare o ripristinare l'applicazione, la cartella di lavoro o altre proprietà necessarie. Questo può essere utile se `Application.EnableEvents = False` in una subroutine genera un errore inaspettatamente, causando la chiusura senza reimpostare il valore su `True` (che può causare funzionalità frustranti e impreviste. In questo caso, i comandi possono essere digitati direttamente nella finestra immediata ed eseguire:

```
? 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
```

Per tecniche di debug più avanzate, i due punti `:` possono essere utilizzati come separatori di riga. Questo può essere usato per espressioni multi-linea come il loop nell'esempio sotto.

```
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
```

Utilizzare il timer per individuare i colli di bottiglia nelle prestazioni

Il primo passo per ottimizzare la velocità è trovare le sezioni di codice più lente. La funzione VBA `Timer` restituisce il numero di secondi trascorsi da mezzanotte con una precisione di 1/25 di secondo (3.90625 millisecondi) su PC basati su Windows. Le funzioni VBA `Now` e `Time` sono accurate solo per un secondo.

```

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"

```

Aggiunta di un punto di interruzione al codice

Puoi aggiungere facilmente un punto di interruzione al tuo codice facendo clic sulla colonna grigia a sinistra della linea del tuo codice VBA in cui desideri interrompere l'esecuzione. Un punto rosso appare nella colonna e il codice di punto di interruzione è anche evidenziato in rosso.

Puoi aggiungere più punti di interruzione nel codice e riprendere l'esecuzione premendo l'icona "Riproduci" nella barra dei menu. Non tutto il codice può essere un punto di interruzione come linee di definizione delle variabili, la prima o l'ultima riga di una procedura e le righe di commento non possono essere selezionate come punto di interruzione.



Finestra dei debugger locali

La finestra Locals fornisce un facile accesso al valore corrente di variabili e oggetti nell'ambito della funzione o subroutine in esecuzione. È uno strumento essenziale per eseguire il debug del codice e modificare le modifiche per trovare i problemi. Permette anche di esplorare proprietà che potresti non sapere esistessero.

Prendi il seguente esempio,

```

Option Explicit
Sub LocalsWindowExample()

```

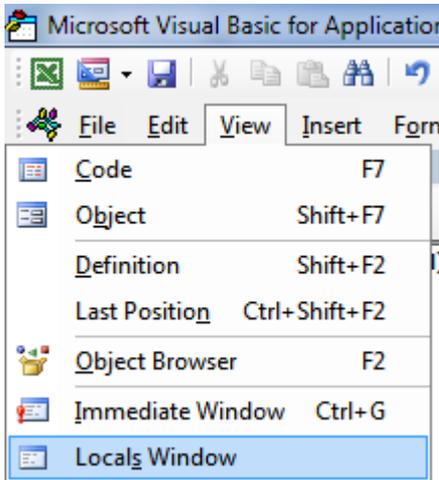
```

Dim findMeInLocals As Integer
Dim findMEInLocals2 As Range

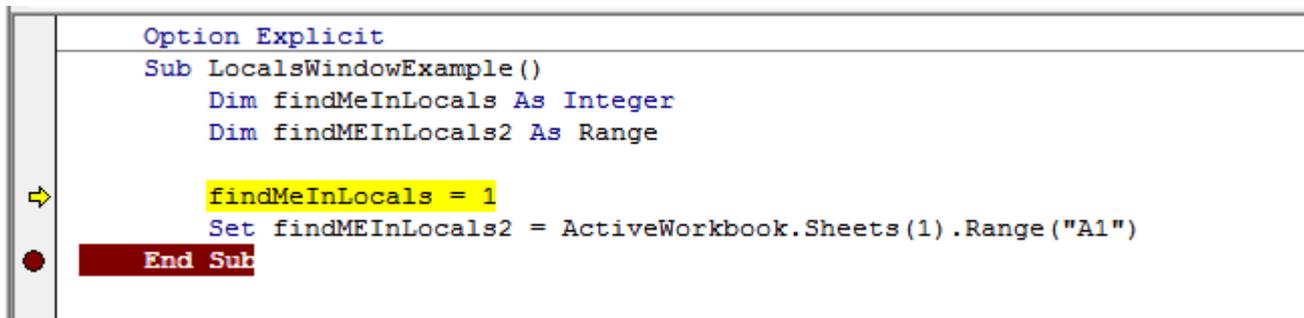
findMeInLocals = 1
Set findMEInLocals2 = ActiveWorkbook.Sheets(1).Range("A1")
End Sub

```

Nell'editor VBA, fare clic su Visualizza -> Finestra locale



Quindi, passando attraverso il codice usando F8 dopo aver fatto clic all'interno della subroutine, ci siamo fermati prima di arrivare all'assegnazione di findMeInLocals. Qui sotto puoi vedere che il valore è 0 --- e questo è quello che verrebbe usato se non gli avessi mai assegnato un valore. L'oggetto range è 'Nothing'.



Locals		
VBAProject.Sheet1.LocalsWindowExample		
Expression	Value	Type
Me		Sheet
findMeInLocals	0	Integer
findMEInLocals2	Nothing	Range

Se ci fermiamo appena prima della fine della subroutine, possiamo vedere i valori finali delle variabili.

```

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

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

```

Possiamo vedere findMeInLocals con un valore di 1 e tipo di numero intero e FindMeInLocals2 con un tipo di intervallo / intervallo. Se clicchiamo sul segno +, possiamo espandere l'oggetto e vedere le sue proprietà, come il conteggio o la colonna.

Locals		
VBAProject.Sheet1.LocalsWindowExample		
Expression	Value	Type
Me		Sheet
findMeInLocals	1	Integer
findMEInLocals2		Range
AddIndent	False	Variation
AllowEdit	True	Boolean
Application		Application
Areas		Area
Borders		Border
Cells		Range
Column	1	Long
ColumnWidth	8.43	Variation
Comment	Nothing	Comment
Count	1	Long
CountLarge	1	Variation
Creator	xlCreatorCode	XICreatorCode
CurrentArray	<No cells were found.>	Range
CurrentRegion		Range
Dependents	<No cells were found.>	Range
DirectDependents	<No cells were found.>	Range
DirectPrecedents	<No cells were found.>	Range
DisplayFormat		DisplayFormat

Leggi Debug e risoluzione dei problemi online: <https://riptutorial.com/it/excel-vba/topic/861/debug-e-risoluzione-dei-problemi>

Capitolo 10: Dichiarazioni condizionali

Examples

La dichiarazione di If

L'istruzione di controllo `If` consente di eseguire codice diverso a seconda della valutazione di un'istruzione condizionale (booleana). Un'istruzione condizionale è una che valuta `True` o `False`, ad esempio `x > 2`.

Esistono tre modelli che possono essere utilizzati quando si implementa un'istruzione `If`, che sono descritti di seguito. Si noti che una valutazione condizionale `If` è sempre seguita da un `Then`.

1. Valutare uno `If` un'istruzione condizionale e fare qualcosa se è `True`

Riga singola `If` affermazione

Questo è il modo più breve per usare un `If` ed è utile quando una sola affermazione deve essere eseguita su una `True` valutazione. Quando si utilizza questa sintassi, tutto il codice deve essere su una singola riga. Non includere una `End If` alla fine della linea.

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

`If` blocco

Se è necessario eseguire più righe di codice su una valutazione `True`, è possibile utilizzare un blocco `If`.

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

Notare che, se si utilizza un blocco `If` più righe, è necessario un `End If` corrispondente.

2. Valutare una istruzione `If` condizionale, facendo una cosa se è `True` e facendo qualcos'altro se è `False`

Riga singola `If`, istruzione `Else`

Questo può essere usato se una dichiarazione deve essere eseguita su una `True` valutazione e una dichiarazione diversa deve essere eseguita su una valutazione `False`. State attenti a usare questa sintassi, poiché è spesso meno chiaro ai lettori che esiste un'istruzione `Else`. Quando si utilizza questa sintassi, tutto il codice deve essere su una singola riga. Non includere una `End If` alla fine della linea.

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

If , Else blocco

Utilizzare un blocco `If , Else` per aggiungere chiarezza al codice o se è necessario eseguire più righe di codice in una valutazione `True` o `False` .

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

Notare che, se si utilizza un blocco `If` più righe, è necessario un `End If` corrispondente.

3. Valutare molte affermazioni condizionali, quando le dichiarazioni precedenti sono tutte `False` e fare qualcosa di diverso per ciascuna di esse

Questo modello è l'uso più generale di `If` e dovrebbe essere usato quando ci sono molte condizioni non sovrapposte che richiedono un trattamento diverso. A differenza dei primi due modelli, questo caso richiede l'uso di un blocco `If` , anche se verrà eseguita solo una riga di codice per ogni condizione.

If , ElseIf , ... , Else blocco

Invece di dover creare molti blocchi `If` uno sotto l'altro, è possibile utilizzare un `ElseIf` valutare una condizione aggiuntiva. L' `ElseIf` viene valutato solo se precedente `If` valutazione è `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
```

Poiché molte istruzioni di controllo `ElseIf` possono essere incluse tra un `If` e un `End If` come richiesto. Non è richiesta un'istruzione di controllo `Else` quando si utilizza `ElseIf` (sebbene sia consigliabile), ma se è inclusa, deve essere l'istruzione di controllo finale prima di `End If` .

Leggi Dichiarazioni condizionali online: <https://riptutorial.com/it/excel-vba/topic/9632/dichiarazioni-condizionali>

Capitolo 11: Errori comuni

Examples

Riferimenti qualificanti

Quando ci si riferisce a un `worksheet`, un `range` o singole `cells`, è importante qualificare completamente il riferimento.

Per esempio:

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

Non è completamente qualificato: i riferimenti di `Cells` non hanno una cartella di lavoro e un foglio di lavoro associati. Senza un riferimento esplicito, `Cells` fa riferimento a `ActiveSheet` per impostazione predefinita. Quindi questo codice fallirà (produrrà risultati errati) se un foglio di lavoro diverso da `Sheet1` è il `ActiveSheet` corrente.

Il modo più semplice per correggere ciò è utilizzare un'istruzione `With` come segue:

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

In alternativa, è possibile utilizzare una variabile del foglio di lavoro. (Questo sarà probabilmente il metodo preferito se il tuo codice deve fare riferimento a più fogli di lavoro, come copiare i dati da un foglio a un altro.)

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

Un altro problema frequente è fare riferimento alla raccolta dei fogli di lavoro senza qualificare la cartella di lavoro. Per esempio:

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

Il foglio di lavoro `Sheet1` non è completo e manca di una cartella di lavoro. Questo potrebbe fallire se nel codice vengono referenziate più cartelle di lavoro. Invece, utilizzare uno dei seguenti:

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

Tuttavia, evitare di utilizzare quanto segue:

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

Allo stesso modo per gli oggetti `range`, se non esplicitamente qualificato, l' `range` si riferirà al foglio attualmente attivo:

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

Equivale a:

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

Eliminazione di righe o colonne in un ciclo

Se vuoi eliminare righe (o colonne) in un ciclo, dovresti sempre eseguire il ciclo partendo dalla fine dell'intervallo e tornare indietro in ogni passaggio. In caso di utilizzo del codice:

```
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
```

Perderai alcune righe. Ad esempio, se il codice cancella la riga 3, la riga 4 diventa la riga 3. Tuttavia, la variabile `i` cambierà in 4. Quindi, in questo caso il codice mancherà di una riga e ne controllerà un'altra, che non era nell'intervallo in precedenza.

Il codice giusto sarebbe

```
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` e `ThisWorkbook` volte vengono utilizzati in modo intercambiabile dai nuovi utenti di VBA senza comprendere appieno a che cosa si riferisce ogni oggetto, questo può causare un comportamento indesiderato in fase di esecuzione. Entrambi questi oggetti appartengono [all'oggetto dell'applicazione](#)

L'oggetto `ActiveWorkbook` fa riferimento alla cartella di lavoro attualmente nella vista più in alto dell'oggetto dell'applicazione Excel al momento dell'esecuzione. (es. *La cartella di lavoro che puoi vedere e interagire con il punto in cui viene fatto riferimento a questo oggetto*)

```

Sub ActiveWorkbookExample()

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

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

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

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

End Sub

```

L'oggetto `ThisWorkbook` fa riferimento alla cartella di lavoro a cui appartiene il codice nel momento in cui viene eseguita.

```

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"

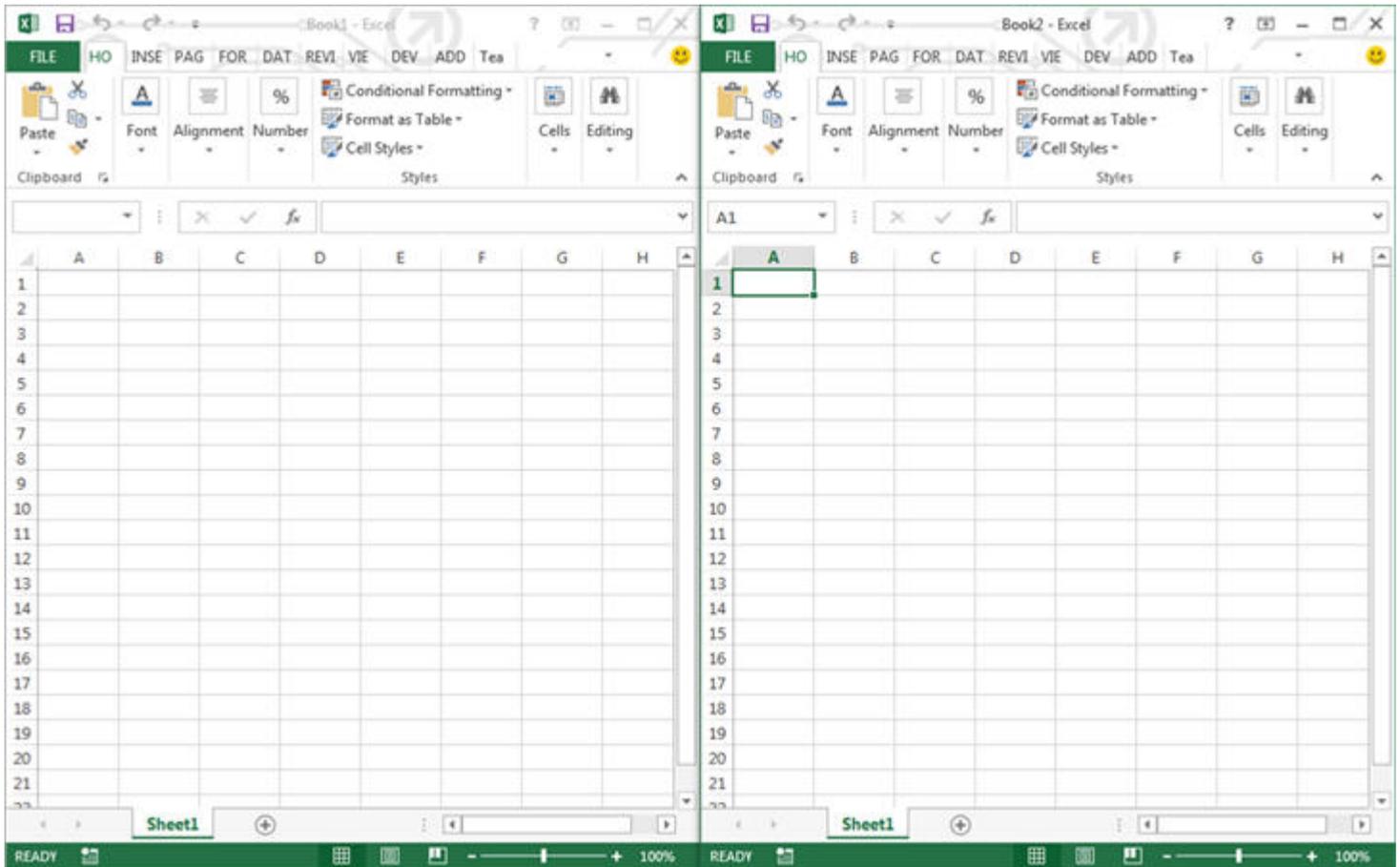
End Sub

