EBook Gratis

APRENDIZAJE VBA

Free unaffiliated eBook created from
Stack Overflow contributors.

#vba

Tabla de contenido

Acerca de 1

Capítulo 1: Empezando con VBA 2

Observaciones 2

Versiones 2

Examples 2

Accediendo al Editor de Visual Basic en Microsoft Office 2

Primer módulo y Hello World 5

Depuración 6

Ejecutar código paso a paso 6

Ventana de relojes 6

Ventana inmediata 6

Depuración de mejores prácticas 7

Capítulo 2: Arrays 8

Examples 8

Declarando un Array en VBA 8

Elementos de acceso 8

Indexación de matrices 8

Índice especifico 8

Declaración dinámica 8

Uso de Split para crear una matriz a partir de una cadena 9

Iterando elementos de una matriz 10

Para ... siguiente 10

Para cada ... siguiente 11

Arreglos dinámicos (cambio de tamaño de matriz y manejo dinámico) 12

Arrays dinámicos 12

Agregando valores dinámicamente 12

Eliminar valores dinámicamente 13

Restablecimiento de una matriz y reutilización dinámica 13

Arreglos irregulares (Arrays of Arrays) 13

Matrices dentadas NO matrices multidimensionales 13

Creación de una matriz irregular ... 14
Creación y lectura dinámica de matrices dentadas ... 14
Arreglos Multidimensionales ... 16
Arreglos Multidimensionales ... 16
Array de dos dimensiones ... 17
Array de tres dimensiones ... 19
Capítulo 3: Asignando cadenas con caracteres repetidos ... 23
Observaciones ... 23
Examples ... 23
Utilice la función de cadena para asignar una cadena con n caracteres repetidos ... 23
Usa las funciones de Cadena y Espacio para asignar una cadena de n caracteres ... 23
Capítulo 4: Atributos ... 24
Sintaxis ... 24
Examples ... 24
VB_Name ... 24
VB_GlobalNameSpace ... 24
VB_Createable ... 24
VB_PredeclaredId ... 25
Declaración ... 25
Llamada ... 25
VB_Expuesto ... 25
VB_Descripción ... 26
VB_ [Var] UserMemId ... 26
Especificando el miembro predeterminado de una clase ... 26
Hacer una clase iteratable con una construcción de bucle For Each ... 27
Capítulo 5: Automatización o uso de otras bibliotecas de aplicaciones. ... 29
Introducción ... 29
Sintaxis ... 29
Observaciones ... 29
Examples ... 29
Expresiones regulares de VBScript ... 30

Código ... 30
Objeto del sistema de archivos de secuencias de comandos ... 31
Diccionario de secuencias de comandos objeto ... 31
Objeto de Internet Explorer ... 32
Miembros básicos de Internet Explorer Objec ... 32
Raspado web ... 33
Hacer clic ... 34
Biblioteca de objetos HTML de Microsoft o mejor amigo de IE ... 35
Principales problemas de IE ... 35
Capítulo 6: Buscando dentro de las cadenas la presencia de subcadenas. ... 36
Observaciones ... 36
Examples ... 36
Usa InStr para determinar si una cadena contiene una subcadena ... 36
Utilice InStr para encontrar la posición de la primera instancia de una subcadena ... 36
Utilice InStrRev para encontrar la posición de la última instancia de una subcadena ... 36
Capítulo 7: Clasificación ... 38
Introducción ... 38
Examples ... 38
Implementación de algoritmos - Ordenación rápida en una matriz unidimensional ... 38
Uso de la biblioteca de Excel para ordenar una matriz unidimensional ... 39
Capítulo 8: Colecciones ... 41
Observaciones ... 41
Comparación de características con matrices y diccionarios ... 41
Examples ... 42
Agregar elementos a una colección ... 42
Eliminar elementos de una colección ... 43
Obtener el recuento de artículos de una colección ... 44
Recuperar elementos de una colección ... 44
Determinar si una clave o elemento existe en una colección ... 46
Llaves ... 46
Artículos ... 47
Borrar todos los artículos de una colección ... 47

Capítulo 9: Comentarios 49
Observaciones 49
Examples 49
Apóstrofe Comentarios 49
REM comentarios 50
Capítulo 10: Compilación condicional 51
Examples 51
Cambiar el comportamiento del código en tiempo de compilación 51
Uso de Declare Imports que funciona en todas las versiones de Office 52
Capítulo 11: Convenciones de nombres 54
Examples 54
Nombres de variables 54
Notación húngara 55
Nombres de procedimientos 57
Capítulo 12: Convertir otros tipos a cadenas 59
Observaciones 59
Examples 59
Usa CStr para convertir un tipo numérico a una cadena 59
Utilice Formato para convertir y formatear un tipo numérico como una cadena 59
Utilice StrConv para convertir una matriz de bytes de caracteres de un solo byte en una ca 59
Convertir implícitamente una matriz de bytes de caracteres de múltiples bytes en una caden 59
Capítulo 13: Copiando, devolviendo y pasando matrices. 61
Examples 61
Copiando Arrays 61
Copiar matrices de objetos 62
Variantes que contienen una matriz 62
Devolviendo Arrays desde Funciones 62
Salida de una matriz a través de un argumento de salida 63
Salida a una matriz fija 63
Salida de una matriz de un método de clase 64
Pasando matrices a procedimientos 64

Capítulo 14: Creación de una clase personalizada 66
Observaciones 66
Examples 66
Agregar una propiedad a una clase 66
Agregando Funcionalidad a una Clase 67
Ámbito del módulo de clase, creación de instancias y reutilización. 68
Capítulo 15: Creando un procedimiento 70
Examples 70
Introducción a los procedimientos. 70
Devolviendo un valor 70
Funcionar con ejemplos 71
Capítulo 16: CreateObject vs. GetObject 73
Observaciones 73
Examples 73
Demostrando GetObject y CreateObject 73
Capítulo 17: Cuerdas de concatenacion 75
Observaciones 75
Examples 75
Concatenar cadenas utilizando el operador & 75
Concatenar una matriz de cadenas mediante la función Unir 75
Capítulo 18: Declarando variables 76
Examples 76
Declaración implícita y explícita 76
Variables 76
Alcance 76
Variables locales 77
Variables estáticas 77
Campos 79
Campos de instancia 79
Campos de encapsulacion 80
Constantes (const) 80
Modificadores de acceso 81

Opción Módulo Privado ... 82
Tipo de sugerencias ... 82
Funciones incorporadas de retorno de cadena ... 83
Declarar cadenas de longitud fija ... 84
Cuándo usar una variable estática ... 84
Capítulo 19: Declarar y asignar cadenas ... 87
Observaciones ... 87
Examples ... 87
Declara una cadena constante ... 87
Declara una variable de cadena de ancho variable ... 87
Declara y asigna una cadena de ancho fijo ... 87
Declara y asigna una cadena de cadenas ... 87
Asigna caracteres específicos dentro de una cadena usando la instrucción Mid ... 88
Asignación ay desde una matriz de bytes ... 88
Capítulo 20: Errores en tiempo de ejecución de VBA ... 90
Introducción ... 90
Examples ... 90
Error en tiempo de ejecución '3': Devolución sin GoSub ... 90
Código incorrecto ... 90
¿Por qué no funciona esto? ... 90
Código correcto ... 90
¿Por qué funciona esto? ... 90
Otras notas ... 91
Error en tiempo de ejecución '6': Desbordamiento ... 91
código incorrecto ... 91
¿Por qué no funciona esto? ... 91
Código correcto ... 91
¿Por qué funciona esto? ... 91
Otras notas ... 91
Error en tiempo de ejecución '9': Subíndice fuera de rango ... 91
código incorrecto ... 92
¿Por qué no funciona esto? ... 92

Código correcto ... 92
¿Por qué funciona esto? ... 92
Otras notas ... 92
Error en tiempo de ejecución '13': No coincide el tipo ... 92
código incorrecto ... 93
¿Por qué no funciona esto? ... 93
Código correcto ... 93
¿Por qué funciona esto? ... 93
Error en tiempo de ejecución '91': variable de objeto o variable de bloque no establecida ... 93
código incorrecto ... 93
¿Por qué no funciona esto? ... 93
Código correcto ... 94
¿Por qué funciona esto? ... 94
Otras notas ... 94
Error en tiempo de ejecución '20': Reanudar sin error ... 94
código incorrecto ... 94
¿Por qué no funciona esto? ... 95
Código correcto ... 95
¿Por qué funciona esto? ... 95
Otras notas ... 95
Capítulo 21: Estructuras de control de flujo ... 96
Examples ... 96
Seleccione el caso ... 96
Para cada bucle ... 97
Sintaxis ... 98
Hacer bucle ... 98
Mientras bucle ... 99
En bucle ... 99
Capítulo 22: Estructuras de datos ... 101
Introducción ... 101
Examples ... 101
Lista enlazada ... 101

Árbol binario ... 102
Capítulo 23: Eventos ... 104
Sintaxis ... 104
Observaciones ... 104
Examples ... 104
Fuentes y manejadores ... 104
¿Qué son los eventos? ... 104
Manipuladores ... 104
Fuentes ... 106
Pasar datos de nuevo al origen del evento ... 107
Usando parámetros pasados por referencia ... 107
Utilizando objetos mutables ... 107
Capítulo 24: Formularios de usuario ... 109
Examples ... 109
Mejores prácticas ... 109
Trabaja con una nueva instancia cada vez ... 109
Implementar la lógica en otro lugar ... 109
La persona que llama no debe ser molestada con los controles ... 110
Manejar el evento QueryClose ... 110
Ocultar, no cerrar ... 111
Nombre las cosas ... 111
Manejo de consultas cerrar ... 111
Un formulario de usuario cancelable ... 112
Capítulo 25: Interfaces ... 114
Introducción ... 114
Examples ... 114
Interfaz simple - Flyable ... 114
Múltiples interfaces en una clase - Volable y Swimable ... 115
Capítulo 26: Lectura de 2GB + archivos en binario en VBA y File Hashes ... 118
Introducción ... 118
Observaciones ... 118

MÉTODOS PARA LA CLASE POR MICROSOFT ... 118
PROPIEDADES DE LA CLASE POR MICROSOFT ... 119
Modulo normal ... 119
Examples ... 119
Esto tiene que estar en un módulo de clase, ejemplos más adelante referidos como "Aleatori ... 119
Código para calcular el hash de archivo en un módulo estándar ... 123
Cálculo de todos los archivos hash de una carpeta raíz ... 125
Ejemplo de hoja de trabajo: ... 125
Código ... 126
Capítulo 27: Literales de cadenas - Escape, caracteres no imprimibles y continuaciones de ... 129
Observaciones ... 129
Examples ... 129
Escapando al "personaje ... 129
Asignando literales de cadena larga ... 129
Usando constantes de cadena VBA ... 130
Capítulo 28: Llamadas API ... 132
Introducción ... 132
Observaciones ... 132
Examples ... 133
Declaración y uso de la API ... 133
API de Windows - Módulo dedicado (1 de 2) ... 136
API de Windows - Módulo dedicado (2 de 2) ... 140
API de Mac ... 144
Consigue monitores totales y resolución de pantalla ... 145
FTP y APIs regionales ... 146
Capítulo 29: Los operadores ... 150
Observaciones ... 150
Examples ... 150
Operadores matematicos ... 150
Operadores de Concatenacion ... 151
Operadores de comparación ... 152
Notas ... 153

Bitwise \ Operadores lógicos ... 155
Capítulo 30: Macro seguridad y firma de proyectos / módulos VBA. ... 158
Examples ... 158
Crear un certificado autofirmado digital válido SELFCERT.EXE ... 158
Capítulo 31: Manejo de errores ... 171
Examples ... 171
Evitando condiciones de error ... 171
Declaración de error ... 172
Estrategias de manejo de errores ... 172
Línea de números ... 173
Reanudar palabra clave ... 174
En error reanudar siguiente ... 175
Errores personalizados ... 176
Elevando tus propios errores de ejecución. ... 176
Capítulo 32: Manipulación de cuerdas de uso frecuente. ... 178
Introducción ... 178
Examples ... 178
Manipulación de cuerdas de uso frecuente ... 178
Capítulo 33: Manipulación de fecha y hora ... 180
Examples ... 180
Calendario ... 180
Ejemplo ... 180
Funciones de base ... 181
Recuperar sistema DateTime ... 181
Función de temporizador ... 181
IsDate () ... 182
Funciones de extraccion ... 182
Función DatePart () ... 183
Funciones de calculo ... 185
DateDiff () ... 185
FechaAgregar () ... 185

Conversión y Creación ... 186
CDate () ... 186
DateSerial () ... 187
Capítulo 34: Midiendo la longitud de las cuerdas ... 189
Observaciones ... 189
Examples ... 189
Usa la función de Len para determinar el número de caracteres en una cadena ... 189
Utilice la función LenB para determinar el número de bytes en una cadena ... 189
Prefiero `If Len (myString) = 0 Then` sobre` If myString = "" Then` ... 189
Capítulo 35: Objeto Scripting.Dictionary ... 191
Observaciones ... 191
Examples ... 191
Propiedades y metodos ... 191
Agregando datos con Scripting.Dictionary (Maximum, Count) ... 193
Obteniendo valores únicos con Scripting.Dictionary ... 195
Capítulo 36: Opción VBA Palabra clave ... 197
Sintaxis ... 197
Parámetros ... 197
Observaciones ... 197
Examples ... 198
Opción explícita ... 198
Opción Comparar {Binary | Texto | Base de datos} ... 199
Opción Comparar Binario ... 199
Opción Comparar texto ... 199
Opción Comparar base de datos ... 200
Base de opciones {0 | 1} ... 200
Ejemplo en Base 0: ... 200
Mismo ejemplo con Base 1 ... 201
El código correcto con Base 1 es: ... 201
Capítulo 37: Pasando Argumentos ByRef o ByVal ... 203
Introducción ... 203

Observaciones ... 203
Pasando matrices ... 203
Examples ... 203
Pasando variables simples ByRef y ByVal ... 203
ByRef ... 204
Modificador por defecto ... 204
Pasando por referencia ... 205
Forzar ByVal en el sitio de la llamada ... 206
ByVal ... 206
Pasando por valor ... 206
Capítulo 38: Personajes no latinos ... 208
Introducción ... 208
Examples ... 208
Texto no latino en código VBA ... 208
Identificadores no latinos y cobertura de idiomas ... 209
Capítulo 39: Procedimiento de llamadas ... 211
Sintaxis ... 211
Parámetros ... 211
Observaciones ... 211
Examples ... 211
Sintaxis de llamada implícita ... 211
Caso extremo ... 211
Valores de retorno ... 212
Esto es confuso. ¿Por qué no usar siempre paréntesis? ... 212
Tiempo de ejecución ... 212
Tiempo de compilación ... 213
Sintaxis explícita de llamadas ... 213
Argumentos opcionales ... 213
Capítulo 40: Recursion ... 215
Introducción ... 215
Observaciones ... 215

Examples ... 215
Factoriales ... 215
Carpeta Recursion ... 215
Capítulo 41: Scripting.FileSystemObject ... 217
Examples ... 217
Creando un FileSystemObject ... 217
Leyendo un archivo de texto usando un FileSystemObject ... 217
Creando un archivo de texto con FileSystemObject ... 218
Escribir en un archivo existente con FileSystemObject ... 218
Enumerar archivos en un directorio usando FileSystemObject ... 218
Enumerar recursivamente carpetas y archivos ... 219
Tira la extensión de archivo de un nombre de archivo ... 220
Recupera solo la extensión de un nombre de archivo ... 220
Recuperar solo la ruta de acceso de un archivo ... 221
Uso de FSO.BuildPath para crear una ruta completa desde la ruta de la carpeta y el nombre ... 221
Capítulo 42: Subcadenas ... 222
Observaciones ... 222
Examples ... 222
Use Left o Left $ para obtener los 3 caracteres más a la izquierda en una cadena ... 222
Use Right o Right $ para obtener los 3 caracteres más a la derecha en una cadena ... 222
Use Mid o Mid $ para obtener caracteres específicos dentro de una cadena ... 222
Utilice Recortar para obtener una copia de la cadena sin espacios iniciales ni finales ... 223
Capítulo 43: Tipos de datos y límites ... 224
Examples ... 224
Byte ... 224
Entero ... 225
Booleano ... 225
Largo ... 226
Soltero ... 226
Doble ... 226
Moneda ... 227
Fecha ... 227

Cuerda ... 228
Longitud variable ... 228
Longitud fija ... 228
Largo largo ... 229
Variante ... 229
LongPtr ... 230
Decimal ... 231
Capítulo 44: Trabajando con ADO ... 232
Observaciones ... 232
Examples ... 232
Haciendo una conexión a una fuente de datos ... 232
Recuperando registros con una consulta ... 233
Ejecutando funciones no escalares ... 234
Creando comandos parametrizados ... 235
Capítulo 45: Trabajar con archivos y directorios sin usar FileSystemObject ... 237
Observaciones ... 237
Examples ... 237
Determinar si existen carpetas y archivos ... 237
Creación y eliminación de carpetas de archivos ... 238
Capítulo 46: VBA orientado a objetos ... 240
Examples ... 240
Abstracción ... 240
Los niveles de abstracción ayudan a determinar cuándo dividir las cosas ... 240
Encapsulacion ... 240
La encapsulación oculta los detalles de implementación del código del cliente ... 241
Usando interfaces para imponer la inmutabilidad ... 241
Usando un método de fábrica para simular un constructor ... 243
Polimorfismo ... 244
El polimorfismo es la capacidad de presentar la misma interfaz para diferentes implementac ... 244
Código verificable depende de abstracciones ... 246
Creditos ... 248

Acerca de

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

Capítulo 1: Empezando con VBA

Observaciones

Esta sección proporciona una descripción general de qué es vba y por qué un desarrollador puede querer usarlo.

También debe mencionar cualquier tema grande dentro de vba, y vincular a los temas relacionados. Dado que la Documentación para vba es nueva, es posible que deba crear versiones iniciales de los temas relacionados.

Versiones

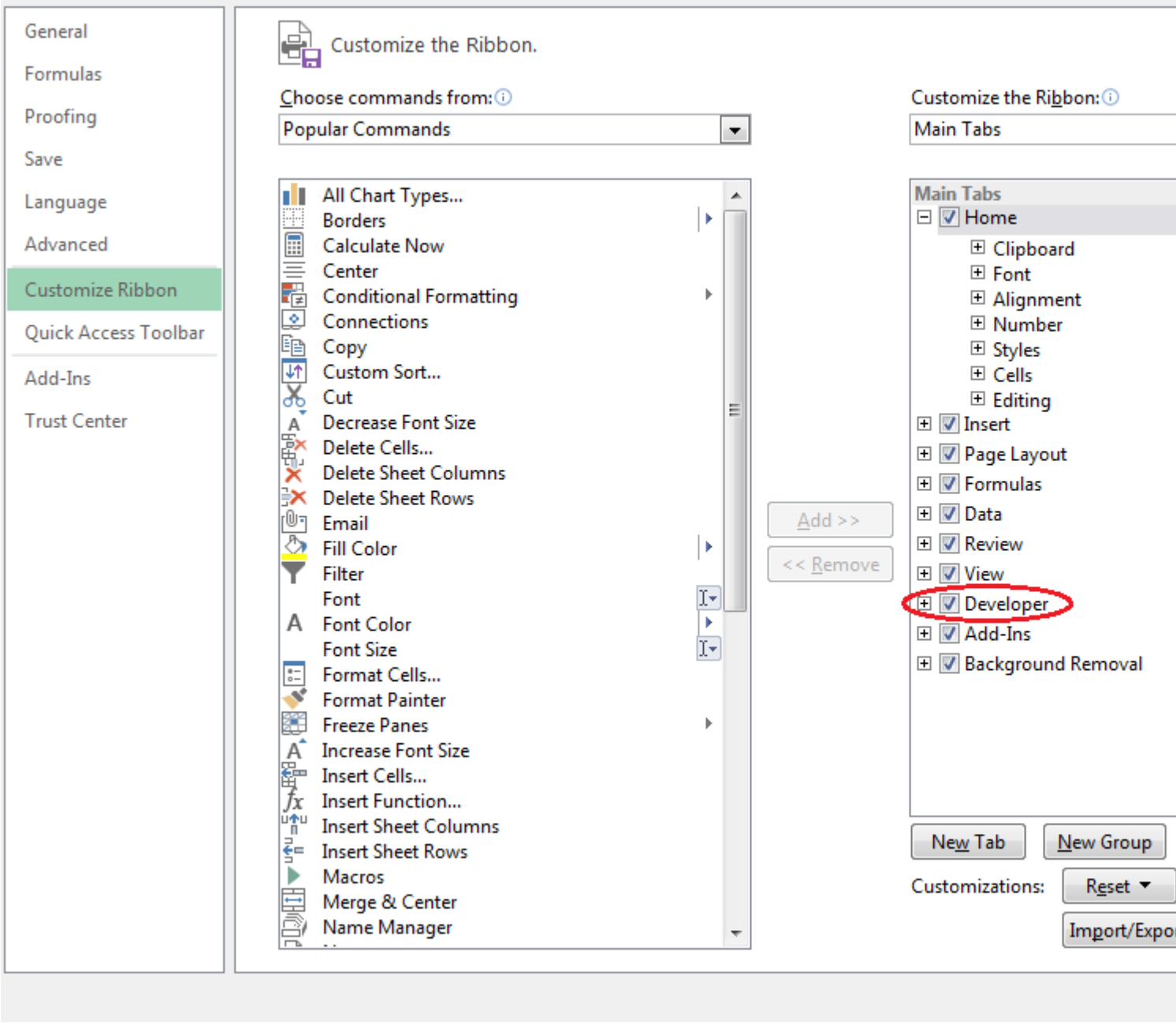
Versión	Versiones de oficina	Notas de fecha de lanzamiento	Fecha de lanzamiento
Vba6	? - 2007	[Algún tiempo después] [1]	1992-06-30
Vba7	2010 - 2016	[blog.techkit.com] [2]	2010-04-15
VBA para Mac	2004, 2011 - 2016		2004-05-11

Examples

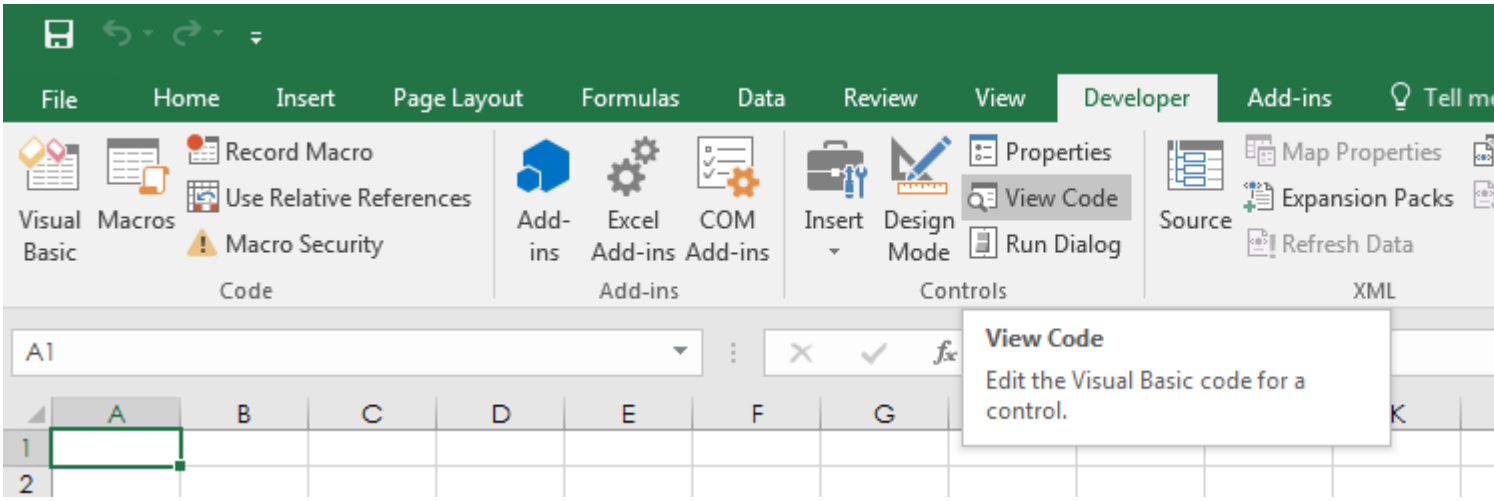
Accediendo al Editor de Visual Basic en Microsoft Office

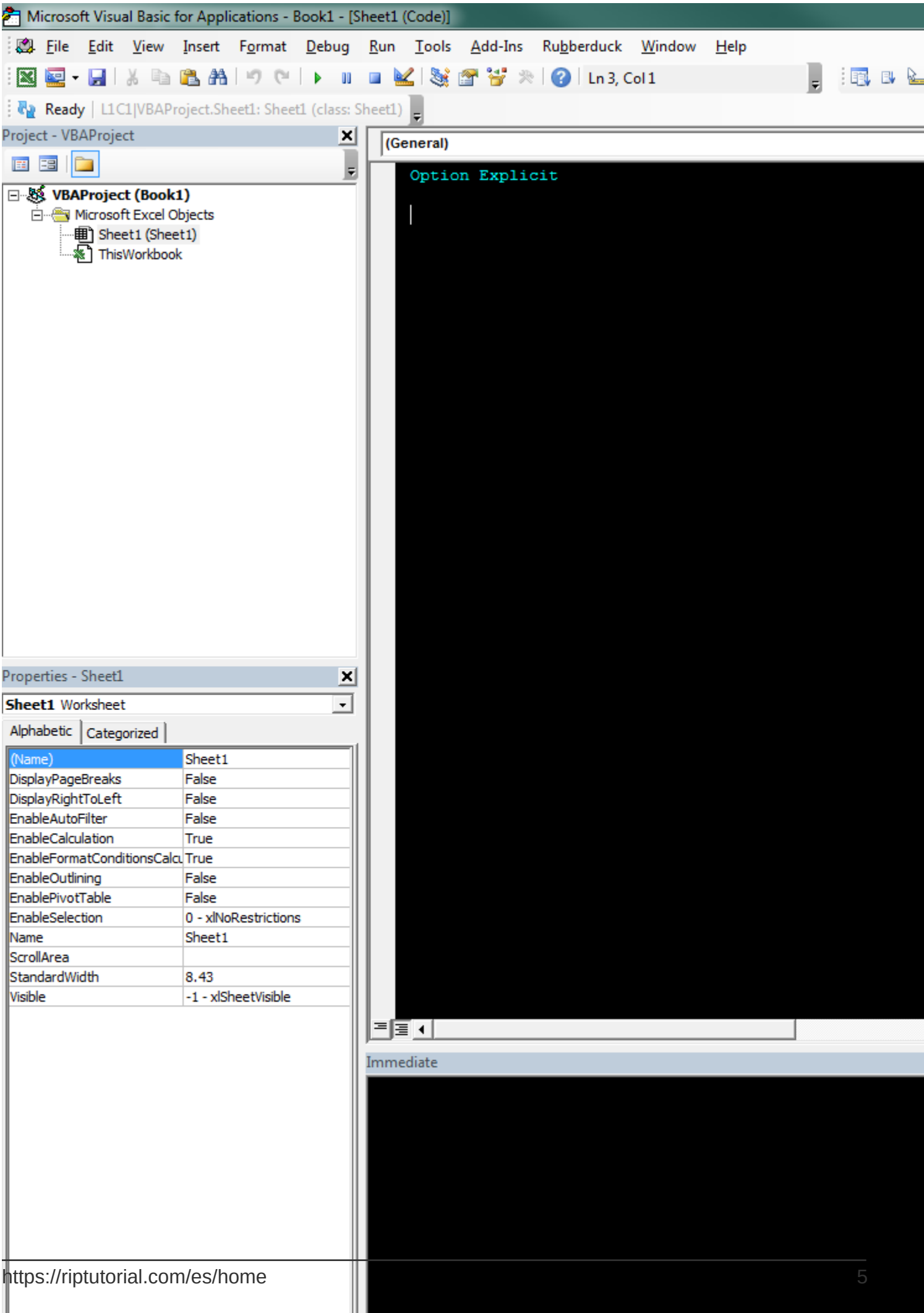
Puede abrir el editor de VB en cualquiera de las aplicaciones de Microsoft Office presionando `Alt` + `F11` o yendo a la pestaña Desarrollador y haciendo clic en el botón "Visual Basic". Si no ve la pestaña Desarrollador en la cinta, verifique si está habilitada.

