EBook Gratuito

APPRENDIMENTO VBA

Free unaffiliated eBook created from
Stack Overflow contributors.

#vba

Sommario

Di

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

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

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

Capitolo 1: Iniziare con VBA

Osservazioni

Questa sezione fornisce una panoramica su cosa sia Vba e perché uno sviluppatore potrebbe volerlo utilizzare.

Dovrebbe anche menzionare qualsiasi argomento di grandi dimensioni all'interno di vba e collegarsi agli argomenti correlati. Poiché la documentazione di vba è nuova, potrebbe essere necessario creare versioni iniziali di tali argomenti correlati.

Versioni

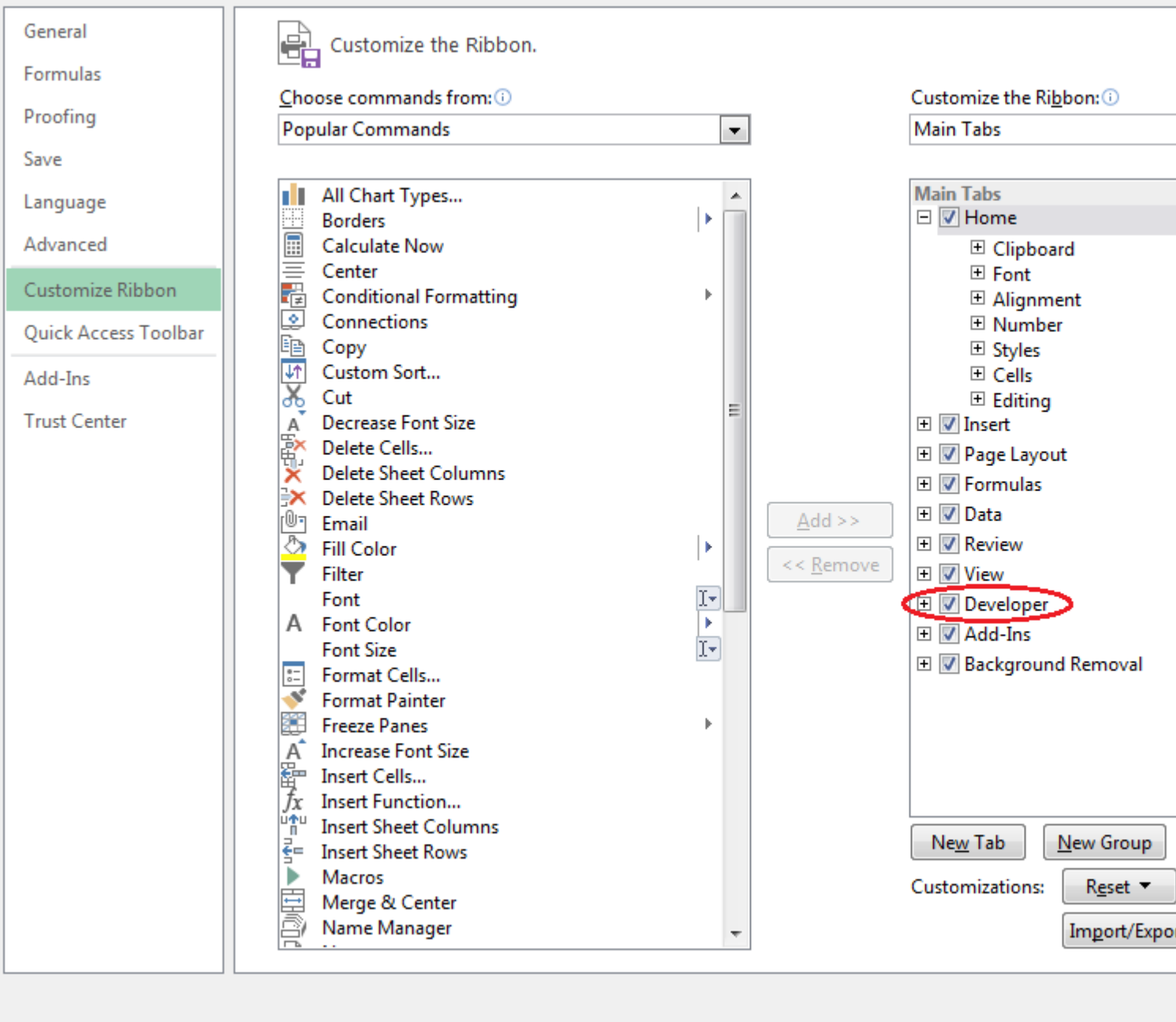
Versione	Versioni di Office	Data di rilascio Note	Data di rilascio
VBA6	? - 2007	[Qualche tempo dopo] [1]	1992/06/30
Vba7	2010 - 2016	[Blog.techkit.com] [2]	2010-04-15
VBA per Mac	2004, 2011 - 2016		2004-05-11

Examples

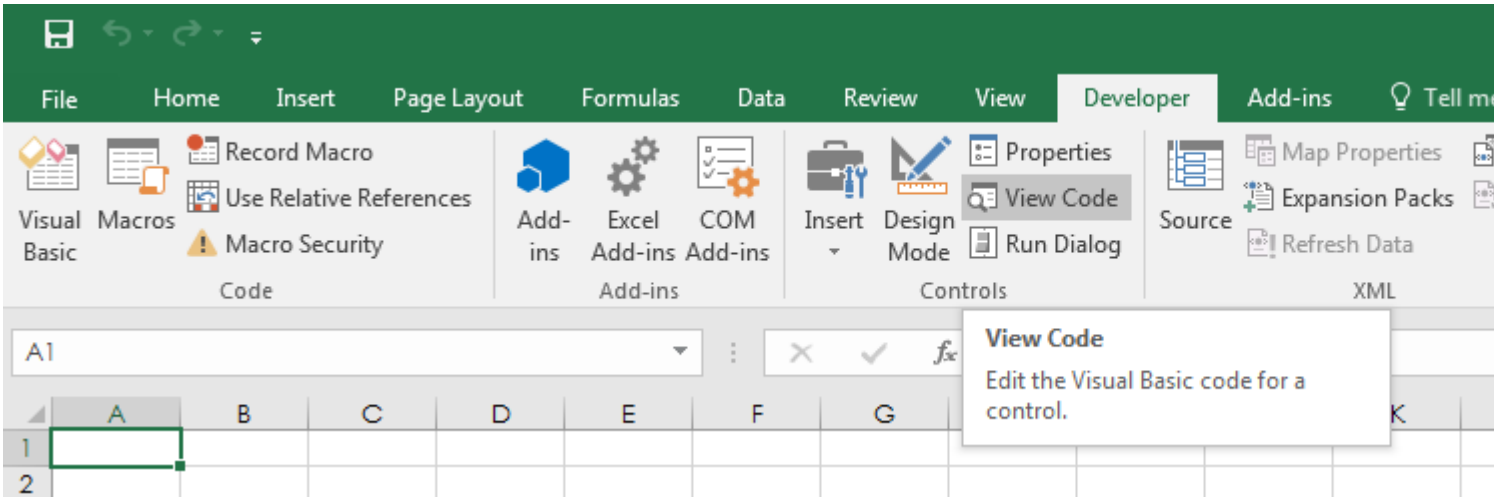
Accesso a Visual Basic Editor in Microsoft Office

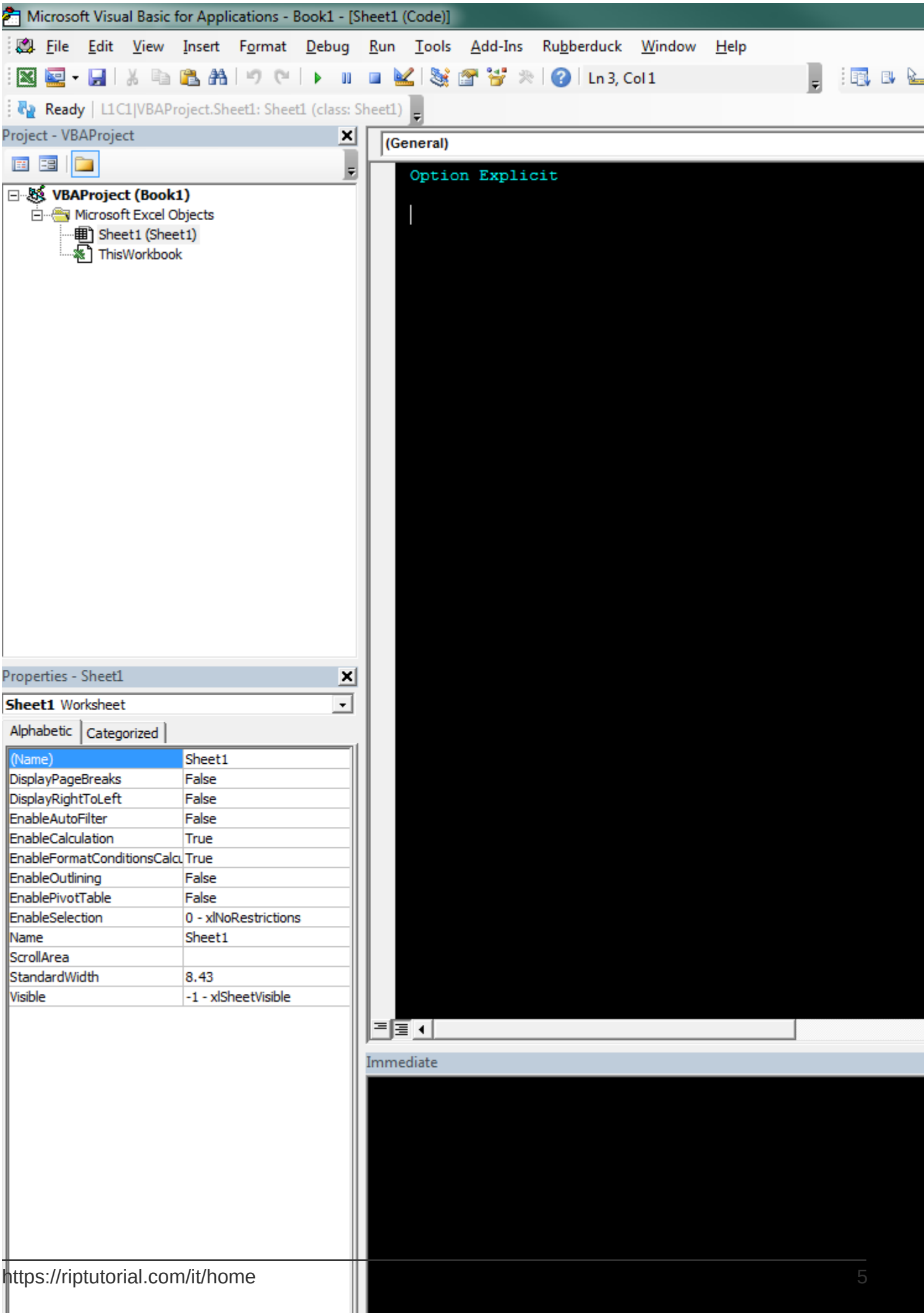
È possibile aprire l'editor VB in qualsiasi applicazione Microsoft Office premendo `Alt` + `F11` o andando alla scheda Sviluppatore e facendo clic sul pulsante "Visual Basic". Se non vedi la scheda Sviluppatore nella barra multifunzione, controlla se è abilitata.

Per impostazione predefinita, la scheda Sviluppatore è disabilitata. Per abilitare la scheda Sviluppatore vai su File -> Opzioni, seleziona Personalizza barra multifunzione nell'elenco a sinistra. A destra nella vista ad albero "Personalizza la barra multifunzione" trova la voce Albero sviluppatore e imposta il controllo per la casella di controllo Sviluppatore su selezionato. Fai clic su OK per chiudere la finestra di dialogo Opzioni.



La scheda Sviluppatore è ora visibile nella barra multifunzione su cui è possibile fare clic su "Visual Basic" per aprire Visual Basic Editor. In alternativa è possibile fare clic su "Visualizza codice" per visualizzare direttamente il riquadro di codice dell'elemento attualmente attivo, ad esempio Foglio di lavoro, Grafico, Forma.


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


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

nella barra degli strumenti o semplicemente premi il tasto `F5` . Congratulazioni! Hai creato il tuo primo modulo VBA.

Debug

Il debugging è un modo molto potente per avere uno sguardo più ravvicinato e correggere un codice errato (o non funzionante) funzionante.

Esegui il codice passo dopo passo

La prima cosa che devi fare durante il debug è fermare il codice in posizioni specifiche e poi eseguirlo riga per riga per vedere se ciò accade come previsto.

- Punto di interruzione (`F9` , Debug - Disattiva punto di interruzione): è possibile aggiungere un punto di interruzione a qualsiasi linea eseguita (ad esempio, non alle dichiarazioni), quando l'esecuzione raggiunge quel punto si arresta e fornisce il controllo all'utente.
- È inoltre possibile aggiungere la parola chiave `Stop` a una riga vuota per fare in modo che il codice si fermi in quella posizione in fase di esecuzione. Ciò è utile se, ad esempio, prima delle righe di dichiarazione a cui non è possibile aggiungere un punto di interruzione con `F9`
- Entra in (`F8` , Debug - Step into): esegue solo una riga di codice, se quella è una chiamata di una sub / funzione definita dall'utente, quindi viene eseguita riga per riga.
- Passaggio (`Maiusc` + `F8` , Debug - Passaggio sopra): esegue una riga di codice, non inserisce sottosistemi / funzioni definiti dall'utente.
- Esci (`Ctrl` + `Shift` + `F8` , Debug - Esci): Esci da sub / funzione corrente (esegui il codice fino alla fine).
- Corri al cursore (`Ctrl` + `F8` , Debug - Esegui al cursore): esegui il codice fino a raggiungere la linea con il cursore.
- È possibile utilizzare `Debug.Print` per stampare le righe sulla finestra immediata in fase di esecuzione. Puoi anche usare `Debug.?` come scorciatoia per `Debug.Print`

La finestra degli orologi

L'esecuzione del codice riga per riga è solo il primo passo, dobbiamo conoscere più dettagli e uno strumento per quella è la finestra di controllo (finestra Visualizza-Guarda), qui puoi vedere i valori delle espressioni definite. Per aggiungere una variabile alla finestra dell'orologio, puoi:

- Fai clic con il tasto destro del mouse e seleziona "Aggiungi orologio".
- Fai clic con il tasto destro del mouse nella finestra di visualizzazione, seleziona "Aggiungi orologio".
- Vai a Debug - Aggiungi orologio.

Quando aggiungi una nuova espressione puoi scegliere se vuoi solo vedere il suo valore, o anche interrompere l'esecuzione del codice quando è vero o quando cambia il suo valore.

Finestra immediata

La finestra immediata consente di eseguire codice arbitrario o stampare elementi precedendoli con la parola chiave `Print` o con un singolo punto interrogativo " `?` "

Qualche esempio:

- `? ActiveSheet.Name` : restituisce il nome del foglio attivo
- `Print ActiveSheet.Name` : restituisce il nome del foglio attivo
- `? foo` - restituisce il valore di `foo` *
- `x = 10` set `x` a 10 *

* Ottenere / Impostare valori per variabili tramite la Finestra Immediata può essere fatto solo durante il runtime

Debug delle migliori pratiche

Ogni volta che il tuo codice non funziona come dovrebbe, la prima cosa che dovresti fare è leggerlo di nuovo con attenzione, cercando gli errori.

Se ciò non aiuta, quindi avviare il debug; per le procedure brevi può essere efficiente eseguirlo riga per riga, per quelli più lunghi è probabilmente necessario impostare punti di interruzione o interruzioni sulle espressioni guardate, l'obiettivo qui è trovare la linea che non funziona come previsto.

Una volta ottenuta la linea che fornisce il risultato errato, ma il motivo non è ancora chiaro, provare a semplificare le espressioni o sostituire le variabili con costanti, in modo da capire se il valore delle variabili è sbagliato.

Se ancora non riesci a risolverlo e chiedi aiuto:

- Includi come possibile parte del tuo codice per capire il tuo problema
- Se il problema non è correlato al valore delle variabili, sostituirle con costanti. (così, invece di `Sheets(a*b*c+d^2).Range(addressOfRange)` write `Sheets(4).Range("A2")`)
- Descrivi quale linea dà il comportamento sbagliato e che cos'è (errore, risultato errato ...)

Leggi Iniziare con VBA online: https://riptutorial.com/it/vba/topic/802/iniziare-con-vba

Capitolo 2: Array

Examples

Dichiarazione di una matrice in VBA

Dichiarare una matrice è molto simile alla dichiarazione di una variabile, tranne che è necessario dichiarare la dimensione della matrice subito dopo il suo nome:

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

Per impostazione predefinita, gli array in VBA sono **indicizzati da ZERO** , quindi il numero all'interno della parentesi non si riferisce alla dimensione dell'array, ma piuttosto **all'indice dell'ultimo elemento**

Accesso agli elementi

L'accesso a un elemento dell'array viene effettuato utilizzando il nome dell'array, seguito dall'indice dell'elemento, tra parentesi:

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

Indicizzazione delle matrici

Puoi modificare l'indicizzazione degli array posizionando questa riga nella parte superiore di un modulo:

```
Option Base 1
```

Con questa linea, tutti gli array dichiarati nel modulo verranno **indicizzati da ONE** .

Indice specifico

È inoltre possibile dichiarare ciascuna matrice con il proprio indice utilizzando la parola chiave `To` e il limite inferiore e superiore (= indice):

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

Dichiarazione Dinamica

Quando non si conosce la dimensione della matrice prima della sua dichiarazione, è possibile utilizzare la dichiarazione dinamica e la parola chiave `ReDim` :

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

Nota che l'uso della parola chiave `ReDim` cancellerà qualsiasi contenuto precedente della tua matrice. Per evitare ciò, è possibile utilizzare la parola chiave `Preserve` dopo `ReDim` :

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

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

Uso di Split per creare una matrice da una stringa

Funzione Split

restituisce una matrice monodimensionale a base zero contenente un numero specificato di sottostringhe.

Sintassi

Dividi (espressione [, **delimitatore** [, **limite** [, **confronta**]]] **)**

Parte	Descrizione
espressione	Necessario. Espressione di stringhe contenente sottostringhe e delimitatori. Se *expression* è una stringa di lunghezza zero ("" o vbNullString), **Split** restituisce un array vuoto che non contiene elementi e nessun dato. In questo caso, l'array restituito avrà un LBound di 0 e un UBound di -1.
delimitatore	Opzionale. Carattere stringa utilizzato per identificare i limiti della sottostringa. Se omesso, si assume che il carattere dello spazio ("") sia il delimitatore. Se il ***delimitatore*** è una stringa di lunghezza zero, viene restituito un array a elemento singolo contenente l'intera stringa di ***espressione*** .
limite	Opzionale. Numero di sottostringhe da restituire; -1 indica che tutte le sottostringhe vengono restituite.
confrontare	Opzionale. Valore numerico che indica il tipo di confronto da utilizzare quando si valutano le sottostringhe. Vedi la sezione Impostazioni per i valori.

impostazioni

L'argomento di ***confronto*** può avere i seguenti valori:

Costante	Valore	Descrizione
Descrizione	-1	Esegue un confronto utilizzando l'impostazione dell'istruzione **Option Compare** .
vbBinaryCompare	0	Esegue un confronto binario.
vbTextCompare	1	Esegue un confronto testuale.
vbDatabaseCompare	2	Solo Microsoft Access. Esegue un confronto in base alle informazioni nel database.

Esempio

In questo esempio è dimostrato come funziona Split mostrando diversi stili. I commenti mostreranno il set di risultati per ciascuna delle diverse opzioni di divisione eseguite. Infine, viene dimostrato come eseguire il looping sull'array di stringhe restituito.

```
Sub Test

    Dim textArray() as String

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

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

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

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

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

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

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

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

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

Iterazione di elementi di un array

Per il prossimo

L'uso della variabile iteratore come numero indice è il modo più veloce per iterare gli elementi di un array:

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

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

I loop annidati possono essere utilizzati per iterare gli array multidimensionali:

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

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

Per ogni ... Avanti

Un ciclo `For Each...Next` può anche essere utilizzato per iterare gli array, se le prestazioni non contano:

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

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

A `For Each` ciclo, itererà tutte le dimensioni da esterno a interno (lo stesso ordine con cui gli elementi sono disposti in memoria), quindi non c'è bisogno di cicli annidati:

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

Dim item As Variant 'must be Variant
```

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

Si noti che `For Each` ciclo I cicli vengono utilizzati al meglio per iterare gli oggetti `Collection` , se le prestazioni sono importanti.

Tutti e 4 i frammenti sopra producono lo stesso risultato:

```
 0
 1
 2
 3
```

Array dinamici (ridimensionamento della matrice e gestione dinamica)

Matrici dinamiche

Aggiungere e ridurre variabili su un array in modo dinamico è un enorme vantaggio per quando le informazioni che si stanno trattando non hanno un determinato numero di variabili.

Aggiunta di valori in modo dinamico

Puoi semplicemente ridimensionare l'array con l'istruzione `ReDim` , questo ridimensionerà l'array ma se tu che conservi le informazioni già memorizzate nell'array avrai bisogno della parte `Preserve` .

Nell'esempio seguente creiamo una matrice e la aumentiamo di un'altra variabile in ogni iterazione, preservando i valori già presenti nella matrice.

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

For n = 1 To 100

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

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

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

```
Next
```

Rimozione dei valori in modo dinamico

Possiamo utilizzare la stessa logica per diminuire la matrice. Nell'esempio il valore "last" sarà rimosso dall'array.

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

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

Ripristino di una matrice e riutilizzo dinamico

Possiamo anche riutilizzare gli array che creiamo per non avere molti in memoria, il che rallenterebbe il tempo di esecuzione. Questo è utile per gli array di varie dimensioni. Uno snippet che è possibile utilizzare per riutilizzare l'array è `ReDim` l'array a `(0)` , attribuire una variabile all'array e aumentare nuovamente l'array.

Nello snippet seguente costruisco un array con i valori da 1 a 40, svuota l'array e riempi l'array con valori da 40 a 100, il tutto fatto dinamicamente.

```
Dim Dynamic_array As Variant

For n = 1 To 100

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

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

Next
```

Array frastagliati (array di array)

Matrici frastagliate senza matrici multidimensionali

Le matrici di array (matrici frastagliate) non sono le stesse di matrici multidimensionali se si pensa

a esse visivamente le matrici multidimensionali apparirebbero come matrici (rettangolari) con un numero definito di elementi sulle loro dimensioni (all'interno di matrici), mentre l'array frastagliato sarebbe come un anno calendario con gli array interni che hanno un numero diverso di elementi, come i giorni in diversi mesi.

Sebbene gli Jagged Array siano piuttosto disordinati e difficili da usare a causa dei livelli nidificati e non hanno molta sicurezza di tipo, ma sono molto flessibili, ti permettono di manipolare diversi tipi di dati abbastanza facilmente, e non c'è bisogno di contenere inutilizzati o elementi vuoti.

Creazione di una matrice seghettata

Nell'esempio seguente inizializzeremo un array frastagliato contenente due array uno per Names e un altro per Numbers, e quindi accedendo a un elemento di ciascun

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

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

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

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

Creare e leggere dinamicamente array frastagliati

Possiamo anche essere più dinamici nella nostra approssimazione per costruire gli array, immaginare di avere una scheda tecnica del cliente in Excel e vogliamo costruire una matrice per produrre i dettagli del cliente.

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

Costruiremo dinamicamente un array Header e un array Customers, l'intestazione conterrà i titoli delle colonne e l'array Customers conterrà le informazioni di ogni cliente / riga come array.

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

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

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

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

            'build array containing customer values
            If IsEmpty(Customer_Values) Then
                ReDim Customer_Values(0)
                Customer_Values(0) = Cells(r, c).Value
            ElseIf Customer_Values(0) = "" Then
                Customer_Values(0) = Cells(r, c).Value
            Else
                ReDim Preserve Customer_Values(0 To UBound(Customer_Values) + 1)
                Customer_Values(UBound(Customer_Values)) = Cells(r, c).Value
            End If
        Next

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

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

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

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

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

Il risultato del frammento di cui sopra è un Jagged Array con due array uno di quegli array con 4 elementi, 2 livelli di indention, e l'altro essendo esso stesso un altro Jagged Array contenente 5 array di 4 elementi ciascuno e 3 livelli di indention, vedi sotto la struttura:

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

Per accedere alle informazioni devi tenere a mente la struttura del Jagged Array che crei, nell'esempio sopra puoi vedere che l' `Main Array` contiene una Array di `Headers` e una Matrice di Array (`Customers`) quindi con diversi modi di accedere agli elementi.

Ora leggeremo le informazioni del `Main Array` e stamperemo ciascuna informazione del cliente come `Info Type: Info` .

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

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

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

RICORDA di tenere traccia della struttura del tuo Jagged Array, nell'esempio sopra per accedere al Nome di un cliente è accedendo a `Main_Array -> Customers -> CustomerNumber -> Name` che è di tre livelli, per restituire `"Person4"` cui avrai bisogno la posizione dei clienti nell'array principale, quindi la posizione del cliente quattro nell'array Jagged dei clienti e infine la posizione dell'elemento necessario, in questo caso `Main_Array(1)(3)(0)` che è `Main_Array(Customers)(CustomerNumber)(Name)` .

Array multidimensionali

Array multidimensionali

Come indica il nome, gli array multidimensionali sono matrici che contengono più di una dimensione, di solito due o tre, ma possono avere fino a 32 dimensioni.

Un multi array funziona come una matrice con vari livelli, prendi ad esempio un confronto tra una, due e tre dimensioni.

Una dimensione è la tipica matrice, sembra una lista di elementi.

```
Dim 1D(3) as Variant

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

Due dimensioni apparirebbero come una griglia Sudoku o un foglio Excel, quando inizializzando l'array si definirebbe quante righe e colonne avrebbe la matrice.

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

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

Tre dimensioni comincerebbero ad assomigliare a Cubo di Rubik, quando inizializzando l'array si definirebbero righe e colonne e livelli / profondità che l'array avrebbe.

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

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

Ulteriori dimensioni potrebbero essere pensate come la moltiplicazione del 3D, quindi un 4D (1,3,3,3) sarebbe costituito da due array 3D affiancati.

Matrice a due dimensioni

Creazione

L'esempio seguente sarà una raccolta di un elenco di dipendenti, ciascun dipendente avrà una serie di informazioni sulla lista (Nome, Cognome, Indirizzo, Email, Telefono ...), l'esempio sarà essenzialmente memorizzato sull'array (dipendente, informazione) essendo il (0,0) è il primo nome del primo dipendente.

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

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

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

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

```
employee and 2 = information
    For information_e = 0 To UBound(Employees, 2)
    'for each information element/column in the array

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

Ridimensionamento

Ridimensionare o `ReDim Preserve` un Multi-Array come la norma per un array a una dimensione otterrebbe un errore, invece le informazioni devono essere trasferite in un array temporaneo con le stesse dimensioni dell'originale più il numero di righe / colonne da aggiungere. Nell'esempio seguente vedremo come inizializzare un array di temp, trasferire le informazioni dall'array originale, riempire gli elementi vuoti rimanenti e sostituire l'array temp con l'array originale.

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

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

    Next
Next

'fill remaining
'after the transfers the Temp array still has unused elements at the end, being that it was
increased
'to fill the remaining elements iterate from the last "row" with values to the last row in the
array
'in this case the last row in Temp will be the size of the Employees array rows + 1, as the
last row of Employees array is already filled in the TempArray

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

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

    Next
Next

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

Modifica dei valori degli elementi

Per cambiare / modificare i valori in un determinato elemento si può fare semplicemente chiamando la coordinata per cambiarla e assegnandole un nuovo valore: `Employees(0, 0) = "NewValue"`

In alternativa, scorrere le coordinate per utilizzare le condizioni per abbinare i valori corrispondenti ai parametri necessari:

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

Lettura

L'accesso agli elementi dell'array può essere fatto con un ciclo annidato (iterando ogni elemento), loop e coordinate (iterare le righe e accedere direttamente alle colonne) o accedere direttamente con entrambe le coordinate.

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

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

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

***Ricorda** , è sempre utile mantenere una mappa di array quando si usano array multidimensionali, possono facilmente diventare confusione.*

Matrice a tre dimensioni

Per l'array 3D, utilizzeremo la stessa premessa dell'array 2D, con l'aggiunta non solo di

memorizzazione di Employee e Information ma anche di Building in cui lavorano.

L'array 3D avrà i Dipendenti (possono essere considerati come Righe), le Informazioni (Colonne) e Edificio che possono essere pensati come fogli diversi su un documento Excel, hanno le stesse dimensioni tra di loro, ma ogni foglio ha un diverso insieme di informazioni nelle sue cellule / elementi. L'array 3D conterrà ***n*** numero di array 2D.

Creazione

Un array 3D necessita di 3 coordinate da inizializzare. `Dim 3Darray(2,5,5) As Variant` la prima coordinata dell'array sarà il numero di Building / Sheets (diversi gruppi di righe e colonne), la seconda coordinata definirà Rows e il terzo colonne. La `Dim` sopra si tradurrà in un array 3D con 108 elementi (`3*6*6`), con 3 diversi set di array 2D.

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

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

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

Ridimensionamento

Il ridimensionamento di un array 3D è simile al ridimensionamento di un 2D, in primo luogo creare un array temporaneo con le stesse dimensioni dell'originale aggiungendo uno nella coordinata del parametro per aumentare, la prima coordinata aumenterà il numero di set nell'array, il secondo e le terze coordinate aumenteranno il numero di righe o colonne in ciascun set.

L'esempio seguente aumenta il numero di Righe in ciascun set di uno e riempie gli elementi aggiunti di recente con nuove informazioni.

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

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

        Next
    Next
Next

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

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

        Next
    Next
Next

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

Modifica dei valori degli elementi e lettura

Leggere e modificare gli elementi sull'array 3D può essere fatto in modo simile al modo in cui facciamo l'array 2D, basta regolarlo per il livello extra nei loop e nelle coordinate.

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

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

```
Next

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

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

Leggi Array online: https://riptutorial.com/it/vba/topic/3064/array

Capitolo 3: Assegnazione di stringhe con caratteri ripetuti

Osservazioni

A volte è necessario assegnare una variabile di stringa con un carattere specifico ripetuto un numero specifico di volte. VBA offre due funzioni principali per questo scopo:

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

Examples

Usa la funzione String per assegnare una stringa con n caratteri ripetuti

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

Utilizzare le funzioni String e Spazio per assegnare una stringa di caratteri n

```
Dim stringOfSpaces As String

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

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

Leggi Assegnazione di stringhe con caratteri ripetuti online:
https://riptutorial.com/it/vba/topic/3581/assegnazione-di-stringhe-con-caratteri-ripetuti

Capitolo 4: attributi

Sintassi

- Attributo VB_Name = "ClassOrModuleName"
- Attributo VB_GlobalNameSpace = False 'Ignorato
- Attributo VB_Creatable = False 'Ignorato
- Attributo VB_PredeclaredId = {True | false}
- Attributo VB_Exposed = {True | false}
- Attribute variableName.VB_VarUserMemId = 0 'Zero indica che questo è il membro predefinito della classe.
- Attribute variableName.VB_VarDescription = "some string" "Aggiunge il testo alle informazioni del Visualizzatore oggetti per questa variabile.
- Attributo procName.VB_Description = "some string" "Aggiunge il testo alle informazioni del Visualizzatore oggetti per la procedura.
- Attributo procName.VB_UserMemId = {0 | -4}
    - '0: rende la funzione il membro predefinito della classe.
    - '-4: specifica che la funzione restituisce un enumeratore.

Examples

VB_Name

VB_Name specifica il nome della classe o del modulo.

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

Una nuova istanza di questa classe verrebbe creata con

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

VB_GlobalNameSpace

In VBA, questo attributo è ignorato. Non è stato trasferito da VB6.

In VB6, crea un'istanza globale predefinita della classe (una "scorciatoia") in modo che i membri della classe possano accedere senza utilizzare il nome della classe. Ad esempio, `DateTime` (come in `DateTime.Now`) è in realtà parte della classe `VBA.Conversion` .

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

VB_Createable

Questo attributo è ignorato. Non è stato trasferito da VB6.

In VB6, è stato utilizzato in combinazione con l'attributo `VB_Exposed` per controllare l'accessibilità delle classi esterne al progetto corrente.

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

Ne risulterebbe una `Public Class` , a cui si potrebbe accedere da altri progetti, ma questa funzionalità non esiste in VBA.

VB_PredeclaredId

Crea un'istanza di default globale di una classe. L'istanza predefinita è accessibile tramite il nome della classe.

Dichiarazione

```
VERSION 1.0 CLASS
BEGIN
  MultiUse = -1  'True
END
Attribute VB_Name = "Class1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Option Explicit

Public Function GiveMeATwo() As Integer
    GiveMeATwo = 2
End Function
```

Chiamata

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

In qualche modo, simula il comportamento delle classi statiche in altre lingue, ma a differenza di altre lingue, puoi comunque creare un'istanza della classe.

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

VB_Exposed

Controlla le caratteristiche istanziate di una classe.

```
Attribute VB_Exposed = False
```

Rende la classe `Private` . Non è possibile accedere al di fuori del progetto corrente.

```
Attribute VB_Exposed = True
```

Espone la classe `Public` , al di fuori del progetto. Tuttavia, poiché `VB_Createable` viene ignorato in VBA, le istanze della classe non possono essere create direttamente. Questo è equivalente a una seguente classe VB.Net.

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

Per ottenere un'istanza dall'esterno del progetto, è necessario esporre una factory per creare istanze. Un modo per farlo è con un normale modulo `Public` .

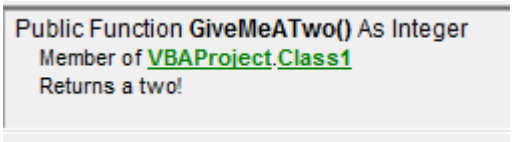
```
Public Function CreateFoo() As Foo
    CreateFoo = New Foo
End Function
```

Dato che i moduli pubblici sono accessibili da altri progetti, questo ci consente di creare nuove istanze delle nostre classi `Public - Not Createable` .

VB_Description

Aggiunge una descrizione testuale a un membro di classe o modulo che diventa visibile nell'Explorer oggetti. Idealmente, tutti i membri pubblici di un'interfaccia pubblica / API dovrebbero avere una descrizione.

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



Nota: tutti i membri di accesso di una proprietà (`Get` , `Let` , `Set`) utilizzano la stessa descrizione.

VB_ [Var] UserMemId

`VB_VarUserMemId` (per variabili modulo-scope) e `VB_UserMemId` (per procedure) gli attributi sono usati in VBA principalmente per due cose.

Specifica del membro predefinito di una

classe

Una classe `List` che incapsulerebbe una `Collection` vorrebbe avere una proprietà `Item` , quindi il codice client può fare ciò:

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

Ma con un attributo `VB_UserMemId` impostato su 0 nella proprietà `Item` , il codice client può fare ciò:

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

Solo un membro può avere `VB_UserMemId = 0` legalmente in qualsiasi classe data. Per le proprietà, specificare l'attributo nella `Get` accesso `Get` :

```
Option Explicit
Private internal As New Collection

Public Property Get Count() As Long
    Count = internal.Count
End Property

Public Property Get Item(ByVal index As Long) As Variant
Attribute Item.VB_Description = "Gets or sets the element at the specified index."
Attribute Item.VB_UserMemId = 0
'Gets the element at the specified index.
    Item = internal(index)
End Property

Public Property Let Item(ByVal index As Long, ByVal value As Variant)
'Sets the element at the specified index.
    With internal
        If index = .Count + 1 Then
            .Add item:=value
        ElseIf index = .Count Then
            .Remove index
            .Add item:=value
        ElseIf index < .Count Then
            .Remove index
            .Add item:=value, before:=index
        End If
    End With
End Property
```

Rendere una classe iterabile con un costrutto `For Each` loop

Con il valore magico `-4` , l'attributo `VB_UserMemId` indica a VBA che questo membro produce un enumeratore - che consente al codice client di eseguire ciò:

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

Il modo più semplice per implementare questo metodo è chiamando il getter di proprietà nascosto `[_NewEnum]` su una `Collection` interna / incapsulata; l'identificatore deve essere racchiuso tra parentesi quadre a causa del carattere di sottolineatura principale che lo rende un identificatore VBA illegale:

```
Public Property Get NewEnum() As IUnknown
Attribute NewEnum.VB_Description = "Gets an enumerator that iterates through the List."
Attribute NewEnum.VB_UserMemId = -4
Attribute NewEnum.VB_MemberFlags = "40" 'would hide the member in VB6. not supported in VBA.
'Gets an enumerator that iterates through the List.
    Set NewEnum = internal.[_NewEnum]
End Property
```

Leggi attributi online: https://riptutorial.com/it/vba/topic/5321/attributi

Capitolo 5: Automazione o utilizzo di altre applicazioni Librerie

introduzione

Se si utilizzano gli oggetti in altre applicazioni come parte dell'applicazione Visual Basic, è possibile che si desideri stabilire un riferimento alle librerie di oggetti di tali applicazioni. Questa documentazione fornisce un elenco, fonti ed esempi su come utilizzare le librerie di diversi software, come Windows Shell, Internet Explorer, XML HttpRequest e altri.

Sintassi

- expression.CreateObject (ObjectName)
- espressione; Necessario. Un'espressione che restituisce un oggetto Application.
- ObjectName; Stringa richiesta. Il nome della classe dell'oggetto da creare. Per informazioni sui nomi di classi validi, vedere Identificatori programmatici OLE.

Osservazioni

- Automazione di MSDN-Understanding

  Quando un'applicazione supporta l'automazione, gli oggetti a cui espone l'applicazione sono accessibili da Visual Basic. Utilizzare Visual Basic per manipolare questi oggetti richiamando metodi sull'oggetto o ottenendo e impostando le proprietà dell'oggetto.

- MSDN-Check o Aggiungi un riferimento alla libreria degli oggetti

  Se si utilizzano gli oggetti in altre applicazioni come parte dell'applicazione Visual Basic, è possibile che si desideri stabilire un riferimento alle librerie di oggetti di tali applicazioni. Prima di poterlo fare, devi prima essere sicuro che l'applicazione fornisca una libreria di oggetti.

- Finestra di dialogo MSDN-References

  Consente di selezionare gli oggetti di un'altra applicazione che si desidera rendere disponibili nel codice impostando un riferimento alla libreria di oggetti dell'applicazione.

- Metodo MSDN-CreateObject

  Crea un oggetto di automazione della classe specificata. Se l'applicazione è già in esecuzione, CreateObject creerà una nuova istanza.

Examples

VBScript Regular Expressions

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

Strumenti> Riferimenti> Microsoft VBScript Regular Expressions #. #
DLL associata: VBScript.dll
Fonte: Internet Explorer 1.0 e 5.5

- MSDN-Microsoft Beefs Up VBScript con espressioni regolari
- Sintassi delle espressioni regolari MSDN (Scripting)
- scambio di esperti: utilizzo delle espressioni regolari in Visual Basic, Applications Edition e Visual Basic 6
- Come utilizzare Regular Expressions (Regex) in Microsoft Excel sia in-cell che loop su SO.
- regular-expressions.info/vbscript
- regular-expressions.info/vbscriptexample
- WIKI: espressione regolare

Codice

È possibile utilizzare queste funzioni per ottenere risultati RegEx, concatenare tutte le corrispondenze (se più di 1) in 1 stringa e visualizzare i risultati nella cella excel.

```
Public Function getRegExResult(ByVal SourceString As String, Optional ByVal RegExPattern As
String = "\d+", _
    Optional ByVal isGlobalSearch As Boolean = True, Optional ByVal isCaseSensitive As Boolean
= False, Optional ByVal Delimiter As String = ";") As String

    Static RegExObject As Object
    If RegExObject Is Nothing Then
        Set RegExObject = createVBScriptRegExObject
    End If

    getRegExResult = removeLeadingDelimiter(concatObjectItems(getRegExMatches(RegExObject,
SourceString, RegExPattern, isGlobalSearch, isCaseSensitive), Delimiter), Delimiter)

End Function

Private Function getRegExMatches(ByRef RegExObj As Object, _
    ByVal SourceString As String, ByVal RegExPattern As String, ByVal isGlobalSearch As
Boolean, ByVal isCaseSensitive As Boolean) As Object

    With RegExObj
        .Global = isGlobalSearch
        .IgnoreCase = Not (isCaseSensitive) 'it is more user friendly to use positive meaning
of argument, like isCaseSensitive, than to use negative IgnoreCase
        .Pattern = RegExPattern
        Set getRegExMatches = .Execute(SourceString)
    End With

End Function

Private Function concatObjectItems(ByRef Obj As Object, Optional ByVal DelimiterCustom As
String = ";") As String
```

```
    Dim ObjElement As Variant
    For Each ObjElement In Obj
        concatObjectItems = concatObjectItems & DelimiterCustom & ObjElement.Value
    Next
End Function

Public Function removeLeadingDelimiter(ByVal SourceString As String, ByVal Delimiter As
String) As String
    If Left$(SourceString, Len(Delimiter)) = Delimiter Then
        removeLeadingDelimiter = Mid$(SourceString, Len(Delimiter) + 1)
    End If
End Function

Private Function createVBScriptRegExObject() As Object
    Set createVBScriptRegExObject = CreateObject("vbscript.RegExp") 'ex.:
createVBScriptRegExObject.Pattern
End Function
```

Scripting File System Object

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

Strumenti> Riferimenti> Runtime di Microsoft Scripting
DLL associata: ScrRun.dll
Fonte: sistema operativo Windows

MSDN-Accesso ai file con FileSystemObject

> Il modello File System Object (FSO) fornisce uno strumento basato su oggetti per lavorare con cartelle e file. Consente di utilizzare la nota sintassi object.method con un ricco set di proprietà, metodi ed eventi per elaborare cartelle e file. È anche possibile utilizzare le istruzioni e i comandi Visual Basic tradizionali.
>
> Il modello FSO offre alla tua applicazione la possibilità di creare, modificare, spostare ed eliminare cartelle o per determinare se e dove esistono cartelle particolari. Consente inoltre di ottenere informazioni sulle cartelle, come i loro nomi e la data in cui sono state create o modificate per ultime.

Argomenti MSDN-FileSystemObject : " *... spiega il concetto di FileSystemObject e come usarlo.* "
Exceltrick-FileSystemObject in VBA - Explained
Scripting.FileSystemObject

Oggetto del dizionario di scripting

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

Strumenti> Riferimenti> Runtime di Microsoft Scripting
DLL associata: ScrRun.dll
Fonte: sistema operativo Windows

Scripting. Oggetto letterario

MSDN-Dictionary Object

Internet Explorer Object

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

Strumenti> Riferimenti> Microsoft Internet Controls
DLL associata: ieframe.dll
Fonte: browser Internet Explorer

Oggetto MSDN-InternetExplorer

> Controlla un'istanza di Windows Internet Explorer tramite l'automazione.

Internet Explorer Objec Basic Members

Il codice seguente dovrebbe illustrare come funziona l'oggetto IE e come manipolarlo tramite VBA. Vi consiglio di superarlo, altrimenti potrebbe verificarsi un errore durante più navigazioni.

```
Sub IEGetToKnow()
    Dim IE As InternetExplorer 'Reference to Microsoft Internet Controls
    Set IE = New InternetExplorer

    With IE
        .Visible = True 'Sets or gets a value that indicates whether the object is visible or
hidden.

        'Navigation
        .Navigate2 "http://www.example.com" 'Navigates the browser to a location that might
not be expressed as a URL, such as a PIDL for an entity in the Windows Shell namespace.
        Debug.Print .Busy 'Gets a value that indicates whether the object is engaged in a
navigation or downloading operation.
        Debug.Print .ReadyState 'Gets the ready state of the object.
        .Navigate2 "http://www.example.com/2"
        .GoBack 'Navigates backward one item in the history list
        .GoForward 'Navigates forward one item in the history list.
        .GoHome 'Navigates to the current home or start page.
        .Stop 'Cancels a pending navigation or download, and stops dynamic page elements, such
as background sounds and animations.
        .Refresh 'Reloads the file that is currently displayed in the object.

        Debug.Print .Silent 'Sets or gets a value that indicates whether the object can
display dialog boxes.
        Debug.Print .Type 'Gets the user type name of the contained document object.

        Debug.Print .Top 'Sets or gets the coordinate of the top edge of the object.
        Debug.Print .Left 'Sets or gets the coordinate of the left edge of the object.
        Debug.Print .Height 'Sets or gets the height of the object.
        Debug.Print .Width 'Sets or gets the width of the object.
    End With

    IE.Quit 'close the application window
End Sub
```

Raschiatura del web

La cosa più comune da fare con IE è di carpire alcune informazioni di un sito Web, o compilare un modulo di sito Web e inviare informazioni. Vedremo come farlo.

Prendiamo in considerazione il codice sorgente di *esempio.com* :

```
<!doctype html>
<html>
    <head>
        <title>Example Domain</title>
        <meta charset="utf-8" />
        <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1" />
        <style ... </style>
    </head>

    <body>
        <div>
            <h1>Example Domain</h1>
            <p>This domain is established to be used for illustrative examples in documents.
You may use this
            domain in examples without prior coordination or asking for permission.</p>
            <p><a href="http://www.iana.org/domains/example">More information...</a></p>
        </div>
    </body>
</html>
```

Possiamo usare il codice come sotto per ottenere e impostare le informazioni:

```
Sub IEWebScrape1()
    Dim IE As InternetExplorer 'Reference to Microsoft Internet Controls
    Set IE = New InternetExplorer

    With IE
        .Visible = True
        .Navigate2 "http://www.example.com"

        'we add a loop to be sure the website is loaded and ready.
        'Does not work consistently. Cannot be relied upon.
        Do While .Busy = True Or .ReadyState <> READYSTATE_COMPLETE 'Equivalent = .ReadyState
<> 4
            ' DoEvents - worth considering. Know implications before you use it.
            Application.Wait (Now + TimeValue("00:00:01")) 'Wait 1 second, then check again.
        Loop

        'Print info in immediate window
        With .Document 'the source code HTML "below" the displayed page.
            Stop 'VBE Stop. Continue line by line to see what happens.
            Debug.Print .GetElementsByTagName("title")(0).innerHtml 'prints "Example Domain"
            Debug.Print .GetElementsByTagName("h1")(0).innerHtml 'prints "Example Domain"
            Debug.Print .GetElementsByTagName("p")(0).innerHtml 'prints "This domain is
established..."
            Debug.Print .GetElementsByTagName("p")(1).innerHtml 'prints "<a
href="http://www.iana.org/domains/example">More information...</a>"
            Debug.Print .GetElementsByTagName("p")(1).innerText 'prints "More information..."
            Debug.Print .GetElementsByTagName("a")(0).innerText 'prints "More information..."
```

```
            'We can change the localy displayed website. Don't worry about breaking the site.
            .GetElementsByTagName("title")(0).innerHtml = "Psst, scraping..."
            .GetElementsByTagName("h1")(0).innerHtml = "Let me try something fishy." 'You have
just changed the local HTML of the site.
            .GetElementsByTagName("p")(0).innerHtml = "Lorem ipsum.......... The End"
            .GetElementsByTagName("a")(0).innerText = "iana.org"
        End With '.document

        .Quit 'close the application window
    End With 'ie

End Sub
```

Cosa sta succedendo? Il giocatore chiave qui è il **documento** . Questo è il codice sorgente HTML. Possiamo applicare alcune query per ottenere le collezioni o l'oggetto che vogliamo.
Ad esempio, `IE.Document.GetElementsByTagName("title")(0).innerHtml` . `GetElementsByTagName` restituisce una **raccolta** di elementi HTML, che hanno il tag " *title* ". C'è solo un tag di questo tipo nel codice sorgente. La **raccolta** è basata su 0. Quindi per ottenere il primo elemento aggiungiamo `(0)` . Ora, nel nostro caso, vogliamo solo `innerHtml` (a String), non l'Oggetto Element stesso. Quindi specifichiamo la proprietà che vogliamo.

Clic

Per seguire un collegamento su un sito, possiamo utilizzare più metodi:

```
Sub IEGoToPlaces()
    Dim IE As InternetExplorer 'Reference to Microsoft Internet Controls
    Set IE = New InternetExplorer

    With IE
        .Visible = True
        .Navigate2 "http://www.example.com"
        Stop 'VBE Stop. Continue line by line to see what happens.

        'Click
        .Document.GetElementsByTagName("a")(0).Click
        Stop 'VBE Stop.

        'Return Back
        .GoBack
        Stop 'VBE Stop.

        'Navigate using the href attribute in the <a> tag, or "link"
        .Navigate2 .Document.GetElementsByTagName("a")(0).href
        Stop 'VBE Stop.

        .Quit 'close the application window
    End With
End Sub
```

Microsoft HTML Object Library o IE Migliore amico

Per ottenere il massimo dall'HTML che viene caricato nell'IE, è possibile (o dovrebbe) utilizzare

un'altra libreria, ad es. *Microsoft HTML Object Library* . Maggiori informazioni su questo in un altro esempio.

IE principali problemi

Il problema principale con IE è verificare che la pagina sia stata caricata ed è pronta per essere interagita con. Il `Do While... Loop` aiuta, ma non è affidabile.

Inoltre, usare IE solo per raschiare il contenuto HTML è OVERKILL. Perché? Poiché il browser è pensato per la navigazione, ovvero la visualizzazione della pagina Web con tutti i CSS, JavaScript, immagini, popup, ecc. Se sono necessari solo i dati grezzi, prendere in considerazione un approccio diverso. Ad esempio, usando XML HTTPRequest . Maggiori informazioni su questo in un altro esempio.