```

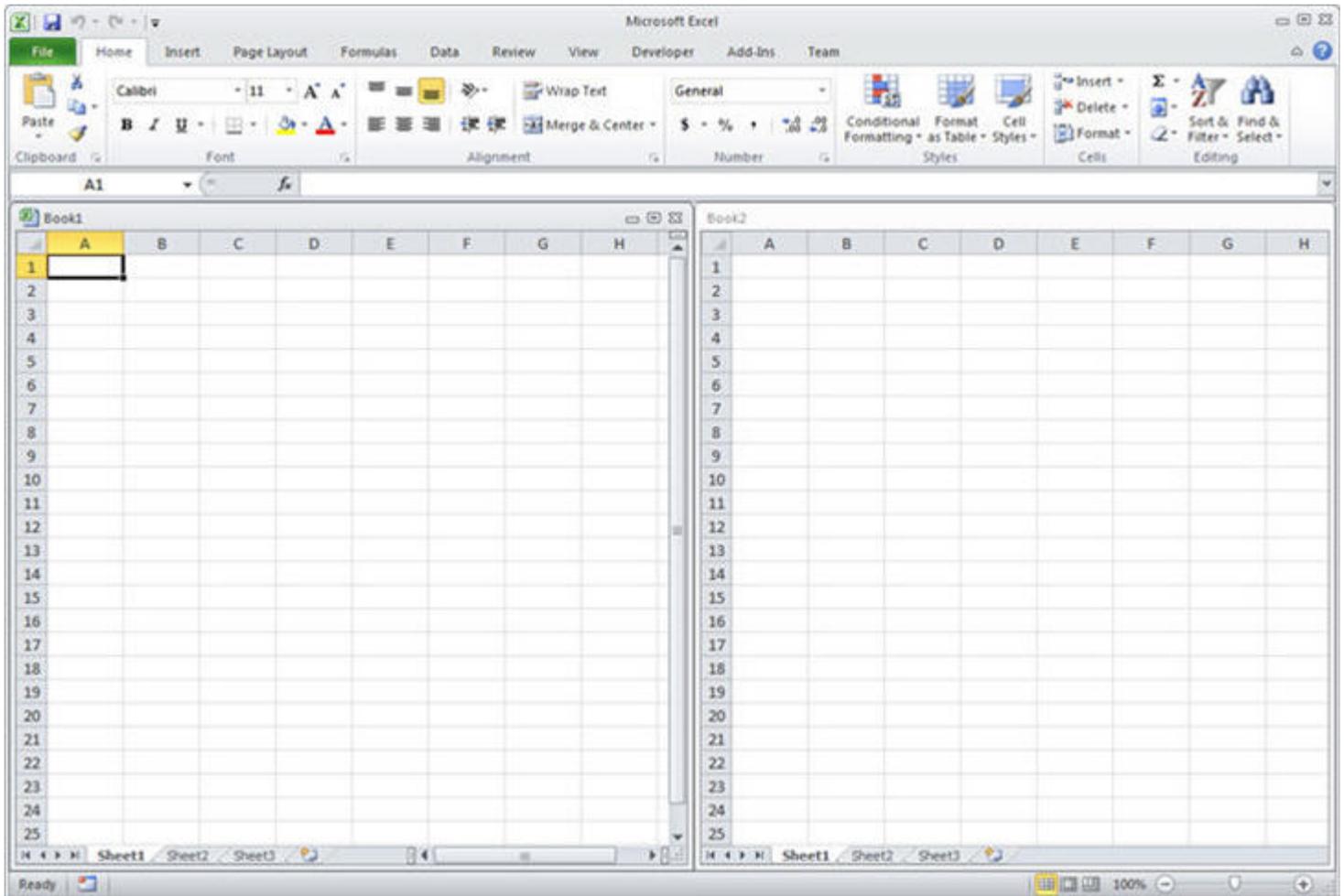
Interfaccia a singolo documento e interfacce a più documenti

Si noti che Microsoft Excel 2013 (e versioni successive) utilizza Single Document Interface (SDI) e che Excel 2010 (e successivi) utilizza Multiple Document Interfaces (MDI).

Ciò implica che per Excel 2013 (SDI), ogni cartella di lavoro in una singola istanza di Excel contenga la **propria** interfaccia utente della barra multifunzione:



Al contrario, per Excel 2010, ciascuna cartella di lavoro in una singola istanza di Excel utilizzava un'interfaccia utente nastro **comune** (MDI):



Ciò solleva alcuni problemi importanti se si desidera eseguire la migrazione di un codice VBA (2010 <-> 2013) che interagisce con la barra multifunzione.

È necessario creare una procedura per aggiornare i controlli dell'interfaccia utente della barra multifunzione nello stesso stato in tutte le cartelle di lavoro per Excel 2013 e versioni successive.

Nota che :

1. Tutti i metodi, gli eventi e le proprietà delle finestre a livello di applicazione di Excel rimangono inalterati. (`Application.ActiveWindow` , `Application.Windows` ...)
2. In Excel 2013 e versioni successive (SDI) tutti i metodi, gli eventi e le proprietà delle finestre a livello di cartella di lavoro ora operano nella finestra di livello superiore. È possibile recuperare l'handle di questa finestra di livello superiore con `Application.Hwnd`

Per ottenere maggiori dettagli, vedere la fonte di questo esempio: [MSDN](#) .

Ciò causa anche alcuni problemi con userforms non modali. Vedi [qui](#) per una soluzione.

Leggi Errori comuni online: <https://riptutorial.com/it/excel-vba/topic/1576/errori-comuni>

Capitolo 12: File System Object

Examples

File, cartella, unità esistente

Il file esiste:

```
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
```

Cartella esiste:

```
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
```

L'unità esiste:

```
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
```

Operazioni di file di base

Copia:

```
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
```

Mossa:

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

Elimina:

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

Operazioni di cartella di base

Creare:

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

Copia:

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

Mossa:

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

Elimina:

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

Altre operazioni

Ottieni il nome del file:

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

Risultato: Makro.txt

Ottieni il nome base:

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

Risultato: Makro

Ottieni il nome dell'estensione:

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

```
End Sub
```

Risultato: txt

Ottieni il nome del drive:

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

Risultato: c:

Leggi File System Object online: <https://riptutorial.com/it/excel-vba/topic/9933/file-system-object>

Capitolo 13: Formattazione condizionale tramite VBA

Osservazioni

Non è possibile definire più di tre formati condizionali per un intervallo. Utilizzare il metodo Modifica per modificare un formato condizionale esistente o utilizzare il metodo Elimina per eliminare un formato esistente prima di aggiungerne uno nuovo.

Examples

FormatConditions.Add

Sintassi:

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

parametri:

Nome	Richiesto / Opzionale	Tipo di dati
genere	necessario	XIFormatConditionType
Operatore	Opzionale	Variante
Formula 1	Opzionale	Variante
formula2	Opzionale	Variante

XIFormatConditionType enumeration:

Nome	Descrizione
xlAboveAverageCondition	Condizioni superiori alla media
xlBlanksCondition	Condizione di spazi
xlCellValue	Valore della cella
xlColorScale	Scala di colori

Nome	Descrizione
xlDatabar	Databar
xlErrorsCondition	Condizione di errore
xlExpression	Espressione
XlIconSet	Set di icone
xlNoBlanksCondition	Nessuna condizione di bianco
xlNoErrorsCondition	Nessuna condizione di errore
xlTextString	Stringa di testo
xlTimePeriod	Periodo di tempo
xlTop10	I 10 valori principali
xlUniqueValues	Valori unici

Formattazione per valore della cella:

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

operatori:

Nome
xlBetween
xlEqual
xlGreater
xlGreaterEqual
xlLess
xlLessEqual
xlNotBetween

Nome

xlNotEqual

Se Type è xlExpression, l'argomento Operator viene ignorato.

La formattazione per testo contiene:

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

operatori:

Nome	Descrizione
xlBeginsWith	Inizia con un valore specificato.
xlContains	Contiene un valore specificato.
xlDoesNotContain	Non contiene il valore specificato.
xlEndsWith	Endswith il valore specificato

Formattazione per periodo

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

operatori:

Nome

xlYesterday

xlTomorrow

xlLast7Days

Nome
xlLastWeek
xlThisWeek
xlNextWeek
xlLastMonth
xlThisMonth
xlNextMonth

Rimuovi il formato condizionale

Rimuovi tutti i formati condizionali nell'intervallo:

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

Rimuovi tutto il formato condizionale nel foglio di lavoro:

```
Cells.FormatConditions.Delete
```

FormatConditions.AddUniqueValues

Evidenziare i valori duplicati

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

Evidenziando i valori unici

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

FormatConditions.AddTop10

Evidenziando i 5 valori principali

```
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
```

operatori:

Nome	Descrizione
XIAboveAverage	Sopra la media
XIAboveStdDev	Sopra la deviazione standard
XIBelowAverage	Sotto la media
XIBelowStdDev	Sotto la deviazione standard
XIEqualAboveAverage	Uguale sopra la media
XIEqualBelowAverage	Uguale sotto 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:

Nome
xl3Arrows
xl3ArrowsGray
xl3Flags
xl3Signs
xl3Stars
xl3Symbols
xl3Symbols2
xl3TrafficLights1

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

Directional

Shapes

Indicators

Ratings

[More Rules...](#)

Genere:

Nome
xlConditionValuePercent
xlConditionValueNumber
xlConditionValuePercentile
xlConditionValueFormula

Operatore:

Nome	Valore
xlGreater	5
xlGreaterEqual	7

Valore:

Restituisce o imposta il valore di soglia per un'icona in un formato condizionale.

Leggi [Formattazione condizionale tramite VBA online](https://riptutorial.com/it/excel-vba/topic/9912/formattazione-condizionale-tramite-vba): <https://riptutorial.com/it/excel-vba/topic/9912/formattazione-condizionale-tramite-vba>

Capitolo 14: Funzioni definite dall'utente (UDF)

Sintassi

- 1. Funzione `functionName (argumentVariable As dataType, argumentVariable2 As dataType, facoltativo argumentVariable3 As dataType) As functionReturnDataType`**
Dichiarazione di base di una funzione. Ogni funzione ha bisogno di un nome, ma non deve prendere alcun argomento. Potrebbero essere necessari 0 argomenti o potrebbe essere necessario un determinato numero di argomenti. Puoi anche dichiarare un argomento come facoltativo (cioè non importa se lo fornisci quando chiami la funzione). È consigliabile fornire il tipo di variabile per ogni argomento e, analogamente, restituire il tipo di dati restituito dalla funzione stessa.
- 2. `functionName = theVariableOrValueBeingReturned`**
Se provieni da altri linguaggi di programmazione, potresti essere abituato alla parola chiave `Return`. Questo non è usato in VBA - invece, usiamo il nome della funzione. È possibile impostarlo sul contenuto di una variabile o su un valore fornito direttamente. Nota che se hai impostato un tipo di dati per il ritorno della funzione, la variabile o i dati che stai fornendo questa volta devono essere di quel tipo di dati.
- 3. Fine Funzione**
Obbligatorio. Significa la fine del codice di blocco della `Function` e deve quindi essere alla fine. Generalmente il VBE lo fornisce automaticamente quando si crea una nuova funzione.

Osservazioni

Una funzione definita dall'utente (nota come UDF) si riferisce a una funzione specifica dell'attività creata dall'utente. Può essere chiamato come una funzione del foglio di lavoro (es: `=SUM(...)`) o utilizzato per restituire un valore a un processo in esecuzione in una procedura Sub. Un UDF restituisce un valore, in genere dalle informazioni passate in esso come uno o più parametri.

Può essere creato da:

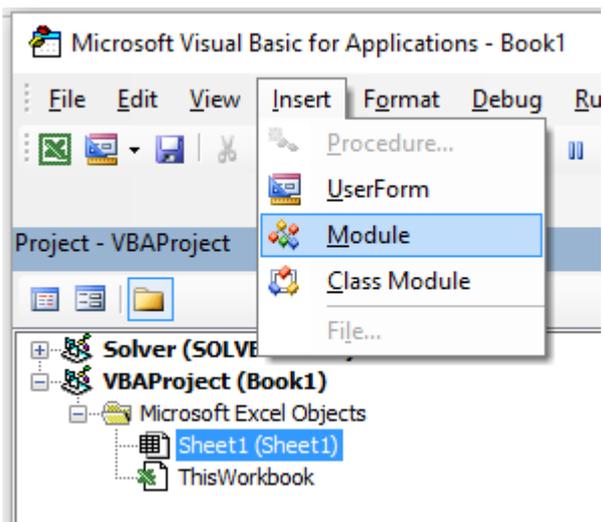
1. usando VBA.
2. utilizzando l'API C di Excel: creando un XLL che esporta le funzioni compilate in Excel.
3. usando l'interfaccia COM.

Examples

UDF - Hello World

1. Apri Excel
2. Aprire l'editor di Visual Basic (vedere [Apertura dell'editor di Visual Basic](#))

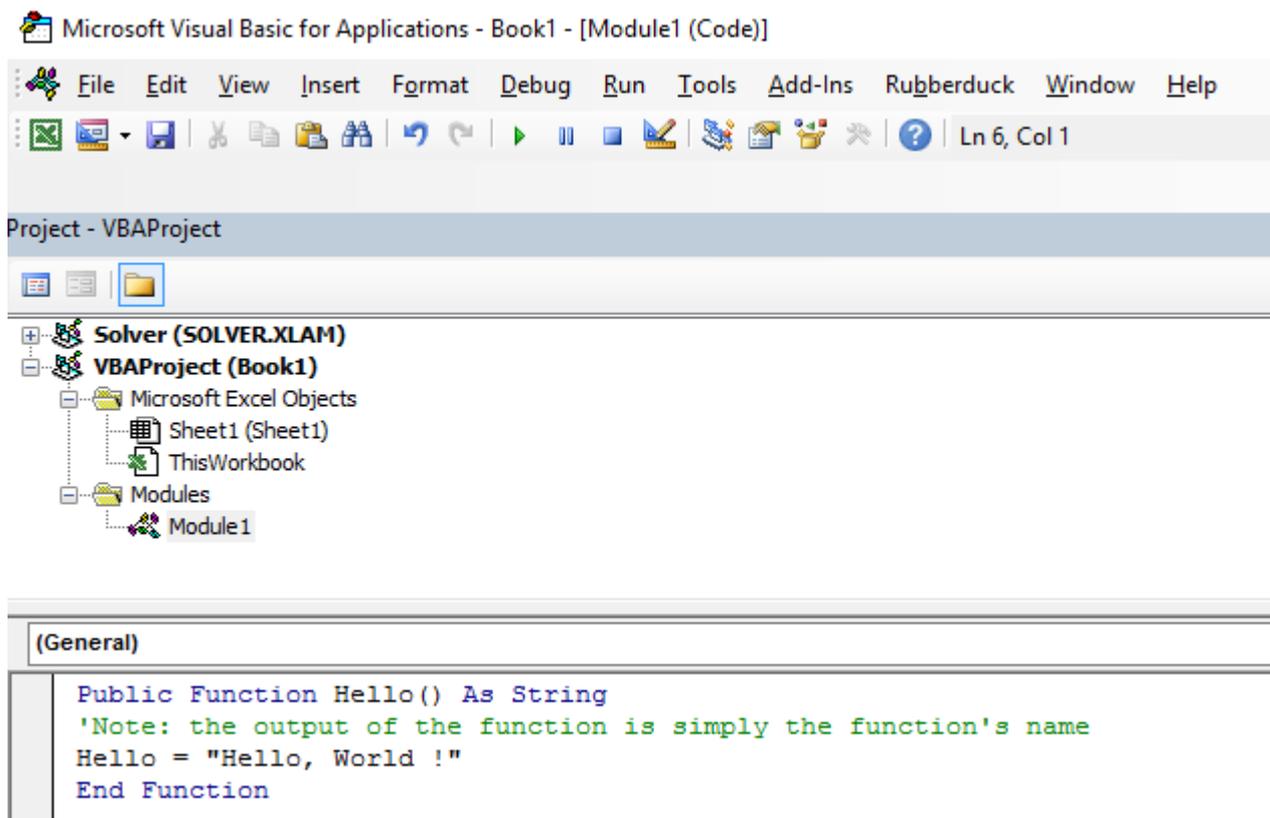
3. Aggiungi un nuovo modulo facendo clic su Inserisci -> Modulo:



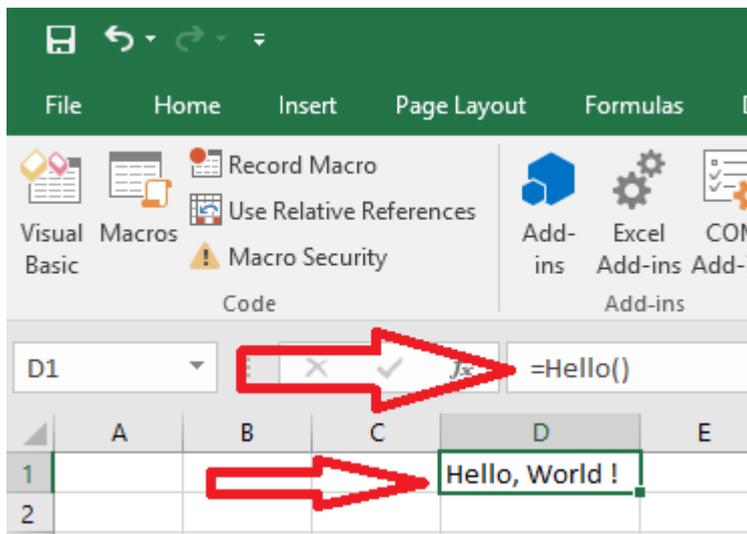
4. Copia e incolla il codice seguente nel nuovo modulo:

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

Ottenere :



5. Torna alla cartella di lavoro e digita "= Hello ()" in una cella per visualizzare "Hello World".



Consenti riferimenti a colonne complete senza penalità

È più semplice implementare alcune UDF nel foglio di lavoro se i riferimenti a colonne complete possono essere passati come parametri. Tuttavia, a causa della natura esplicita della codifica, qualsiasi loop che coinvolge questi intervalli può elaborare centinaia di migliaia di celle completamente vuote. Questo riduce il tuo progetto VBA (e la cartella di lavoro) a un pasticcio congelato mentre i non-valori non necessari vengono elaborati.

Il looping delle celle di un foglio di lavoro è uno dei metodi più lenti per eseguire un'attività, ma a volte è inevitabile. Tagliare il lavoro eseguito fino a ciò che è effettivamente necessario ha perfettamente senso.

La soluzione è di troncare la colonna completa o i riferimenti di riga completa alla [proprietà Worksheet.UsedRange](#) con il [metodo Intersect](#). Nell'esempio riportato di seguito sarà liberamente replicare la funzione SUMIF nativa di un foglio di lavoro in modo che il *criteri_intervallo* anche essere ridimensionato per adattarsi alla *int_somma* poiché ogni valore *nell'int_somma* deve essere accompagnato da un valore nel *criteri_intervallo*.

[Application.Caller](#) per una UDF utilizzata su un foglio di lavoro è la cella in cui risiede. La proprietà [.Parent](#) della cella è il foglio di lavoro. Questo sarà usato per definire [.UsedRange](#).

In un foglio di codice del modulo:

```
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

```

Sintassi:

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

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

Mentre questo è un esempio abbastanza semplicistico, dimostra adeguatamente il passaggio di due riferimenti a colonne complete (1.048.576 righe ciascuno) ma l'elaborazione solo di 15 righe di dati e criteri.

Documentazione ufficiale MSDN collegata dei singoli metodi e proprietà forniti da Microsoft™.

Contare i valori unici nell'intervallo

```

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

```

[collezioni](#)

Leggi Funzioni definite dall'utente (UDF) online: <https://riptutorial.com/it/excel-vba/topic/1070/funzioni-definite-dall-utente--udf->

Capitolo 15: Gamme e celle

Sintassi

- **Set** - L'operatore utilizzato per impostare un riferimento a un oggetto, ad esempio un intervallo
- **Per ogni** : l'operatore ha utilizzato per scorrere tutti gli elementi di una raccolta

Osservazioni

Nota che i nomi delle variabili `r`, `cell` e altri possono essere nominati come vuoi, ma dovrebbero essere nominati in modo appropriato in modo che il codice sia più facile da capire per te e per gli altri.

Examples

Creazione di un intervallo

Un [Range](#) non può essere creato o popolato nello stesso modo in cui una stringa:

```
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
```

È considerata la migliore pratica per [qualificare i tuoi riferimenti](#), quindi d'ora in poi utilizzeremo lo stesso approccio qui.

Ulteriori informazioni sulla [creazione di variabili oggetto \(ad es. Intervallo\)](#) su [MSDN](#). Ulteriori informazioni su [Imposta dichiarazione su MSDN](#).

Esistono diversi modi per creare la stessa gamma:

```
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

```

Nota nell'esempio che Cells (2, 1) è equivalente a Range ("A2"). Questo perché le celle restituiscono un oggetto Range.

Alcune fonti: [Chip Pearson-Cells Within Ranges](#) ; [Oggetto MSDN-Range](#) ; [John Walkenback-Referring To Ranges Nel tuo codice VBA](#) .

Si noti inoltre che in qualsiasi istanza in cui un numero viene utilizzato nella dichiarazione dell'intervallo e che il numero stesso non rientra tra virgolette, come Intervallo ("A" e 2), è possibile scambiare quel numero per una variabile che contiene un integer / lungo. Per esempio:

```

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

```

Se si utilizzano i cicli doppi, le celle sono migliori:

```

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
```

Modi per fare riferimento a una singola cella

Il modo più semplice per riferirsi a una singola cella nel foglio di lavoro Excel corrente è semplicemente racchiudere la forma A1 del suo riferimento tra parentesi quadre:

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

Nota che le parentesi quadre sono solo convenienti [zucchero sintattico](#) per il metodo di `Evaluate` dell'oggetto `Application`, quindi tecnicamente, questo è identico al seguente codice:

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

Puoi anche chiamare il metodo `Cells` che prende una riga e una colonna e restituisce un riferimento di cella.

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

Ricorda che ogni volta che passi una riga e una colonna in Excel da VBA, la riga è sempre la prima, seguita dalla colonna, che è confusa perché è l'opposto della notazione A1 comune in cui la colonna viene visualizzata per prima.

In entrambi questi esempi, non abbiamo specificato un foglio di lavoro, quindi Excel utilizzerà il foglio attivo (il foglio che si trova di fronte nell'interfaccia utente). È possibile specificare il foglio attivo esplicitamente:

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

Oppure puoi fornire il nome di un particolare foglio:

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

Esiste un'ampia varietà di metodi che possono essere utilizzati per passare da un intervallo a un altro. Ad esempio, il metodo `Rows` può essere utilizzato per ottenere le singole righe di qualsiasi intervallo e il metodo `Cells` può essere utilizzato per ottenere singole celle di una riga o di una colonna, quindi il seguente codice fa riferimento alla cella C1:

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

Salvataggio di un riferimento a una cella in una variabile

Per salvare un riferimento a una cella in una variabile, è necessario utilizzare la sintassi `Set`, ad

esempio:

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

dopo...

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

Perché è necessaria la parola chiave `Set` ? `Set` dice a Visual Basic che il valore sul lato destro di `=` è pensato per essere un oggetto.

Proprietà offset

- **Offset (Righe, Colonne)** - L'operatore ha utilizzato il riferimento statico a un altro punto della cella corrente. Spesso utilizzato nei loop. Dovrebbe essere chiaro che i numeri positivi nella sezione delle righe si spostano a destra, mentre i negativi si spostano a sinistra. Con la sezione delle colonne, i positivi si spostano verso il basso e i negativi si spostano verso l'alto.

vale a dire

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

Questo codice seleziona B2, inserisce una nuova stringa lì, quindi sposta quella stringa su A1 dopo aver eliminato B2.

Come trasporre gli intervalli (da orizzontale a verticale e 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` ha anche un'opzione `Trasporta Incolla` che aggiorna anche le formule delle celle trasposte.

Leggi [Gamme e celle online](https://riptutorial.com/it/excel-vba/topic/1503/gamme-e-celle): <https://riptutorial.com/it/excel-vba/topic/1503/gamme-e-celle>

Capitolo 16: Gamme nominate

introduzione

L'argomento dovrebbe includere informazioni specificamente correlate agli intervalli denominati in Excel, compresi i metodi per la creazione, la modifica, l'eliminazione e l'accesso agli intervalli denominati definiti.

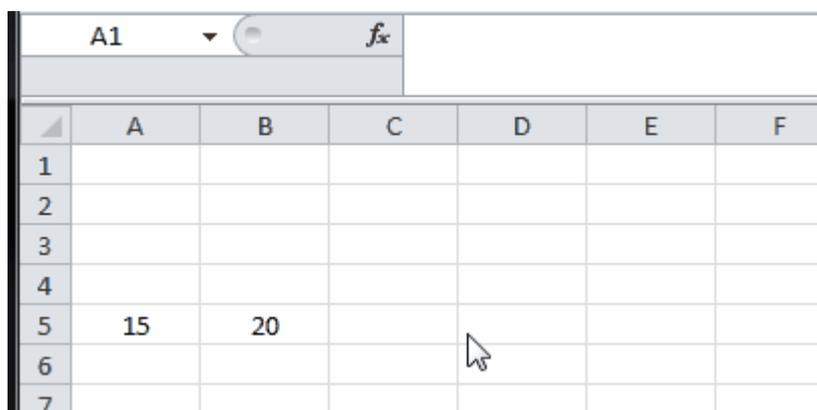
Examples

Definisci un intervallo con nome

L'utilizzo degli intervalli denominati consente di descrivere il significato di un contenuto di celle e di utilizzare questo nome definito al posto di un effettivo indirizzo di cella.

Ad esempio, formula `=A5*B5` può essere sostituita con `=Width*Height` per rendere la formula più facile da leggere e capire.

Per definire un nuovo intervallo denominato, selezionare la cella o le celle da assegnare al nome e quindi digitare il nuovo nome nella casella Nome accanto alla barra della formula.



	A	B	C	D	E	F
1						
2						
3						
4						
5	15	20				
6						
7						

Nota: gli intervalli denominati sono predefiniti come ambito globale, il che significa che è possibile accedervi da qualsiasi posizione all'interno della cartella di lavoro. Le versioni precedenti di Excel consentono nomi duplicati, quindi è necessario prestare attenzione per evitare nomi duplicati di ambito globale, altrimenti i risultati saranno imprevedibili. Utilizzare la Gestione nomi dalla scheda Formule per modificare l'ambito.

Utilizzo di intervalli denominati in VBA

Crea un nuovo intervallo denominato "MyRange" assegnato alla cella A1

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

Elimina l'intervallo con nome definito per nome

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

Accedi all'intervallo con nome per nome

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

Accedi a un intervallo con nome con un collegamento

[Proprio come qualsiasi altro intervallo](#), è possibile accedere direttamente agli intervalli denominati tramite una notazione di collegamento che non richiede la creazione di un oggetto `Range`. Le tre righe del codice di cui sopra possono essere sostituite da una singola riga:

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

Nota: la proprietà predefinita per un intervallo è il valore, quindi `[MyRange]` è uguale a `[MyRange].Value`

Puoi anche chiamare i metodi nell'intervallo. Quanto segue seleziona `MyRange`:

```
[MyRange].Select
```

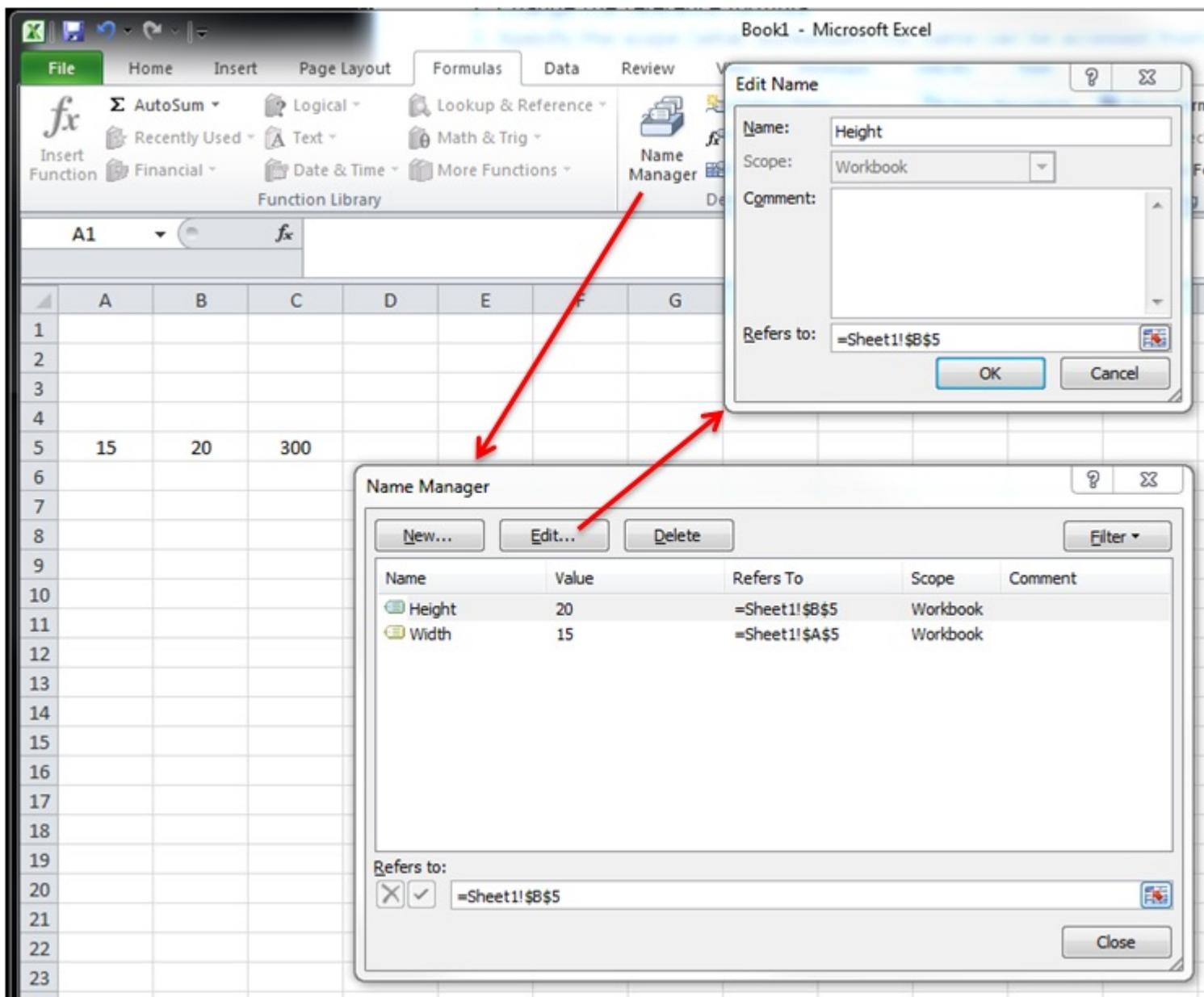
Nota: un avvertimento è che la notazione di collegamento non funziona con le parole che vengono utilizzate altrove nella libreria VBA. Ad esempio, un intervallo denominato `Width` non sarebbe accessibile come `[Width]` ma funzionerebbe come previsto se si accede tramite `ThisWorkbook.Worksheets("Sheet1").Range("Width")`