Por defecto, la pestaña Desarrollador está deshabilitada. Para habilitar la pestaña Desarrollador, vaya a Archivo -> Opciones, seleccione Personalizar cinta en la lista de la izquierda. En la vista de árbol derecha "Personalizar la cinta de opciones", busque el elemento del árbol Desarrollador y configure la casilla de verificación Activar Desarrollador. Haga clic en Aceptar para cerrar el cuadro de diálogo Opciones.



La pestaña Desarrollador ahora está visible en la cinta de opciones en la que puede hacer clic en "Visual Basic" para abrir el Editor de Visual Basic. Alternativamente, puede hacer clic en "Ver código" para ver directamente el panel de código del elemento activo actualmente, por ejemplo, Hoja de trabajo, Gráfico, 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
(General)
Option Explicit
Properties - Sheet1
Sheet1 Worksheet
Alphabetic
Categorized
(Name) Sheet1
DisplayPageBreaks False
DisplayRightToLeft False
EnableAutoFilter False
EnableCalculation True
EnableFormatConditionsCalcu True
EnableOutlining False
EnablePivotTable False
EnableSelection 0 - xlNoRestrictions
Name Sheet1
ScrollArea
StandardWidth 8.43
Visible -1 - xlSheetVisible
Immediate

en su barra de herramientas o simplemente presione la tecla `F5` . ¡Felicidades! Has construido tu primer módulo VBA propio.

Depuración

La depuración es una forma muy poderosa de observar de cerca y corregir el código que funciona incorrectamente (o que no funciona).

Ejecutar código paso a paso

Lo primero que debe hacer durante la depuración es detener el código en ubicaciones específicas y luego ejecutarlo línea por línea para ver si sucede lo que se espera.

- Punto de interrupción (`F9` , Depuración - Alternar punto de interrupción): puede agregar un punto de interrupción a cualquier línea ejecutada (por ejemplo, no a declaraciones), cuando la ejecución llega a ese punto, se detiene y le da el control al usuario.
- También puede agregar la palabra clave `Stop` a una línea en blanco para que el código se detenga en esa ubicación en tiempo de ejecución. Esto es útil si, por ejemplo, antes de las líneas de declaración a las que no puede agregar un punto de interrupción con `F9`
- Paso a (`F8` , Depuración - Paso a): ejecuta solo una línea de código, si es una llamada de una función / sub definida por el usuario, se ejecuta línea por línea.
- Paso a paso (`Shift` + `F8` , Depuración - Paso a paso): ejecuta una línea de código, no ingresa funciones / funciones definidas por el usuario.
- Salir (`Ctrl` + `Shift` + `F8` , Depurar - Salir): sale de la sub / función actual (ejecute el código hasta su final).
- Ejecutar al cursor (`Ctrl` + `F8` , Depurar - Ejecutar al cursor): ejecute el código hasta llegar a la línea con el cursor.
- Puede usar `Debug.Print` para imprimir líneas en la ventana Inmediata en tiempo de ejecución. También puede utilizar la `Debug.?` como un atajo para `Debug.Print`

Ventana de relojes

Ejecutar el código línea por línea es solo el primer paso, necesitamos conocer más detalles y una herramienta para eso es la ventana de visualización (Ver - Ventana de visualización), aquí puede ver los valores de las expresiones definidas. Para agregar una variable a la ventana de visualización, ya sea:

- Haga clic derecho en él y luego seleccione "Agregar reloj".
- Haga clic derecho en la ventana del reloj, seleccione "Agregar reloj".
- Ir a depurar - Añadir reloj.

Cuando agrega una nueva expresión, puede elegir si solo desea ver su valor, o también interrumpir la ejecución del código cuando es verdadera o cuando cambia su valor.

Ventana inmediata

La ventana inmediata le permite ejecutar código arbitrario o imprimir elementos precediéndolos con la palabra clave `Print` o con un único signo de interrogación " `?` "

Algunos ejemplos:

- `? ActiveSheet.Name` - devuelve el nombre de la hoja activa
- `Print ActiveSheet.Name` : devuelve el nombre de la hoja activa
- `? foo` - devuelve el valor de `foo` *
- `x = 10` series de `x` a 10 *

* La obtención / configuración de valores para variables a través de la ventana Inmediata solo se puede realizar durante el tiempo de ejecución

Depuración de mejores prácticas

Siempre que su código no funcione como se esperaba, lo primero que debe hacer es leerlo con cuidado, en busca de errores.

Si eso no ayuda, entonces comience a depurarlo; para procedimientos cortos, puede ser eficiente simplemente ejecutarlo línea por línea, para los más largos probablemente necesite establecer puntos de interrupción o interrupciones en las expresiones observadas, el objetivo aquí es encontrar que la línea no funcione como se espera.

Una vez que tenga la línea que da el resultado incorrecto, pero la razón aún no está clara, intente simplificar expresiones o reemplace las variables con constantes, que pueden ayudar a entender si el valor de las variables es incorrecto.

Si todavía no puedes resolverlo, y pide ayuda:

- Incluya la menor parte posible de su código para comprender su problema.
- Si el problema no está relacionado con el valor de las variables, entonces reemplácelas por constantes. (así, en lugar de `Sheets(a*b*c+d^2).Range(addressOfRange)` escriba `Sheets(4).Range("A2")`)
- Describa qué línea da el comportamiento incorrecto y cuál es (error, resultado incorrecto ...)

Lea Empezando con VBA en línea: https://riptutorial.com/es/vba/topic/802/empezando-con-vba

Capítulo 2: Arrays

Examples

Declarando un Array en VBA

Declarar una matriz es muy similar a declarar una variable, excepto que necesita declarar la dimensión de la Matriz justo después de su nombre:

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

De forma predeterminada, las matrices en VBA se **indexan desde CERO** , por lo tanto, el número dentro del paréntesis no se refiere al tamaño de la matriz, sino al **índice del último elemento**

Elementos de acceso

El acceso a un elemento de la matriz se realiza utilizando el nombre de la matriz, seguido del índice del elemento, entre paréntesis:

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

Indexación de matrices

Puede cambiar la indexación de Arrays colocando esta línea en la parte superior de un módulo:

```
Option Base 1
```

Con esta línea, todos los Arrays declarados en el módulo serán **indexados desde UNO** .

Índice especifico

También puede declarar cada Array con su propio índice usando la palabra clave `To` , y el límite inferior y superior (= índice):

```
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
```

Declaración dinámica

Cuando no conoce el tamaño de su Array antes de su declaración, puede usar la declaración dinámica y la palabra clave `ReDim` :

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

Tenga en cuenta que el uso de la palabra clave `ReDim` eliminará cualquier contenido anterior de su Array. Para evitar esto, puede usar la palabra clave `Preserve` después de `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 de Split para crear una matriz a partir de una cadena

Función de división

devuelve una matriz unidimensional basada en cero que contiene un número específico de subcadenas.

Sintaxis

División (expresión [, **delimitador** [, **límite** [, **comparar**]]] **)**

Parte	Descripción
expresión	Necesario. Expresión de cadena que contiene subcadenas y delimitadores. Si *expresión* es una cadena de longitud cero ("" o vbNullString), **Split** devuelve una matriz vacía que no contiene elementos ni datos. En este caso, la matriz devuelta tendrá un LBound de 0 y un UBound de -1.
delimitador	Opcional. Carácter de cadena utilizado para identificar los límites de subcadena. Si se omite, se supone que el carácter de espacio ("") es el delimitador. Si el ***delimitador*** es una cadena de longitud cero, se devuelve una matriz de un solo elemento que contiene la cadena de ***expresión*** completa.
límite	Opcional. Número de subcadenas a devolver; -1 indica que se devuelven todas las subcadenas.
comparar	Opcional. Valor numérico que indica el tipo de comparación que se utilizará al evaluar las subcadenas. Ver la sección de configuración para los valores.

Ajustes

El argumento de ***comparación*** puede tener los siguientes valores:

Constante	Valor	Descripción
Descripción	-1	Realiza una comparación utilizando la configuración de la

Constante	Valor	Descripción
		instrucción de **comparación de opciones** .
vbBinaryCompare	0	Realiza una comparación binaria.
vbTextCompare	1	Realiza una comparación textual.
vbDatabaseCompare	2	Sólo Microsoft Access. Realiza una comparación basada en información en tu base de datos.

Ejemplo

En este ejemplo se muestra cómo funciona la división mostrando varios estilos. Los comentarios mostrarán el conjunto de resultados para cada una de las diferentes opciones de división realizadas. Finalmente, se demuestra cómo recorrer la matriz de cadena devuelta.

```
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
```

Iterando elementos de una matriz

Para ... siguiente

Usar la variable iterador como el número de índice es la forma más rápida de iterar los elementos

de una matriz:

```
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
```

Los bucles anidados se pueden usar para iterar matrices multidimensionales:

```
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
```

Para cada ... siguiente

A `For Each...Next` loop también se puede utilizar para iterar matrices, si el rendimiento no importa:

```
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` loop iterará todas las dimensiones de exterior a interno (el mismo orden que los elementos están en la memoria), por lo que no hay necesidad de bucles anidados:

```
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
```

Tenga en cuenta que los bucles `For Each` se utilizan mejor para iterar objetos de la `Collection` , si el rendimiento es importante.

Los 4 fragmentos anteriores producen el mismo resultado:

```
 0
 1
 2
 3
```

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

Arrays dinámicos

Agregar y reducir dinámicamente las variables en una matriz es una gran ventaja para cuando la información que está tratando no tiene un número determinado de variables.

Agregando valores dinámicamente

Simplemente puede cambiar el tamaño de la matriz con la declaración `ReDim` , esto cambiará el tamaño de la matriz, pero si desea conservar la información ya almacenada en la matriz, necesitará la parte `Preserve` .

En el siguiente ejemplo, creamos una matriz y la aumentamos en una variable más en cada iteración, a la vez que conservamos los valores que ya se encuentran en la matriz.

```
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
```

Eliminar valores dinámicamente

Podemos utilizar la misma lógica para disminuir la matriz. En el ejemplo, el valor "último" se eliminará de la matriz.

```
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
```

Restablecimiento de una matriz y reutilización dinámica

También podemos reutilizar los arreglos que creamos para no tener muchos en la memoria, lo que haría más lento el tiempo de ejecución. Esto es útil para matrices de varios tamaños. Un fragmento de código que podría utilizar para reutilizar la matriz es volver a `ReDim` la matriz a `(0)` , atribuir una variable a la matriz y volver a aumentar la matriz libremente.

En el siguiente fragmento, construyo una matriz con los valores de 1 a 40, vacío la matriz y la recargo con los valores de 40 a 100, todo esto se hace de forma dinámica.

```
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
```

Arreglos irregulares (Arrays of Arrays)

Matrices dentadas NO matrices multidimensionales

Las matrices de matrices (matrices irregulares) no son lo mismo que las matrices multidimensionales si las piensa visualmente Las matrices multidimensionales se verían como matrices (rectangulares) con un número definido de elementos en sus dimensiones (matrices

internas), mientras que la matriz dentada sería como una anual Calendario con las matrices internas que tienen un número diferente de elementos, como días en diferentes meses.

Aunque los Arreglos Jagged son bastante complicados y difíciles de usar debido a sus niveles anidados y no tienen muchos tipos de seguridad, pero son muy flexibles, le permiten manipular diferentes tipos de datos con bastante facilidad y no necesitan contener datos no utilizados o elementos vacios.

Creación de una matriz irregular

En el siguiente ejemplo, iniciaremos una matriz dentada que contiene dos matrices, una para Nombres y otra para Números, y luego accederemos a un elemento de cada una.

```
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
```

Creación y lectura dinámica de matrices dentadas

También podemos ser más dinámicos en nuestro tiempo aproximado para construir los arreglos, imaginemos que tenemos una hoja de datos del cliente en Excel y queremos construir un arreglo para generar los detalles 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
```

Construiremos dinámicamente una matriz de encabezado y una matriz de clientes, el encabezado contendrá los títulos de columna y la matriz de clientes contendrá la información de cada cliente / fila como matrices.

```
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.

El resultado del fragmento de código anterior es una matriz dentada con dos matrices, una de esas matrices con 4 elementos, 2 niveles de sangría y la otra es otra matriz dentada que contiene 5 matrices de 4 elementos cada una y 3 niveles de sangría, vea a continuación la estructura:

```
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)
```

Para acceder a la información, deberá tener en cuenta la estructura de la matriz irregular que cree, en el ejemplo anterior puede ver que la `Main Array` contiene una matriz de `Headers` y una matriz de matrices (`Customers`), por lo tanto, con diferentes formas de Accediendo a los elementos.

Ahora leeremos la información de la `Main Array` e imprimiremos cada información de los clientes como `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
```

RECUERDE para hacer un seguimiento de la estructura de su Jagged Array, en el ejemplo anterior para acceder al nombre de un cliente es accediendo a `Main_Array -> Customers -> CustomerNumber -> Name` que es de tres niveles, para devolver `"Person4"` que necesitará la ubicación de los Clientes en Main_Array, luego la Ubicación del cliente cuatro en la matriz Jagged Clientes y por último la ubicación del elemento que necesita, en este caso `Main_Array(1)(3)(0)` que es `Main_Array(Customers)(CustomerNumber)(Name)` .

Arreglos Multidimensionales

Arreglos Multidimensionales

Como su nombre lo indica, las matrices multidimensionales son matrices que contienen más de una dimensión, generalmente dos o tres, pero pueden tener hasta 32 dimensiones.

Una matriz múltiple funciona como una matriz con varios niveles, tome como ejemplo una comparación entre una, dos y tres dimensiones.

One Dimension es su matriz típica, parece una lista de elementos.

```
Dim 1D(3) as Variant

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

Two Dimensions se vería como una cuadrícula de Sudoku o una hoja de Excel. Al inicializar la

matriz, definiría cuántas filas y columnas tendría la matriz.

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

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

Three Dimensions comenzaría a parecerse al Cubo de Rubik, al inicializar la matriz, definiría filas y columnas y las capas / profundidades que tendría la matriz.

```
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)
```

Otras dimensiones podrían pensarse como la multiplicación de la 3D, por lo que una 4D (1,3,3,3) sería dos matrices 3D de lado a lado.

Array de dos dimensiones

Creando

El siguiente ejemplo será una compilación de una lista de empleados, cada empleado tendrá un conjunto de información en la lista (Nombre, Apellido, Dirección, Correo electrónico, Teléfono ...), el ejemplo esencialmente se almacenará en la matriz (empleado, información) siendo el (0,0) es el primer nombre del primer empleado.

```
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
```

Redimensionamiento

Cambiar el tamaño o `ReDim Preserve` una Multi-Array como la norma para una matriz de One-Dimension obtendría un error, en lugar de eso, la información debe transferirse a una matriz Temporal con el mismo tamaño que el original más el número de filas / columnas para agregar. En el siguiente ejemplo, veremos cómo inicializar una matriz temporal, transferir la información de la matriz original, llenar los elementos vacíos restantes y reemplazar la matriz temporal por la matriz original.

```
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
```

Cambiar los valores de los elementos

Para cambiar / alterar los valores en un elemento determinado se puede hacer simplemente llamando a la coordenada para cambiar y dándole un nuevo valor: `Employees(0, 0) = "NewValue"`

Alternativamente, iterar a través de las coordenadas usa las condiciones para hacer coincidir los valores correspondientes a los parámetros necesarios:

```
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
```

Leyendo

El acceso a los elementos de la matriz se puede hacer con un bucle anidado (iterando cada elemento), Loop y Coordenadas (iterar filas y acceder a columnas directamente), o acceder directamente con ambas coordenadas.

```
'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)
```

***Recuerde** , siempre es útil mantener un mapa de matriz cuando se usan matrices multidimensionales, ya que pueden convertirse fácilmente en confusión.*

Array de tres dimensiones

Para la matriz 3D, usaremos la misma premisa que la matriz 2D, con la adición de no solo almacenar el empleado y la información, sino también la construcción en la que trabajan.

La matriz 3D tendrá los Empleados (se pueden considerar como Filas), la Información (Columnas) y la Construcción que se pueden considerar como hojas diferentes en un documento de Excel, tienen el mismo tamaño entre ellas, pero cada hoja tiene un Diferentes conjuntos de información en sus celdas / elementos. La matriz 3D contendrá ***n*** número de matrices 2D.

Creando

Una matriz 3D necesita 3 coordenadas para ser inicializada `Dim 3Darray(2,5,5) As Variant` la primera coordenada en la matriz será el número de Building / Sheets (diferentes conjuntos de filas y columnas), la segunda coordenada definirá Rows y la tercera Columnas La `Dim` arriba dará como resultado una matriz 3D con 108 elementos (`3*6*6`), que tendrá efectivamente 3 conjuntos diferentes de matrices 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
```

Redimensionamiento

Cambiar el tamaño de una matriz 3D es similar a cambiar el tamaño de una 2D, primero cree una matriz temporal con el mismo tamaño del original agregando una en la coordenada del parámetro para aumentar, la primera coordenada aumentará el número de conjuntos en la matriz, la segunda y Las terceras coordenadas aumentarán el número de filas o columnas en cada conjunto.

El siguiente ejemplo aumenta la cantidad de Filas en cada conjunto en uno, y llena los nuevos elementos agregados con nueva información.

```
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
```

Cambio de valores de elementos y lectura

La lectura y el cambio de los elementos en la matriz 3D se pueden hacer de manera similar a la forma en que hacemos la matriz 2D, simplemente ajuste el nivel adicional en los bucles y coordenadas.

```
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)
```

Lea Arrays en línea: https://riptutorial.com/es/vba/topic/3064/arrays

Capítulo 3: Asignando cadenas con caracteres repetidos

Observaciones

Hay veces que necesita asignar una variable de cadena con un carácter específico repetido un número específico de veces. VBA proporciona dos funciones principales para este propósito:

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

Examples

Utilice la función de cadena para asignar una cadena con n caracteres repetidos

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

Usa las funciones de Cadena y Espacio para asignar una cadena de n caracteres

```
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, " ")
```

Lea Asignando cadenas con caracteres repetidos en línea:
https://riptutorial.com/es/vba/topic/3581/asignando-cadenas-con-caracteres-repetidos

Capítulo 4: Atributos

Sintaxis

- Atributo VB_Name = "ClassOrModuleName"
- Atributo VB_GlobalNameSpace = False 'Ignored
- Atributo VB_Creatable = Falso 'Ignorado
- Atributo VB_PredeclaredId = {True | Falso}
- Atributo VB_Exposed = {True | Falso}
- Atributo variableName.VB_VarUserMemId = 0 'Cero indica que este es el miembro predeterminado de la clase.
- Atributo variableName.VB_VarDescription = "alguna cadena" 'Agrega el texto a la información del Examinador de objetos para esta variable.
- Atributo procName.VB_Description = "alguna cadena" 'Agrega el texto a la información del Examinador de objetos para el procedimiento.
- Atributo procName.VB_UserMemId = {0 | -4}
    - '0: hace que la función sea el miembro predeterminado de la clase.
    - '-4: Especifica que la función devuelve un Enumerador.

Examples

VB_Name

VB_Name especifica la clase o el nombre del módulo.

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

Una nueva instancia de esta clase sería creada con

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

VB_GlobalNameSpace

En VBA, este atributo se ignora. No fue portado desde VB6.

En VB6, crea una instancia global predeterminada de la clase (un "acceso directo") para que se pueda acceder a los miembros de la clase sin usar el nombre de la clase. Por ejemplo, `DateTime` (como en `DateTime.Now`) es en realidad parte de la clase `VBA.Conversion` .

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

VB_Createable

Este atributo se ignora. No fue portado desde VB6.

En VB6, se usó en combinación con el atributo `VB_Exposed` para controlar la accesibilidad de clases fuera del proyecto actual.

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

Resultaría en una `Public Class` , a la que se podría acceder desde otros proyectos, pero esta funcionalidad no existe en VBA.

VB_PredeclaredId

Crea una instancia global predeterminada de una clase. Se accede a la instancia predeterminada a través del nombre de la clase.

Declaración

```
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
```

Llamada

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

De alguna manera, esto simula el comportamiento de las clases estáticas en otros idiomas, pero a diferencia de otros idiomas, aún puede crear una instancia de la clase.

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

VB_Expuesto

Controla las características de creación de instancias de una clase.

```
Attribute VB_Exposed = False
```

Hace que la clase sea `Private` . No se puede acceder fuera del proyecto actual.

```
Attribute VB_Exposed = True
```

Expone la clase de `Public` , fuera del proyecto. Sin embargo, dado que `VB_Createable` se ignora en VBA, las instancias de la clase no se pueden crear directamente. Esto es equivalente a la siguiente clase VB.Net.

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

Para obtener una instancia externa al proyecto, debe exponer una fábrica para crear instancias. Una forma de hacerlo es con un módulo `Public` regular.

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

Dado que los módulos públicos son accesibles desde otros proyectos, esto nos permite crear nuevas instancias de nuestras clases `Public - Not Createable` .

VB_Descripción

Agrega una descripción de texto a un miembro de clase o módulo que se hace visible en el Explorador de objetos. Idealmente, todos los miembros públicos de una interfaz / API pública deberían tener una descripción.

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

Nota: todos los miembros de acceso de una propiedad (`Get` , `Let` , `Set`) usan la misma descripción.

VB_ [Var] UserMemId

`VB_VarUserMemId` (para variables de alcance de módulo) y `VB_UserMemId` (para procedimientos) se usan en VBA principalmente para dos cosas.