Leggi Automazione o utilizzo di altre applicazioni Librerie online:
https://riptutorial.com/it/vba/topic/8916/automazione-o-utilizzo-di-altre-applicazioni-librerie

Capitolo 6: Chiamate API

introduzione

API sta per Application Programming Interface

Le API per VBA implicano una serie di metodi che consentono l'interazione diretta con il sistema operativo

Le chiamate di sistema possono essere effettuate eseguendo le procedure definite nei file DLL

Osservazioni

File di libreria dell'ambiente operativo comune (DLL):

Libreria di collegamento dinamico	Descrizione
advapi32.dll	Libreria di servizi avanzati per API che include molte chiamate di sicurezza e di registro
Comdlg32.dll	Libreria API di dialogo comune
gdi32.dll	Libreria API di interfaccia dispositivo grafico
Kernel32.dll	Supporto base API di base a 32 bit per Windows
lz32.dll	Routine di compressione a 32 bit
Mpr.dll	Libreria di router di provider multipli
Netapi32.dll	Libreria API di rete a 32 bit
Shell32.dll	Libreria API shell a 32 bit
user32.dll	Libreria per le routine di interfaccia utente
Version.dll	Libreria di versioni
Winmm.dll	Libreria multimediale di Windows
Winspool.drv	Interfaccia dello spooler di stampa che contiene le chiamate API dello spooler di stampa

Nuovi argomenti usati per il sistema 64:

genere	Articolo	Descrizione
Qualifier	PtrSafe	Indica che l'istruzione Declare è compatibile con 64 bit. Questo attributo è obbligatorio sui sistemi a 64 bit
Tipo di dati	LongPtr	Un tipo di dati variabili che è un tipo di dati di 4 byte su versioni a 32 bit e un tipo di dati a 8 byte su versioni a 64 bit di Office 2010. Questo è il modo consigliato di dichiarare un puntatore o un handle per il nuovo codice ma anche per il codice legacy se deve essere eseguito nella versione a 64 bit di Office 2010. È supportato solo nel runtime VBA 7 su 32-bit e 64-bit. Si noti che è possibile assegnare valori numerici ma non tipi numerici
Tipo di dati	Lungo lungo	Questo è un tipo di dati a 8 byte che è disponibile solo nelle versioni a 64 bit di Office 2010. È possibile assegnare valori numerici ma non tipi numerici (per evitare il troncamento)
Conversione	Operatore	CLngPtr Converte un'espressione semplice in un tipo di dati LongPtr
Conversione	Operatore	CLngLng Converte un'espressione semplice in un tipo di dati LongLong
Funzione	VarPtr	Convertitore di varianti. Restituisce un LongPtr su versioni a 64 bit e un lungo su 32 bit (4 byte)
Funzione	ObjPtr	Convertitore di oggetti. Restituisce un LongPtr su versioni a 64 bit e un lungo su 32 bit (4 byte)
Funzione	StrPtr	Convertitore di stringhe. Restituisce un LongPtr su versioni a 64 bit e un lungo su 32 bit (4 byte)

Riferimento completo delle firme di chiamata:

- Win32api32.txt per Visual Basic 5.0 (vecchie dichiarazioni API, ultima revisione marzo 2005, Microsoft)
- Win32API_PtrSafe con supporto a 64 bit (Office 2010, Microsoft)

Examples

Dichiarazione e utilizzo dell'API

Dichiarare una procedura DLL per lavorare con diverse versioni VBA:

```
Option Explicit

#If Win64 Then
```

```
    Private Declare PtrSafe Sub xLib "Kernel32" Alias "Sleep" (ByVal dwMilliseconds As Long)
#ElseIf Win32 Then
    Private Declare Sub apiSleep Lib "Kernel32" Alias "Sleep" (ByVal dwMilliseconds As Long)
#End If
```

La dichiarazione sopra dice a VBA come chiamare la funzione "Sleep" definita nel file Kernel32.dll

Win64 e Win32 sono costanti predefinite utilizzate per la compilazione condizionale

---

Costanti predefinite

Alcune costanti di compilazione sono già predefinite. Quelli esistenti dipenderanno dalla versione della versione di Office in cui VBA è in esecuzione. Si noti che Vba7 è stato introdotto insieme a Office 2010 per supportare le versioni a 64 bit di Office.

Costante	16 bit	32 bit	64 bit
VBA6	falso	Se Vba6	falso
Vba7	falso	Se Vba7	Vero
Win16	Vero	falso	falso
Win32	falso	Vero	Vero
Win64	falso	falso	Vero
Mac	falso	Se Mac	Se Mac

Queste costanti si riferiscono alla versione di Office, non alla versione di Windows. Ad esempio Win32 = TRUE in Office a 32 bit, anche se il sistema operativo è una versione a 64 bit di Windows.

La principale differenza quando si dichiarano le API è tra le versioni di Office a 32 bit e 64 bit che hanno introdotto nuovi tipi di parametri (vedere la sezione Commenti per ulteriori dettagli)

Gli appunti:

- Le dichiarazioni sono posizionate nella parte superiore del modulo e all'esterno di qualsiasi sottoscheda o funzione
- Le procedure dichiarate nei moduli standard sono pubbliche per impostazione predefinita
- Per dichiarare una procedura privata a un modulo, precedere la dichiarazione con la parola chiave `Private`
- Le procedure DLL dichiarate in qualsiasi altro tipo di modulo sono private di quel

modulo

Semplice esempio per la chiamata API Sleep:

```
Public Sub TestPause()

    Dim start As Double

    start = Timer

    Sleep 9000      'Pause execution for 9 seconds

    Debug.Print "Paused for " & Format(Timer - start, "#,###.000") & " seconds"

    'Immediate window result: Paused for 9.000 seconds

End Sub
```

Si consiglia di creare un modulo API dedicato per fornire un facile accesso alle funzioni di sistema dai wrapper VBA: normali VBA Sub o Funzioni che incapsulano i dettagli necessari per la chiamata di sistema effettiva come i parametri utilizzati nelle librerie e l'inizializzazione di tali parametri

Il modulo può contenere tutte le dichiarazioni e le dipendenze:

- Firme dei metodi e strutture dati richieste
- Wrapper che eseguono la convalida dell'input e assicurano che tutti i parametri siano passati come previsto

---

Per dichiarare una procedura DLL, aggiungere un'istruzione `Declare` alla sezione Dichiarazioni della finestra del codice.

Se la procedura restituisce un valore, dichiararlo come una **funzione** :

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

Se una procedura non restituisce un valore, dichiararlo come **Sub** :

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

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

  Si noti inoltre che la **maggior parte delle chiamate non valide all'API interromperà Excel** e probabilmente corromperà i file di dati

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

Office 2011 per Mac

```
Private Declare Function system Lib "libc.dylib" (ByVal command As String) As Long

Sub RunSafari()
    Dim result As Long
    result = system("open -a Safari --args http://www.google.com")
    Debug.Print Str(result)
End Sub
```

Gli esempi seguenti (Windows API - Dedicated Module (1 e 2)) mostrano un modulo API che include dichiarazioni comuni per Win64 e Win32

API Windows - Modulo dedicato (1 di 2)

```
Option Explicit

#If Win64 Then  'Win64 = True, Win32 = False, Win16 = False
    Private Declare PtrSafe Sub apiCopyMemory Lib "Kernel32" Alias "RtlMoveMemory" (MyDest As
Any, MySource As Any, ByVal MySize As Long)
    Private Declare PtrSafe Sub apiExitProcess Lib "Kernel32" Alias "ExitProcess" (ByVal
uExitCode As Long)
    Private Declare PtrSafe Sub apiSetCursorPos Lib "User32" Alias "SetCursorPos" (ByVal X As
Integer, ByVal Y As Integer)
    Private Declare PtrSafe Sub apiSleep Lib "Kernel32" Alias "Sleep" (ByVal dwMilliseconds As
Long)
    Private Declare PtrSafe Function apiAttachThreadInput Lib "User32" Alias
"AttachThreadInput" (ByVal idAttach As Long, ByVal idAttachTo As Long, ByVal fAttach As Long)
As Long
    Private Declare PtrSafe Function apiBringWindowToTop Lib "User32" Alias "BringWindowToTop"
(ByVal lngHWnd As Long) As Long
    Private Declare PtrSafe Function apiCloseWindow Lib "User32" Alias "CloseWindow" (ByVal
hWnd As Long) As Long
    Private Declare PtrSafe Function apiDestroyWindow Lib "User32" Alias "DestroyWindow"
(ByVal hWnd As Long) As Boolean
    Private Declare PtrSafe Function apiEndDialog Lib "User32" Alias "EndDialog" (ByVal hWnd
As Long, ByVal result As Long) As Boolean
    Private Declare PtrSafe Function apiEnumChildWindows Lib "User32" Alias "EnumChildWindows"
(ByVal hWndParent As Long, ByVal pEnumProc As Long, ByVal lParam As Long) As Long
    Private Declare PtrSafe Function apiExitWindowsEx Lib "User32" Alias "ExitWindowsEx"
(ByVal uFlags As Long, ByVal dwReserved As Long) As Long
    Private Declare PtrSafe Function apiFindExecutable Lib "Shell32" Alias "FindExecutableA"
(ByVal lpFile As String, ByVallpDirectory As String, ByVal lpResult As String) As Long
    Private Declare PtrSafe Function apiFindWindow Lib "User32" Alias "FindWindowA" (ByVal
lpClassName As String, ByVal lpWindowName As String) As Long
    Private Declare PtrSafe Function apiFindWindowEx Lib "User32" Alias "FindWindowExA" (ByVal
hWnd1 As Long, ByVal hWnd2 As Long, ByVal lpsz1 As String, ByVal lpsz2 As String) As Long
    Private Declare PtrSafe Function apiGetActiveWindow Lib "User32" Alias "GetActiveWindow"
() As Long
    Private Declare PtrSafe Function apiGetClassNameA Lib "User32" Alias "GetClassNameA"
(ByVal hWnd As Long, ByVal szClassName As String, ByVal lLength As Long) As Long
    Private Declare PtrSafe Function apiGetCommandLine Lib "Kernel32" Alias "GetCommandLineW"
() As Long
    Private Declare PtrSafe Function apiGetCommandLineParams Lib "Kernel32" Alias
"GetCommandLineA" () As Long
    Private Declare PtrSafe Function apiGetDiskFreeSpaceEx Lib "Kernel32" Alias
"GetDiskFreeSpaceExA" (ByVal lpDirectoryName As String, lpFreeBytesAvailableToCaller As
```

```
Currency, lpTotalNumberOfBytes As Currency, lpTotalNumberOfFreeBytes As Currency) As Long
    Private Declare PtrSafe Function apiGetDriveType Lib "Kernel32" Alias "GetDriveTypeA"
(ByVal nDrive As String) As Long
    Private Declare PtrSafe Function apiGetExitCodeProcess Lib "Kernel32" Alias
"GetExitCodeProcess" (ByVal hProcess As Long, lpExitCode As Long) As Long
    Private Declare PtrSafe Function apiGetForegroundWindow Lib "User32" Alias
"GetForegroundWindow" () As Long
    Private Declare PtrSafe Function apiGetFrequency Lib "Kernel32" Alias
"QueryPerformanceFrequency" (cyFrequency As Currency) As Long
    Private Declare PtrSafe Function apiGetLastError Lib "Kernel32" Alias "GetLastError" () As
Integer
    Private Declare PtrSafe Function apiGetParent Lib "User32" Alias "GetParent" (ByVal hWnd
As Long) As Long
    Private Declare PtrSafe Function apiGetSystemMetrics Lib "User32" Alias "GetSystemMetrics"
(ByVal nIndex As Long) As Long
    Private Declare PtrSafe Function apiGetSystemMetrics32 Lib "User32" Alias
"GetSystemMetrics" (ByVal nIndex As Long) As Long
    Private Declare PtrSafe Function apiGetTickCount Lib "Kernel32" Alias
"QueryPerformanceCounter" (cyTickCount As Currency) As Long
    Private Declare PtrSafe Function apiGetTickCountMs Lib "Kernel32" Alias "GetTickCount" ()
As Long
    Private Declare PtrSafe Function apiGetUserName Lib "AdvApi32" Alias "GetUserNameA" (ByVal
lpBuffer As String, nSize As Long) As Long
    Private Declare PtrSafe Function apiGetWindow Lib "User32" Alias "GetWindow" (ByVal hWnd
As Long, ByVal wCmd As Long) As Long
    Private Declare PtrSafe Function apiGetWindowRect Lib "User32" Alias "GetWindowRect"
(ByVal hWnd As Long, lpRect As winRect) As Long
    Private Declare PtrSafe Function apiGetWindowText Lib "User32" Alias "GetWindowTextA"
(ByVal hWnd As Long, ByVal szWindowText As String, ByVal lLength As Long) As Long
    Private Declare PtrSafe Function apiGetWindowThreadProcessId Lib "User32" Alias
"GetWindowThreadProcessId" (ByVal hWnd As Long, lpdwProcessId As Long) As Long
    Private Declare PtrSafe Function apiIsCharAlphaNumericA Lib "User32" Alias
"IsCharAlphaNumericA" (ByVal byChar As Byte) As Long
    Private Declare PtrSafe Function apiIsIconic Lib "User32" Alias "IsIconic" (ByVal hWnd As
Long) As Long
    Private Declare PtrSafe Function apiIsWindowVisible Lib "User32" Alias "IsWindowVisible"
(ByVal hWnd As Long) As Long
    Private Declare PtrSafe Function apiIsZoomed Lib "User32" Alias "IsZoomed" (ByVal hWnd As
Long) As Long
    Private Declare PtrSafe Function apiLStrCpynA Lib "Kernel32" Alias "lstrcpynA" (ByVal
pDestination As String, ByVal pSource As Long, ByVal iMaxLength As Integer) As Long
    Private Declare PtrSafe Function apiMessageBox Lib "User32" Alias "MessageBoxA" (ByVal
hWnd As Long, ByVal lpText As String, ByVal lpCaption As String, ByVal wType As Long) As Long
    Private Declare PtrSafe Function apiOpenIcon Lib "User32" Alias "OpenIcon" (ByVal hWnd As
Long) As Long
    Private Declare PtrSafe Function apiOpenProcess Lib "Kernel32" Alias "OpenProcess" (ByVal
dwDesiredAccess As Long, ByVal bInheritHandle As Long, ByVal dwProcessId As Long) As Long
    Private Declare PtrSafe Function apiPathAddBackslashByPointer Lib "ShlwApi" Alias
"PathAddBackslashW" (ByVal lpszPath As Long) As Long
    Private Declare PtrSafe Function apiPathAddBackslashByString Lib "ShlwApi" Alias
"PathAddBackslashW" (ByVal lpszPath As String) As Long 'http://msdn.microsoft.com/en-
us/library/aa155716%28office.10%29.aspx
    Private Declare PtrSafe Function apiPostMessage Lib "User32" Alias "PostMessageA" (ByVal
hWnd As Long, ByVal wMsg As Long, ByVal wParam As Long, ByVal lParam As Long) As Long
    Private Declare PtrSafe Function apiRegQueryValue Lib "AdvApi32" Alias "RegQueryValue"
(ByVal hKey As Long, ByVal sValueName As String, ByVal dwReserved As Long, ByRef lValueType As
Long, ByVal sValue As String, ByRef lResultLen As Long) As Long
    Private Declare PtrSafe Function apiSendMessage Lib "User32" Alias "SendMessageA" (ByVal
hWnd As Long, ByVal wMsg As Long, ByVal wParam As Long, lParam As Any) As Long
    Private Declare PtrSafe Function apiSetActiveWindow Lib "User32" Alias "SetActiveWindow"
(ByVal hWnd As Long) As Long
```

```
    Private Declare PtrSafe Function apiSetCurrentDirectoryA Lib "Kernel32" Alias
"SetCurrentDirectoryA" (ByVal lpPathName As String) As Long
    Private Declare PtrSafe Function apiSetFocus Lib "User32" Alias "SetFocus" (ByVal hWnd As
Long) As Long
    Private Declare PtrSafe Function apiSetForegroundWindow Lib "User32" Alias
"SetForegroundWindow" (ByVal hWnd As Long) As Long
    Private Declare PtrSafe Function apiSetLocalTime Lib "Kernel32" Alias "SetLocalTime"
(lpSystem As SystemTime) As Long
    Private Declare PtrSafe Function apiSetWindowPlacement Lib "User32" Alias
"SetWindowPlacement" (ByVal hWnd As Long, ByRef lpwndpl As winPlacement) As Long
    Private Declare PtrSafe Function apiSetWindowPos Lib "User32" Alias "SetWindowPos" (ByVal
hWnd As Long, ByVal hWndInsertAfter As Long, ByVal X As Long, ByVal Y As Long, ByVal cx As
Long, ByVal cy As Long, ByVal wFlags As Long) As Long
    Private Declare PtrSafe Function apiSetWindowText Lib "User32" Alias "SetWindowTextA"
(ByVal hWnd As Long, ByVal lpString As String) As Long
    Private Declare PtrSafe Function apiShellExecute Lib "Shell32" Alias "ShellExecuteA"
(ByVal hWnd As Long, ByVal lpOperation As String, ByVal lpFile As String, ByVal lpParameters
As String, ByVal lpDirectory As String, ByVal nShowCmd As Long) As Long
    Private Declare PtrSafe Function apiShowWindow Lib "User32" Alias "ShowWindow" (ByVal hWnd
As Long, ByVal nCmdShow As Long) As Long
    Private Declare PtrSafe Function apiShowWindowAsync Lib "User32" Alias "ShowWindowAsync"
(ByVal hWnd As Long, ByVal nCmdShow As Long) As Long
    Private Declare PtrSafe Function apiStrCpy Lib "Kernel32" Alias "lstrcpynA" (ByVal
pDestination As String, ByVal pSource As String, ByVal iMaxLength As Integer) As Long
    Private Declare PtrSafe Function apiStringLen Lib "Kernel32" Alias "lstrlenW" (ByVal
lpString As Long) As Long
    Private Declare PtrSafe Function apiStrTrimW Lib "ShlwApi" Alias "StrTrimW" () As Boolean
    Private Declare PtrSafe Function apiTerminateProcess Lib "Kernel32" Alias
"TerminateProcess" (ByVal hWnd As Long, ByVal uExitCode As Long) As Long
    Private Declare PtrSafe Function apiTimeGetTime Lib "Winmm" Alias "timeGetTime" () As Long
    Private Declare PtrSafe Function apiVarPtrArray Lib "MsVbVm50" Alias "VarPtr" (Var() As
Any) As Long
    Private Type browseInfo     'used by apiBrowseForFolder
        hOwner As Long
        pidlRoot As Long
        pszDisplayName As String
        lpszTitle As String
        ulFlags As Long
        lpfn As Long
        lParam As Long
        iImage As Long
    End Type
    Private Declare PtrSafe Function apiBrowseForFolder Lib "Shell32" Alias
"SHBrowseForFolderA" (lpBrowseInfo As browseInfo) As Long
    Private Type CHOOSECOLOR    'used by apiChooseColor;
http://support.microsoft.com/kb/153929 and http://www.cpearson.com/Excel/Colors.aspx
        lStructSize As Long
        hWndOwner As Long
        hInstance As Long
        rgbResult As Long
        lpCustColors As String
        flags As Long
        lCustData As Long
        lpfnHook As Long
        lpTemplateName As String
    End Type
    Private Declare PtrSafe Function apiChooseColor Lib "ComDlg32" Alias "ChooseColorA"
(pChoosecolor As CHOOSECOLOR) As Long
    Private Type FindWindowParameters   'Custom structure for passing in the parameters in/out
of the hook enumeration function; could use global variables instead, but this is nicer
        strTitle As String  'INPUT
```

```
        hWnd As Long        'OUTPUT
    End Type                            'Find a specific window with dynamic caption from a
list of all open windows: http://www.everythingaccess.com/tutorials.asp?ID=Bring-an-external-
application-window-to-the-foreground
    Private Declare PtrSafe Function apiEnumWindows Lib "User32" Alias "EnumWindows" (ByVal
lpEnumFunc As LongPtr, ByVal lParam As LongPtr) As Long
    Private Type lastInputInfo  'used by apiGetLastInputInfo, getLastInputTime
        cbSize As Long
        dwTime As Long
    End Type
    Private Declare PtrSafe Function apiGetLastInputInfo Lib "User32" Alias "GetLastInputInfo"
(ByRef plii As lastInputInfo) As Long
    'http://www.pgacon.com/visualbasic.htm#Take%20Advantage%20of%20Conditional%20Compilation
    'Logical and Bitwise Operators in Visual Basic: http://msdn.microsoft.com/en-
us/library/wz3k228a(v=vs.80).aspx and http://stackoverflow.com/questions/1070863/hidden-
features-of-vba
    Private Type SystemTime
          wYear          As Integer
          wMonth         As Integer
          wDayOfWeek     As Integer
          wDay           As Integer
          wHour          As Integer
          wMinute        As Integer
          wSecond        As Integer
          wMilliseconds  As Integer
    End Type
    Private Declare PtrSafe Sub apiGetLocalTime Lib "Kernel32" Alias "GetLocalTime" (lpSystem
As SystemTime)
    Private Type pointAPI       'used by apiSetWindowPlacement
         X As Long
         Y As Long
    End Type
    Private Type rectAPI       'used by apiSetWindowPlacement
        Left_Renamed As Long
        Top_Renamed As Long
        Right_Renamed As Long
        Bottom_Renamed As Long
    End Type
    Private Type winPlacement   'used by apiSetWindowPlacement
        length As Long
        flags As Long
        showCmd As Long
        ptMinPosition As pointAPI
        ptMaxPosition As pointAPI
        rcNormalPosition As rectAPI
    End Type
    Private Declare PtrSafe Function apiGetWindowPlacement Lib "User32" Alias
"GetWindowPlacement" (ByVal hWnd As Long, ByRef lpwndpl As winPlacement) As Long
    Private Type winRect     'used by apiMoveWindow
        Left As Long
        Top As Long
        Right As Long
        Bottom As Long
    End Type
    Private Declare PtrSafe Function apiMoveWindow Lib "User32" Alias "MoveWindow" (ByVal hWnd
As Long, xLeft As Long, ByVal yTop As Long, wWidth As Long, ByVal hHeight As Long, ByVal
repaint As Long) As Long

    Private Declare PtrSafe Function apiInternetOpen Lib "WiniNet" Alias "InternetOpenA"
(ByVal sAgent As String, ByVal lAccessType As Long, ByVal sProxyName As String, ByVal
sProxyBypass As String, ByVal lFlags As Long) As Long    'Open the Internet object    'ex:
```

```
lngINet = InternetOpen("MyFTP Control", 1, vbNullString, vbNullString, 0)
    Private Declare PtrSafe Function apiInternetConnect Lib "WiniNet" Alias "InternetConnectA"
(ByVal hInternetSession As Long, ByVal sServerName As String, ByVal nServerPort As Integer,
ByVal sUsername As String, ByVal sPassword As String, ByVal lService As Long, ByVal lFlags As
Long, ByVal lContext As Long) As Long  'Connect to the network  'ex: lngINetConn =
InternetConnect(lngINet, "ftp.microsoft.com", 0, "anonymous", "wally@wallyworld.com", 1, 0, 0)
    Private Declare PtrSafe Function apiFtpGetFile Lib "WiniNet" Alias "FtpGetFileA" (ByVal
hFtpSession As Long, ByVal lpszRemoteFile As String, ByVal lpszNewFile As String, ByVal
fFailIfExists As Boolean, ByVal dwFlagsAndAttributes As Long, ByVal dwFlags As Long, ByVal
dwContext As Long) As Boolean    'Get a file 'ex: blnRC = FtpGetFile(lngINetConn,
"dirmap.txt", "c:\dirmap.txt", 0, 0, 1, 0)
    Private Declare PtrSafe Function apiFtpPutFile Lib "WiniNet" Alias "FtpPutFileA" (ByVal
hFtpSession As Long, ByVal lpszLocalFile As String, ByVal lpszRemoteFile As String, ByVal
dwFlags As Long, ByVal dwContext As Long) As Boolean  'Send a file  'ex: blnRC =
FtpPutFile(lngINetConn, "c:\dirmap.txt", "dirmap.txt", 1, 0)
    Private Declare PtrSafe Function apiFtpDeleteFile Lib "WiniNet" Alias "FtpDeleteFileA"
(ByVal hFtpSession As Long, ByVal lpszFileName As String) As Boolean 'Delete a file 'ex: blnRC
= FtpDeleteFile(lngINetConn, "test.txt")
    Private Declare PtrSafe Function apiInternetCloseHandle Lib "WiniNet" (ByVal hInet As
Long) As Integer  'Close the Internet object  'ex: InternetCloseHandle lngINetConn    'ex:
InternetCloseHandle lngINet
    Private Declare PtrSafe Function apiFtpFindFirstFile Lib "WiniNet" Alias
"FtpFindFirstFileA" (ByVal hFtpSession As Long, ByVal lpszSearchFile As String, lpFindFileData
As WIN32_FIND_DATA, ByVal dwFlags As Long, ByVal dwContent As Long) As Long
    Private Type FILETIME
        dwLowDateTime As Long
        dwHighDateTime As Long
    End Type
    Private Type WIN32_FIND_DATA
        dwFileAttributes As Long
        ftCreationTime As FILETIME
        ftLastAccessTime As FILETIME
        ftLastWriteTime As FILETIME
        nFileSizeHigh As Long
        nFileSizeLow As Long
        dwReserved0 As Long
        dwReserved1 As Long
        cFileName As String * 1 'MAX_FTP_PATH
        cAlternate As String * 14
    End Type    'ex: lngHINet = FtpFindFirstFile(lngINetConn, "*.*", pData, 0, 0)
    Private Declare PtrSafe Function apiInternetFindNextFile Lib "WiniNet" Alias
"InternetFindNextFileA" (ByVal hFind As Long, lpvFindData As WIN32_FIND_DATA) As Long  'ex:
blnRC = InternetFindNextFile(lngHINet, pData)
#ElseIf Win32 Then  'Win32 = True, Win16 = False
```

(continua nel secondo esempio)

API Windows - Modulo dedicato (2 di 2)

```
#ElseIf Win32 Then  'Win32 = True, Win16 = False
    Private Declare Sub apiCopyMemory Lib "Kernel32" Alias "RtlMoveMemory" (MyDest As Any,
MySource As Any, ByVal MySize As Long)
    Private Declare Sub apiExitProcess Lib "Kernel32" Alias "ExitProcess" (ByVal uExitCode As
Long)
    'Private Declare Sub apiGetStartupInfo Lib "Kernel32" Alias "GetStartupInfoA"
(lpStartupInfo As STARTUPINFO)
    Private Declare Sub apiSetCursorPos Lib "User32" Alias "SetCursorPos" (ByVal X As Integer,
ByVal Y As Integer)  'Logical and Bitwise Operators in Visual Basic:
http://msdn.microsoft.com/en-us/library/wz3k228a(v=vs.80).aspx and
```

```
http://stackoverflow.com/questions/1070863/hidden-features-of-vba
'http://www.pgacon.com/visualbasic.htm#Take%20Advantage%20of%20Conditional%20Compilation
    Private Declare Sub apiSleep Lib "Kernel32" Alias "Sleep" (ByVal dwMilliseconds As Long)
    Private Declare Function apiAttachThreadInput Lib "User32" Alias "AttachThreadInput"
(ByVal idAttach As Long, ByVal idAttachTo As Long, ByVal fAttach As Long) As Long
    Private Declare Function apiBringWindowToTop Lib "User32" Alias "BringWindowToTop" (ByVal
lngHWnd As Long) As Long
    Private Declare Function apiCloseHandle Lib "Kernel32" (ByVal hObject As Long) As Long
    Private Declare Function apiCloseWindow Lib "User32" Alias "CloseWindow" (ByVal hWnd As
Long) As Long
    'Private Declare Function apiCreatePipe Lib "Kernel32" (phReadPipe As Long, phWritePipe As
Long, lpPipeAttributes As SECURITY_ATTRIBUTES, ByVal nSize As Long) As Long
    'Private Declare Function apiCreateProcess Lib "Kernel32" Alias "CreateProcessA" (ByVal
lpApplicationName As Long, ByVal lpCommandLine As String, lpProcessAttributes As Any,
lpThreadAttributes As Any, ByVal bInheritHandles As Long, ByVal dwCreationFlags As Long,
lpEnvironment As Any, ByVal lpCurrentDriectory As String, lpStartupInfo As STARTUPINFO,
lpProcessInformation As PROCESS_INFORMATION) As Long
    Private Declare Function apiDestroyWindow Lib "User32" Alias "DestroyWindow" (ByVal hWnd
As Long) As Boolean
    Private Declare Function apiEndDialog Lib "User32" Alias "EndDialog" (ByVal hWnd As Long,
ByVal result As Long) As Boolean
    Private Declare Function apiEnumChildWindows Lib "User32" Alias "EnumChildWindows" (ByVal
hWndParent As Long, ByVal pEnumProc As Long, ByVal lParam As Long) As Long
    Private Declare Function apiExitWindowsEx Lib "User32" Alias "ExitWindowsEx" (ByVal uFlags
As Long, ByVal dwReserved As Long) As Long
    Private Declare Function apiFindExecutable Lib "Shell32" Alias "FindExecutableA" (ByVal
lpFile As String, ByVallpDirectory As String, ByVal lpResult As String) As Long
    Private Declare Function apiFindWindow Lib "User32" Alias "FindWindowA" (ByVal lpClassName
As String, ByVal lpWindowName As String) As Long
    Private Declare Function apiFindWindowEx Lib "User32" Alias "FindWindowExA" (ByVal hWnd1
As Long, ByVal hWnd2 As Long, ByVal lpsz1 As String, ByVal lpsz2 As String) As Long
    Private Declare Function apiGetActiveWindow Lib "User32" Alias "GetActiveWindow" () As
Long
    Private Declare Function apiGetClassNameA Lib "User32" Alias "GetClassNameA" (ByVal hWnd
As Long, ByVal szClassName As String, ByVal lLength As Long) As Long
    Private Declare Function apiGetCommandLine Lib "Kernel32" Alias "GetCommandLineW" () As
Long
    Private Declare Function apiGetCommandLineParams Lib "Kernel32" Alias "GetCommandLineA" ()
As Long
    Private Declare Function apiGetDiskFreeSpaceEx Lib "Kernel32" Alias "GetDiskFreeSpaceExA"
(ByVal lpDirectoryName As String, lpFreeBytesAvailableToCaller As Currency,
lpTotalNumberOfBytes As Currency, lpTotalNumberOfFreeBytes As Currency) As Long
    Private Declare Function apiGetDriveType Lib "Kernel32" Alias "GetDriveTypeA" (ByVal
nDrive As String) As Long
    Private Declare Function apiGetExitCodeProcess Lib "Kernel32" (ByVal hProcess As Long,
lpExitCode As Long) As Long
    Private Declare Function apiGetFileSize Lib "Kernel32" (ByVal hFile As Long,
lpFileSizeHigh As Long) As Long
    Private Declare Function apiGetForegroundWindow Lib "User32" Alias "GetForegroundWindow"
() As Long
    Private Declare Function apiGetFrequency Lib "Kernel32" Alias "QueryPerformanceFrequency"
(cyFrequency As Currency) As Long
    Private Declare Function apiGetLastError Lib "Kernel32" Alias "GetLastError" () As Integer
    Private Declare Function apiGetParent Lib "User32" Alias "GetParent" (ByVal hWnd As Long)
As Long
    Private Declare Function apiGetSystemMetrics Lib "User32" Alias "GetSystemMetrics" (ByVal
nIndex As Long) As Long
    Private Declare Function apiGetTickCount Lib "Kernel32" Alias "QueryPerformanceCounter"
(cyTickCount As Currency) As Long
    Private Declare Function apiGetTickCountMs Lib "Kernel32" Alias "GetTickCount" () As Long
    Private Declare Function apiGetUserName Lib "AdvApi32" Alias "GetUserNameA" (ByVal
```

```
lpBuffer As String, nSize As Long) As Long
    Private Declare Function apiGetWindow Lib "User32" Alias "GetWindow" (ByVal hWnd As Long,
ByVal wCmd As Long) As Long
    Private Declare Function apiGetWindowRect Lib "User32" Alias "GetWindowRect" (ByVal hWnd
As Long, lpRect As winRect) As Long
    Private Declare Function apiGetWindowText Lib "User32" Alias "GetWindowTextA" (ByVal hWnd
As Long, ByVal szWindowText As String, ByVal lLength As Long) As Long
    Private Declare Function apiGetWindowThreadProcessId Lib "User32" Alias
"GetWindowThreadProcessId" (ByVal hWnd As Long, lpdwProcessId As Long) As Long
    Private Declare Function apiIsCharAlphaNumericA Lib "User32" Alias "IsCharAlphaNumericA"
(ByVal byChar As Byte) As Long
    Private Declare Function apiIsIconic Lib "User32" Alias "IsIconic" (ByVal hWnd As Long) As
Long
    Private Declare Function apiIsWindowVisible Lib "User32" Alias "IsWindowVisible" (ByVal
hWnd As Long) As Long
    Private Declare Function apiIsZoomed Lib "User32" Alias "IsZoomed" (ByVal hWnd As Long) As
Long
    Private Declare Function apiLStrCpynA Lib "Kernel32" Alias "lstrcpynA" (ByVal pDestination
As String, ByVal pSource As Long, ByVal iMaxLength As Integer) As Long
    Private Declare Function apiMessageBox Lib "User32" Alias "MessageBoxA" (ByVal hWnd As
Long, ByVal lpText As String, ByVal lpCaption As String, ByVal wType As Long) As Long
    Private Declare Function apiOpenIcon Lib "User32" Alias "OpenIcon" (ByVal hWnd As Long) As
Long
    Private Declare Function apiOpenProcess Lib "Kernel32" Alias "OpenProcess" (ByVal
dwDesiredAccess As Long, ByVal bInheritHandle As Long, ByVal dwProcessId As Long) As Long
    Private Declare Function apiPathAddBackslashByPointer Lib "ShlwApi" Alias
"PathAddBackslashW" (ByVal lpszPath As Long) As Long
    Private Declare Function apiPathAddBackslashByString Lib "ShlwApi" Alias
"PathAddBackslashW" (ByVal lpszPath As String) As Long 'http://msdn.microsoft.com/en-
us/library/aa155716%28office.10%29.aspx
    Private Declare Function apiPostMessage Lib "User32" Alias "PostMessageA" (ByVal hWnd As
Long, ByVal wMsg As Long, ByVal wParam As Long, ByVal lParam As Long) As Long
    Private Declare Function apiReadFile Lib "Kernel32" (ByVal hFile As Long, lpBuffer As Any,
ByVal nNumberOfBytesToRead As Long, lpNumberOfBytesRead As Long, lpOverlapped As Any) As Long
    Private Declare Function apiRegQueryValue Lib "AdvApi32" Alias "RegQueryValue" (ByVal hKey
As Long, ByVal sValueName As String, ByVal dwReserved As Long, ByRef lValueType As Long, ByVal
sValue As String, ByRef lResultLen As Long) As Long
    Private Declare Function apiSendMessage Lib "User32" Alias "SendMessageA" (ByVal hWnd As
Long, ByVal wMsg As Long, ByVal wParam As Long, lParam As Any) As Long
    Private Declare Function apiSetActiveWindow Lib "User32" Alias "SetActiveWindow" (ByVal
hWnd As Long) As Long
    Private Declare Function apiSetCurrentDirectoryA Lib "Kernel32" Alias
"SetCurrentDirectoryA" (ByVal lpPathName As String) As Long
    Private Declare Function apiSetFocus Lib "User32" Alias "SetFocus" (ByVal hWnd As Long) As
Long
    Private Declare Function apiSetForegroundWindow Lib "User32" Alias "SetForegroundWindow"
(ByVal hWnd As Long) As Long
    Private Declare Function apiSetLocalTime Lib "Kernel32" Alias "SetLocalTime" (lpSystem As
SystemTime) As Long
    Private Declare Function apiSetWindowPlacement Lib "User32" Alias "SetWindowPlacement"
(ByVal hWnd As Long, ByRef lpwndpl As winPlacement) As Long
    Private Declare Function apiSetWindowPos Lib "User32" Alias "SetWindowPos" (ByVal hWnd As
Long, ByVal hWndInsertAfter As Long, ByVal X As Long, ByVal Y As Long, ByVal cx As Long, ByVal
cy As Long, ByVal wFlags As Long) As Long
    Private Declare Function apiSetWindowText Lib "User32" Alias "SetWindowTextA" (ByVal hWnd
As Long, ByVal lpString As String) As Long
    Private Declare Function apiShellExecute Lib "Shell32" Alias "ShellExecuteA" (ByVal hWnd
As Long, ByVal lpOperation As String, ByVal lpFile As String, ByVal lpParameters As String,
ByVal lpDirectory As String, ByVal nShowCmd As Long) As Long
    Private Declare Function apiShowWindow Lib "User32" Alias "ShowWindow" (ByVal hWnd As
Long, ByVal nCmdShow As Long) As Long
```

```
    Private Declare Function apiShowWindowAsync Lib "User32" Alias "ShowWindowAsync" (ByVal
hWnd As Long, ByVal nCmdShow As Long) As Long
    Private Declare Function apiStrCpy Lib "Kernel32" Alias "lstrcpynA" (ByVal pDestination As
String, ByVal pSource As String, ByVal iMaxLength As Integer) As Long
    Private Declare Function apiStringLen Lib "Kernel32" Alias "lstrlenW" (ByVal lpString As
Long) As Long
    Private Declare Function apiStrTrimW Lib "ShlwApi" Alias "StrTrimW" () As Boolean
    Private Declare Function apiTerminateProcess Lib "Kernel32" Alias "TerminateProcess"
(ByVal hWnd As Long, ByVal uExitCode As Long) As Long
    Private Declare Function apiTimeGetTime Lib "Winmm" Alias "timeGetTime" () As Long
    Private Declare Function apiVarPtrArray Lib "MsVbVm50" Alias "VarPtr" (Var() As Any) As
Long
    Private Declare Function apiWaitForSingleObject Lib "Kernel32" (ByVal hHandle As Long,
ByVal dwMilliseconds As Long) As Long
    Private Type browseInfo     'used by apiBrowseForFolder
        hOwner As Long
        pidlRoot As Long
        pszDisplayName As String
        lpszTitle As String
        ulFlags As Long
        lpfn As Long
        lParam As Long
        iImage As Long
    End Type
    Private Declare Function apiBrowseForFolder Lib "Shell32" Alias "SHBrowseForFolderA"
(lpBrowseInfo As browseInfo) As Long
    Private Type CHOOSECOLOR    'used by apiChooseColor;
http://support.microsoft.com/kb/153929 and http://www.cpearson.com/Excel/Colors.aspx
        lStructSize As Long
        hWndOwner As Long
        hInstance As Long
        rgbResult As Long
        lpCustColors As String
        flags As Long
        lCustData As Long
        lpfnHook As Long
        lpTemplateName As String
    End Type
    Private Declare Function apiChooseColor Lib "ComDlg32" Alias "ChooseColorA" (pChoosecolor
As CHOOSECOLOR) As Long
    Private Type FindWindowParameters   'Custom structure for passing in the parameters in/out
of the hook enumeration function; could use global variables instead, but this is nicer
        strTitle As String  'INPUT
        hWnd As Long        'OUTPUT
    End Type                            'Find a specific window with dynamic caption from a
list of all open windows: http://www.everythingaccess.com/tutorials.asp?ID=Bring-an-external-
application-window-to-the-foreground
    Private Declare Function apiEnumWindows Lib "User32" Alias "EnumWindows" (ByVal lpEnumFunc
As Long, ByVal lParam As Long) As Long
    Private Type lastInputInfo  'used by apiGetLastInputInfo, getLastInputTime
        cbSize As Long
        dwTime As Long
    End Type
    Private Declare Function apiGetLastInputInfo Lib "User32" Alias "GetLastInputInfo" (ByRef
plii As lastInputInfo) As Long
    Private Type SystemTime
          wYear          As Integer
          wMonth         As Integer
          wDayOfWeek     As Integer
          wDay           As Integer
          wHour          As Integer
```

```
          wMinute        As Integer
          wSecond        As Integer
          wMilliseconds  As Integer
    End Type
    Private Declare Sub apiGetLocalTime Lib "Kernel32" Alias "GetLocalTime" (lpSystem As
SystemTime)
    Private Type pointAPI
        X As Long
        Y As Long
    End Type
    Private Type rectAPI
        Left_Renamed As Long
        Top_Renamed As Long
        Right_Renamed As Long
        Bottom_Renamed As Long
    End Type
    Private Type winPlacement
        length As Long
        flags As Long
        showCmd As Long
        ptMinPosition As pointAPI
        ptMaxPosition As pointAPI
        rcNormalPosition As rectAPI
    End Type
    Private Declare Function apiGetWindowPlacement Lib "User32" Alias "GetWindowPlacement"
(ByVal hWnd As Long, ByRef lpwndpl As winPlacement) As Long
    Private Type winRect
        Left As Long
        Top As Long
        Right As Long
        Bottom As Long
    End Type
    Private Declare Function apiMoveWindow Lib "User32" Alias "MoveWindow" (ByVal hWnd As
Long, xLeft As Long, ByVal yTop As Long, wWidth As Long, ByVal hHeight As Long, ByVal repaint
As Long) As Long
#Else   ' Win16 = True
#End If
```

API Mac

Microsoft non supporta ufficialmente le API, ma con alcune ricerche è possibile trovare più dichiarazioni online

Office 2016 per Mac è in modalità sandbox

A differenza di altre versioni delle app di Office che supportano VBA, le app di Office 2016 per Mac sono in modalità sandbox.

Il sandboxing impedisce alle app di accedere alle risorse al di fuori del contenitore dell'app. Ciò influisce su eventuali componenti aggiuntivi o macro che implicano l'accesso ai file o la comunicazione tra i processi. È possibile ridurre al minimo gli effetti del sandboxing utilizzando i nuovi comandi descritti nella sezione seguente. Nuovi comandi VBA per Office 2016 per Mac

I seguenti comandi VBA sono nuovi e unici per Office 2016 per Mac.

Comando	Utilizzare a
GrantAccessToMultipleFiles	Richiedi il permesso di un utente per accedere a più file contemporaneamente
AppleScriptTask	Chiama script AppleScript esterni da VB
MAC_OFFICE_VERSION	IFDEF tra diverse versioni di Mac Office in fase di compilazione

---

Office 2011 per Mac

```
Private Declare Function system Lib "libc.dylib" (ByVal command As String) As Long
Private Declare Function popen Lib "libc.dylib" (ByVal command As String, ByVal mode As
String) As Long
Private Declare Function pclose Lib "libc.dylib" (ByVal file As Long) As Long
Private Declare Function fread Lib "libc.dylib" (ByVal outStr As String, ByVal size As Long,
ByVal items As Long, ByVal stream As Long) As Long
Private Declare Function feof Lib "libc.dylib" (ByVal file As Long) As Long
```

-

Office 2016 per Mac

```
Private Declare PtrSafe Function popen Lib "libc.dylib" (ByVal command As String, ByVal mode
As String) As LongPtr
Private Declare PtrSafe Function pclose Lib "libc.dylib" (ByVal file As LongPtr) As Long
Private Declare PtrSafe Function fread Lib "libc.dylib" (ByVal outStr As String, ByVal size As
LongPtr, ByVal items As LongPtr, ByVal stream As LongPtr) As Long
Private Declare PtrSafe Function feof Lib "libc.dylib" (ByVal file As LongPtr) As LongPtr
```

Ottieni monitor totali e risoluzione dello schermo

```
Option Explicit

'GetSystemMetrics32 info: http://msdn.microsoft.com/en-us/library/ms724385(VS.85).aspx
#If Win64 Then
    Private Declare PtrSafe Function GetSystemMetrics32 Lib "User32" Alias "GetSystemMetrics"
(ByVal nIndex As Long) As Long
#ElseIf Win32 Then
    Private Declare Function GetSystemMetrics32 Lib "User32" Alias "GetSystemMetrics" (ByVal
nIndex As Long) As Long
#End If

'VBA Wrappers:
Public Function dllGetMonitors() As Long
    Const SM_CMONITORS = 80
    dllGetMonitors = GetSystemMetrics32(SM_CMONITORS)
End Function

Public Function dllGetHorizontalResolution() As Long
    Const SM_CXVIRTUALSCREEN = 78
    dllGetHorizontalResolution = GetSystemMetrics32(SM_CXVIRTUALSCREEN)
```

```
End Function

Public Function dllGetVerticalResolution() As Long
    Const SM_CYVIRTUALSCREEN = 79
    dllGetVerticalResolution = GetSystemMetrics32(SM_CYVIRTUALSCREEN)
End Function

Public Sub ShowDisplayInfo()
    Debug.Print "Total monitors: " & vbTab & vbTab & dllGetMonitors
    Debug.Print "Horizontal Resolution: " & vbTab & dllGetHorizontalResolution
    Debug.Print "Vertical Resolution: " & vbTab & dllGetVerticalResolution

    'Total monitors:         1
    'Horizontal Resolution:  1920
    'Vertical Resolution:    1080
End Sub
```

FTP e API regionali

modFTP

```
Option Explicit
Option Compare Text
Option Private Module

'http://msdn.microsoft.com/en-us/library/aa384180(v=VS.85).aspx
'http://www.dailydoseofexcel.com/archives/2006/01/29/ftp-via-vba/
'http://www.15seconds.com/issue/981203.htm

'Open the Internet object
Private Declare Function InternetOpen Lib "wininet.dll" Alias "InternetOpenA" ( _
    ByVal sAgent As String, _
    ByVal lAccessType As Long, _
    ByVal sProxyName As String, _
    ByVal sProxyBypass As String, _
    ByVal lFlags As Long _
) As Long
'ex: lngINet = InternetOpen("MyFTP Control", 1, vbNullString, vbNullString, 0)

'Connect to the network
Private Declare Function InternetConnect Lib "wininet.dll" Alias "InternetConnectA" ( _
    ByVal hInternetSession As Long, _
    ByVal sServerName As String, _
    ByVal nServerPort As Integer, _
    ByVal sUsername As String, _
    ByVal sPassword As String, _
    ByVal lService As Long, _
    ByVal lFlags As Long, _
    ByVal lContext As Long _
) As Long
'ex: lngINetConn = InternetConnect(lngINet, "ftp.microsoft.com", 0, "anonymous",
"wally@wallyworld.com", 1, 0, 0)

'Get a file
Private Declare Function FtpGetFile Lib "wininet.dll" Alias "FtpGetFileA" ( _
    ByVal hFtpSession As Long, _
    ByVal lpszRemoteFile As String, _
    ByVal lpszNewFile As String, _
```

```
    ByVal fFailIfExists As Boolean, _
    ByVal dwFlagsAndAttributes As Long, _
    ByVal dwFlags As Long, _
    ByVal dwContext As Long _
) As Boolean
'ex: blnRC = FtpGetFile(lngINetConn, "dirmap.txt", "c:\dirmap.txt", 0, 0, 1, 0)

'Send a file
Private Declare Function FtpPutFile Lib "wininet.dll" Alias "FtpPutFileA" _
( _
    ByVal hFtpSession As Long, _
    ByVal lpszLocalFile As String, _
    ByVal lpszRemoteFile As String, _
    ByVal dwFlags As Long, ByVal dwContext As Long _
) As Boolean
'ex: blnRC = FtpPutFile(lngINetConn, "c:\dirmap.txt", "dirmap.txt", 1, 0)

'Delete a file
Private Declare Function FtpDeleteFile Lib "wininet.dll" Alias "FtpDeleteFileA" _
( _
    ByVal hFtpSession As Long, _
    ByVal lpszFileName As String _
) As Boolean
'ex: blnRC = FtpDeleteFile(lngINetConn, "test.txt")

'Close the Internet object
Private Declare Function InternetCloseHandle Lib "wininet.dll" (ByVal hInet As Long) As
Integer
'ex: InternetCloseHandle lngINetConn
'ex: InternetCloseHandle lngINet



Private Declare Function FtpFindFirstFile Lib "wininet.dll" Alias "FtpFindFirstFileA" _
( _
    ByVal hFtpSession As Long, _
    ByVal lpszSearchFile As String, _
    lpFindFileData As WIN32_FIND_DATA, _
    ByVal dwFlags As Long, _
    ByVal dwContent As Long _
) As Long
Private Type FILETIME
        dwLowDateTime As Long
        dwHighDateTime As Long
End Type
Private Type WIN32_FIND_DATA
        dwFileAttributes As Long
        ftCreationTime As FILETIME
        ftLastAccessTime As FILETIME
        ftLastWriteTime As FILETIME
        nFileSizeHigh As Long
        nFileSizeLow As Long
        dwReserved0 As Long
        dwReserved1 As Long
        cFileName As String * MAX_FTP_PATH
        cAlternate As String * 14
End Type
'ex: lngHINet = FtpFindFirstFile(lngINetConn, "*.*", pData, 0, 0)

Private Declare Function InternetFindNextFile Lib "wininet.dll" Alias "InternetFindNextFileA"
_
```

```
( _
    ByVal hFind As Long, _
    lpvFindData As WIN32_FIND_DATA _
) As Long
'ex: blnRC = InternetFindNextFile(lngHINet, pData)


Public Sub showLatestFTPVersion()
    Dim ftpSuccess As Boolean, msg As String, lngFindFirst As Long
    Dim lngINet As Long, lngINetConn As Long
    Dim pData As WIN32_FIND_DATA
    'init the filename buffer
    pData.cFileName = String(260, 0)

    msg = "FTP Error"
    lngINet = InternetOpen("MyFTP Control", 1, vbNullString, vbNullString, 0)
    If lngINet > 0 Then
        lngINetConn = InternetConnect(lngINet, FTP_SERVER_NAME, FTP_SERVER_PORT,
FTP_USER_NAME, FTP_PASSWORD, 1, 0, 0)
        If lngINetConn > 0 Then
                FtpPutFile lngINetConn, "C:\Tmp\ftp.cls", "ftp.cls", FTP_TRANSFER_BINARY, 0
                'lngFindFirst = FtpFindFirstFile(lngINetConn, "ExcelDiff.xlsm", pData, 0, 0)
            If lngINet = 0 Then
                msg = "DLL error: " & Err.LastDllError & ", Error Number: " & Err.Number & ",
Error Desc: " & Err.Description
            Else
                msg = left(pData.cFileName, InStr(1, pData.cFileName, String(1, 0),
vbBinaryCompare) - 1)
            End If
            InternetCloseHandle lngINetConn
        End If
    InternetCloseHandle lngINet
    End If
        MsgBox msg
End Sub
```

---

modRegional:

```
Option Explicit

Private Const LOCALE_SDECIMAL = &HE
Private Const LOCALE_SLIST = &HC

Private Declare Function GetLocaleInfo Lib "Kernel32" Alias "GetLocaleInfoA" (ByVal Locale As
Long, ByVal LCType As Long, ByVal lpLCData As String, ByVal cchData As Long) As Long
Private Declare Function SetLocaleInfo Lib "Kernel32" Alias "SetLocaleInfoA" (ByVal Locale As
Long, ByVal LCType As Long, ByVal lpLCData As String) As Boolean
Private Declare Function GetUserDefaultLCID% Lib "Kernel32" ()

Public Function getTimeSeparator() As String
    getTimeSeparator = Application.International(xlTimeSeparator)
End Function
Public Function getDateSeparator() As String
    getDateSeparator = Application.International(xlDateSeparator)
End Function
Public Function getListSeparator() As String
    Dim ListSeparator As String, iRetVal1 As Long, iRetVal2 As Long, lpLCDataVar As String,
Position As Integer, Locale As Long
    Locale = GetUserDefaultLCID()
```

```
    iRetVal1 = GetLocaleInfo(Locale, LOCALE_SLIST, lpLCDataVar, 0)
    ListSeparator = String$(iRetVal1, 0)
    iRetVal2 = GetLocaleInfo(Locale, LOCALE_SLIST, ListSeparator, iRetVal1)
    Position = InStr(ListSeparator, Chr$(0))
    If Position > 0 Then ListSeparator = Left$(ListSeparator, Position - 1) Else ListSeparator
= vbNullString
    getListSeparator = ListSeparator
End Function

Private Sub ChangeSettingExample()  'change the setting of the character displayed as the
decimal separator.
    Call SetLocalSetting(LOCALE_SDECIMAL, ",")  'to change to ","
    Stop                                        'check your control panel to verify or use the
GetLocaleInfo API function
    Call SetLocalSetting(LOCALE_SDECIMAL, ".")  'to back change to "."
End Sub

Private Function SetLocalSetting(LC_CONST As Long, Setting As String) As Boolean
    Call SetLocaleInfo(GetUserDefaultLCID(), LC_CONST, Setting)
End Function
```

Leggi Chiamate API online: https://riptutorial.com/it/vba/topic/10569/chiamate-api

Capitolo 7: collezioni

Osservazioni

Una `Collection` è un oggetto contenitore incluso nel runtime VBA. Non sono necessari ulteriori riferimenti per poterlo utilizzare. È possibile utilizzare una `Collection` per memorizzare elementi di qualsiasi tipo di dati e consentirne il recupero tramite l'indice ordinale dell'articolo o utilizzando una chiave univoca opzionale.