Gestisci intervalli denominati utilizzando Name Manager

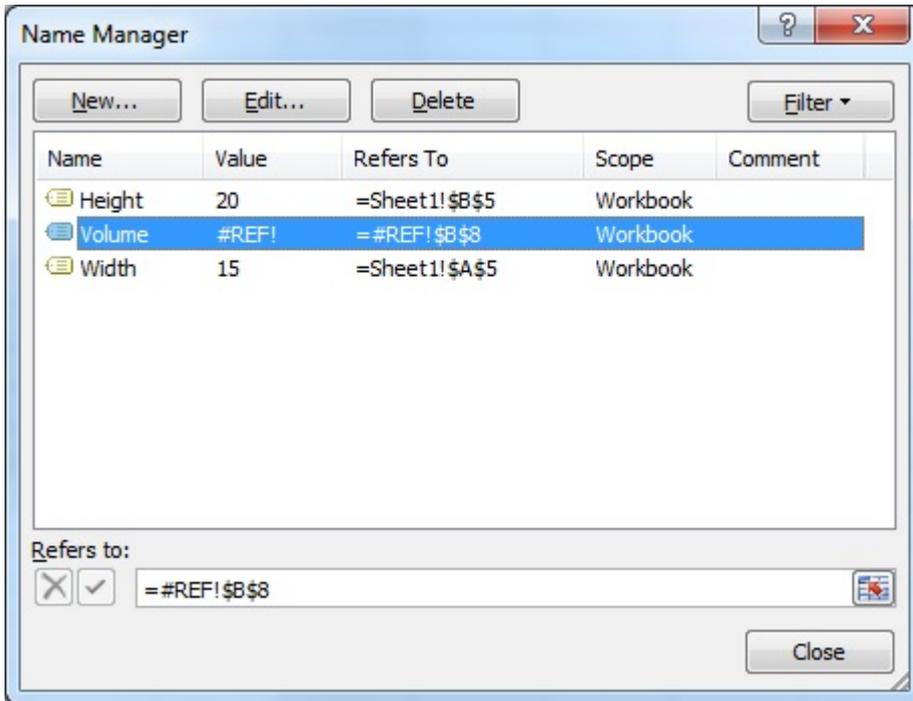
Scheda Formule > gruppo Nomi definiti > pulsante Gestione nomi

Named Manager ti consente di:

1. Crea o cambia nome
2. Crea o cambia riferimento di cella
3. Crea o cambia ambito
4. Elimina l'intervallo con nome esistente



Named Manager fornisce un'utile ricerca rapida di link non funzionanti.



Array gamma nominati

Foglio di esempio

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

Name	Value	Refers To	Scope	Comment
Units	{"50";"52..."}	=Sheet1!\$B\$5:\$B\$16	Workbook	
Year_Max		=Sheet1!\$E\$7	Workbook	
Year_Min		=Sheet1!\$E\$8	Workbook	

Refers to: =Sheet1!\$B\$5:\$B\$16

Codice

```
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

```

Risultato

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			

Leggi Gamme nominate online: <https://riptutorial.com/it/excel-vba/topic/8360/gamme-nominate>

Capitolo 17: Grafici e grafici

Examples

Creazione di un grafico con intervalli e un nome fisso

I grafici possono essere creati lavorando direttamente con l'oggetto `Series` che definisce i dati del grafico. Per arrivare alla `Series` senza un grafico esistente, si crea un oggetto `ChartObject` su un determinato `Worksheet` e si ottiene l'oggetto `Chart` da esso. Il vantaggio di lavorare con l'oggetto `Series` è che puoi impostare `Values` e `XValues` facendo riferimento agli oggetti `Range`. Queste proprietà dei dati definiranno correttamente la `Series` con riferimenti a tali intervalli. Lo svantaggio di questo approccio è che la stessa conversione non viene gestita durante l'impostazione del `Name`; è un valore fisso. Non si regola con i dati sottostanti l'originale `Range`. Controllo della formula `SERIES` ed è ovvio che il nome è fisso. Questo deve essere gestito creando direttamente la formula `SERIES`.

Codice utilizzato per creare un grafico

Si noti che questo codice contiene dichiarazioni di variabili aggiuntive per il `Chart` e il `Worksheet`. Questi possono essere omessi se non vengono utilizzati. Possono essere utili tuttavia se si modifica lo stile o altre proprietà del grafico.

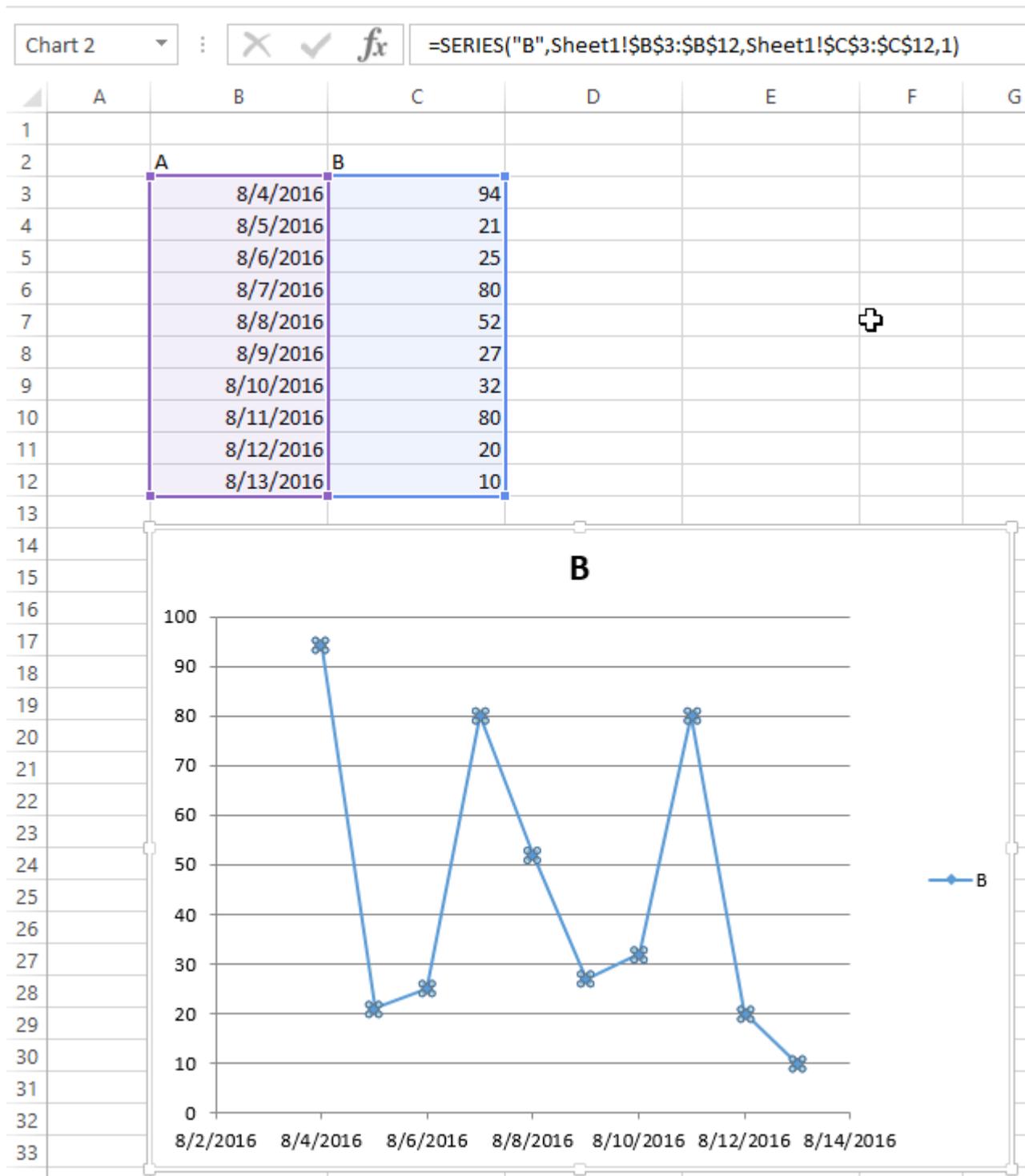
```
Sub CreateChartWithRangesAndFixedName()  
  
    Dim xData As Range  
    Dim yData As Range  
    Dim serName As Range  
  
    'set the ranges to get the data and y value label  
    Set xData = Range("B3:B12")  
    Set yData = Range("C3:C12")  
    Set serName = Range("C2")  
  
    'get reference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

Dati / intervalli originali e Chart risultante dopo l'esecuzione del codice

Si noti che la SERIES formula include una "B" per il nome della serie, invece di un riferimento alla Range che lo ha creato.



Creare un grafico vuoto

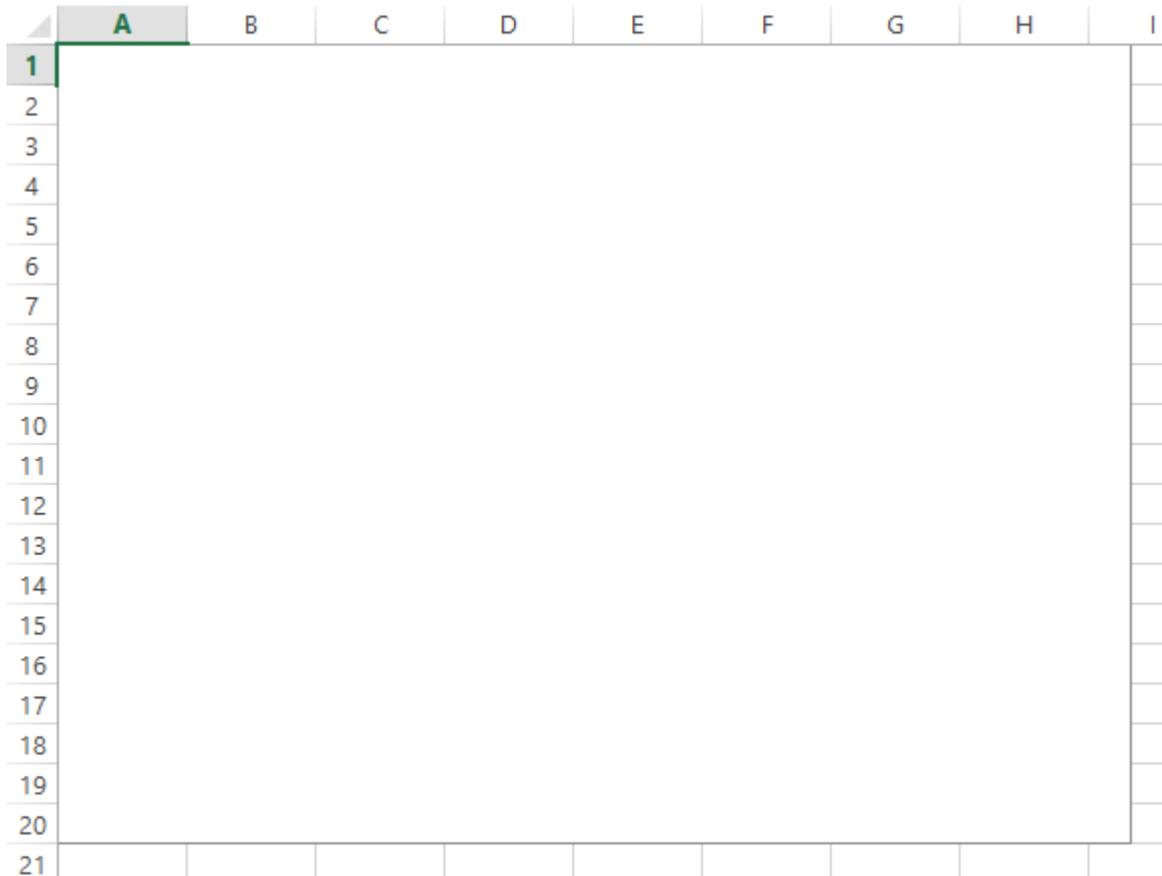
Il punto di partenza per la maggior parte del codice dei grafici è creare un `Chart` vuoto. Si noti che questo `Chart` è soggetto al modello di grafico predefinito che è attivo e potrebbe non essere effettivamente vuoto (se il modello è stato modificato).

La chiave di `ChartObject` sta determinando la sua posizione. La sintassi per la chiamata è `ChartObjects.Add(Left, Top, Width, Height)`. Una volta creato `ChartObject`, è possibile utilizzare il relativo oggetto `Chart` per modificare effettivamente il grafico. `ChartObject` si comporta più come una `Shape` per posizionare il grafico sul foglio.

Codice per creare un grafico vuoto

```
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
```

Grafico risultante



Creare un grafico modificando la formula SERIE

Per il controllo completo su un nuovo oggetto `Chart` e `Series` (in particolare per un nome `Series` dinamico), è necessario ricorrere alla modifica della formula `SERIES` direttamente. Il processo per impostare gli oggetti `Range` è semplice e l'ostacolo principale è semplicemente la costruzione di stringhe per la formula `SERIES`.

La formula `SERIES` prende la seguente sintassi:

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

Questi contenuti possono essere forniti come riferimenti o come valori di matrice per gli elementi di dati. `Order` rappresenta la posizione della serie all'interno del grafico. Si noti che i riferimenti ai dati non funzioneranno a meno che non siano pienamente qualificati con il nome del foglio. Per un esempio di una formula di lavoro, fai clic su qualsiasi serie esistente e controlla la barra della formula.

Codice per creare un grafico e impostare i dati utilizzando la formula `SERIES`

Si noti che la creazione di stringhe per creare la formula `SERIES` utilizza `.Address(,,, True)`. Ciò garantisce che venga utilizzato il riferimento `Range` esterno in modo da includere un indirizzo completo con il nome del foglio. Si **verificherà un errore se il nome del foglio è escluso**.