Especificando el miembro predeterminado de

una clase

Una clase de `List` que encapsularía una `Collection` desearía tener una propiedad de `Item` , por lo que el código del cliente puede hacer esto:

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

Pero con un atributo `VB_UserMemId` establecido en 0 en la propiedad `Item` , el código del cliente puede hacer esto:

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

Solo un miembro puede tener legalmente `VB_UserMemId = 0` en cualquier clase dada. Para propiedades, especifique el atributo en el `Get` acceso `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
```

Hacer una clase iteratable con una construcción de bucle `For Each`

Con el valor mágico `-4` , el atributo `VB_UserMemId` le dice a VBA que este miembro produce un enumerador, lo que permite que el código del cliente haga esto:

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

La forma más fácil de implementar este método es llamar al captador de propiedades ocultas `[_NewEnum]` en una `Collection` interna / encapsulada; el identificador debe estar entre corchetes debido al subrayado principal que lo convierte en un identificador de VBA ilegal:

```
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
```

Lea Atributos en línea: https://riptutorial.com/es/vba/topic/5321/atributos

Capítulo 5: Automatización o uso de otras bibliotecas de aplicaciones.

Introducción

Si usa los objetos en otras aplicaciones como parte de su aplicación de Visual Basic, es posible que desee establecer una referencia a las bibliotecas de objetos de esas aplicaciones. Esta documentación proporciona una lista, fuentes y ejemplos de cómo usar bibliotecas de diferentes softwares, como Windows Shell, Internet Explorer, XML HttpRequest y otros.

Sintaxis

- expression.CreateObject (ObjectName)
- expresión; Necesario. Una expresión que devuelve un objeto de aplicación.
- Nombre del objeto; Cadena requerida. El nombre de clase del objeto a crear. Para obtener información acerca de los nombres de clase válidos, vea OLE Programmatic Identifiers.

Observaciones

- MSDN-Understanding Automation

  Cuando una aplicación admite la automatización, Visual Basic puede acceder a los objetos que la aplicación expone. Use Visual Basic para manipular estos objetos invocando métodos en el objeto u obteniendo y configurando las propiedades del objeto.

- MSDN-Verifique o agregue una referencia de biblioteca de objetos

  Si usa los objetos en otras aplicaciones como parte de su aplicación de Visual Basic, es posible que desee establecer una referencia a las bibliotecas de objetos de esas aplicaciones. Antes de poder hacer eso, primero debe asegurarse de que la aplicación proporciona una biblioteca de objetos.

- Cuadro de diálogo MSDN-References

  Le permite seleccionar los objetos de otra aplicación que desea que estén disponibles en su código configurando una referencia a la biblioteca de objetos de esa aplicación.

- Método MSDN-CreateObject

  Crea un objeto de automatización de la clase especificada. Si la aplicación ya se está ejecutando, CreateObject creará una nueva instancia.

Examples

Expresiones regulares de VBScript

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

Herramientas> Referencias> Expresiones regulares de Microsoft VBScript #. #
DLL asociado: VBScript.dll
Fuente: Internet Explorer 1.0 y 5.5

- MSDN-Microsoft refuerza VBScript con expresiones regulares
- Sintaxis de expresiones regulares de MSDN (secuencias de comandos)
- expertos-intercambio - Uso de expresiones regulares en Visual Basic para aplicaciones y Visual Basic 6
- Cómo usar expresiones regulares (Regex) en Microsoft Excel tanto en la celda como en loops en el SO.
- regular-expressions.info/vbscript
- regular-expressions.info/vbscriptexample
- WIKI-Expresión regular

Código

Puede usar estas funciones para obtener resultados RegEx, concatenar todas las coincidencias (si son más de 1) en una cadena y mostrar el resultado en la celda de 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
```

Objeto del sistema de archivos de secuencias de comandos

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

Herramientas> Referencias> Microsoft Scripting Runtime
DLL asociado: ScrRun.dll
Fuente: Windows OS

Acceso a archivos de MSDN con FileSystemObject

> El modelo de objeto de sistema de archivos (FSO) proporciona una herramienta basada en objetos para trabajar con carpetas y archivos. Le permite usar la sintaxis conocida de object.method con un rico conjunto de propiedades, métodos y eventos para procesar carpetas y archivos. También puede emplear las instrucciones y comandos tradicionales de Visual Basic.
>
> El modelo FSO le da a su aplicación la capacidad de crear, alterar, mover y eliminar carpetas, o de determinar si y dónde existen carpetas particulares. También le permite obtener información sobre las carpetas, como sus nombres y la fecha en que se crearon o modificaron por última vez.

Temas de MSDN-FileSystemObject : " *... explica el concepto de FileSystemObject y cómo usarlo* " . Exceltrick-FileSystemObject en VBA - Explicación Scripting.FileSystemObject

Diccionario de secuencias de comandos objeto

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

Herramientas> Referencias> Microsoft Scripting Runtime
DLL asociado: ScrRun.dll
Fuente: Windows OS

Objeto Scripting.Dictionary
MSDN-objeto del diccionario

Objeto de Internet Explorer

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

Herramientas> Referencias> Controles de Internet de Microsoft
DLL asociado: ieframe.dll
Fuente: navegador de Internet Explorer

Objeto MSDN-InternetExplorer

> Controla una instancia de Windows Internet Explorer a través de la automatización.

Miembros básicos de Internet Explorer Objec

El siguiente código debe introducir cómo funciona el objeto de IE y cómo manipularlo a través de VBA. Recomiendo pasar a través de él, de lo contrario podría fallar durante múltiples navegaciones.

```
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
```

Raspado web

Lo más común que se puede hacer con IE es raspar parte de la información de un sitio web, o completar un formulario de sitio web y enviar información. Vamos a ver cómo hacerlo.

Consideremos el código fuente de *example.com* :

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

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

Podemos usar el código como abajo para obtener y configurar información:

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

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

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

        'Print info in immediate window
        With .Document 'the source code HTML "below" the displayed page.
            Stop 'VBE Stop. Continue line by line to see what happens.
            Debug.Print .GetElementsByTagName("title")(0).innerHtml 'prints "Example Domain"
            Debug.Print .GetElementsByTagName("h1")(0).innerHtml 'prints "Example Domain"
            Debug.Print .GetElementsByTagName("p")(0).innerHtml 'prints "This domain is
established..."
            Debug.Print .GetElementsByTagName("p")(1).innerHtml 'prints "<a
```

```
href="http://www.iana.org/domains/example">More information...</a>"
            Debug.Print .GetElementsByTagName("p")(1).innerText 'prints "More information..."
            Debug.Print .GetElementsByTagName("a")(0).innerText 'prints "More information..."

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

        .Quit 'close the application window
    End With 'ie

End Sub
```

Que esta pasando? El jugador clave aquí es el **.Documento** , que es el código fuente HTML. Podemos aplicar algunas consultas para obtener las Colecciones u Objetos que deseamos. Por ejemplo, el `IE.Document.GetElementsByTagName("title")(0).innerHtml` . `GetElementsByTagName` devuelve una **colección** de elementos HTML, que tienen la etiqueta " *título* ". Sólo hay una etiqueta de este tipo en el código fuente. La **colección** está basada en 0 Entonces para obtener el primer elemento agregamos `(0)` . Ahora, en nuestro caso, solo queremos el `innerHtml` (un String), no el Element Object en sí mismo. Por eso especificamos la propiedad que queremos.

Hacer clic

Para seguir un enlace en un sitio, podemos usar múltiples métodos:

```
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
```

Biblioteca de objetos HTML de Microsoft o mejor amigo de IE

Para aprovechar al máximo el HTML que se carga en el IE, puede (o debería) usar otra Biblioteca, es decir, la *Biblioteca de Objetos HTML de Microsoft* . Más sobre esto en otro ejemplo.

Principales problemas de IE

El problema principal con IE es verificar que la página haya terminado de cargarse y esté lista para interactuar con ella. The `Do While... Loop` ayuda, pero no es confiable.

Además, el uso de IE solo para raspar el contenido HTML es OVERKILL. ¿Por qué? Debido a que el navegador está diseñado para navegar, es decir, mostrar la página web con todos los CSS, JavaScripts, imágenes, ventanas emergentes, etc. Si solo necesita los datos en bruto, considere un enfoque diferente. Por ejemplo, utilizando XML HTTPRequest . Más sobre esto en otro ejemplo.

Lea Automatización o uso de otras bibliotecas de aplicaciones. en línea: https://riptutorial.com/es/vba/topic/8916/automatizacion-o-uso-de-otras-bibliotecas-de-aplicaciones-

Capítulo 6: Buscando dentro de las cadenas la presencia de subcadenas.

Observaciones

Cuando necesita verificar la presencia o posición de una subcadena dentro de una cadena, VBA ofrece las funciones `InStr` e `InStrRev` que devuelven la posición de carácter de la subcadena en la cadena, si está presente.

Examples

Usa InStr para determinar si una cadena contiene una subcadena

```
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
```

Utilice InStr para encontrar la posición de la primera instancia de una subcadena

```
Const baseString As String = "Foo Bar"
Dim containsBar As Boolean

Dim posB As Long
posB = InStr(1, baseString, "B", vbBinaryCompare)
'posB = 5
```

Utilice InStrRev para encontrar la posición de la última instancia de una subcadena

```
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
```

Lea Buscando dentro de las cadenas la presencia de subcadenas. en línea:

https://riptutorial.com/es/vba/topic/3480/buscando-dentro-de-las-cadenas-la-presencia-de-subcadenas-

Capítulo 7: Clasificación

Introducción

A diferencia de .NET Framework, la biblioteca de Visual Basic para aplicaciones no incluye rutinas para ordenar matrices.

Hay dos tipos de soluciones: 1) implementar un algoritmo de clasificación desde cero, o 2) usar rutinas de clasificación en otras bibliotecas comúnmente disponibles.

Examples

Implementación de algoritmos - Ordenación rápida en una matriz unidimensional

De la función de ordenamiento de matriz 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
```

Uso de la biblioteca de Excel para ordenar una matriz unidimensional

Este código aprovecha la clase de `Sort` en la biblioteca de objetos de Microsoft Excel.

Para más información, consulte:

- Copiar un rango a un rango virtual
- ¿Cómo copiar el rango seleccionado en una matriz dada?

```
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
```

Lea Clasificación en línea: https://riptutorial.com/es/vba/topic/8836/clasificacion

Capítulo 8: Colecciones

Observaciones

Una `Collection` es un objeto contenedor que se incluye en el tiempo de ejecución de VBA. No se requieren referencias adicionales para usarlo. Una `Collection` se puede utilizar para almacenar artículos de cualquier tipo de datos y permite la recuperación mediante el índice ordinal del artículo o mediante el uso de una clave única opcional.

Comparación de características con matrices y diccionarios

	Colección	Formación	Diccionario
Puede ser redimensionado	Sí	A veces [1]	Sí
Los artículos son ordenados	Sí	Sí	Si [2]
Los artículos son fuertemente tipados	No	Sí	No
Los artículos pueden ser recuperados por ordinal	Sí	Sí	No
Nuevos artículos pueden ser insertados en ordinal	Sí	No	No
Cómo determinar si un artículo existe	Iterar todos los elementos	Iterar todos los elementos	Iterar todos los elementos
Los artículos pueden ser recuperados por clave	Sí	No	Sí
Las claves distinguen entre mayúsculas y minúsculas	No	N / A	Opcional [3]
Cómo determinar si existe una clave	Manejador de errores	N / A	Función `.Exists`
Eliminar todos los elementos	`.Remove` y. `.Remove`	`Erase` , `ReDim`	`.RemoveAll` función

[1] Solo se puede cambiar el tamaño de las matrices dinámicas, y solo la última dimensión de las matrices multidimensionales.

[2] Las `.Keys` y `.Items` subyacentes están ordenados.

[3] Determinado por la propiedad `.CompareMode` .

Examples

Agregar elementos a una colección

Los elementos se agregan a una `Collection` llamando a su método `.Add` :

Sintaxis:

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

Parámetro	Descripción
ít	El artículo para almacenar en la `Collection` . Esto puede ser esencialmente cualquier valor al que se pueda asignar una variable, incluidos los tipos primitivos, las matrices, los objetos y `Nothing` .
llave	Opcional. Una `String` que sirve como un identificador único para recuperar elementos de la `Collection` . Si la clave especificada ya existe en la `Collection` , se producirá un error de tiempo de ejecución 457: "Esta clave ya está asociada con un elemento de esta colección".
antes de	Opcional. Una clave existente (valor de `String`) *o* índice (valor numérico) para insertar el elemento antes en la `Collection` . Si se proporciona un valor, el parámetro *posterior* ***debe*** estar vacío o ***debe*** aparecer un error de tiempo de ejecución 5: "Llamada o argumento de procedimiento no válido". Si se pasa una clave de `String` que no existe en la `Collection` , se generará un error de tiempo de ejecución 5: "Llamada o argumento de procedimiento no válido". Si se pasa un índice numérico que no existe en la `Collection` , se generará un error de tiempo de ejecución 9: "Subíndice fuera de rango".
después	Opcional. Una clave existente (valor de `String`) *o* índice (valor numérico) para insertar el elemento después en la `Collection` . Si se da un valor, el parámetro *anterior* ***debe*** estar vacío. Los errores planteados son idénticos al parámetro *anterior* .

Notas:

- Las claves ***no*** distinguen entre mayúsculas y minúsculas. `.Add "Bar", "Foo"` y `.Add "Baz", "foo"` resultará en una colisión de teclas.
- Si no se proporciona ninguno de los parámetros opcionales *antes* o *después* , el elemento se agregará después del último elemento de la `Collection` .
- Las inserciones realizadas especificando un parámetro *antes* o *después* alterarán los índices numéricos de los miembros existentes para que coincidan con su nueva posición.

Esto significa que se debe tener cuidado al realizar inserciones en bucles utilizando índices numéricos.

Uso de la muestra:

```
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
```

Eliminar elementos de una colección

Los elementos se eliminan de una `Collection` llamando a su método `.Remove` :

Sintaxis:

```
.Remove(index)
```

Parámetro	Descripción
índice	El elemento a eliminar de la `Collection` . Si el valor pasado es un tipo numérico o `Variant` con un subtipo numérico, se interpretará como un índice numérico. Si el valor pasado es una `String` o `Variant` contiene una cadena, se interpretará como una clave. Si se pasa una clave de cadena que no existe en la `Collection` , se generará un error de tiempo de ejecución 5: "Llamada o argumento de procedimiento no válido". Si se pasa un índice numérico que no existe en la `Collection` , se generará un error de tiempo de ejecución 9: "Subíndice fuera de rango".

Notas:

- La eliminación de un elemento de una `Collection` cambiará los índices numéricos de todos los elementos que se encuentren después de él en la `Collection` . `For` bucles que usan índices numéricos y eliminan elementos, deben ejecutarse *hacia atrás* (`Step -1`) para evitar excepciones de subíndices y elementos omitidos.
- Por lo general, los elementos ***no*** deben eliminarse de una `Collection` desde dentro de un bucle `For Each` ya que puede dar resultados impredecibles.

Uso de la muestra:

```
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
```

Obtener el recuento de artículos de una colección

El número de elementos en una `Collection` se puede obtener llamando a su función `.Count` :

Sintaxis:

```
.Count()
```

Uso de la muestra:

```
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
```

Recuperar elementos de una colección

Los elementos se pueden recuperar de una `Collection` llamando a la función `.Item` .

Sintaxis:

```
.Item(index)
```

Parámetro	Descripción
índice	El elemento a recuperar de la `Collection` . Si el valor pasado es un tipo numérico o `Variant` con un subtipo numérico, se interpretará como un índice numérico. Si el valor pasado es una `String` o `Variant` contiene una cadena, se interpretará como una clave. Si se pasa una clave de cadena que no existe en la `Collection` , se generará un error de tiempo de ejecución 5: "Llamada o argumento de procedimiento no válido". Si se pasa un índice numérico que no existe en la `Collection` , se generará un error de tiempo de ejecución 9: "Subíndice fuera de rango".

Notas:

- `.Item` es el miembro predeterminado de la `Collection` . Esto permite flexibilidad en la sintaxis como se muestra en el uso de muestra a continuación.
- Los índices numéricos están basados en 1.
- Las claves ***no*** distinguen entre mayúsculas y minúsculas. `.Item("Foo")` y `.Item("foo")` refieren a la misma clave.
- El parámetro de *índice* ***no se*** convierte implícitamente a un número desde una `String` o viceversa. Es totalmente posible que `.Item(1)` y `.Item("1")` refieran a diferentes elementos de la `Collection` .

Uso de la muestra (índices):

```
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
```

Uso de la muestra (claves):

```
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
```

Uso de muestra (sintaxis 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
```

Tenga en cuenta que la sintaxis de bang (`!`) Está permitida porque `.Item` es el miembro predeterminado y puede tomar un solo argumento de `String` . La utilidad de esta sintaxis es cuestionable.

Determinar si una clave o elemento existe en una colección

Llaves

A diferencia de un Scripting.Dictionary , una `Collection` no tiene un método para determinar si existe una clave determinada *o* una forma de recuperar las claves que están presentes en la `Collection` . El único método para determinar si una clave está presente es usar el controlador de errores:

```
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
```

Artículos

La única manera de determinar si un artículo está contenido en una `Collection` es iterar sobre la `Collection` hasta que se encuentre el artículo. Tenga en cuenta que debido a que una `Collection` puede contener primitivos u objetos, se necesita un manejo adicional para evitar errores de tiempo de ejecución durante las comparaciones:

```
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
```

Borrar todos los artículos de una colección

La forma más fácil de eliminar todos los elementos de una `Collection` es simplemente reemplazarlo con una nueva `Collection` y dejar que el anterior quede fuera del alcance:

```
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
```

Sin embargo, si hay varias referencias a la `Collection` retenida, este método solo le dará una `Collection` vacía *para la variable asignada* .

```
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
```

En este caso, la forma más fácil de borrar el contenido es recorrer la cantidad de elementos en la `Collection` y eliminar repetidamente el elemento más bajo:

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

Lea Colecciones en línea: https://riptutorial.com/es/vba/topic/5838/colecciones

Capítulo 9: Comentarios

Observaciones

Bloques de comentarios

Si necesita comentar o descomentar varias líneas a la vez, puede usar los botones de la **barra de herramientas de edición** del IDE:

> **Bloque de comentarios** : agrega un solo apóstrofe al inicio de todas las líneas seleccionadas

> **Descomprimir bloque** : elimina el primer apóstrofe del inicio de todas las líneas seleccionadas

Comentarios multilínea Muchos otros idiomas admiten comentarios de bloque multilínea, pero VBA solo permite comentarios de una sola línea.

Examples

Apóstrofe Comentarios

Un comentario se marca con un apóstrofe (`'`) y se ignora cuando se ejecuta el código. Los comentarios ayudan a explicar su código a futuros lectores, incluido usted mismo.

Como todas las líneas que comienzan con un comentario se ignoran, también se pueden usar para evitar que el código se ejecute (mientras se depura o refactoriza). Colocar un apóstrofe `'` antes de que su código lo convierta en un comentario. (Esto se llama *comentar* la línea.)

```
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
```

REM comentarios

```
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
```

Lea Comentarios en línea: https://riptutorial.com/es/vba/topic/2059/comentarios

Capítulo 10: Compilación condicional

Examples

Cambiar el comportamiento del código en tiempo de compilación

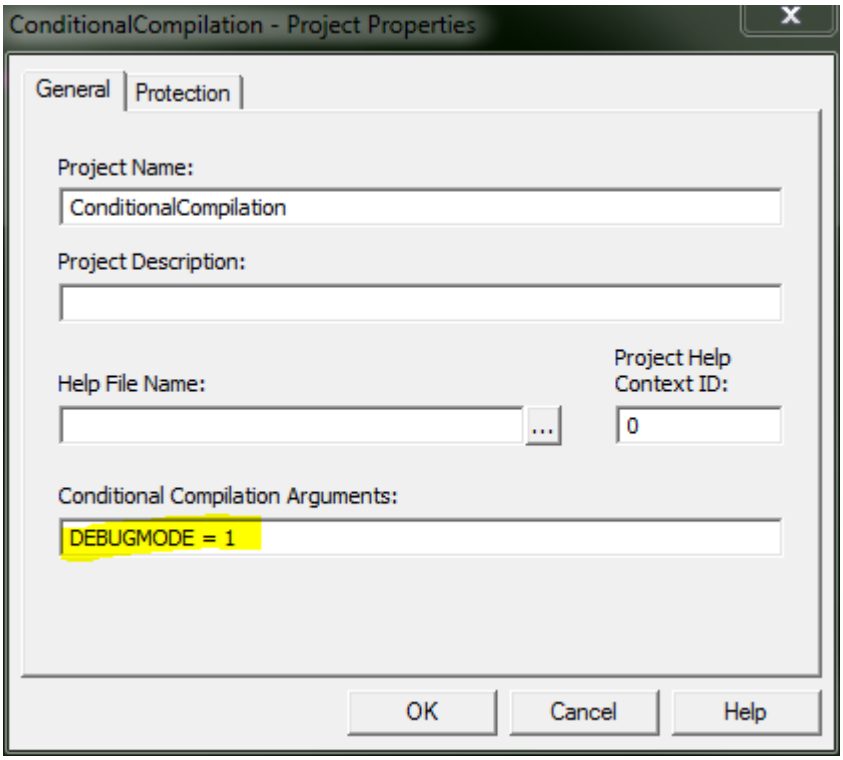
La directiva `#Const` se usa para definir una constante de preprocesador personalizada. Estos pueden ser utilizados posteriormente por `#If` para controlar qué bloques de código se compilan y ejecutan.

```
#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
```

Esto hace que el valor de la `filepath` de `filepath` se establezca en `"C:\Users\UserName\Path\To\File.txt"` . Eliminar la línea `#Const` , o cambiarla a `#Const DEBUGMODE = 0` resultará en que la `"\\server\share\path\to\file.txt" filepath` se establezca en `"\\server\share\path\to\file.txt"` .

#Const Scope

La directiva `#Const` solo es efectiva para un archivo de código único (módulo o clase). Debe declararse para todos y cada uno de los archivos en los que desea utilizar su constante personalizada. Alternativamente, puede declarar un `#Const` globalmente para su proyecto yendo a Herramientas >> [Nombre de su proyecto] Propiedades del proyecto. Esto abrirá el cuadro de diálogo de propiedades del proyecto donde ingresaremos la declaración constante. En el cuadro "Argumentos de compilación condicional", escriba `[constName] = [value]` . Puede ingresar más de 1 constante separándolas con dos puntos, como `[constName1] = [value1] : [constName2] = [value2]` .



Constantes predefinidas

Algunas constantes de compilación ya están predefinidas. Las que existan dependerán del grado de bit de la versión de Office en la que esté ejecutando VBA. Tenga en cuenta que Vba7 se introdujo junto con Office 2010 para admitir versiones de Office de 64 bits.

Constante	16 bits	32 bits	64 bits
Vba6	Falso	Si vba6	Falso
Vba7	Falso	Si vba7	Cierto
Win16	Cierto	Falso	Falso
Win32	Falso	Cierto	Cierto
Win64	Falso	Falso	Cierto
Mac	Falso	Si mac	Si mac

Tenga en cuenta que Win64 / Win32 se refiere a la versión de Office, no a la versión de Windows. Por ejemplo, Win32 = TRUE en Office de 32 bits, incluso si el sistema operativo es una versión de Windows de 64 bits.

Uso de Declare Imports que funciona en todas las versiones de 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
```

Esto se puede simplificar un poco según las versiones de Office que necesite admitir. Por ejemplo, no hay mucha gente que aún admita versiones de Office de 16 bits. La última versión de 16 bit office fue la versión 4.3, lanzada en 1994 , por lo que la siguiente declaración es suficiente para casi todos los casos modernos (incluido Office 2007).

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

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

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

Si no tiene que admitir nada más antiguo que Office 2010, esta declaración funciona bien.

```
' 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
```

Lea Compilación condicional en línea: https://riptutorial.com/es/vba/topic/3364/compilacion-condicional

Capítulo 11: Convenciones de nombres

Examples

Nombres de variables

Las variables mantienen los datos. Nómbrelos con el nombre para el que se usan, **no con su tipo de datos** o alcance, usando un **nombre** . Si se siente obligado a *numerar* sus variables (por ejemplo `thing1, thing2, thing3`), entonces considere usar una estructura de datos apropiada en su lugar (por ejemplo, una matriz, una `Collection` o un `Dictionary`).

Los nombres de las variables que representan un *conjunto* de valores iterativo, por ejemplo, una matriz, una `Collection` , un `Dictionary` o un `Range` de celdas, deben ser plurales.

Algunas convenciones comunes de nomenclatura de VBA van así:

Para variables de nivel de procedimiento :

`camelCase`

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

    Dim procedureVariable As Long
    Dim someOtherVariable As String

End Sub
```

Para las variables de nivel de módulo:

`PascalCase`

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

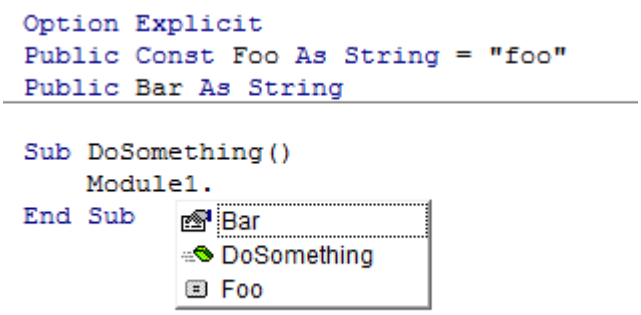
Para constantes:

`SHOUTY_SNAKE_CASE` se usa comúnmente para diferenciar constantes de variables:

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

Public Sub SomeProcedure()

    Const PROCEDURE_CONSTANT As Long = 10

End Sub
```

Sin embargo, los nombres de `PascalCase` hacen que el código tenga un aspecto más `PascalCase` y

sean tan buenos, dado que IntelliSense usa diferentes íconos para variables y constantes:



Notación húngara

Asígnele un nombre según su uso, **no después de su tipo de datos** o alcance.

"La notación húngara hace que sea más fácil ver cuál es el tipo de variable"

Si escribe su código, como los procedimientos se adhieren al *Principio de Responsabilidad Única* (como debería ser), nunca debe mirar una serie de declaraciones de variables en la parte superior de cualquier procedimiento; declare las variables lo más cerca posible de su primer uso, y su tipo de datos siempre estará a la vista si los declara con un tipo explícito. El acceso directo `Ctrl` + `i` de VBE también se puede usar para mostrar el tipo de una variable en una información sobre herramientas.

Para lo que se usa una variable es mucho más información útil que su tipo de datos, *especialmente* en un lenguaje como VBA que convierte feliz e implícitamente un tipo en otro según sea necesario.

Considere `iFile` y `strFile` en este ejemplo:

```
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
```

Comparar 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` se pasa `ByRef` en el ejemplo de arriba, pero junto al hecho de que tenemos la suerte de ver que se pasa *explícitamente* como tal, no hay ninguna indicación de que `strData` está realmente *devuelto* por la función.

El ejemplo de abajo lo nombra fuera de `outContent` ; este prefijo de `out` es para lo que se inventó la notación húngara: para ayudar a aclarar para *qué se utiliza una variable* , en este caso para identificarla claramente como un parámetro "fuera".

Esto es útil, porque IntelliSense por sí mismo no muestra `ByRef` , incluso cuando el parámetro se pasa *explícitamente* por referencia:

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

Lo que lleva a...

Húngaro Hecho a la derecha

La notación húngara originalmente no tenía nada que ver con tipos de variables . De hecho, la notación húngara *hecha correctamente* es realmente útil. Considere este pequeño ejemplo (`ByVal` y `As Integer` eliminado por brevedad):

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

Comparar con:

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

`src` y `dst` son *los* prefijos de *notación húngara* aquí, y transmiten información *útil* que de otra manera no se puede inferir de los nombres de parámetros o IntelliSense que nos muestra el tipo declarado.