Confronto delle funzionalità con array e dizionari

	Collezione	schieramento	Dizionario
Può essere ridimensionato	sì	A volte [1]	sì
Gli articoli sono ordinati	sì	sì	Sì [2]
Gli articoli sono fortemente tipizzati	No	sì	No
Gli oggetti possono essere recuperati per ordinale	sì	sì	No
Nuovi oggetti possono essere inseriti in ordinale	sì	No	No
Come determinare se esiste un oggetto	Iterare tutti gli elementi	Iterare tutti gli elementi	Iterare tutti gli elementi
Gli articoli possono essere recuperati con la chiave	sì	No	sì
Le chiavi sono sensibili al maiuscolo / minuscolo	No	N / A	Opzionale [3]
Come determinare se esiste una chiave	Gestore degli errori	N / A	`.Exists` funzione
Rimuovi tutti gli oggetti	`.Remove` e. `.Remove`	`Erase` , `ReDim`	`.RemoveAll` funzione

[1] Solo gli array dinamici possono essere ridimensionati e solo l'ultima dimensione di array multidimensionali.

[2] I `.Keys` e `.Items` sottostanti sono ordinati.

[3] Determinato dalla proprietà `.CompareMode` .

Examples

Aggiunta di elementi a una raccolta

Gli oggetti vengono aggiunti a una `Collection` chiamando il suo metodo `.Add` :

Sintassi:

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

Parametro	Descrizione
articolo	L'oggetto da conservare nella `Collection` . Questo può essere essenzialmente qualsiasi valore a cui può essere assegnata una variabile, inclusi tipi primitivi, matrici, oggetti e `Nothing` .
chiave	Opzionale. Una `String` che funge da identificatore univoco per il recupero di elementi dalla `Collection` . Se la chiave specificata esiste già nella `Collection` , verrà generato un errore di runtime 457: "Questa chiave è già associata a un elemento di questa raccolta".
prima	Opzionale. Una chiave esistente (valore `String`) *o* indice (valore numerico) per inserire l'elemento nella `Collection` . Se viene fornito un valore, il parametro *after* ***deve*** essere vuoto o un errore run-time 5: "Risulterà una chiamata o un argomento di procedura non valida". Se viene passata una chiave `String` che non esiste nella `Collection` , verrà generato un errore 5 in fase di esecuzione: "Chiamata o argomento procedura non valida". Se viene passato un indice numerico che non esiste nella `Collection` , verrà generato un errore 9 in fase di esecuzione: "Sottoscritto fuori intervallo".
dopo	Opzionale. Una chiave esistente (valore `String`) *o* indice (valore numerico) per inserire l'elemento dopo nella `Collection` . Se viene fornito un valore, il parametro *before* ***deve*** essere vuoto. Gli errori generati sono identici al parametro *precedente* .

Gli appunti:

- Le chiavi ***non*** sono sensibili al maiuscolo / minuscolo. `.Add "Bar", "Foo"` e. `.Add "Baz", "foo"` provocherà una collisione tra chiavi.
- Se nessuno dei due parametri opzionali *prima* o *dopo* viene assegnato, l'elemento verrà aggiunto dopo l'ultimo elemento nella `Collection` .
- Gli inserimenti effettuati specificando un parametro *before* o *after* modificheranno gli indici numerici dei membri esistenti per adattarli alla nuova posizione. Ciò significa che è

necessario prestare attenzione quando si effettuano inserimenti nei loop utilizzando indici numerici.

Esempio di utilizzo:

```
Public Sub Example()
    Dim foo As New Collection

    With foo
        .Add "One"            'No key. This item can only be retrieved by index.
        .Add "Two", "Second"  'Key given. Can be retrieved by key or index.
        .Add "Three", , 1     'Inserted at the start of the collection.
        .Add "Four", , , 1    'Inserted at index 2.
    End With

    Dim member As Variant
    For Each member In foo
        Debug.Print member    'Prints "Three, Four, One, Two"
    Next
End Sub
```

Rimozione di elementi da una raccolta

Gli elementi vengono rimossi da una `Collection` chiamando il suo metodo `.Remove` :

Sintassi:

```
.Remove(index)
```

Parametro	Descrizione
indice	L'elemento da rimuovere dalla `Collection` . Se il valore passato è un tipo numerico o `Variant` con un sottotipo numerico, verrà interpretato come indice numerico. Se il valore passato è una `String` o `Variant` contenente una stringa, verrà interpretata come una chiave. Se viene passata una chiave stringa che non esiste nella `Collection` , verrà generato un errore 5 in fase di esecuzione: "Chiamata o argomento procedura non valida". Se viene passato un indice numerico che non esiste nella `Collection` , verrà generato un errore 9 in fase di esecuzione: "Sottoscritto fuori intervallo".

Gli appunti:

- La rimozione di un oggetto da una `Collection` cambierà gli indici numerici di tutti gli elementi dopo di essa nella `Collection` . `For` cicli che utilizzano indici numerici e rimuovi elementi, è necessario eseguire l'operazione *all'indietro* (`Step -1`) per evitare eccezioni degli indici e articoli saltati.
- Articoli ***non*** dovrebbero generalmente essere rimossi da una `Collection` all'interno di un `For Each` ciclo come può dare risultati imprevedibili.

Esempio di utilizzo:

```
Public Sub Example()
    Dim foo As New Collection

    With foo
        .Add "One"
        .Add "Two", "Second"
        .Add "Three"
        .Add "Four"
    End With

    foo.Remove 1            'Removes the first item.
    foo.Remove "Second"     'Removes the item with key "Second".
    foo.Remove foo.Count    'Removes the last item.

    Dim member As Variant
    For Each member In foo
        Debug.Print member  'Prints "Three"
    Next
End Sub
```

Ottenere il numero di oggetti di una collezione

Il numero di elementi in una `Collection` può essere ottenuto chiamando la sua funzione `.Count` :

Sintassi:

```
.Count()
```

Esempio di utilizzo:

```
Public Sub Example()
    Dim foo As New Collection

    With foo
        .Add "One"
        .Add "Two"
        .Add "Three"
        .Add "Four"
    End With

    Debug.Print foo.Count   'Prints 4
End Sub
```

Recupero di oggetti da una collezione

Gli oggetti possono essere recuperati da una `Collection` chiamando la funzione `.Item` .

Sintassi:

```
.Item(index)
```

Parametro	Descrizione
indice	L'oggetto da recuperare dalla `Collection` . Se il valore passato è un tipo numerico o `Variant` con un sottotipo numerico, verrà interpretato come indice numerico. Se il valore passato è una `String` o `Variant` contenente una stringa, verrà interpretata come una chiave. Se viene passata una chiave stringa che non esiste nella `Collection` , verrà generato un errore 5 in fase di esecuzione: "Chiamata o argomento procedura non valida". Se viene passato un indice numerico che non esiste nella `Collection` , verrà generato un errore 9 in fase di esecuzione: "Sottoscritto fuori intervallo".

Gli appunti:

- `.Item` è il membro predefinito di `Collection` . Ciò consente flessibilità nella sintassi come dimostrato nell'uso di esempio riportato di seguito.
- Gli indici numerici sono basati su 1.
- Le chiavi ***non*** sono sensibili al maiuscolo / minuscolo. `.Item("Foo")` e `.Item("foo")` riferiscono alla stessa chiave.
- Il parametro *index* ***non*** viene convertito implicitamente in un numero da `String` o viceversa. È del tutto possibile che `.Item(1)` e `.Item("1")` facciano riferimento a diversi elementi della `Collection` .

Esempio di utilizzo (indici):

```
Public Sub Example()
    Dim foo As New Collection

    With foo
        .Add "One"
        .Add "Two"
        .Add "Three"
        .Add "Four"
    End With

    Dim index As Long
    For index = 1 To foo.Count
        Debug.Print foo.Item(index) 'Prints One, Two, Three, Four
    Next
End Sub
```

Esempio di utilizzo (chiavi):

```
Public Sub Example()
    Dim keys() As String
    keys = Split("Foo,Bar,Baz", ",")
    Dim values() As String
    values = Split("One,Two,Three", ",")

    Dim foo As New Collection
    Dim index As Long
    For index = LBound(values) To UBound(values)
```

```
        foo.Add values(index), keys(index)
    Next

    Debug.Print foo.Item("Bar") 'Prints "Two"
End Sub
```

Esempio di utilizzo (sintassi alternativa):

```
Public Sub Example()
    Dim foo As New Collection

    With foo
        .Add "One", "Foo"
        .Add "Two", "Bar"
        .Add "Three", "Baz"
    End With

    'All lines below print "Two"
    Debug.Print foo.Item("Bar")     'Explicit call syntax.
    Debug.Print foo("Bar")          'Default member call syntax.
    Debug.Print foo!Bar             'Bang syntax.
End Sub
```

Si noti che la sintassi bang (`!`) È consentita perché `.Item` è il membro predefinito e può accettare un singolo argomento `String` . L'utilità di questa sintassi è discutibile.

Determinazione della presenza di una chiave o di un oggetto in una raccolta

chiavi

A differenza di Scripting.Dictionary , una `Collection` non ha un metodo per determinare se esiste una determinata chiave *o* un modo per recuperare le chiavi che sono presenti nella `Collection` . L'unico metodo per determinare se una chiave è presente è utilizzare il gestore degli errori:

```
Public Function KeyExistsInCollection(ByVal key As String, _
                                      ByRef container As Collection) As Boolean
    With Err
        If container Is Nothing Then .Raise 91
        On Error Resume Next
        Dim temp As Variant
        temp = container.Item(key)
        On Error GoTo 0

        If .Number = 0 Then
            KeyExistsInCollection = True
        ElseIf .Number <> 5 Then
            .Raise .Number
        End If
    End With
End Function
```

Elementi

L'unico modo per determinare se un elemento è contenuto in una `Collection` è iterare sulla `Collection` fino a quando l'elemento non si trova. Tieni presente che poiché una `Collection` può contenere primitive o oggetti, è necessaria una gestione aggiuntiva per evitare errori di runtime durante i confronti:

```
Public Function ItemExistsInCollection(ByRef target As Variant, _
                                       ByRef container As Collection) As Boolean
    Dim candidate As Variant
    Dim found As Boolean

    For Each candidate In container
        Select Case True
            Case IsObject(candidate) And IsObject(target)
                found = candidate Is target
            Case IsObject(candidate), IsObject(target)
                found = False
            Case Else
                found = (candidate = target)
        End Select
        If found Then
            ItemExistsInCollection = True
            Exit Function
        End If
    Next
End Function
```

Cancellare tutti gli oggetti da una collezione

Il modo più semplice per cancellare tutti gli elementi da una `Collection` è semplicemente sostituirlo con una nuova `Collection` e lasciare che quello vecchio vada fuori ambito:

```
Public Sub Example()
    Dim foo As New Collection

    With foo
        .Add "One"
        .Add "Two"
        .Add "Three"
    End With

    Debug.Print foo.Count   'Prints 3
    Set foo = New Collection
    Debug.Print foo.Count   'Prints 0
End Sub
```

Tuttavia, se sono presenti più riferimenti alla `Collection` , questo metodo fornirà solo una `Collection` vuota *per la variabile che viene assegnata* .

```
Public Sub Example()
    Dim foo As New Collection
    Dim bar As Collection

    With foo
        .Add "One"
        .Add "Two"
```

```
        .Add "Three"
    End With

    Set bar = foo
    Set foo = New Collection

    Debug.Print foo.Count   'Prints 0
    Debug.Print bar.Count   'Prints 3
End Sub
```

In questo caso, il modo più semplice per cancellare i contenuti consiste nel ripetere il numero di elementi nella `Collection` e rimuovere ripetutamente l'elemento più basso:

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

Leggi collezioni online: https://riptutorial.com/it/vba/topic/5838/collezioni

Capitolo 8: Commenti

Osservazioni

Blocchi di commento

Se è necessario commentare o decommentare più righe contemporaneamente, è possibile utilizzare i pulsanti **Modifica barra degli strumenti** IDE:

Blocco commenti : aggiunge un singolo apostrofo all'inizio di tutte le righe selezionate

Blocco commento : rimuove il primo apostrofo dall'inizio di tutte le righe selezionate

Commenti a più righe Molte altre lingue supportano commenti a blocchi su più righe, ma VBA consente solo commenti a riga singola.

Examples

Commenti apostrofo

Un commento è contrassegnato da un apostrofo (`'`) e ignorato quando il codice viene eseguito. I commenti aiutano a spiegare il tuo codice ai futuri lettori, incluso te stesso.

Poiché tutte le righe che iniziano con un commento vengono ignorate, possono anche essere utilizzate per impedire l'esecuzione del codice (mentre esegui il debug o il refactoring). Posizionando un apostrofo `'` prima che il tuo codice lo trasformi in un commento. (Questo è chiamato *commentando* la linea.)

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

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

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

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

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

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

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

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

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

Commenti REM

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

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

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

End Sub

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

Leggi Commenti online: https://riptutorial.com/it/vba/topic/2059/commenti

Capitolo 9: Compilazione condizionale

Examples

Modifica del comportamento del codice in fase di compilazione

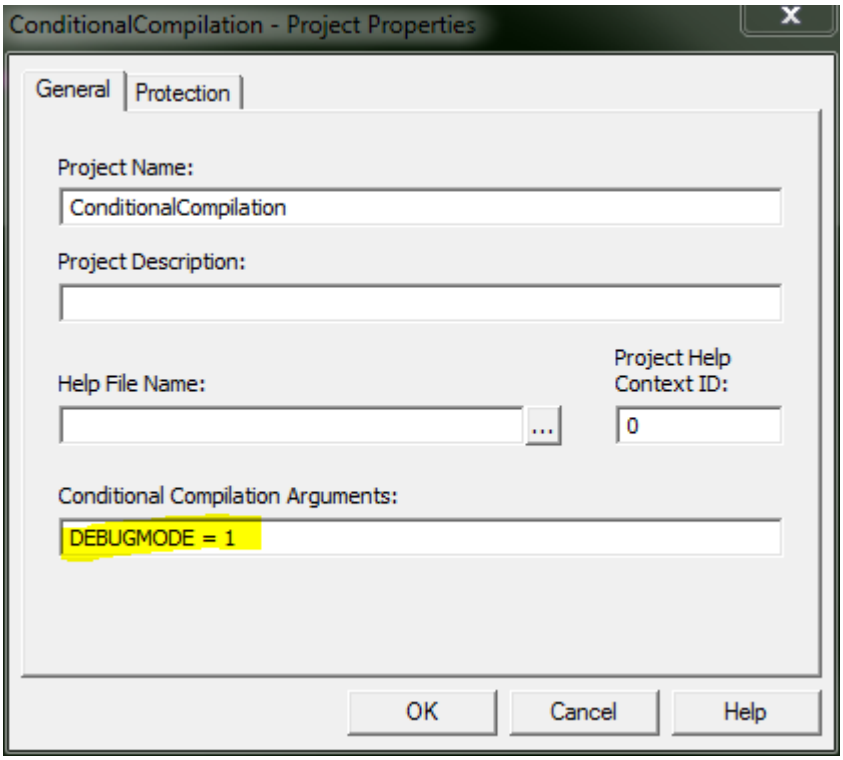
La direttiva `#Const` viene utilizzata per definire una costante del preprocessore personalizzata. Questi possono essere successivamente utilizzati da `#If` per controllare quali blocchi di codice vengono compilati ed eseguiti.

```
#Const DEBUGMODE = 1

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

Ciò si traduce nel valore di `filepath` impostato su `"C:\Users\UserName\Path\To\File.txt"` . La rimozione della riga `#Const` o la modifica di `#Const DEBUGMODE = 0` comportano l'impostazione del `filepath` su `"\\server\share\path\to\file.txt"` .

#Const Scope

La direttiva `#Const` è efficace solo per un singolo file di codice (modulo o classe). Deve essere dichiarato per ogni file in cui desideri utilizzare la costante personalizzata. In alternativa, puoi dichiarare un `#Const` globalmente per il tuo progetto andando su Strumenti >> Proprietà del progetto [Nome del tuo progetto]. Questo farà apparire la finestra di dialogo delle proprietà del progetto dove inseriremo la dichiarazione costante. Nella casella "Argomenti di compilazione condizionale", digitare `[constName] = [value]` . Puoi inserire più di una costante separandole con due punti, come `[constName1] = [value1] : [constName2] = [value2]` .



Costanti predefinite

Alcune costanti di compilazione sono già predefinite. Quelli esistenti dipenderanno dalla versione della versione di Office in cui VBA è in esecuzione. Si noti che Vba7 è stato introdotto insieme a Office 2010 per supportare le versioni a 64 bit di Office.

Costante	16 bit	32 bit	64 bit
VBA6	falso	Se Vba6	falso
Vba7	falso	Se Vba7	Vero
Win16	Vero	falso	falso
Win32	falso	Vero	Vero
Win64	falso	falso	Vero
Mac	falso	Se Mac	Se Mac

Si noti che Win64 / Win32 si riferisce alla versione di Office, non alla versione di Windows. Ad esempio Win32 = TRUE in Office a 32 bit, anche se il sistema operativo è una versione a 64 bit di Windows.

Utilizzo di Declare Imports che funzionano su tutte le versioni di Office

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

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

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

Questo può essere semplificato un po 'a seconda delle versioni di ufficio che è necessario supportare. Ad esempio, non molte persone stanno ancora supportando le versioni a 16 bit di Office. L'ultima versione di Office 16 bit era la versione 4.3, rilasciata nel 1994 , quindi la seguente dichiarazione è sufficiente per quasi tutti i casi moderni (incluso Office 2007).

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

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

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

Se non è necessario supportare elementi precedenti a Office 2010, questa dichiarazione funziona correttamente.

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

#If Win64 Then
    Declare PtrSafe Function GetFoo64 Lib "exampleLib32" () As LongLong
#Else
    Declare PtrSafe Function GetFoo Lib "exampleLib32" () As Long
#End If
```

Leggi Compilazione condizionale online: https://riptutorial.com/it/vba/topic/3364/compilazione-condizionale

Capitolo 10: Concatenazione di stringhe

Osservazioni

Le stringhe possono essere concatenate o riunite utilizzando uno o più operatori di concatenazione `&` .

Gli array di stringhe possono anche essere concatenati usando la funzione `Join` e fornendo una stringa (che può essere di lunghezza zero) da utilizzare tra ogni elemento dell'array.

Examples

Concatena le stringhe usando l'operatore &

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

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

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

Concatena una serie di stringhe usando la funzione Join

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

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

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

Leggi Concatenazione di stringhe online: https://riptutorial.com/it/vba/topic/3580/concatenazione-di-stringhe

Capitolo 11: Convenzioni di denominazione

Examples

Nomi variabili

Le variabili contengono dati. Assegnagli un nome dopo quello per cui sono stati usati, **non dopo il loro tipo di dati** o ambito, usando un **nome** . Se ti senti obbligato a *numerare le* tue variabili (es. `thing1, thing2, thing3`), allora considera di usare una struttura dati appropriata (per esempio una matrice, una `Collection` o un `Dictionary`).

I nomi di variabili che rappresentano un *insieme* di valori iterabile, ad esempio un array, una `Collection` , un `Dictionary` o un `Range` di celle, dovrebbero essere plurali.

Alcune convenzioni di denominazione VBA comuni vanno così:

Per variabili a livello di procedura :

`camelCase`

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

    Dim procedureVariable As Long
    Dim someOtherVariable As String

End Sub
```

Per variabili a livello di modulo:

`PascalCase`

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

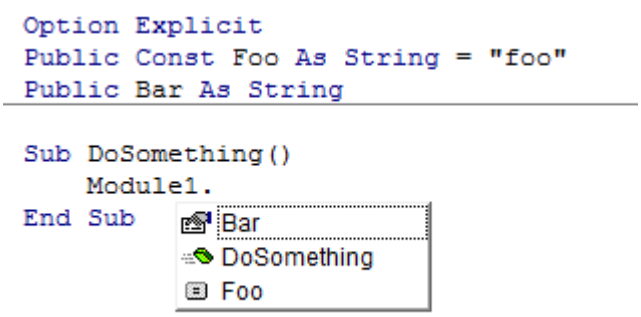
Per costanti:

`SHOUTY_SNAKE_CASE` è comunemente usato per differenziare le costanti dalle variabili:

```
Public Const GLOBAL_CONSTANT As String = "Project Version #1.000.000.001"
Private Const MODULE_CONSTANT As String = "Something relevant to this Module"

Public Sub SomeProcedure()

    Const PROCEDURE_CONSTANT As Long = 10

End Sub
```

Tuttavia, i nomi `PascalCase` rendono il codice più pulito e sono altrettanto validi, dato che

IntelliSense utilizza icone diverse per variabili e costanti:



Notazione ungherese

> Assegnagli un nome dopo quello per cui sono utilizzati, **non dopo il tipo di dati** o l'ambito.

"Notazione ungherese rende più facile vedere quale sia il tipo di variabile"

Se scrivi il tuo codice, come le procedure, aderisci al *Principio di Responsabilità Unica* (come dovrebbe), non dovresti mai guardare una schermata di dichiarazioni variabili nella parte superiore di qualsiasi procedura; dichiarare le variabili il più vicino possibile al loro primo utilizzo e il loro tipo di dati sarà sempre in bella vista se le dichiarate con un tipo esplicito. La scorciatoia `Ctrl` + `i` del VBE può essere usata anche per mostrare il tipo di una variabile in un suggerimento.

L'uso di una variabile è molto più utile del suo tipo di dati, *specialmente* in un linguaggio come VBA che converte felicemente e implicitamente un tipo in un altro secondo necessità.

Considera `iFile` e `strFile` in questo esempio:

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

    On Error GoTo CleanFail

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

    bRetVal = True

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

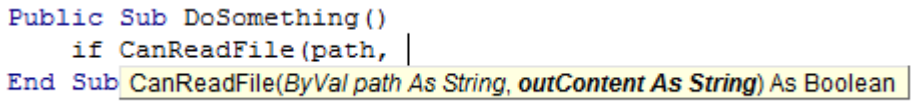
Confrontare con:

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

    Dim handle As Integer
    handle = FreeFile

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

    Dim result As Boolean
    result = True

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

`strData` è passato da `ByRef` nell'esempio in alto, ma accanto al fatto che siamo abbastanza fortunati da vedere che è passato *esplicitamente* come tale, non c'è alcuna indicazione che `strData` sia effettivamente *restituito* dalla funzione.

L'esempio in basso lo `outContent` ; questo prefisso `out` è quello per cui è stata inventata la notazione ungherese: per aiutare a chiarire a *cosa serve una variabile* , in questo caso identificarla chiaramente come un parametro "out".

Questo è utile, perché IntelliSense di per sé non visualizza `ByRef` , anche quando il parametro è passato *esplicitamente* per riferimento:



Che porta a...

Ungherese fatto bene

La notazione ungherese in origine non aveva nulla a che fare con i tipi variabili . In effetti, la notazione ungherese *fatta correttamente* è effettivamente utile. Considera questo piccolo esempio (`ByVal` e `As Integer` rimossi per brevità):

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

Confrontare con:

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

`src` e `dst` sono i prefissi della *notazione ungherese* qui e trasmettono informazioni *utili* che non possono altrimenti essere dedotte dai nomi dei parametri o IntelliSense che ci mostra il tipo dichiarato.

Ovviamente c'è un modo migliore per trasmettere tutto, usando *un'astrazione* appropriata e parole reali che possono essere pronunciate a voce alta e avere un senso - come esempio forzato:

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

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

Nomi delle procedure

Le procedure *fanno qualcosa* . Chiamali dopo quello che stanno facendo, usando un **verbo** . Se non è possibile nominare con precisione una procedura, è probabile che la procedura stia *facendo troppe cose* e debba essere suddivisa in procedure più piccole e più specializzate.

Alcune convenzioni di denominazione VBA comuni vanno così:

Per tutte le procedure:

`PascalCase`

```
Public Sub DoThing()

End Sub

Private Function ReturnSomeValue() As [DataType]

End Function
```

Per le procedure del gestore di eventi:

`ObjectName_EventName`

```
Public Sub Workbook_Open()

End Sub

Public Sub Button1_Click()

End Sub
```

I gestori di eventi sono di solito nominati automaticamente dal VBE; rinominandole senza rinominare l'oggetto e / o l'evento gestito interromperà il codice: il codice verrà eseguito e compilato, ma la procedura del gestore sarà orfana e non verrà mai eseguita.

Membri booleani

Considera una funzione di ritorno booleano:

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

Confrontare con:

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

Il prefisso `Can` *ha* lo stesso scopo del prefisso `b` : identifica il valore di ritorno della funzione come `Boolean` . Ma `Can` leggere meglio di `b` :

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

Rispetto a:

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

Prendi in considerazione l'uso di prefissi come `Can` , `Is` o `Has` davanti ai membri di ritorno booleani (funzioni e proprietà), ma solo quando aggiunge valore. Questo è conforme alle attuali linee guida per la denominazione di Microsoft .

Leggi Convenzioni di denominazione online: https://riptutorial.com/it/vba/topic/1184/convenzioni-di-denominazione

Capitolo 12: Convertire altri tipi di stringhe

Osservazioni

VBA convertirà implicitamente alcuni tipi in string, se necessario e senza alcun lavoro aggiuntivo sulla parte del programmatore, ma VBA fornisce anche un certo numero di funzioni di conversione stringhe esplicite e puoi anche scriverne di tue.

Tre delle funzioni più utilizzate sono `CStr` , `Format` e `StrConv` .

Examples

Utilizzare CStr per convertire un tipo numerico in una stringa

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

Usa Formato per convertire e formattare un tipo numerico come una stringa

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

Utilizzare StrConv per convertire una matrice di byte di caratteri a byte singolo in una stringa

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

Converti implicitamente un array di byte di caratteri multi-byte in una stringa

```
'Declare an array of bytes, assign multi-byte character codes, and convert to a string
Dim multiByteChars(9) As Byte
multiByteChars(0) = 87
multiByteChars(1) = 0
multiByteChars(2) = 111
```

```
multiByteChars(3) = 0
multiByteChars(4) = 114
multiByteChars(5) = 0
multiByteChars(6) = 108
multiByteChars(7) = 0
multiByteChars(8) = 100
multiByteChars(9) = 0

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

Leggi Convertire altri tipi di stringhe online: https://riptutorial.com/it/vba/topic/3467/convertire-altri-tipi-di-stringhe

Capitolo 13: Copia, restituisce e passa array

Examples

Copia di array

È possibile copiare un array VBA in un array dello stesso tipo usando l'operatore `=` . Gli array devono essere dello stesso tipo, altrimenti il codice genererà un errore di compilazione "Can not assign to array".

```
Dim source(0 to 2) As Long
Dim destinationLong() As Long
Dim destinationDouble() As Double

destinationLong = source     ' copies contents of source into destinationLong
destinationDouble = source   ' does not compile
```

L'array sorgente può essere fisso o dinamico, ma l'array di destinazione deve essere dinamico. Cercando di copiare su un array fisso si genera un errore di compilazione "Can not assign to array". Tutti i dati preesistenti nell'array di ricezione vengono persi e i relativi limiti e dimensioni vengono modificati allo stesso array di origine.

```
Dim source() As Long
ReDim source(0 To 2)

Dim fixed(0 To 2) As Long
Dim dynamic() As Long

fixed = source   ' does not compile
dynamic = source ' does compile

Dim dynamic2() As Long
ReDim dynamic2(0 to 6, 3 to 99)

dynamic2 = source ' dynamic2 now has dimension (0 to 2)
```

Una volta eseguita la copia, i due array sono separati in memoria, ovvero le due variabili non sono riferimenti agli stessi dati sottostanti, quindi le modifiche apportate a un array non vengono visualizzate nell'altro.

```
Dim source(0 To 2) As Long
Dim destination() As Long

source(0) = 3
source(1) = 1
source(2) = 4

destination = source
destination(0) = 2

Debug.Print source(0); source(1); source(2)                ' outputs: 3 1 4
```

```
Debug.Print destination(0); destination(1); destination(2) ' outputs: 2 1 4
```

Copia di matrici di oggetti

Con le matrici di oggetti vengono copiati i *riferimenti* a quegli oggetti, non gli oggetti stessi. Se viene apportata una modifica a un oggetto in una matrice, sembrerà che sia stata modificata nell'altra matrice: entrambi fanno riferimento allo stesso oggetto. Tuttavia, l'impostazione di un elemento su un oggetto diverso in un array non lo imposterà sull'altro array.

```
Dim source(0 To 2) As Range
Dim destination() As Range

Set source(0) = Range("A1"): source(0).Value = 3
Set source(1) = Range("A2"): source(1).Value = 1
Set source(2) = Range("A3"): source(2).Value = 4

destination = source

Set destination(0) = Range("A4")   'reference changed in destination but not source

destination(0).Value = 2           'affects an object only in destination
destination(1).Value = 5           'affects an object in both source and destination

Debug.Print source(0); source(1); source(2)                  ' outputs 3 5 4
Debug.Print destination(0); destination(1); destination(2)   ' outputs 2 5 4
```

Varianti contenenti una matrice

È anche possibile copiare un array in e da una variabile variante. Quando si copia da una variante, deve contenere una matrice dello stesso tipo dell'array ricevente, altrimenti genererà un errore di runtime "Tipo non corrispondente".

```
Dim var As Variant
Dim source(0 To 2) As Range
Dim destination() As Range

var = source
destination = var

var = 5
destination = var  ' throws runtime error
```

Restituzione di matrici da funzioni

Una funzione in un modulo normale (ma non in un modulo di classe) può restituire un array inserendo `()` dopo il tipo di dati.

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

    outputArray(0) = 3
```

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

    arrayOfPiDigits = outputArray
End Function
```

Il risultato della funzione può quindi essere inserito in una matrice dinamica dello stesso tipo o di una variante. È anche possibile accedere agli elementi direttamente utilizzando una seconda serie di parentesi, tuttavia questa funzione chiamerà la funzione ogni volta, quindi è meglio memorizzare i risultati in una nuova matrice se si prevede di utilizzarli più di una volta

```
Sub arrayExample()

    Dim destination() As Long
    Dim var As Variant

    destination = arrayOfPiDigits()
    var = arrayOfPiDigits

    Debug.Print destination(0)          ' outputs 3
    Debug.Print var(1)                  ' outputs 1
    Debug.Print arrayOfPiDigits()(2)    ' outputs 4

End Sub
```

Si noti che ciò che viene restituito è in realtà una copia dell'array all'interno della funzione, non un riferimento. Quindi se la funzione restituisce il contenuto di una matrice statica, i suoi dati non possono essere modificati dalla procedura chiamante.

Emissione di una matrice tramite un argomento di output

Normalmente è buona pratica di codifica che gli argomenti di una procedura siano input e output tramite il valore di ritorno. Tuttavia, le limitazioni di VBA talvolta rendono necessaria una procedura per l'output di dati tramite un argomento `ByRef` .

Emissione su una matrice fissa

```
Sub threePiDigits(ByRef destination() As Long)
    destination(0) = 3
    destination(1) = 1
    destination(2) = 4
End Sub

Sub printPiDigits()
    Dim digits(0 To 2) As Long

    threePiDigits digits
    Debug.Print digits(0); digits(1); digits(2) ' outputs 3 1 4
End Sub
```

Emissione di una matrice da un metodo di classe

Un argomento di output può anche essere utilizzato per generare un array da un metodo / procedimento in un modulo di classe

```
' Class Module 'MathConstants'
Sub threePiDigits(ByRef destination() As Long)
    ReDim destination(0 To 2)

    destination(0) = 3
    destination(1) = 1
    destination(2) = 4
End Sub

' Standard Code Module
Sub printPiDigits()
    Dim digits() As Long
    Dim mathConsts As New MathConstants

    mathConsts.threePiDigits digits
    Debug.Print digits(0); digits(1); digits(2) ' outputs 3 1 4
End Sub
```

Passare gli array alle procedure

Le matrici possono essere passate alle procedure mettendo `()` dopo il nome della variabile di matrice.

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

Le matrici *devono* essere passate per riferimento. Se non viene specificato alcun meccanismo di passaggio, ad esempio `myFunction(arr())` , VBA assumerà `ByRef` per impostazione predefinita, tuttavia è buona pratica di codifica renderlo esplicito. Provando a passare un array in base al valore, ad esempio `myFunction(ByVal arr())` comporterà un errore di compilazione "Argomento array deve essere ByRef" (o un errore di compilazione "Errore di sintassi" se `Auto Syntax Check` non è selezionato nelle opzioni VBE) .

Passare per riferimento significa che eventuali modifiche alla matrice verranno mantenute nella procedura di chiamata.

```
Sub testArrayPassing()
    Dim source(0 To 1) As Long
    source(0) = 3
    source(1) = 1

    Debug.Print doubleAndSum(source)  ' outputs 8
    Debug.Print source(0); source(1)  ' outputs 6 2
End Sub

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

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

Se vuoi evitare di cambiare la matrice originale, fai attenzione a scrivere la funzione in modo che non cambi alcun elemento.

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

In alternativa, crea una copia funzionante della matrice e lavora con la copia.

```
Function doubleAndSum(ByRef arr() As Long)
    Dim copyOfArr() As Long
    copyOfArr = arr

    copyOfArr(0) = copyOfArr(0) * 2
    copyOfArr(1) = copyOfArr(1) * 2

    doubleAndSum = copyOfArr(0) + copyOfArr(1)
End Function
```

Leggi Copia, restituisce e passa array online: https://riptutorial.com/it/vba/topic/9069/copia--restituisce-e-passa-array

Capitolo 14: Creare una procedura

Examples

Introduzione alle procedure

Un `Sub` è una procedura che esegue un'attività specifica ma non restituisce un valore specifico.

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

Se non viene specificato alcun modificatore di accesso, una procedura è `Public` per impostazione predefinita.

Una `Function` è una procedura a cui vengono dati dati e restituisce un valore, idealmente senza effetti collaterali globali o di ambito modulo.

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

Una `Property` è una procedura che *incapsula i* dati del modulo. Una proprietà può avere fino a 3 di accesso: `Get` per restituire un valore di riferimento o un oggetto, `Let` per assegnare un valore, e / o `Set` per assegnare un riferimento a un oggetto.

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

Le proprietà vengono solitamente utilizzate nei moduli di classe (sebbene siano consentite anche nei moduli standard), esponendo l'accesso a dati altrimenti inaccessibili al codice chiamante. Una proprietà che espone solo un accessorio `Get` è "di sola lettura"; una proprietà che esporrebbe solo un accessorio `Let` e / o `Set` è "di sola scrittura". Le proprietà di sola scrittura non sono considerate una buona pratica di programmazione: se il codice client può *scrivere* un valore, dovrebbe essere in grado di *leggerlo* . Considerare l'implementazione di una procedura `Sub` invece di creare una proprietà di sola scrittura.

Restituzione di un valore

Una procedura `Function` o `Property Get` può (e dovrebbe!) Restituire un valore al proprio chiamante. Questo viene fatto assegnando l'identificatore della procedura:

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

Funzione con esempi

Come detto sopra, le funzioni sono procedure più piccole che contengono piccoli pezzi di codice che possono essere ripetitivi all'interno di una procedura.

Le funzioni vengono utilizzate per ridurre la ridondanza nel codice.

Simile a una procedura, una funzione può essere dichiarata con o senza un elenco di argomenti.

La funzione è dichiarata come un tipo di ritorno, poiché tutte le funzioni restituiscono un valore. Il nome e la variabile di ritorno di una funzione sono gli stessi.

1. Funzione con parametro:

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

2. Funzione senza parametro:

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

La funzione può essere chiamata in vari modi all'interno di una funzione. Poiché una funzione dichiarata con un tipo di ritorno è fondamentalmente una variabile. è usato simile a una variabile.

Chiamate funzionali:

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

Inoltre, la funzione può essere utilizzata anche come condizioni per if e altre istruzioni condizionali.

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

Ulteriori ulteriori funzioni possono avere modificatori come Per ref e Per val per i loro argomenti.

Leggi Creare una procedura online: https://riptutorial.com/it/vba/topic/1474/creare-una-procedura

Capitolo 15: CreateObject vs. GetObject

Osservazioni

Nel modo più semplice, `CreateObject` crea un'istanza di un oggetto, mentre `GetObject` ottiene un'istanza esistente di un oggetto. Determinare se un oggetto può essere creato o ottenuto dipenderà dalla sua proprietà di Instancing . Alcuni oggetti sono SingleUse (ad esempio, WMI) e non possono essere creati se già esistono. Altri oggetti (ad esempio Excel) sono MultiUse e consentono l'esecuzione di più istanze contemporaneamente. Se un'istanza di un oggetto non esiste già e si tenta `GetObject` , si riceverà il seguente messaggio intercettabile: `Run-time error '429': ActiveX component can't create object .`

GetObject richiede che almeno uno di questi due parametri opzionali sia presente:

1. *Pathname* - Variant (String): il percorso completo, incluso il nomefile, del file che contiene l'oggetto. Questo parametro è facoltativo, ma è necessario *Class* se *Pathname* è omesso.
2. *Class* - Variant (String): stringa che rappresenta la definizione formale (Application e ObjectType) dell'oggetto. *La classe* è richiesta se *Pathname* è omesso.

CreateObject ha un parametro obbligatorio e un parametro facoltativo:

1. *Class* - Variant (String): stringa che rappresenta la definizione formale (Application e ObjectType) dell'oggetto. *La classe* è un parametro richiesto.
2. *Servername* - Variant (String): il nome del computer remoto su cui verrà creato l'oggetto. Se omesso, l'oggetto verrà creato sul computer locale.

La classe è sempre composta da due parti sotto forma di `Application.ObjectType` :

1. *Applicazione* : il nome dell'applicazione a cui appartiene l'oggetto. |
2. *Tipo di oggetto* : il tipo di oggetto che si sta creando. |

Alcune classi di esempio sono:

1. Word.Application
2. Foglio Excel
3. Scripting.FileSystemObject

Examples

Dimostrazione di GetObject e CreateObject

Funzione MSDN-GetObject

> Restituisce un riferimento a un oggetto fornito da un componente ActiveX.

Utilizzare la funzione GetObject quando è presente un'istanza corrente dell'oggetto o se si desidera creare l'oggetto con un file già caricato. Se non esiste un'istanza corrente e non si desidera che l'oggetto venga avviato con un file caricato, utilizzare la funzione CreateObject.

```
Sub CreateVSGet()
    Dim ThisXLApp As Excel.Application 'An example of early binding
    Dim AnotherXLApp As Object 'An example of late binding
    Dim ThisNewWB As Workbook
    Dim AnotherNewWB As Workbook
    Dim wb As Workbook

    'Get this instance of Excel
    Set ThisXLApp = GetObject(ThisWorkbook.Name).Application
    'Create another instance of Excel
    Set AnotherXLApp = CreateObject("Excel.Application")
    'Make the 2nd instance visible
    AnotherXLApp.Visible = True
    'Add a workbook to the 2nd instance
    Set AnotherNewWB = AnotherXLApp.Workbooks.Add
    'Add a sheet to the 2nd instance
    AnotherNewWB.Sheets.Add

    'You should now have 2 instances of Excel open
    'The 1st instance has 1 workbook: Book1
    'The 2nd instance has 1 workbook: Book2

    'Lets add another workbook to our 1st instance
    Set ThisNewWB = ThisXLApp.Workbooks.Add
    'Now loop through the workbooks and show their names
    For Each wb In ThisXLApp.Workbooks
        Debug.Print wb.Name
    Next
    'Now the 1st instance has 2 workbooks: Book1 and Book3
    'If you close the first instance of Excel,
    'Book1 and Book3 will close, but book2 will still be open

End Sub
```

Leggi CreateObject vs. GetObject online: https://riptutorial.com/it/vba/topic/7729/createobject-vs--getobject

Capitolo 16: Creazione di una classe personalizzata

Osservazioni

Questo articolo mostrerà come creare una classe personalizzata completa in VBA. Utilizza l'esempio di un oggetto `DateRange` , perché una data di inizio e di fine viene spesso passata insieme alle funzioni.

Examples

Aggiunta di una proprietà a una classe

Una procedura `Property` è una serie di istruzioni che recupera o modifica una proprietà personalizzata su un modulo.

Esistono tre tipi di accessori di proprietà:

1. Una procedura `Get` che restituisce il valore di una proprietà.
2. Un `Let` procedura che assegna un (non `Object` value) per un oggetto.
3. Una procedura `Set` che assegna un riferimento `Object` .

Gli accessor di proprietà sono spesso definiti in coppia, utilizzando sia un `Get` e `Let` / `Set` per ogni proprietà. Una proprietà con solo una procedura `Get` sarebbe di sola lettura, mentre una proprietà con solo una procedura `Let` / `Set` sarebbe di sola scrittura.

Nell'esempio seguente, vengono definiti quattro accessor di proprietà per la classe `DateRange` :

1. `StartDate` (*lettura / scrittura*). Valore data che rappresenta la data precedente in un intervallo. Ogni procedura utilizza il valore della variabile del modulo, `mStartDate` .
2. `EndDate` (*lettura / scrittura*). Valore data che rappresenta la data successiva in un intervallo. Ogni procedura utilizza il valore della variabile del modulo, `mEndDate` .
3. `DaysBetween` (`DaysBetween` *lettura*). Valore intero calcolato che rappresenta il numero di giorni tra le due date. Poiché esiste solo una procedura `Get` , questa proprietà non può essere modificata direttamente.
4. `RangeToCopy` (*solo scrittura*). Una procedura `Set` utilizzata per copiare i valori di un oggetto `DateRange` esistente.

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

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

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

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

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

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

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

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

End Property
```

Aggiunta di funzionalità a una classe

Qualsiasi `Sub` , `Function` o `Property` all'interno di un modulo di classe può essere richiamata precedendo la chiamata con un riferimento a un oggetto:

```
Object.Procedure
```

In una classe `DateRange` , è possibile utilizzare una `Sub` per aggiungere un numero di giorni alla data di fine:

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

Una `Function` potrebbe restituire l'ultimo giorno del prossimo mese (nota che `GetFirstDayOfMonth` non sarebbe visibile al di fuori della classe perché è privato):

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

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

Le procedure possono accettare argomenti di qualsiasi tipo, compresi i riferimenti agli oggetti della classe in fase di definizione.

L'esempio seguente verifica se l'oggetto `DateRange` corrente ha una data di inizio e una data di fine che include la data di inizio e di fine di un altro oggetto `DateRange` .

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

Notare l'uso della notazione `Me` come modo per accedere al valore dell'oggetto che esegue il codice.

Scopo del modulo di classe, istanziazione e riutilizzo

Per impostazione predefinita, un nuovo modulo di classe è una classe privata, quindi è disponibile *solo* per l'istanziazione e l'utilizzo all'interno del VBProject in cui è definito. Puoi dichiarare, creare un'istanza e utilizzare la classe ovunque nello *stesso* progetto:

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

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

Ma spesso scrivi classi che ti piacerebbe usare in altri progetti *senza* copiare il modulo tra i progetti. Se si definisce una classe denominata `List` in `ProjectA` e si desidera utilizzare tale classe in `ProjectB` , sarà necessario eseguire 4 azioni:

1. Modificare la proprietà di `PublicNotCreatable` della classe `List` in `ProjectA` nella finestra Proprietà, da `Private` a `PublicNotCreatable`

2. Creare una funzione "factory" pubblica in `ProjectA` che crea e restituisce un'istanza di una classe `List` . In genere, la funzione factory include argomenti per l'inizializzazione dell'istanza di classe. La funzione factory è necessaria perché la classe può essere utilizzata da `ProjectB` ma `ProjectB` non può creare direttamente un'istanza della classe `ProjectA` .

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

3. In `ProjectB` aggiungi un riferimento a `ProjectA` usando il menu `Tools..References...` .

4. In `ProjectB` , dichiarare una variabile e assegnargli un'istanza di `List` utilizzando la funzione factory di `ProjectA`

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

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

Leggi Creazione di una classe personalizzata online:
https://riptutorial.com/it/vba/topic/4464/creazione-di-una-classe-personalizzata

Capitolo 17: Data Manipolazione del tempo

Examples

Calendario

VBA supporta 2 calendari: Gregorian e Hijri

La proprietà `Calendar` viene utilizzata per modificare o visualizzare il calendario corrente.

I 2 valori per il calendario sono:

Valore	Costante	Descrizione
0	vbCalGreg	Calendario gregoriano (predefinito)
1	vbCalHijri	Calendario Hijri

Esempio

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

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

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

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

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

Questo Sub stampa quanto segue;

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

Recupera il sistema DateTime

VBA supporta 3 funzioni integrate per recuperare la data e / o l'ora dall'orologio del sistema.

Funzione	Tipo di reso	Valore di ritorno
Adesso	Data	Restituisce la data e l'ora correnti
Data	Data	Restituisce la parte di data della data e ora correnti
Tempo	Data	Restituisce la porzione di tempo della data e ora correnti

```
Sub DateTimeExample()

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

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

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

End Sub
```

Funzione timer

La funzione `Timer` restituisce un singolo che rappresenta il numero di secondi trascorsi da mezzanotte. La precisione è un centesimo di secondo.

```
Sub TimerExample()

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

End Sub
```

Poiché le funzioni `Now` e `Time` sono precise solo a secondi, `Timer` offre un modo conveniente per aumentare la precisione della misurazione del tempo:

```
Sub GetBenchmark()
```

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

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

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

End Sub
```

IsDate ()

IsDate () verifica se un'espressione è una data valida o meno. Restituisce un `Boolean` .

```
Sub IsDateExamples()

    Dim anything As Variant

    anything = "September 11, 2001"

    Debug.Print IsDate(anything)    'Prints True

    anything = #9/11/2001#

    Debug.Print IsDate(anything)    'Prints True

    anything = "just a string"

    Debug.Print IsDate(anything)    'Prints False

    anything = vbNull

    Debug.Print IsDate(anything)    'Prints False

End Sub
```

Funzioni di estrazione

Queste funzioni prendono una `Variant` che può essere convertita in una `Date` come parametro e restituisce un `Integer` rappresenta una porzione di una data o un'ora. Se il parametro non può essere convertito in una `Date` , si verificherà un errore di run-time 13: tipo mancata corrispondenza.

Funzione	Descrizione	Valore restituito
Anno()	Restituisce la parte dell'anno dell'argomento della data.	Intero (da 100 a 9999)
Mese()	Restituisce la parte del mese dell'argomento della data.	Intero (da 1 a

Funzione	Descrizione	Valore restituito
		12)
Giorno()	Restituisce la parte del giorno dell'argomento della data.	Intero (da 1 a 31)
Giorno della settimana ()	Restituisce il giorno della settimana dell'argomento della data. Accetta un secondo argomento opzionale che definisce il primo giorno della settimana	Intero (da 1 a 7)
Ora()	Restituisce la parte dell'ora dell'argomento della data.	Numero intero (da 0 a 23)
Minute ()	Restituisce la parte dei minuti dell'argomento della data.	Intero (da 0 a 59)
Secondo()	Restituisce la seconda parte dell'argomento della data.	Intero (da 0 a 59)

Esempi:

```
Sub ExtractionExamples()

    Dim MyDate As Date

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

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

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

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

End Sub
```

Funzione DatePart ()

`DatePart()` è anche una funzione che restituisce una parte di una data, ma funziona in modo diverso e consente più possibilità rispetto alle funzioni precedenti. Può ad esempio restituire il trimestre dell'anno o la settimana dell'anno.

Sintassi:

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

l' argomento *intervallo* può essere:

Intervallo	Descrizione
"Aaaa"	Anno (da 100 a 9999)
"Y"	Giorno dell'anno (da 1 a 366)
"M"	Mese (da 1 a 12)
"Q"	Trimestre (da 1 a 4)
"Ww"	Settimana (da 1 a 53)
"W"	Giorno della settimana (da 1 a 7)
"D"	Giorno del mese (da 1 a 31)
"H"	Ora (da 0 a 23)
"N"	Minuto (da 0 a 59)
"S"	Secondo (da 0 a 59)

firstdayofweek è facoltativo. è una costante che specifica il primo giorno della settimana. Se non specificato, `vbSunday` è assunto.

la prima settimana di vita è facoltativa. è una costante che specifica la prima settimana dell'anno. Se non specificato, si presume che la prima settimana sia la settimana in cui si verifica il 1 ° gennaio.

Esempi:

```
Sub DatePartExample()

    Dim MyDate As Date

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

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

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

```
End Sub
```

Funzioni di calcolo

DateDiff ()

`DateDiff()` restituisce un `Long` rappresenta il numero di intervalli di tempo tra due date specificate.

Sintassi

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

- *intervallo* può essere uno qualsiasi degli intervalli definiti nella funzione `DatePart()`
- *data1* e *data2* sono le due date che si desidera utilizzare nel calcolo
- *il primo giorno di settimana* e il primo di *anno* sono opzionali. Fare riferimento alla funzione `DatePart()` per le spiegazioni

Esempi

```
Sub DateDiffExamples()

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

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

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

End Sub
```

DateAdd ()

`DateAdd()` restituisce una `Date` a cui è stata aggiunta una data o un intervallo di tempo specificato.