```
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)"

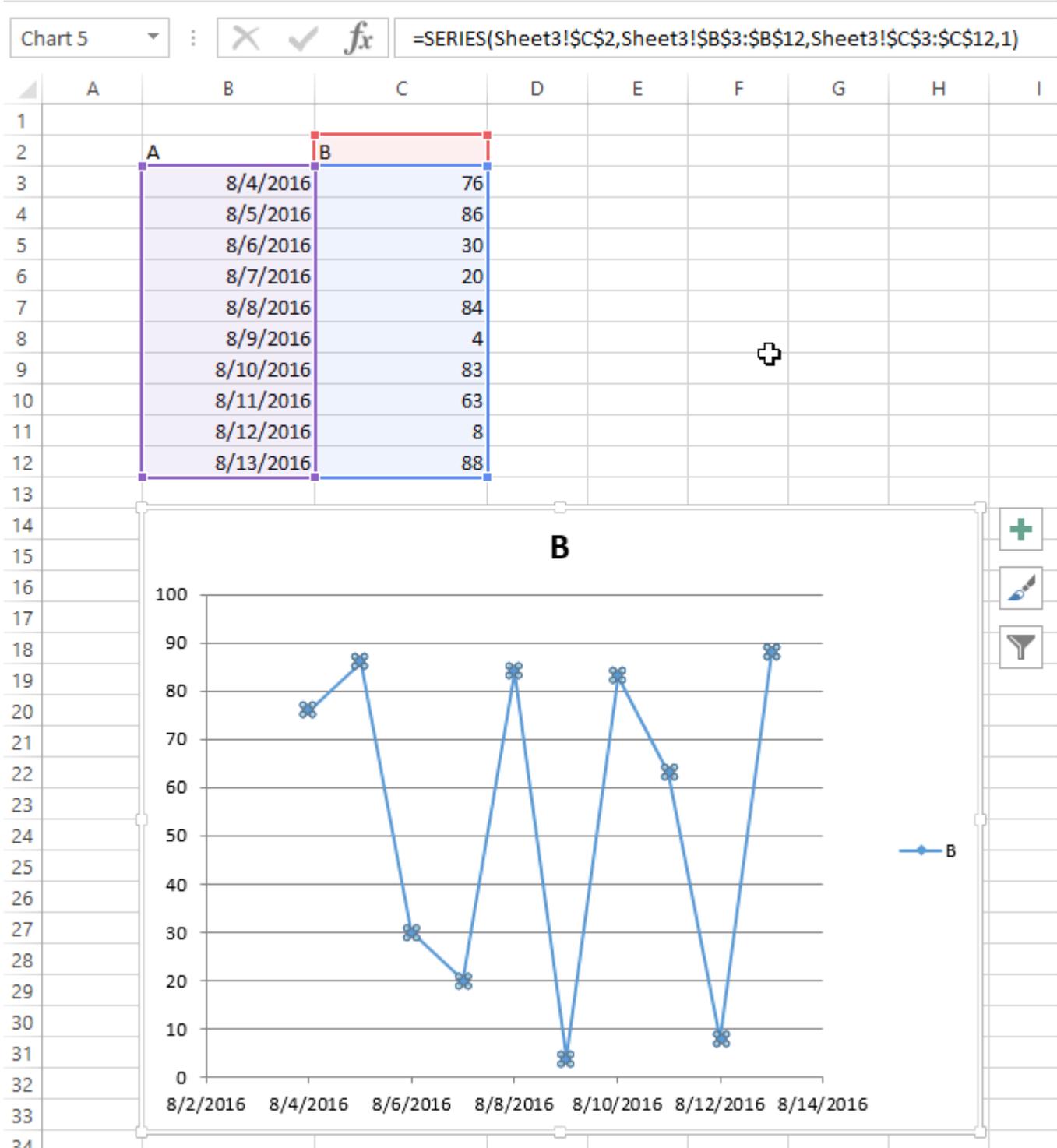
ser.Formula = formulaValue
ser.ChartType = xlXYScatterLines

End Sub

```

Dati originali e grafico risultante

Si noti che per questo grafico, il nome della serie è impostato correttamente con un intervallo per la cella desiderata. Ciò significa che gli aggiornamenti si propagheranno al `Chart` .



Organizzazione di grafici in una griglia

Una routine comune con i grafici in Excel è la standardizzazione delle dimensioni e del layout di più grafici su un singolo foglio. Se fatto manualmente, puoi tenere premuto **ALT** mentre ridimensiona o spostando il grafico per "attaccare" ai contorni delle celle. Questo funziona per un paio di grafici, ma un approccio VBA è molto più semplice.

Codice per creare una griglia

Questo codice creerà una griglia di grafici a partire da una determinata posizione (in alto a sinistra), con un numero definito di colonne e una dimensione di grafico comune definita. I grafici

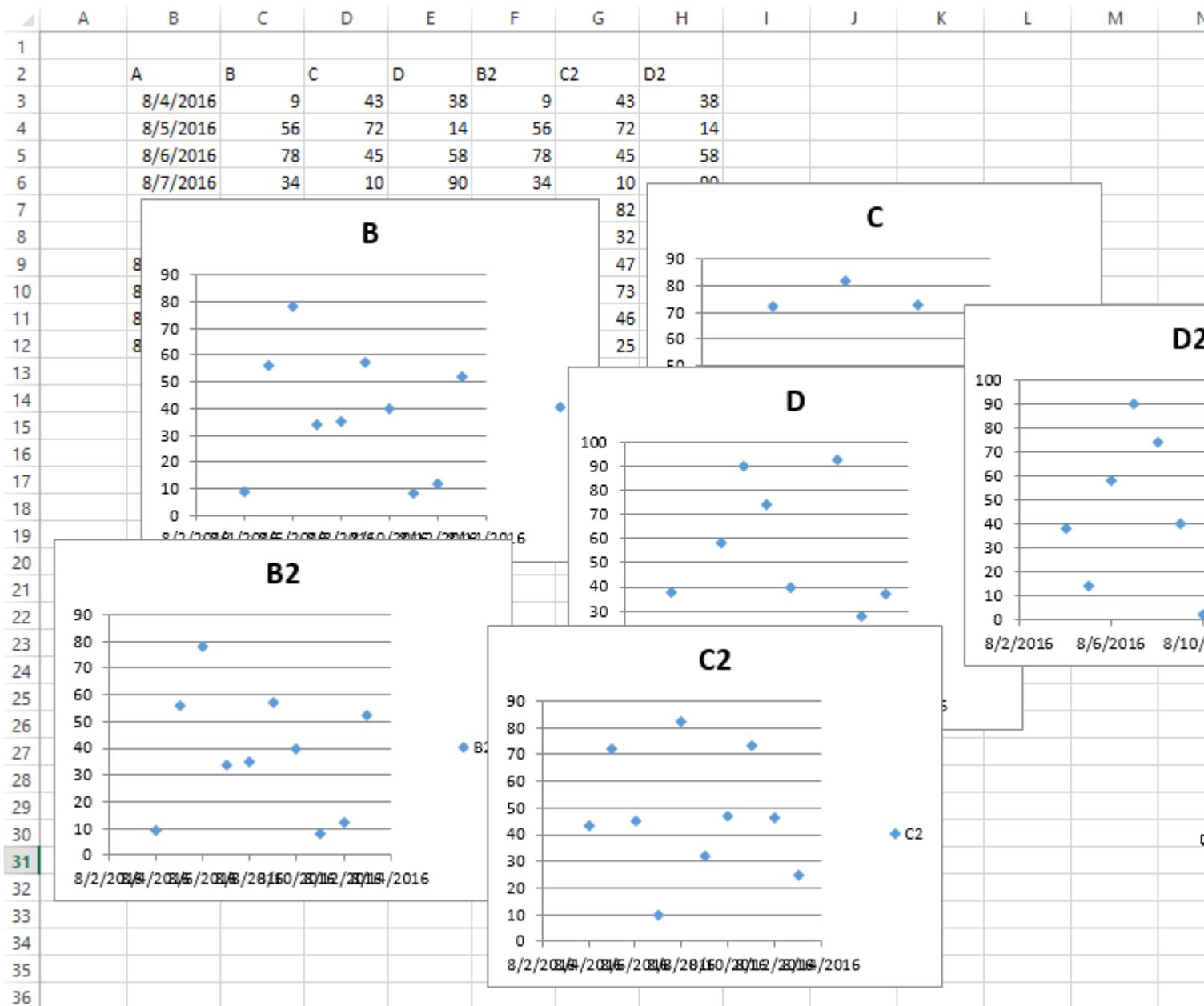
verranno posizionati nell'ordine in cui sono stati creati e avvolti attorno al bordo per formare una nuova riga.

```
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
```

Risultato con diversi grafici

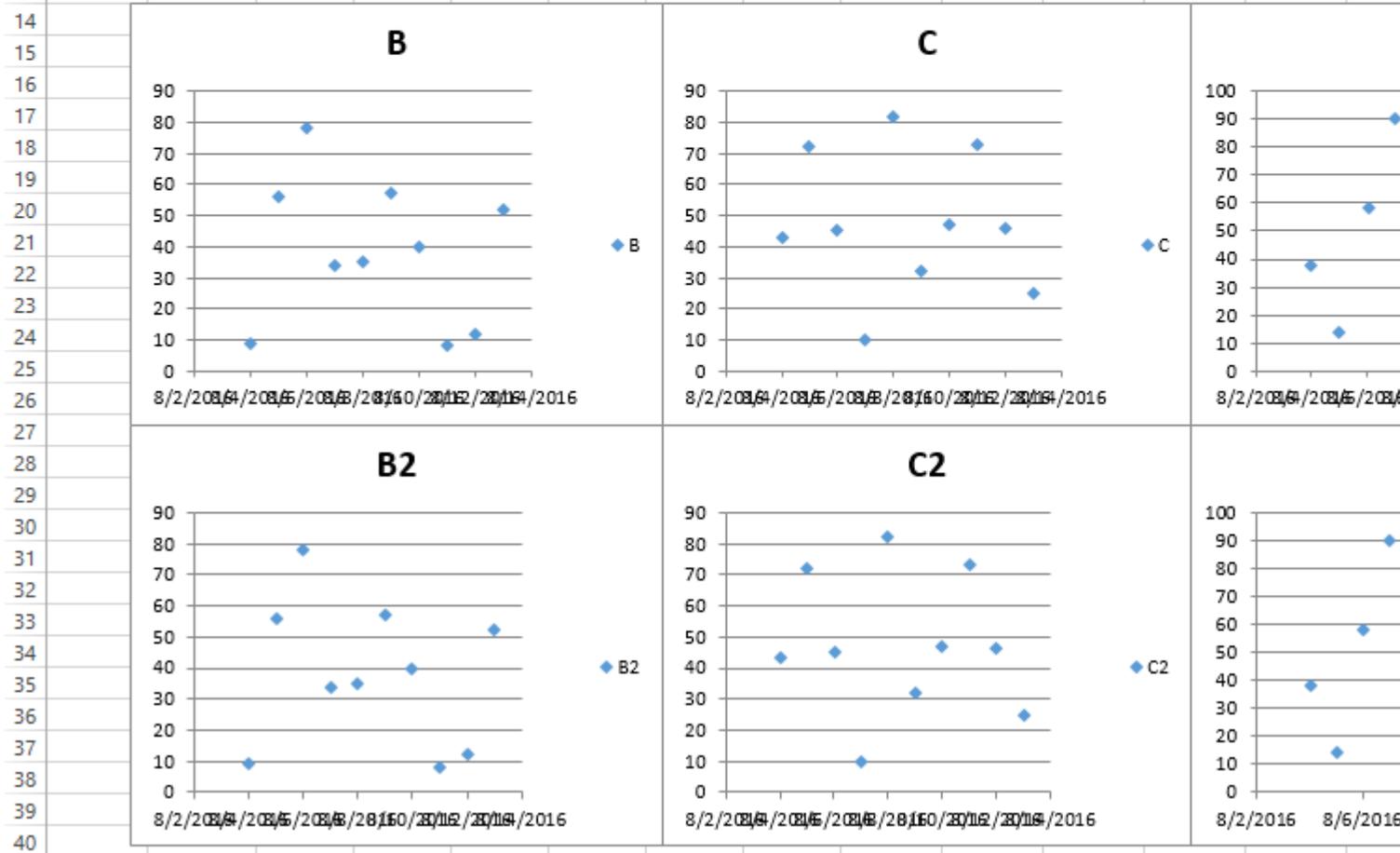
Queste immagini mostrano il layout casuale originale dei grafici e la griglia risultante dall'esecuzione del codice sopra.

Prima



Dopo

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
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						



Leggi Grafici e grafici online: <https://riptutorial.com/it/excel-vba/topic/4968/grafici-e-grafici>

Capitolo 18: Individuazione di valori duplicati in un intervallo

introduzione

In alcuni punti, si valuterà una serie di dati e sarà necessario individuare i duplicati in essa contenuti. Per i set di dati più grandi, è possibile adottare diversi approcci che utilizzano il codice VBA o le funzioni condizionali. Questo esempio utilizza una condizione if-then semplice all'interno di due cicli for-next annidati per verificare se ciascuna cella nell'intervallo è uguale in valore a qualsiasi altra cella dell'intervallo.

Examples

Trova duplicati in un intervallo

I seguenti test vanno da A2 a A7 per valori duplicati. **Nota:** questo esempio illustra una possibile soluzione come primo approccio a una soluzione. È più rapido utilizzare un array che un intervallo e uno potrebbe utilizzare raccolte o dizionari o metodi xml per verificare la presenza di duplicati.

```
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
```

A seconda delle esigenze, l'esempio può essere modificato - ad esempio, il limite superiore di n può essere il valore di riga dell'ultima cella con i dati nell'intervallo o l'azione in caso di una condizione True If può essere modificata per estrarre il duplicato valore da qualche altra parte. Tuttavia, i meccanismi della routine non cambierebbero.

Leggi Individuazione di valori duplicati in un intervallo online: <https://riptutorial.com/it/excel-vba/topic/8295/individuazione-di-valori-duplicati-in-un-intervallo>

Capitolo 19: Integrazione di PowerPoint tramite VBA

Osservazioni

Questa sezione dimostra una varietà di modi per interagire con PowerPoint tramite VBA. Dalla visualizzazione dei dati sulle diapositive alla creazione di grafici, PowerPoint è uno strumento molto potente se utilizzato in combinazione con Excel. Pertanto, questa sezione cerca di dimostrare i vari modi in cui VBA può essere utilizzato per automatizzare questa interazione.

Examples

Nozioni di base: avvio di PowerPoint da VBA

Mentre ci sono molti parametri che possono essere modificati e varianti che possono essere aggiunte a seconda della funzionalità desiderata, questo esempio illustra il framework di base per l'avvio di PowerPoint.

Nota: questo codice richiede che il riferimento PowerPoint sia stato aggiunto al progetto VBA attivo. Vedere la voce Documentazione di [riferimento](#) per informazioni su come abilitare il riferimento.

Innanzitutto, definire le variabili per l'applicazione, la presentazione e gli oggetti diapositiva. Mentre questo può essere fatto con l'associazione tardiva, è sempre meglio usare il binding anticipato quando applicabile.

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

Quindi, aprì o crea una nuova istanza dell'applicazione PowerPoint. Qui, la chiamata `On Error Resume Next` viene utilizzata per evitare che venga generato un errore da `GetObject` se PowerPoint non è stato ancora aperto. Vedere l'esempio di [gestione degli errori](#) del Best Practices Topic per una spiegazione più dettagliata.

```
'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 volta avviata l'applicazione, viene generata una nuova presentazione e successivamente una

diapositiva.

```
'Generate new Presentation and slide for graphic creation
Set PPPres = PPAApp.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 completamento di questo codice, verrà aperta una nuova finestra di PowerPoint con una diapositiva vuota. Utilizzando le variabili dell'oggetto, è possibile aggiungere le forme, il testo, la grafica e gli intervalli di Excel come desiderato

Leggi **Integrazione di PowerPoint tramite VBA online**: <https://riptutorial.com/it/excel-vba/topic/2327/integrazione-di-powerpoint-tramite-vba>

Capitolo 20: Lavorare con le tabelle di Excel in VBA

introduzione

Questo argomento riguarda l'utilizzo delle tabelle in VBA e presuppone la conoscenza delle tabelle di Excel. In VBA, o meglio nel modello a oggetti di Excel, le tabelle sono conosciute come ListObjects. Le proprietà più utilizzate di un oggetto ListObject sono ListRow (s), ListColumn (s), DataBodyRange, Range e HeaderRowRange.

Examples

Istanziare un oggetto ListObject

```
Dim lo as ListObject
Dim MyRange as Range

Set lo = Sheet1.ListObjects(1)

'or

Set lo = Sheet1.ListObjects("Table1")

'or

Set lo = MyRange.ListObject
```

Lavorare 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
```

Conversione di una tabella di Excel in un intervallo normale

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

Leggi **Lavorare con le tabelle di Excel in VBA online**: <https://riptutorial.com/it/excel-vba/topic/9753/lavorare-con-le-tabelle-di-excel-in-vba>

Capitolo 21: Le cartelle di lavoro

Examples

Cartelle di lavoro dell'applicazione

In molte applicazioni Excel, il codice VBA accetta azioni dirette alla cartella di lavoro in cui è contenuta. Si salva quella cartella di lavoro con un'estensione ".xlsm" e le macro VBA si concentrano solo sui fogli di lavoro e sui dati all'interno. Tuttavia, spesso è necessario combinare o unire i dati di altre cartelle di lavoro o scrivere alcuni dei tuoi dati in una cartella di lavoro separata. Aprire, chiudere, salvare, creare e cancellare altre cartelle di lavoro è una necessità comune per molte applicazioni VBA.