Por supuesto, hay una mejor manera de transmitirlo todo, mediante una *abstracción* adecuada y palabras reales que se pueden pronunciar en voz alta y tiene sentido, como un ejemplo artificial:

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

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

Nombres de procedimientos

Los procedimientos *hacen algo* . Nómbrelos después de lo que están haciendo, usando un **verbo** . Si no es posible nombrar con precisión un procedimiento, es probable que el procedimiento esté *haciendo demasiadas cosas* y deba dividirse en procedimientos más pequeños y más especializados.

Algunas convenciones comunes de nomenclatura de VBA van así:

Para todos los procedimientos:

`PascalCase`

```
Public Sub DoThing()

End Sub

Private Function ReturnSomeValue() As [DataType]

End Function
```

Para procedimientos de manejo de eventos:

`ObjectName_EventName`

```
Public Sub Workbook_Open()

End Sub

Public Sub Button1_Click()

End Sub
```

Los manejadores de eventos generalmente son nombrados automáticamente por el VBE; el cambio de nombre sin cambiar el nombre del objeto y / o el evento manejado romperá el código; el código se ejecutará y compilará, pero el procedimiento del controlador quedará huérfano y nunca se ejecutará.

Miembros booleanos

Considere una función de retorno booleano:

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

Comparar con:

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

El prefijo `Can` *tiene* el mismo propósito que el prefijo `b` : identifica el valor de retorno de la función como `Boolean` . Pero `Can` leer mejor que `b` :

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

Comparado con:

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

Considere el uso de prefijos como `Can` , `Is` o `Has` delante de los miembros que regresan a Boolean (funciones y propiedades), pero solo cuando agrega valor. Esto cumple con las directrices actuales de nomenclatura de Microsoft .

Lea Convenciones de nombres en línea: https://riptutorial.com/es/vba/topic/1184/convenciones-de-nombres

Capítulo 12: Convertir otros tipos a cadenas

Observaciones

VBA convertirá implícitamente algunos tipos en cadenas según sea necesario y sin ningún trabajo adicional por parte del programador, pero VBA también proporciona una serie de funciones de conversión de cadenas explícitas, y también puede escribir las suyas propias.

Tres de las funciones más utilizadas son `CStr` , `Format` y `StrConv` .

Examples

Usa CStr para convertir un tipo numérico a una cadena

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

Utilice Formato para convertir y formatear un tipo numérico como una cadena

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

Utilice StrConv para convertir una matriz de bytes de caracteres de un solo byte en una cadena

```
'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"
```

Convertir implícitamente una matriz de bytes de caracteres de múltiples bytes en una cadena

```
'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"
```

Lea Convertir otros tipos a cadenas en línea: https://riptutorial.com/es/vba/topic/3467/convertir-otros-tipos-a-cadenas

Capítulo 13: Copiando, devolviendo y pasando matrices.

Examples

Copiando Arrays

Puede copiar una matriz VBA en una matriz del mismo tipo utilizando el operador `=` . Las matrices deben ser del mismo tipo, de lo contrario, el código arrojará un error de compilación "No se puede asignar a la matriz".

```
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
```

La matriz de origen puede ser fija o dinámica, pero la matriz de destino debe ser dinámica. Si intenta copiar en una matriz fija, se producirá un error de compilación "No se puede asignar a la matriz". Cualquier dato preexistente en la matriz receptora se pierde y sus límites y dimensiones se cambian al mismo que la matriz de origen.

```
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 vez que se realiza la copia, las dos matrices se separan en la memoria, es decir, las dos variables no son referencias a los mismos datos subyacentes, por lo que los cambios realizados en una matriz no aparecen en la otra.

```
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
```

Copiar matrices de objetos

Con matrices de objetos, las *referencias* a esos objetos se copian, no los objetos en sí. Si se realiza un cambio en un objeto en una matriz, también parecerá que se cambia en la otra matriz; ambos hacen referencia al mismo objeto. Sin embargo, establecer un elemento en un objeto diferente en una matriz no lo establecerá en ese objeto en la otra matriz.

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

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

destination = source

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

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

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

Variantes que contienen una matriz

También puede copiar una matriz en y desde una variable variable. Al copiar desde una variante, debe contener una matriz del mismo tipo que la matriz receptora, de lo contrario lanzará un error de tiempo de ejecución "No coincide el tipo".

```
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
```

Devolviendo Arrays desde Funciones

Una función en un módulo normal (pero no en un módulo de clase) puede devolver una matriz poniendo `()` después del tipo de datos.

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

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

    arrayOfPiDigits = outputArray
End Function
```

El resultado de la función se puede colocar en una matriz dinámica del mismo tipo o una variante. También se puede acceder directamente a los elementos utilizando un segundo conjunto de corchetes, sin embargo, esto llamará a la función cada vez, por lo que es mejor almacenar los resultados en una nueva matriz si planea usarlos más de una vez.

```
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
```

Tenga en cuenta que lo que se devuelve es en realidad una copia de la matriz dentro de la función, no una referencia. Por lo tanto, si la función devuelve el contenido de una matriz estática, sus datos no pueden modificarse mediante el procedimiento de llamada.

Salida de una matriz a través de un argumento de salida

Normalmente es una buena práctica de codificación que los argumentos de un procedimiento sean entradas y resultados a través del valor de retorno. Sin embargo, las limitaciones de VBA a veces hacen que sea necesario que un procedimiento genere datos a través de un argumento `ByRef`.

Salida a una matriz fija

```
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
```

Salida de una matriz de un método de clase

Un argumento de salida también se puede usar para generar una matriz desde un método / procedimiento en un módulo de clase

```
' 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
```

Pasando matrices a procedimientos

Las matrices se pueden pasar a los procedimientos poniendo `()` después del nombre de la variable de matriz.

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

Las matrices *deben* ser pasadas por referencia. Si no se especifica ningún mecanismo de paso, por ejemplo, `myFunction(arr())` , VBA asumirá `ByRef` de forma predeterminada, sin embargo, es una buena práctica de codificación para hacerlo explícito. Intentar pasar una matriz por valor, por ejemplo, `myFunction(ByVal arr())` generará un error de compilación "El argumento de la matriz debe ser ByRef" (o un error de compilación "Error de sintaxis" si la `Auto Syntax Check` no está marcada en las opciones de VBE) .

Pasar por referencia significa que cualquier cambio en la matriz se conservará en el procedimiento de la llamada.

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

    Debug.Print doubleAndSum(source)  ' outputs 8
```

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

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

Si desea evitar cambiar la matriz original, tenga cuidado de escribir la función para que no cambie ningún elemento.

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

También puede crear una copia de trabajo de la matriz y trabajar 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
```

Lea Copiando, devolviendo y pasando matrices. en línea:
https://riptutorial.com/es/vba/topic/9069/copiando--devolviendo-y-pasando-matrices-

Capítulo 14: Creación de una clase personalizada

Observaciones

Este artículo mostrará cómo crear una clase personalizada completa en VBA. Utiliza el ejemplo de un objeto `DateRange` , porque las fechas de inicio y finalización a menudo se pasan juntas a las funciones.

Examples

Agregar una propiedad a una clase

Un procedimiento de `Property` es una serie de sentencias que recupera o modifica una propiedad personalizada en un módulo.

Hay tres tipos de accesores de propiedad:

1. Una `Get` procedimiento que devuelve el valor de una propiedad.
2. Un procedimiento `Let` que asigna un valor (no de `Object`) a un objeto.
3. Un procedimiento `Set` que asigna una referencia de `Object` .

Los accesores de propiedades a menudo se definen en pares, utilizando tanto `Get` como `Let` / `Set` para cada propiedad. Una propiedad con sólo una `Get` procedimiento sería de sólo lectura, mientras que una propiedad con sólo una `Let` / `Set` procedimiento sería de sólo escritura.

En el siguiente ejemplo, se definen cuatro `DateRange` propiedades para la clase `DateRange` :

1. `StartDate` (*lectura / escritura*). Valor de fecha que representa la fecha anterior en un rango. Cada procedimiento utiliza el valor de la variable del módulo, `mStartDate` .
2. `EndDate` (*lectura / escritura*). Valor de fecha que representa la fecha posterior en un rango. Cada procedimiento utiliza el valor de la variable del módulo, `mEndDate` .
3. `DaysBetween` (*solo lectura*). Valor entero calculado que representa el número de días entre las dos fechas. Como solo hay un procedimiento de `Get` , esta propiedad no se puede modificar directamente.
4. `RangeToCopy` (*solo escritura*). Un procedimiento de `Set` utilizado para copiar los valores de un objeto `DateRange` existente.

```
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
```

Agregando Funcionalidad a una Clase

Se puede llamar a cualquier `Sub` , `Function` , o `Property` pública dentro de un módulo de clase precediendo a la llamada con una referencia de objeto:

```
Object.Procedure
```

En una clase de `DateRange` , se puede usar un `Sub` para agregar un número de días a la fecha de finalización:

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

Una `Function` podría devolver el último día del próximo fin de mes (tenga en cuenta que `GetFirstDayOfMonth` no sería visible fuera de la clase porque es privada):

```
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
```

Los procedimientos pueden aceptar argumentos de cualquier tipo, incluidas las referencias a los objetos de la clase que se está definiendo.

El siguiente ejemplo prueba si el objeto `DateRange` actual tiene una fecha de inicio y una fecha de finalización que incluye la fecha de inicio y finalización de otro objeto `DateRange` .

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

Tenga en cuenta el uso de la notación `Me` como una forma de acceder al valor del objeto que ejecuta el código.

Ámbito del módulo de clase, creación de instancias y reutilización.

De forma predeterminada, un nuevo módulo de clase es una clase privada, por lo que *solo* está disponible para la creación de instancias y su uso dentro del VBProject en el que está definido. Puede declarar, crear instancias y usar la clase en cualquier lugar del *mismo* proyecto:

```
'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
```

Pero a menudo escribirá clases que le gustaría usar en otros proyectos *sin* copiar el módulo entre proyectos. Si define una clase llamada `List` en `ProjectA` , y desea usar esa clase en `ProjectB` , entonces deberá realizar 4 acciones:

1. Cambie la propiedad de `PublicNotCreatable` instancias de la clase `List` en `ProjectA` en la ventana Propiedades, de `Private` a `PublicNotCreatable`

2. Cree una función pública de "fábrica" en `ProjectA` que cree y devuelva una instancia de una clase de `List` . Normalmente, la función de fábrica incluiría argumentos para la inicialización de la instancia de clase. La función de fábrica es necesaria porque `ProjectB` puede usar la clase, pero `ProjectB` no puede crear directamente una instancia de la clase de `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. En `ProjectB` agregue una referencia a `ProjectA` usando el menú `Tools..References...`

4. En `ProjectB` , declare una variable y asígnele una instancia de `List` usando la función de fábrica de `ProjectA`

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

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

Lea Creación de una clase personalizada en línea:
https://riptutorial.com/es/vba/topic/4464/creacion-de-una-clase-personalizada

Capítulo 15: Creando un procedimiento

Examples

Introducción a los procedimientos.

Un `Sub` es un procedimiento que realiza una tarea específica pero no devuelve un valor específico.

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

Si no se especifica ningún modificador de acceso, un procedimiento es `Public` por defecto.

Una `Function` es un procedimiento que recibe datos y devuelve un valor, idealmente sin efectos secundarios globales o de alcance de módulo.

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

Una `Property` es un procedimiento que *encapsula los* datos del módulo. Una propiedad puede tener hasta 3 accesores: `Get` para devolver un valor o una referencia de objeto, `Let` asignar un valor y / o `Set` para asignar una referencia de objeto.

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

Las propiedades generalmente se usan en módulos de clase (aunque también se permiten en módulos estándar), exponiendo el acceso a datos que de otra manera no son accesibles para el código que llama. Una propiedad que solo expone un acceso `Get` es "solo lectura"; una propiedad que solo expondría un acceso `Let` y / o `Set` es "solo escritura". Las propiedades de solo escritura no se consideran una buena práctica de programación: si el código del cliente puede *escribir* un valor, debería poder *leerlo* nuevamente. Considere implementar un procedimiento `Sub` en lugar de hacer una propiedad de solo escritura.

Devolviendo un valor

Un procedimiento de `Property Get Function` o `Property Get` puede (¡y debería!) Devolver un valor a su interlocutor. Esto se hace asignando el identificador del procedimiento:

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

Funcionar con ejemplos

Como se indicó anteriormente, las funciones son procedimientos más pequeños que contienen piezas pequeñas de código que pueden ser repetitivas dentro de un procedimiento.

Las funciones se utilizan para reducir la redundancia en el código.

Similar a un procedimiento, una función puede ser declarada con o sin una lista de argumentos.

La función se declara como un tipo de retorno, ya que todas las funciones devuelven un valor. El nombre y la variable de retorno de una función son los mismos.

1. Función con parámetro:

```
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. Función sin parámetro:

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

La función puede ser llamada de varias maneras dentro de una función. Dado que una función declarada con un tipo de retorno es básicamente una variable. Se usa similar a una variable.

Llamadas funcionales:

```
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
```

Además, la función también se puede utilizar como condiciones para y otras declaraciones condicionales.

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

Además, más funciones pueden tener modificadores como By ref y By val para sus argumentos.

Lea Creando un procedimiento en línea: https://riptutorial.com/es/vba/topic/1474/creando-un-

procedimiento

Capítulo 16: CreateObject vs. GetObject

Observaciones

En su forma más simple, `CreateObject` crea una instancia de un objeto, mientras que `GetObject` obtiene una instancia existente de un objeto. Determinar si un objeto se puede crear o obtener dependerá de su propiedad de Instancing . Algunos objetos son SingleUse (por ejemplo, WMI) y no se pueden crear si ya existen. Otros objetos (por ejemplo, Excel) son MultiUse y permiten que varias instancias se ejecuten a la vez. Si aún no existe una instancia de un objeto e intenta `GetObject` , recibirá el siguiente mensaje atrapable: `Run-time error '429': ActiveX component can't create object` .

GetObject requiere que al menos uno de estos dos parámetros opcionales esté presente:

1. *Nombre de ruta* - Variante (cadena): la ruta completa, incluido el nombre de archivo, del archivo que contiene el objeto. Este parámetro es opcional, pero se requiere *clase* si se omite el nombre de *ruta* .
2. *Clase* - Variante (String): una cadena que representa la definición formal (Application y ObjectType) del objeto. Se requiere *clase* si se omite el nombre de *ruta* .

CreateObject tiene un parámetro requerido y un parámetro opcional:

1. *Clase* - Variante (String): una cadena que representa la definición formal (Application y ObjectType) del objeto. *La clase* es un parámetro requerido.
2. *Servername* - Variant (String): el nombre del equipo remoto en el que se creará el objeto. Si se omite, el objeto se creará en la máquina local.

La clase siempre se compone de dos partes en forma de `Application.ObjectType` :

1. *Aplicación* : el nombre de la aplicación de la que forma parte el objeto. |
2. *Tipo de objeto* : el tipo de objeto que se crea. |

Algunas clases de ejemplo son:

1. Aplicación de Word.
2. Hoja de Excel
3. Scripting.FileSystemObject

Examples

Demostrando GetObject y CreateObject

Función MSDN-GetObject

> Devuelve una referencia a un objeto proporcionado por un componente ActiveX.

Utilice la función GetObject cuando haya una instancia actual del objeto o si desea crear el objeto con un archivo ya cargado. Si no hay una instancia actual y no desea que el objeto comience con un archivo cargado, use la función 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
```

Lea CreateObject vs. GetObject en línea: https://riptutorial.com/es/vba/topic/7729/createobject-vs--getobject

Capítulo 17: Cuerdas de concatenacion

Observaciones

Las cadenas se pueden concatenar, o unir, utilizando uno o más operadores de concatenación `&` .

Las matrices de cadenas también se pueden concatenar utilizando la función `Join` y proporcionar una cadena (que puede ser de longitud cero) para ser utilizada entre cada elemento de la matriz.

Examples

Concatenar cadenas utilizando el operador &

```
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"
```

Concatenar una matriz de cadenas mediante la función Unir

```
'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"
```

Lea Cuerdas de concatenacion en línea: https://riptutorial.com/es/vba/topic/3580/cuerdas-de-concatenacion

Capítulo 18: Declarando variables

Examples

Declaración implícita y explícita

Si un módulo de código no contiene `Option Explicit` en la parte superior del módulo, entonces el compilador creará automáticamente (es decir, "implícitamente") variables cuando los use. Ellos por defecto serán de tipo variable `Variant` .

```
Public Sub ExampleDeclaration()

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

End Sub
```

En el código anterior, si se especifica `Option Explicit` , el código se interrumpirá porque faltan las sentencias `Dim` requeridas para `someVariable` y `someOtherVariable` .

```
Option Explicit

Public Sub ExampleDeclaration()

    Dim someVariable As Long
    someVariable = 10

    Dim someOtherVariable As String
    someOtherVariable = "Hello World"

End Sub
```

Se considera una buena práctica usar Option Explicit en los módulos de código, para garantizar que declare todas las variables.

Vea VBA Best Practices cómo configurar esta opción por defecto.

Variables

Alcance

Se puede declarar una variable (en aumento del nivel de visibilidad):

- A nivel de procedimiento, usando la palabra clave `Dim` en cualquier procedimiento; una *variable local* .
- A nivel de módulo, usando la palabra clave `Private` en cualquier tipo de módulo; Un *campo privado* .
- A nivel de instancia, utilizando la palabra clave `Friend` en cualquier tipo de módulo de clase;

un *campo de amigos*
- A nivel de instancia, utilizando la palabra clave `Public` en cualquier tipo de módulo de clase; Un *campo público* .
- A nivel mundial, utilizando la palabra clave `Public` en un *módulo estándar* ; una *variable global* .

Las variables siempre deben declararse con el menor alcance posible: prefiera pasar parámetros a procedimientos, en lugar de declarar variables globales.

Ver modificadores de acceso para más información.

Variables locales

Use la palabra clave `Dim` para declarar una *variable local* :

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

La parte `[As Type]` de la sintaxis de la declaración es opcional. Cuando se especifica, establece el tipo de datos de la variable, que determina cuánta memoria se asignará a esa variable. Esto declara una variable de `String` :

```
Dim identifierName As String
```

Cuando no se especifica un tipo, el tipo es implícitamente `Variant` :

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

La sintaxis de VBA también admite la declaración de múltiples variables en una sola declaración:

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

Observe que se debe especificar `[As Type]` para cada variable (que no sean las de 'Variante'). Esta es una trampa relativamente común:

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

Variables estáticas

Las variables locales también pueden ser `Static` . En VBA, la palabra clave `Static` se usa para hacer que una variable "recuerde" el valor que tenía, la última vez que se llamó a un procedimiento:

```
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
```

Aquí la colección de `values` se declara como un `Static` local; Debido a que es una *variable de objeto* , se inicializa en `Nothing` . La condición que sigue a la declaración verifica si la referencia del objeto se `Set` antes: si es la primera vez que se ejecuta el procedimiento, la colección se inicializa. `DoSomethingElse` podría estar agregando o eliminando elementos, y aún estarán en la colección la próxima vez que se `DoSomething` .

Alternativa

> La palabra clave `Static` de VBA se puede malinterpretar fácilmente, *especialmente* por programadores experimentados que generalmente trabajan en otros idiomas. En muchos idiomas, `static` se utiliza para hacer que un miembro de clase (campo, propiedad, método, ...) pertenezca al *tipo en* lugar de a la *instancia* . El código en contexto `static` no puede hacer referencia al código en contexto de *instancia* . La palabra clave `Static` VBA significa algo muy diferente.

A menudo, un local `Static` podría implementarse como una variable (campo) `Private` a nivel de módulo; sin embargo, esto pone en tela de juicio el principio por el cual una variable debe declararse con el menor alcance posible; confíe en sus instintos, use el que prefiera, ambos funcionarán ... pero usar `Static` sin entender lo que hace podría provocar errores interesantes.

Dim vs. Private

La palabra clave `Dim` es legal en los niveles de procedimiento y módulo; su uso a nivel de módulo es equivalente a usar la palabra clave `Private` :

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

La palabra clave `Private` solo es legal a nivel de módulo; esto invita a reservar `Dim` para variables locales y declarar variables de módulo con `Private` , especialmente con la palabra clave `Public` contrastante que debería usarse de todos modos para declarar un miembro público. Alternativamente, use `Dim` *todas partes* , lo que importa es la *consistencia* :

"Campos privados"

- **Use el** `Private` para declarar una variable de nivel de módulo.
- **Use** `Dim` para declarar una variable local.
- **NO** use `Dim` para declarar una variable de nivel de módulo.

"Oscuro en todas partes"

- **Use** `Dim` para declarar cualquier cosa privada / local.

- **NO** use `Private` para declarar una variable de nivel de módulo.
- **EVITA** declarar campos `Public` . *

* En general, se debe evitar declarar campos `Public` o `Global` todos modos.

Campos

Una variable declarada a nivel de módulo, en la *sección de declaraciones* en la parte superior del cuerpo del módulo, es un *campo* . Un campo `Public` declarado en un *módulo estándar* es una *variable global* :

```
Public PublicField As Long
```

Se puede acceder a una variable con un alcance global desde cualquier lugar, incluidos otros proyectos de VBA que hagan referencia al proyecto en el que se ha declarado.

Para hacer una variable global / pública, pero solo visible desde el proyecto, use el modificador de `Friend` :

```
Friend FriendField As Long
```

Esto es especialmente útil en los complementos, donde la intención es que otros proyectos de VBA hagan referencia al proyecto del complemento y puedan consumir la API pública.

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

Los campos de amigos no están disponibles en módulos estándar.

Campos de instancia

Una variable declarada a nivel de módulo, en la *sección de declaraciones* en la parte superior del cuerpo de un módulo de clase (incluyendo `ThisWorkbook` , `ThisDocument` , `Worksheet` , `UserForm` y *módulos de clase*), es un *campo de instancia* : solo existe mientras exista una *instancia* de la clase alrededor

```
'> 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
```

Campos de encapsulacion

Los datos de instancia a menudo se mantienen `Private` , y doblados *encapsulados* . Un campo privado puede ser expuesto usando un procedimiento de `Property` . Para exponer públicamente una variable privada sin otorgar acceso de escritura al llamante, un módulo de clase (o un módulo estándar) implementa un miembro de `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 clase en sí puede modificar el valor encapsulado, pero el código de llamada solo puede acceder a los miembros `Public` (y miembros de `Friend` , si la persona que llama está en el mismo proyecto).

Para permitir que la persona que llama modifique:

- Un **valor** encapsulado, un módulo expone un miembro `Property Let` .
- Una **referencia de objeto** encapsulada, un módulo expone un miembro del `Property Set` .

Constantes (const)

Si tiene un valor que nunca cambia en su aplicación, puede definir una constante con nombre y usarla en lugar de un valor literal.

Puede utilizar Const solo a nivel de módulo o procedimiento. Esto significa que el contexto de declaración para una variable debe ser una clase, estructura, módulo, procedimiento o bloque, y no puede ser un archivo de origen, espacio de nombres o interfaz.

```
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
```

Si bien puede considerarse una buena práctica especificar tipos de constantes, no es estrictamente necesario. Si no especifica el tipo, se obtendrá el tipo correcto:

```
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
```

Tenga en cuenta que esto es específico de las Constantes y en contraste con las variables donde no se especifica el tipo de resultado en un tipo Variant.

Si bien es posible declarar explícitamente una constante como una cadena, no es posible declarar una constante como una cadena usando una sintaxis de cadena de ancho fijo

```
'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"
```

Modificadores de acceso

La declaración `Dim` debe estar reservada para las variables locales. A nivel de módulo, prefiera modificadores de acceso explícitos:

- `Private` para campos privados, al que solo se puede acceder dentro del módulo en el que están declarados.
- `Public` para campos públicos y variables globales, al que se puede acceder mediante cualquier código de llamada.
- `Friend` para variables públicas dentro del proyecto, pero inaccesible para otros proyectos VBA de referencia (relevantes para complementos)
- `Global` también se puede usar para campos `Public` en módulos estándar, pero es ilegal en módulos de clase y está obsoleto de todos modos, prefiera el modificador `Public` lugar. Este modificador tampoco es legal para los procedimientos.

Los modificadores de acceso son aplicables a variables y procedimientos por igual.

```
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
```

Opción Módulo Privado

Los procedimientos `Sub` parámetros públicos en módulos estándar se exponen como macros y se pueden adjuntar a controles y atajos de teclado en el documento host.

A la inversa, los procedimientos de `Function` públicas en módulos estándar se exponen como funciones definidas por el usuario (UDF) en la aplicación host.

La especificación del `Option Private Module` en la parte superior de un módulo estándar evita que sus miembros se vean expuestos como macros y UDF a la aplicación host.

Tipo de sugerencias

Las sugerencias de tipo están **muy** desanimadas. Existen y se documentan aquí por razones históricas y de compatibilidad con versiones anteriores. Debería usar la sintaxis `As [DataType]` lugar.

```
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
```

Las sugerencias de tipo disminuyen significativamente la legibilidad del código y fomentan una notación húngara heredada que *también* dificulta la legibilidad:

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

En su lugar, declare las variables más cerca de su uso y nombre las cosas por lo que usan, no por su tipo:

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

Las sugerencias de tipo también se pueden usar en literales, para imponer un tipo específico. De forma predeterminada, un literal numérico menor que 32,768 se interpretará como un literal `Integer` , pero con una sugerencia de tipo puede controlar que:

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

Las sugerencias de tipo generalmente no son necesarias en los literales, ya que se asignarían a una variable declarada con un tipo explícito, o se convertirían implícitamente al tipo apropiado cuando se pasaran como parámetros. Las conversiones implícitas se pueden evitar usando una de las funciones explícitas de conversión de tipos:

```
'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)
```

Funciones incorporadas de retorno de cadena

La mayoría de las funciones integradas que manejan cadenas vienen en dos versiones: una versión escrita de forma holgada que devuelve una `Variant` y una versión fuertemente tipada (que termina con `$`) que devuelve una `String` . A menos que esté asignando el valor de retorno a una `Variant` , debe preferir la versión que devuelve una `String` ; de lo contrario, hay una conversión implícita del valor de retorno.