Sintassi

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

- *intervallo* può essere uno qualsiasi degli intervalli definiti nella funzione `DatePart()`
- *numero* Espressione numerica che corrisponde al numero di intervalli che si desidera aggiungere. Può essere positivo (per ottenere date in futuro) o negativo (per ottenere date in

passato).
- *data* è una `Date` o letterale che rappresenta la data a cui viene aggiunto l'intervallo

Esempi:

```
Sub DateAddExamples()

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

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

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

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

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

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

End Sub
```

Conversione e creazione

CDate ()

`CDate()` converte qualcosa da qualsiasi tipo di dati in un tipo di dati `Date`

```
Sub CDateExamples()

    Dim sample As Date

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

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

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

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

```
End Sub
```

Si noti che VBA dispone anche di un `CVDate()` vagamente tipizzato che funziona allo stesso modo della funzione `CDate()` oltre a restituire una data digitata `Variant` invece di una `Date` fortemente tipizzata. La versione `CDate()` dovrebbe essere preferita quando si passa a un parametro `Date` o si assegna a una variabile `Date` e la versione `CVDate()` deve essere preferita quando si passa a un parametro `Variant` o si assegna a una variabile `Variant` . Questo evita il cast di tipo implicito.

DateSerial ()

`DateSerial()` viene utilizzata per creare una data. Restituisce una `Date` per un anno, mese e giorno specificati.

Sintassi:

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

Gli argomenti relativi a anno, mese e giorno sono validi Numeri interi (Anno da 100 a 9999, Mese da 1 a 12, Giorno da 1 a 31).

Esempi

```
Sub DateSerialExamples()

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

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

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

End Sub
```

Si noti che `DateSerial()` accetterà date "non valide" e calcolerà una data valida da esso. Questo può essere usato creativamente per bene:

Esempio positivo

```
Sub GoodDateSerialExample()

    'Calculate 45 days from today
    Dim today As Date
    today = DateSerial (2001, 9, 11)
    Dim futureDate As Date
    futureDate = DateSerial(Year(today), Month(today), Day(today) + 45)
```

```
    Debug.Print Format$(futureDate, "yyyy-mm-dd")            'prints 2009-10-26

End Sub
```

Tuttavia, è più probabile che causi dolore quando si tenta di creare una data da un input utente non convalidato:

Esempio negativo

```
Sub BadDateSerialExample()

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

End Sub
```

Leggi Data Manipolazione del tempo online: https://riptutorial.com/it/vba/topic/4452/data-manipolazione-del-tempo

Capitolo 18: Dichiarazione delle variabili

Examples

Dichiarazione implicita ed esplicita

Se un modulo di codice non contiene `Option Explicit` nella parte superiore del modulo, allora il compilatore automaticamente (cioè, "implicitamente") creerà delle variabili per te quando le utilizzerai. Verrà impostato di default su `Variant` tipo variabile.

```
Public Sub ExampleDeclaration()

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

End Sub
```

Nel codice precedente, se viene specificata l' `Option Explicit` , il codice verrà interrotto perché mancano le istruzioni `Dim` necessarie per `someVariable` e `someOtherVariable` `someVariable` `someOtherVariable` .

```
Option Explicit

Public Sub ExampleDeclaration()

    Dim someVariable As Long
    someVariable = 10

    Dim someOtherVariable As String
    someOtherVariable = "Hello World"

End Sub
```

È consigliabile utilizzare l'opzione Explicit nei moduli di codice per assicurarsi di dichiarare tutte le variabili.

Consulta le best practice di VBA su come impostare questa opzione per impostazione predefinita.

variabili

Scopo

Una variabile può essere dichiarata (in aumento del livello di visibilità):

- A livello di procedura, utilizzando la parola chiave `Dim` in qualsiasi procedura; una *variabile locale* .
- A livello di modulo, utilizzando la parola chiave `Private` in qualsiasi tipo di modulo; un *campo privato* .

- A livello di istanza, utilizzando la parola chiave `Friend` in qualsiasi tipo di modulo di classe; un *campo amico*
- A livello di istanza, utilizzando la parola chiave `Public` in qualsiasi tipo di modulo di classe; un *campo pubblico* .
- Globalmente, usando la parola chiave `Public` in un *modulo standard* ; una *variabile globale* .

Le variabili dovrebbero sempre essere dichiarate con il più piccolo ambito possibile: preferire il passaggio dei parametri alle procedure, piuttosto che la dichiarazione delle variabili globali.

Vedi Modificatori di accesso per ulteriori informazioni.

Variabili locali

Usa la parola chiave `Dim` per dichiarare una *variabile locale* :

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

La parte `[As Type]` della sintassi della dichiarazione è facoltativa. Se specificato, imposta il tipo di dati della variabile, che determina la quantità di memoria che verrà allocata a tale variabile. Questo dichiara una variabile `String` :

```
Dim identifierName As String
```

Quando un tipo non è specificato, il tipo è implicitamente `Variant` :

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

La sintassi VBA supporta anche la dichiarazione di più variabili in una singola istruzione:

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

Si noti che `[As Type]` deve essere specificato per ogni variabile (diversa da quelle 'Variant'). Questa è una trappola relativamente comune:

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

Variabili statiche

Le variabili locali possono anche essere `Static` . In VBA la parola chiave `Static` viene utilizzata per fare in modo che una variabile "ricordi" il valore che aveva, l'ultima volta che è stata chiamata una procedura:

```
Private Sub DoSomething()
    Static values As Collection
    If values Is Nothing Then
        Set values = New Collection
```

```
        values.Add "foo"
        values.Add "bar"
    End If
    DoSomethingElse values
End Sub
```

Qui la collezione di `values` è dichiarata come Locale `Static` ; perché è una *variabile oggetto* , è inizializzata su `Nothing` . La condizione che segue la dichiarazione verifica se il riferimento all'oggetto è stato `Set` precedenza - se è la prima volta che viene eseguita la procedura, la raccolta viene inizializzata. `DoSomethingElse` potrebbe aggiungere o rimuovere elementi e saranno ancora presenti nella raccolta alla successiva chiamata a `DoSomething` .

Alternativa

> La parola chiave `Static` di VBA può essere facilmente fraintesa, *specialmente* da programmatori esperti che di solito lavorano in altre lingue. In molte lingue, la `static` è usata per far sì che un membro della classe (campo, proprietà, metodo, ...) appartenga al *tipo* piuttosto che *all'istanza* . Il codice nel contesto `static` non può fare riferimento al codice nel contesto *dell'istanza* . La parola chiave `Static` VBA significa qualcosa di molto diverso.

Spesso, un `Static` locale potrebbe altrettanto bene essere implementato come un `Private` variabile (campo), a livello di modulo - tuttavia questo sfida il principio secondo il quale una variabile deve essere dichiarata con il più piccolo campo di applicazione possibile; fidati del tuo istinto, usa quello che preferisci - entrambi funzioneranno ... ma usare `Static` senza capire che cosa potrebbe portare a bug interessanti.

Dim vs. Privato

La parola chiave `Dim` è legale a livello di procedura e modulo; il suo utilizzo a livello di modulo equivale all'utilizzo della parola chiave `Private` :

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

La parola chiave `Private` è legale solo a livello di modulo; ciò invita a riservare `Dim` per le variabili locali e a dichiarare le variabili del modulo con `Private` , in particolare con la parola chiave `Public` contrastante che dovrebbe essere comunque utilizzata per dichiarare un membro pubblico. In alternativa usa `Dim` *ovunque* - ciò che conta è la *coerenza* :

"Campi privati"

- **DO** usa `Private` per dichiarare una variabile a livello di modulo.
- **USARE** `Dim` per dichiarare una variabile locale.
- **NON** usare `Dim` per dichiarare una variabile a livello di modulo.

"Dim ovunque"

- **USARE** `Dim` per dichiarare nulla private / locale.
- **NON** usare `Private` per dichiarare una variabile a livello di modulo.
- **EVITARE la** dichiarazione di campi `Public` . *

* In generale, si dovrebbe comunque evitare di dichiarare campi `Public` o `Global` .

---

campi

Una variabile dichiarata a livello di modulo, nella *sezione delle dichiarazioni* nella parte superiore del corpo del modulo, è un *campo* . Un campo `Public` dichiarato in un *modulo standard* è una *variabile globale* :

```
Public PublicField As Long
```

È possibile accedere a una variabile con ambito globale da qualsiasi luogo, inclusi altri progetti VBA che facciano riferimento al progetto in cui è stato dichiarato.

Per rendere una variabile globale / pubblica, ma visibile solo all'interno del progetto, usa il modificatore `Friend` :

```
Friend FriendField As Long
```

Ciò è particolarmente utile nei componenti aggiuntivi, dove l'intento è che altri progetti VBA facciano riferimento al progetto del componente aggiuntivo e possano utilizzare l'API pubblica.

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

I campi amici non sono disponibili nei moduli standard.

Campi di istanza

Una variabile dichiarata a livello di modulo, nella *sezione delle dichiarazioni* nella parte superiore del corpo di un modulo di classe (inclusi `ThisWorkbook` , `ThisDocument` , `Worksheet` , `UserForm` e *moduli di classe*), è un *campo di istanza* : esiste solo finché c'è *un'istanza* di la classe in giro.

```
'> Class1
Option Explicit
Public PublicField As Long
```

```
'> Module1
Option Explicit
Public Sub DoSomething()
    'Class1.PublicField means nothing here
    With New Class1
        .PublicField = 42
    End With
    'Class1.PublicField means nothing here
```

```
End Sub
```

Campi incapsulanti

I dati di istanza vengono spesso mantenuti `Private` e duplicati *incapsulati* . Un campo privato può essere esposto utilizzando una procedura `Property` . Per esporre una variabile privata pubblicamente senza fornire accesso in scrittura al chiamante, un modulo di classe (o un modulo standard) implementa un membro `Property Get` :

```
Option Explicit
Private encapsulated As Long

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

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

La classe stessa può modificare il valore incapsulato, ma il codice chiamante può accedere solo ai membri `Public` (e ai membri `Friend` , se il chiamante si trova nello stesso progetto).

Per consentire al chiamante di modificare:

- Un **valore** incapsulato, un modulo espone un membro `Property Let` .
- Un **riferimento di oggetto** incapsulato, un modulo espone un membro di `Property Set` .

Costanti (Const)

Se si dispone di un valore che non cambia mai nell'applicazione, è possibile definire una costante denominata e utilizzarla al posto di un valore letterale.

È possibile utilizzare Const solo a livello di modulo o procedura. Ciò significa che il contesto di dichiarazione per una variabile deve essere una classe, una struttura, un modulo, una procedura o un blocco e non può essere un file di origine, uno spazio dei nomi o un'interfaccia.

```
Public Const GLOBAL_CONSTANT As String = "Project Version #1.000.000.001"
Private Const MODULE_CONSTANT As String = "Something relevant to this Module"

Public Sub ExampleDeclaration()

    Const SOME_CONSTANT As String = "Hello World"

    Const PI As Double = 3.141592653

End Sub
```

Sebbene possa essere considerata una buona pratica specificare i tipi di Costante, non è strettamente necessario. Non specificando il tipo verrà comunque restituito il tipo corretto:

```
Public Const GLOBAL_CONSTANT = "Project Version #1.000.000.001" 'Still a string
Public Sub ExampleDeclaration()

    Const SOME_CONSTANT = "Hello World"           'Still a string
    Const DERIVED_CONSTANT = SOME_CONSTANT        'DERIVED_CONSTANT is also a string
    Const VAR_CONSTANT As Variant = SOME_CONSTANT 'VAR_CONSTANT is Variant/String

    Const PI = 3.141592653        'Still a double
    Const DERIVED_PI = PI         'DERIVED_PI is also a double
    Const VAR_PI As Variant = PI  'VAR_PI is Variant/Double

End Sub
```

Si noti che questo è specifico per le costanti e in contrasto con le variabili in cui non si specifica il tipo di risultati in un tipo Variant.

Mentre è possibile dichiarare esplicitamente una costante come String, non è possibile dichiarare una costante come una stringa usando la sintassi della stringa a larghezza fissa

```
'This is a valid 5 character string constant
Const FOO As String = "ABCDE"

'This is not valid syntax for a 5 character string constant
Const FOO As String * 5 = "ABCDE"
```

Modificatori di accesso

L'istruzione `Dim` deve essere riservata alle variabili locali. A livello di modulo, preferisci i modificatori di accesso esplicito:

- `Private` per campi privati, a cui è possibile accedere solo all'interno del modulo in cui sono dichiarati.
- `Public` per campi pubblici e variabili globali, a cui è possibile accedere tramite qualsiasi codice di chiamata.
- `Friend` per le variabili pubbliche all'interno del progetto, ma inaccessibili ad altri progetti VBA di riferimento (rilevanti per i componenti aggiuntivi)
- `Global` può essere utilizzato anche per `Public` campi `Public` in moduli standard, ma è illegale nei moduli di classe ed è comunque obsoleto - preferire invece il modificatore `Public` . Anche questo modificatore non è legale per le procedure.

I modificatori di accesso sono applicabili a variabili e procedure allo stesso modo.

```
Private ModuleVariable As String
Public GlobalVariable As String

Private Sub ModuleProcedure()

    ModuleVariable = "This can only be done from within the same Module"

End Sub

Public Sub GlobalProcedure()
```

```
    GlobalVariable = "This can be done from any Module within this Project"

End Sub
```

Opzione Modulo privato

Le procedure `Sub` parametri pubbliche nei moduli standard sono esposte come macro e possono essere allegate ai comandi e alle scorciatoie da tastiera nel documento host.

Viceversa, le procedure di `Function` pubblica nei moduli standard sono esposte come funzioni definite dall'utente (UDF) nell'applicazione host.

La specifica `Option Private Module` nella parte superiore di un modulo standard impedisce ai suoi membri di essere esposti come macro e UDF all'applicazione host.

Tipo Suggerimenti

I suggerimenti di tipo sono **fortemente** scoraggiati. Esistono e sono documentati qui per motivi di compatibilità storica e retrocompatibile. Dovresti invece utilizzare la sintassi `As [DataType]` .

```
Public Sub ExampleDeclaration()

        Dim someInteger% '% Equivalent to "As Integer"
        Dim someLong&    '& Equivalent to "As Long"
        Dim someDecimal@ '@ Equivalent to "As Currency"
        Dim someSingle!  '! Equivalent to "As Single"
        Dim someDouble#  '# Equivalent to "As Double"
        Dim someString$  '$ Equivalent to "As String"

        Dim someLongLong^  '^ Equivalent to "As LongLong" in 64-bit VBA hosts
End Sub
```

Gli hint del tipo riducono significativamente la leggibilità del codice e incoraggiano una notazione ungherese legacy che impedisce *anche la* leggibilità:

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

Invece, dichiarare le variabili più vicine al loro utilizzo e nominare le cose per quello che vengono utilizzate, non dopo il loro tipo:

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

Gli hint di tipo possono essere utilizzati anche su letterali, per imporre un tipo specifico. Per impostazione predefinita, un valore letterale numerico inferiore a 32.768 verrà interpretato come un valore letterale `Integer` , ma con un suggerimento sul tipo è possibile controllare che:

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

Gli hint di tipo solitamente non sono necessari sui letterali, perché verrebbero assegnati a una variabile dichiarata con un tipo esplicito o convertiti implicitamente al tipo appropriato quando passati come parametri. Le conversioni implicite possono essere evitate usando una delle funzioni di conversione di tipo esplicito:

```
'Calls procedure DoSomething and passes a literal 42 as a Long using a type hint
DoSomething 42&

'Calls procedure DoSomething and passes a literal 42 explicitly converted to a Long
DoSomething CLng(42)
```

Funzioni built-in che restituiscono stringhe

La maggior parte delle funzioni integrate che gestiscono le stringhe sono disponibili in due versioni: una versione con caratteri generici che restituisce una `Variant` e una versione fortemente tipizzata (che termina con `$`) che restituisce una `String` . A meno che non si assegni il valore restituito a una `Variant` , è preferibile la versione che restituisce una `String` , altrimenti esiste una conversione implicita del valore restituito.

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

Queste funzioni sono:

- VBA.Conversion.Error -> VBA.Conversion.Error $
- VBA.Conversion.Hex -> VBA.Conversion.Hex $
- VBA.Conversion.Oct -> VBA.Conversion.Oct $
- VBA.Conversion.Str -> VBA.Conversion.Str $
- VBA.FileSystem.CurDir -> VBA.FileSystem.CurDir $
- VBA. [_ HiddenModule] .Input -> VBA. [_ HiddenModule] .Input $
- VBA. [_ HiddenModule] .InputB -> VBA. [_ HiddenModule] .InputB $
- VBA.Interaction.Command -> VBA.Interaction.Command $
- VBA.Interaction.Environ -> VBA.Interaction.Environ $
- VBA.Strings.Chr -> VBA.Strings.Chr $
- VBA.Strings.ChrB -> VBA.Strings.ChrB $
- VBA.Strings.ChrW -> VBA.Strings.ChrW $
- VBA.Strings.Format -> VBA.Strings.Format $
- VBA.Strings.LCase -> VBA.Strings.LCase $
- VBA.Strings.Left -> VBA.Strings.Left $
- VBA.Strings.LeftB -> VBA.Strings.LeftB $
- VBA.Strings.LTtrim -> VBA.Strings.LTrim $

- VBA.Strings.Mid -> VBA.Strings.Mid $
- VBA.Strings.MidB -> VBA.Strings.MidB $
- VBA.Strings.Right -> VBA.Strings.Right $
- VBA.Strings.RightB -> VBA.Strings.RightB $
- VBA.Strings.RTrim -> VBA.Strings.RTrim $
- VBA.Strings.Space -> VBA.Strings.Space $
- VBA.Strings.Str -> VBA.Strings.Str $
- VBA.Strings.String -> VBA.Strings.String $
- VBA.Strings.Trim -> VBA.Strings.Trim $
- VBA.Strings.UCase -> VBA.Strings.UCase $

Nota che questi sono *alias di* funzione, non del tutto *tipi di suggerimenti* . La funzione `Left` corrisponde alla funzione nascosta `B_Var_Left` , mentre la versione `Left$` corrisponde alla funzione nascosta `B_Str_Left` .

Nelle prime versioni di VBA il segno `$` non è un carattere consentito e il nome della funzione doveva essere racchiuso tra parentesi quadre. In Word Basic c'erano molte, molte più funzioni che restituivano stringhe che terminavano in $.

Dichiarazione di stringhe a lunghezza fissa

In VBA, le stringhe possono essere dichiarate con una lunghezza specifica; vengono automaticamente riempiti o troncati per mantenere quella lunghezza dichiarata.

```
Public Sub TwoTypesOfStrings()

    Dim FixedLengthString As String * 5 ' declares a string of 5 characters
    Dim NormalString As String

    Debug.Print FixedLengthString       ' Prints "     "
    Debug.Print NormalString            ' Prints ""

    FixedLengthString = "123"           ' FixedLengthString now equals "123  "
    NormalString = "456"                ' NormalString now equals "456"

    FixedLengthString = "123456"        ' FixedLengthString now equals "12345"
    NormalString = "456789"             ' NormalString now equals "456789"

End Sub
```

Quando usare una variabile statica

Una variabile statica dichiarata localmente non viene distrutta e non perde il suo valore quando si esce dalla procedura Sub. Le chiamate successive alla procedura non richiedono la reinizializzazione o l'assegnazione, sebbene sia possibile "azzerare" qualsiasi valore memorizzato.

Questi sono particolarmente utili quando legano in ritardo un oggetto in un sub "helper" che viene chiamato ripetutamente.

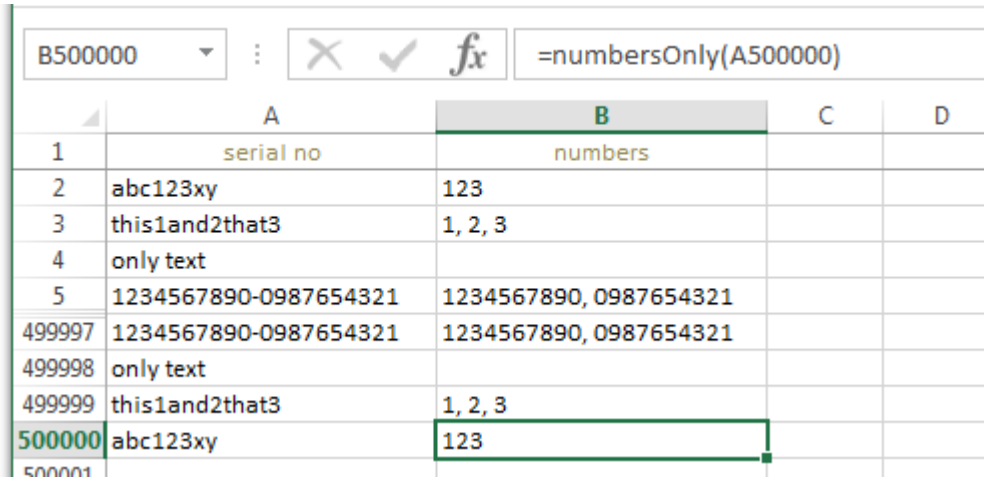
Frammento 1: riutilizzare un oggetto Scripting.Dictionary su più fogli di lavoro

```
Option Explicit

Sub main()
    Dim w As Long

    For w = 1 To Worksheets.Count
        processDictionary ws:=Worksheets(w)
    Next w
End Sub

Sub processDictionary(ws As Worksheet)
    Dim i As Long, rng As Range
    Static dict As Object

    If dict Is Nothing Then
        'initialize and set the dictionary object
        Set dict = CreateObject("Scripting.Dictionary")
        dict.CompareMode = vbTextCompare
    Else
        'remove all pre-existing dictionary entries
        ' this may or may not be desired if a single dictionary of entries
        ' from all worksheets is preferred
        dict.RemoveAll
    End If

    With ws

        'work with a fresh dictionary object for each worksheet
        ' without constructing/destructing a new object each time
        ' or do not clear the dictionary upon subsequent uses and
        ' build a dictionary containing entries from all worksheets

    End With
End Sub
```

Snippet 2: crea un foglio di lavoro UDF che lega in ritardo l'oggetto VBScript.RegExp

```
Option Explicit

Function numbersOnly(str As String, _
                     Optional delim As String = ", ")
    Dim n As Long, nums() As Variant
    Static rgx As Object, cmat As Object

    'with rgx as static, it only has to be created once
    'this is beneficial when filling a long column with this UDF
    If rgx Is Nothing Then
        Set rgx = CreateObject("VBScript.RegExp")
    Else
        Set cmat = Nothing
    End If

    With rgx
        .Global = True
        .MultiLine = True
        .Pattern = "[0-9]{1,999}"
        If .Test(str) Then
            Set cmat = .Execute(str)
            'resize the nums array to accept the matches
            ReDim nums(cmat.Count - 1)
```

```
            'populate the nums array with the matches
            For n = LBound(nums) To UBound(nums)
                nums(n) = cmat.Item(n)
            Next n
            'convert the nums array to a delimited string
            numbersOnly = Join(nums, delim)
        Else
            numbersOnly = vbNullString
        End If
    End With
End Function
```



Esempio di UDF con oggetto statico riempito attraverso mezzo milione di righe

* Tempi trascorsi per riempire righe da 500K con UDF:
  - con **Dim rgx As Object** : 148.74 secondi
  - con **Statico rgx As Object** : 26.07 secondi

* Questi dovrebbero essere considerati solo per confronto relativo. I tuoi risultati varieranno in base alla complessità e
portata delle operazioni eseguite.

Ricorda che una UDF non viene calcolata una volta nella vita di una cartella di lavoro. Anche una UDF non volatile ricalcolerà ogni volta che i valori all'interno degli intervalli a cui fa riferimento sono soggetti a modifiche. Ogni evento di ricalcolo successivo aumenta solo i vantaggi di una variabile dichiarata staticamente.

- Una variabile statica è disponibile per la durata del modulo, non la procedura o la funzione in cui è stata dichiarata e assegnata.
- Le variabili statiche possono essere dichiarate solo localmente.
- Le variabili statiche contengono molte delle stesse proprietà di una variabile a livello di modulo privato ma con un ambito più limitato.

Riferimenti correlati: Statico (Visual Basic)

Leggi Dichiarazione delle variabili online: https://riptutorial.com/it/vba/topic/877/dichiarazione-delle-variabili

Capitolo 19: Dichiarazione e assegnazione di stringhe

Osservazioni

Le stringhe sono un tipo di riferimento e sono fondamentali per la maggior parte delle attività di programmazione. Alle stringhe viene assegnato del testo, anche se il testo sembra essere numerico. Le stringhe possono essere di lunghezza zero o qualsiasi lunghezza fino a 2 GB. Versioni interne di stringhe di archivio VBA moderne che utilizzano una matrice Byte di byte Set di caratteri a byte multiplo (un'alternativa a Unicode).

Examples

Dichiara una costante di stringa

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

Dichiarare una variabile di stringa a larghezza variabile

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

Dichiara e assegna una stringa a larghezza fissa

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

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

Dichiarare e assegnare un array di stringhe

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

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

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

Assegna caratteri specifici all'interno di una stringa usando l'istruzione Mid

VBA offre una funzione Mid per *restituire* sottostringhe all'interno di una stringa, ma offre anche l' *istruzione* Mid che può essere utilizzata per assegnare sottostringhe o singoli caratteri con una stringa.

La funzione `Mid` verrà in genere visualizzata sul lato destro di un'istruzione di assegnazione o in una condizione, ma in genere l'istruzione `Mid` viene visualizzata sul lato sinistro di un'istruzione di assegnazione.

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

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

'Output:
'Smyth
```

Nota: se è necessario assegnare singoli *byte* in una stringa anziché singoli *caratteri* all'interno di una stringa (vedere le Note seguenti per quanto riguarda il set di caratteri Multi-Byte), è possibile utilizzare l'istruzione `MidB` . In questo caso, il secondo argomento per l'istruzione `MidB` è la posizione a 1 del byte in cui verrà `MidB(surname, 5, 2) = "y"` la sostituzione in modo che la riga equivalente dell'esempio precedente sia `MidB(surname, 5, 2) = "y"` .

Assegnazione ae da una matrice di byte

Le stringhe possono essere assegnate direttamente agli array di byte e viceversa. Ricordare che le stringhe sono memorizzate in un set di caratteri Multi-Byte (vedere Note sotto) in modo che solo ogni altro indice dell'array risultante sia la porzione del carattere che rientra nell'intervallo ASCII.

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

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

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

Dim reverted As String
reverted = bytes            'Direct assignment.
```

```
Debug.Print reverted        'Prints "Testing."
```

Leggi Dichiarazione e assegnazione di stringhe online:
https://riptutorial.com/it/vba/topic/3446/dichiarazione-e-assegnazione-di-stringhe

Capitolo 20: Errori in fase di esecuzione VBA

introduzione

Il codice che compila può ancora incorrere in errori, in fase di esecuzione. Questo argomento elenca i più comuni, le loro cause e come evitarli.

Examples

Errore di run-time '3': Ritorno senza GoSub

Codice non corretto

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

Perché non funziona?

L'esecuzione entra nella procedura `DoSomething` , `DoThis` all'etichetta `DoThis` , stampa "Ciao!" all'output di debug, *ritorna* all'istruzione immediatamente dopo la chiamata `GoSub` , stampa "Ciao!" di nuovo, e poi incontra un'istruzione `Return` , ma non c'è nessun luogo in cui *tornare* ora, perché non siamo arrivati qui con un'istruzione `GoSub` .

Codice corretto

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

Perché funziona?

Introducendo un'istruzione `Exit Sub` *prima* `DoThis` Line `DoThis` , abbiamo separato la subroutine `DoThis` dal resto del corpo della procedura - l'unico modo per eseguire la subroutine `DoThis` è tramite `GoSub` jump.

Altre note

`GoSub` / `Return` è deprecato e dovrebbe essere evitato a favore delle chiamate di procedura effettive. Una procedura non dovrebbe contenere subroutine, tranne che gestori di errori.

Questo è molto simile all'errore di run-time '20': riprendi senza errori ; in entrambe le situazioni, la soluzione è garantire che il *normale percorso di esecuzione* non possa entrare in una sottorete (identificata da un'etichetta di linea) senza un salto esplicito (supponendo che `On Error GoTo` sia considerato un *salto esplicito*).

Errore di run-time '6': Overflow

codice non corretto

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

Perché non funziona?

Il tipo di dati `Integer` è un intero con segno a 16 bit con un valore massimo di 32.767; assegnandolo a qualcosa più grande di quello *traboccherà* il tipo e aumenterà questo errore.

Codice corretto

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

Perché funziona?

Usando invece un intero `Long` (32-bit), possiamo ora creare un ciclo che itera più di 32.767 volte senza sovraccaricare il tipo della variabile del contatore.

Altre note

Vedi Tipi di dati e limiti per ulteriori informazioni.

Errore di run-time '9': indice fuori intervallo

codice non corretto

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

Perché non funziona?

`foo` è un array che contiene 10 elementi. Quando il contatore di loop `i` raggiunge un valore di 11, `foo(i)` è *fuori portata* . Questo errore si verifica ogni volta che si accede a una matrice o a una raccolta con un indice che non esiste in quell'array o raccolta.

Codice corretto

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

Perché funziona?

Utilizzare le funzioni `LBound` e `UBound` per determinare rispettivamente i limiti inferiore e superiore di un array.

Altre note

Quando l'indice è una stringa, ad esempio `ThisWorkbook.Worksheets("I don't exist")` , questo errore indica che il nome fornito non esiste nella raccolta interrogata.

L'errore effettivo è però specifico per l'implementazione; `Collection` aumenterà l'errore di runtime 5 "Chiamata o argomento procedura non valida" invece:

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

Errore di run-time "13": tipo mancata corrispondenza

codice non corretto

```
Public Sub DoSomething()
```

```
    DoSomethingElse "42?"
End Sub

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

Perché non funziona?

VBA sta provando davvero a convertire il `"42?"` argomento in un valore `Date` . Quando fallisce, la chiamata a `DoSomethingElse` non può essere eseguita, perché VBA non sa quale data passare, quindi solleva la *mancata corrispondenza di* errore di *tipo* 13, perché il tipo dell'argomento non corrisponde al tipo previsto (e può essere implicitamente convertiti entrambi).

Codice corretto

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

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

Perché funziona?

Passando un argomento `Date` a una procedura che prevede un parametro `Date` , la chiamata può avere successo.

Errore di run-time '91': variabile dell'oggetto o variabile di blocco With non impostata

codice non corretto

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

Perché non funziona?

Le variabili oggetto contengono un *riferimento* e i riferimenti devono essere *impostati* utilizzando la parola chiave `Set` . Questo errore si verifica ogni volta che viene effettuata una chiamata di membro su un oggetto il cui riferimento è `Nothing` . In questo caso, `foo` è un riferimento alla `Collection` , ma non è inizializzato, quindi il riferimento contiene `Nothing` - e non possiamo

chiamare `.Add` su `Nothing` .

Codice corretto

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

Perché funziona?

Assegnando alla variabile oggetto un riferimento valido utilizzando la parola chiave `Set` , le chiamate `.Add` esito positivo.

Altre note

Spesso, una funzione o una proprietà possono restituire un riferimento a un oggetto: un esempio comune è il metodo `Range.Find` di Excel, che restituisce un oggetto `Range` :

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

Tuttavia, la funzione può restituire `Nothing` (se il termine di ricerca non viene trovato), quindi è probabile che la chiamata del membro `.Row` concatenata non riesca.

Prima di chiamare i membri dell'oggetto, verificare che il riferimento sia impostato con una condizione `If Not xxxx Is Nothing` :

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

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

Errore di run-time '20': riprendi senza errori

codice non corretto

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

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

```
End Sub
```

Perché non funziona?

Se la procedura `DoSomethingElse` genera un errore, l'esecuzione salta `CleanFail` linea `CleanFail` , stampa il numero dell'errore e l'istruzione `Resume Next` torna all'istruzione che segue immediatamente la riga in cui si è verificato l'errore, che in questo caso è `Debug.Print` istruzione: la subroutine di gestione degli errori viene eseguita senza un contesto di errore e quando viene raggiunta l'istruzione `Resume Next` , viene generato l'errore di run-time 20 perché non è possibile riprendere da nessuna parte.

Codice corretto

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

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

Perché funziona?

Introducendo un'istruzione `Exit Sub` prima `CleanFail` riga `CleanFail` , abbiamo separato la subroutine di gestione degli errori di `CleanFail` dal resto del corpo della procedura - l'unico modo per eseguire la subroutine di gestione degli errori è tramite un `On Error` ; pertanto, nessun percorso di esecuzione raggiunge l'istruzione `Resume` al di fuori di un contesto di errore, che evita l'errore di runtime 20.

Altre note

Questo è molto simile all'errore di run-time '3': Return senza GoSub ; in entrambe le situazioni, la soluzione è garantire che il *normale percorso di esecuzione* non possa entrare in una sottorete (identificata da un'etichetta di linea) senza un salto esplicito (supponendo che `On Error GoTo` sia considerato un *salto esplicito*).

Leggi Errori in fase di esecuzione VBA online: https://riptutorial.com/it/vba/topic/8917/errori-in-fase-di-esecuzione-vba

Capitolo 21: eventi

Sintassi

- **Modulo sorgente** : `[Public] Event [identifier]([argument_list])`
- **Modulo gestore** : `Dim|Private|Public WithEvents [identifier] As [type]`

Osservazioni

- Un evento può essere solo `Public` . Il modificatore è facoltativo perché i membri del modulo di classe (inclusi gli eventi) sono implicitamente `Public` per impostazione predefinita.
- Una variabile `WithEvents` può essere `Private` o `Public` , ma non `Friend` . Il modificatore è obbligatorio perché `WithEvents` non è una parola chiave che dichiara una variabile, ma una parte parola chiave modificatore della sintassi della dichiarazione di variabile. Quindi la parola chiave `Dim` deve essere utilizzata se non è presente un modificatore di accesso.

Examples

Fonti e gestori

Quali sono gli eventi?

VBA è *basato sugli eventi* : il codice VBA viene eseguito in risposta agli eventi generati dall'applicazione host o dal documento host: la comprensione degli eventi è fondamentale per comprendere VBA.

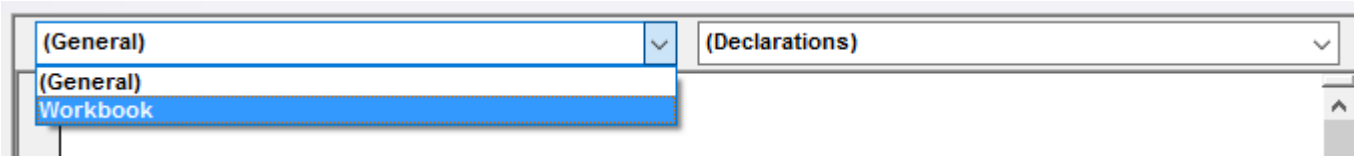
Le API espongono spesso oggetti che generano un numero di *eventi* in risposta a vari stati. Ad esempio, un oggetto `Excel.Application` genera un evento ogni volta che viene creata, aperta, attivata o chiusa una nuova cartella di lavoro. O ogni volta che viene calcolato un foglio di lavoro. O poco prima che un file venga salvato. O subito dopo. Un pulsante su un modulo solleva un evento `Click` quando l'utente fa clic su di esso, lo stesso modulo utente solleva un evento subito dopo l'attivazione e un altro appena prima che venga chiuso.

Dal punto di vista dell'API, gli eventi sono *punti di estensione* : il codice client può scegliere di implementare il codice che *gestisce* questi eventi ed eseguire codice personalizzato ogni volta che questi eventi vengono generati: è così che è possibile eseguire automaticamente il codice personalizzato ogni volta che la selezione cambia su qualsiasi foglio di lavoro - gestendo l'evento che viene generato quando la selezione cambia in qualsiasi foglio di lavoro.

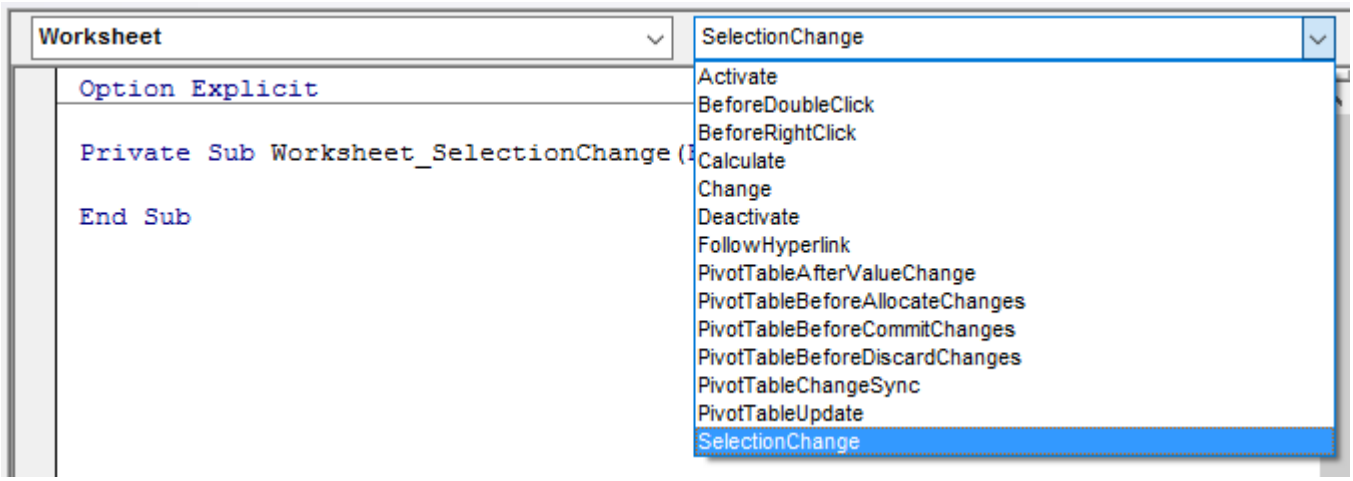
Un oggetto che espone eventi è una *fonte di eventi* . Un metodo che gestisce un evento è un *gestore* .

handlers

I moduli del documento VBA (ad es. `ThisDocument` , `ThisWorkbook` , `Sheet1` , ecc.) `ThisWorkbook` moduli `UserForm` sono *moduli di classe* che *implementano* interfacce speciali che espongono un numero di *eventi* . Puoi sfogliare queste interfacce nel menu a discesa a sinistra nella parte superiore del riquadro del codice:



Il menu a discesa a destra elenca i membri dell'interfaccia selezionata nel menu a discesa a sinistra:



Il VBE genera automaticamente uno stub del gestore di eventi quando un elemento è selezionato nell'elenco di destra, oppure naviga lì se il gestore esiste.

È possibile definire una variabile `WithEvents` ambito modulo in qualsiasi modulo:

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

Ogni dichiarazione `WithEvents` diventa disponibile per la selezione dal menu a discesa sul lato sinistro. Quando un evento è selezionato nel menu a discesa a destra, il VBE genera uno stub del gestore di eventi chiamato dopo l'oggetto `WithEvents` e il nome dell'evento, unito a un trattino basso:

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

Private Sub Foo_Open()

End Sub

Private Sub Bar_SelectionChange(ByVal Target As Range)

```

```
End Sub
```

Solo i tipi che espongono almeno un evento possono essere utilizzati con `WithEvents` e alle dichiarazioni `WithEvents` non può essere assegnato un riferimento sul posto con la parola chiave `New` . Questo codice è illegale:

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

Il riferimento all'oggetto deve essere `Set` esplicito; in un modulo di classe, un buon posto per farlo è spesso nel gestore `Class_Initialize` , perché la classe gestisce gli eventi di quell'oggetto per tutto il tempo in cui esiste l'istanza.

---

fonti

Qualsiasi modulo di classe (o modulo documento o modulo utente) può essere una fonte di eventi. Utilizzare la parola chiave `Event` per definire la *firma* per l'evento, nella *sezione dichiarazioni* del modulo:

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

La firma dell'evento determina come viene generato l'evento e come saranno i gestori di eventi.

Gli eventi possono essere *raccolti* solo all'interno della classe in cui sono definiti: il codice client può *gestirli* solo. Gli eventi vengono `RaiseEvent` con la parola chiave `RaiseEvent` ; gli argomenti dell'evento sono forniti in quel punto:

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

Senza il codice che gestisce l'evento `SomethingHappened` , l'esecuzione della procedura `DoSomething` aumenterà comunque l'evento, ma non accadrà nulla. Supponendo che la sorgente dell'evento sia il codice sopra riportato in una classe chiamata `Something` , questo codice in `ThisWorkbook` mostrerà una finestra di messaggio che dice "ciao" ogni volta `test.DoSomething` viene chiamato il `test.DoSomething` :

```
Private WithEvents test As Something

Private Sub Workbook_Open()
    Set test = New Something
    test.DoSomething
End Sub

Private Sub test_SomethingHappened(ByVal bar As String)
'this procedure runs whenever 'test' raises the 'SomethingHappened' event
    MsgBox bar
End Sub
```

Utilizzo dei parametri passati per riferimento

Un evento può definire un parametro `ByRef` che deve essere restituito al chiamante:

```
Public Event BeforeSomething(ByRef cancel As Boolean)
Public Event AfterSomething()

Public Sub DoSomething()
    Dim cancel As Boolean
    RaiseEvent BeforeSomething(cancel)
    If cancel Then Exit Sub

    'todo: actually do something

    RaiseEvent AfterSomething
End Sub
```

Se l'evento `BeforeSomething` ha un gestore che imposta il parametro `cancel` su `True` , quando l'esecuzione viene restituita dal gestore, l' `cancel` sarà `True` e `AfterSomething` non verrà mai generato.

```
Private WithEvents foo As Something

Private Sub foo_BeforeSomething(ByRef cancel As Boolean)
    cancel = MsgBox("Cancel?", vbYesNo) = vbYes
End Sub

Private Sub foo_AfterSomething()
    MsgBox "Didn't cancel!"
End Sub
```

Supponendo che il riferimento all'oggetto `foo` sia assegnato da qualche parte, quando `foo.DoSomething` eseguito `foo.DoSomething` , una finestra di messaggio richiede se annullare, e una seconda casella di messaggio dice "non annullato" solo quando `No` è stato selezionato.

Usando oggetti mutabili

È anche possibile passare una copia di un oggetto mutabile `ByVal` e consentire ai gestori di modificare le proprietà di quell'oggetto; il chiamante può quindi leggere i valori delle proprietà modificate e agire di conseguenza.

```
'class module ReturnBoolean
Option Explicit
Private encapsulated As Boolean

Public Property Get ReturnValue() As Boolean
'Attribute ReturnValue.VB_UserMemId = 0
```

```
    ReturnValue = encapsulated
End Property

Public Property Let ReturnValue(ByVal value As Boolean)
    encapsulated = value
End Property
```

Combinato con il tipo `Variant` , questo può essere usato per creare modi non ovvi per restituire un valore al chiamante:

```
Public Event SomeEvent(ByVal foo As Variant)

Public Sub DoSomething()
    Dim result As ReturnBoolean
    result = New ReturnBoolean

    RaiseEvent SomeEvent(result)

    If result Then ' If result.ReturnValue Then
        'handler changed the value to True
    Else
        'handler didn't modify the value
    End If
End Sub
```

Il gestore sarebbe simile a questo:

```
Private Sub source_SomeEvent(ByVal foo As Variant) 'foo is actually a ReturnBoolean object
    foo = True 'True is actually assigned to foo.ReturnValue, the class' default member
End Sub
```

Leggi eventi online: https://riptutorial.com/it/vba/topic/5278/eventi

Capitolo 22: Gestione degli errori

Examples

Evitare condizioni di errore

Quando si verifica un errore di runtime, un buon codice dovrebbe gestirlo. La migliore strategia di gestione degli errori è quella di scrivere codice che controlli le condizioni di errore e di evitare semplicemente l'esecuzione di codice che si traduce in un errore di runtime.

Un elemento chiave nella riduzione degli errori di runtime, è la scrittura di piccole procedure che *fanno una cosa* . Minore è il motivo per cui le procedure devono fallire, più facile è il codice nel suo complesso per eseguire il debug.

Evitare l'errore di runtime 91 - Oggetto o variabile di blocco Con non impostata:

Questo errore verrà generato quando un oggetto viene utilizzato prima che venga assegnato il suo riferimento. Si potrebbe avere una procedura che riceve un parametro oggetto:

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

Se a `target` non viene assegnato un riferimento, il codice sopra riportato genererà un errore che può essere facilmente evitato controllando se l'oggetto contiene un riferimento a un oggetto reale:

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

Se alla `target` non viene assegnato un riferimento, il riferimento non assegnato non viene mai utilizzato e non si verifica alcun errore.

Questo modo di uscire anticipatamente da una procedura quando uno o più parametri non sono validi, è chiamato *clausola di guardia* .

Evitare l'errore di runtime 9 - Indice fuori intervallo:

Questo errore viene generato quando si accede a un array al di fuori dei suoi limiti.

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

Dato un indice superiore al numero di fogli di lavoro in `ActiveWorkbook` , il codice sopra riportato

genererà un errore di runtime. Una semplice clausola di salvaguardia può evitare che:

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

La maggior parte degli errori di runtime può essere evitata verificando attentamente i valori che usiamo *prima di* utilizzarli, e ramificando su un altro percorso di esecuzione di conseguenza usando una semplice dichiarazione `If` - nelle clausole di salvaguardia che non fanno presupposizioni e convalida i parametri di una procedura, o anche nel corpo di procedure più grandi.

Sulla dichiarazione di errore

Anche con *le clausole di guardia,* non si può realisticamente *sempre* tenere conto di tutte le possibili condizioni di errore che potrebbero essere sollevate nel corpo di una procedura. L'istruzione `On Error GoTo` indica a VBA di passare a *un'etichetta di riga* e immettere "modalità di gestione degli errori" ogni volta che si verifica un errore imprevisto in fase di esecuzione. Dopo aver gestito un errore, il codice può *riprendere* in esecuzione "normale" utilizzando la parola chiave `Resume` .

Etichette delle linee denotano *subroutine:* perché subroutine provengono da eredità codice BASIC e utilizza `GoTo` e `GoSub` salti e `Return` dichiarazioni per tornare indietro alla routine "principale", è abbastanza facile da scrivere difficile da seguire *spaghetti code* se le cose non sono rigorosamente strutturati . Per questo motivo, è meglio che:

- una procedura ha **una sola** subroutine di gestione degli errori
- la subroutine di gestione degli errori **viene eseguita sempre in uno stato di errore**

Ciò significa che una procedura che gestisce i suoi errori deve essere strutturata in questo modo:

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

    'procedure code here

CleanExit:
    'cleanup code here
    Exit Sub

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

Strategie di gestione degli errori

A volte vuoi gestire errori diversi con azioni diverse. In questo caso ispezionerai l'oggetto `Err` globale, che conterrà informazioni sull'errore che è stato sollevato e agirà di conseguenza:

```
CleanExit:
    Exit Sub

CleanFail:
    Select Case Err.Number
        Case 9
            MsgBox "Specified number doesn't exist. Please try again.", vbExclamation
            Resume
        Case 91
            'woah there, this shouldn't be happening.
            Stop 'execution will break here
            Resume 'hit F8 to jump to the line that raised the error
        Case Else
            MsgBox "An unexpected error has occurred:" & vbNewLine & Err.Description,
vbCritical
            Resume CleanExit
    End Select
End Sub
```

Come regola generale, prendere in considerazione l'attivazione della gestione degli errori per l'intera subroutine o funzione e gestire tutti gli errori che potrebbero verificarsi nell'ambito del proprio ambito. Se devi gestire solo gli errori nella sezione piccola sezione del codice - attiva e disattiva la gestione degli errori allo stesso livello:

```
Private Sub DoSomething(CheckValue as Long)

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

CleanExit:
    Exit Sub

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

End Sub
```

Numeri di linea

VBA supporta i numeri di linea in stile legacy (ad es. QBASIC). La proprietà nascosta di `Erl` può essere utilizzata per identificare il numero di riga che ha generato l'ultimo errore. Se non stai usando i numeri di riga, `Erl` restituirà sempre solo 0.

```
Sub DoSomething()
10 On Error GoTo 50
```

```
20 Debug.Print 42 / 0
30 Exit Sub
40
50 Debug.Print "Error raised on line " & Erl ' returns 20
End Sub
```

Se *si* utilizza i numeri di riga, ma non in modo coerente, allora `Erl` restituirà *l'ultimo numero di riga prima che l'istruzione che ha generato l'errore.*

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

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

Tieni presente che anche `Erl` ha solo precisione di `Integer` e silenziosamente traboccherà. Ciò significa che i numeri di riga al di fuori dell'intervallo intero forniranno risultati errati:

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

Il numero di riga non è tanto rilevante quanto l'affermazione che ha causato l'errore, e le linee di numerazione diventano rapidamente noiose e non abbastanza manutenibili.

Riprendi parola chiave

Una subroutine di gestione degli errori:

- eseguire fino alla fine della procedura, nel qual caso l'esecuzione riprende nella procedura di chiamata.
- oppure, utilizzare la parola chiave `Resume` per *riprendere l'* esecuzione all'interno della stessa procedura.

La parola chiave `Resume` dovrebbe sempre essere utilizzata solo all'interno di una subroutine di gestione degli errori, poiché se VBA rileva `Resume` senza essere in uno stato di errore, viene generato l'errore di runtime 20 "Resume without error".

Esistono diversi modi in cui una subroutine di gestione degli errori può utilizzare la parola chiave `Resume` :

- `Resume` usato da solo, l'esecuzione continua **sulla dichiarazione che ha causato l'errore** . Se l'errore non viene *effettivamente* gestito prima di farlo, lo stesso errore verrà nuovamente generato e l'esecuzione potrebbe entrare in un ciclo infinito.
- `Resume Next` continua l'esecuzione **sull'istruzione immediatamente successiva** all'istruzione che ha causato l'errore. Se l'errore non viene *effettivamente* gestito prima di

farlo, l'esecuzione può continuare con dati potenzialmente non validi, il che può causare errori logici e comportamenti imprevisti.

- `Resume [line label]` continua l'esecuzione **all'etichetta della linea specificata** (o al numero di riga, se si utilizzano numeri di linea in stile legacy). Ciò in genere consente di eseguire un codice di pulitura prima di chiudere in modo pulito la procedura, ad esempio assicurandosi che una connessione al database sia chiusa prima di tornare al chiamante.

In caso di errore, riprendi

L'istruzione `On Error` può utilizzare la parola chiave `Resume` per indicare al runtime VBA di **ignorare** efficacemente **tutti gli errori** .

> *Se l'errore non viene **effettivamente gestito** prima di farlo, l'esecuzione può continuare con dati potenzialmente non validi, il che può causare **errori logici e comportamenti imprevisti** .*

L'enfasi sopra non può essere enfatizzata abbastanza. ***Su Errore Riprendi Avanti in modo efficace ignora tutti gli errori e li spinge sotto il tappeto*** . Un programma che esplode con un errore di runtime dato input non valido è un programma migliore di uno che continua a funzionare con dati sconosciuti / non voluti - sia solo perché il bug è molto più facilmente identificabile. `On Error Resume Next` può facilmente **nascondere bug** .

L'istruzione `On Error` è a livello di procedura - ecco perché dovrebbe esserci *normalmente* una **sola** , singola istruzione `On Error` in una determinata procedura.

Tuttavia a *volte* una condizione di errore non può essere completamente evitata, e saltare a una subroutine di gestione degli errori solo per `Resume Next` non sembra giusto. In questo caso specifico, l'istruzione known-to-fail può essere **racchiusa** tra due istruzioni `On Error` :

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

L'istruzione `On Error GoTo 0` reimposta la gestione degli errori nella procedura corrente, in modo tale che qualsiasi ulteriore istruzione che causa un errore di runtime *non venga gestita all'interno di tale procedura* e passi invece allo stack di chiamate finché non viene catturato da un gestore di errori attivo. Se non è presente alcun gestore di errori attivo nello stack di chiamate, verrà considerato come un'eccezione non gestita.