In qualsiasi momento nell'Editor VBA, è possibile visualizzare e accedere a qualsiasi cartella di lavoro attualmente aperta da tale istanza di Excel utilizzando la proprietà `Workbooks` dell'oggetto `Application`. La [documentazione MSDN](#) la spiega con riferimenti.

Quando utilizzare `ActiveWorkbook` e `ThisWorkbook`

È una best practice VBA per specificare sempre quale cartella di lavoro si riferisce al codice VBA. Se questa specifica viene omessa, VBA assume che il codice sia indirizzato alla cartella di lavoro attualmente attiva (`ActiveWorkbook`).

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

Tuttavia, quando più cartelle di lavoro sono aperte contemporaneamente - in particolare e in particolare quando il codice VBA è in esecuzione da un componente aggiuntivo di Excel - i riferimenti a `ActiveWorkbook` potrebbero essere confusi o indirizzati male. Ad esempio, un componente aggiuntivo con una UDF che controlla l'ora del giorno e lo confronta con un valore memorizzato su uno dei fogli di lavoro del componente aggiuntivo (che in genere non sono facilmente visibili all'utente) dovrà identificare esplicitamente quale cartella di lavoro è essere referenziato. Nel nostro esempio, la nostra cartella di lavoro aperta (e attiva) ha una formula nella cella A1 `=EarlyOrLate()` e NON ha alcun VBA scritto per quella cartella di lavoro attiva. Nel nostro componente aggiuntivo, abbiamo la seguente funzione definita dall'utente (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
```

Il codice per l'UDF viene scritto e memorizzato nel componente aggiuntivo di Excel installato. Usa i dati memorizzati su un foglio di lavoro nel componente aggiuntivo chiamato "WatchTime". Se

l'UDF avesse utilizzato `ActiveWorkbook` invece di `ThisWorkbook` , non sarebbe mai stato in grado di garantire quale cartella di lavoro era destinata.

Aprire una (nuova) cartella di lavoro, anche se è già aperta

Se desideri accedere a una cartella di lavoro che è già aperta, ottenere il compito dall'insieme di `Workbooks` è semplice:

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

Se si desidera creare una nuova cartella di lavoro, utilizzare l'oggetto di raccolta `Workbooks` per `Add` una nuova voce.

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

Ci sono momenti in cui non puoi o (o ti importi) se la cartella di lavoro che ti serve è già aperta o no, o se non esiste. La funzione di esempio mostra come restituire sempre un oggetto di cartella di lavoro valido.

```
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. "
        End Select
        & _
        "Please use a known extension: xlsx, xlsm, xls, or xlsb."
    End If
    returnedWB = Workbooks.Add(folderFile, fileFormatNum)
    Set GetWorkbook = returnedWB
End Function
```

```

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

```

Salvataggio di una cartella di lavoro senza chiedere l'utente

Salvare spesso nuovi dati in una cartella di lavoro esistente utilizzando VBA causerà una domanda a comparsa che informa che il file esiste già.

Per evitare questa domanda pop-up, devi sopprimere questi tipi di avvisi.

```

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

```

Modifica del numero predefinito di fogli di lavoro in una nuova cartella di lavoro

Il numero "factory default" di fogli di lavoro creati in una nuova cartella di lavoro di Excel è generalmente impostato su tre. Il tuo codice VBA può impostare esplicitamente il numero di fogli di lavoro in una nuova cartella di lavoro.

```

'--- 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 = oldsheetscount
End With

```

Leggi Le cartelle di lavoro online: <https://riptutorial.com/it/excel-vba/topic/2969/le-cartelle-di-lavoro>

Capitolo 22: Metodi per trovare l'ultima riga o colonna usata in un foglio di lavoro

Osservazioni

Puoi trovare una buona spiegazione sul perché altri metodi sono scoraggiati / inaccurati qui: <http://stackoverflow.com/a/11169920/4628637>

Examples

Trova l'ultima cella non vuota in una colonna

In questo esempio, esamineremo un metodo per restituire l'ultima riga non vuota in una colonna per un set di dati.

Questo metodo funziona indipendentemente dalle regioni vuote all'interno del set di dati.

Tuttavia, è *necessario prestare attenzione se sono coinvolte le celle unite*, poiché il metodo `End` verrà "arrestato" contro una regione unita, restituendo la prima cella della regione unita.

Inoltre, le celle non vuote nelle *righe nascoste* non saranno prese in considerazione.

```
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
```

Per affrontare le limitazioni sopra indicate, la linea:

```
LastRow = wS.Cells(wS.Rows.Count, "A").End(xlUp).Row
```

può essere sostituito con:

1. per l'ultima riga utilizzata di "Sheet1" :

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

2. per l'ultima cella non vuota della colonna "A" in "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
```

Trova l'ultima riga usando l'intervallo con nome

Nel caso in cui tu abbia un intervallo denominato nel tuo foglio e desideri ottenere in modo dinamico l'ultima riga di tale intervallo di nomi dinamici. Copre anche i casi in cui l'intervallo denominato non inizia dalla prima riga.

```
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
```

Aggiornare:

Una potenziale scappatoia è stata evidenziata da @Jeeped per un intervallo denominato con righe non contigue poiché genera risultati imprevisti. Per risolvere questo problema, il codice è stato modificato come di seguito.

Asunci: `targes sheet = form` , `named range = MyNameRange`

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

Prendi la riga dell'ultima cella in un intervallo

```
'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
```

Trova l'ultima colonna non vuota nel foglio di lavoro

```
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
```

Puoi scegliere tra 2 opzioni, per quanto riguarda se vuoi sapere se non ci sono dati nel foglio di lavoro:

- **NO:** usa LastCol_1: puoi usarlo direttamente all'interno di `wS.Cells(..., LastCol_1(wS))`
- **SÌ:** utilizzare LastCol_0: è necessario verificare se il risultato ottenuto dalla funzione è 0 o meno prima di utilizzarlo

```
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
```

Le proprietà dell'oggetto Err vengono automaticamente ripristinate a zero all'uscita dalla funzione.

```
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
```

Ultima cella in Range.CurrentRegion

[Range.CurrentRegion](#) è un'area di intervallo rettangolare circondata da celle vuote. Le celle vuote con formule come `=""` o `'` non sono considerate vuote (anche con la funzione Excel di [ISBLANK](#)).

```
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)
```

Trova l'ultima riga non vuota nel foglio di lavoro

```
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
```

Puoi scegliere tra 2 opzioni, per quanto riguarda se vuoi sapere se non ci sono dati nel foglio di lavoro:

- **NO:** usa `LastRow_1`: puoi usarlo direttamente all'interno di `wS.Cells(LastRow_1(wS), ...)`
- **SÌ:** utilizzare `LastRow_0`: è necessario verificare se il risultato ottenuto dalla funzione è 0 o meno prima di utilizzarlo

```
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
```

Trova l'ultima cella non vuota in una riga

In questo esempio, esamineremo un metodo per restituire l'ultima colonna non vuota in una riga.

Questo metodo funziona indipendentemente dalle regioni vuote all'interno del set di dati.

Tuttavia, è *necessario prestare attenzione se sono coinvolte le celle unite*, poiché il metodo `End` verrà "arrestato" contro una regione unita, restituendo la prima cella della regione unita.

Inoltre, le celle non vuote nelle *colonne nascoste* non saranno prese in considerazione.

```

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

```

Trova l'ultima cella non vuota nel foglio di lavoro - Prestazioni (matrice)

- La prima funzione, utilizzando un array, è **molto più veloce**
- Se chiamato senza il parametro opzionale, verrà `.ThisWorkbook.ActiveSheet` predefinito in `.ThisWorkbook.ActiveSheet`
- Se l'intervallo è vuoto, restituirà `Cell(1, 1)` come valore predefinito, anziché `Nothing`

Velocità:

```

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

```

.Misurato 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
        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
```

Leggi Metodi per trovare l'ultima riga o colonna usata in un foglio di lavoro online:

<https://riptutorial.com/it/excel-vba/topic/918/metodi-per-trovare-l-ultima-riga-o-colonna-usata-in-un-foglio-di-lavoro>

Capitolo 23: Oggetto dell'applicazione

Osservazioni

Excel VBA è dotato di un *modello di oggetti* completo che contiene classi e oggetti che è possibile utilizzare per manipolare qualsiasi parte dell'applicazione Excel in esecuzione. Uno degli oggetti più comuni che utilizzerai è l'oggetto **Application**. Questo è un catchall di livello superiore che rappresenta l'istanza corrente in esecuzione di Excel. Quasi tutto ciò che non è collegato a una determinata cartella di lavoro di Excel è nell'oggetto **Application**.

L'oggetto *Application*, come oggetto di livello superiore, ha letteralmente centinaia di proprietà, metodi ed eventi che possono essere utilizzati per controllare ogni aspetto di Excel.

Examples

Esempio di oggetto applicazione semplice: ridurre a icona la finestra di Excel

Questo codice utilizza l'oggetto **Application** di livello superiore per ridurre a icona la finestra principale di Excel.

```
Sub MinimizeExcel()  
  
    Application.WindowState = xlMinimized  
  
End Sub
```

Esempio di oggetto applicazione semplice: visualizzazione di Excel e versione VBE

```
Sub DisplayExcelVersions()  
  
    MsgBox "The version of Excel is " & Application.Version  
    MsgBox "The version of the VBE is " & Application.VBE.Version  
  
End Sub
```

L'utilizzo della proprietà `Application.Version` è utile per garantire che il codice funzioni solo su una versione compatibile di Excel.

Leggi Oggetto dell'applicazione online: <https://riptutorial.com/it/excel-vba/topic/5645/oggetto-dell-applicazione>

Capitolo 24: Ottimizzazione Excel-VBA

introduzione

L'ottimizzazione VBA di Excel si riferisce anche alla codifica della migliore gestione degli errori mediante documentazione e dettagli aggiuntivi. Questo è mostrato qui.

Osservazioni

*) I numeri di riga rappresentano numeri interi, ovvero un tipo di dati a 16 bit con segno compreso nell'intervallo compreso tra -32.768 e 32.767, altrimenti viene generato un overflow. Di solito i numeri di riga vengono inseriti in passaggi di 10 su una parte del codice o tutte le procedure di un modulo nel suo insieme.

Examples

Disabilitare l'aggiornamento del foglio di lavoro

Disabilitare il calcolo del foglio di lavoro può ridurre significativamente il tempo di esecuzione della macro. Inoltre, la disattivazione di eventi, l'aggiornamento dello schermo e le interruzioni di pagina sarebbero utili. A seguito di `Sub` può essere utilizzato in qualsiasi macro per questo scopo.

```
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
```

Per l'ottimizzazione segui il seguente pseudo-codice:

```
Sub MyCode()
    OptimizeVBA True

    'Your code goes here

    OptimizeVBA False
End Sub
```

Controllo del tempo di esecuzione

Procedure diverse possono dare lo stesso risultato, ma utilizzerebbero tempi di elaborazione diversi. Per verificare quale è più veloce, è possibile utilizzare un codice come questo:

```
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

```

Utilizzo dei blocchi

L'utilizzo con i blocchi può accelerare il processo di esecuzione di una macro. Invece di scrivere un intervallo, un nome di grafico, un foglio di lavoro, ecc. Puoi usare con blocchi come sotto;

```

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

```

Quale è più veloce di questo:

```

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

```

Gli appunti:

- Una volta inserito un blocco With, l'oggetto non può essere modificato. Di conseguenza, non è possibile utilizzare una singola istruzione With per influire su un numero di oggetti diversi
- **Non saltare dentro o fuori con i blocchi** . Se vengono eseguite istruzioni in un blocco With, ma l'istruzione With o End With non viene eseguita, **una variabile temporanea contenente un riferimento all'oggetto rimane in memoria finché non si esce dalla procedura**
- Non eseguire il ciclo all'interno di istruzioni With, in particolare se l'oggetto memorizzato nella cache viene utilizzato come iteratore
- È possibile annidare le affermazioni inserendo un blocco With in un altro. Tuttavia, poiché i membri dei blocchi esterni With sono mascherati all'interno dei blocchi With interni, è necessario fornire un riferimento oggetto completo in un blocco With interno a qualsiasi membro di un oggetto in un blocco With esterno.

Esempio di annidamento:

Questo esempio utilizza l'istruzione With per eseguire una serie di istruzioni su un singolo oggetto.

L'oggetto e le sue proprietà sono nomi generici usati solo a scopo illustrativo.

```
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
```

Ulteriori informazioni su [MSDN](#)

Cancellazione riga - Prestazioni

- L'eliminazione delle righe è lenta, specialmente quando si esegue il looping delle celle e si eliminano le righe, una alla volta
- Un approccio diverso utilizza un filtro automatico per nascondere le righe da eliminare
- Copia l'intervallo visibile e incollalo in un nuovo foglio di lavoro
- Rimuovere completamente il foglio iniziale
- Con questo metodo, più file verranno eliminate, più velocemente sarà

Esempio:

```
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 Criterial and 2 accordingly)  
        .AutoFilter Field:=1, Criterial:="<>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
```

Disabilitazione di tutte le funzionalità di Excel Prima di eseguire macro di grandi dimensioni

Le procedure qui sotto disabilitano temporaneamente tutte le funzionalità di Excel a livello di Workbook e di Worksheet

- FastWB () è un interruttore che accetta i flag On o Off
- FastWS () accetta un oggetto Worksheet facoltativo o nessuno
- Se manca il parametro ws, attiva e disattiva tutte le funzionalità per tutti i fogli di lavoro nella raccolta
 - È possibile utilizzare un tipo personalizzato per acquisire tutte le impostazioni prima di spegnerle
 - Alla fine del processo, le impostazioni iniziali possono essere ripristinate

```
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  
    End With
```

```

    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

```

Ripristina tutte le impostazioni di Excel sui valori predefiniti

```

Public Sub XlResetSettings() 'default Excel settings
    With Application
        .Calculation = xlCalculationAutomatic
        .DisplayAlerts = True
        .DisplayStatusBar = True
        .EnableAnimations = False
        .EnableEvents = True
        .ScreenUpdating = True
    End With
    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

```

Ottimizzazione della ricerca errori tramite debug esteso

Usare i numeri di riga ... e documentarli in caso di errore ("L'importanza di vedere Erl")

Rilevare quale linea genera un errore è una parte sostanziale di qualsiasi debug e restringe la ricerca della causa. Per documentare le righe di errore identificate con una breve descrizione si completa un corretto rilevamento degli errori, al meglio insieme ai nomi del modulo e della procedura. L'esempio seguente salva questi dati in un file di registro.

Sfondo

L'oggetto error restituisce il numero di errore (Err.Number) e la descrizione dell'errore (Err.Description), ma non risponde esplicitamente alla domanda su dove individuare l'errore. La funzione di **Erl** , tuttavia, lo fa, ma a condizione di aggiungere * *numeri di linea*) al codice (una delle numerose altre concessioni alle precedenti ore di base).

Se non ci sono righe di errore, la funzione Erl restituisce 0, se la numerazione è incompleta si otterrà l'ultimo numero di riga precedente della procedura.

```
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
```

' Codice aggiuntivo per mostrare il file di 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
```

Leggi Ottimizzazione Excel-VBA online: <https://riptutorial.com/it/excel-vba/topic/9798/ottimizzazione-excel-vba>

Capitolo 25: Passa in rassegna tutti i fogli in Active Workbook

Examples

Recupera tutti i nomi dei fogli di lavoro nella cartella di lavoro attiva

```
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
```

Passa attraverso tutti i fogli in tutti i file in una cartella

```
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
specifying 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
```

Leggi **Passa in rassegna tutti i fogli in Active Workbook online**: <https://riptutorial.com/it/excel-vba/topic/1144/passa-in-rassegna-tutti-i-fogli-in-active-workbook>

Capitolo 26: Rilegatura

Examples

Early Binding vs Late Binding

Binding è il processo di assegnazione di un oggetto a un identificatore o nome di variabile. L'associazione anticipata (nota anche come associazione statica) si verifica quando un oggetto dichiarato in Excel è di un tipo di oggetto specifico, ad esempio un foglio di lavoro o una cartella di lavoro. L'associazione tardiva si verifica quando vengono create associazioni di oggetti generali, ad esempio i tipi di dichiarazione Object e Variant.