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

Estas funciones son:

- 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 $

Tenga en cuenta que estos son *alias de* funciones, no son exactamente *sugerencias de tipo* . La función `Left` corresponde a la función oculta `B_Var_Left` , mientras que la versión `Left$` corresponde a la función oculta `B_Str_Left` .

En versiones muy tempranas de VBA, el signo `$` no es un carácter permitido y el nombre de la función tenía que estar entre corchetes. En Word Basic, había muchas, muchas más funciones que devolvían cadenas que terminaban en $.

Declarar cadenas de longitud fija

En VBA, las cadenas se pueden declarar con una longitud específica; se rellenan o truncan automáticamente para mantener esa longitud según lo declarado.

```
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
```

Cuándo usar una variable estática

Una variable estática declarada localmente no se destruye y no pierde su valor cuando se sale del procedimiento Sub. Las llamadas subsiguientes al procedimiento no requieren reinicialización o asignación, aunque es posible que desee "cero" cualquier valor (s) recordado (s).

Estos son particularmente útiles cuando se vincula tarde un objeto en un sub 'auxiliar' que se llama repetidamente.

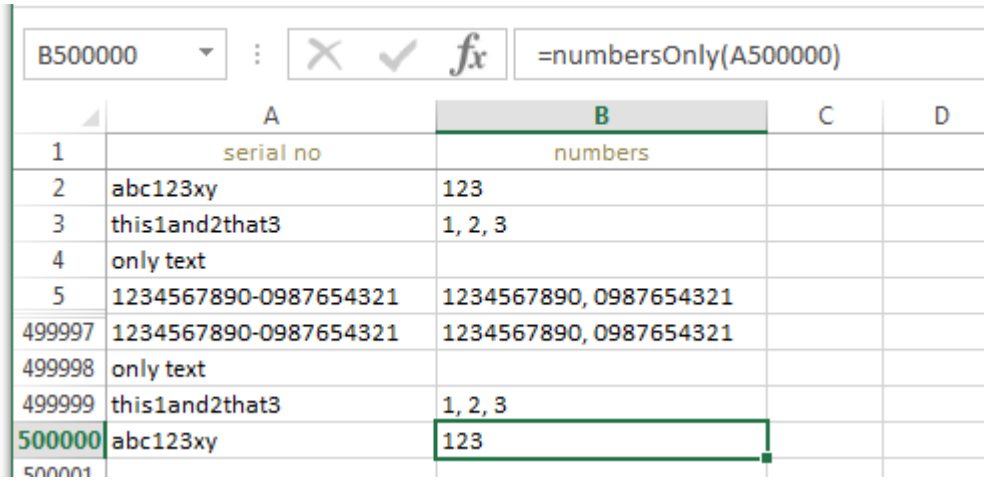
Fragmento 1: reutilizar un objeto Scripting.Dictionary en muchas hojas de trabajo

```
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
```

Fragmento 2: cree un UDF de hoja de cálculo que tarde enlaza el objeto 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
```



Ejemplo de UDF con objeto estático rellenado a través de medio millón de filas

* Tiempos transcurridos para llenar 500K filas con UDF:
- con **Dim rgx como objeto** : 148.74 segundos
- con **rgx estático como objeto** : 26.07 segundos

* Estos deben ser considerados para comparación relativa solamente. Sus propios resultados variarán según la complejidad y
Alcance de las operaciones realizadas.

Recuerde que un UDF no se calcula una vez en la vida útil de un libro de trabajo. Incluso un UDF no volátil se volverá a calcular siempre que los valores dentro del rango (s) a los que hace referencia estén sujetos a cambios. Cada evento de recálculo posterior solo aumenta los beneficios de una variable declarada estáticamente.

- Una variable estática está disponible durante la vida útil del módulo, no el procedimiento o la función en la que se declaró y asignó.
- Las variables estáticas solo pueden ser declaradas localmente.
- La variable estática contiene muchas de las mismas propiedades de una variable de nivel de módulo privado pero con un alcance más restringido.

Referencia relacionada: Estático (Visual Basic)

Lea Declarando variables en línea: https://riptutorial.com/es/vba/topic/877/declarando-variables

Capítulo 19: Declarar y asignar cadenas

Observaciones

Las cadenas son un tipo de referencia y son fundamentales para la mayoría de las tareas de programación. Se asigna texto a las cadenas, incluso si el texto resulta ser numérico. Las cadenas pueden ser de longitud cero, o cualquier longitud de hasta 2GB. Las versiones modernas de VBA almacenan cadenas internamente utilizando una matriz de bytes de bytes de conjuntos de caracteres de múltiples bytes (una alternativa a Unicode).

Examples

Declara una cadena constante

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

Declara una variable de cadena de ancho variable

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

Declara y asigna una cadena de ancho fijo

```
'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"
```

Declara y asigna una cadena de cadenas

```
'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"
```

Asigna caracteres específicos dentro de una cadena usando la instrucción Mid

VBA ofrece una función Mid para *devolver* subcadenas dentro de una cadena, pero también ofrece Mid *Statement* que se puede usar para asignar subcadenas o caracteres individuales dentro de una cadena.

La función `Mid` aparece normalmente en el lado derecho de una declaración de asignación o en una condición, pero la declaración `Mid` aparece normalmente en el lado izquierdo de una declaración de asignación.

```
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: Si necesita asignar *bytes* individuales en una cadena en lugar de *caracteres* individuales dentro de una cadena (consulte las observaciones a continuación sobre el conjunto de caracteres de múltiples bytes), se puede usar la declaración `MidB` . En este caso, el segundo argumento para la declaración `MidB` es la posición basada en 1 del byte donde se iniciará la sustitución, por lo que la línea equivalente al ejemplo anterior sería `MidB(surname, 5, 2) = "y"` .

Asignación ay desde una matriz de bytes

Las cadenas pueden asignarse directamente a matrices de bytes y viceversa. Recuerde que las cadenas se almacenan en un conjunto de caracteres de múltiples bytes (consulte las Notas a continuación), por lo que solo el resto del índice de la matriz resultante será la parte del carácter que se encuentra dentro del rango 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."
```

Lea Declarar y asignar cadenas en línea: https://riptutorial.com/es/vba/topic/3446/declarar-y-asignar-cadenas

Capítulo 20: Errores en tiempo de ejecución de VBA

Introducción

El código que se compila todavía puede tener errores, en tiempo de ejecución. Este tema enumera los más comunes, sus causas y cómo evitarlos.

Examples

Error en tiempo de ejecución '3': Devolución sin GoSub

Código incorrecto

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

¿Por qué no funciona esto?

La ejecución ingresa al procedimiento `DoSomething` , salta a la etiqueta `DoThis` , imprime "¡Hola!" a la salida de depuración, *vuelve* a las instrucciones inmediatamente después de la llamada `GoSub` , imprime "¡Hola!" de nuevo, y luego encuentra una declaración de `Return` , pero no hay a dónde *regresar* ahora, porque no llegamos aquí con una declaración de `GoSub` .

Código correcto

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

¿Por qué funciona esto?

Al introducir una instrucción `Exit Sub` *antes de* la `DoThis` línea `DoThis` , hemos separado la subrutina `DoThis` del resto del cuerpo del procedimiento: la única forma de ejecutar la subrutina `DoThis` es a través del salto `GoSub` .

Otras notas

`GoSub` / `Return` está en desuso, y debe evitarse en favor de las llamadas a procedimientos reales. Un procedimiento no debe contener subrutinas, excepto los controladores de errores.

Esto es muy similar al error en tiempo de ejecución '20': Reanudar sin error ; en ambas situaciones, la solución es garantizar que la *ruta de ejecución normal* no pueda ingresar a una subrutina (identificada por una etiqueta de línea) sin un salto explícito (asumiendo que `On Error GoTo` se considera un *salto explícito*).

Error en tiempo de ejecución '6': Desbordamiento

código incorrecto

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

¿Por qué no funciona esto?

El tipo de datos `Integer` es un entero con signo de 16 bits con un valor máximo de 32,767; asignarlo a algo más grande que eso *desbordará* el tipo y generará este error.

Código correcto

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

¿Por qué funciona esto?

Al usar un entero `Long` (32 bits) en su lugar, ahora podemos hacer un bucle que itere más de 32,767 veces sin desbordar el tipo de variable del contador.

Otras notas

Consulte Tipos de datos y límites para obtener más información.

Error en tiempo de ejecución '9': Subíndice fuera de rango

código incorrecto

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

¿Por qué no funciona esto?

`foo` es una matriz que contiene 10 elementos. Cuando el contador de bucle `i` alcanza un valor de 11, `foo(i)` está *fuera de rango* . Este error se produce siempre que se accede a una matriz o colección con un índice que no existe en esa matriz o colección.

Código correcto

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

¿Por qué funciona esto?

Use las funciones `LBound` y `UBound` para determinar los límites inferior y superior de una matriz, respectivamente.

Otras notas

Cuando el índice es una cadena, por ejemplo, `ThisWorkbook.Worksheets("I don't exist")` , este error significa que el nombre proporcionado no existe en la colección consultada.

Sin embargo, el error real es específico de la implementación; `Collection` generará el error de tiempo de ejecución 5 "Llamada o argumento de procedimiento no válido" en su lugar:

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

Error en tiempo de ejecución '13': No coincide el tipo

código incorrecto

```
Public Sub DoSomething()
    DoSomethingElse "42?"
End Sub

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

¿Por qué no funciona esto?

VBA está realmente tratando de convertir el `"42?"` argumento en un valor de `Date` . Cuando falla, la llamada a `DoSomethingElse` no se puede ejecutar, porque VBA no sabe qué fecha pasar, por lo que aumenta el *tipo* de error de tiempo de ejecución 13, porque el tipo de argumento no coincide con el tipo esperado (y puede Tampoco se convertirá implícitamente).

Código correcto

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

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

¿Por qué funciona esto?

Al pasar un argumento de `Date` a un procedimiento que espera un parámetro de `Date` , la llamada puede tener éxito.

Error en tiempo de ejecución '91': variable de objeto o variable de bloque no establecida

código incorrecto

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

¿Por qué no funciona esto?

Las variables de objeto contienen una *referencia* y las referencias deben *establecerse* con la

palabra clave `Set` . Este error se produce cuando se realiza una llamada de miembro en un objeto cuya referencia es `Nothing` . En este caso, `foo` es una referencia de `Collection` , pero no está inicializada, por lo que la referencia no contiene `Nothing` , y no podemos llamar a `.Add` en `Nothing` .

Código correcto

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

¿Por qué funciona esto?

Al asignar a la variable de objeto una referencia válida utilizando la palabra clave `Set` , las llamadas `.Add` se `.Add` correctamente.

Otras notas

A menudo, una función o propiedad puede devolver una referencia de objeto; un ejemplo común es el método `Range.Find` de Excel, que devuelve un objeto `Range` :

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

Sin embargo, la función puede devolver `Nothing` (si no se encuentra el término de búsqueda), por lo que es probable que la llamada al miembro `.Row` encadenado falle.

Antes de llamar a los miembros del objeto, verifique que la referencia esté establecida con una condición `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
```

Error en tiempo de ejecución '20': Reanudar sin error

código incorrecto

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

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

¿Por qué no funciona esto?

Si el procedimiento `DoSomethingElse` genera un error, la ejecución salta a la `CleanFail` línea `CleanFail` , imprime el número de error y la instrucción `Resume Next` vuelve a la instrucción que sigue inmediatamente a la línea donde ocurrió el error, que en este caso es `Debug.Print` instrucción: la subrutina de manejo de errores se ejecuta sin un contexto de error, y cuando se alcanza la instrucción `Resume Next` , se genera el error 20 en tiempo de ejecución porque no hay ningún lugar para reanudar.

Código correcto

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

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

¿Por qué funciona esto?

Al introducir una instrucción `Exit Sub` antes de la `CleanFail` línea `CleanFail` , hemos segregado la subrutina de manejo de errores `CleanFail` del resto del cuerpo del procedimiento: la única forma de ejecutar la subrutina de manejo de errores es a través de un salto `On Error` ; por lo tanto, ninguna ruta de ejecución llega a la instrucción `Resume` fuera de un contexto de error, lo que evita el error 20 en tiempo de ejecución.

Otras notas

Esto es muy similar al error en tiempo de ejecución '3': Devolución sin GoSub ; en ambas situaciones, la solución es garantizar que la *ruta de ejecución normal* no pueda ingresar a una subrutina (identificada por una etiqueta de línea) sin un salto explícito (asumiendo que `On Error GoTo` se considera un *salto explícito*).

Lea Errores en tiempo de ejecución de VBA en línea:
https://riptutorial.com/es/vba/topic/8917/errores-en-tiempo-de-ejecucion-de-vba

Capítulo 21: Estructuras de control de flujo

Examples

Seleccione el caso

`Select Case` puede usarse cuando son posibles muchas condiciones diferentes. Las condiciones se verifican de arriba a abajo y solo se ejecutará el primer caso que coincida.

```
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 lógica del bloque `Select Case` se puede invertir para admitir la prueba de diferentes variables, en este tipo de escenario también podemos usar operadores lógicos:

```
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
```

Las declaraciones de casos también pueden usar operadores aritméticos. Cuando se usa un operador aritmético contra el valor de `Select Case` , debe ir precedido por la palabra clave `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
```

Para cada bucle

La construcción de bucle `For Each` es ideal para iterar todos los elementos de una colección.

```
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
```

Utilice `For Each` para iterar colecciones de objetos:

```
Dim sheet As Worksheet
For Each sheet In ActiveWorkbook.Worksheets
    Debug.Print sheet.Name
Next
```

Evitar `For Each` al iterar matrices; un bucle `For` ofrecerá un rendimiento significativamente mejor con las matrices. Por el contrario, un bucle `For Each` ofrecerá un mejor rendimiento al iterar una `Collection` .

Sintaxis

```
For Each [item] In [collection]
    [statements]
Next [item]
```

La palabra clave `Next` puede ser seguida opcionalmente por la variable iterador; Esto puede ayudar a aclarar los bucles anidados, aunque hay mejores formas de aclarar el código anidado, como extraer el bucle interno en su propio procedimiento.

```
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
```

Hacer bucle

```
 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
```

Mientras bucle

```
'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
```

En bucle

El bucle `For` se usa para repetir la sección adjunta del código un número determinado de veces. El siguiente ejemplo simple ilustra la sintaxis básica:

```
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
```

El código anterior declara un entero `i` . El bucle `For` asigna cada valor entre 1 y 10 a `i` y luego ejecuta `Debug.Print i` , es decir, el código imprime los números del 1 al 10 en la ventana inmediata. Tenga en cuenta que la variable de bucle se incrementa con la `Next` instrucción, es decir, después de que se ejecuta el código adjunto y no antes de que se ejecute.

De forma predeterminada, el contador se incrementará en 1 cada vez que se ejecute el bucle. Sin embargo, se puede especificar un `Step` para cambiar la cantidad del incremento como un valor literal o de retorno de una función. Si el valor inicial, el valor final o el valor de `Step` es un número de punto flotante, se redondeará al valor entero más cercano. `Step` puede ser un valor 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
```

En general, un bucle `For` se usaría en situaciones en las que se sabe antes de que el bucle comience cuántas veces se ejecuta el código adjunto (de lo contrario, un bucle `Do` o `While` puede ser más apropiado). Esto se debe a que la condición de salida se corrige después de la primera

entrada en el bucle, ya que este código demuestra:

```
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 bucle `For` se puede salir antes con la instrucción `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
```

Lea Estructuras de control de flujo en línea: https://riptutorial.com/es/vba/topic/1873/estructuras-de-control-de-flujo

Capítulo 22: Estructuras de datos

Introducción

[TODO: Este tema debe ser un ejemplo de todas las estructuras de datos básicas de CS 101 junto con una explicación como una descripción general de cómo se pueden implementar las estructuras de datos en VBA. Esta sería una buena oportunidad para vincular y reforzar los conceptos introducidos en los temas relacionados con la Clase en la documentación de VBA.]

Examples

Lista enlazada

Este ejemplo de lista enlazada implementa operaciones de tipos de datos abstractos establecidos .

Clase **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
```

Clase **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
```

Árbol binario

Este es un ejemplo de un árbol de búsqueda binario desequilibrado. Un árbol binario se estructura conceptualmente como una jerarquía de nodos que descienden hacia abajo desde una raíz común, donde cada nodo tiene dos hijos: izquierdo y derecho. Por ejemplo, supongamos que los números 7, 5, 9, 3, 11, 6, 12, 14 y 15 se insertaron en un BinaryTree. La estructura sería la siguiente. Tenga en cuenta que este árbol binario no está equilibrado , lo que puede ser una característica deseable para garantizar el rendimiento de las búsquedas; consulte los árboles AVL para ver un ejemplo de un árbol de búsqueda binaria con equilibrio automático.

```
            7
           / \
          5   9
         / \   \
        3   6   11
                  \
                   12
                     \
                      14
```

```
                        \
                         15
```

Clase **BinaryTreeNode**

```
Option Explicit

Public left As BinaryTreeNode
Public right As BinaryTreeNode
Public key As Variant
Public value As Variant
```

Clase **binario**

[QUE HACER]

Lea Estructuras de datos en línea: https://riptutorial.com/es/vba/topic/8628/estructuras-de-datos

Capítulo 23: Eventos

Sintaxis

- **Módulo de origen** : `[Public] Event [identifier]([argument_list])`
- **Módulo de controlador** : `Dim|Private|Public WithEvents [identifier] As [type]`

Observaciones

- Un evento solo puede ser `Public` . El modificador es opcional porque los miembros del módulo de clase (incluidos los eventos) son implícitamente `Public` de forma predeterminada.
- Una variable `WithEvents` puede ser `Private` o `Public` , pero no `Friend` . El modificador es obligatorio porque `WithEvents` no es una palabra clave que declara una variable, sino una palabra clave modificadora que forma parte de la sintaxis de declaración de la variable. Por lo tanto, la palabra clave `Dim` debe usarse si no está presente un modificador de acceso.

Examples

Fuentes y manejadores

¿Qué son los eventos?

VBA se basa en *eventos* : el código VBA se ejecuta en respuesta a los eventos generados por la aplicación host o el documento host. La comprensión de los eventos es fundamental para comprender VBA.

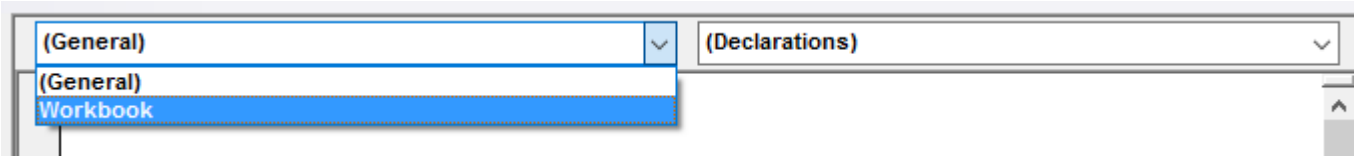
Las API a menudo exponen objetos que generan una serie de *eventos* en respuesta a varios estados. Por ejemplo, un objeto `Excel.Application` genera un evento cada vez que se crea, abre, activa o cierra un nuevo libro de trabajo. O cuando se calcula una hoja de cálculo. O justo antes de guardar un archivo. O inmediatamente después. Un botón en un formulario genera un evento `Click` cuando el usuario hace clic en él, el formulario del usuario genera un evento justo después de que se activa y otro justo antes de que se cierre.

Desde la perspectiva de la API, los eventos son *puntos de extensión* : el código del cliente puede elegir implementar un código que *maneje* estos eventos y ejecutar código personalizado cada vez que se activen: así es como puede ejecutar su código personalizado automáticamente cada vez que la selección cambia en cualquier hoja de trabajo - manejando el evento que se activa cuando la selección cambia en cualquier hoja de cálculo.

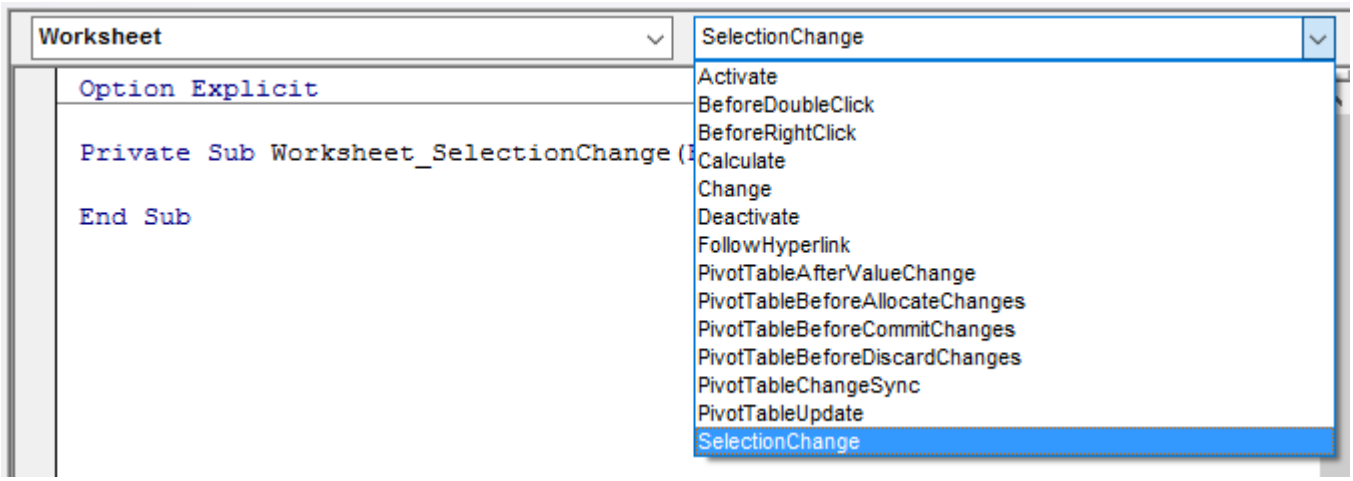
Un objeto que expone eventos es un *origen de eventos* . Un método que maneja un evento es un *manejador* .

Manipuladores

Los módulos de documentos VBA (por ejemplo, `ThisDocument` , `ThisWorkbook` , `Sheet1` , etc.) y los módulos `UserForm` son *módulos de clase* que *implementan* interfaces especiales que exponen una serie de *eventos* . Puede navegar por estas interfaces en el menú desplegable del lado izquierdo en la parte superior del panel de código:



El menú desplegable del lado derecho muestra los miembros de la interfaz seleccionada en el menú desplegable del lado izquierdo:



El VBE genera automáticamente un apéndice de controlador de eventos cuando se selecciona un elemento en la lista del lado derecho, o navega allí si existe el controlador.

Puede definir una variable `WithEvents` ámbito de módulo en cualquier módulo:

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

Cada declaración `WithEvents` está disponible para seleccionar desde el menú desplegable del lado izquierdo. Cuando se selecciona un evento en el menú desplegable del lado derecho, el VBE genera un apéndice de controlador de eventos con el nombre del objeto `WithEvents` y el nombre del evento, unido con un guión bajo:

```
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 los tipos que exponen al menos un evento pueden usarse con `WithEvents` , y las declaraciones `WithEvents` no pueden ser asignadas una referencia en el momento con la palabra clave `New` . Este código es ilegal:

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

La referencia del objeto se debe `Set` explícitamente; en un módulo de clase, un buen lugar para hacerlo es a menudo en el controlador `Class_Initialize` , porque entonces la clase maneja los eventos de ese objeto mientras exista su instancia.

Fuentes

Cualquier módulo de clase (o módulo de documento o formulario de usuario) puede ser un origen de evento. Use la palabra clave `Event` para definir la *firma* del evento, en la *sección* de *declaraciones* del módulo:

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

La firma del evento determina cómo se genera el evento y cómo se verán los controladores de eventos.

Los eventos solo se pueden *generar* dentro de la clase en la que están definidos, el código del cliente solo puede *manejarlos* . Los eventos se `RaiseEvent` con la palabra clave `RaiseEvent` ; Los argumentos del evento se proporcionan en ese punto:

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

Sin el código que maneja el evento `SomethingHappened` , la ejecución del procedimiento `DoSomething` aún `DoSomething` el evento, pero no ocurrirá nada. Suponiendo que el origen del evento es el código anterior en una clase llamada `Something` , este código en `ThisWorkbook` mostraría un cuadro de mensaje que dice "hola" cada vez que se realiza una `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
```

Usando parámetros pasados por referencia

Un evento puede definir un parámetro `ByRef` destinado a ser devuelto al llamante:

```
Public Event BeforeSomething(ByRef cancel As Boolean)
Public Event AfterSomething()

Public Sub DoSomething()
    Dim cancel As Boolean
    RaiseEvent BeforeSomething(cancel)
    If cancel Then Exit Sub

    'todo: actually do something

    RaiseEvent AfterSomething
End Sub
```

Si el evento `BeforeSomething` tiene un controlador que establece su parámetro de `cancel` en `True` , entonces, cuando la ejecución vuelva del controlador, la `cancel` será `True` y `AfterSomething` nunca se generará.

```
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
```

Suponiendo que la referencia de objeto `foo` se asigna en algún lugar, cuando `foo.DoSomething` ejecuta, un cuadro de mensaje le `foo.DoSomething` si desea cancelar, y un segundo cuadro de mensaje dice "no cancelar" solo cuando se seleccionó `No`.

Utilizando objetos mutables

También puede pasar una copia de un objeto mutable `ByVal` y dejar que los controladores modifiquen las propiedades de ese objeto; la persona que llama puede leer los valores de propiedad modificados y actuar en consecuencia.

```
'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
```

Combinado con el tipo `Variant` , esto se puede usar para crear formas no obvias de devolver un valor a la persona que llama:

```
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
```

El manejador se vería así:

```
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
```

Lea Eventos en línea: https://riptutorial.com/es/vba/topic/5278/eventos

Capítulo 24: Formularios de usuario

Examples

Mejores prácticas

Un `UserForm` es un módulo de clase con un diseñador y una **instancia predeterminada** . Se puede acceder al *diseñador* presionando `Shift` + `F7` mientras se ve el *código subyacente* , y se puede acceder al *código subyacente* presionando la tecla `F7` mientras se ve el *diseñador* .