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

    Callee

    Exit Sub
Handler:
    Debug.Print "Error " & Err.Number & " in Caller."
```

```
End Sub

Public Sub Callee()
    On Error GoTo Handler

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

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

Errori personalizzati

Spesso quando si scrive una classe specializzata, si vorrà che aumenti i propri errori specifici e si vorrà un modo pulito per il codice utente / chiamante per gestire questi errori personalizzati. Un modo efficace per raggiungere questo obiettivo è definire un tipo `Enum` dedicato:

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

L'uso della `vbObjectError` integrata `vbObjectError` garantisce che i codici di errore personalizzati non si sovrappongano ai codici di errore riservati / esistenti. Solo il primo valore enum deve essere specificato esplicitamente, poiché il valore sottostante di ciascun membro `Enum` è `1` maggiore del membro precedente, quindi il valore sottostante di `Err_BarNotInitialized` è implicitamente `vbObjectError + 1025` .

Aumentare i propri errori di runtime

Un errore di runtime può essere generato utilizzando l'istruzione `Err.Raise` , quindi l'errore `Err_FooWasNotBarred` personalizzato può essere generato come segue:

```
Err.Raise Err_FooWasNotBarred
```

Il metodo `Err.Raise` può anche utilizzare i parametri `Description` e `Source` : per questo motivo è una buona idea definire anche le costanti per contenere la descrizione di ciascun errore personalizzato:

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

E quindi creare un metodo privato dedicato per aumentare ogni errore:

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

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

L'implementazione della classe può quindi semplicemente chiamare queste procedure specializzate per sollevare l'errore:

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

Il codice client può quindi gestire `Err_BarNotInitialized` come qualsiasi altro errore, all'interno della propria subroutine di gestione degli errori.

Nota: la parola chiave `Error` legacy può anche essere utilizzata al posto di `Err.Raise` , ma è obsoleta / deprecata.

Leggi Gestione degli errori online: https://riptutorial.com/it/vba/topic/3211/gestione-degli-errori

Capitolo 23: interfacce

introduzione

Un'interfaccia è un modo per definire un insieme di comportamenti che eseguirà una classe. La definizione di un'interfaccia è un elenco di firme del metodo (nome, parametri e tipo di ritorno). Si dice che una classe con tutti i metodi "implementa" quell'interfaccia.

In VBA, l'uso delle interfacce consente al compilatore di verificare che un modulo implementa tutti i suoi metodi. Una variabile o un parametro possono essere definiti in termini di un'interfaccia anziché di una classe specifica.

Examples

Interfaccia semplice - Flyable

L'interfaccia `Flyable` è un modulo di classe con il seguente codice:

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

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

Un modulo di classe, `Airplane` , utilizza la parola chiave `Implements` per dire al compilatore di generare un errore a meno che non abbia due metodi: un sub `Flyable_Fly()` e una funzione `Flyable_GetAltitude()` che restituisce `Long` .

```
Implements Flyable

Public Sub Flyable_Fly()
    Debug.Print "Flying With Jet Engines!"
End Sub

Public Function Flyable_GetAltitude() As Long
    Flyable_GetAltitude = 10000
End Function
```

Un modulo di seconda classe, `Duck` , implementa anche `Flyable` :

```
Implements Flyable

Public Sub Flyable_Fly()
    Debug.Print "Flying With Wings!"
End Sub

Public Function Flyable_GetAltitude() As Long
```

```
    Flyable_GetAltitude = 30
End Function
```

Possiamo scrivere una routine che accetta qualsiasi valore `Flyable` , sapendo che risponderà a un comando di `Fly` o `GetAltitude` :

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

Poiché l'interfaccia è definita, la finestra a `GetAltitude` IntelliSense mostrerà `Fly` e `GetAltitude` per `F`

Quando eseguiamo il seguente codice:

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

FlyAndCheckAltitude MyDuck
FlyAndCheckAltitude MyAirplane
```

L'output è:

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

Si noti che anche se la subroutine è denominata `Flyable_Fly` sia in `Airplane` che in `Duck` , può essere chiamata come `Fly` quando la variabile o parametro è definita come `Flyable` . Se la variabile è definita specificamente come `Duck` , dovrebbe essere chiamata come `Flyable_Fly` .

Interfacce multiple in una classe - Flyable e Swimable

Utilizzando l'esempio `Flyable` come punto di partenza, possiamo aggiungere una seconda interfaccia, `Swimmable` , con il seguente codice:

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

L'oggetto `Duck` può `Implement` sia il volo che il nuoto:

```
Implements Flyable
Implements Swimmable

Public Sub Flyable_Fly()
    Debug.Print "Flying With Wings!"
End Sub

Public Function Flyable_GetAltitude() As Long
    Flyable_GetAltitude = 30
```

```
End Function

Public Sub Swimmable_Swim()
    Debug.Print "Floating on the water"
End Sub
```

Una classe di `Fish` può implementare anche `Swimmable` :

```
Implements Swimmable

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

Ora, possiamo vedere che l'oggetto `Duck` può essere passato ad un Sub come `Flyable` da un lato e `Swimmable` dall'altro:

```
Sub InterfaceTest()

Dim MyDuck As New Duck
Dim MyAirplane As New Airplane
Dim MyFish As New Fish

Debug.Print "Fly Check..."

FlyAndCheckAltitude MyDuck
FlyAndCheckAltitude MyAirplane

Debug.Print "Swim Check..."

TrySwimming MyDuck
TrySwimming MyFish

End Sub

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

Public Sub TrySwimming(S As Swimmable)
    S.Swim
End Sub
```

L'output di questo codice è:

Vola Controlla ...

Volare con le ali!

30

Volare con i motori a reazione!

10000

Swim Check ...

Galleggia sull'acqua

Nuotando sotto l'acqua

Leggi interfacce online: https://riptutorial.com/it/vba/topic/8784/interfacce

Capitolo 24: Lavorare con ADO

Osservazioni

Gli esempi illustrati in questo argomento utilizzano l'associazione anticipata per chiarezza e richiedono un riferimento alla libreria xx Microsoft ActiveX Data Object. Possono essere convertiti in un binding tardivo sostituendo i riferimenti fortemente tipizzati con `Object` e sostituendo la creazione di oggetti utilizzando `New` con `CreateObject` ove appropriato.

Examples

Effettuare una connessione a un'origine dati

Il primo passo per accedere a un'origine dati tramite ADO è la creazione di un oggetto `Connection` ADO. Questo viene in genere eseguito utilizzando una stringa di connessione per specificare i parametri dell'origine dati, sebbene sia anche possibile aprire una connessione DSN passando il DSN, l'ID utente e la password al metodo `.Open` .

Si noti che non è richiesto un DSN per connettersi a un'origine dati tramite ADO - qualsiasi origine dati a cui è collegato un provider ODBC con la stringa di connessione appropriata. Mentre stringhe di connessione specifiche per diversi provider non rientrano nell'ambito di questo argomento, ConnectionStrings.com è un eccellente riferimento per trovare la stringa appropriata per il provider.

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

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

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

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

    OpenDatabaseConnection = database

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

```
End Function
```

Si noti che la password del database è inclusa nella stringa di connessione nell'esempio precedente solo per motivi di chiarezza. Le migliori pratiche imporranno di ***non*** memorizzare le password del database nel codice. Questo può essere ottenuto prendendo la password tramite l'input dell'utente o usando l'autenticazione di Windows.

Recupero di record con una query

Le query possono essere eseguite in due modi, entrambi restituiscono un oggetto `Recordset` ADO che è una raccolta di righe restituite. Si noti che entrambi gli esempi di seguito utilizzano la funzione `OpenDatabaseConnection` dall'esempio Effettuare una connessione a un'origine dati a scopo di brevità. Ricordare che la sintassi del codice SQL passato all'origine dati è specifica del provider.

Il primo metodo consiste nel passare l'istruzione SQL direttamente all'oggetto Connection ed è il metodo più semplice per l'esecuzione di query semplici:

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

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

Il secondo metodo consiste nel creare un oggetto `Command` ADO per la query che si desidera eseguire. Ciò richiede un po 'più di codice, ma è necessario per utilizzare le query parametrizzate:

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

    If Not database Is Nothing Then
```

```
        Dim query As ADODB.Command
        Set query = New ADODB.Command
        'Build the command to pass to the data source.
        With query
            .ActiveConnection = database
            .CommandText = "SELECT DISTINCT Item FROM Table"
            .CommandType = adCmdText
        End With
        Dim records As ADODB.Recordset
        'Execute the command to retrieve the recordset.
        Set records = query.Execute()

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

Si noti che i comandi inviati all'origine dati sono **vulnerabili all'iniezione SQL** , intenzionale o non intenzionale. In generale, le query non dovrebbero essere create concatenando input utente di alcun tipo. Invece, dovrebbero essere parametrizzati (vedere Creazione di comandi parametrizzati).

Esecuzione di funzioni non scalari

Le connessioni ADO possono essere utilizzate per eseguire praticamente qualsiasi funzione di database supportata dal provider tramite SQL. In questo caso non è sempre necessario utilizzare il `Recordset` restituito dalla funzione `Execute` , sebbene possa essere utile per ottenere assegnazioni di chiavi dopo le istruzioni INSERT con @@ Identity o comandi SQL simili. Si noti che nell'esempio seguente viene utilizzata la funzione `OpenDatabaseConnection` dall'esempio Effettuare una connessione a un'origine dati a scopo di brevità.

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

    If Not database Is Nothing Then
        Dim update As ADODB.Command
        Set update = New ADODB.Command
        'Build the command to pass to the data source.
        With update
            .ActiveConnection = database
            .CommandText = "UPDATE Table SET Foo = 42 WHERE Bar IS NULL"
            .CommandType = adCmdText
            .Execute        'We don't need the return from the DB, so ignore it.
```

```
        End With
    End If
CleanExit:
    If Not database Is Nothing And database.State = adStateOpen Then
        database.Close
    End If
    Exit Sub
Handler:
    Debug.Print "Error " & Err.Number & ": " & Err.Description
    Resume CleanExit
End Sub
```

Si noti che i comandi inviati all'origine dati sono **vulnerabili all'iniezione SQL** , intenzionale o non intenzionale. In generale, le istruzioni SQL non dovrebbero essere create concatenando input utente di alcun tipo. Invece, dovrebbero essere parametrizzati (vedere Creazione di comandi parametrizzati).

Creazione di comandi parametrizzati

Ogni volta che SQL eseguito tramite una connessione ADO deve contenere input dell'utente, è consigliabile parametrizzarlo al fine di ridurre al minimo la possibilità di SQL injection. Questo metodo è anche più leggibile rispetto alle concatenazioni lunghe e facilita il codice più robusto e gestibile (ovvero utilizzando una funzione che restituisce un array di `Parameter`).

Nella sintassi ODBC standard, vengono forniti i parametri `?` "segnaposto" nel testo della query, quindi i parametri vengono aggiunti al `Command` nello stesso ordine in cui appaiono nella query.

Si noti che nell'esempio seguente viene utilizzata la funzione `OpenDatabaseConnection` dalla creazione di una connessione a un'origine dati per brevità.

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

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

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

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

            'Add the parameters to the Command
            .Parameters.Append fooValue
```

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

Nota: l'esempio sopra mostra un'istruzione UPDATE parametrizzata, ma a qualsiasi istruzione SQL possono essere dati parametri.

Leggi Lavorare con ADO online: https://riptutorial.com/it/vba/topic/3578/lavorare-con-ado

Capitolo 25: Lavorare con file e directory senza utilizzare FileSystemObject

Osservazioni

`Scripting.FileSystemObject` è molto più robusto dei metodi legacy in questo argomento. Dovrebbe essere preferito in quasi tutti i casi.

Examples

Determinare se esistono cartelle e file

File:

Per determinare se esiste un file, basta passare il nome del file alla funzione `Dir$` e testare per vedere se restituisce un risultato. Nota che `Dir$` supporta le wild card, quindi per testare un file *specifico* , il `pathName` passato deve essere testato per assicurarsi che non li contenga. L'esempio seguente genera un errore: se questo non è il comportamento desiderato, è possibile modificare la funzione per restituire semplicemente `False` .

```
Public Function FileExists(pathName As String) As Boolean
    If InStr(1, pathName, "*") Or InStr(1, pathName, "?") Then
        'Exit Function   'Return False on wild-cards.
        Err.Raise 52    'Raise error on wild-cards.
    End If
    FileExists = Dir$(pathName) <> vbNullString
End Function
```

Cartelle (metodo $ Dir):

La funzione `Dir$()` può anche essere utilizzata per determinare se esiste una cartella specificando il passaggio di `vbDirectory` per il parametro facoltativo degli `attributes` . In questo caso, il valore `pathName` passato deve terminare con un separatore di percorso (`\`), poiché i *nomi di file* corrispondenti causano falsi positivi. Tieni presente che le wild card sono consentite solo dopo l'ultimo separatore del percorso, quindi la funzione di esempio seguente genererà un errore 52 - "Nome file o numero di file errato" se l'input contiene una wild card. Se questo non è il comportamento desiderato, il commento `On Error Resume Next` nella parte superiore della funzione. Ricorda inoltre che `Dir$` supporta i percorsi dei file relativi (ad esempio `..\Foo\Bar`), pertanto i risultati sono validi solo se la directory di lavoro corrente non viene modificata.

```
Public Function FolderExists(ByVal pathName As String) As Boolean
    'Uncomment the "On Error" line if paths with wild-cards should return False
    'instead of raising an error.
    'On Error Resume Next
    If pathName = vbNullString Or Right$(pathName, 1) <> "\" Then
        Exit Function
```

```
    End If
    FolderExists = Dir$(pathName, vbDirectory) <> vbNullString
End Function
```

Cartelle (metodo ChDir):

L'istruzione `ChDir` può anche essere utilizzata per verificare se esiste una cartella. Nota che questo metodo cambierà temporaneamente l'ambiente in cui VBA è in esecuzione, quindi se questa è una considerazione, dovrebbe essere usato il metodo `Dir$` . Ha il vantaggio di essere molto meno indulgente con i suoi parametri. Questo metodo supporta anche i percorsi dei file relativi, quindi ha lo stesso avvertimento del metodo `Dir$` .

```
Public Function FolderExists(ByVal pathName As String) As Boolean
    'Cache the current working directory
    Dim cached As String
    cached = CurDir$

    On Error Resume Next
    ChDir pathName
    FolderExists = Err.Number = 0
    On Error GoTo 0
    'Change back to the cached working directory.
    ChDir cached
End Function
```

Creazione ed eliminazione di cartelle di file

NOTA: per brevità, gli esempi seguenti utilizzano la funzione FolderExists dall'esempio **Determinare se le cartelle e i file esistono** in questo argomento.

L'istruzione `MkDir` può essere utilizzata per creare una nuova cartella. Accetta percorsi contenenti lettere di unità (`C:\Foo`), nomi UNC (`\\Server\Foo`), percorsi relativi (`..\Foo`) o la directory di lavoro corrente (`Foo`).

Se viene omessa l'unità o il nome UNC (ad es. `\Foo`), la cartella viene creata nell'unità corrente. Questo potrebbe essere o meno la stessa unità della directory di lavoro corrente.

```
Public Sub MakeNewDirectory(ByVal pathName As String)
    'MkDir will fail if the directory already exists.
    If FolderExists(pathName) Then Exit Sub
    'This may still fail due to permissions, etc.
    MkDir pathName
End Sub
```

L'istruzione `RmDir` può essere utilizzata per eliminare le cartelle esistenti. Accetta percorsi nelle stesse forme di `MkDir` e utilizza la stessa relazione con la directory di lavoro e l'unità correnti. Si noti che l'istruzione è simile al comando di Windows `rd` shell, pertanto verrà generato un errore 75 di run-time: "Errore di accesso al percorso / file" se la directory di destinazione non è vuota.

```
Public Sub DeleteDirectory(ByVal pathName As String)
    If Right$(pathName, 1) <> "\" Then
        pathName = pathName & "\"
    End If
    'Rmdir will fail if the directory doesn't exist.
    If Not FolderExists(pathName) Then Exit Sub
    'Rmdir will fail if the directory contains files.
    If Dir$(pathName & "*") <> vbNullString Then Exit Sub

    'Rmdir will fail if the directory contains directories.
    Dim subDir As String
    subDir = Dir$(pathName & "*", vbDirectory)
    Do
        If subDir <> "." And subDir <> ".." Then Exit Sub
        subDir = Dir$(, vbDirectory)
    Loop While subDir <> vbNullString

    'This may still fail due to permissions, etc.
    RmDir pathName
End Sub
```

Leggi Lavorare con file e directory senza utilizzare FileSystemObject online: https://riptutorial.com/it/vba/topic/5706/lavorare-con-file-e-directory-senza-utilizzare-filesystemobject

Capitolo 26: Lettura di file da 2 GB + in binario in VBA e File Hash

introduzione

C'è un modo semplice per leggere i file in binario all'interno di VBA, ma ha una limitazione di 2 GB (2.147.483.647 byte - max di tipo di dati lunghi). Con l'evolversi della tecnologia, questo limite di 2 GB viene facilmente violato. ad esempio, un'immagine ISO del sistema operativo installa il disco DVD. Microsoft fornisce un modo per ovviare a questo tramite API Windows di basso livello e qui ne viene fornito un backup.

Dimostrare anche (parte di lettura) per il calcolo degli hash di file senza un programma esterno come `fciv.exe` di Microsoft.

Osservazioni

METODI PER LA CLASSE DI MICROSOFT

Nome del metodo	Descrizione
È aperto	Restituisce un valore booleano per indicare se il file è aperto.
OpenFile (*sFileNam* e As String)	Apre il file specificato dall'argomento sFileName.
CloseFile	Chiude il file attualmente aperto.
ReadByte (*ByteCount* As Long)	Legge ByteCount byte e li restituisce in un array di byte Variant e sposta il puntatore.
WriteBytes (*DataBytes ()* As Byte)	Scrive il contenuto della matrice di byte nella posizione corrente nel file e sposta il puntatore.
rossore	Forza Windows a svuotare la cache di scrittura.
SeekAbsolute (*HighPos* As Long, *LowPos* As Long)	Sposta il puntatore del file nella posizione designata dall'inizio del file. Sebbene VBA consideri i DWORD come valori firmati, l'API li considera come non firmati. Rendere l'argomento di ordine superiore diverso da zero per superare 4 GB. Il DWORD di basso ordine sarà negativo per valori compresi tra 2 GB e 4 GB.
SeekRelative (*Offset* come lungo)	Sposta il puntatore del file fino a +/- 2 GB dalla posizione corrente. È possibile riscrivere questo metodo per consentire offset superiori a 2 GB convertendo un offset con segno a 64 bit in due valori a 32 bit.

PROPRIETA DELLA CLASSE DA MICROSOFT

Proprietà	Descrizione
FileHandle	L'handle del file per il file attualmente aperto. Questo non è compatibile con gli handle di file VBA.
Nome del file	Il nome del file attualmente aperto.
AutoFlush	Imposta / indica se WriteBytes chiamerà automaticamente il metodo Flush.

MODULO NORMALE

Funzione	Gli appunti
GetFileHash (*sFile* As String, *uBlockSize* As Double, *sHashType* As String)	È sufficiente inserire il percorso completo da sottoporre a hash, utilizzare Blocksize (numero di byte) e il tipo di hash da utilizzare: una delle costanti private: **HashTypeMD5** , **HashTypeSHA1** , **HashTypeSHA256** , **HashTypeSHA384** , **HashTypeSHA512** . Questo è stato progettato per essere il più generico possibile.

Dovresti annullare / commentare **uFileSize come Double di** conseguenza. Ho provato MD5 e SHA1.

Examples

Questo deve essere in un modulo di classe, esempi in seguito indicati come "Casuale"

```
' How To Seek Past VBA's 2GB File Limit
' Source: https://support.microsoft.com/en-us/kb/189981 (Archived)
' This must be in a Class Module

Option Explicit

Public Enum W32F_Errors
    W32F_UNKNOWN_ERROR = 45600
    W32F_FILE_ALREADY_OPEN
    W32F_PROBLEM_OPENING_FILE
    W32F_FILE_ALREADY_CLOSED
    W32F_Problem_seeking
End Enum

Private Const W32F_SOURCE = "Win32File Object"
Private Const GENERIC_WRITE = &H40000000
Private Const GENERIC_READ = &H80000000
Private Const FILE_ATTRIBUTE_NORMAL = &H80
```

```
Private Const CREATE_ALWAYS = 2
Private Const OPEN_ALWAYS = 4
Private Const INVALID_HANDLE_VALUE = -1

Private Const FILE_BEGIN = 0, FILE_CURRENT = 1, FILE_END = 2

Private Const FORMAT_MESSAGE_FROM_SYSTEM = &H1000

Private Declare Function FormatMessage Lib "kernel32" Alias "FormatMessageA" ( _
    ByVal dwFlags As Long, _
    lpSource As Long, _
    ByVal dwMessageId As Long, _
    ByVal dwLanguageId As Long, _
    ByVal lpBuffer As String, _
    ByVal nSize As Long, _
    Arguments As Any) As Long

Private Declare Function ReadFile Lib "kernel32" ( _
    ByVal hFile As Long, _
    lpBuffer As Any, _
    ByVal nNumberOfBytesToRead As Long, _
    lpNumberOfBytesRead As Long, _
    ByVal lpOverlapped As Long) As Long

Private Declare Function CloseHandle Lib "kernel32" (ByVal hObject As Long) As Long

Private Declare Function WriteFile Lib "kernel32" ( _
    ByVal hFile As Long, _
    lpBuffer As Any, _
    ByVal nNumberOfBytesToWrite As Long, _
    lpNumberOfBytesWritten As Long, _
    ByVal lpOverlapped As Long) As Long

Private Declare Function CreateFile Lib "kernel32" Alias "CreateFileA" ( _
    ByVal lpFileName As String, _
    ByVal dwDesiredAccess As Long, _
    ByVal dwShareMode As Long, _
    ByVal lpSecurityAttributes As Long, _
    ByVal dwCreationDisposition As Long, _
    ByVal dwFlagsAndAttributes As Long, _
    ByVal hTemplateFile As Long) As Long

Private Declare Function SetFilePointer Lib "kernel32" ( _
    ByVal hFile As Long, _
    ByVal lDistanceToMove As Long, _
    lpDistanceToMoveHigh As Long, _
    ByVal dwMoveMethod As Long) As Long

Private Declare Function FlushFileBuffers Lib "kernel32" (ByVal hFile As Long) As Long

Private hFile As Long, sFName As String, fAutoFlush As Boolean

Public Property Get FileHandle() As Long
    If hFile = INVALID_HANDLE_VALUE Then
        RaiseError W32F_FILE_ALREADY_CLOSED
    End If
    FileHandle = hFile
End Property

Public Property Get FileName() As String
    If hFile = INVALID_HANDLE_VALUE Then
```

```
        RaiseError W32F_FILE_ALREADY_CLOSED
    End If
    FileName = sFName
End Property

Public Property Get IsOpen() As Boolean
    IsOpen = hFile <> INVALID_HANDLE_VALUE
End Property

Public Property Get AutoFlush() As Boolean
    If hFile = INVALID_HANDLE_VALUE Then
        RaiseError W32F_FILE_ALREADY_CLOSED
    End If
    AutoFlush = fAutoFlush
End Property

Public Property Let AutoFlush(ByVal NewVal As Boolean)
    If hFile = INVALID_HANDLE_VALUE Then
        RaiseError W32F_FILE_ALREADY_CLOSED
    End If
    fAutoFlush = NewVal
End Property

Public Sub OpenFile(ByVal sFileName As String)
    If hFile <> INVALID_HANDLE_VALUE Then
        RaiseError W32F_FILE_ALREADY_OPEN, sFName
    End If
    hFile = CreateFile(sFileName, GENERIC_WRITE Or GENERIC_READ, 0, 0, OPEN_ALWAYS,
FILE_ATTRIBUTE_NORMAL, 0)
    If hFile = INVALID_HANDLE_VALUE Then
        RaiseError W32F_PROBLEM_OPENING_FILE, sFileName
    End If
    sFName = sFileName
End Sub

Public Sub CloseFile()
    If hFile = INVALID_HANDLE_VALUE Then
        RaiseError W32F_FILE_ALREADY_CLOSED
    End If
    CloseHandle hFile
    sFName = ""
    fAutoFlush = False
    hFile = INVALID_HANDLE_VALUE
End Sub

Public Function ReadBytes(ByVal ByteCount As Long) As Variant
    Dim BytesRead As Long, Bytes() As Byte
    If hFile = INVALID_HANDLE_VALUE Then
        RaiseError W32F_FILE_ALREADY_CLOSED
    End If
    ReDim Bytes(0 To ByteCount - 1) As Byte
    ReadFile hFile, Bytes(0), ByteCount, BytesRead, 0
    ReadBytes = Bytes
End Function

Public Sub WriteBytes(DataBytes() As Byte)
    Dim fSuccess As Long, BytesToWrite As Long, BytesWritten As Long
    If hFile = INVALID_HANDLE_VALUE Then
        RaiseError W32F_FILE_ALREADY_CLOSED
    End If
    BytesToWrite = UBound(DataBytes) - LBound(DataBytes) + 1
```

```
    fSuccess = WriteFile(hFile, DataBytes(LBound(DataBytes)), BytesToWrite, BytesWritten, 0)
    If fAutoFlush Then Flush
End Sub

Public Sub Flush()
    If hFile = INVALID_HANDLE_VALUE Then
        RaiseError W32F_FILE_ALREADY_CLOSED
    End If
    FlushFileBuffers hFile
End Sub

Public Sub SeekAbsolute(ByVal HighPos As Long, ByVal LowPos As Long)
    If hFile = INVALID_HANDLE_VALUE Then
        RaiseError W32F_FILE_ALREADY_CLOSED
    End If
    LowPos = SetFilePointer(hFile, LowPos, HighPos, FILE_BEGIN)
End Sub

Public Sub SeekRelative(ByVal Offset As Long)
    Dim TempLow As Long, TempErr As Long
    If hFile = INVALID_HANDLE_VALUE Then
        RaiseError W32F_FILE_ALREADY_CLOSED
    End If
    TempLow = SetFilePointer(hFile, Offset, ByVal 0&, FILE_CURRENT)
    If TempLow = -1 Then
        TempErr = Err.LastDllError
        If TempErr Then
            RaiseError W32F_Problem_seeking, "Error " & TempErr & "." & vbCrLf & CStr(TempErr)
        End If
    End If
End Sub

Private Sub Class_Initialize()
    hFile = INVALID_HANDLE_VALUE
End Sub

Private Sub Class_Terminate()
    If hFile <> INVALID_HANDLE_VALUE Then CloseHandle hFile
End Sub

Private Sub RaiseError(ByVal ErrorCode As W32F_Errors, Optional sExtra)
    Dim Win32Err As Long, Win32Text As String
    Win32Err = Err.LastDllError
    If Win32Err Then
        Win32Text = vbCrLf & "Error " & Win32Err & vbCrLf & _
        DecodeAPIErrors(Win32Err)
    End If
    Select Case ErrorCode
        Case W32F_FILE_ALREADY_OPEN
            Err.Raise W32F_FILE_ALREADY_OPEN, W32F_SOURCE, "The file '" & sExtra & "' is
already open." & Win32Text
        Case W32F_PROBLEM_OPENING_FILE
            Err.Raise W32F_PROBLEM_OPENING_FILE, W32F_SOURCE, "Error opening '" & sExtra &
"'." & Win32Text
        Case W32F_FILE_ALREADY_CLOSED
            Err.Raise W32F_FILE_ALREADY_CLOSED, W32F_SOURCE, "There is no open file."
        Case W32F_Problem_seeking
            Err.Raise W32F_Problem_seeking, W32F_SOURCE, "Seek Error." & vbCrLf & sExtra
        Case Else
            Err.Raise W32F_UNKNOWN_ERROR, W32F_SOURCE, "Unknown error." & Win32Text
    End Select
```

```
End Sub

Private Function DecodeAPIErrors(ByVal ErrorCode As Long) As String
    Dim sMessage As String, MessageLength As Long
    sMessage = Space$(256)
    MessageLength = FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, 0&, ErrorCode, 0&, sMessage,
256&, 0&)
    If MessageLength > 0 Then
        DecodeAPIErrors = Left(sMessage, MessageLength)
    Else
        DecodeAPIErrors = "Unknown Error."
    End If
End Function
```

Codice per il calcolo dell'hash del file in un modulo standard

```
Private Const HashTypeMD5 As String = "MD5" ' https://msdn.microsoft.com/en-
us/library/system.security.cryptography.md5cryptoserviceprovider(v=vs.110).aspx
Private Const HashTypeSHA1 As String = "SHA1" ' https://msdn.microsoft.com/en-
us/library/system.security.cryptography.sha1cryptoserviceprovider(v=vs.110).aspx
Private Const HashTypeSHA256 As String = "SHA256" ' https://msdn.microsoft.com/en-
us/library/system.security.cryptography.sha256cryptoserviceprovider(v=vs.110).aspx
Private Const HashTypeSHA384 As String = "SHA384" ' https://msdn.microsoft.com/en-
us/library/system.security.cryptography.sha384cryptoserviceprovider(v=vs.110).aspx
Private Const HashTypeSHA512 As String = "SHA512" ' https://msdn.microsoft.com/en-
us/library/system.security.cryptography.sha512cryptoserviceprovider(v=vs.110).aspx

Private uFileSize As Double ' Comment out if not testing performance by FileHashes()

Sub FileHashes()
    Dim tStart As Date, tFinish As Date, sHash As String, aTestFiles As Variant, oTestFile As
Variant, aBlockSizes As Variant, oBlockSize As Variant
    Dim BLOCKSIZE As Double

    ' This performs performance testing on different file sizes and block sizes
    aBlockSizes = Array("2^12-1", "2^13-1", "2^14-1", "2^15-1", "2^16-1", "2^17-1", "2^18-1",
"2^19-1", "2^20-1", "2^21-1", "2^22-1", "2^23-1", "2^24-1", "2^25-1", "2^26-1")
    aTestFiles = Array("C:\ISO\clonezilla-live-2.2.2-37-amd64.iso",
"C:\ISO\HPIP201.2014_0902.29.iso",
"C:\ISO\SW_DVD5_Windows_Vista_Business_W32_32BIT_English.ISO",
"C:\ISO\Win10_1607_English_x64.iso",
"C:\ISO\SW_DVD9_Windows_Svr_Std_and_DataCtr_2012_R2_64Bit_English.ISO")
    Debug.Print "Test files: " & Join(aTestFiles, " | ")
    Debug.Print "BlockSizes: " & Join(aBlockSizes, " | ")
    For Each oTestFile In aTestFiles
        Debug.Print oTestFile
        For Each oBlockSize In aBlockSizes
            BLOCKSIZE = Evaluate(oBlockSize)
            tStart = Now
            sHash = GetFileHash(CStr(oTestFile), BLOCKSIZE, HashTypeMD5)
            tFinish = Now
            Debug.Print sHash, uFileSize, Format(tFinish - tStart, "hh:mm:ss"), oBlockSize & "
(" & BLOCKSIZE & ")"
        Next
    Next
End Sub

Private Function GetFileHash(ByVal sFile As String, ByVal uBlockSize As Double, ByVal
sHashType As String) As String
```

```
    Dim oFSO As Object ' "Scripting.FileSystemObject"
    Dim oCSP As Object ' One of the "CryptoServiceProvider"
    Dim oRnd As Random ' "Random" Class by Microsoft, must be in the same file
    Dim uBytesRead As Double, uBytesToRead As Double, bDone As Boolean
    Dim aBlock() As Byte, aBytes As Variant ' Arrays to store bytes
    Dim aHash() As Byte, sHash As String, i As Long
    'Dim uFileSize As Double ' Un-Comment if GetFileHash() is to be used individually

    Set oRnd = New Random ' Class by Microsoft: Random
    Set oFSO = CreateObject("Scripting.FileSystemObject")
    Set oCSP = CreateObject("System.Security.Cryptography." & sHashType &
"CryptoServiceProvider")

    If oFSO Is Nothing Or oRnd Is Nothing Or oCSP Is Nothing Then
        MsgBox "One or more required objects cannot be created"
        GoTo CleanUp
    End If

    uFileSize = oFSO.GetFile(sFile).Size ' FILELEN() has 2GB max!
    uBytesRead = 0
    bDone = False
    sHash = String(oCSP.HashSize / 4, "0") ' Each hexadecimal has 4 bits

    Application.ScreenUpdating = False
    ' Process the file in chunks of uBlockSize or less
    If uFileSize = 0 Then
        ReDim aBlock(0)
        oCSP.TransformFinalBlock aBlock, 0, 0
        bDone = True
    Else
        With oRnd
            .OpenFile sFile
            Do
                If uBytesRead + uBlockSize < uFileSize Then
                    uBytesToRead = uBlockSize
                Else
                    uBytesToRead = uFileSize - uBytesRead
                    bDone = True
                End If
                ' Read in some bytes
                aBytes = .ReadBytes(uBytesToRead)
                aBlock = aBytes
                If bDone Then
                    oCSP.TransformFinalBlock aBlock, 0, uBytesToRead
                    uBytesRead = uBytesRead + uBytesToRead
                Else
                    uBytesRead = uBytesRead + oCSP.TransformBlock(aBlock, 0, uBytesToRead,
aBlock, 0)
                End If
                DoEvents
            Loop Until bDone
            .CloseFile
        End With
    End If
    If bDone Then
        ' convert Hash byte array to an hexadecimal string
        aHash = oCSP.hash
        For i = 0 To UBound(aHash)
            Mid$(sHash, i * 2 + (aHash(i) > 15) + 2) = Hex(aHash(i))
        Next
    End If
```

```
    Application.ScreenUpdating = True
    ' Clean up
    oCSP.Clear
CleanUp:
    Set oFSO = Nothing
    Set oRnd = Nothing
    Set oCSP = Nothing
    GetFileHash = sHash
End Function
```

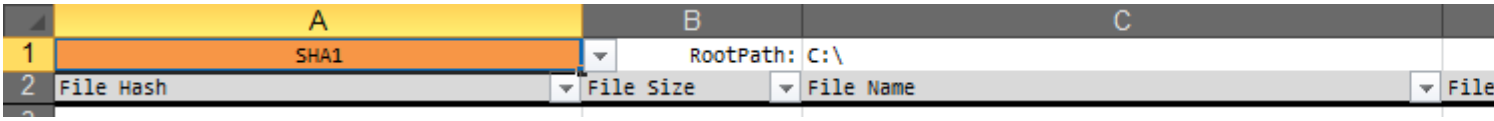
L'output è piuttosto interessante, i miei file di test indicano che `BLOCKSIZE = 131071` **(2 ^ 17-1)** offre le migliori prestazioni generali con Office 2010 a 32 bit su Windows 7 x64, il migliore è *2 ^ 16-1 (65535)* . Nota `2^27-1` produce *memoria* `2^27-1` .

Dimensione del file (byte)	Nome del file
146.800.640	Clonezilla-live-2.2.2-37-amd64.iso
798.210.048	HPIP201.2014_0902.29.iso
2.073.016,32 mila	SW_DVD5_Windows_Vista_Business_W32_32BIT_English.ISO
4.380.387,328 mila	Win10_1607_English_x64.iso
5400,1152 milioni	SW_DVD9_Windows_Svr_Std_and_DataCtr_2012_R2_64Bit_English.ISO

Calcolo di tutti i file Hash da una cartella principale

Un'altra variante del codice sopra offre maggiori prestazioni quando si desidera ottenere i codici hash di tutti i file da una cartella principale incluse tutte le sottocartelle.

Esempio di foglio di lavoro:



Codice

```
Option Explicit

Private Const HashTypeMD5 As String = "MD5" ' https://msdn.microsoft.com/en-
us/library/system.security.cryptography.md5cryptoserviceprovider(v=vs.110).aspx
Private Const HashTypeSHA1 As String = "SHA1" ' https://msdn.microsoft.com/en-
us/library/system.security.cryptography.sha1cryptoserviceprovider(v=vs.110).aspx
```

```
Private Const HashTypeSHA256 As String = "SHA256" ' https://msdn.microsoft.com/en-
us/library/system.security.cryptography.sha256cryptoserviceprovider(v=vs.110).aspx
Private Const HashTypeSHA384 As String = "SHA384" ' https://msdn.microsoft.com/en-
us/library/system.security.cryptography.sha384cryptoserviceprovider(v=vs.110).aspx
Private Const HashTypeSHA512 As String = "SHA512" ' https://msdn.microsoft.com/en-
us/library/system.security.cryptography.sha512cryptoserviceprovider(v=vs.110).aspx

Private Const BLOCKSIZE As Double = 131071 ' 2^17-1

Private oFSO As Object
Private oCSP As Object
Private oRnd As Random ' Requires the Class from Microsoft https://support.microsoft.com/en-
us/kb/189981
Private sHashType As String
Private sRootFDR As String
Private oRng As Range
Private uFileCount As Double

Sub AllFileHashes() ' Active-X button calls this
    Dim oWS As Worksheet
    ' | A: FileHash | B: FileSize | C: FileName | D: FilaName and Path | E: File Last
Modification Time | F: Time required to calculate has code (seconds)
    With ThisWorkbook
        ' Clear All old entries on all worksheets
        For Each oWS In .Worksheets
            Set oRng = Intersect(oWS.UsedRange, oWS.UsedRange.Offset(2))
            If Not oRng Is Nothing Then oRng.ClearContents
        Next
        With .Worksheets(1)
            sHashType = Trim(.Range("A1").Value) ' Range(A1)
            sRootFDR = Trim(.Range("C1").Value) ' Range(C1) Column B for file size
            If Len(sHashType) = 0 Or Len(sRootFDR) = 0 Then Exit Sub
            Set oRng = .Range("A3") ' First entry on First Page
        End With
    End With

    uFileCount = 0
    If oRnd Is Nothing Then Set oRnd = New Random ' Class by Microsoft: Random
    If oFSO Is Nothing Then Set oFSO = CreateObject("Scripting.FileSystemObject") ' Just to
get correct FileSize
    If oCSP Is Nothing Then Set oCSP = CreateObject("System.Security.Cryptography." &
sHashType & "CryptoServiceProvider")

    ProcessFolder oFSO.GetFolder(sRootFDR)

    Application.StatusBar = False
    Application.ScreenUpdating = True
    oCSP.Clear
    Set oCSP = Nothing
    Set oRng = Nothing
    Set oFSO = Nothing
    Set oRnd = Nothing
    Debug.Print "Total file count: " & uFileCount
End Sub

Private Sub ProcessFolder(ByRef oFDR As Object)
    Dim oFile As Object, oSubFDR As Object, sHash As String, dStart As Date, dFinish As Date
    Application.ScreenUpdating = False
    For Each oFile In oFDR.Files
        uFileCount = uFileCount + 1
        Application.StatusBar = uFileCount & ": " & Right(oFile.Path, 255 - Len(uFileCount) -
```

```
2)
        oCSP.Initialize ' Reinitialize the CryptoServiceProvider
        dStart = Now
        sHash = GetFileHash(oFile, BLOCKSIZE, sHashType)
        dFinish = Now
        With oRng
            .Value = sHash
            .Offset(0, 1).Value = oFile.Size ' File Size in bytes
            .Offset(0, 2).Value = oFile.Name ' File name with extension
            .Offset(0, 3).Value = oFile.Path ' Full File name and Path
            .Offset(0, 4).Value = FileDateTime(oFile.Path) ' Last modification timestamp of
file
            .Offset(0, 5).Value = dFinish - dStart ' Time required to calculate hash code
        End With
        If oRng.Row = Rows.Count Then
            ' Max rows reached, start on Next sheet
            If oRng.Worksheet.Index + 1 > ThisWorkbook.Worksheets.Count Then
                MsgBox "All rows in all worksheets have been used, please create more sheets"
                End
            End If
            Set oRng = ThisWorkbook.Sheets(oRng.Worksheet.Index + 1).Range("A3")
            oRng.Worksheet.Activate
        Else
            ' Move to next row otherwise
            Set oRng = oRng.Offset(1)
        End If
    Next
    'Application.StatusBar = False
    Application.ScreenUpdating = True
    oRng.Activate
    For Each oSubFDR In oFDR.SubFolders
        ProcessFolder oSubFDR
    Next
End Sub

Private Function GetFileHash(ByVal sFile As String, ByVal uBlockSize As Double, ByVal
sHashType As String) As String
    Dim uBytesRead As Double, uBytesToRead As Double, bDone As Boolean
    Dim aBlock() As Byte, aBytes As Variant ' Arrays to store bytes
    Dim aHash() As Byte, sHash As String, i As Long, oTmp As Variant
    Dim uFileSize As Double ' Un-Comment if GetFileHash() is to be used individually

    If oRnd Is Nothing Then Set oRnd = New Random ' Class by Microsoft: Random
    If oFSO Is Nothing Then Set oFSO = CreateObject("Scripting.FileSystemObject") ' Just to
get correct FileSize
    If oCSP Is Nothing Then Set oCSP = CreateObject("System.Security.Cryptography." &
sHashType & "CryptoServiceProvider")

    If oFSO Is Nothing Or oRnd Is Nothing Or oCSP Is Nothing Then
        MsgBox "One or more required objects cannot be created"
        Exit Function
    End If

    uFileSize = oFSO.GetFile(sFile).Size ' FILELEN() has 2GB max
    uBytesRead = 0
    bDone = False
    sHash = String(oCSP.HashSize / 4, "0") ' Each hexadecimal is 4 bits

    ' Process the file in chunks of uBlockSize or less
    If uFileSize = 0 Then
        ReDim aBlock(0)
```

```
        oCSP.TransformFinalBlock aBlock, 0, 0
        bDone = True
    Else
        With oRnd
            On Error GoTo CannotOpenFile
            .OpenFile sFile
            Do
                If uBytesRead + uBlockSize < uFileSize Then
                    uBytesToRead = uBlockSize
                Else
                    uBytesToRead = uFileSize - uBytesRead
                    bDone = True
                End If
                ' Read in some bytes
                aBytes = .ReadBytes(uBytesToRead)
                aBlock = aBytes
                If bDone Then
                    oCSP.TransformFinalBlock aBlock, 0, uBytesToRead
                    uBytesRead = uBytesRead + uBytesToRead
                Else
                    uBytesRead = uBytesRead + oCSP.TransformBlock(aBlock, 0, uBytesToRead,
aBlock, 0)
                End If
                DoEvents
            Loop Until bDone
            .CloseFile
CannotOpenFile:
            If Err.Number <> 0 Then ' Change the hash code to the Error description
                oTmp = Split(Err.Description, vbCrLf)
                sHash = oTmp(1) & ":" & oTmp(2)
            End If
        End With
    End If
    If bDone Then
        ' convert Hash byte array to an hexadecimal string
        aHash = oCSP.hash
        For i = 0 To UBound(aHash)
            Mid$(sHash, i * 2 + (aHash(i) > 15) + 2) = Hex(aHash(i))
        Next
    End If
    GetFileHash = sHash
End Function
```

Leggi Lettura di file da 2 GB + in binario in VBA e File Hash online:
https://riptutorial.com/it/vba/topic/8786/lettura-di-file-da-2-gb-plus-in-binario-in-vba-e-file-hash

Capitolo 27: Manipolazione delle stringhe usata frequentemente

introduzione

Esempi rapidi per le funzioni di stringa MID LEFT e RIGHT utilizzando INSTR FIND e LEN.

Come trovi il testo tra due termini di ricerca (Say: dopo i due punti e prima di una virgola)? Come si ottiene il resto di una parola (utilizzando MID o utilizzando DESTRA)? Quale di queste funzioni utilizza parametri basati su Zero e codici di ritorno rispetto a One-based? Cosa succede quando le cose vanno male? Come gestiscono stringhe vuote, risultati non fondati e numeri negativi?

Examples

Manipolazione delle stringhe esempi usati frequentemente

Meglio MID () e altri esempi di estrazione di stringhe, attualmente mancanti dal web. Per favore aiutami a fare un buon esempio, o completa questo qui. Qualcosa come questo:

```
DIM strEmpty as String, strNull as String, theText as String
DIM idx as Integer
DIM letterCount as Integer
DIM result as String

strNull = NOTHING
strEmpty = ""
theText = "1234, 78910"

' _________________
' Extract the word after the comma ", "  and before "910"  result: "78" ***
' _________________

' Get index (place) of comma using INSTR
idx = ...   ' some explanation here
if idx < ... ' check if no comma found in text

' or get index of comma using FIND
idx = ...   ' some explanation here... Note: The difference is...
if idx < ...  ' check if no comma found in text

result = MID(theText, ..., LEN(...

' Retrieve remaining word after the comma
result = MID(theText, idx+1, LEN(theText) - idx+1)

' Get word until the comma using LEFT
result = LEFT(theText, idx - 1)

' Get remaining text after the comma-and-space using RIGHT
result = ...
```

```
' What happens when things go wrong
result = MID(strNothing, 1, 2)    ' this causes ...
result = MID(strEmpty, 1, 2) ' which causes...
result = MID(theText, 30, 2) ' and now...
result = MID(theText, 2, 999) ' no worries...
result = MID(theText, 0, 2)
result = MID(theText, 2, 0)
result = MID(theText -1, 2)
result = MID(theText 2, -1)
idx = INSTR(strNothing, "123")
idx = INSTR(theText, strNothing)
idx = INSTR(theText, strEmpty)
i = LEN(strEmpty)
i = LEN(strNothing) '...
```

Sentiti libero di modificare questo esempio e renderlo migliore. Finché rimane chiaro, e ha in comune pratiche di utilizzo.

Leggi Manipolazione delle stringhe usata frequentemente online:
https://riptutorial.com/it/vba/topic/8890/manipolazione-delle-stringhe-usata-frequentemente

Capitolo 28: Misurare la lunghezza delle stringhe

Osservazioni

La lunghezza di una stringa può essere misurata in due modi: La misura della lunghezza più frequentemente utilizzata è il numero di caratteri che usano le funzioni `Len` , ma VBA può anche rivelare il numero di byte usando `LenB` funzioni `LenB` . Un carattere a doppio byte o Unicode ha una lunghezza superiore a un byte.

Examples

Usa la funzione Len per determinare il numero di caratteri in una stringa

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

Dim charLength As Long

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

Utilizzare la funzione LenB per determinare il numero di byte in una stringa

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

Dim byteLength As Long

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

Preferisci `Se Len (myString) = 0 Then` su` If myString = "" Then`

Quando si controlla se una stringa è a lunghezza zero, è meglio fare pratica e più efficiente ispezionare la lunghezza della stringa piuttosto che confrontare la stringa con una stringa vuota.

```
Const myString As String = vbNullString

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

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

Leggi Misurare la lunghezza delle stringhe online: https://riptutorial.com/it/vba/topic/3576/misurare-la-lunghezza-delle-stringhe

Capitolo 29: Moduli utente

Examples

Migliori pratiche

Un `UserForm` è un modulo di classe con un designer e **un'istanza predefinita** . È possibile accedere al *progettista* premendo `Maiusc` + `F7` mentre si visualizza il *code-behind* e si può accedere al *code-behind* premendo `F7` durante la visualizzazione del *designer* .

Lavora con una nuova istanza ogni volta.

Essendo un *modulo di classe* , una forma è quindi un *progetto* per un *oggetto* . Poiché un modulo può contenere dati e stato, è preferibile lavorare con una nuova *istanza* della classe, anziché con quella predefinita / globale:

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

Invece di:

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

Lavorare con l'istanza predefinita può portare a piccoli bug quando il modulo viene chiuso con il pulsante rosso "X" e / o quando `Unload Me` viene usato nel code-behind.

Attuare la logica altrove.

Un modulo dovrebbe riguardare solo la *presentazione* : un pulsante `Click` gestore che si collega a un database ed esegue una query parametrizzata basata sull'input dell'utente, sta **facendo troppe cose** .

Invece, implementare la *logica applicativa* nel codice che è responsabile per la visualizzazione del modulo, o anche meglio, in moduli e procedure dedicate.

Scrivi il codice in modo tale che UserForm sia sempre e solo responsabile di sapere come visualizzare e raccogliere i dati: da dove provengono i dati, o che cosa succede successivamente ai dati, non ne preoccupa.

Il chiamante non dovrebbe essere disturbato dai controlli.

Crea un *modello* ben definito per il modulo con cui lavorare, nel proprio modulo di classe dedicato o incapsulato all'interno del codice stesso del modulo stesso - esporre il *modello* con le procedure `Property Get` e far lavorare il codice client con questi: questo rende la forma *un'astrazione* sui controlli e i loro dettagli nitty-grintosi, che espongono solo i dati rilevanti al codice cliente.

Questo significa codice che assomiglia a questo:

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

Invece di questo:

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

Gestire l'evento QueryClose.

I moduli hanno in genere un pulsante `Chiudi` e le finestre di dialogo / prompt hanno pulsanti `Ok` e `Annulla` ; l'utente può chiudere il modulo utilizzando la *casella di controllo* del modulo (il pulsante rosso "X"), che distrugge l'istanza del modulo per impostazione predefinita (un'altra buona ragione per *lavorare ogni volta con una nuova istanza*).

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

Il modo più semplice per gestire l'evento `QueryClose` è impostare il parametro `Cancel` su `True` e quindi *nascondere* il modulo anziché *chiuderlo* :

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

In questo modo il pulsante "X" non distruggerà mai l'istanza e il chiamante può accedere in sicurezza a tutti i membri pubblici.

Nascondi, non chiudere.

Il codice che crea un oggetto dovrebbe essere responsabile della sua distruzione: non è responsabilità del modulo scaricare e terminare se stesso.

Evita l'uso di `Unload Me` nel code-behind di un modulo. Chiama invece. `Me.Hide` , in modo che il codice chiamante possa ancora utilizzare l'oggetto creato al momento della chiusura del modulo.

Nome cose.

Utilizzare la finestra degli strumenti *Proprietà* (`F4`) per assegnare un nome a ciascun controllo su un modulo. Il nome di un controllo viene utilizzato nel code-behind, quindi a meno che non si stia utilizzando uno strumento di refactoring in grado di gestirlo, **rinominare un controllo interromperà il codice** , quindi è molto più facile fare le cose in modo corretto in primo luogo, piuttosto che provare per scoprire esattamente quale delle 20 caselle di testo `TextBox12` rappresenta.

Tradizionalmente, i controlli UserForm sono denominati con prefissi in stile ungherese:

- `lblUserName` per un controllo `Label` che indica un nome utente.
- `txtUserName` per un controllo `TextBox` cui l'utente può immettere un nome utente.
- `cboUserName` per un controllo `ComboBox` cui l'utente può immettere o selezionare un nome utente.
- `lstUserName` per un controllo `ListBox` cui l'utente può selezionare un nome utente.
- `btnOk` o `cmdOk` per un controllo `Button` etichettato "Ok".

Il problema è che quando ad esempio l'interfaccia utente viene ridisegnata e un `ComboBox` trasforma in un `ListBox` , il nome deve cambiare per riflettere il nuovo tipo di controllo: è meglio nominare i controlli per quello che rappresentano, piuttosto che dopo il loro tipo di controllo - per *disaccoppiare* il controllo codice dall'interfaccia utente il più possibile.

- `UserNameLabel` per un'etichetta di sola lettura che indica un nome utente.
- `UserNameInput` per un controllo in cui l'utente può immettere o selezionare un nome utente.
- `OkButton` per un pulsante di comando con l'etichetta "Ok".

Qualunque sia lo stile scelto, tutto è meglio che lasciare a tutti i controlli i loro nomi predefiniti. Anche la coerenza nello stile dei nomi è ideale.

Gestione di QueryClose

L'evento `QueryClose` viene generato ogni volta che un modulo sta per essere chiuso, sia tramite azione dell'utente che a livello di programmazione. Il parametro `CloseMode` contiene un valore enum `VbQueryClose` che indica come è stato chiuso il modulo:

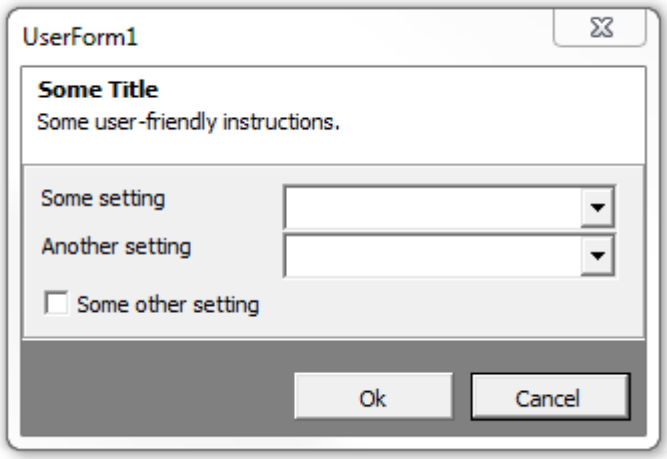
Costante	Descrizione	Valore
`vbFormControlMenu`	Il modulo si sta chiudendo in risposta all'azione dell'utente	0

Costante	Descrizione	Valore
`vbFormCode`	Il modulo si sta chiudendo in risposta a un'istruzione `Unload`	1
`vbAppWindows`	La sessione di Windows sta finendo	2
`vbAppTaskManager`	Task Manager di Windows sta chiudendo l'applicazione host	3
`vbFormMDIForm`	Non supportato in VBA	4

Per una migliore leggibilità, è preferibile utilizzare queste costanti anziché utilizzare direttamente il loro valore.

Un form utente cancellabile

Dato un modulo con un pulsante `Annulla`



Il code-behind del modulo potrebbe assomigliare a questo:

```
Option Explicit
Private Type TView
    IsCancelled As Boolean
    SomeOtherSetting As Boolean
    'other properties skipped for brievety
End Type
Private this As TView

Public Property Get IsCancelled() As Boolean
    IsCancelled = this.IsCancelled
End Property

Public Property Get SomeOtherSetting() As Boolean
    SomeOtherSetting = this.SomeOtherSetting
End Property

'...more properties...