Legame anticipato delle referenze alcuni vantaggi rispetto al legame tardivo.

- L'associazione anticipata è operativamente più veloce dell'aggiornamento tardivo durante l'esecuzione. La creazione dell'oggetto con l'associazione tardiva in fase di esecuzione richiede tempo per l'associazione anticipata quando il progetto VBA viene inizialmente caricato.
- L'associazione anticipata offre funzionalità aggiuntive attraverso l'identificazione delle coppie chiave / oggetto in base alla loro posizione ordinale.
- A seconda della struttura del codice, l'associazione anticipata può offrire un ulteriore livello di controllo del tipo e riduzione degli errori.
- La correzione della maiuscola del VBE durante la digitazione delle proprietà e dei metodi di un oggetto associato è attiva con l'associazione anticipata ma non è disponibile con l'associazione tardiva.

Nota: è necessario aggiungere il riferimento appropriato al progetto VBA tramite il comando Strumenti → Riferimenti di VBE per implementare l'associazione anticipata.

Questo riferimento bibliografico viene quindi portato con il progetto; non deve essere rinviato quando il progetto VBA viene distribuito ed eseguito su un altro computer.

```
'Looping through a dictionary that was created with late binding1
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

```

Tuttavia, se si utilizza l'associazione anticipata e il documento viene eseguito su un sistema privo di una delle librerie a cui si fa riferimento, si verificheranno dei problemi. Non solo le routine che utilizzano la libreria mancante non funzioneranno correttamente, ma il comportamento di tutto il codice all'interno del documento diventerà irregolare. È probabile che nessuno del codice del documento funzioni su quel computer.

È qui che l'associazione tardiva è vantaggiosa. Quando si utilizza l'associazione tardiva non è necessario aggiungere il riferimento nel menu Strumenti> Riferimenti. Su macchine che dispongono della libreria appropriata, il codice funzionerà ancora. Su macchine senza quella libreria, i comandi che fanno riferimento alla libreria non funzioneranno, ma tutto il resto del codice continuerà a funzionare.

Se non si ha familiarità con la libreria a cui si fa riferimento, potrebbe essere utile utilizzare l'associazione anticipata durante la scrittura del codice, quindi passare all'associazione tardiva prima della distribuzione. In questo modo è possibile sfruttare l'IntelliSense e il Visualizzatore oggetti di VBE durante lo sviluppo.

Leggi Rilegatura online: <https://riptutorial.com/it/excel-vba/topic/3811/rilegatura>

Capitolo 27: Sicurezza VBA

Examples

Password Proteggi il tuo VBA

A volte hai informazioni sensibili nel tuo VBA (ad es. Password) a cui non vuoi che gli utenti abbiano accesso. È possibile ottenere la protezione di base su queste informazioni proteggendo la password del progetto VBA.

Segui questi passi:

1. Apri il tuo editor di Visual Basic (Alt + F11)
2. Passa a Strumenti -> Proprietà VBAProject ...
3. Passare alla scheda Protezione
4. Seleziona la casella di controllo "Blocca il progetto per la visualizzazione"
5. Immettere la password desiderata nelle caselle di testo Password e Conferma password

Ora, quando qualcuno vuole accedere al tuo codice all'interno di un'applicazione di Office, dovrà prima inserire la password. Si noti, tuttavia, che anche una password di progetto VBA forte è banale da rompere.

Leggi Sicurezza VBA online: <https://riptutorial.com/it/excel-vba/topic/7642/sicurezza-vba>

Capitolo 28: SQL in Excel VBA: best practice

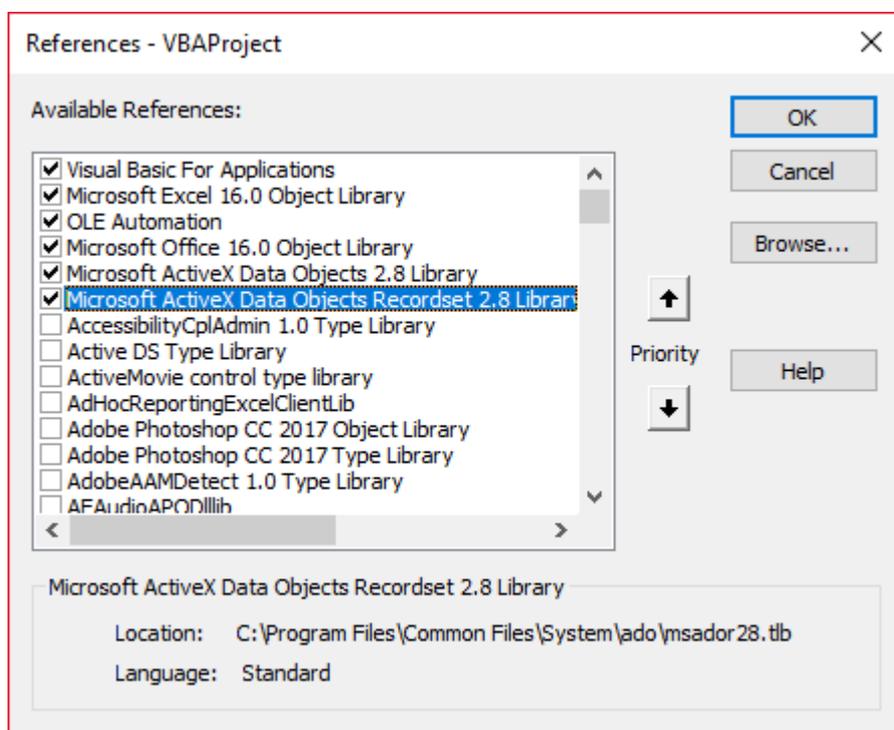
Examples

Come utilizzare ADODB.Connection in VBA?

Requisiti:

Aggiungi i seguenti riferimenti al progetto:

- Libreria Microsoft ActiveX Data Objects 2.8
- Libreria Microsoft ActiveX Data Objects Recordset 2.8



Dichiarare le variabili

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

Crea una connessione

un. con autenticazione di Windows

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

b. con autenticazione di 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
```

Esegui il comando sql

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

Leggi i dati dal set di record

```
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
```

Chiudere la connessione

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

Come 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
```

Risultato

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

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

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

ShipperID: 4 CompanyName: telefono di spedizione veloce: (503) 555-1234

Leggi SQL in Excel VBA: best practice online: <https://riptutorial.com/it/excel-vba/topic/9958/sql-in-excel-vba--best-practice>

Capitolo 29: Suggerimenti e trucchi Excel VBA

Osservazioni

Questo argomento consiste in una vasta gamma di suggerimenti e trucchi utili scoperti dagli utenti SO attraverso la loro esperienza nella codifica. Questi sono spesso esempi di modi per eludere le comuni frustrazioni o modi di usare Excel in un modo più "intelligente".

Examples

Usando i fogli xlVeryHidden

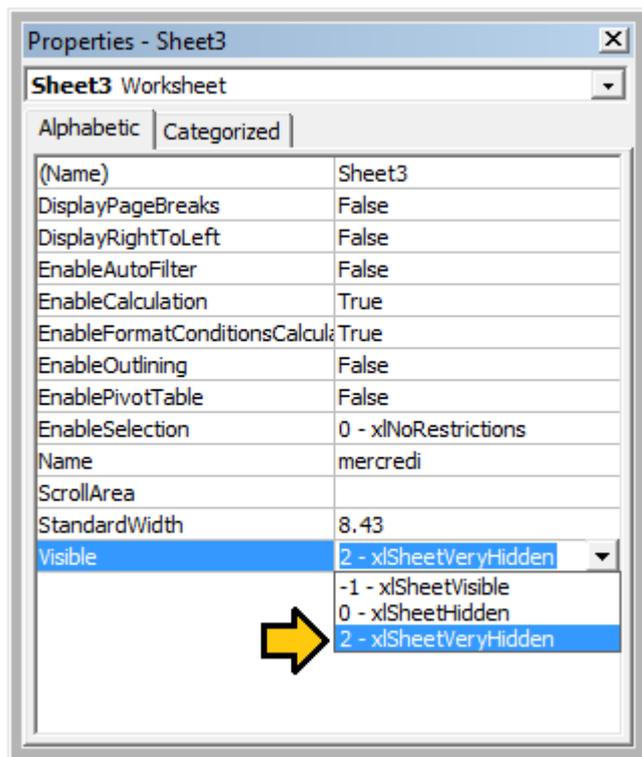
I fogli di lavoro in Excel hanno tre opzioni per la proprietà `Visible`. Queste opzioni sono rappresentate da costanti nell'enumerazione `xlSheetVisibility` e sono le seguenti:

1. `xlVisible` o `xlSheetVisible` : -1 (il valore predefinito per i nuovi fogli)
2. `xlHidden` o `xlSheetHidden` valore: 0
3. `xlVeryHidden` o `xlSheetVeryHidden` valore: 2

I fogli visibili rappresentano la visibilità predefinita per i fogli. Sono visibili nella barra delle schede dei fogli e possono essere liberamente selezionati e visualizzati. I fogli nascosti sono nascosti dalla barra delle schede dei fogli e non sono quindi selezionabili. Tuttavia, i fogli nascosti possono essere nascosti dalla finestra di Excel facendo clic con il tasto destro sulle schede del foglio e selezionando "Scopri"

I fogli Very Hidden, d'altra parte, sono accessibili *solo* tramite Visual Basic Editor. Questo li rende uno strumento incredibilmente utile per l'archiviazione dei dati tra le istanze di Excel e per l'archiviazione dei dati che devono essere nascosti agli utenti finali. È possibile accedere ai fogli mediante un riferimento denominato all'interno del codice VBA, consentendo un facile utilizzo dei dati memorizzati.

Per modificare manualmente la proprietà `.Visible` di un foglio di lavoro su `xlSheetVeryHidden`, aprire la finestra Proprietà di VBE (`F4`), selezionare il foglio di lavoro che si desidera modificare e utilizzare il menu a discesa nella tredicesima riga per effettuare la selezione.



Per modificare la proprietà `.Visible` di un foglio di lavoro su `xlSheetVeryHidden`¹ nel codice, accedere in modo analogo alla proprietà `.Visible` e assegnare un nuovo valore.

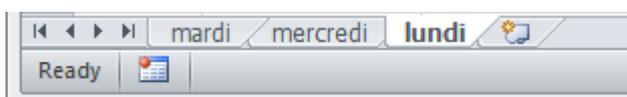
```
with Sheet3
    .Visible = xlSheetVeryHidden
end with
```

¹ Sia `xlVeryHidden` che `xlSheetVeryHidden` restituiscono un valore numerico di **2** (sono intercambiabili).

Foglio di lavoro. Nome, `.Index` o `.CodeName`

Sappiamo che "best practice" impone che a un oggetto range debba essere esplicitamente fatto riferimento il foglio di lavoro principale. Un foglio di lavoro può essere riferito alla sua proprietà `.Name`, alla proprietà numerica `.Index` o alla sua proprietà `.CodeName` ma un utente può riordinare la coda del foglio semplicemente trascinando una scheda nome o rinominare il foglio con un doppio clic sulla stessa scheda e alcuni digitando una cartella di lavoro non protetta.

Considera un foglio di lavoro standard tre. In questo ordine sono stati rinominati i tre fogli di lavoro lunedì, martedì e mercoledì e le sottosezioni VBA codificate che fanno riferimento a questi. Ora considera che un utente arriva e decide che il lunedì appartiene alla fine della coda del foglio di lavoro, poi ne arriva un altro e decide che i nomi del foglio di lavoro sembrano migliori in francese. Ora hai una cartella di lavoro con una coda della scheda del nome del foglio di lavoro che assomiglia a quanto segue.



Se avessi usato uno dei seguenti metodi di riferimento per il foglio di lavoro, il tuo codice sarebbe stato rotto.

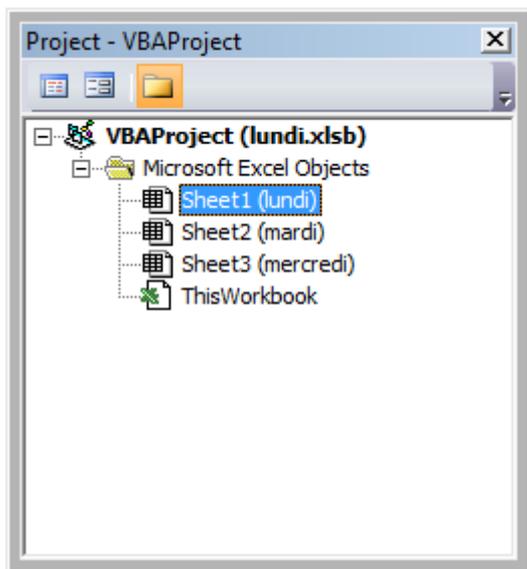
```
'reference worksheet by .Name
with worksheets("Monday")
    'operation code here; for example:
    .Range(.Cells(2, "A"), .Cells(.Rows.Count, "A").End(xlUp)) = 1
end with

'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
```

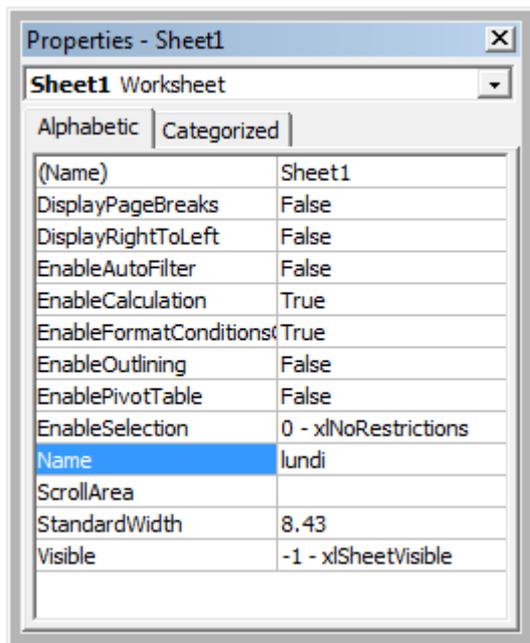
Sia l'ordine originale che il nome del foglio di lavoro originale sono stati compromessi. Tuttavia, se si fosse utilizzata la proprietà `.CodeName` del foglio di lavoro, la sottoprocedura sarebbe ancora operativa

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

L'immagine seguente mostra la finestra del progetto VBA ([Ctrl] + R) che elenca i fogli di lavoro di `.CodeName` e poi di `.Name` (tra parentesi). L'ordine in cui sono visualizzati non cambia; l'ordinale. L'indice viene preso dall'ordine in cui vengono visualizzati nella coda della scheda nome nella finestra del foglio di lavoro.



Mentre è raro rinominare un `.CodeName`, non è impossibile. Basta aprire la finestra delle proprietà di VBE ([F4]).



Il foglio di lavoro `.CodeName` si trova nella prima riga. Il foglio di lavoro. Nome è nel decimo. Entrambi sono modificabili.