Trabaja con una nueva instancia cada vez.

Al ser un *módulo de clase* , un formulario es, por lo tanto, un *modelo* para un *objeto* . Debido a que un formulario puede contener el estado y los datos, es una mejor práctica trabajar con una nueva *instancia* de la clase, en lugar de con la predeterminada / global:

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

En lugar de:

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

Trabajar con la instancia predeterminada puede provocar errores sutiles cuando el formulario se cierra con el botón rojo "X" y / o cuando se utiliza `Unload Me` en el código subyacente.

Implementar la lógica en otro lugar.

Un formulario no debe ocuparse de nada más que de la *presentación* : un botón El controlador de `Click` que se conecta a una base de datos y ejecuta una consulta parametrizada basada en la entrada del usuario, está **haciendo muchas cosas** .

En su lugar, implemente la *lógica aplicativa* en el código que es responsable de mostrar el formulario, o incluso mejor, en módulos y procedimientos dedicados.

Escriba el código de tal manera que UserForm solo sea responsable de saber cómo mostrar y recopilar datos: de dónde provienen los datos, o qué sucede con los datos después, no es motivo de preocupación.

La persona que llama no debe ser molestada con los controles.

Cree un *modelo* bien definido para el formulario con el que trabajar, ya sea en su propio módulo de clase dedicado o encapsulado dentro del propio código subyacente del formulario: exponga el *modelo* con procedimientos de `Property Get` y haga que el código del cliente trabaje con estos: esto hace que la forma una *abstracción* sobre los controles y sus detalles esenciales, exponiendo solo los datos relevantes al código del cliente.

Esto significa código que se ve así:

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

En lugar de esto:

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

Manejar el evento QueryClose.

Por lo general, los formularios tienen un botón `Cerrar` , y las indicaciones / diálogos tienen los botones `Aceptar` y `Cancelar` ; el usuario puede cerrar el formulario utilizando el *cuadro de control* del formulario (el botón rojo "X"), que destruye la instancia del formulario de forma predeterminada (otra buena razón para *trabajar con una nueva instancia cada vez*).

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

La forma más sencilla de controlar el evento `QueryClose` es establecer el parámetro `Cancel` en `True` y luego *ocultar* el formulario en lugar de *cerrarlo* :

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

De esa manera, el botón "X" nunca destruirá la instancia, y la persona que llama puede acceder de forma segura a todos los miembros públicos.

Ocultar, no cerrar.

El código que crea un objeto debe ser responsable de destruirlo: no es responsabilidad del formulario descargarse y terminarse.

Evite usar `Unload Me` en el código subyacente de un formulario. Call `Me.Hide` en `Me.Hide` lugar, para que el código de llamada aún pueda usar el objeto que creó cuando se cierra el formulario.

Nombre las cosas.

Use la ventana de herramientas de *propiedades* (`F4`) para nombrar cuidadosamente cada control en un formulario. El nombre de un control se usa en el código subyacente, por lo que, a menos que esté usando una herramienta de refactorización que pueda manejar esto, **cambiar el nombre de un control romperá el código** , por lo que es mucho más fácil hacer las cosas bien en primer lugar, que intentarlo para descifrar exactamente cuál de los 20 cuadros de texto significa `TextBox12` .

Tradicionalmente, los controles de UserForm se nombran con prefijos de estilo húngaro:

- `lblUserName` para un control `Label` que indica un nombre de usuario.
- `txtUserName` para un control `TextBox` donde el usuario puede ingresar un nombre de usuario.
- `cboUserName` para un control `ComboBox` donde el usuario puede ingresar o elegir un nombre de usuario.
- `lstUserName` para un control `ListBox` donde el usuario puede elegir un nombre de usuario.
- `btnOk` o `cmdOk` para un control de `Button` etiqueta "Ok".

El problema es que cuando, por ejemplo, la IU se rediseña y un `ComboBox` cambia a un `ListBox` , el nombre debe cambiar para reflejar el nuevo tipo de control: es mejor nombrar los controles por lo que representan, en lugar de después de su tipo de control, para *desacoplarlos.* Código de la interfaz de usuario tanto como sea posible.

- `UserNameLabel` para una etiqueta de solo lectura que indica un nombre de usuario.
- `UserNameInput` para un control donde el usuario puede ingresar o elegir un nombre de usuario.
- `OkButton` para un botón de comando etiquetado "Ok".

Cualquiera que sea el estilo elegido, cualquier cosa es mejor que dejar que todos los controles tengan sus nombres predeterminados. La consistencia en el estilo de nombre también es ideal.

Manejo de consultas cerrar

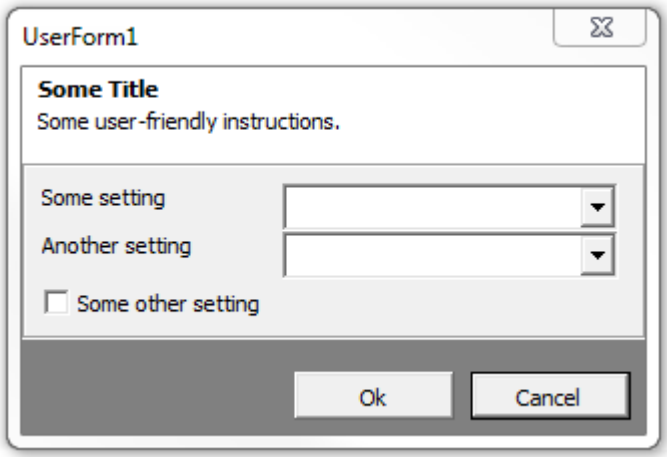
El evento `QueryClose` se `QueryClose` cada vez que se cierra un formulario, ya sea a través de la acción del usuario o mediante programación. El parámetro `CloseMode` contiene un valor de enumeración `VbQueryClose` que indica cómo se cerró el formulario:

Constante	Descripción	Valor
`vbFormControlMenu`	El formulario se está cerrando en respuesta a la acción del usuario.	0
`vbFormCode`	El formulario se está cerrando en respuesta a una declaración de `Unload`	1
`vbAppWindows`	Sesión de Windows está terminando	2
`vbAppTaskManager`	El Administrador de tareas de Windows está cerrando la aplicación host	3
`vbFormMDIForm`	No soportado en VBA	4

Para una mejor legibilidad, es mejor usar estas constantes en lugar de usar su valor directamente.

Un formulario de usuario cancelable

Dada una forma con un botón `Cancelar`



El código subyacente del formulario podría verse así:

```
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
```

El código de llamada podría mostrar el formulario y saber si fue cancelado:

```
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 propiedad `IsCancelled` devuelve `True` cuando se hace clic en el botón `Cancelar` , o cuando el usuario cierra el formulario utilizando el *cuadro de control* .

Lea Formularios de usuario en línea: https://riptutorial.com/es/vba/topic/5351/formularios-de-usuario

Capítulo 25: Interfaces

Introducción

Una **interfaz** es una forma de definir un conjunto de comportamientos que realizará una clase. La definición de una interfaz es una lista de firmas de métodos (nombre, parámetros y tipo de retorno). Una clase que tiene todos los métodos se dice que "implementa" esa interfaz.

En VBA, el uso de interfaces le permite al compilador verificar que un módulo implemente todos sus métodos. Una variable o parámetro se puede definir en términos de una interfaz en lugar de una clase específica.

Examples

Interfaz simple - Flyable

La interfaz `Flyable` es un módulo de clase con el siguiente código:

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

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

Un módulo de clase, `Airplane` , usa la palabra clave `Implements` para decirle al compilador que genere un error a menos que tenga dos métodos: un sub `Flyable_Fly()` y una función `Flyable_GetAltitude()` que devuelva un `Long` .

```
Implements Flyable

Public Sub Flyable_Fly()
    Debug.Print "Flying With Jet Engines!"
End Sub

Public Function Flyable_GetAltitude() As Long
    Flyable_GetAltitude = 10000
End Function
```

Un módulo de segunda clase, `Duck` , también implementa `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
```

Podemos escribir una rutina que acepte cualquier valor de `Flyable` , sabiendo que responderá a un comando de `Fly` o `GetAltitude` :

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

Debido a que la interfaz está definida, la ventana emergente de IntelliSense mostrará `Fly` y `GetAltitude` for `F`

Cuando ejecutamos el siguiente código:

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

FlyAndCheckAltitude MyDuck
FlyAndCheckAltitude MyAirplane
```

La salida es:

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

Tenga en cuenta que aunque la subrutina se llama `Flyable_Fly` tanto en `Airplane` como en `Duck` , se puede llamar como `Fly` cuando la variable o el parámetro se definen como `Flyable` . Si la variable se define específicamente como un `Duck` , debería llamarse como `Flyable_Fly` .

Múltiples interfaces en una clase - Volable y Swimable

Usando el ejemplo de `Flyable` como punto de partida, podemos agregar una segunda interfaz, `Swimmable` , con el siguiente código:

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

El objeto `Duck` puede `Implement` tanto el vuelo como la natación:

```
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 clase de `Fish` puede implementar `Swimmable` , también:

```
Implements Swimmable

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

Ahora, podemos ver que el objeto `Duck` se puede pasar a un Sub como un `Flyable` por un lado, y un `Swimmable` por el otro:

```
Sub InterfaceTest()

Dim MyDuck As New Duck
Dim MyAirplane As New Airplane
Dim MyFish As New Fish

Debug.Print "Fly Check..."

FlyAndCheckAltitude MyDuck
FlyAndCheckAltitude MyAirplane

Debug.Print "Swim Check..."

TrySwimming MyDuck
TrySwimming MyFish

End Sub

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

Public Sub TrySwimming(S As Swimmable)
    S.Swim
End Sub
```

La salida de este código es:

> Cheque de vuelo ...
>
> Volando Con Alas!
>
> 30
>
> Volando con motores a reacción!

10000

Cheque de natacion ...

Flotando en el agua

Nadando bajo el agua

Lea Interfaces en línea: https://riptutorial.com/es/vba/topic/8784/interfaces

Capítulo 26: Lectura de 2GB + archivos en binario en VBA y File Hashes

Introducción

Existe una forma fácil de leer archivos en binario dentro de VBA, sin embargo, tiene una restricción de 2GB (2,147,483,647 bytes - máx del tipo de datos Long). A medida que la tecnología evoluciona, este límite de 2 GB se viola fácilmente. Por ejemplo, una imagen ISO del sistema operativo instala un disco DVD. Microsoft proporciona una manera de superar esto mediante la API de Windows de bajo nivel y aquí hay una copia de seguridad de la misma.

También demuestre (lea la parte) para calcular Hashes de archivos sin un programa externo como `fciv.exe` de Microsoft.

Observaciones

MÉTODOS PARA LA CLASE POR MICROSOFT

Nombre del método	Descripción
Esta abierto	Devuelve un valor booleano para indicar si el archivo está abierto.
OpenFile (*sFileNam* e As String)	Abre el archivo especificado por el argumento sFileName.
Cerrar el archivo	Cierra el archivo actualmente abierto.
ReadBytes (*ByteCount* como largo)	Lee bytes ByteCount y los devuelve en una matriz de bytes Variant y mueve el puntero.
WriteBytes (*DataBytes ()* como byte)	Escribe el contenido de la matriz de bytes en la posición actual en el archivo y mueve el puntero.
Rubor	Obliga a Windows a vaciar el caché de escritura.
SeekAbsolute (*HighPos* Long, *LowPos* Long)	Mueve el puntero del archivo a la posición designada desde el principio del archivo. Aunque VBA trata a los DWORDS como valores firmados, la API los trata como no firmados. Haga que el argumento de orden superior sea distinto de cero para superar los 4 GB. El DWORD de orden inferior será negativo para valores entre 2 GB y 4

Nombre del método	Descripción
	GB.
SeekRelative (*Offset* As Long)	Mueve el puntero del archivo hasta +/- 2GB desde la ubicación actual. Puede volver a escribir este método para permitir desplazamientos mayores de 2 GB convirtiendo un desplazamiento firmado de 64 bits en dos valores de 32 bits.

PROPIEDADES DE LA CLASE POR MICROSOFT.

Propiedad	Descripción
FileHandle	El identificador de archivo para el archivo abierto actualmente. Esto no es compatible con los manejadores de archivos VBA.
Nombre del archivo	El nombre del archivo abierto actualmente.
AutoFlush	Establece / indica si WriteBytes llamará automáticamente al método Flush.

Modulo normal

Función	Notas
GetFileHash (*sFile* As String, *uBlockSize* As Double, *sHashType* As String)	Simplemente agregue la ruta completa que se va a hash, Blocksize to use (número de bytes) y el tipo de Hash que se usará: una de las constantes privadas: **HashTypeMD5** , **HashTypeSHA1** , **HashTypeSHA256** , **HashTypeSHA384** , **HashTypeSHA512** . Esto fue diseñado para ser lo más genérico posible.

Debe **des** / comentar el **uFileSize como doble en** consecuencia. He probado MD5 y SHA1.

Examples

Esto tiene que estar en un módulo de clase, ejemplos más adelante referidos como "Aleatorio"

```
' 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
```

Código para calcular el hash de archivo en un módulo estándar

```
Private Const HashTypeMD5 As String = "MD5" ' https://msdn.microsoft.com/en-
us/library/system.security.cryptography.md5cryptoserviceprovider(v=vs.110).aspx
Private Const HashTypeSHA1 As String = "SHA1" ' https://msdn.microsoft.com/en-
us/library/system.security.cryptography.sha1cryptoserviceprovider(v=vs.110).aspx
Private Const HashTypeSHA256 As String = "SHA256" ' https://msdn.microsoft.com/en-
us/library/system.security.cryptography.sha256cryptoserviceprovider(v=vs.110).aspx
Private Const HashTypeSHA384 As String = "SHA384" ' https://msdn.microsoft.com/en-
us/library/system.security.cryptography.sha384cryptoserviceprovider(v=vs.110).aspx
Private Const HashTypeSHA512 As String = "SHA512" ' https://msdn.microsoft.com/en-
us/library/system.security.cryptography.sha512cryptoserviceprovider(v=vs.110).aspx

Private uFileSize As Double ' Comment out if not testing performance by FileHashes()

Sub FileHashes()
    Dim tStart As Date, tFinish As Date, sHash As String, aTestFiles As Variant, oTestFile As
Variant, aBlockSizes As Variant, oBlockSize As Variant
    Dim BLOCKSIZE As Double

    ' This performs performance testing on different file sizes and block sizes
    aBlockSizes = Array("2^12-1", "2^13-1", "2^14-1", "2^15-1", "2^16-1", "2^17-1", "2^18-1",
"2^19-1", "2^20-1", "2^21-1", "2^22-1", "2^23-1", "2^24-1", "2^25-1", "2^26-1")
    aTestFiles = Array("C:\ISO\clonezilla-live-2.2.2-37-amd64.iso",
"C:\ISO\HPIP201.2014_0902.29.iso",
"C:\ISO\SW_DVD5_Windows_Vista_Business_W32_32BIT_English.ISO",
"C:\ISO\Win10_1607_English_x64.iso",
"C:\ISO\SW_DVD9_Windows_Svr_Std_and_DataCtr_2012_R2_64Bit_English.ISO")
    Debug.Print "Test files: " & Join(aTestFiles, " | ")
    Debug.Print "BlockSizes: " & Join(aBlockSizes, " | ")
    For Each oTestFile In aTestFiles
        Debug.Print oTestFile
        For Each oBlockSize In aBlockSizes
```

```
            BLOCKSIZE = Evaluate(oBlockSize)
            tStart = Now
            sHash = GetFileHash(CStr(oTestFile), BLOCKSIZE, HashTypeMD5)
            tFinish = Now
            Debug.Print sHash, uFileSize, Format(tFinish - tStart, "hh:mm:ss"), oBlockSize & "
(" & BLOCKSIZE & ")"
        Next
    Next
End Sub

Private Function GetFileHash(ByVal sFile As String, ByVal uBlockSize As Double, ByVal
sHashType As String) As String
    Dim oFSO As Object ' "Scripting.FileSystemObject"
    Dim oCSP As Object ' One of the "CryptoServiceProvider"
    Dim oRnd As Random ' "Random" Class by Microsoft, must be in the same file
    Dim uBytesRead As Double, uBytesToRead As Double, bDone As Boolean
    Dim aBlock() As Byte, aBytes As Variant ' Arrays to store bytes
    Dim aHash() As Byte, sHash As String, i As Long
    'Dim uFileSize As Double ' Un-Comment if GetFileHash() is to be used individually

    Set oRnd = New Random ' Class by Microsoft: Random
    Set oFSO = CreateObject("Scripting.FileSystemObject")
    Set oCSP = CreateObject("System.Security.Cryptography." & sHashType &
"CryptoServiceProvider")

    If oFSO Is Nothing Or oRnd Is Nothing Or oCSP Is Nothing Then
        MsgBox "One or more required objects cannot be created"
        GoTo CleanUp
    End If

    uFileSize = oFSO.GetFile(sFile).Size ' FILELEN() has 2GB max!
    uBytesRead = 0
    bDone = False
    sHash = String(oCSP.HashSize / 4, "0") ' Each hexadecimal has 4 bits

    Application.ScreenUpdating = False
    ' Process the file in chunks of uBlockSize or less
    If uFileSize = 0 Then
        ReDim aBlock(0)
        oCSP.TransformFinalBlock aBlock, 0, 0
        bDone = True
    Else
        With oRnd
            .OpenFile sFile
            Do
                If uBytesRead + uBlockSize < uFileSize Then
                    uBytesToRead = uBlockSize
                Else
                    uBytesToRead = uFileSize - uBytesRead
                    bDone = True
                End If
                ' Read in some bytes
                aBytes = .ReadBytes(uBytesToRead)
                aBlock = aBytes
                If bDone Then
                    oCSP.TransformFinalBlock aBlock, 0, uBytesToRead
                    uBytesRead = uBytesRead + uBytesToRead
                Else
                    uBytesRead = uBytesRead + oCSP.TransformBlock(aBlock, 0, uBytesToRead,
aBlock, 0)
                End If
```

```
                DoEvents
            Loop Until bDone
            .CloseFile
        End With
    End If
    If bDone Then
        ' convert Hash byte array to an hexadecimal string
        aHash = oCSP.hash
        For i = 0 To UBound(aHash)
            Mid$(sHash, i * 2 + (aHash(i) > 15) + 2) = Hex(aHash(i))
        Next
    End If
    Application.ScreenUpdating = True
    ' Clean up
    oCSP.Clear
CleanUp:
    Set oFSO = Nothing
    Set oRnd = Nothing
    Set oCSP = Nothing
    GetFileHash = sHash
End Function
```

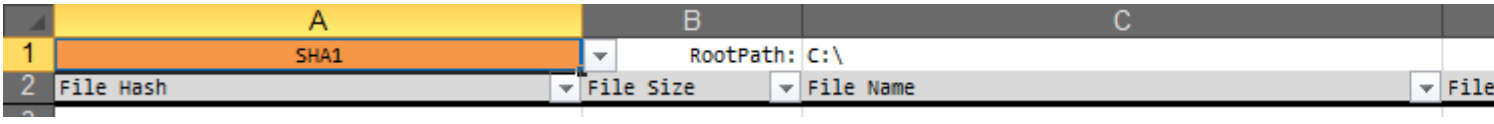
La salida es bastante interesante, mis archivos de prueba indican que **`BLOCKSIZE = 131071` (2 ^ 17-1)** ofrece el mejor rendimiento general con 32bit Office 2010 en Windows 7 x64, la mejor *opción* es *2 ^ 16-1 (65535)* . Nota `2^27-1` produce *memoria* `2^27-1` .

Tamaño del archivo (bytes)	Nombre del archivo
146,800,640	clonezilla-live-2.2.2-37-amd64.iso
798,210,048	HPIP201.2014_0902.29.iso
2,073,016,320	SW_DVD5_Windows_Vista_Business_W32_32BIT_English.ISO
4.380.387.328	Win10_1607_English_x64.iso
5,400,115,200	SW_DVD9_Windows_Svr_Std_and_DataCtr_2012_R2_64Bit_English.ISO

Cálculo de todos los archivos hash de una carpeta raíz

Otra variación del código anterior le brinda más rendimiento cuando desea obtener códigos hash de todos los archivos de una carpeta raíz, incluidas todas las subcarpetas.

Ejemplo de hoja de trabajo:



Código

```
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
```

Lea Lectura de 2GB + archivos en binario en VBA y File Hashes en línea: https://riptutorial.com/es/vba/topic/8786/lectura-de-2gb-plus-archivos-en-binario-en-vba-y-file-hashes

Capítulo 27: Literales de cadenas - Escape, caracteres no imprimibles y continuaciones de línea

Observaciones

La asignación de literales de cadena en VBA está limitada por las limitaciones del IDE y la página de códigos de la configuración de idioma del usuario actual. Los ejemplos anteriores demuestran los casos especiales de cadenas escapadas, cadenas especiales no imprimibles y cadenas largas literales.

Al asignar literales de cadena que contienen caracteres que son específicos de una determinada página de códigos, es posible que deba tener en cuenta los problemas de internacionalización asignando una cadena de un archivo de recursos Unicode separado.

Examples

Escapando al "personaje

La sintaxis de VBA requiere que aparezca una cadena literal dentro de `"` marcas `"` , de modo que cuando su cadena deba *contener* comillas, deberá escapar / anteponer el `"` carácter con un extra `"` para que VBA entienda que desea que `""` sea interpretado como una `"` cadena.

```
'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"
```

Asignando literales de cadena larga.

El editor de VBA solo permite 1023 caracteres por línea, pero generalmente solo los primeros 100-150 caracteres son visibles sin desplazamiento. Si necesita asignar literales de cadena larga, pero desea mantener su código legible, deberá usar continuaciones de línea y concatenación para asignar su cadena.

```
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 le permitirá usar un número limitado de continuaciones de línea (el número real varía según la longitud de cada línea dentro del bloque continuado), por lo que si tiene cadenas muy largas, deberá asignar y reasignar con concatenación .

```
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.
```

Usando constantes de cadena VBA

VBA define una serie de constantes de cadena para caracteres especiales como:

- vbCr: Carriage-Return 'Igual que "\ r" en los idiomas de estilo C.
- vbLf: Line-Feed 'Igual que "\ n" en los lenguajes de estilo C
- vbCrLf: Carriage-Return & Line-Feed (una nueva línea en Windows)
- vbTab: carácter de tabulación
- vbNullString: una cadena vacía, como ""

Puede usar estas constantes con concatenación y otras funciones de cadena para construir cadenas literales con caracteres especiales.

```
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
```

Usar `vbNullString` se considera una mejor práctica que el valor equivalente de `""` debido a las diferencias en la forma en que se compila el código. Se accede a las cadenas a través de un puntero a un área de memoria asignada, y el compilador VBA es lo suficientemente inteligente como para usar un puntero nulo para representar `vbNullString` . El literal `""` se asigna a la memoria como si fuera una variante de tipo String, lo que hace que el uso de la constante sea

mucho más eficiente:

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

Lea Literales de cadenas - Escape, caracteres no imprimibles y continuaciones de línea en línea: https://riptutorial.com/es/vba/topic/3445/literales-de-cadenas---escape--caracteres-no-imprimibles-y-continuaciones-de-linea

Capítulo 28: Llamadas API

Introducción

API significa interfaz de programación de aplicaciones

Las API para VBA implican un conjunto de métodos que permiten la interacción directa con el sistema operativo

Se pueden hacer llamadas al sistema ejecutando procedimientos definidos en archivos DLL

Observaciones

Archivos comunes de la biblioteca del entorno operativo (DLL):

Biblioteca de enlace dinámico	Descripción
Advapi32.dll	Biblioteca de servicios avanzados para API que incluye muchas llamadas de seguridad y de registro
Comdlg32.dll	Biblioteca de API de diálogo común
Gdi32.dll	Biblioteca de API de interfaz de dispositivo gráfico
Kernel32.dll	Soporte básico de API de base de Windows de 32 bits
Lz32.dll	Rutinas de compresión de 32 bits
Mpr.dll	Biblioteca de enrutador de múltiples proveedores
Netapi32.dll	Biblioteca de API de red de 32 bits
Shell32.dll	Biblioteca de la API de Shell de 32 bits
User32.dll	Biblioteca para rutinas de interfaz de usuario.
Version.dll	Biblioteca de versiones
Winmm.dll	Biblioteca multimedia de Windows
Winspool.drv	Interfaz de cola de impresión que contiene las llamadas a la API de cola de impresión

Nuevos argumentos utilizados para el sistema 64:

Tipo	ít	Descripción
Índice	PtrSafe	Indica que la declaración Declare es compatible con 64 bits. Este atributo es obligatorio en sistemas de 64 bits.
Tipo de datos	LongPtr	Un tipo de datos variable que es un tipo de datos de 4 bytes en las versiones de 32 bits y un tipo de datos de 8 bytes en las versiones de 64 bits de Office 2010. Esta es la forma recomendada de declarar un puntero o un identificador para el nuevo código, pero también para el código heredado si tiene que ejecutarse en la versión de 64 bits de Office 2010. Solo se admite en el tiempo de ejecución de VBA 7 en 32 bits y 64 bits. Tenga en cuenta que puede asignarle valores numéricos pero no tipos numéricos
Tipo de datos	Largo largo	Este es un tipo de datos de 8 bytes que está disponible solo en las versiones de 64 bits de Office 2010. Puede asignar valores numéricos pero no tipos numéricos (para evitar el truncamiento)
Conversión	Operador	CLngPtr Convierte una expresión simple a un tipo de datos LongPtr
Conversión	Operador	CLngLng Convierte una expresión simple a un tipo de datos LongLong
Función	VarPtr	Convertidor de variantes. Devuelve un LongPtr en versiones de 64 bits y un largo en 32 bits (4 bytes)
Función	ObjPtr	Convertidor de objetos. Devuelve un LongPtr en versiones de 64 bits y un largo en 32 bits (4 bytes)
Función	StrPtr	Convertidor de cuerdas. Devuelve un LongPtr en versiones de 64 bits y un largo en 32 bits (4 bytes)

Referencia completa de firmas de llamadas:

- Win32api32.txt para Visual Basic 5.0 (antiguas declaraciones de API, revisado por última vez en marzo de 2005, Microsoft)
- Win32API_PtrSafe con soporte de 64 bits (Office 2010, Microsoft)

Examples

Declaración y uso de la API

Declarar un procedimiento DLL para trabajar con diferentes versiones de VBA:

```
Option Explicit
```

```
#If Win64 Then

    Private Declare PtrSafe Sub xLib "Kernel32" Alias "Sleep" (ByVal dwMilliseconds As Long)

#ElseIf Win32 Then

    Private Declare Sub apiSleep Lib "Kernel32" Alias "Sleep" (ByVal dwMilliseconds As Long)

#End If
```

La declaración anterior le dice a VBA cómo llamar a la función "Sleep" definida en el archivo Kernel32.dll

Win64 y Win32 son constantes predefinidas usadas para la compilación condicional

---

Constantes predefinidas

Algunas constantes de compilación ya están predefinidas. Las que existan dependerán del grado de bit de la versión de Office en la que esté ejecutando VBA. Tenga en cuenta que Vba7 se introdujo junto con Office 2010 para admitir versiones de Office de 64 bits.