Private Sub SomeOtherSettingInput_Change()
```

```
    this.SomeOtherSetting = CBool(SomeOtherSettingInput.Value)
End Sub

Private Sub OkButton_Click()
    Me.Hide
End Sub

Private Sub CancelButton_Click()
    this.IsCancelled = True
    Me.Hide
End Sub

Private Sub UserForm_QueryClose(Cancel As Integer, CloseMode As Integer)
    If CloseMode = VbQueryClose.vbFormControlMenu Then
        Cancel = True
        this.IsCancelled = True
        Me.Hide
    End If
End Sub
```

Il codice chiamante potrebbe quindi visualizzare il modulo e sapere se è stato cancellato:

```
Public Sub DoSomething()
    With New UserForm1
        .Show vbModal
        If .IsCancelled Then Exit Sub
        If .SomeOtherSetting Then
            'setting is enabled
        Else
            'setting is disabled
        End If
    End With
End Sub
```

La proprietà `IsCancelled` restituisce `True` quando si fa clic sul pulsante `Annulla` o quando l'utente chiude il modulo utilizzando la *casella di controllo* .

Leggi Moduli utente online: https://riptutorial.com/it/vba/topic/5351/moduli-utente

Capitolo 30: operatori

Osservazioni

Gli operatori sono valutati nel seguente ordine:

- Operatori matematici
- Operatori bit a bit
- Operatori di concatenazione
- Operatori di confronto
- Operatori logici

Gli operatori con precedenza corrispondente vengono valutati da sinistra a destra. L'ordine predefinito può essere sovrascritto usando parentesi `(` e `)` per raggruppare le espressioni.

Examples

Operatori matematici

Elencati in ordine di precedenza:

Gettone	Nome	Descrizione
`^`	elevamento a potenza	Restituisce il risultato dell'innalzamento dell'operando di sinistra alla potenza dell'operando di destra. Si noti che il valore restituito da esponenziazione è *sempre* un `Double` , indipendentemente dal tipo di valore che viene diviso. Qualsiasi coercizione del risultato in un tipo di variabile avviene ***dopo che*** il calcolo è stato eseguito.
`/`	Divisione [1]	Restituisce il risultato della divisione dell'operando di sinistra per l'operando di destra. Si noti che il valore restituito dalla divisione è *sempre* un `Double` , indipendentemente dal tipo di valore che viene diviso. Qualsiasi coercizione del risultato in un tipo di variabile avviene ***dopo che*** il calcolo è stato eseguito.
`*`	Moltiplicazione [1]	Restituisce il prodotto di 2 operandi.
`\`	Divisione intera	Restituisce il risultato intero della divisione dell'operando di sinistra per l'operando di destra ***dopo aver*** arrotondato entrambi i lati con un arrotondamento di 0,5. Qualsiasi resto della divisione viene ignorato. Se l'operando di destra (il divisore) è `0` , verrà generato un errore di runtime 11: Divisione per zero. Notare che questo è ***dopo*** che viene eseguito l'arrotondamento - espressioni come `3 \ 0.4` genereranno anche un errore di

Gettone	Nome	Descrizione
		divisione per zero.
`Mod`	Modulo	Restituisce il resto dell'intero di divisione dell'operando di sinistra per l'operando di destra. L'operando su ciascun lato è arrotondato a un intero *prima* della divisione, con 0,5 arrotondamento all'indietro. Ad esempio, entrambi `8.6 Mod 3` e `12 Mod 2.6` risultano in `0` . Se l'operando di destra (il divisore) è `0` , verrà generato un errore di runtime 11: Divisione per zero. Notare che questo è ***dopo*** che viene eseguito l'arrotondamento - espressioni come `3 Mod 0.4` causeranno anche un errore di divisione per zero.
`-`	Sottrazione [2]	Restituisce il risultato della sottrazione dell'operando di destra dall'operando di sinistra.
`+`	Aggiunta [2]	Restituisce la somma di 2 operandi. Si noti che questo token viene anche considerato come operatore di concatenazione quando viene applicato a una `String` . Vedi **Operatori di concatenazione** .

[1] La moltiplicazione e la divisione sono considerate come aventi la stessa precedenza.

[2] Addizione e sottrazione vengono considerate come aventi la stessa precedenza.

Operatori di concatenazione

VBA supporta 2 diversi operatori di concatenazione, `+` e `&` ed entrambi eseguono la stessa identica funzione quando vengono usati con i tipi di `String` - la `String` destra viene aggiunta alla fine della `String` sinistra.

Se l'operatore `&` viene utilizzato con un tipo di variabile diverso da `String` , viene implicitamente convertito in una `String` prima di essere concatenato.

Si noti che l'operatore di concatenazione `+` è un sovraccarico dell'operatore `+` addizione. Il comportamento di `+` è determinato dai tipi di variabili degli operandi e dalla precedenza dei tipi di operatore. Se entrambi gli operandi vengono digitati come `String` o `Variant` con un sottotipo di `String` , vengono concatenati:

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

    left = "5"
    right = "5"

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

Se *entrambi i* lati sono di tipo numerico e l'altro lato è una `String` che può essere forzata in un numero, la precedenza dei tipi degli operatori matematici fa sì che l'operatore venga trattato come operatore di addizione e vengono aggiunti i valori numerici:

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

    left = 5
    right = "5"

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

Questo comportamento può portare a errori delicati, difficili da debug, soprattutto se vengono utilizzati tipi `Variant` , quindi in genere viene utilizzato solo l'operatore `&` per la concatenazione.

Operatori di confronto

Gettone	Nome	Descrizione
`=`	Uguale a	Restituisce `True` se gli operandi di sinistra e di destra sono uguali. Si noti che questo è un sovraccarico dell'operatore di assegnazione.
`<>`	Non uguale a	Restituisce `True` se gli operandi di sinistra e di destra non sono uguali.
`>`	Più grande di	Restituisce `True` se l'operando di sinistra è maggiore dell'operando di destra.
`<`	Meno di	Restituisce `True` se l'operando di sinistra è inferiore all'operando di destra.
`>=`	Maggiore o uguale	Restituisce `True` se il se l'operando di sinistra è maggiore o uguale all'operando di destra.
`<=`	Meno o uguale	Restituisce `True` se l'operando di sinistra è minore o uguale all'operando di destra.
`Is`	Equità di riferimento	Restituisce `True` se il riferimento all'oggetto a sinistra è la stessa istanza del riferimento all'oggetto a destra. Può anche essere utilizzato con `Nothing` (il riferimento a oggetti null) su entrambi i lati. **Nota:** l'operatore `Is` tenterà di forzare entrambi gli operandi in un `Object` prima di eseguire il confronto. Se un lato è un tipo primitivo *o* un `Variant` che non contiene un oggetto (un sottotipo non dell'oggetto o `vtEmpty`), il confronto genererà un errore di runtime 424 - "Oggetto richiesto". Se l'operando appartiene ad una diversa *interfaccia* dello stesso oggetto, il confronto restituisce `True` . Se è necessario verificare l'equità dell'istanza *e* dell'interfaccia, utilizzare invece `ObjPtr(left) = ObjPtr(right)` .

Gli appunti

La sintassi VBA consente "catene" di operatori di confronto, ma questi costrutti dovrebbero essere generalmente evitati. I confronti vengono sempre eseguiti da sinistra a destra su solo 2 operandi per volta e ogni confronto risulta in un `Boolean` . Ad esempio, l'espressione ...

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

... può essere letto in alcuni contesti come un test di se `b` è tra `a` e `c` . In VBA, questo valuta come segue:

```
a = 2: b = 1: c = 0
expr = a > b > c
expr = (2 > 1) > 0
expr = True > 0
expr = -1 > 0 'CInt(True) = -1
expr = False
```

Qualsiasi operatore di confronto diverso da `Is` utilizzato con un `Object` come un operando verrà eseguito sul valore restituito del membro predefinito `Object` . Se l'oggetto non ha un membro predefinito, il confronto genererà un errore di run-time 438 - "L'oggetto non supporta la sua proprietà o il metodo".

Se l' `Object` è stato inizializzato, il confronto genererà un errore 91 di runtime: "Variabile oggetto o variabile di blocco Con non impostata".

Se il valore letterale `Nothing` viene utilizzato con qualsiasi operatore di confronto diverso da `Is` , verrà generato un errore Compile - "Uso non valido dell'oggetto".

Se il membro predefinito `Object` è *un altro* `Object` , VBA chiamerà continuamente il membro predefinito di ciascun valore di ritorno successivo fino a quando viene restituito un tipo primitivo o viene generato un errore. Ad esempio, supponiamo che `SomeClass` abbia un membro predefinito di `Value` , che è un'istanza di `ChildClass` con un membro predefinito di `ChildValue` . Il confronto...

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

... sarà valutato come:

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

Se uno degli operandi è un tipo numerico e l' *altro* operando è `String` o `Variant` del sottotipo `String` , verrà eseguito un confronto numerico. In questo caso, se la `String` non può essere trasmessa a un numero, un errore di runtime 13 - "Tipo non corrispondente" sarà il risultato del confronto.

Se ***entrambi gli*** operandi sono una `String` o una `Variant` del sottotipo `String` , verrà eseguito un confronto tra le stringhe in base all'impostazione Confronta opzioni del modulo codice. Questi confronti sono eseguiti su base carattere per carattere. Si noti che la *rappresentazione* del *carattere* di una `String` contenente un numero ***non*** è uguale a un confronto dei valori numerici:

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

    left = "42"
    right = "5"
    Debug.Print left > right              'Prints False
    Debug.Print Val(left) > Val(right)    'Prints True
End Sub
```

Per questo motivo, assicurati che le variabili `String` o `Variant` vengano convertite in numeri prima di eseguire confronti di ingiustizie numeriche su di esse.

Se un operando è una `Date` , verrà eseguito un confronto numerico sul valore doppio sottostante se l'altro operando è numerico o può essere convertito in un tipo numerico.

Se l'altro operando è una `String` o una `Variant` del sottotipo `String` che può essere trasmessa a una `Date` utilizzando le impostazioni internazionali correnti, la `String` verrà convertita in una `Date` . Se non è possibile eseguire il cast su una `Date` nelle impostazioni internazionali correnti, verrà generato un errore di run-time 13 - "Tipo non corrispondente".

È necessario prestare attenzione quando si effettuano confronti tra valori `Double` o `Single` e Booleans . A differenza di altri tipi numerici, valori diversi da zero non possono essere considerate `True` causa del comportamento di VBA di promuovere il tipo di dati di un confronto che coinvolge un numero decimale a `Double` :

```
Public Sub Example()
    Dim Test As Double

    Test = 42        Debug.Print CBool(Test)             'Prints True.
    'True is promoted to Double - Test is not cast to Boolean
    Debug.Print Test = True             'Prints False

    'With explicit casts:
    Debug.Print CBool(Test) = True      'Prints True
    Debug.Print CDbl(-1) = CDbl(True)   'Prints True
End Sub
```

Operatori logici a bit

Tutti gli operatori logici in VBA possono essere pensati come "override" degli operatori bit a bit con lo stesso nome. Tecnicamente, vengono *sempre* considerati come operatori bit a bit. Tutti gli operatori di confronto in VBA restituiscono un booleano , che non avrà mai nessuno dei suoi bit impostati (`False`) o *tutti i* suoi bit impostati (`True`). Ma tratterà un valore con *qualsiasi* bit impostato come `True` . Ciò significa che il risultato del casting del risultato bitwise di un'espressione

in un `Boolean` (vedere Operatori di confronto) sarà sempre lo stesso del trattarlo come un'espressione logica.

Assegnare il risultato di un'espressione usando uno di questi operatori darà il risultato bit a bit. Notare che nelle tabelle di verità di seguito, `0` equivale a `False` e `1` equivale a `True` .

`And`

Restituisce `True` se le espressioni su entrambi i lati valutano `True` .

Operando di sinistra	Operando a destra	Risultato
0	0	0
0	1	0
1	0	0
1	1	1

`Or`

Restituisce `True` se entrambi i lati dell'espressione valutano `True` .

Operando di sinistra	Operando a destra	Risultato
0	0	0
0	1	1
1	0	1
1	1	1

`Not`

Restituisce `True` se l'espressione restituisce `False` e `False` se l'espressione è valutata su `True` .

Operando a destra	Risultato
0	1
1	0

`Not` è l'unico operando senza un operando di sinistra. Il Visual Basic Editor semplificherà automaticamente le espressioni con un argomento a sinistra. Se digiti ...

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

... il VBE cambierà la linea in:

```
Debug.Print Not x
```

Semplificazioni simili verranno apportate a qualsiasi espressione che contiene un operando di sinistra (incluse le espressioni) per `Not` .

`Xor`

Conosciuto anche come "esclusivo o". Restituisce `True` se entrambe le espressioni valutano risultati diversi.

Operando di sinistra	Operando a destra	Risultato
0	0	0
0	1	1
1	0	1
1	1	0

Si noti che sebbene l'operatore `Xor` possa essere *utilizzato* come un operatore logico, non vi è assolutamente alcun motivo per farlo in quanto fornisce lo stesso risultato dell'operatore di confronto `<>` .

`Eqv`

Conosciuto anche come "equivalenza". Restituisce `True` quando entrambe le espressioni valutano lo stesso risultato.

Operando di sinistra	Operando a destra	Risultato
0	0	1
0	1	0
1	0	0
1	1	1

Notare che la funzione `Eqv` è usata *molto* raramente poiché `x Eqv y` è equivalente al `Not (x Xor y)` molto più leggibile `Not (x Xor y)` .

`Imp`

Conosciuto anche come "implicazione". Restituisce `True` se entrambi gli operandi sono uguali *o* il

secondo operando è `True` .

Operando di sinistra	Operando a destra	Risultato
0	0	1
0	1	1
1	0	0
1	1	1

Si noti che la funzione `Imp` è utilizzata molto raramente. Una buona regola è che se non riesci a spiegare cosa significa, dovresti usare un altro costrutto.

Leggi operatori online: https://riptutorial.com/it/vba/topic/5813/operatori

Capitolo 31: Ordinamento

introduzione

A differenza del framework .NET, la libreria di Visual Basic non include le routine per ordinare gli array.

Esistono due tipi di soluzioni: 1) implementare un algoritmo di ordinamento partendo da zero o 2) utilizzando le routine di ordinamento in altre librerie comunemente disponibili.

Examples

Implementazione dell'algoritmo: ordinamento rapido su una matrice monodimensionale

Dalla funzione di ordinamento dell'array VBA?

```
Public Sub QuickSort(vArray As Variant, inLow As Long, inHi As Long)

  Dim pivot   As Variant
  Dim tmpSwap As Variant
  Dim tmpLow  As Long
  Dim tmpHi   As Long

  tmpLow = inLow
  tmpHi = inHi

  pivot = vArray((inLow + inHi) \ 2)

  While (tmpLow <= tmpHi)

     While (vArray(tmpLow) < pivot And tmpLow < inHi)
        tmpLow = tmpLow + 1
     Wend

     While (pivot < vArray(tmpHi) And tmpHi > inLow)
        tmpHi = tmpHi - 1
     Wend

     If (tmpLow <= tmpHi) Then
        tmpSwap = vArray(tmpLow)
        vArray(tmpLow) = vArray(tmpHi)
        vArray(tmpHi) = tmpSwap
        tmpLow = tmpLow + 1
        tmpHi = tmpHi - 1
     End If

  Wend

  If (inLow < tmpHi) Then QuickSort vArray, inLow, tmpHi
  If (tmpLow < inHi) Then QuickSort vArray, tmpLow, inHi

End Sub
```

Utilizzo della libreria di Excel per ordinare una matrice monodimensionale

Questo codice sfrutta la classe `Sort` nella libreria di oggetti di Microsoft Excel.

Per ulteriori letture, vedere:

- Copia un intervallo in un intervallo virtuale
- Come copiare l'intervallo selezionato in un determinato array?

```
Sub testExcelSort()

Dim arr As Variant

InitArray arr
ExcelSort arr

End Sub

Private Sub InitArray(arr As Variant)

Const size = 10
ReDim arr(size)

Dim i As Integer

' Add descending numbers to the array to start
For i = 0 To size
    arr(i) = size - i
Next i

End Sub

Private Sub ExcelSort(arr As Variant)

' Ininitialize the Excel objects (required)
Dim xl As New Excel.Application
Dim wbk As Workbook
Set wbk = xl.Workbooks.Add
Dim sht As Worksheet
Set sht = wbk.ActiveSheet

' Copy the array to the Range object
Dim rng As Range
Set rng = sht.Range("A1")
Set rng = rng.Resize(UBound(arr, 1), 1)
rng.Value = xl.WorksheetFunction.Transpose(arr)

' Run the worksheet's sort routine on the Range
Dim MySort As Sort
Set MySort = sht.Sort

With MySort
    .SortFields.Clear
    .SortFields.Add rng, xlSortOnValues, xlAscending, xlSortNormal
    .SetRange rng
    .Header = xlNo
    .Apply
End With
```

```
' Copy the results back to the array
CopyRangeToArray rng, arr

' Clear the objects
Set rng = Nothing
wbk.Close False
xl.Quit

End Sub

Private Sub CopyRangeToArray(rng As Range, arr)

Dim i As Long
Dim c As Range

' Can't just set the array to Range.value (adds a dimension)
For Each c In rng.Cells
    arr(i) = c.Value
    i = i + 1
Next c

End Sub
```

Leggi Ordinamento online: https://riptutorial.com/it/vba/topic/8836/ordinamento

Capitolo 32: Parola chiave dell'opzione VBA

Sintassi

- Opzione optionName [valore]
- Opzione esplicita
- Opzione Confronta {Testo | Binario | Banca dati}
- Opzione Modulo privato
- Opzione Base {0 | 1}

Parametri

Opzione	Dettaglio
Esplicito	*Richiedere la dichiarazione delle variabili* nel modulo in cui è specificata (idealmente tutte); con questa opzione specificata, l'uso di una variabile non dichiarata (/ misplicata) diventa un errore di compilazione.
Confronta testo	Rende i confronti delle stringhe del modulo senza distinzione tra maiuscole e minuscole, in base alle impostazioni locali del sistema, dando la priorità all'equivalenza alfabetica (ad es. "A" = "A").
Confronta binario	Modalità di comparazione delle stringhe predefinita. Fa in modo che i confronti tra stringhe del modulo siano case sensitive, confrontando stringhe usando la rappresentazione binaria / valore numerico di ciascun carattere (ad es. ASCII).
Confronta il database	(Solo MS-Access) Fa in modo che i confronti tra stringhe del modulo funzionino come farebbero in un'istruzione SQL.
Modulo privato	Impedisce l'accesso al membro `Public` del modulo dall'esterno del progetto in cui risiede il modulo, nascondendo in modo efficace le procedure dall'applicazione host (ovvero non disponibile per l'utilizzo come macro o funzioni definite dall'utente).
Opzione Base 0	Impostazione predefinita. Imposta l'array implicito inferiore a `0` in un modulo. Quando viene dichiarata una matrice senza un valore limite inferiore esplicito, verrà utilizzato `0` .
Opzione Base 1	Imposta il limite inferiore dell'array implicito su `1` in un modulo. Quando un array viene dichiarato senza un valore limite inferiore esplicito, `1` verrà utilizzato.

Osservazioni

È molto più facile controllare i confini degli array dichiarando esplicitamente i limiti piuttosto che lasciare che il compilatore ricada su una dichiarazione di `Option Base {0|1}` . Questo può essere fatto in questo modo:

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

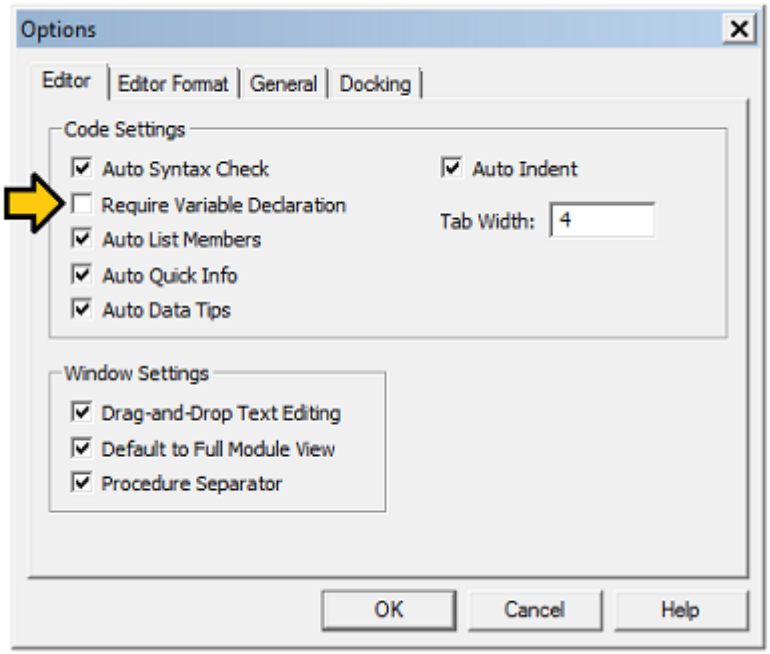
Examples

Opzione esplicita

È consigliabile utilizzare sempre `Option Explicit` in VBA in quanto costringe lo sviluppatore a dichiarare tutte le loro variabili prima dell'uso. Questo ha anche altri vantaggi, come l'auto-capitalizzazione per i nomi delle variabili dichiarate e IntelliSense.

```
Option Explicit

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

L'impostazione **richiede la dichiarazione delle variabili** all'interno degli strumenti di VBE ▸ `Opzioni ▸ Pagina delle proprietà dell'editor inserisce l'istruzione` **Explicit Option** nella parte superiore di ogni foglio di codice appena creato.



Ciò eviterà errori di codifica stupidi come errori di ortografia e influirà sull'uso del tipo di variabile corretto nella dichiarazione delle variabili. (Alcuni altri esempi sono forniti SEMPRE in "Option Explicit" .)

Opzione Confronta binario

Confronto binario rende tutti i controlli per l'uguaglianza di stringa all'interno di un involucro Modulo / classe *sensibile.* Tecnicamente, con questa opzione, i confronti delle stringhe vengono eseguiti utilizzando l'ordinamento delle rappresentazioni binarie di ciascun carattere.

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

Se in un modulo non è specificata alcuna opzione Confronta, Binary viene utilizzato per impostazione predefinita.

```
Option Compare Binary

Sub CompareBinary()

    Dim foo As String
    Dim bar As String

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

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

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

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

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

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

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

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

End Sub
```

Opzione Confronta testo

Opzione Confronta testo rende tutti i confronti di stringa all'interno di un modulo / classe utilizzando un confronto senza distinzione tra maiuscole e *minuscole* .

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

```
Option Compare Text

Sub CompareText()

    Dim foo As String
    Dim bar As String

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

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

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

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

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

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

End Sub
```

Opzione Confronta database

Option Compare Database è disponibile solo all'interno di MS Access. Imposta il modulo / classe per utilizzare le impostazioni correnti del database per determinare se utilizzare la modalità Testo o Binario.

Nota: l'uso di questa impostazione è sconsigliato a meno che il modulo non venga utilizzato per la scrittura di UDF di accesso personalizzate (funzioni definite dall'utente) che dovrebbero trattare i confronti del testo nello stesso modo delle query SQL in quel database.

Opzione Base {0 | 1}

`Option Base` è usato per dichiarare il limite inferiore predefinito degli elementi **dell'array** . È dichiarato a livello di modulo ed è valido solo per il modulo corrente.

Di default (e quindi se non viene specificata alcuna Option Base), la Base è 0. Ciò significa che il primo elemento di ogni array dichiarato nel modulo ha un indice di 0.

Se viene specificata l' `Option Base 1` , il primo elemento dell'array ha l'indice 1

Esempio in base 0:

```
Option Base 0

Sub BaseZero()
```

```
    Dim myStrings As Variant

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

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

    For i = 0 To UBound(myStrings)

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

    Next i

End Sub
```

Lo stesso esempio con Base 1

```
Option Base 1

Sub BaseOne()

    Dim myStrings As Variant

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

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

    For i = 0 To UBound(myStrings)

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

    Next i

End Sub
```

Il secondo esempio ha generato un Sottoscritto fuori intervallo (Errore 9) al primo stadio del ciclo perché è stato effettuato un tentativo di accedere all'indice 0 dell'array e questo indice non esiste poiché il modulo viene dichiarato con `Base 1`

Il codice corretto con Base 1 è:

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

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

    Next i
```

Va notato che la funzione Dividi crea **sempre** una matrice con un indice di elemento a base zero indipendentemente da qualsiasi impostazione di `Option Base` . Gli esempi su come usare la

funzione **Split** possono essere trovati qui

> Funzione Split
>
> > Restituisce un array unidimensionale a base zero contenente un numero specificato di sottostringhe.

In Excel, le proprietà `Range.Value` e `Range.Formula` per un intervallo `Range.Formula` restituiscono *sempre* un array Variant 2D basato su 1.

Allo stesso modo, in ADO, il metodo `Recordset.GetRows` restituisce *sempre* un array 2D basato su 1.

Una "best practice" consigliata è quella di utilizzare sempre le funzioni LBound e UBound per determinare le estensioni di un array.

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

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

L' `Option Base 1` deve trovarsi nella parte superiore di ogni modulo di codice in cui viene creato o ridimensionato un array se gli array devono essere creati in modo coerente con un limite inferiore di 1.

Leggi Parola chiave dell'opzione VBA online: https://riptutorial.com/it/vba/topic/3992/parola-chiave-dell-opzione-vba

Capitolo 33: Passando argomenti ByRef o ByVal

introduzione

I modificatori `ByRef` e `ByVal` fanno parte della firma di una procedura e indicano come viene passato un argomento a una procedura. In VBA un parametro è passato a `ByRef` se non diversamente specificato (es. `ByRef` è implicito se assente).

Nota In molti altri linguaggi di programmazione (incluso VB.NET), i parametri vengono passati implicitamente dal valore se non viene specificato alcun modificatore: considerare di specificare esplicitamente i modificatori `ByRef` per evitare possibili confusioni.

Osservazioni

Passare gli array

Le matrici **devono** essere passate per riferimento. Questo codice viene compilato, ma solleva l'errore di run-time 424 "Object Required":

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

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

Questo codice non viene compilato:

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

Examples

Passare variabili semplici ByRef e ByVal

Passando `ByRef` o `ByVal` indica se il valore effettivo di un argomento viene passato a `CalledProcedure` da `CallingProcedure` o se un riferimento (chiamato un puntatore in alcuni altri linguaggi) viene passato a `CalledProcedure` .

Se viene passato un argomento `ByRef` , l'indirizzo di memoria dell'argomento viene passato a `CalledProcedure` e qualsiasi modifica a tale parametro da `CalledProcedure` viene apportata al valore in `CallingProcedure` .

Se un argomento viene passato a `ByVal` , il valore effettivo, non un riferimento alla variabile, viene passato a `CalledProcedure` .

Un semplice esempio illustrerà chiaramente questo:

```
Sub CalledProcedure(ByRef X As Long, ByVal Y As Long)
    X = 321
    Y = 654
End Sub

Sub CallingProcedure()
    Dim A As Long
    Dim B As Long
    A = 123
    B = 456

    Debug.Print "BEFORE CALL => A: " & CStr(A), "B: " & CStr(B)
    ''Result : BEFORE CALL => A: 123 B: 456

    CalledProcedure X:=A, Y:=B

    Debug.Print "AFTER CALL =  A: " & CStr(A), "B: " & CStr(B)
    ''Result : AFTER CALL => A: 321 B: 456
End Sub
```

Un altro esempio:

```
Sub Main()
    Dim IntVarByVal As Integer
    Dim IntVarByRef As Integer

    IntVarByVal = 5
    IntVarByRef = 10

    SubChangeArguments IntVarByVal, IntVarByRef '5 goes in as a "copy". 10 goes in as a
reference
    Debug.Print "IntVarByVal: " & IntVarByVal 'prints 5 (no change made by SubChangeArguments)
    Debug.Print "IntVarByRef: " & IntVarByRef 'prints 99 (the variable was changed in
SubChangeArguments)
End Sub

Sub SubChangeArguments(ByVal ParameterByVal As Integer, ByRef ParameterByRef As Integer)
    ParameterByVal = ParameterByVal + 2 ' 5 + 2 = 7 (changed only inside this Sub)
    ParameterByRef = ParameterByRef + 89 ' 10 + 89 = 99 (changes the IntVarByRef itself - in
the Main Sub)
End Sub
```

ByRef

Modificatore predefinito

Se non è specificato alcun modificatore per un parametro, quel parametro viene passato implicitamente per riferimento.

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

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

Il parametro `foo` viene passato da `ByRef` in `DoSomething1` e `DoSomething2` .

> **Attento!** Se vieni in VBA con esperienza in altre lingue, è molto probabile che sia esattamente il contrario di quello a cui sei abituato. In molti altri linguaggi di programmazione (incluso VB.NET), il modificatore implicito / predefinito passa i parametri per valore.

Passando per riferimento

- Quando un *valore* viene passato `ByRef` , la procedura riceve **un riferimento** al valore.

```
Public Sub Test()
    Dim foo As Long
    foo = 42
    DoSomething foo
    Debug.Print foo
End Sub

Private Sub DoSomething(ByRef foo As Long)
    foo = foo * 2
End Sub
```

Richiamo delle uscite della procedura `Test` sopra 84. `DoSomething` viene assegnato a `foo` e riceve un *riferimento* al valore e pertanto funziona con lo stesso indirizzo di memoria del chiamante.

- Quando viene passato un *riferimento* `ByRef` , la procedura riceve **un riferimento** al puntatore.

```
Public Sub Test()
    Dim foo As Collection
    Set foo = New Collection
    DoSomething foo
    Debug.Print foo.Count
End Sub

Private Sub DoSomething(ByRef foo As Collection)
    foo.Add 42
    Set foo = Nothing
End Sub
```

Il codice precedente solleva l' errore 91 di runtime , poiché il chiamante chiama il membro `Count` di un oggetto che non esiste più, perché a `DoSomething` stato assegnato un *riferimento* al puntatore dell'oggetto e lo ha assegnato a `Nothing` prima di tornare.

Forcing ByVal sul sito di chiamata

Utilizzando le parentesi sul sito di chiamata, è possibile ignorare `ByRef` e forzare un argomento a passare `ByVal` :

```
Public Sub Test()
    Dim foo As Long
    foo = 42
    DoSomething (foo)
    Debug.Print foo
End Sub

Private Sub DoSomething(ByRef foo As Long)
    foo = foo * 2
End Sub
```

Il codice precedente emette 42, indipendentemente dal fatto che `ByRef` sia specificato in modo implicito o esplicito.

> **Attento!** Per questo motivo, l'uso di parentesi estranee nelle chiamate di procedura può facilmente introdurre bug. Presta attenzione agli spazi bianchi tra il nome della procedura e l'elenco degli argomenti:

```
bar = DoSomething(foo) 'function call, no whitespace; parens are part of args list
DoSomething (foo) 'procedure call, notice whitespace; parens are NOT part of args
list
DoSomething foo 'procedure call does not force the foo parameter to be ByVal
```

ByVal

Passando per valore

- Quando un *valore* viene passato a `ByVal` , la procedura riceve **una copia** del valore.

  ```
  Public Sub Test()
      Dim foo As Long
      foo = 42
      DoSomething foo
      Debug.Print foo
  End Sub

  Private Sub DoSomething(ByVal foo As Long)
      foo = foo * 2
  End Sub
  ```

  Chiamando le uscite della procedura `Test` sopra descritte 42. `DoSomething` viene assegnato a `foo` e riceve **una copia** del valore. La copia viene moltiplicata per 2 e quindi scartata quando la procedura termina; la copia del chiamante non è mai stata modificata.

- Quando viene passato un *riferimento* `ByVal` , la procedura riceve **una copia** del puntatore.

```
Public Sub Test()
    Dim foo As Collection
    Set foo = New Collection
    DoSomething foo
    Debug.Print foo.Count
End Sub

Private Sub DoSomething(ByVal foo As Collection)
    foo.Add 42
    Set foo = Nothing
End Sub
```

Richiamo delle uscite della procedura `Test` sopra 1. `DoSomething` viene fornito `foo` e riceve *una copia* del **puntatore** sull'oggetto `Collection` . Poiché la variabile oggetto `foo` nell'ambito `Test` punta allo stesso oggetto, l'aggiunta di un elemento in `DoSomething` aggiunge l'elemento allo stesso oggetto. Poiché si tratta di *una copia* del puntatore, l'impostazione del suo riferimento a `Nothing` non influisce sulla copia del chiamante.

Leggi Passando argomenti ByRef o ByVal online:
https://riptutorial.com/it/vba/topic/7363/passando-argomenti-byref-o-byval

Capitolo 34: Personaggi non latini

introduzione

VBA può leggere e scrivere stringhe in qualsiasi lingua o script utilizzando Unicode . Tuttavia, esistono regole più severe per i token di identificazione .

Examples

Testo non latino in codice VBA

Nella cell A1 del foglio di calcolo, abbiamo il seguente pangram arabo:

صف خلق خود كمثل الشمس إذا بزغت - يحظى الضجيع بها نجلاء معطار

VBA fornisce le funzioni `AscW` e `ChrW` per lavorare con codici carattere multibyte. Possiamo anche utilizzare gli array `Byte` per manipolare direttamente la variabile stringa:

```
Sub NonLatinStrings()

Dim rng As Range
Set rng = Range("A1")
Do Until rng = ""
    Dim MyString As String
    MyString = rng.Value

    ' AscW functions
    Dim char As String
    char = AscW(Left(MyString, 1))
    Debug.Print "First char (ChrW): " & char
    Debug.Print "First char (binary): " & BinaryFormat(char, 12)

    ' ChrW functions
    Dim uString As String
    uString = ChrW(char)
    Debug.Print "String value (text): " & uString        ' Fails! Appears as '?'
    Debug.Print "String value (AscW): " & AscW(uString)

    ' Using a Byte string
    Dim StringAsByt() As Byte
    StringAsByt = MyString
    Dim i As Long
    For i = 0 To 1 Step 2
        Debug.Print "Byte values (in decimal): " & _
            StringAsByt(i) & "|" & StringAsByt(i + 1)
        Debug.Print "Byte values (binary): " & _
            BinaryFormat(StringAsByt(i)) & "|" & BinaryFormat(StringAsByt(i + 1))
    Next i
    Debug.Print ""

    ' Printing the entire string to the immediate window fails (all '?'s)
    Debug.Print "Whole String" & vbNewLine & rng.Value
    Set rng = rng.Offset(1)
```

```
Loop

End Sub
```

Questo produce il seguente risultato per la lettera araba triste :

> Primo carattere (ChrW): 1589
> Primo carattere (binario): 00011000110101
> Valore stringa (testo):?
> Valore stringa (AscW): 1589
> Valori byte (in decimale): 53 | 6
> Valori byte (binari): 00110101 | 00000110
>
> Tutta la stringa
> *??? ????? ????? ??????? ??????? ??? ??????? - ????? ???????? ???? ??????? ???????*

Si noti che VBA non è in grado di stampare testo non latino nella finestra immediata anche se le funzioni stringa funzionano correttamente. Questa è una limitazione dell'IDE e non della lingua.

Identificatori non latini e copertura linguistica

Gli identificatori VBA (nomi di variabili e funzioni) possono utilizzare lo script latino e possono anche essere in grado di utilizzare script in giapponese , coreano , cinese semplificato e cinese tradizionale .

Lo script latino esteso ha una copertura completa per molte lingue:
Inglese, francese, spagnolo, tedesco, italiano, bretone, catalano, danese, estone, finlandese, islandese, indonesiano, irlandese, lojban, mapudungun, norvegese, portoghese, gaelico scozzese, svedese, tagalog

Alcune lingue sono solo parzialmente coperte:
Azero, croato, ceco, esperanto, ungherese, lettone, lituano, polacco, rumeno, serbo, slovacco, sloveno, turco, yoruba, gallese

Alcune lingue hanno una copertura scarsa o nulla:
Arabo, bulgaro, cherokee, dzongkha, greco, hindi, macedone, malayalam, mongolo, russo, sanscrito, thailandese, tibetano, urdu, uiguro

Le seguenti dichiarazioni variabili sono tutte valide:

```
Dim Yec’hed As String 'Breton
Dim «Dóna» As String 'Catalan
Dim fræk As String 'Danish
Dim tšellomängija As String 'Estonian
Dim Törkylempijävongahdus As String 'Finnish
Dim j’examine As String 'French
Dim Paß As String 'German
Dim þjófum As String 'Icelandic
Dim hÓighe As String 'Irish
Dim sofybakni As String 'Lojban (.o’i does not work)
```

```
Dim ñizol As String 'Mapudungun
Dim Vår As String 'Norwegian
Dim «brações» As String 'Portuguese
Dim d’fhàg As String 'Scottish Gaelic
```

Si noti che nell'IDE VBA, un singolo apostrofo all'interno di un nome di variabile non trasforma la linea in un commento (come avviene su Stack Overflow).

Inoltre, le lingue che usano due angoli per indicare una citazione «» possono usare quelle nei nomi di variabili deside- rano il fatto che le virgolette tipo "" non lo sono.

Leggi Personaggi non latini online: https://riptutorial.com/it/vba/topic/10555/personaggi-non-latini

Capitolo 35: Procedura Chiamate

Sintassi

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

Parametri

Parametro	Informazioni
IdentifierName	Il nome della procedura da chiamare.
argomenti	Un elenco di argomenti separati da virgola da passare alla procedura.

Osservazioni

Le prime due sintassi servono per chiamare le procedure `Sub` ; notare che la prima sintassi non prevede parentesi.

Vedi Questo è confuso. Perché non usare sempre le parentesi? per una spiegazione approfondita delle differenze tra le prime due sintassi.

La terza sintassi serve per chiamare le procedure `Function` e `Property Get` ; quando ci sono parametri, le parentesi sono sempre obbligatorie. La parola chiave `Let` è facoltativa quando si assegna un *valore* , ma la parola chiave `Set` è **necessaria** quando si assegna un *riferimento* .

Quarta sintassi per chiamare le procedure `Property Let` e `Property Set` ; l' *`expression`* sul lato destro dell'assegnazione viene passata al parametro del valore della proprietà.

Examples

Sintassi di chiamata implicita

```
ProcedureName
ProcedureName argument1, argument2
```

Chiamare una procedura con il suo nome senza parentesi.

Edge case

La parola chiave `Call` è richiesta solo in un caso limite:

```
Call DoSomething : DoSomethingElse
```

`DoSomething` e `DoSomethingElse` sono le procedure che vengono chiamate. Se la parola chiave `Call` stata rimossa, allora `DoSomething` sarebbe stato analizzato come *un'etichetta di linea* anziché come una chiamata di procedura, che avrebbe interrotto il codice:

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

Valori di ritorno

Per recuperare il risultato di una chiamata di procedura (ad esempio, procedure `Function` o `Property Get`), mettere la chiamata sul lato destro di un incarico:

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

Le parentesi devono essere presenti se ci sono dei parametri. Se la procedura non ha parametri, le parentesi sono ridondanti.

Questo è confusionario. Perché non usare sempre le parentesi?

Le parentesi sono usate per racchiudere gli argomenti delle *chiamate* di *funzione* . Usarli per *le chiamate di procedura* può causare problemi imprevisti.

Perché possono introdurre bug, sia in fase di esecuzione, passando un valore probabilmente non intenzionale alla procedura, sia in fase di compilazione semplicemente essendo una sintassi non valida.

Run-time

Le parentesi ridondanti possono introdurre bug. Data una procedura che accetta un riferimento a un oggetto come parametro ...

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

... e chiamato con parentesi:

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

Ciò solleverà un errore di runtime "Object Required" n. 424. Altri errori sono possibili in altre circostanze: qui il riferimento all'oggetto `Range Application.ActiveCell` viene *valutato* e passato per valore **indipendentemente** dalla firma della procedura che specifica che la `target` sarà passata `ByRef` . Il valore effettivo passato da `ByVal` a `DoSomething` nello snippet sopra riportato è

`Application.ActiveCell.Value` .

Le parentesi costringono VBA a valutare il valore dell'espressione tra parentesi e passano il risultato `ByVal` alla procedura chiamata. Quando il tipo del risultato valutato non corrisponde al tipo previsto della procedura e non può essere convertito in modo implicito, viene generato un errore di runtime.

A tempo di compilazione

Questo codice non riuscirà a compilare:

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

Poiché l'espressione `("Invalid Code!", vbCritical)` non può essere *valutata* su un valore.

Questo potrebbe compilare e lavorare:

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

Ma sicuramente sembrerebbe sciocco. Evitare parentesi ridondanti.

Sintassi di chiamata esplicita

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

La sintassi esplicita della chiamata richiede la parola chiave `Call` e le parentesi attorno all'elenco degli argomenti; le parentesi sono ridondanti se non ci sono parametri. Questa sintassi è stata resa obsoleta quando la sintassi di chiamata implicita più moderna è stata aggiunta a VB.

Argomenti opzionali

Alcune procedure hanno argomenti opzionali. Gli argomenti opzionali vengono sempre dopo gli argomenti richiesti, ma la procedura può essere chiamata senza di essi.

Ad esempio, se la funzione, `ProcedureName` avesse due argomenti richiesti (`argument1` , `argument2`) e un argomento facoltativo, `optArgument3` , potrebbe essere chiamato almeno in quattro modi:

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

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

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

' Mixing named and unnamed arguments
```

```
result = ProcedureName("A", "B", optArgument3:="C")
```

La struttura dell'intestazione della funzione che viene chiamata qui sarà simile a questa:

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

La parola chiave `Optional` indica che questo argomento può essere omesso. Come accennato in precedenza, qualsiasi argomento opzionale introdotto nell'intestazione **deve** apparire alla fine, dopo ogni argomento richiesto.

È inoltre possibile fornire un valore *predefinito* per l'argomento nel caso in cui non venga passato un valore alla funzione:

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

In questa funzione, se l'argomento per `c` non viene fornito, il suo valore verrà impostato su `"C"` . Se *viene* fornito un valore, questo sostituirà il valore predefinito.

Leggi Procedura Chiamate online: https://riptutorial.com/it/vba/topic/1179/procedura-chiamate

Capitolo 36: Ricerca all'interno di stringhe per la presenza di sottostringhe

Osservazioni

Quando è necessario verificare la presenza o la posizione di una sottostringa all'interno di una stringa, VBA offre le funzioni `InStr` e `InStrRev` che restituiscono la posizione del carattere della sottostringa nella stringa, se presente.

Examples

Utilizzare InStr per determinare se una stringa contiene una sottostringa

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

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

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

Utilizzare InStr per trovare la posizione della prima istanza di una sottostringa

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

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

Utilizzare InStrRev per trovare la posizione dell'ultima istanza di una sottostringa

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

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

Leggi Ricerca all'interno di stringhe per la presenza di sottostringhe online:
https://riptutorial.com/it/vba/topic/3480/ricerca-all-interno-di-stringhe-per-la-presenza-di-

sottostringhe

Capitolo 37: ricorsione

introduzione

Si dice che una funzione che chiama se stessa sia *ricorsiva* . Anche la logica ricorsiva può essere implementata come un ciclo. La ricorsione deve essere controllata con un parametro, in modo che la funzione sappia quando interrompere la ricorsione e approfondire lo stack di chiamate. *La ricorsione infinita* alla fine causa un errore di run-time "28": "Out of stack space".

Vedi Ricorsione .

Osservazioni

La ricorsione consente chiamate ripetute e autoreferenziali di una procedura.

Examples

fattoriali

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

Ricorsione cartella

Early Bound (con riferimento a `Microsoft Scripting Runtime`)

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

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

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

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

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

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

Leggi ricorsione online: https://riptutorial.com/it/vba/topic/3236/ricorsione

Capitolo 38: Scripting. Oggetto letterario

Osservazioni

È necessario aggiungere Microsoft Scripting Runtime al progetto VBA tramite il comando Strumenti → Riferimenti di VBE per implementare l'associazione anticipata dell'oggetto Dizionario di script. Questo riferimento bibliografico è portato con il progetto; non deve essere rinviato quando il progetto VBA viene distribuito ed eseguito su un altro computer.

Examples

Proprietà e metodi

Un oggetto Scripting Dictionary memorizza le informazioni nelle coppie chiave / oggetto. Le chiavi devono essere uniche e non un array, ma gli articoli associati possono essere ripetuti (la loro unicità è detenuta dalla chiave companion) e possono essere di qualsiasi tipo di variante o oggetto.

Un dizionario può essere considerato come un database di due campi in memoria con un indice univoco primario sul primo 'campo' (la *chiave*). Questo indice univoco sulla proprietà Keys consente "ricerche" molto rapide per recuperare il valore dell'articolo associato di una chiave.

Proprietà

nome	leggere scrivere	genere	descrizione
CompareMode	*leggere scrivere*	CompareMode costante	L'impostazione di CompareMode può essere eseguita solo su un dizionario vuoto. I valori accettati sono 0 (vbBinaryCompare), 1 (vbTextCompare), 2 (vbDatabaseCompare).
Contare	*sola lettura*	intero lungo senza segno	Un conteggio basato su un singolo delle coppie chiave / articolo nell'oggetto del dizionario di scripting.
Chiave	*leggere scrivere*	variante non array	Ogni singola chiave univoca nel dizionario.
Articolo (*chiave*)	*leggere scrivere*	qualsiasi variante	Proprietà di default. Ogni singolo elemento associato a una chiave nel dizionario. Si noti che il tentativo di recuperare un elemento con una chiave che non esiste nel dizionario *aggiungerà implicitamente* la chiave passata.

---

metodi

nome	descrizione
Aggiungi (*chiave* , *articolo*)	Aggiunge una nuova chiave ed elemento al dizionario. La nuova chiave non deve esistere nella collezione Keys corrente del dizionario, ma un elemento può essere ripetuto tra molte chiavi univoche.
Esiste (*chiave*)	Test booleano per determinare se una chiave esiste già nel dizionario.
chiavi	Restituisce la matrice o la raccolta di chiavi univoche.
Elementi	Restituisce la matrice o la raccolta di elementi associati.
Rimuovi (*chiave*)	Rimuove una singola chiave del dizionario e il relativo oggetto associato.
Rimuovi tutto	Cancella tutte le chiavi e gli oggetti di un dizionario.

Codice di esempio

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

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

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

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

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

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

End Sub

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

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

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

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

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

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

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

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

End Sub
```

Dati aggregati con Scripting.Dictionary (Maximum, Count)

I dizionari sono grandi per la gestione delle informazioni in cui si verificano più voci, ma si è interessati solo a un singolo valore per ogni insieme di voci: il primo o l'ultimo valore, il mininmum o il valore massimo, una media, una somma, ecc.

Considera una cartella di lavoro che contiene un registro dell'attività dell'utente, con uno script che inserisce il nome utente e modifica la data ogni volta che qualcuno modifica la cartella di lavoro:

`Log` foglio di lavoro

UN	B
peso	10/12/2016 9:00
alice	13/10/2016 13:00

UN	B
peso	13/10/2016 13:30
alice	13/10/2016 14:00
alice	14/10/2016 13:00

Supponiamo di voler emettere l'ultima ora di modifica per ciascun utente in un foglio di lavoro denominato `Summary` .

Gli appunti:

1. Si presume che i dati siano in `ActiveWorkbook` .
2. Stiamo usando una matrice per estrarre i valori dal foglio di lavoro; questo è più efficiente di iterare su ogni cella.
3. Il `Dictionary` viene creato utilizzando l'associazione anticipata.

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

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

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

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

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

e l'output sarà simile a questo:

Foglio di lavoro `Summary`

UN	B
peso	13/10/2016 13:30
alice	14/10/2016 13:00

Se invece vuoi mostrare quante volte ogni utente ha modificato la cartella di lavoro, il corpo del ciclo `For` dovrebbe apparire così:

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

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

e l'output sarà simile a questo:

Foglio di lavoro `Summary`

UN	B
peso	2
alice	3

Ottenere valori univoci con Scripting.Dictionary

Il `Dictionary` consente di ottenere un insieme di valori unico in modo molto semplice. Considera la seguente funzione:

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

che puoi chiamare così:

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

e `uniqueVals` conterrebbe solo `{1,2,3}` .