Utilizzo di stringhe con delimitatori al posto di matrici dinamiche

L'utilizzo di matrici dinamiche in VBA può essere piuttosto complesso e richiedere tempo per set di dati molto grandi. Quando si memorizzano tipi di dati semplici in una matrice dinamica (stringhe, numeri, booleani ecc.), È possibile evitare le istruzioni `ReDim Preserve` richieste per gli array dinamici in VBA utilizzando la funzione `split()` con alcune procedure di stringa intelligenti. Ad esempio, esamineremo un ciclo che aggiunge una serie di valori da un intervallo a una stringa in base a determinate condizioni, quindi utilizza quella stringa per popolare i valori di un controllo `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
occurrence of the delimiter within the string
    End If
Next Count

ListBox1.List = Split(DataString, Delimiter)

End Sub
```

La stringa `Delimiter` stessa può essere impostata su qualsiasi valore, ma è prudente scegliere un valore che non si verificherà naturalmente all'interno dell'insieme. Supponi, ad esempio, di elaborare una colonna di date. In tal caso, usando `.`, `-`, o `/` sarebbe imprudente come delimitatore, in quanto le date potrebbero essere formattate per utilizzare uno qualsiasi di questi, generando più punti di dati di quanto previsto.

Nota: Esistono limitazioni all'utilizzo di questo metodo (ovvero la lunghezza massima delle stringhe), quindi dovrebbe essere usato con cautela in caso di set di dati molto grandi. Questo non è necessariamente il metodo più veloce o più efficace per la creazione di array dinamici in VBA, ma è un'alternativa valida.

Evento doppio clic per forme Excel

Per impostazione predefinita, le forme in Excel non hanno un modo specifico per gestire clic singoli o doppi, contenente solo la proprietà "OnAction" per consentire all'utente di gestire i clic. Tuttavia, potrebbero esserci casi in cui il codice richiede all'utente di agire in modo diverso (o esclusivo) con un doppio clic. La seguente subroutine può essere aggiunta al tuo progetto VBA e, se impostata come routine `OnAction` per la tua forma, ti consente di agire con doppio 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
```

Questa routine farà in modo che la forma ignori funzionalmente il primo clic, eseguendo solo il codice desiderato al secondo clic entro l'intervallo di tempo specificato.

Finestra di dialogo Apri file - Più file

Questa subroutine è un rapido esempio su come consentire a un utente di selezionare più file e quindi fare qualcosa con quei percorsi di file, come ad esempio ottenere i nomi dei file e inviarli alla console tramite `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

```

Leggi Suggerimenti e trucchi Excel VBA online: <https://riptutorial.com/it/excel-vba/topic/2240/suggerimenti-e-trucchi-excel-vba>

Capitolo 30: Tabelle pivot

Osservazioni

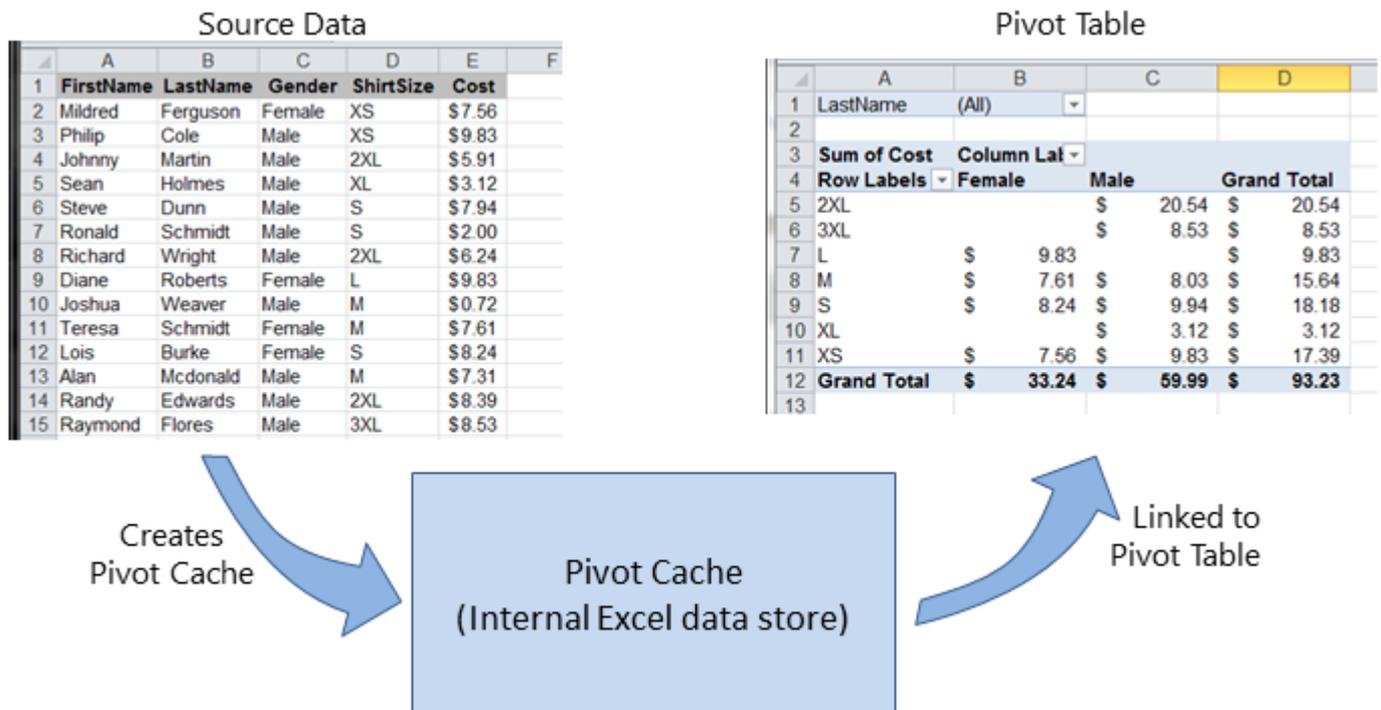
Ci sono molte fonti di riferimento e esempi eccellenti sul Web. Alcuni esempi e spiegazioni vengono creati qui come punto di raccolta per risposte rapide. Illustrazioni più dettagliate possono essere collegate a contenuti esterni (invece di copiare materiale originale esistente).

Examples

Creazione di una tabella pivot

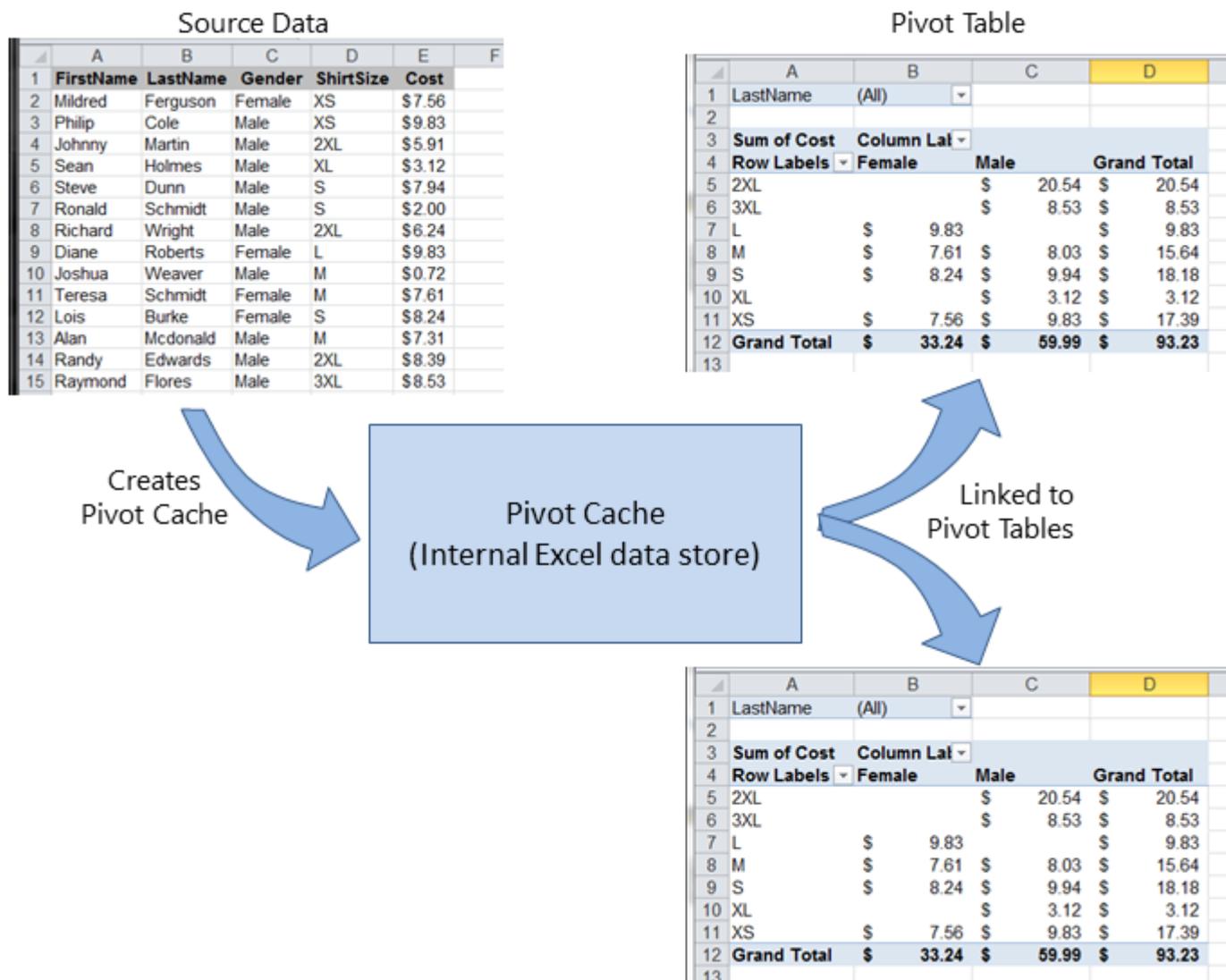
Una delle funzionalità più potenti di Excel è l'uso di tabelle pivot per ordinare e analizzare i dati. L'utilizzo di VBA per creare e manipolare i pivot è più semplice se si capisce la relazione tra tabelle pivot e cache pivot e come fare riferimento e utilizzare le diverse parti delle tabelle.

Nella sua forma più semplice, i dati di origine sono un'area di dati `Range` su un `Worksheet`. Questa area dati **DEVE** identificare le colonne di dati con una riga di intestazione come prima riga dell'intervallo. Una volta creata la tabella pivot, l'utente può visualizzare e modificare i dati di origine in qualsiasi momento. Tuttavia, le modifiche potrebbero non essere riflesse automaticamente o immediatamente nella tabella pivot stessa poiché esiste una struttura di archiviazione dei dati intermedia denominata Pivot Cache direttamente connessa alla tabella pivot stessa.



Se sono necessarie più tabelle pivot, in base agli stessi dati di origine, la Pivot Cache può essere riutilizzata come archivio dati interno per ciascuna tabella pivot. Questa è una buona pratica

perché salva la memoria e riduce la dimensione del file Excel per l'archiviazione.



Ad esempio, per creare una tabella pivot basata sui dati di origine mostrati nelle figure sopra:

```

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
```

Riferimenti [Oggetto tabella pivot MSDN](#)

Intervalli di tabella pivot

Queste eccellenti fonti di riferimento forniscono descrizioni e illustrazioni dei vari intervalli nelle tabelle pivot.

Riferimenti

- [Fare riferimento a intervalli di tabelle pivot in VBA](#) - dal blog tecnico di Jon Peltier
- [Riferimento a un intervallo di tabelle pivot di Excel mediante VBA](#) : da Globalconnect Excel VBA

Aggiunta di campi a una tabella pivot

Due cose importanti da notare quando si aggiungono campi a una tabella pivot sono Orientamento e Posizione. A volte uno sviluppatore può assumere dove viene inserito un campo, quindi è sempre più chiaro definire esplicitamente questi parametri. Queste azioni influiscono solo sulla tabella pivot fornita, non sulla Pivot Cache.

```
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
```

Formattazione dei dati della tabella pivot

Questo esempio modifica / imposta diversi formati nell'area di intervallo dati (`DataBodyRange`) della tabella pivot fornita. Sono disponibili tutti i parametri formattabili in una `Range` standard. La formattazione dei dati ha effetto solo sulla tabella pivot stessa, non sulla pivot cache.

NOTA: la proprietà è denominata `TableStyle2` perché la proprietà `TableStyle` non è un membro

delle proprietà dell'oggetto della `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
```

Leggi Tabelle pivot online: <https://riptutorial.com/it/excel-vba/topic/3797/tabelle-pivot>

Capitolo 31: Usa oggetto foglio di lavoro e non oggetto Foglio

introduzione

Un sacco di utenti VBA considerano sinonimi di oggetti Fogli di lavoro e Fogli. Non sono.

L'oggetto Fogli è costituito da entrambi i fogli di lavoro e i grafici. Pertanto, se disponiamo di grafici nella nostra cartella di lavoro di Excel, dovremmo fare attenzione, non utilizzare `Sheets` e `Worksheets` come sinonimi.

Examples

Stampa il nome del primo oggetto



```
Option Explicit

Sub CheckWorksheetsDiagram()

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

End Sub
```

Il risultato:

```
Sheet1
Chart1
Chart1
```

Leggi Usa oggetto foglio di lavoro e non oggetto Foglio online: <https://riptutorial.com/it/excel-vba/topic/9996/usa-oggetto-foglio-di-lavoro-e-non-oggetto-foglio>

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare 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	Array	Alon Adler , Hubisan , Miguel_Ryu , Shahin
3	autofilter; Usi e buone pratiche	Sgdva
4	Best practice 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
5	Celle / gamme unite	R3uK
6	Come registrare una macro	Mike , Robby
7	Creazione di un menu a discesa nel foglio di lavoro attivo con una casella combinata	Macro Man , quadrature , R3uK
8	CustomDocumentProperties in pratica	T.M.
9	Debug e risoluzione dei problemi	Cody G. , Etheur , Gregor y , Julian Kuchlbauer , Kyle , Malick , Michael Russo , RGA , Ron McMahon , Slai , Steven Schroeder , Taylor Ostberg
10	Dichiarazioni condizionali	SteveES
11	Errori comuni	Egan Wolf , Gordon Bell , Macro Man , Malick , Peh , SWa , Taylor Ostberg
12	File System Object	Zsmaster

13	Formattazione condizionale tramite VBA	Zsmaster
14	Funzioni definite dall'utente (UDF)	Jeeped , Malick , Slai , user3561813 , Vegard
15	Gamme e celle	Adam , Branislav Kollár , Doug Coats , Gregor y , Jbjstam , Joel Spolsky , Julian Kuchlbauer , Máté Juhász , Miguel_Ryu , Patrick Wynne , Vegard
16	Gamme nominate	Andre Terra , Portland Runner
17	Grafici e grafici	Byron Wall
18	Individuazione di valori duplicati in un intervallo	quadrature , T.M.
19	Integrazione di PowerPoint tramite VBA	mnoronha , RGA
20	Lavorare con le tabelle di Excel in VBA	Excel Developers
21	Le cartelle di lavoro	PeterT
22	Metodi per trovare l'ultima riga o colonna usata in un foglio di lavoro	curious , Hubisan , Máté Juhász , Michael Russo , Miqi180 , paul bica , R3uK , Raystafarian , RGA , Shai Rado , Slai , Thomas Inzina , YowE3K
23	Oggetto dell'applicazione	Captain Grumpy , Joel Spolsky
24	Ottimizzazione Excel-VBA	Masoud , paul bica , T.M.
25	Passa in rassegna tutti i fogli in Active Workbook	Doug Coats , Shai Rado
26	Rilegatura	Captain Grumpy , EEM , Jeeped , jlookup , Malick , Raystafarian
27	Sicurezza VBA	Chel , TheGuyThatDoesn'tKnowMuch
28	SQL in Excel VBA: best practice	Zsmaster
29	Suggerimenti e trucchi Excel VBA	Andre Terra , Cody G. , Jeeped , Kumar Sourav , Macro Man , RGA
30	Tabelle pivot	PeterT
31	Usa oggetto foglio di lavoro e non oggetto Foglio	Vityata