Constante	16 bits	32 bits	64 bits
Vba6	Falso	Si vba6	Falso
Vba7	Falso	Si vba7	Cierto
Win16	Cierto	Falso	Falso
Win32	Falso	Cierto	Cierto
Win64	Falso	Falso	Cierto
Mac	Falso	Si mac	Si mac

Estas constantes se refieren a la versión de Office, no a la versión de Windows. Por ejemplo, Win32 = TRUE en Office de 32 bits, incluso si el sistema operativo es una versión de Windows de 64 bits.

La diferencia principal al declarar las API es entre las versiones de Office de 32 y 64 bits que introdujeron nuevos tipos de parámetros (consulte la sección de Comentarios para más detalles)

Notas:

- Las declaraciones se colocan en la parte superior del módulo, y fuera de cualquier Subs o Functions
- Los procedimientos declarados en módulos estándar son públicos por defecto.
- Para declarar un procedimiento privado a un módulo preceda a la declaración con la palabra clave `Private`

- Los procedimientos de DLL declarados en cualquier otro tipo de módulo son privados para ese módulo

---

Ejemplo simple para la llamada a la API de suspensión:

```
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
```

Se recomienda crear un módulo API dedicado para proporcionar un fácil acceso a las funciones del sistema desde los envoltorios de VBA: las subscripciones o funciones normales de VBA que encapsulan los detalles necesarios para la llamada real del sistema, como los parámetros utilizados en las bibliotecas y la inicialización de esos parámetros.

El módulo puede contener todas las declaraciones y dependencias:

- Método de firmas y estructuras de datos requeridas.
- Envoltorios que realizan la validación de entrada y aseguran que todos los parámetros se pasan como se espera

---

Para declarar un procedimiento de DLL, agregue una declaración `Declare` a la sección Declaraciones de la ventana de código.

Si el procedimiento devuelve un valor, declararlo como una **función** :

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

Si un procedimiento no devuelve un valor, declararlo como un **Sub** :

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

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

  También es de destacar que la **mayoría de las llamadas no válidas a las API colapsarán Excel** y posiblemente dañarán los archivos de datos

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

Office 2011 para 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
```

Los siguientes ejemplos (API de Windows - Módulo dedicado (1 y 2)) muestran un módulo de API que incluye declaraciones comunes para Win64 y Win32

API de Windows - Módulo dedicado (1 de 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
```

(continúa en el segundo ejemplo)

API de Windows - Módulo dedicado (2 de 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 de Mac

Microsoft no admite oficialmente las API, pero con algunas investigaciones se pueden encontrar más declaraciones en línea

Office 2016 para Mac es un espacio aislado

A diferencia de otras versiones de aplicaciones de Office que admiten VBA, las aplicaciones de Office 2016 para Mac están en un espacio aislado.

El sandboxing restringe que las aplicaciones accedan a recursos fuera del contenedor de la aplicación. Esto afecta a los complementos o macros que involucran el acceso a archivos o la comunicación a través de procesos. Puede minimizar los efectos del sandboxing utilizando los nuevos comandos que se describen en la siguiente sección. Nuevos comandos VBA para Office 2016 para Mac

Los siguientes comandos de VBA son nuevos y exclusivos de Office 2016 para Mac.

Mando	Utilizar para
GrantAccessToMultipleFiles	Solicite el permiso de un usuario para acceder a varios archivos a la vez
Tarea de AppleScript	Llame a los scripts externos de AppleScript desde VB
MAC_OFFICE_VERSION	IFDEF entre diferentes versiones de Mac Office en tiempo de compilación

---

Office 2011 para 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 para 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
```

Consigue monitores totales y resolución de pantalla.

```
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 y APIs regionales

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
```

Lea Llamadas API en línea: https://riptutorial.com/es/vba/topic/10569/llamadas-api

Capítulo 29: Los operadores

Observaciones

Los operadores son evaluados en el siguiente orden:

- Operadores matematicos
- Operadores bitwise
- Operadores de concatenacion
- Operadores de comparación
- Operadores logicos

Los operadores con prioridad coincidente se evalúan de izquierda a derecha. El orden predeterminado se puede anular utilizando paréntesis `(` y `)` para agrupar expresiones.

Examples

Operadores matematicos

Listado por orden de precedencia:

Simbólico	Nombre	Descripción
`^`	Exposiciónción	Devuelva el resultado de elevar el operando de la izquierda a la potencia del operando de la derecha. Tenga en cuenta que el valor devuelto por exponenciación es *siempre* `Double` , independientemente de los tipos de valor que se dividan. Cualquier coacción del resultado en un tipo de variable tiene lugar ***después de que*** se realiza el cálculo.
`/`	División [1]	Devuelve el resultado de dividir el operando de la izquierda por el operando de la derecha. Tenga en cuenta que el valor devuelto por división *siempre* es `Double` , independientemente de los tipos de valor que se dividan. Cualquier coacción del resultado en un tipo de variable tiene lugar ***después de que*** se realiza el cálculo.
`*`	Multiplicación [1]	Devuelve el producto de 2 operandos.
`\`	División entera	Devuelve el resultado entero de dividir el operando de la izquierda por el operando de la derecha ***después de*** redondear ambos lados con .5 redondeando hacia abajo. Cualquier resto de la división se ignora. Si el operando de la derecha (el divisor) es `0` , se producirá un error en tiempo de ejecución 11: División por cero. Tenga en cuenta que esto

Simbólico	Nombre	Descripción
		ocurre ***después de que*** se realiza todo el redondeo: expresiones como `3 \ 0.4` también darán como resultado un error de división por cero.
`Mod`	Modulo	Devuelve el resto entero de dividir el operando de la izquierda por el operando de la derecha. El operando en cada lado se redondea a un entero *antes de* la división, con .5 redondeando hacia abajo. Por ejemplo, tanto `8.6 Mod 3` como `12 Mod 2.6` dan como resultado `0` . Si el operando de la derecha (el divisor) es `0` , se producirá un error en tiempo de ejecución 11: División por cero. Tenga en cuenta que esto ocurre ***después de que*** se realiza todo el redondeo: expresiones como `3 Mod 0.4` también darán como resultado un error de división por cero.
`-`	Resta [2]	Devuelve el resultado de restar el operando de la derecha del operando de la izquierda.
`+`	Adición [2]	Devuelve la suma de 2 operandos. Tenga en cuenta que este token también se trata como un operador de concatenación cuando se aplica a una `String` . Ver **Operadores de Concatenación** .

[1] Se considera que la multiplicación y la división tienen la misma precedencia.

[2] La suma y la resta se tratan con la misma prioridad.

Operadores de Concatenacion

VBA es compatible con 2 operadores de concatenación diferentes, `+` y `&` y ambos realizan la misma función cuando se usan con los tipos de `String` : el `String` mano derecha se agrega al final del `String` de la izquierda.

Si el operador `&` se usa con un tipo de variable que no sea una `String` , se convierte implícitamente en una `String` antes de concatenar.

Tenga en cuenta que el operador de `+` concatenación es una sobrecarga del operador de suma `+` . El comportamiento de `+` está determinado por los tipos de variables de los operandos y la precedencia de los tipos de operadores. Si ambos operandos se escriben como una `String` o `Variant` con un subtipo de `String` , se concatenan:

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

    left = "5"
    right = "5"

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

```
End Sub
```

Si *cualquiera de los* lados es un tipo numérico y el otro lado es una `String` que se puede convertir en un número, la prioridad de tipo de los operadores matemáticos hace que el operador sea tratado como el operador de suma y se agreguen los valores numéricos:

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

    left = 5
    right = "5"

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

Este comportamiento puede dar lugar a errores sutiles y difíciles de depurar, especialmente si se utilizan los tipos `Variant` , por lo que normalmente solo se debe usar el operador `&` para la concatenación.

Operadores de comparación

Simbólico	Nombre	Descripción
`=`	Igual a	Devuelve `True` si los operandos de la izquierda y la derecha son iguales. Tenga en cuenta que esta es una sobrecarga del operador de asignación.
`<>`	No igual a	Devuelve `True` si los operandos de la izquierda y la derecha no son iguales.
`>`	Mas grande que	Devuelve `True` si el operando de la izquierda es mayor que el de la derecha.
`<`	Menos que	Devuelve `True` si el operando de la izquierda es menor que el de la derecha.
`>=`	Mayor que o igual	Devuelve `True` si el operando de la izquierda es mayor o igual que el operando de la derecha.
`<=`	Menor o igual	Devuelve `True` si el operando de la izquierda es menor o igual que el operando de la derecha.
`Is`	Equidad de referencia	Devuelve `True` si la referencia del objeto de la izquierda es la misma instancia que la referencia del objeto de la derecha. También se puede utilizar con `Nothing` (la referencia de objeto nulo) en cualquier lado. **Nota:** el operador `Is` intentará forzar ambos operandos en un `Object` antes de realizar la comparación. Si cualquiera de los lados es un tipo primitivo *o* una `Variant` que no

Simbólico	Nombre	Descripción
		contiene un objeto (ya sea un subtipo sin objeto o `vtEmpty`), la comparación dará como resultado un error 424 en tiempo de ejecución: "Objeto requerido". Si cualquiera de los operandos pertenece a una *interfaz* diferente del mismo objeto, la comparación devolverá `True` . Si necesita probar la equidad tanto de la instancia *como de* la interfaz, utilice `ObjPtr(left) = ObjPtr(right)` lugar.

Notas

La sintaxis VBA permite "cadenas" de operadores de comparación, pero estas construcciones generalmente deben evitarse. Las comparaciones siempre se realizan de izquierda a derecha en solo 2 operandos a la vez, y cada comparación da como resultado un `Boolean` . Por ejemplo, la expresión ...

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

... se puede leer en algunos contextos como una prueba de si `b` está entre `a` y `c` . En VBA, esto se evalúa de la siguiente manera:

```
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
```

Cualquier operador de comparación que no `Is` use con un `Object` como operando se realizará en el valor de retorno del miembro predeterminado del `Object` . Si el objeto no tiene un miembro predeterminado, la comparación dará como resultado un error 438 en tiempo de ejecución: "El objeto no admite su propiedad o método".

Si el `Object` está inicializado, la comparación resultará en un error de tiempo de ejecución 91 - "Variable de objeto o Con variable de bloque no establecida".

Si se usa el literal `Nothing` con cualquier operador de comparación que no sea `Is` , se producirá un error de compilación: "Uso no válido del objeto".

Si el miembro predeterminado del `Object` es *otro* *`Object`* , VBA llamará continuamente al miembro predeterminado de cada valor de retorno sucesivo hasta que se devuelva un tipo primitivo o se genere un error. Por ejemplo, suponga que `SomeClass` tiene un miembro predeterminado de `Value` , que es una instancia de `ChildClass` con un miembro predeterminado de `ChildValue` . La comparación...

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

... serán evaluados como:

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

Si cualquiera de los operandos es de tipo numérico y el *otro* es una `String` o `Variant` de subtipo `String` , se realizará una comparación numérica. En este caso, si la `String` no se puede convertir a un número, se generará un error de tiempo de ejecución 13 - "Falta de coincidencia de tipo" en la comparación.

Si ***ambos*** operandos son una `String` o `Variant` de subtipo `String` , se realizará una comparación de cadena en función de la configuración de Opción Comparación del módulo de código. Estas comparaciones se realizan en una base de carácter por carácter. Tenga en cuenta que la *representación* de *caracteres* de una `String` contiene un número ***no*** es ***lo*** mismo que una comparación de los valores numéricos:

```
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
```

Por este motivo, asegúrese de que las variables `String` o `Variant` se conviertan en números antes de realizar comparaciones numéricas de inequidad en ellos.

Si un operando es una `Date` , se realizará una comparación numérica en el valor Doble subyacente si el otro operando es numérico o se puede convertir a un tipo numérico.

Si el otro operando es una `String` o `Variant` de subtipo `String` que se puede convertir en una `Date` utilizando la configuración regional actual, la `String` se convertirá en una `Date` . Si no se puede convertir a una `Date` en la configuración regional actual, se generará un error de tiempo de ejecución 13 - "No coincide el tipo" en la comparación.

Se debe tener cuidado al hacer comparaciones entre valores `Double` o `Single` y valores booleanos . A diferencia de otros tipos numéricos, no se puede suponer que los valores que no son cero sean `True` debido al comportamiento de VBA de promover el tipo de datos de una comparación que implique un número de punto flotante 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
```

Bitwise \ Operadores lógicos

Todos los operadores lógicos en VBA se pueden considerar como "anulaciones" de los operadores a nivel de bits del mismo nombre. Técnicamente, *siempre se* tratan como operadores bitwise. Todos los operadores de comparación en VBA devuelven un valor booleano , que siempre tendrá ninguno de sus bits establecidos (`False`) o *todos* sus bits establecidos (`True`). Pero tratará un valor con *cualquier* bit establecido como `True` . Esto significa que el resultado de convertir el resultado a nivel de bits de una expresión a un operador `Boolean` (ver Operadores de comparación) siempre será el mismo que tratarlo como una expresión lógica.

Asignar el resultado de una expresión usando uno de estos operadores dará el resultado a nivel de bits. Tenga en cuenta que en las tablas de verdad a continuación, `0` es equivalente a `False` y `1` es equivalente a `True` .

`And`

Devuelve `True` si las expresiones de ambos lados se evalúan como `True` .

Operando de mano izquierda	Operando de mano derecha	Resultado
0	0	0
0	1	0
1	0	0
1	1	1

`Or`

Devuelve `True` si cualquiera de los lados de la expresión se evalúa como `True` .

Operando de mano izquierda	Operando de mano derecha	Resultado
0	0	0
0	1	1
1	0	1
1	1	1

`Not`

Devuelve `True` si la expresión se evalúa como `False` y `False` si las evaluaciones de expresión son `True` .

Operando de mano derecha	Resultado
0	1
1	0

`Not` es el único operando sin un operando de la mano izquierda. El Editor de Visual Basic simplificará automáticamente las expresiones con un argumento de la mano izquierda. Si escribe ...

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

... el VBE cambiará la línea a:

```
Debug.Print Not x
```

Se realizarán simplificaciones similares a cualquier expresión que contenga un operando de la izquierda (incluidas las expresiones) para `Not`

`Xor`

También conocido como "exclusivo o". Devuelve `True` si ambas expresiones evalúan resultados diferentes.

Operando de mano izquierda	Operando de mano derecha	Resultado
0	0	0
0	1	1
1	0	1
1	1	0

Tenga en cuenta que aunque el operador `Xor` se puede *usar* como un operador lógico, no hay absolutamente ninguna razón para hacerlo, ya que da el mismo resultado que el operador de comparación `<>` .

`Eqv`

También conocido como "equivalencia". Devuelve `True` cuando ambas expresiones evalúan al mismo resultado.

Operando de mano izquierda	Operando de mano derecha	Resultado
0	0	1
0	1	0
1	0	0
1	1	1

Tenga en cuenta que la función `Eqv` se usa *muy* raramente, ya que `x Eqv y` es equivalente a `Not (x Xor y)` mucho más legible.

`Imp`

También conocido como "implicación". Devuelve `True` si ambos operandos son iguales *o si* el segundo es `True` .

Operando de mano izquierda	Operando de mano derecha	Resultado
0	0	1
0	1	1
1	0	0
1	1	1

Tenga en cuenta que la función `Imp` es muy rara vez utilizada. Una buena regla general es que si no puede explicar lo que significa, debe usar otra construcción.

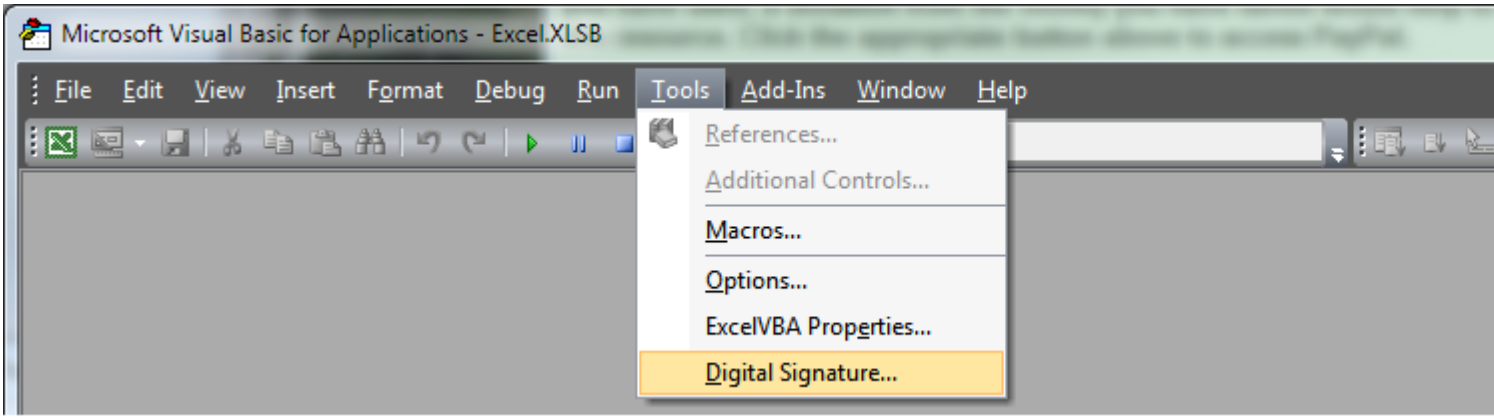
Lea Los operadores en línea: https://riptutorial.com/es/vba/topic/5813/los-operadores

Capítulo 30: Macro seguridad y firma de proyectos / módulos VBA.

Examples

Crear un certificado autofirmado digital válido SELFCERT.EXE

Para ejecutar macros y mantener la seguridad que proporcionan las aplicaciones de Office contra el código malicioso, es necesario firmar digitalmente el VBAProject.OTM desde el *editor de VBA> Herramientas> Firma digital* .

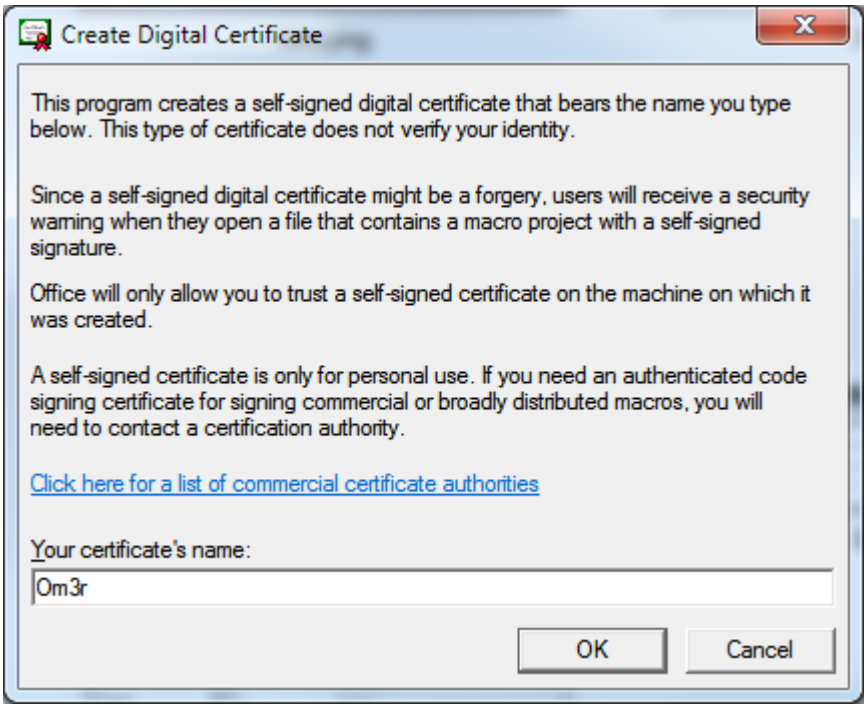


Office viene con una utilidad para crear un certificado digital autofirmado que puede emplear en la PC para firmar sus proyectos.

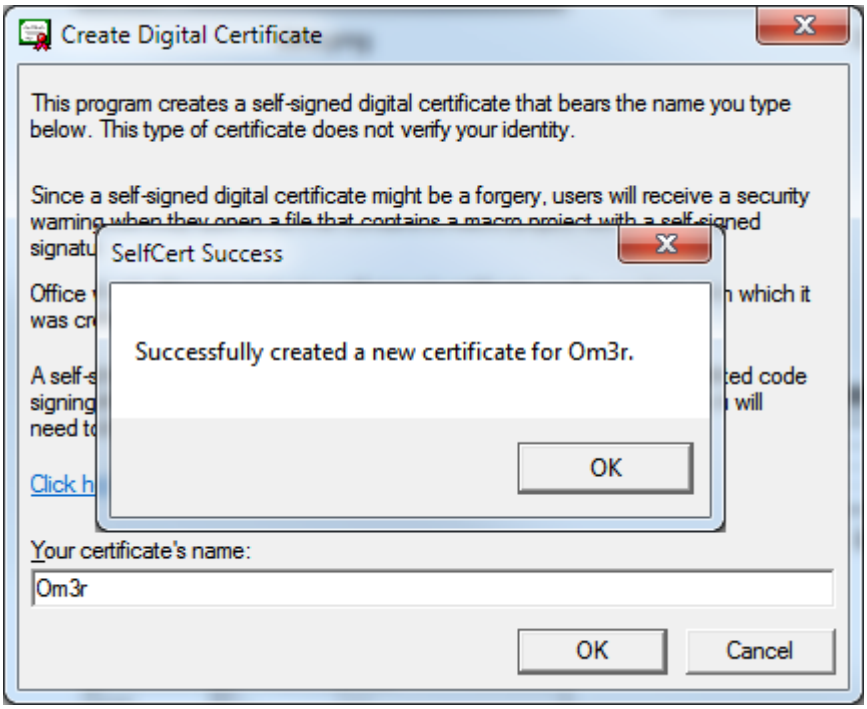
Esta utilidad **SELFCERT.EXE** está en la carpeta del programa de Office,

Haga clic en Certificado digital para proyectos VBA para abrir el *asistente de* certificados.

En el cuadro de diálogo, ingrese un nombre adecuado para el certificado y haga clic en Aceptar.

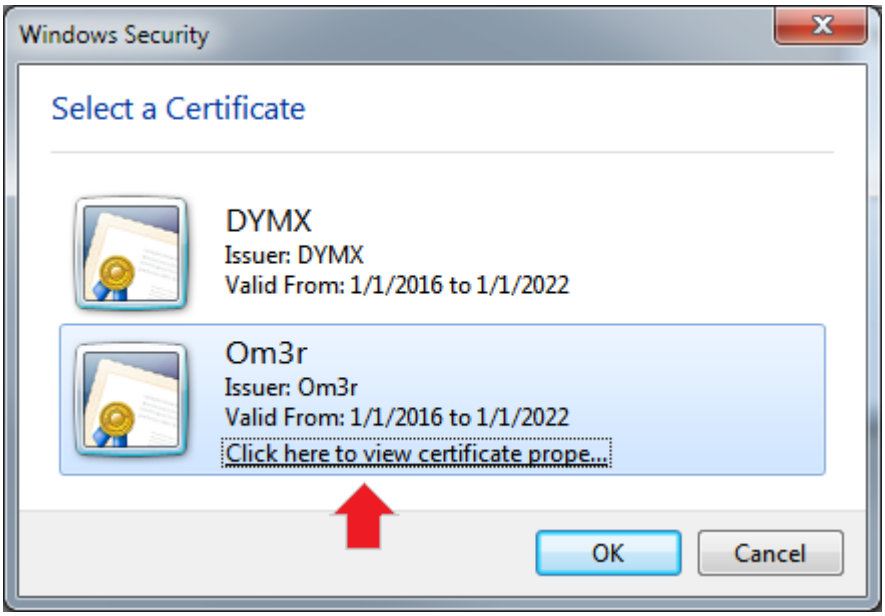


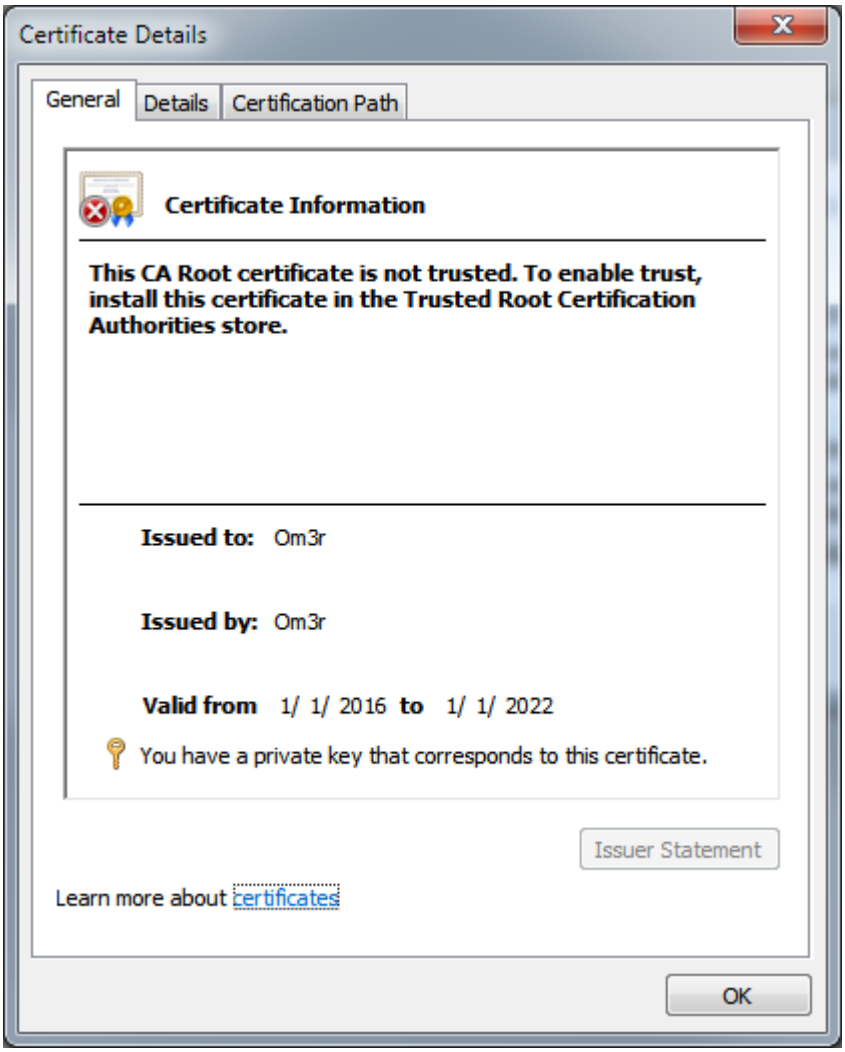
Si todo va bien verás una confirmación:



Ahora puede cerrar el asistente **SELFCERT** y dirigir su atención al certificado que ha creado.