Nota: questa funzione può essere utilizzata con qualsiasi oggetto enumerabile.

Leggi Scripting. Oggetto letterario online: https://riptutorial.com/it/vba/topic/3667/scripting--oggetto-letterario

Capitolo 39: Scripting.FileSystemObject

Examples

Creazione di un oggetto FileSystemObject

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

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

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

Lettura di un file di testo utilizzando un FileSystemObject

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

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

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

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

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

Creazione di un file di testo con FileSystemObject

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

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

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

Scrivere su un file esistente con FileSystemObject

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


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

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

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

Enumera i file in una directory usando FileSystemObject

Limitazione anticipata (richiede un riferimento a Microsoft Scripting Runtime):

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

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

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

Ritardato:

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

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

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

Elenca in modo ricorsivo cartelle e file

Early Bound (con riferimento a `Microsoft Scripting Runtime`)

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

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

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

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

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

```
            End If
        Next subFldr

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

Output quando chiamato con argomenti come: `EnumerateFilesAndFolders "C:\Test"`

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

Output quando chiamato con argomenti come: `EnumerateFilesAndFolders "C:\Test", 0`

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

Output quando chiamato con argomenti come: `EnumerateFilesAndFolders "C:\Test", 1, 4`

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

Estrai l'estensione del file da un nome file

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

Stampa `MyFile.something`

Si noti che il metodo `GetBaseName()` gestisce già più periodi in un nome file.

Recupera solo l'estensione da un nome file

```
Dim fso As New Scripting.FileSystemObject
```

```
Debug.Print fso.GetExtensionName("MyFile.something.txt")
```

Stampa `txt` Si noti che il `GetExtensionName()` gestisce già più periodi in un nome file.

Recupera solo il percorso da un percorso file

Il metodo GetParentFolderName restituisce la cartella principale per qualsiasi percorso. Anche se questo può essere utilizzato anche con le cartelle, è probabilmente più utile per estrarre il percorso da un percorso di file assoluto:

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

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

Si noti che il separatore del percorso finale non è incluso nella stringa restituita.

Utilizzo di FSO.BuildPath per creare un percorso completo dal percorso della cartella e dal nome del file

Se stai accettando l'input dell'utente per i percorsi delle cartelle, potresti dover controllare le barre retroverse (`\`) prima di creare un percorso file. Il metodo `FSO.BuildPath` semplifica la procedura:

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

  Const fileName As String = "Results.txt"

  Dim FSO As FileSystemObject
  Set FSO = New FileSystemObject

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

Produzione:

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

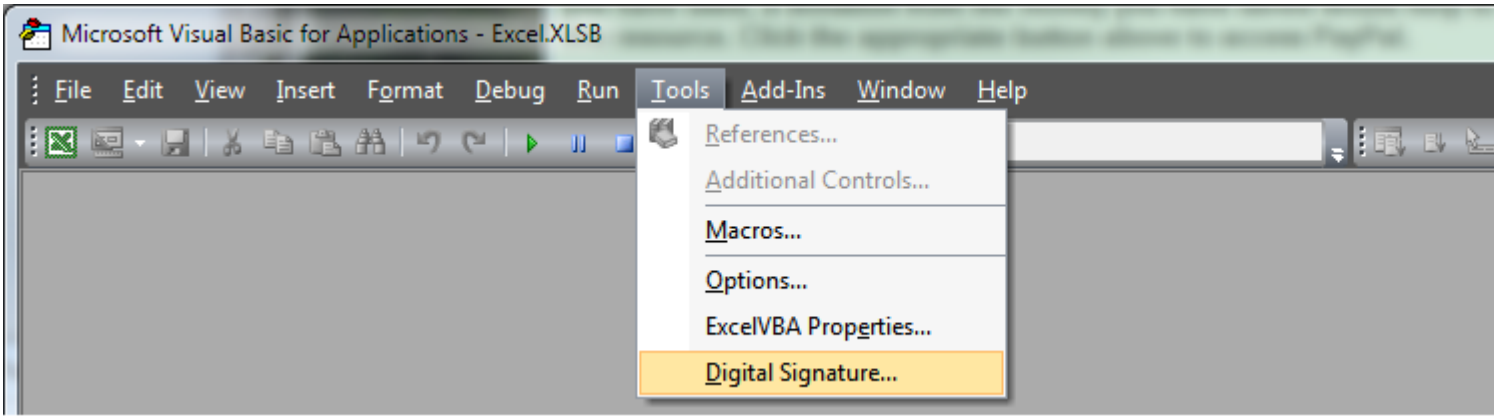
Leggi Scripting.FileSystemObject online: https://riptutorial.com/it/vba/topic/990/scripting-filesystemobject

Capitolo 40: Sicurezza macro e firma di progetti / moduli VBA

Examples

Creare un certificato autofirmato digitale valido SELFCERT.EXE

Per eseguire macro e mantenere le applicazioni di sicurezza Office fornite contro il codice dannoso, è necessario firmare digitalmente VBAProject.OTM dall'editor *VBA> Strumenti> Firma digitale* .

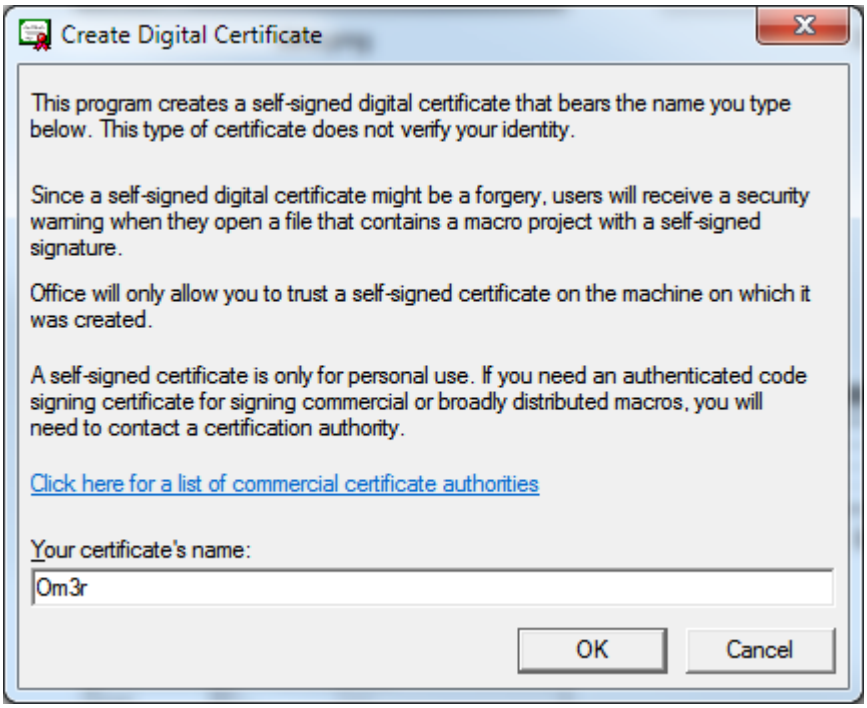


Office è dotato di un'utilità per creare un certificato digitale autofirmato che puoi utilizzare sul PC per firmare i tuoi progetti.

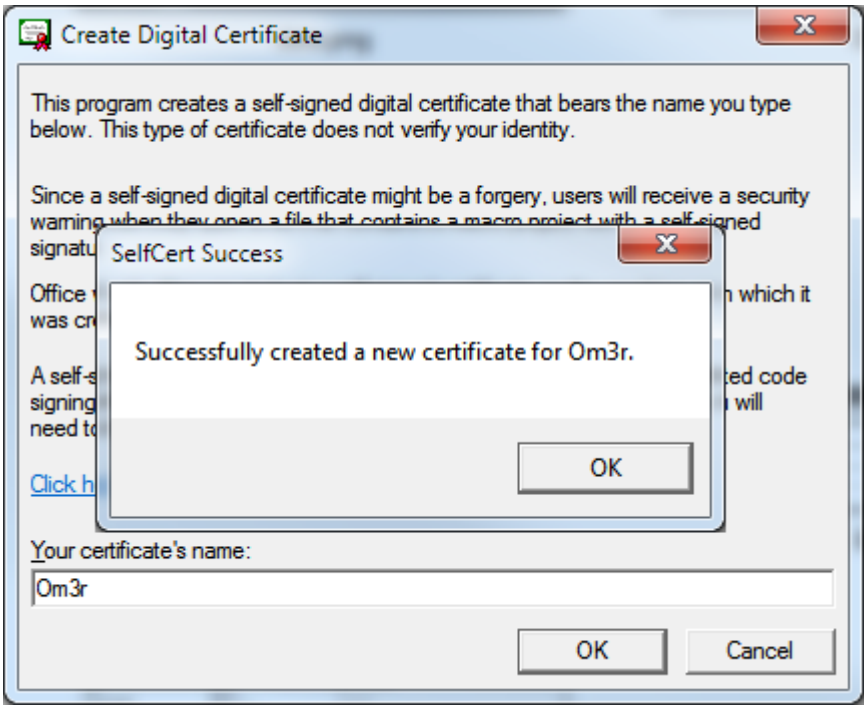
Questa utilità **SELFCERT.EXE si** trova nella cartella del programma di Office,

Fare clic su Certificato digitale per progetti VBA per aprire la *procedura guidata* del certificato.

Nella finestra di dialogo inserire un nome adatto per il certificato e fare clic su OK.

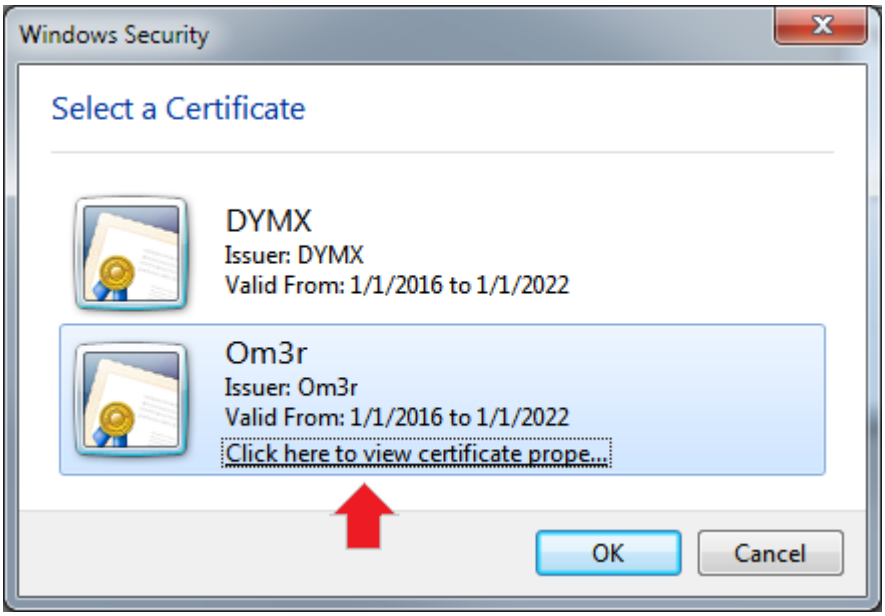


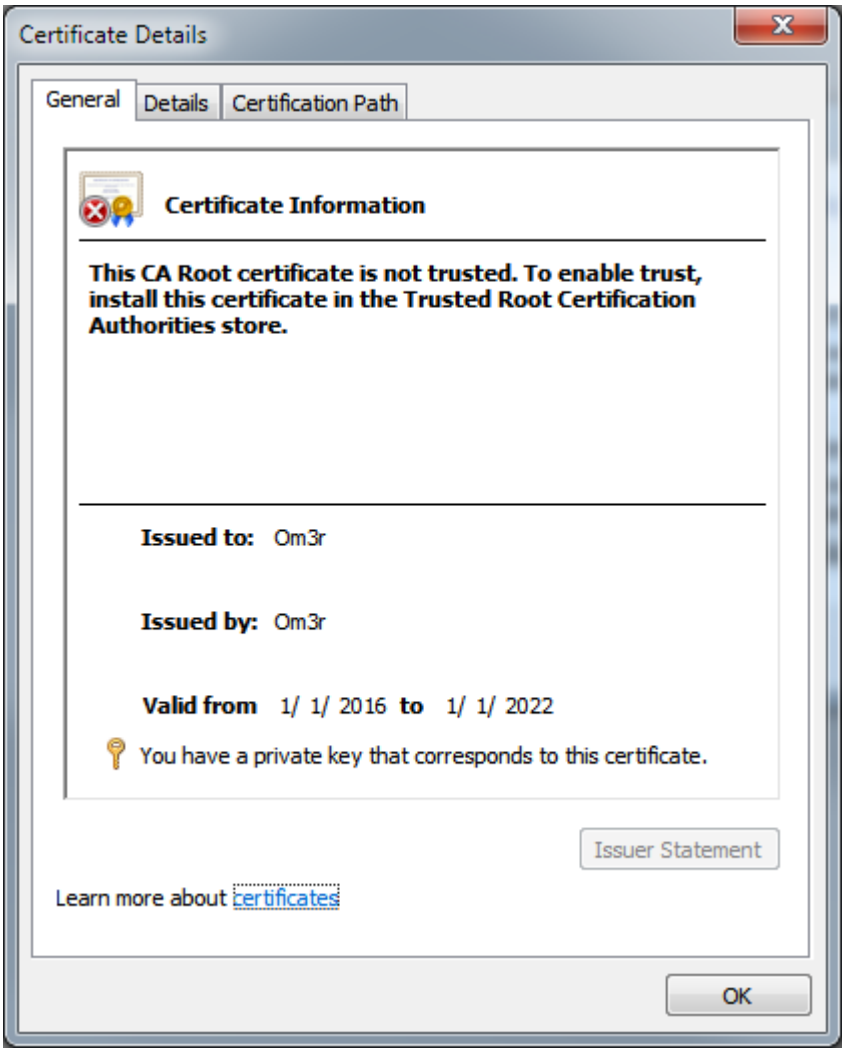
Se tutto va bene, vedrai una conferma:



Ora puoi chiudere la procedura guidata **SELFCERT** e rivolgere la tua attenzione al certificato che hai creato.

Se cerchi di utilizzare il certificato che hai appena creato e ne controlli le proprietà



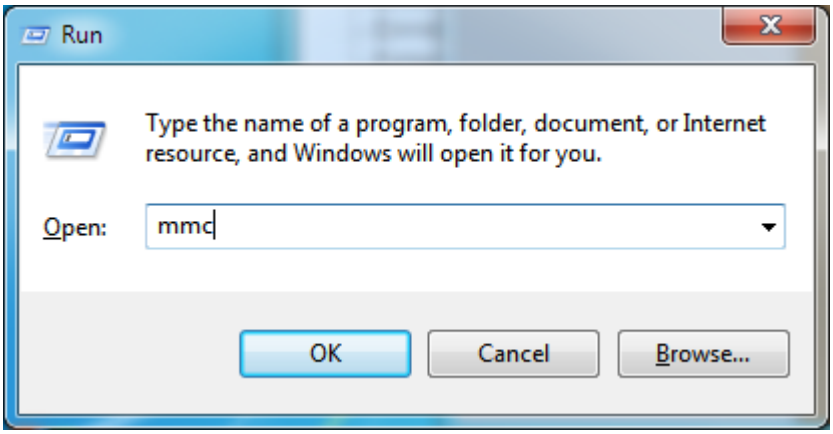


Vedrai che il certificato non è attendibile e il motivo è indicato nella finestra di dialogo.

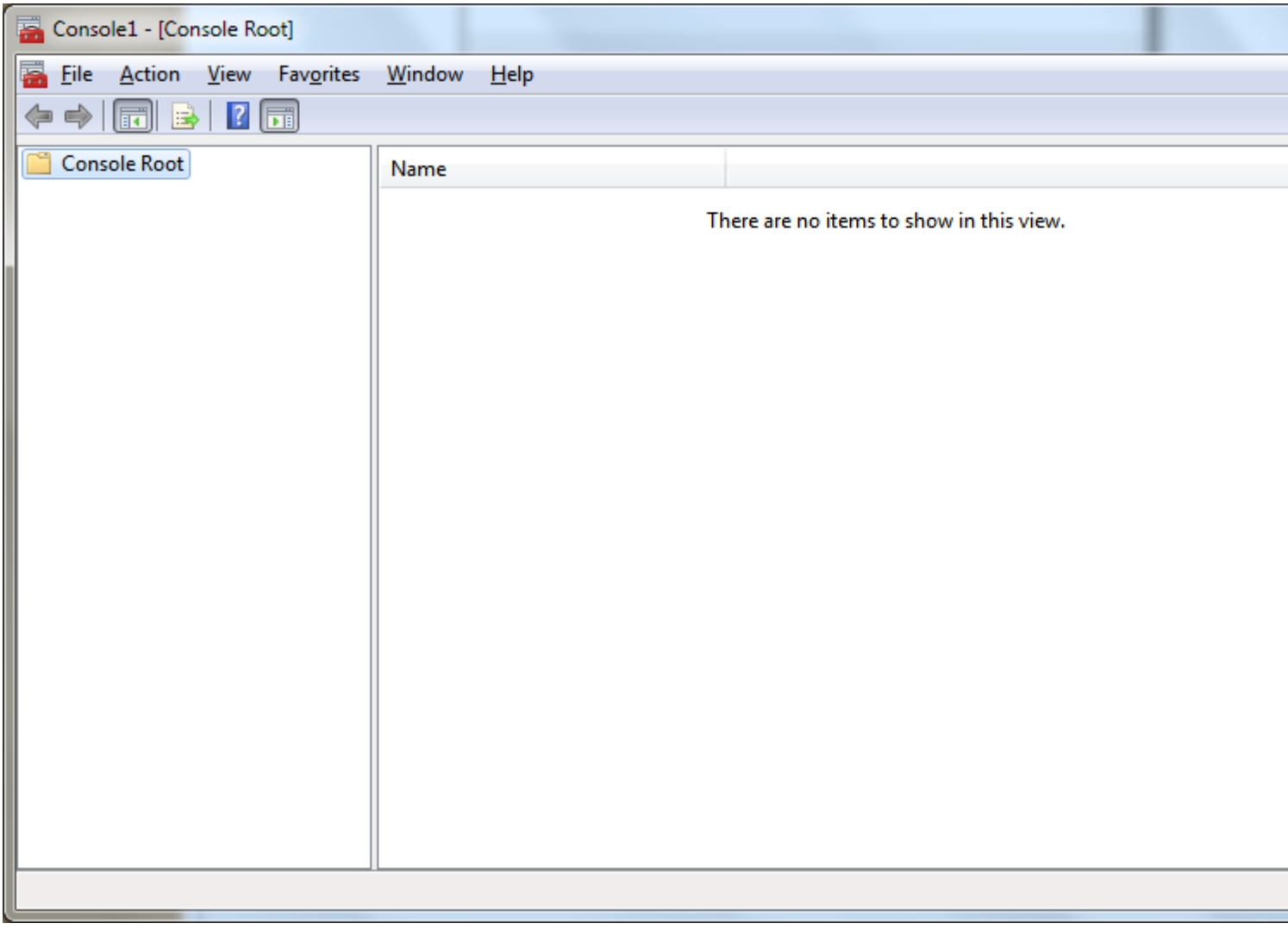
Il certificato è stato creato nell'archivio Utente corrente> Personale> Certificati. Deve andare in Computer locale> Autorità di certificazione radice attendibile> Archivio certificati, quindi è necessario esportare dal precedente e importarlo nel secondo.

Premendo il `tasto` Windows `+ R` che aprirà la finestra 'Esegui'. quindi immettere 'mmc' nella

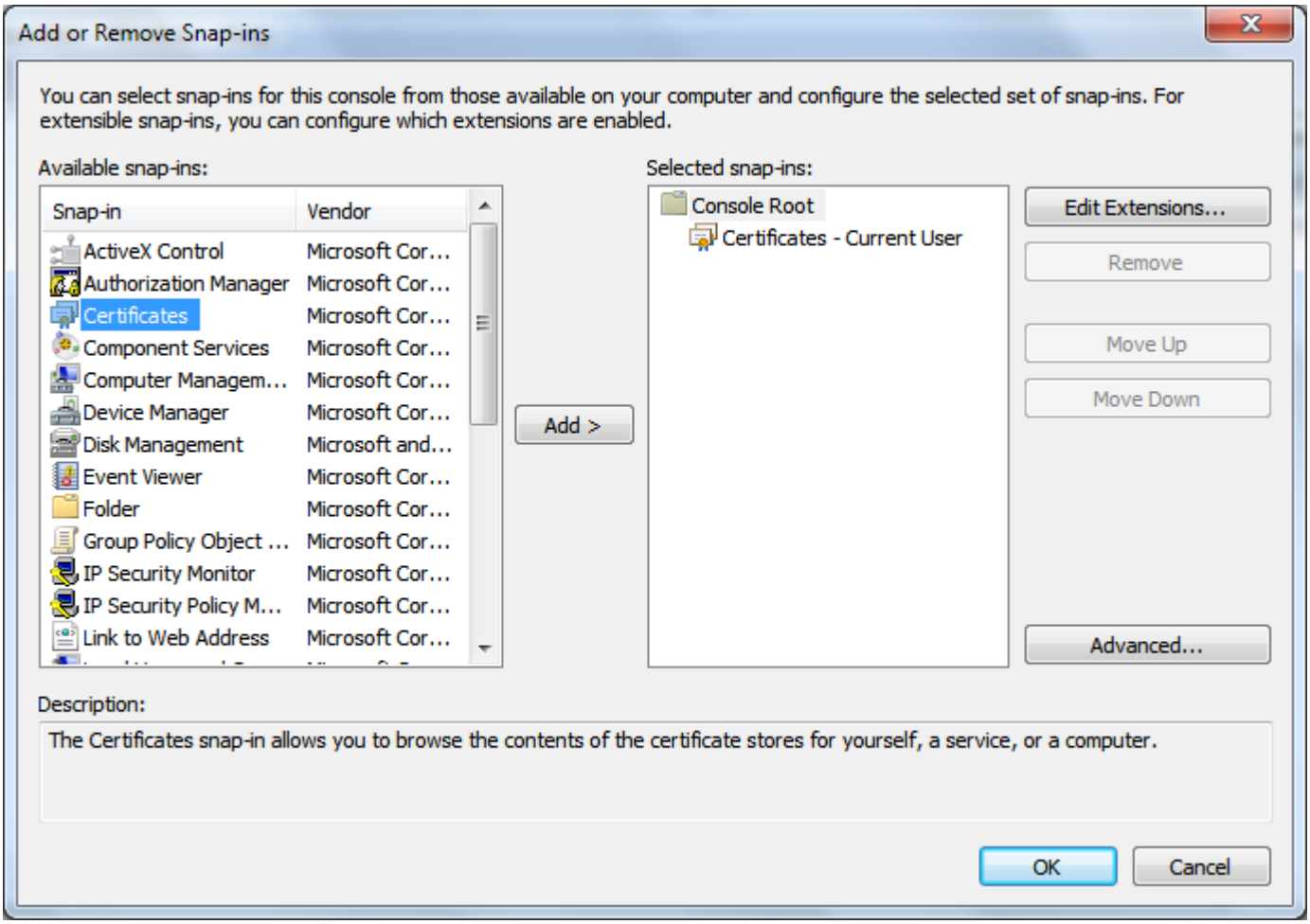
finestra come mostrato di seguito e fare clic su 'OK'.



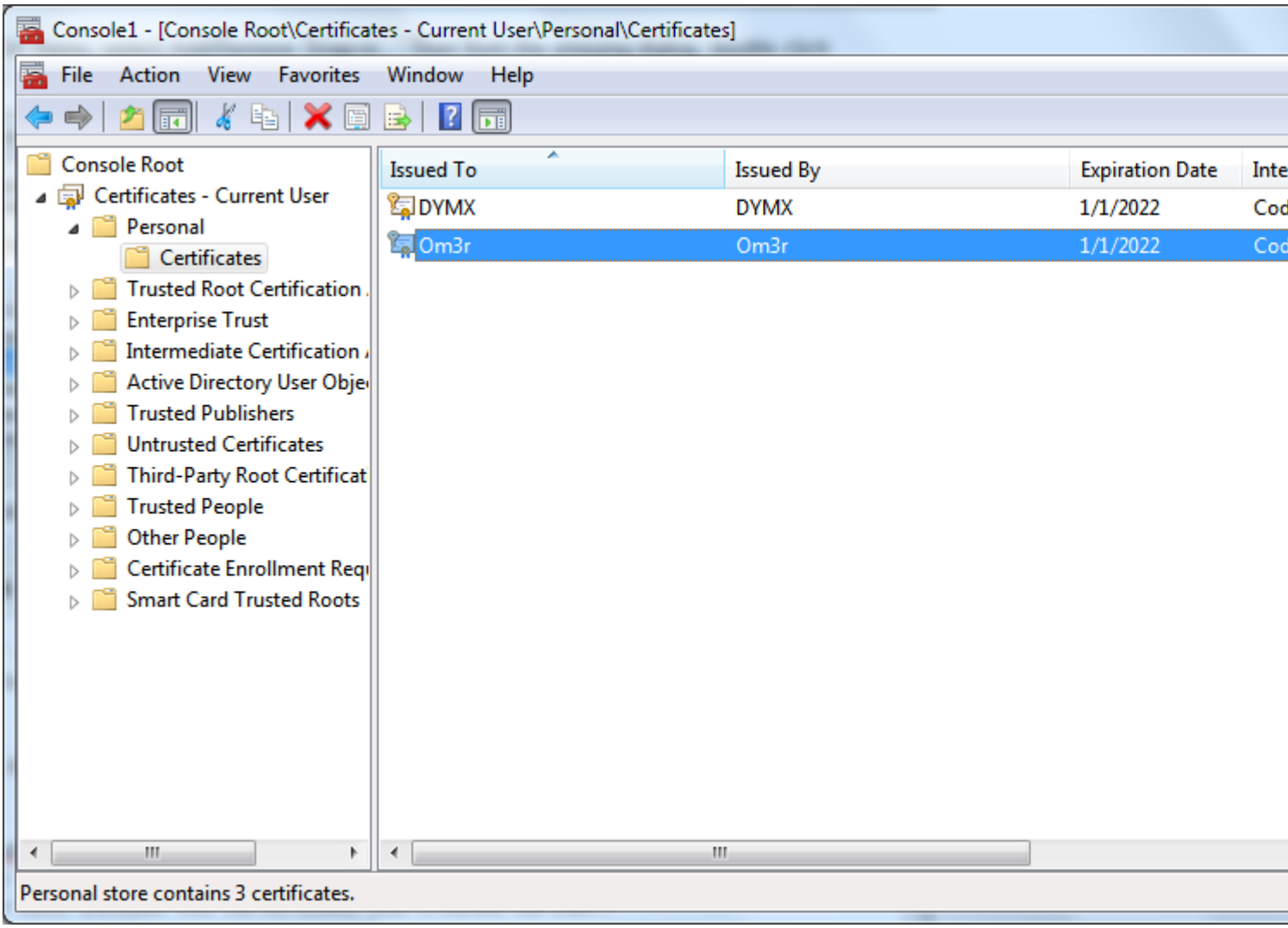
Microsoft Management Console si aprirà e avrà il seguente aspetto.



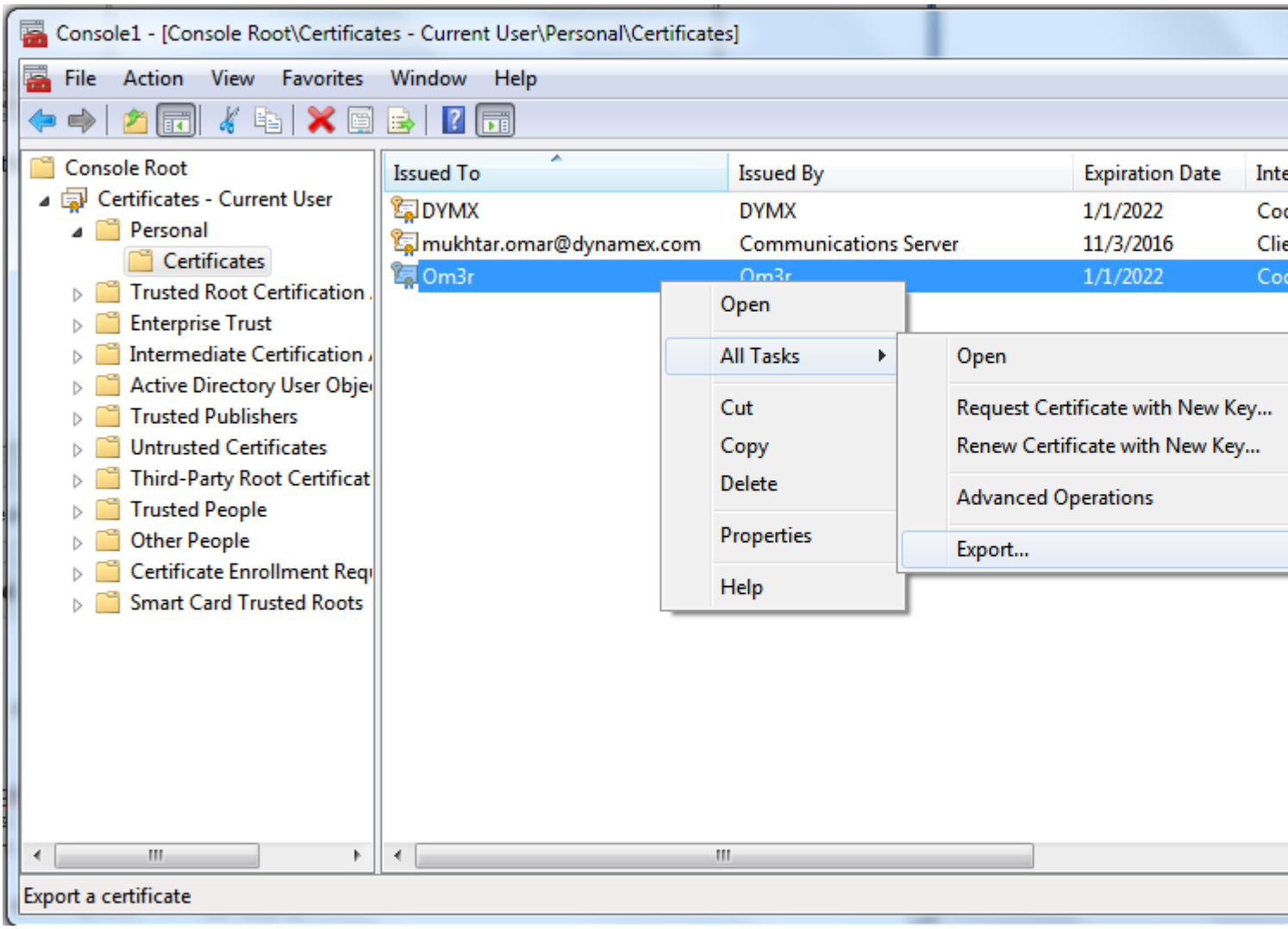
Dal menu File, selezionare Aggiungi / Rimuovi snap-in ... Quindi dalla finestra di dialogo seguente, fare doppio clic su Certificati e quindi fare clic su OK



Espandi il menu a discesa nella finestra di sinistra per *Certificati - Utente corrente* 'e seleziona i certificati come mostrato di seguito. Il pannello centrale mostrerà quindi i certificati in quella posizione, che includeranno il certificato creato in precedenza:

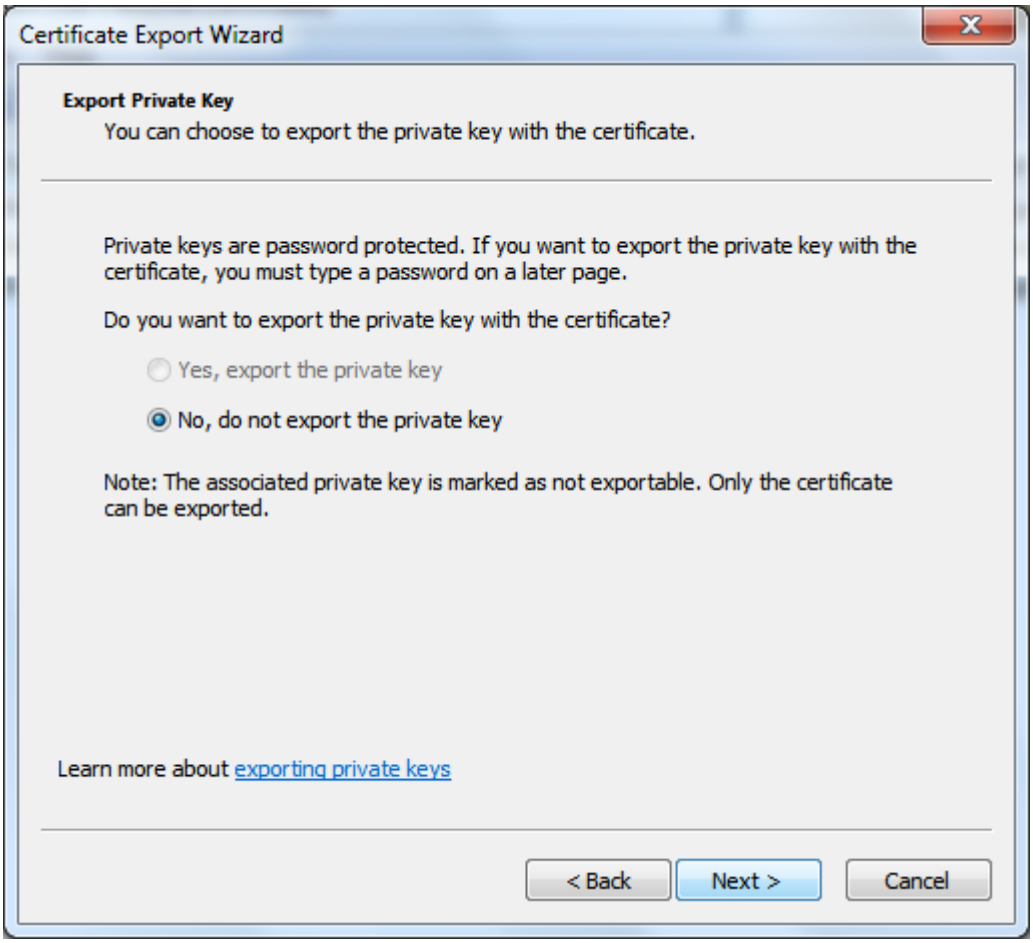


Fare clic con il tasto destro sul certificato e selezionare Tutte le attività> Esporta:

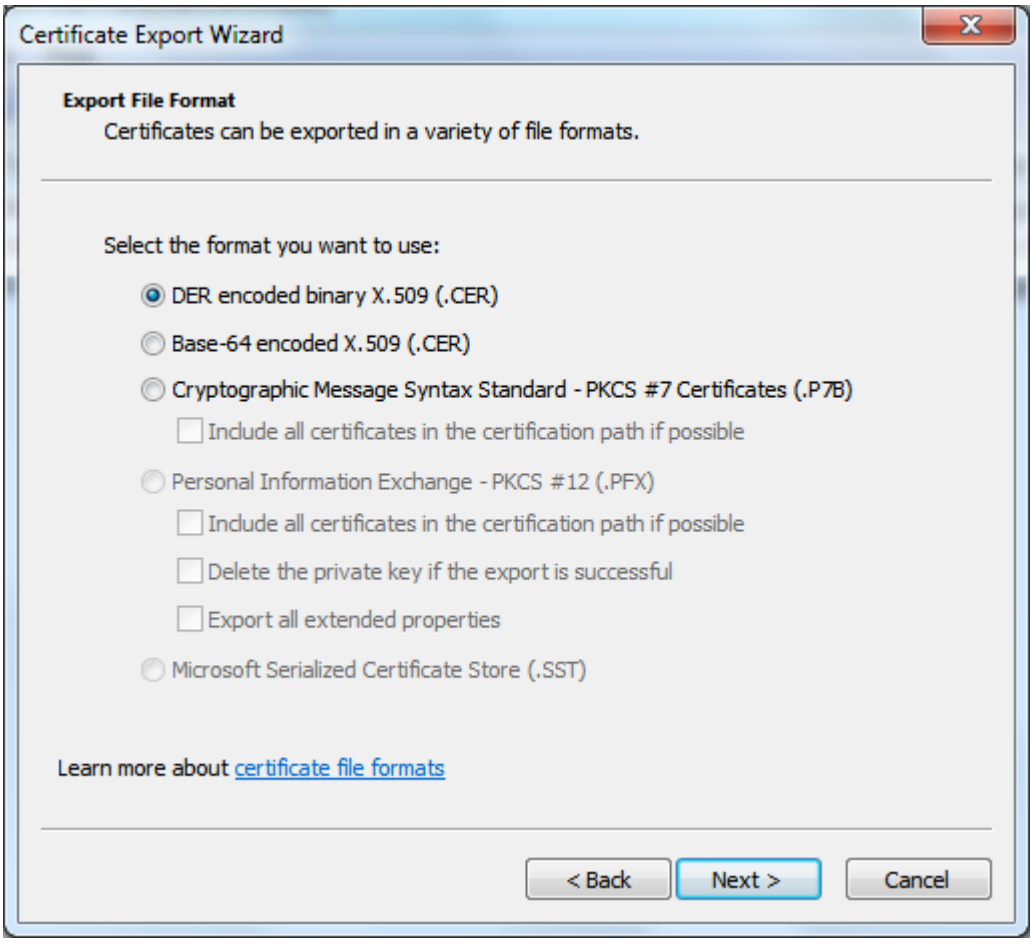


Esportazione guidata

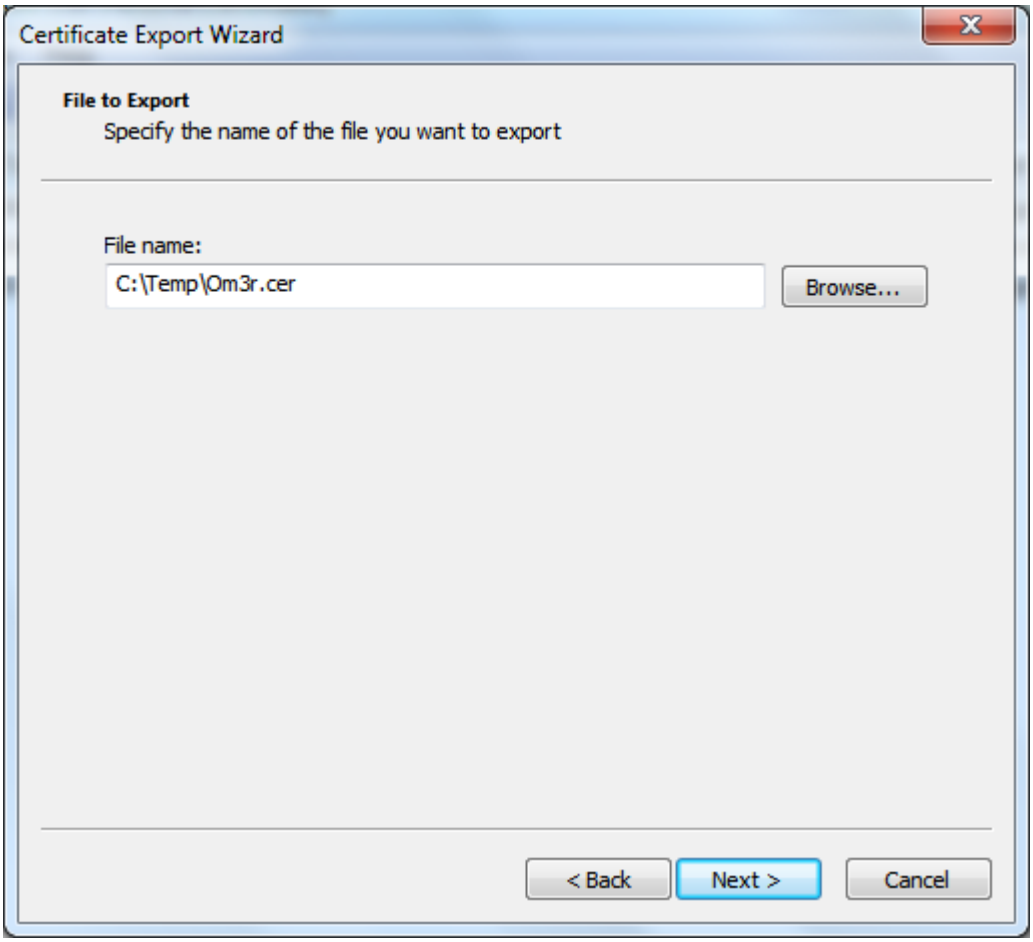
Fare clic su Avanti



l'unica opzione preselezionata sarà disponibile, quindi fai nuovamente clic su "Avanti":



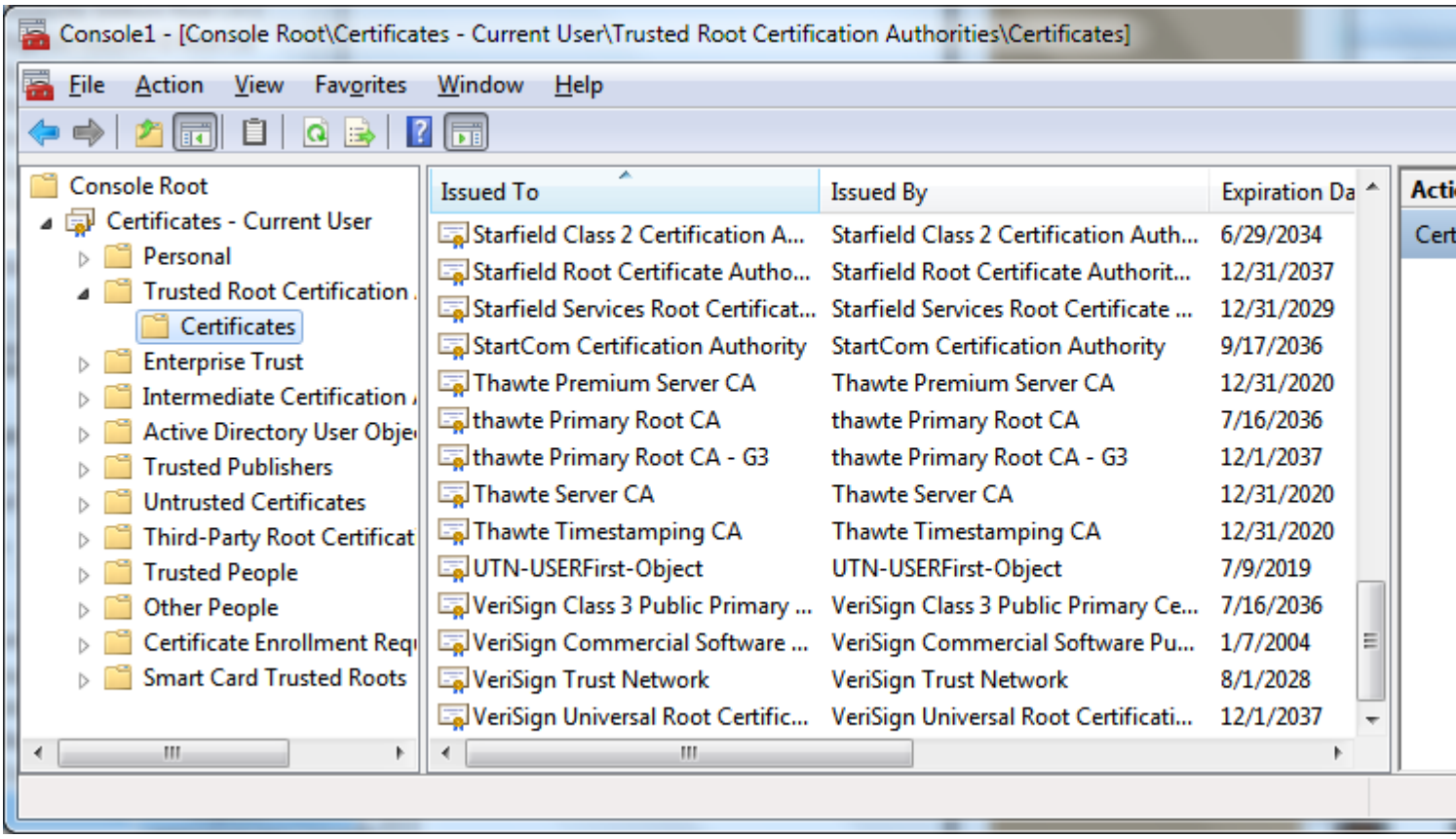
L'oggetto in cima sarà già preselezionato. Fare nuovamente clic su Avanti e scegliere un nome e una posizione per salvare il certificato esportato.



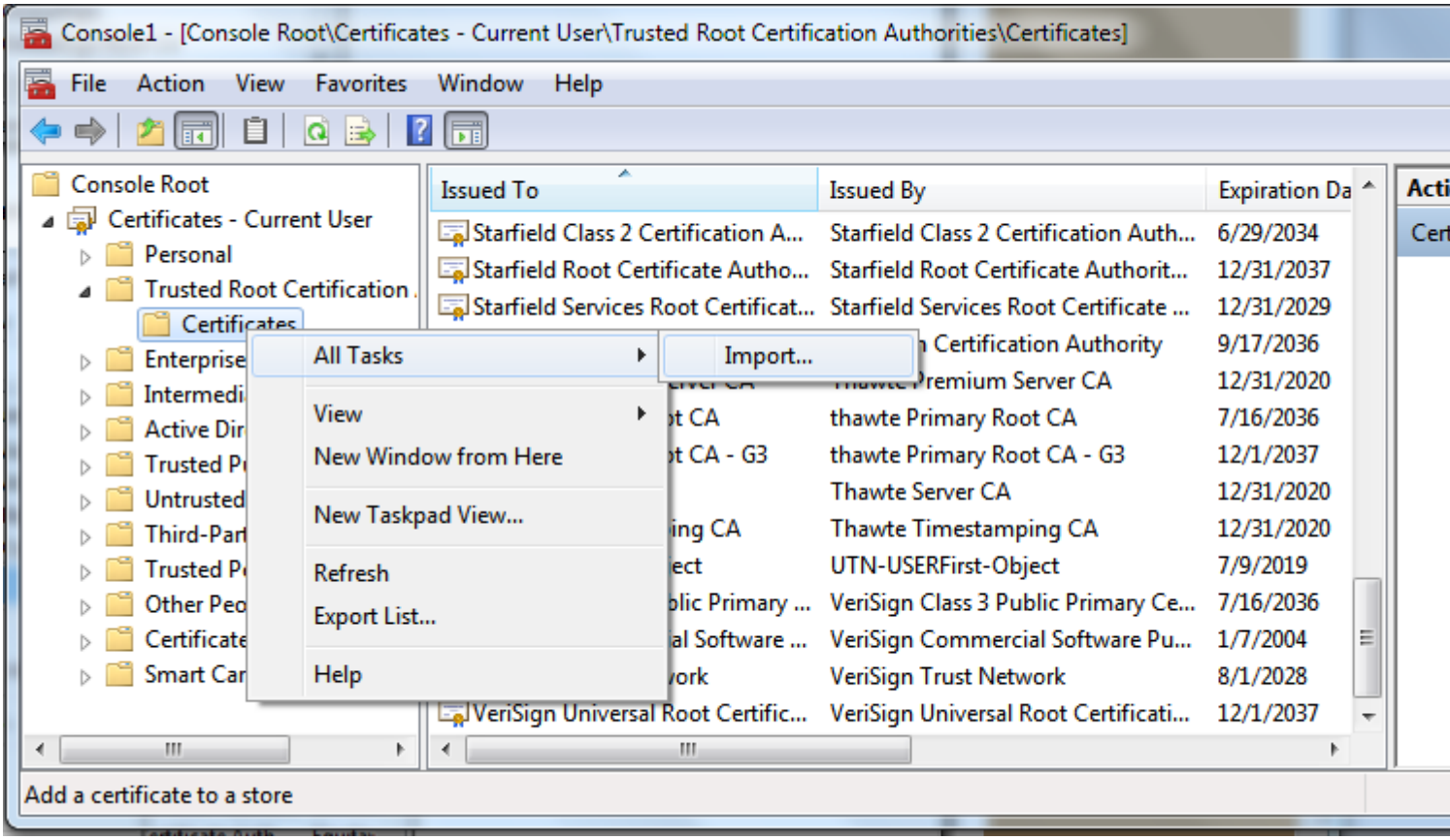
Fare nuovamente clic su Avanti per salvare il certificato

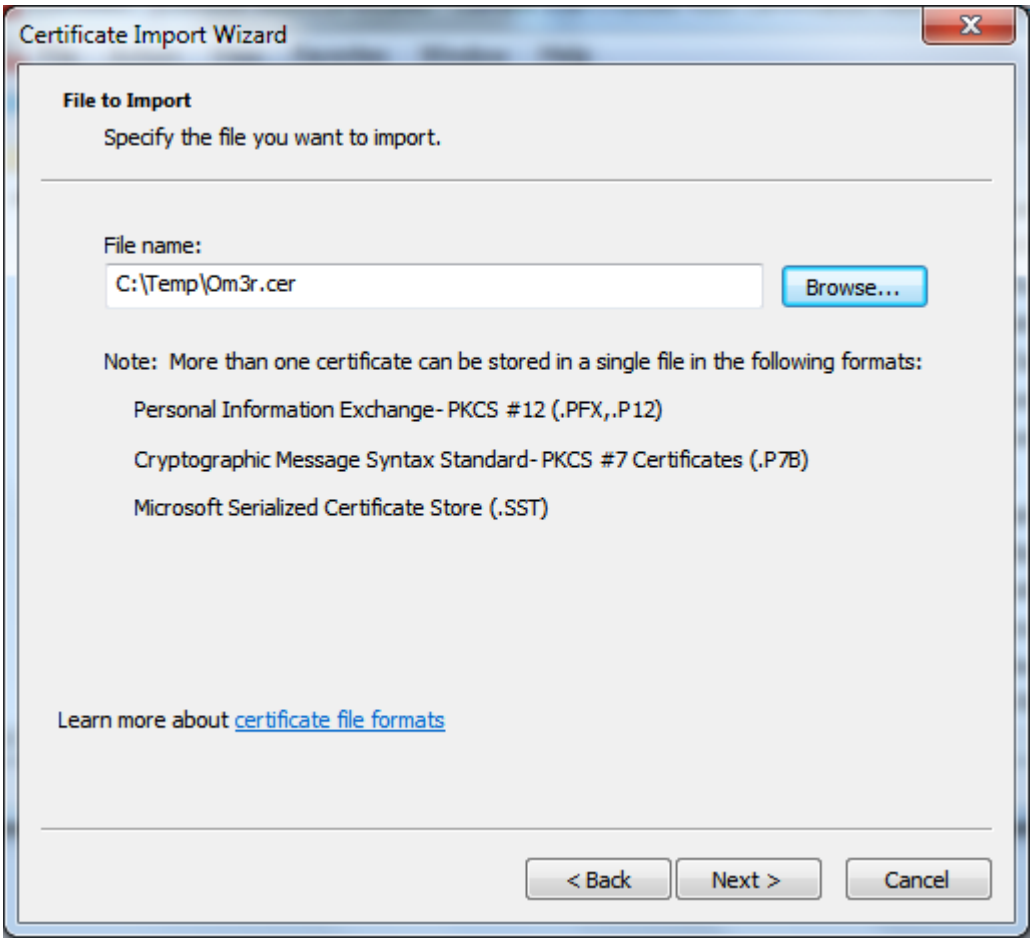
Una volta che lo stato attivo viene restituito alla console di gestione.

Espandere il menu *Certificati* e dal menu Autorità di certificazione radice attendibili, selezionare *Certificati* .

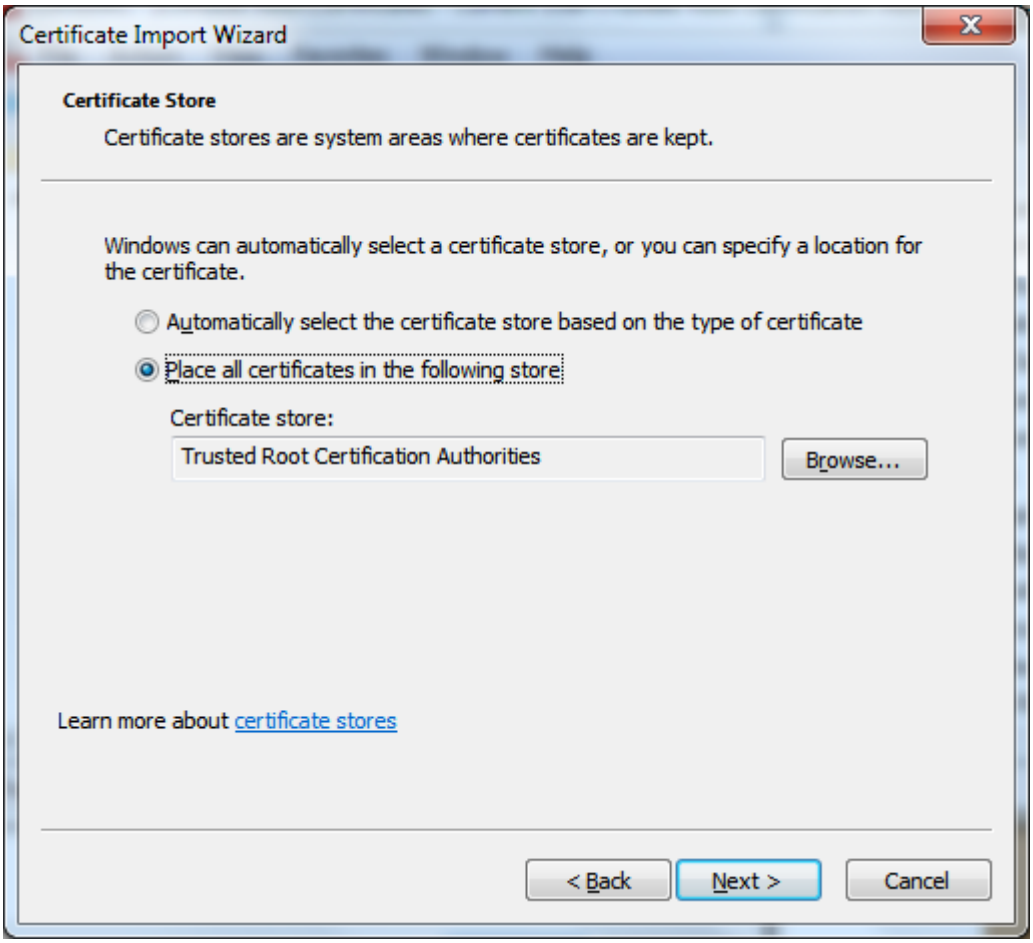


Clic destro. Seleziona *Tutte le attività* e *importa*



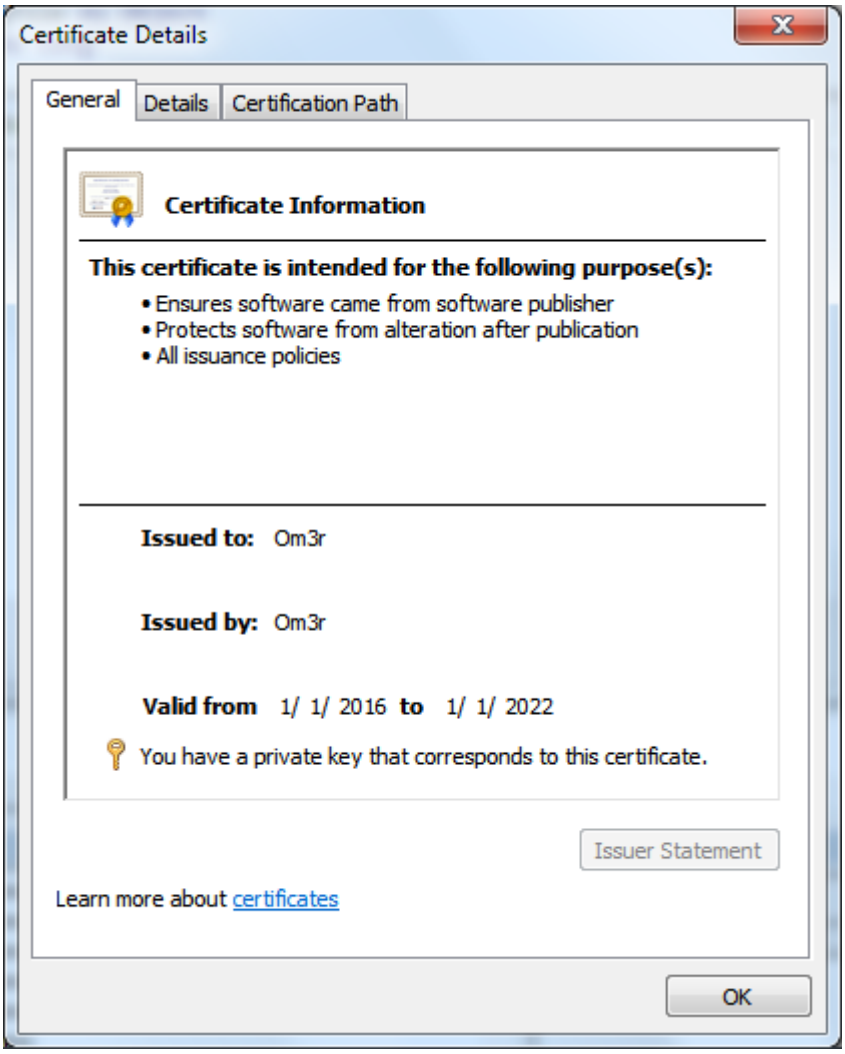


Fare clic su Avanti e Salva *nell'archivio Autorità di certificazione fonti attendibili* :



Quindi Avanti> Fine, ora chiudi la Console.

Se ora usi il certificato e controlli le sue proprietà, vedrai che si tratta di un certificato attendibile e puoi usarlo per firmare il tuo progetto:



Leggi Sicurezza macro e firma di progetti / moduli VBA online:
https://riptutorial.com/it/vba/topic/7733/sicurezza-macro-e-firma-di-progetti---moduli-vba

Capitolo 41: sottostringhe

Osservazioni

VBA ha funzioni integrate per estrarre parti specifiche di stringhe, tra cui:

- `Left` / `Left$`
- `Right` / `Right$`
- `Mid` / `Mid$`
- `Trim` / `Trim$`

Per evitare la conversione di tipo implicita su onhead (e quindi per prestazioni migliori), utilizzare la versione $ -suffixed della funzione quando una variabile stringa viene passata alla funzione e / o se il risultato della funzione è assegnato a una variabile stringa.

Passare un valore di parametro `Null` a una funzione $ -suffixed solleverà un errore di runtime ("uso non valido di null") - questo è particolarmente rilevante per il codice che coinvolge un database.

Examples

Usa sinistra o sinistra $ per ottenere i 3 caratteri più a sinistra in una stringa

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

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

Usa Destra o Destra $ per ottenere i 3 caratteri più a destra in una stringa

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

Usa Mid o Mid $ per ottenere caratteri specifici all'interno di una stringa

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

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

Usa Trim per ottenere una copia della stringa senza spazi iniziali o finali

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

Leggi sottostringhe online: https://riptutorial.com/it/vba/topic/3481/sottostringhe

Capitolo 42: String letterali: caratteri di escape, non stampabili e continuazioni di riga

Osservazioni

L'assegnazione di stringhe letterali in VBA è limitata dalle limitazioni dell'IDE e dalla codepage delle impostazioni della lingua dell'utente corrente. Gli esempi precedenti dimostrano i casi speciali di stringhe con escape, stringhe speciali non stampabili e stringhe letterali lunghe.

Quando si assegnano stringhe letterali contenenti caratteri specifici di una determinata tabella codici, potrebbe essere necessario prendere in considerazione problemi di internazionalizzazione assegnando una stringa da un file di risorse Unicode separato.

Examples

Sfuggire al "personaggio"

La sintassi VBA richiede che una stringa letterale appaia all'interno di `"` marks `"` , quindi quando la stringa deve *contenere* virgolette, è necessario sfuggire / anteporre `"` carattere con un extra `"` modo che VBA capisca che si intende che `""` sia interpretato come una `"` stringa `"` .

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

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

Assegnazione di valori letterali stringa lunghi

L'editor VBA consente solo 1023 caratteri per riga, ma in genere solo i primi 100-150 caratteri sono visibili senza scorrimento. Se è necessario assegnare valori letterali stringa lunghi, ma si desidera mantenere il codice leggibile, è necessario utilizzare la continuazione della riga e la concatenazione per assegnare la stringa.

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

VBA ti consente di utilizzare un numero limitato di continuazioni di riga (il numero effettivo varia in base alla lunghezza di ogni riga all'interno del blocco continua), quindi se hai stringhe molto lunghe, dovrai assegnarle e riassegnarle con la concatenazione .

```
Dim loremIpsum As String

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

Debug.Print loremIpsum

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

Utilizzo delle costanti di stringa VBA

VBA definisce un numero di costanti di stringa per caratteri speciali come:

- vbCr: carriage-Return 'Uguale a "\ r" nelle lingue in stile C.
- vbLf: Line-Feed 'Uguale a "\ n" nelle lingue in stile C.
- vbCrLf: Carriage-Return & Line-Feed (una nuova riga in Windows)
- vbTab: Tab Character
- vbNullString: una stringa vuota, come ""

È possibile utilizzare queste costanti con concatenazione e altre funzioni di stringa per creare stringhe letterali con caratteri speciali.

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

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

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

L'utilizzo di `vbNullString` è considerato una pratica migliore rispetto al valore equivalente di `""` causa delle differenze nella modalità di compilazione del codice. Le stringhe sono accessibili tramite un puntatore a un'area allocata della memoria e il compilatore VBA è abbastanza intelligente da utilizzare un puntatore nullo per rappresentare `vbNullString` . Il letterale `""` è allocata alla memoria come se fosse una variante Variant tipizzata, rendendo l'uso della costante molto più

efficiente:

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

Leggi String letterali: caratteri di escape, non stampabili e continuazioni di riga online: https://riptutorial.com/it/vba/topic/3445/string-letterali--caratteri-di-escape--non-stampabili-e-continuazioni-di-riga

Capitolo 43: Strutture dati

introduzione

[TODO: Questo argomento dovrebbe essere un esempio di tutte le strutture di dati CS 101 di base insieme ad alcune spiegazioni come una panoramica di come le strutture dati possono essere implementate in VBA. Questa sarebbe una buona opportunità per legare e rafforzare i concetti introdotti negli argomenti relativi alla Classe nella documentazione VBA.]

Examples

Lista collegata

Questo esempio di elenco collegato implementa le operazioni sui tipi di dati astratti .

Classe **SinglyLinkedNode**

```
Option Explicit

Private Value As Variant
Private NextNode As SinglyLinkedNode '"Next" is a keyword in VBA and therefore is not a valid
variable name
```

Classe **LinkedList**

```
Option Explicit

Private head As SinglyLinkedNode

'Set type operations

Public Sub Add(value As Variant)
    Dim node As SinglyLinkedNode

    Set node = New SinglyLinkedNode
    node.value = value
    Set node.nextNode = head

    Set head = node
End Sub

Public Sub Remove(value As Variant)
    Dim node As SinglyLinkedNode
    Dim prev As SinglyLinkedNode

    Set node = head

    While Not node Is Nothing
        If node.value = value Then
            'remove node
            If node Is head Then
                Set head = node.nextNode
```

```
            Else
                Set prev.nextNode = node.nextNode
            End If
            Exit Sub
        End If
        Set prev = node
        Set node = node.nextNode
    Wend

End Sub

Public Function Exists(value As Variant) As Boolean
    Dim node As SinglyLinkedNode

    Set node = head
    While Not node Is Nothing
        If node.value = value Then
            Exists = True
            Exit Function
        End If
        Set node = node.nextNode
    Wend
End Function

Public Function Count() As Long
    Dim node As SinglyLinkedNode

    Set node = head

    While Not node Is Nothing
        Count = Count + 1
        Set node = node.nextNode
    Wend

End Function
```

Albero binario

Questo è un esempio di albero di ricerca binaria sbilanciato. Un albero binario è strutturato concettualmente come una gerarchia di nodi discendente verso il basso da una radice comune, in cui ogni nodo ha due figli: sinistra e destra. Ad esempio, supponiamo che i numeri 7, 5, 9, 3, 11, 6, 12, 14 e 15 siano stati inseriti in un BinaryTree. La struttura sarebbe come qui sotto. Si noti che questo albero binario non è bilanciato , il che può essere una caratteristica auspicabile per garantire le prestazioni delle ricerche - vedi gli alberi AVL per un esempio di albero di ricerca binaria autobilanciante.

```
            7
           / \
          5   9
         / \   \
        3   6   11
                  \
                   12
                     \
                      14
                       \
                        15
```

Classe **BinaryTreeNode**

```
Option Explicit

Public left As BinaryTreeNode
Public right As BinaryTreeNode
Public key As Variant
Public value As Variant
```

Classe **BinaryTree**

[FARE]

Leggi Strutture dati online: https://riptutorial.com/it/vba/topic/8628/strutture-dati

Capitolo 44: Strutture di controllo del flusso

Examples

Seleziona caso

`Select Case` può essere utilizzato quando sono possibili molte condizioni diverse. Le condizioni vengono controllate dall'alto verso il basso e verrà eseguito solo il primo caso corrispondente.

```
Sub TestCase()
    Dim MyVar As String

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

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

La logica del blocco `Select Case` può essere invertita per supportare anche il test di diverse variabili, in questo tipo di scenario possiamo anche utilizzare operatori logici:

```
Dim x As Integer
Dim y As Integer

x = 2
y = 5

Select Case True
    Case x > 3
        MsgBox "x is greater than 3"
    Case y < 2
        MsgBox "y is less than 2"
    Case x = 1
        MsgBox "x is equal to 1"
    Case x = 2 Xor y = 3
        MsgBox "Go read about ""Xor"""
```

```
    Case Not y = 5
        MsgBox "y is not 5"
    Case x = 3 Or x = 10
        MsgBox "x = 3 or 10"
    Case y < 10 And x < 10
        MsgBox "x and y are less than 10"
    Case Else
        MsgBox "No match found"
End Select
```

Le dichiarazioni di caso possono anche utilizzare operatori aritmetici. Quando viene utilizzato un operatore aritmetico rispetto al valore `Select Case` , deve essere preceduto dalla parola chiave `Is` :

```
Dim x As Integer

x = 5

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

Per ogni ciclo

Il costrutto Ciclo `For Each` è ideale per iterare tutti gli elementi di una collezione.

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

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

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

End Sub
```

Utilizzare `For Each` quando si iterano raccolte di oggetti:

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

Evitare `For Each` quando si iterano gli array; un ciclo `For` offrirà prestazioni significativamente migliori con gli array. Viceversa, un ciclo `For Each` offrirà prestazioni migliori durante l'iterazione di una `Collection` .

Sintassi

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

La parola chiave `Next` può essere facoltativamente seguita dalla variabile iteratore; questo può aiutare a chiarire i cicli annidati, sebbene ci siano modi migliori per chiarire il codice annidato, come l'estrazione del ciclo interno nella propria procedura.

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

    Debug.Print book.FullName

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

Fai il ciclo

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

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


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

```
    Loop

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

Mentre loop

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

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

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

Per ciclo

Il ciclo `For` viene utilizzato per ripetere la sezione di codice racchiusa un determinato numero di volte. Il seguente semplice esempio illustra la sintassi di base:

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

Il codice sopra dichiara un intero `i` . Il ciclo `For` assegna ogni valore compreso tra 1 e 10 a `i` e quindi esegue `Debug.Print i` - ovvero il codice stampa i numeri da 1 a 10 nella finestra immediata. Si noti che la variabile di ciclo viene incrementata dall'istruzione `Next` , ovvero dopo l'esecuzione del codice allegato anziché prima dell'esecuzione.

Per impostazione predefinita, il contatore verrà incrementato di 1 ogni volta che viene eseguito il ciclo. Tuttavia, è possibile specificare una `Step` per modificare la quantità dell'incremento come valore letterale o di ritorno di una funzione. Se il valore iniziale, il valore finale o il valore `Step` è un numero in virgola mobile, verrà arrotondato al valore intero più vicino. `Step` può essere un valore positivo o negativo.

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

In generale un ciclo `For` dovrebbe essere utilizzato in situazioni in cui è noto prima che il ciclo inizi quante volte eseguire il codice allegato (altrimenti un ciclo `Do` o `While` potrebbe essere più appropriato). Questo perché la condizione di uscita è fissa dopo la prima entrata in loop, come

dimostra questo codice:

```
Private Iterations As Long              'Module scope

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

Un ciclo `For` può essere chiuso in anticipo con la dichiarazione `Exit For` :

```
Dim i As Integer

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

Leggi Strutture di controllo del flusso online: https://riptutorial.com/it/vba/topic/1873/strutture-di-controllo-del-flusso

Capitolo 45: Tipi di dati e limiti

Examples

Byte

```
Dim Value As Byte
```

Un byte è un tipo di dati a 8 bit senza segno. Può rappresentare numeri interi compresi tra 0 e 255 e il tentativo di memorizzare un valore al di fuori di tale intervallo provocherà l'errore di runtime 6: `Overflow` . Byte è l'unico tipo senza segno intrinseco disponibile in VBA.

La funzione di fusione per convertire in byte è `CByte()` . Per i cast dai tipi a virgola mobile, il risultato viene arrotondato al valore intero più vicino con 0,5 arrotondamento.

Array e stringhe di byte

Stringhe e array di byte possono essere sostituiti l'uno con l'altro tramite l'assegnazione semplice (non sono necessarie funzioni di conversione).

Per esempio:

```
Sub ByteToStringAndBack()

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

Dim byt() As Byte
byt = str

Debug.Print byt(0)  ' 72

Dim str2 As String
str2 = byt

Debug.Print str2    ' Hello, World!

End Sub
```

Per poter codificare i caratteri Unicode , ogni carattere nella stringa occupa due byte nell'array, con il byte meno significativo per primo. Per esempio:

```
Sub UnicodeExample()

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

Dim byt() As Byte
byt = str

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

```
End Sub
```

Numero intero

```
Dim Value As Integer
```

Un intero è un tipo di dati a 16 bit con segno. Può memorizzare numeri interi nell'intervallo compreso tra -32.768 e 32.767 e il tentativo di memorizzare un valore al di fuori di tale intervallo provocherà l'errore di runtime 6: Overflow.

I numeri interi sono memorizzati in memoria come valori little-endian con valori negativi rappresentati come complemento a due .

Si noti che in generale, è prassi meglio usare un lungo anziché un numero intero a meno che il tipo più piccolo è un membro di un tipo o è richiesto (o da una convenzione di chiamata API o altri motivi) ad essere di 2 byte. Nella maggior parte dei casi VBA considera gli interi come 32 bit internamente, quindi di solito non vi è alcun vantaggio nell'usare il tipo più piccolo. Inoltre, viene applicata una penalità legata alle prestazioni ogni volta che viene utilizzato un tipo Integer in quanto viene silenziosamente lanciato come Long.

La funzione di casting per convertire in un intero è `CInt()` . Per i cast dai tipi a virgola mobile, il risultato viene arrotondato al valore intero più vicino con 0,5 arrotondamento.

booleano

```
Dim Value As Boolean
```

Un booleano viene utilizzato per memorizzare valori che possono essere rappresentati come True o False. Internamente, il tipo di dati viene memorizzato come valore a 16 bit con 0 che rappresenta False e qualsiasi altro valore che rappresenta True.

Va notato che quando un booleano viene convertito in un tipo numerico, tutti i bit sono impostati su 1. Ciò determina una rappresentazione interna di -1 per i tipi firmati e il valore massimo per un tipo senza segno (Byte).

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

La funzione di casting per convertire in un booleano è `CBool()` . Anche se è rappresentato internamente come un numero a 16 bit, il casting su un booleano da valori esterni a tale intervallo è sicuro dall'overflow, anche se imposta tutti i 16 bit su 1:

```
Dim Example As Boolean
Example = CBool(2 ^ 17)
```

```
Debug.Print CInt(Example)   'Prints -1
Debug.Print CByte(Example)  'Prints 255
```

Lungo

```
Dim Value As Long
```

A Long è un tipo di dati a 32 bit con segno. Può memorizzare numeri interi compresi tra -2.147.483.648 e 2.147.483.647 e il tentativo di memorizzare un valore al di fuori di tale intervallo comporterà l'errore di runtime 6: Overflow.

I long vengono archiviati in memoria come valori little-endian con i negativi rappresentati come complemento a due .

Si noti che poiché Long corrisponde alla larghezza di un puntatore in un sistema operativo a 32 bit, i Long vengono comunemente usati per archiviare e passare i puntatori alle e dalle funzioni API.

La funzione di fusione per convertire in Long è `CLng()` . Per i cast dai tipi a virgola mobile, il risultato viene arrotondato al valore intero più vicino con 0,5 arrotondamento.

singolo

```
Dim Value As Single
```

Un singolo è un tipo di dati a virgola mobile a 32 bit con segno. Viene memorizzato internamente utilizzando un layout di memoria IEEE 754 little-endian . Pertanto, non esiste un intervallo di valori fisso che può essere rappresentato dal tipo di dati: ciò che è limitato è la precisione del valore memorizzato. Un singolo può memorizzare valori ***interi di*** valori nell'intervallo da -16.777.216 a 16.777.216 senza una perdita di precisione. La precisione dei numeri in virgola mobile dipende dall'esponente.

Una singola trabocchera se viene assegnato un valore superiore a circa 2^{128} . Non trabocchera con esponenti negativi, anche se la precisione utilizzabile sarà discutibile prima che venga raggiunto il limite superiore.

Come con tutti i numeri in virgola mobile, è necessario prestare attenzione quando si effettuano confronti di uguaglianza. La migliore pratica consiste nell'includere un valore delta appropriato alla precisione richiesta.

La funzione di fusione per convertire in un singolo è `CSng()` .

Doppio

```
Dim Value As Double
```

Un Double è un tipo di dati a virgola mobile a 64 bit con segno. Come il Single , è memorizzato

internamente utilizzando un layout di memoria IEEE 754 little endian e devono essere prese le stesse precauzioni per quanto riguarda la precisione. Un Double può memorizzare valori ***interi*** nell'intervallo da -9,007,199,254,740,992 a 9,007,199,254,740,992 senza perdita di precisione. La precisione dei numeri in virgola mobile dipende dall'esponente.

Una doppia sarà overflow se assegnata a un valore superiore a circa 2^{1024} . Non trabocchera con esponenti negativi, anche se la precisione utilizzabile sarà discutibile prima che venga raggiunto il limite superiore.

La funzione di casting per convertire in Double è `CDbl()` .

Moneta

```
Dim Value As Currency
```

Una valuta è un tipo di dati in virgola mobile a 64 bit con segno simile a un doppio , ma ridimensionato di 10.000 per dare maggiore precisione alle 4 cifre a destra del punto decimale. Una variabile di valuta può memorizzare valori da -922,337,203,685,477,5808 a 922,337,203,685,477,5807, fornendo la massima capacità di qualsiasi tipo intrinseco in un'applicazione a 32 bit. Come suggerisce il nome del tipo di dati, è consigliabile utilizzare questo tipo di dati quando si rappresentano calcoli monetari poiché il ridimensionamento aiuta a evitare errori di arrotondamento.

La funzione di casting per convertire in una valuta è `CCur()` .

Data

```
Dim Value As Date
```

Un tipo Date viene rappresentato internamente come un tipo di dati con segno a 64 bit in virgola mobile con il valore alla sinistra del decimale che rappresenta il numero di giorni dalla data epoca del 30 dicembre 1899 (anche se vedi nota sotto). Il valore alla destra del decimale rappresenta il tempo come un giorno frazionario. Pertanto, una data intera avrebbe un componente orario di 12:00 AM e x.5 avrebbe una componente temporale di 12:00:00 PM.

I valori validi per le date sono compresi tra 1 ° gennaio 100 e il 31 dicembre 9999. Dal momento che un doppio ha una gamma più ampia, è possibile traboccare una data assegnando valori al di fuori di tale intervallo.

Come tale, può essere usato in modo intercambiabile con i calcoli Double for Date:

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

La funzione di fusione per convertire in una data è `CDate()` , che accetta qualsiasi

rappresentazione di data / ora stringa di tipo numerico. È importante notare che le rappresentazioni di stringa delle date verranno convertite in base all'impostazione locale corrente in uso, quindi i cast diretti dovrebbero essere evitati se il codice è pensato per essere portabile.

Stringa

Una stringa rappresenta una sequenza di caratteri ed è disponibile in due versioni:

Lunghezza variabile

```
Dim Value As String
```

Una stringa di lunghezza variabile consente l'aggiunta e il troncamento e viene archiviata in memoria come un BSTR COM. Si tratta di un intero senza segno a 4 byte che memorizza la lunghezza della stringa in byte seguita dai dati della stringa stessa come caratteri di larghezza (2 byte per carattere) e terminata con 2 byte nulli. Pertanto, la lunghezza massima della stringa che può essere gestita da VBA è di 2.147.483.647 caratteri.

Il puntatore interno alla struttura (recuperabile dalla funzione `StrPtr()`) punta alla posizione di memoria dei *dati* , non al prefisso di lunghezza. Ciò significa che una stringa VBA può essere passata direttamente alle funzioni API che richiedono un puntatore a un array di caratteri.

Poiché la lunghezza può variare, VBA rialloca la memoria per una stringa *ogni volta che viene assegnata la variabile* , il che può imporre sanzioni per le prestazioni per le procedure che le modificano ripetutamente.

Lunghezza fissa

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

Le stringhe a lunghezza fissa sono assegnate a 2 byte per ogni carattere e vengono memorizzate come semplice array di byte. Una volta assegnato, la lunghezza della stringa è immutabile. Essi ***non*** sono zero finale in memoria, quindi una stringa che riempie la memoria allocata con caratteri non nulli è adatto per il passaggio di funzioni API che prevedono una terminazione null stringa.

Le stringhe a lunghezza fissa supportano una limitazione di indice legacy di 16 bit, quindi possono avere una lunghezza massima di 65.535 caratteri. Il tentativo di assegnare un valore più lungo dello spazio di memoria disponibile non comporterà un errore di runtime, ma il valore risultante verrà semplicemente troncato:

```
Dim Foobar As String * 5
Foobar = "Foo" & "bar"
Debug.Print Foobar          'Prints "Fooba"
```

La funzione di fusione per convertire in una stringa di entrambi i tipi è `CStr()` .

Lungo lungo

```
Dim Value As LongLong
```

LongLong è un tipo di dati a 64 bit con segno ed è disponibile solo in applicazioni a 64 bit. **Non** è disponibile in applicazioni a 32 bit in esecuzione su sistemi operativi a 64 bit. Può memorizzare valori interi nell'intervallo compreso tra -9,223,372,036,854,775,808 e 9,223,372,036,854,775,807 e il tentativo di memorizzare un valore al di fuori di tale intervallo provocherà l'errore di runtime 6: Overflow.

I LongLongs sono archiviati in memoria come valori little-endian con i negativi rappresentati come complemento a due .

Il tipo di dati LongLong è stato introdotto come parte del supporto del sistema operativo a 64 bit di VBA. Nelle applicazioni a 64 bit, questo valore può essere utilizzato per archiviare e passare i puntatori a API a 64 bit.

La funzione di fusione per convertire in LongLong è `CLngLng()` . Per i cast dai tipi a virgola mobile, il risultato viene arrotondato al valore intero più vicino con 0,5 arrotondamento.

Variante

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

Un Variant è un tipo di dati COM che viene utilizzato per l'archiviazione e lo scambio di valori di tipi arbitrari e qualsiasi altro tipo in VBA può essere assegnato a un Variant. Le variabili dichiarate senza un tipo esplicito specificato da `As [Type]` impostate su Variant.

Le varianti sono archiviate in memoria come una struttura VARIANT che consiste in un descrittore di tipo byte (VARTYPE) seguito da 6 byte riservati quindi da un'area di dati da 8 byte. Per i tipi numerici (inclusi Date e Boolean), il valore sottostante viene memorizzato nella Variant stessa. Per tutti gli altri tipi, l'area dati contiene un puntatore al valore sottostante.

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

Il tipo sottostante di Variant può essere determinato con la funzione `VarType()` che restituisce il valore numerico memorizzato nel descrittore di tipo o la funzione `TypeName()` che restituisce la rappresentazione di stringa:

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

Poiché le varianti possono memorizzare valori di qualsiasi tipo, gli assegnamenti dai valori letterali senza i suggerimenti di tipo verranno convertiti implicitamente in una variante del tipo appropriato in base alla tabella seguente. I valori letterali con suggerimenti tipo verranno convertiti in una variante del tipo suggerito.

Valore	Tipo risultante
Valori di stringa	Stringa
Numeri senza virgola mobile nell'intervallo dei numeri interi	Numero intero
Numeri a virgola mobile in lungo raggio	Lungo
Numeri senza virgola mobile fuori da Lungo raggio	Doppio
Tutti i numeri in virgola mobile	Doppio

Nota: a meno che non vi sia un motivo specifico per utilizzare una variante (ovvero un iteratore in un ciclo For Each o un requisito API), il tipo dovrebbe essere generalmente evitato per le attività di routine per i seguenti motivi:

- Non sono sicuri, aumentando la possibilità di errori di runtime. Ad esempio, un valore Variant che contiene un valore intero si modifica automaticamente in un valore Long anziché in overflow.
- Introducono l'overhead di elaborazione richiedendo almeno un ulteriore dereferenziamento del puntatore.
- Il requisito di memoria per un valore Variant è sempre superiore di ***almeno*** 8 byte rispetto a quello necessario per memorizzare il tipo sottostante.

La funzione di fusione per convertire in Variant è `CVar()` .

LongPtr

```
Dim Value As LongPtr
```

Il LongPtr è stato introdotto in VBA per supportare piattaforme a 64 bit. Su un sistema a 32 bit, viene trattato come un sistema Long e su sistemi a 64 bit viene trattato come LongLong .

Il suo scopo principale è fornire un modo portatile per archiviare e passare i puntatori su entrambe le architetture (vedere Modifica del comportamento del codice in fase di compilazione .

Sebbene sia utilizzato dal sistema operativo come indirizzo di memoria quando viene utilizzato nelle chiamate API, è necessario notare che VBA lo considera come un tipo firmato (e quindi soggetto a overflow con segno non firmato). Per questo motivo, qualsiasi aritmetica del puntatore eseguita usando LongPtrs non dovrebbe usare `>` o `<` confronti. Questa "stranezza" rende anche possibile che l'aggiunta di offset semplici che puntano a indirizzi validi in memoria possa causare errori di overflow, quindi occorre prestare attenzione quando si lavora con i puntatori in VBA.

La funzione di fusione per convertire in LongPtr è `CLngPtr()` . Per i cast dai tipi a virgola mobile, il risultato viene arrotondato al valore intero più vicino con 0,5 arrotondamenti (sebbene poiché di solito è un indirizzo di memoria, usarlo come target di assegnazione per un calcolo a virgola mobile è pericoloso al meglio).

Decimale

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

'Set Value to the smallest possible Decimal value
Value = CDec("0.00000000000000000000000001")
```

Il tipo di dati `Decimal` è disponibile *solo* come sottotipo di `Variant` , pertanto è necessario dichiarare qualsiasi variabile che deve contenere un `Decimal` come `Variant` e *quindi* assegnare un valore `Decimal` utilizzando la funzione `CDec` . La parola chiave `Decimal` è una parola riservata (che suggerisce che VBA alla fine avrebbe aggiunto il supporto di prima classe per il tipo), quindi `Decimal` non può essere utilizzato come variabile o nome di procedura.

Il tipo `Decimal` richiede 14 byte di memoria (oltre ai byte richiesti dalla variante padre) e può memorizzare numeri con un massimo di 28 cifre decimali. Per i numeri senza cifre decimali, l'intervallo di valori consentiti è compreso tra -79,228,162,514,264,337,593,543,950,335 e +79,228,162,514,264,337,593,543,950,335 inclusi. Per i numeri con un massimo di 28 cifre decimali, l'intervallo di valori consentiti è compreso tra -7.9228162514264337593543950335 e +7.9228162514264337593543950335.

Leggi Tipi di dati e limiti online: https://riptutorial.com/it/vba/topic/3418/tipi-di-dati-e-limiti

Capitolo 46: VBA orientato agli oggetti

Examples

Astrazione

I livelli di astrazione aiutano a determinare quando suddividere le cose.

L'astrazione si ottiene implementando funzionalità con codice sempre più dettagliato. Il punto di ingresso di una macro dovrebbe essere una piccola procedura con un *alto livello di astrazione* che renda facile capire a colpo d'occhio cosa sta succedendo:

```
Public Sub DoSomething()
    With New SomeForm
        Set .Model = CreateViewModel
        .Show vbModal
        If .IsCancelled Then Exit Sub
        ProcessUserData .Model
    End With
End Sub
```

La procedura `DoSomething` ha un alto *livello di astrazione* : possiamo dire che sta visualizzando un modulo e creando un modello e passando quell'oggetto ad una procedura `ProcessUserData` che sa cosa fare con esso - il modo in cui il modello viene creato è il lavoro di un'altra procedura:

```
Private Function CreateViewModel() As ISomeModel
    Dim result As ISomeModel
    Set result = SomeModel.Create(Now, Environ$("UserName"))
    result.AvailableItems = GetAvailableItems
    Set CreateViewModel = result
End Function
```

La funzione `CreateViewModel` è responsabile solo della creazione di alcune istanze di `ISomeModel` . Parte di questa responsabilità consiste nell'acquisire una serie di *elementi disponibili* : il modo in cui questi elementi vengono acquisiti è un dettaglio di implementazione che è astratto rispetto alla procedura `GetAvailableItems` :

```
Private Function GetAvailableItems() As Variant
    GetAvailableItems = DataSheet.Names("AvailableItems").RefersToRange
End Function
```

Qui la procedura sta leggendo i valori disponibili da un intervallo denominato su un foglio di lavoro `DataSheet` . Potrebbe benissimo leggerli da un database, oppure i valori potrebbero essere hardcoded: è un *dettaglio di implementazione* che non preoccupa nessuno dei più alti livelli di astrazione.

incapsulamento

L'incapsulamento nasconde i dettagli di implementazione dal codice client.

L'esempio Handling QueryClose dimostra l'incapsulamento: il modulo ha un controllo casella di controllo, ma il suo codice client non funziona direttamente con esso: la casella di spunta è un *dettaglio dell'implementazione* , ciò che il codice client deve sapere è se l'impostazione è abilitata o meno.

Quando il valore della casella di controllo cambia, il gestore assegna un membro del campo privato:

```
Private Type TView
    IsCancelled As Boolean
    SomeOtherSetting As Boolean
    'other properties skipped for brievety
End Type
Private this As TView

'...

Private Sub SomeOtherSettingInput_Change()
    this.SomeOtherSetting = CBool(SomeOtherSettingInput.Value)
End Sub
```

E quando il codice client vuole leggere quel valore, non ha bisogno di preoccuparsi di una checkbox - invece usa semplicemente la proprietà `SomeOtherSetting` :

```
Public Property Get SomeOtherSetting() As Boolean
    SomeOtherSetting = this.SomeOtherSetting
End Property
```

La proprietà `SomeOtherSetting` *incapsula* lo stato della casella di controllo; il codice cliente non ha bisogno di sapere che c'è una casella di controllo, solo che c'è un'impostazione con un valore booleano. *Incapsulando* il valore `Boolean` , abbiamo aggiunto un *livello di astrazione* attorno alla casella di controllo.

Utilizzare le interfacce per rafforzare l'immutabilità

Facciamo un ulteriore passo avanti *incapsulando* il *modello* del modulo in un modulo di classe dedicato. Ma se abbiamo creato una `Public Property` per `UserName` e `Timestamp` , dovremmo esporre `Property Let` accessors, rendendo le proprietà mutabili, e non vogliamo che il codice client abbia la possibilità di modificare questi valori dopo che sono stati impostati.

La funzione `CreateViewModel` nell'esempio **Abstraction** restituisce una classe `ISomeModel` : questa è la nostra *interfaccia* e assomiglia a qualcosa del genere:

```
Option Explicit

Public Property Get Timestamp() As Date
End Property
```

```
Public Property Get UserName() As String
End Property

Public Property Get AvailableItems() As Variant
End Property

Public Property Let AvailableItems(ByRef value As Variant)
End Property

Public Property Get SomeSetting() As String
End Property

Public Property Let SomeSetting(ByVal value As String)
End Property

Public Property Get SomeOtherSetting() As Boolean
End Property

Public Property Let SomeOtherSetting(ByVal value As Boolean)
End Property
```

Nota Le proprietà `Timestamp` e `UserName` espongono solo una `Property Get` accesso. Ora la classe `SomeModel` può implementare quell'interfaccia:

```
Option Explicit
Implements ISomeModel

Private Type TModel
    Timestamp As Date
    UserName As String
    SomeSetting As String
    SomeOtherSetting As Boolean
    AvailableItems As Variant
End Type
Private this As TModel

Private Property Get ISomeModel_Timestamp() As Date
    ISomeModel_Timestamp = this.Timestamp
End Property

Private Property Get ISomeModel_UserName() As String
    ISomeModel_UserName = this.UserName
End Property

Private Property Get ISomeModel_AvailableItems() As Variant
    ISomeModel_AvailableItems = this.AvailableItems
End Property

Private Property Let ISomeModel_AvailableItems(ByRef value As Variant)
    this.AvailableItems = value
End Property

Private Property Get ISomeModel_SomeSetting() As String
    ISomeModel_SomeSetting = this.SomeSetting
End Property

Private Property Let ISomeModel_SomeSetting(ByVal value As String)
    this.SomeSetting = value
End Property
```

```
Private Property Get ISomeModel_SomeOtherSetting() As Boolean
    ISomeModel_SomeOtherSetting = this.SomeOtherSetting
End Property

Private Property Let ISomeModel_SomeOtherSetting(ByVal value As Boolean)
    this.SomeOtherSetting = value
End Property

Public Property Get Timestamp() As Date
    Timestamp = this.Timestamp
End Property

Public Property Let Timestamp(ByVal value As Date)
    this.Timestamp = value
End Property

Public Property Get UserName() As String
    UserName = this.UserName
End Property

Public Property Let UserName(ByVal value As String)
    this.UserName = value
End Property

Public Property Get AvailableItems() As Variant
    AvailableItems = this.AvailableItems
End Property

Public Property Let AvailableItems(ByRef value As Variant)
    this.AvailableItems = value
End Property

Public Property Get SomeSetting() As String
    SomeSetting = this.SomeSetting
End Property

Public Property Let SomeSetting(ByVal value As String)
    this.SomeSetting = value
End Property

Public Property Get SomeOtherSetting() As Boolean
    SomeOtherSetting = this.SomeOtherSetting
End Property

Public Property Let SomeOtherSetting(ByVal value As Boolean)
    this.SomeOtherSetting = value
End Property
```

I membri dell'interfaccia sono tutti `Private` e tutti i membri dell'interfaccia devono essere implementati affinché il codice possa essere compilato. I membri `Public` non fanno parte dell'interfaccia e pertanto non sono esposti al codice scritto contro l'interfaccia `ISomeModel` .

Utilizzo di un metodo di fabbrica per simulare un costruttore

Utilizzando un attributo VB_PredeclaredId , possiamo rendere la classe `SomeModel` *un'istanza predefinita* e scrivere una funzione che funzioni come un membro a livello di testo (`Shared` in VB.NET, `static` in C #) che il codice client può chiamare senza dover prima creare un esempio,

come abbiamo fatto qui:

```
Private Function CreateViewModel() As ISomeModel
    Dim result As ISomeModel
    Set result = SomeModel.Create(Now, Environ$("UserName"))
    result.AvailableItems = GetAvailableItems
    Set CreateViewModel = result
End Function
```

Questo *metodo factory* assegna i valori di proprietà che sono di sola lettura quando si accede dall'interfaccia `ISomeModel` , qui `Timestamp` e `UserName` :

```
Public Function Create(ByVal pTimeStamp As Date, ByVal pUserName As String) As ISomeModel
    With New SomeModel
        .Timestamp = pTimeStamp
        .UserName = pUserName
        Set Create = .Self
    End With
End Function

Public Property Get Self() As ISomeModel
    Set Self = Me
End Property
```

E ora possiamo codificare l'interfaccia `ISomeModel` , che espone `Timestamp` e `UserName` come proprietà di sola lettura che non possono mai essere riassegnate (purché il codice sia scritto sull'interfaccia).

Polimorfismo

Il polimorfismo è la capacità di presentare la stessa interfaccia per diverse implementazioni sottostanti.

La possibilità di implementare interfacce consente di disaccoppiare completamente la logica dell'applicazione dall'interfaccia utente o dal database o da questo o quel foglio di lavoro.

Supponiamo tu abbia un'interfaccia `ISomeView` che il modulo stesso implementa:

```
Option Explicit

Public Property Get IsCancelled() As Boolean
End Property

Public Property Get Model() As ISomeModel
End Property

Public Property Set Model(ByVal value As ISomeModel)
End Property

Public Sub Show()
End Sub
```

Il code-behind del modulo potrebbe assomigliare a questo:

```
Option Explicit
Implements ISomeView

Private Type TView
    IsCancelled As Boolean
    Model As ISomeModel
End Type
Private this As TView

Private Property Get ISomeView_IsCancelled() As Boolean
    ISomeView_IsCancelled = this.IsCancelled
End Property

Private Property Get ISomeView_Model() As ISomeModel
    Set ISomeView_Model = this.Model
End Property

Private Property Set ISomeView_Model(ByVal value As ISomeModel)
    Set this.Model = value
End Property

Private Sub ISomeView_Show()
    Me.Show vbModal
End Sub

Private Sub SomeOtherSettingInput_Change()
    this.Model.SomeOtherSetting = CBool(SomeOtherSettingInput.Value)
End Sub

'...other event handlers...

Private Sub OkButton_Click()
    Me.Hide
End Sub

Private Sub CancelButton_Click()
    this.IsCancelled = True
    Me.Hide
End Sub

Private Sub UserForm_QueryClose(Cancel As Integer, CloseMode As Integer)
    If CloseMode = VbQueryClose.vbFormControlMenu Then
        Cancel = True
        this.IsCancelled = True
        Me.Hide
    End If
End Sub
```

Ma poi, nulla vieta di creare un altro modulo di classe che implementa l'interfaccia di `ISomeView` *senza essere un modulo utente* - questa potrebbe essere una classe `SomeViewMock` :

```
Option Explicit
Implements ISomeView

Private Type TView
    IsCancelled As Boolean
    Model As ISomeModel
End Type
Private this As TView
```

```
Public Property Get IsCancelled() As Boolean
    IsCancelled = this.IsCancelled
End Property

Public Property Let IsCancelled(ByVal value As Boolean)
    this.IsCancelled = value
End Property

Private Property Get ISomeView_IsCancelled() As Boolean
    ISomeView_IsCancelled = this.IsCancelled
End Property

Private Property Get ISomeView_Model() As ISomeModel
    Set ISomeView_Model = this.Model
End Property

Private Property Set ISomeView_Model(ByVal value As ISomeModel)
    Set this.Model = value
End Property

Private Sub ISomeView_Show()
    'do nothing
End Sub
```

E ora possiamo cambiare il codice che funziona con un `UserForm` e farlo funzionare sull'interfaccia di `ISomeView` , ad esempio assegnandogli il modulo come parametro invece di istanziarlo:

```
Public Sub DoSomething(ByVal view As ISomeView)
    With view
        Set .Model = CreateViewModel
        .Show
        If .IsCancelled Then Exit Sub
        ProcessUserData .Model
    End With
End Sub
```

Poiché il metodo `DoSomething` dipende da un'interfaccia (ad esempio *un'astrazione*) e non da una *classe concreta* (ad es. Un `UserForm` specifico), possiamo scrivere un test unitario che garantisce che `ProcessUserData` non venga eseguito quando `view.IsCancelled` è `True` , rendendo il nostro test crea un'istanza `SomeViewMock` , impostando la proprietà `IsCancelled` su `True` e passando a `DoSomething` .

Il codice verificabile dipende dalle astrazioni

È possibile eseguire test di unità in VBA, ci sono componenti aggiuntivi che possono essere integrati nell'IDE. Ma quando il codice è *strettamente associato* a un foglio di lavoro, a un database, a un modulo o al file system, il test dell'unità inizia a richiedere un foglio di lavoro, un database, un modulo o un file system effettivo e queste *dipendenze* sono un nuovo errore fuori controllo indica che il codice verificabile deve essere isolato, in modo che i test unitari *non* richiedano un foglio di lavoro, un database, un modulo o un file system effettivi.

Scrivendo il codice contro le interfacce, in un modo che consente al codice di test di *iniettare* implementazioni di stub / mock (come nell'esempio `SomeViewMock` sopra), puoi scrivere test in un

"ambiente controllato" e simulare cosa succede quando ognuno dei 42 possibili permutazioni delle interazioni dell'utente sui dati del modulo, anche senza visualizzare una volta un modulo e facendo clic manualmente su un controllo modulo.

Leggi VBA orientato agli oggetti online: https://riptutorial.com/it/vba/topic/5357/vba-orientato-agli-oggetti

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con VBA	0m3r, Andre Terra, Benno Grimm, Bookeater, Comintern, Community, Derpcode, Kaz, lfrandom, litelite, Maarten van Stam, Macro Man, Máté Juhász, Nick Dewitt, PankajKushwaha, RubberDuck, Stefan Pinnow
2	Array	Comintern, Dave, Hubisan, jamheadart, Josan Iracheta, Maarten van Stam, Mark.R, Mat's Mug, Miguel_Ryu, Tazaf
3	Assegnazione di stringhe con caratteri ripetuti	ThunderFrame
4	attributi	hymced, Mat's Mug, RamenChef, RubberDuck
5	Automazione o utilizzo di altre applicazioni Librerie	Branislav Kollár
6	Chiamate API	paul bica
7	collezioni	Comintern
8	Commenti	Comintern, Hosch250, Johnny C, litelite, Macro Man, Nijin22, Shawn V. Wilson, ThunderFrame
9	Compilazione condizionale	Macro Man, Mat's Mug, RubberDuck, Steve Rindsberg
10	Concatenazione di stringhe	ThunderFrame
11	Convenzioni di denominazione	FreeMan, Kaz, Mat's Mug, Victor Moraes
12	Convertire altri tipi di stringhe	ThunderFrame
13	Copia, restituisce e passa array	Mark.R
14	Creare una procedura	Comintern, LiamH, Mat's Mug, Sivaprasath Vadivel, Tomas Zubiri
15	CreateObject vs. GetObject	Branislav Kollár, Dave, Tim

16	Creazione di una classe personalizzata	Branislav Kollár, Macro Man, Mat's Mug, Neil Mussett, ThunderFrame
17	Data Manipolazione del tempo	Comintern, FreeMan, Thomas G
18	Dichiarazione delle variabili	Comintern, dadde, Dave, Franck Dernoncourt, Jeeped, Kaz, Ifrandom, litelite, Macro Man, Mark.R, Mat's Mug, Neil Mussett, RubberDuck, Shawn V. Wilson, SWa, Thierry Dalon, ThunderFrame, Tom, Victor Moraes, Zaider
19	Dichiarazione e assegnazione di stringhe	Comintern, ThunderFrame
20	Errori in fase di esecuzione VBA	Branislav Kollár, Macro Man, Mat's Mug
21	eventi	Mat's Mug
22	Gestione degli errori	Comintern, Logan Reed, Mat's Mug
23	interfacce	Neil Mussett
24	Lavorare con ADO	Comintern, SandPiper, Tazaf
25	Lavorare con file e directory senza utilizzare FileSystemObject	Comintern, Macro Man, SandPiper
26	Lettura di file da 2 GB + in binario in VBA e File Hash	PatricK
27	Manipolazione delle stringhe usata frequentemente	pashute
28	Misurare la lunghezza delle stringhe	Steve Rindsberg, ThunderFrame
29	Moduli utente	Mat's Mug
30	operatori	Comintern, Macro Man
31	Ordinamento	Neil Mussett
32	Parola chiave dell'opzione VBA	Jeeped, Maarten van Stam, Macro Man, Mat's Mug, RamenChef, RubberDuck, Stefan Pinnow, Thomas G, ThunderFrame
33	Passando argomenti	Branislav Kollár, Comintern, Mat's Mug, R3uK,

	ByRef o ByVal	RamenChef, ZygD
34	Personaggi non latini	Neil Mussett
35	Procedura Chiamate	Macro Man, Mat's Mug, Neil Mussett, Sam Johnson
36	Ricerca all'interno di stringhe per la presenza di sottostringhe	ThunderFrame
37	ricorsione	Mat's Mug, ThunderFrame
38	Scripting. Oggetto letterario	Comintern, Jeeped, Kyle, RamenChef, Tim, Wolf, Zev Spitz
39	Scripting.FileSystemObject	Comintern, Dave, Macro Man, Mikegrann, RubberDuck, Siva, Steve Rindsberg, ThunderFrame
40	Sicurezza macro e firma di progetti / moduli VBA	0m3r
41	sottostringhe	Mat's Mug, ThunderFrame
42	String letterali: caratteri di escape, non stampabili e continuazioni di riga	Comintern, ThunderFrame
43	Strutture dati	Blackhawk
44	Strutture di controllo del flusso	Benno Grimm, Comintern, Kelly Tessena Keck, Leviathan, litelite, Macro Man, Martin, Mat's Mug, Roland, Siva, ThunderFrame
45	Tipi di dati e limiti	Comintern, FreeMan, Neil Mussett, StackzOfZtuff, Stephen Leppik, ThunderFrame
46	VBA orientato agli oggetti	IvenBach, Mat's Mug