Si intenta emplear el certificado que acaba de crear y verifica sus propiedades



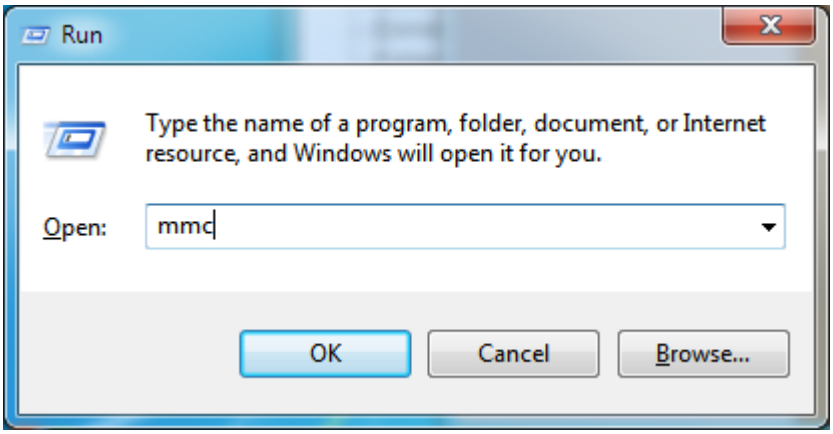


Verá que el certificado no es de confianza y la razón se indica en el cuadro de diálogo.

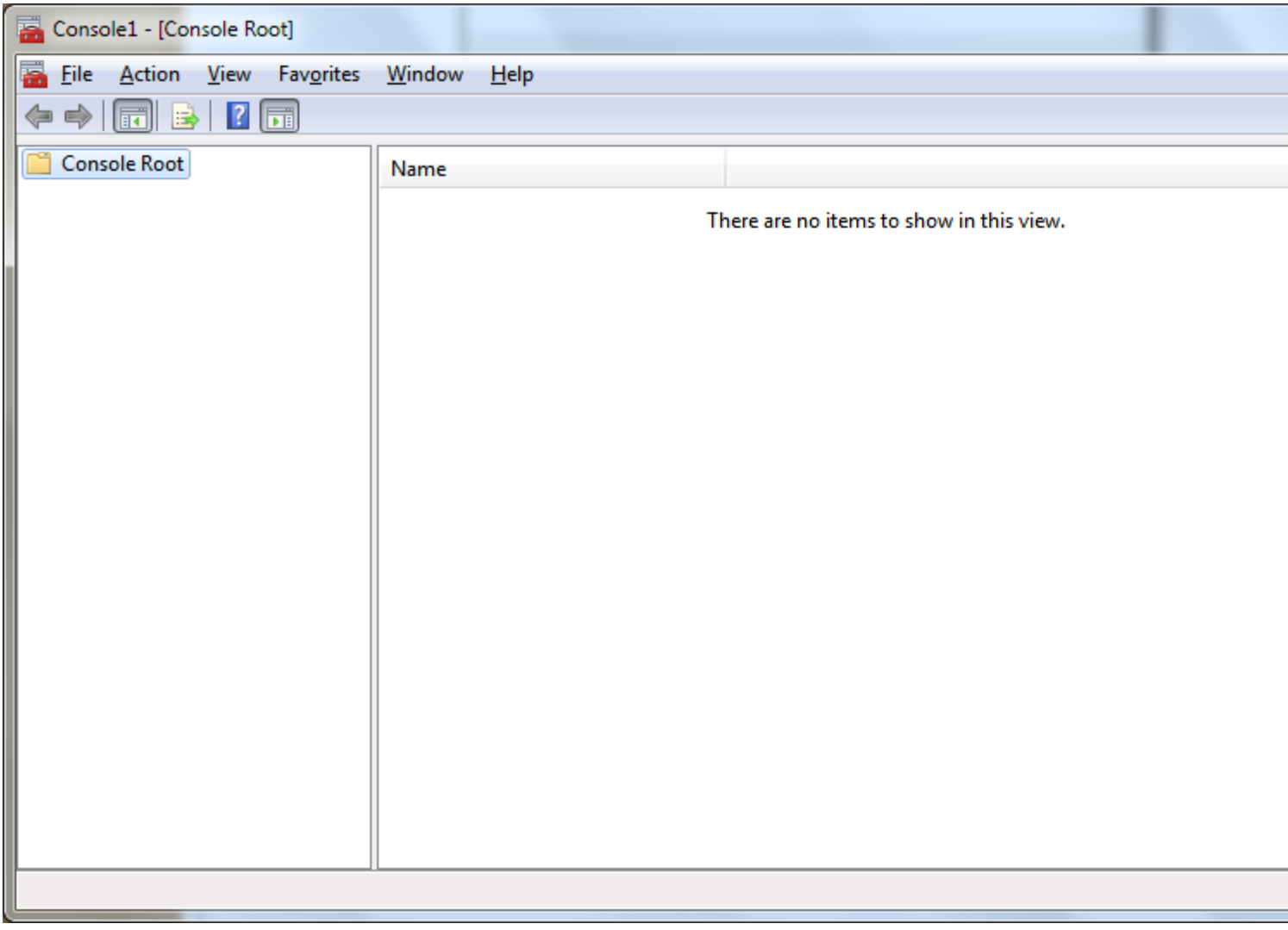
El certificado se ha creado en el almacén Usuario actual> Personal> Certificados. Debe ir a la tienda de Computadoras locales> Autoridades de certificación raíz de confianza> Certificados, por lo que debe exportar desde la primera e importar a la última.

Al presionar la tecla de Windows + R, se abrirá la ventana 'Ejecutar'. luego ingrese 'mmc' en la

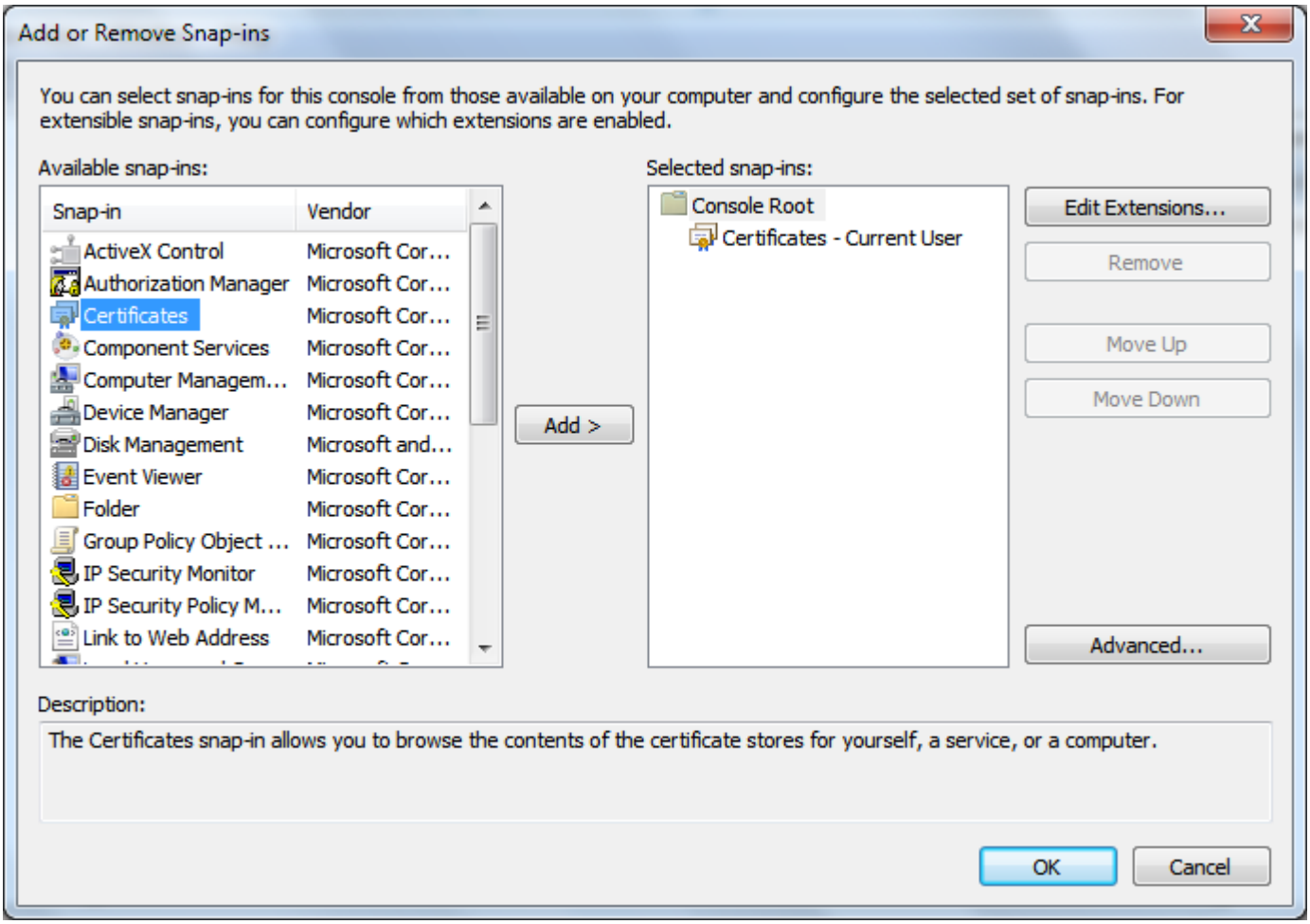
ventana como se muestra a continuación y haga clic en 'Aceptar'.



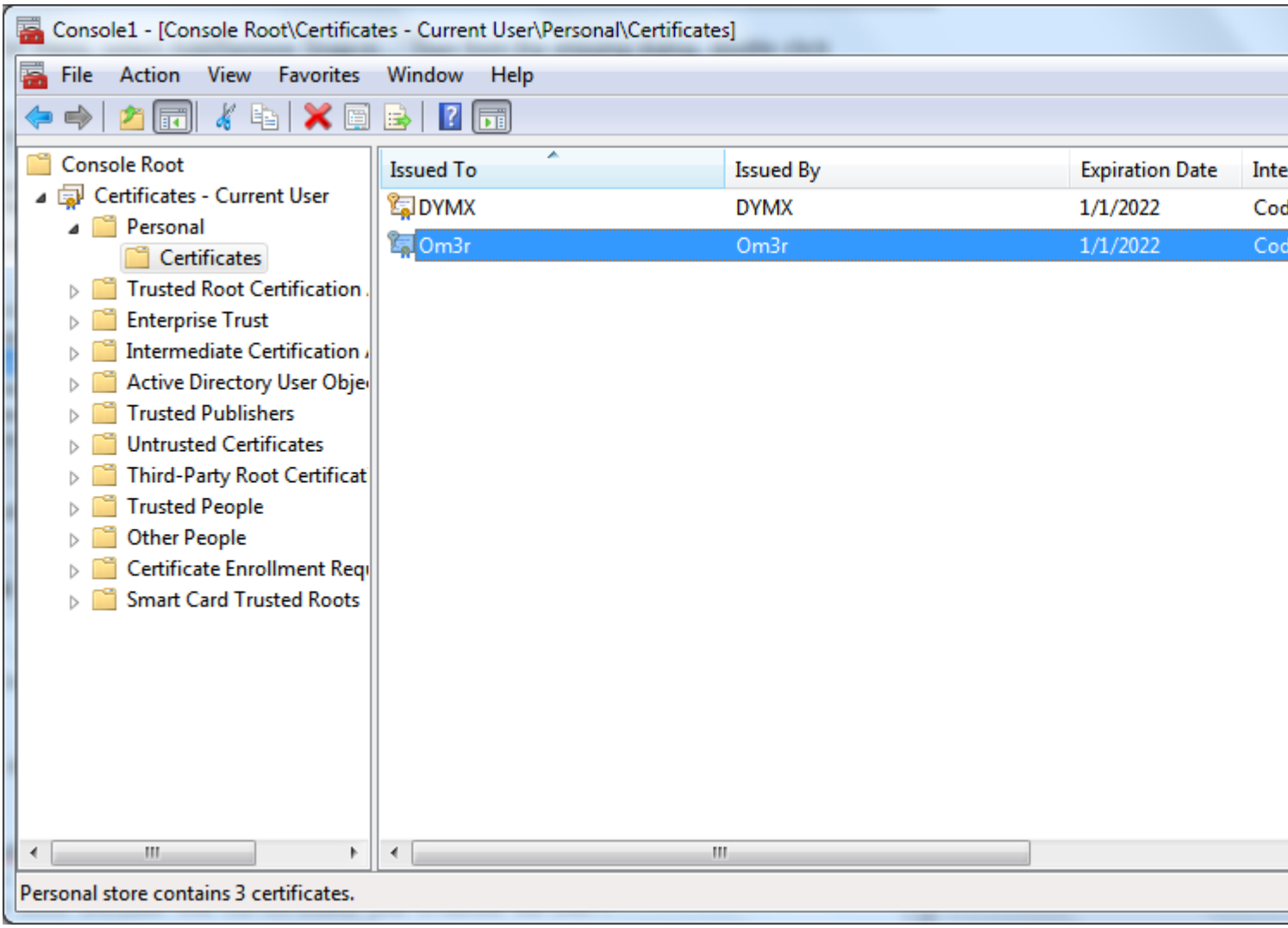
La Microsoft Management Console se abrirá y tendrá el siguiente aspecto.



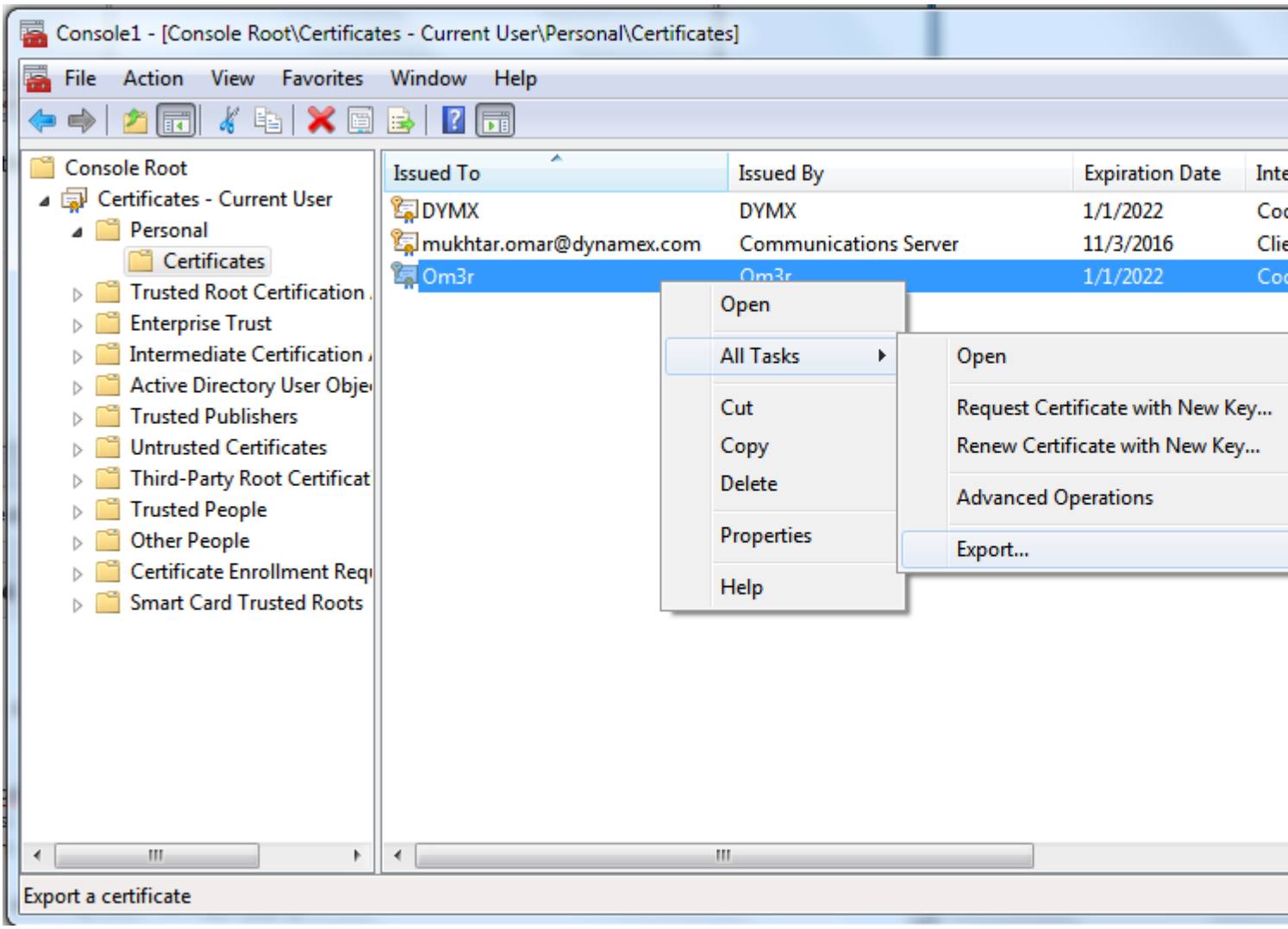
En el menú Archivo, seleccione Agregar o quitar complemento ... Luego, en el cuadro de diálogo que sigue, haga doble clic en Certificados y luego haga clic en Aceptar



Expanda el menú desplegable en la ventana izquierda para *Certificados - Usuario actual* 'y seleccione los certificados como se muestra a continuación. El panel central mostrará los certificados en esa ubicación, que incluirá el certificado que creó anteriormente:

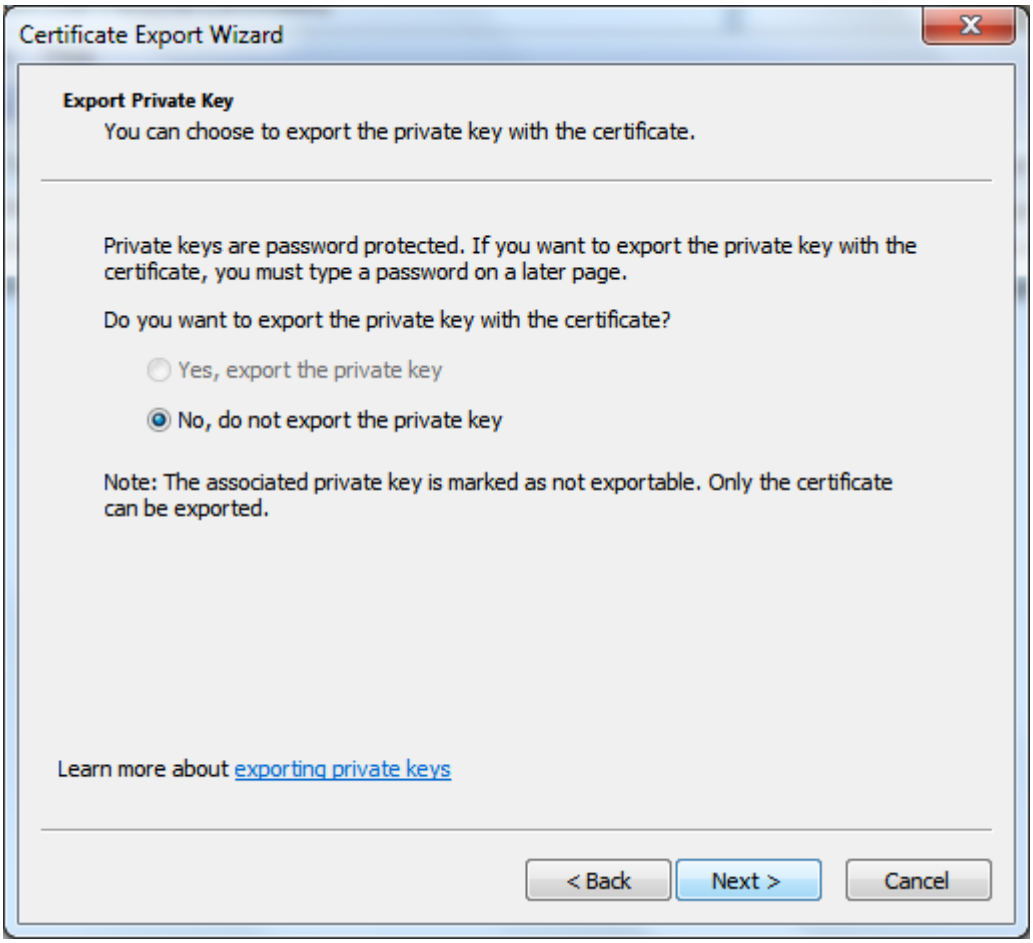


Haga clic derecho en el certificado y seleccione Todas las tareas> Exportar:

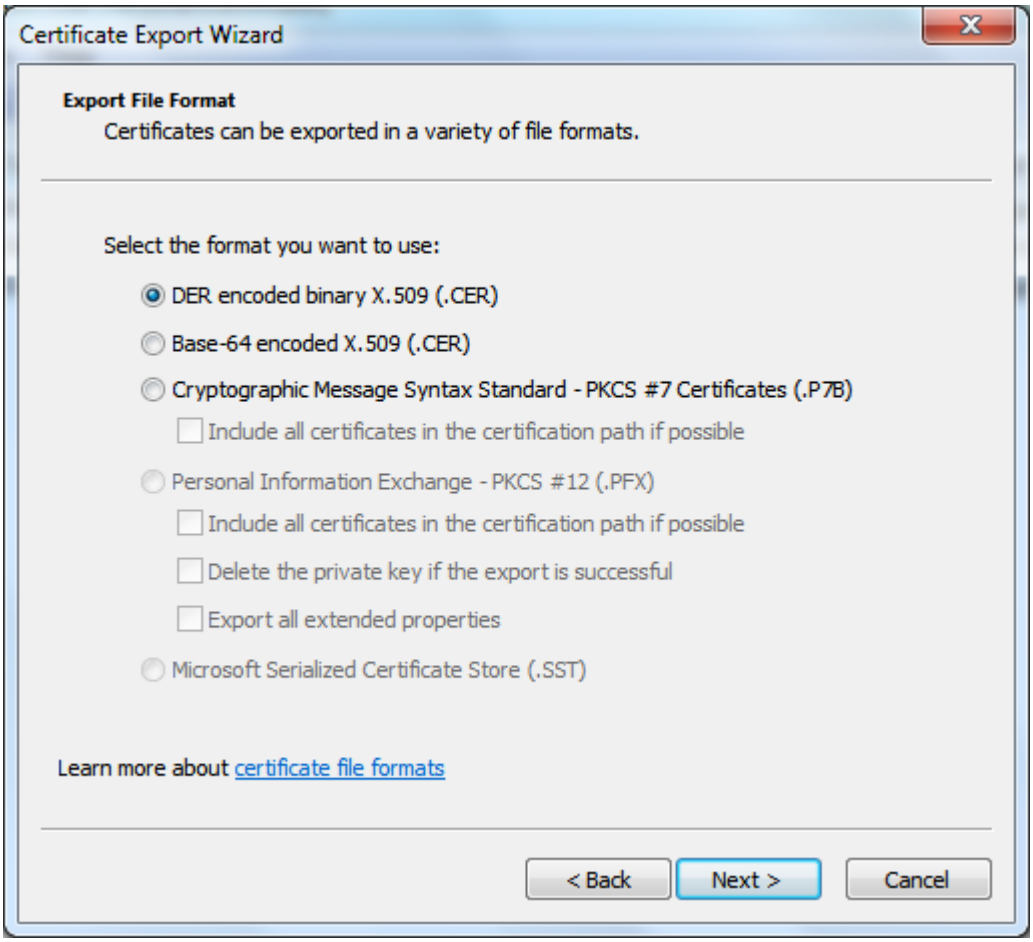


Asistente de exportación

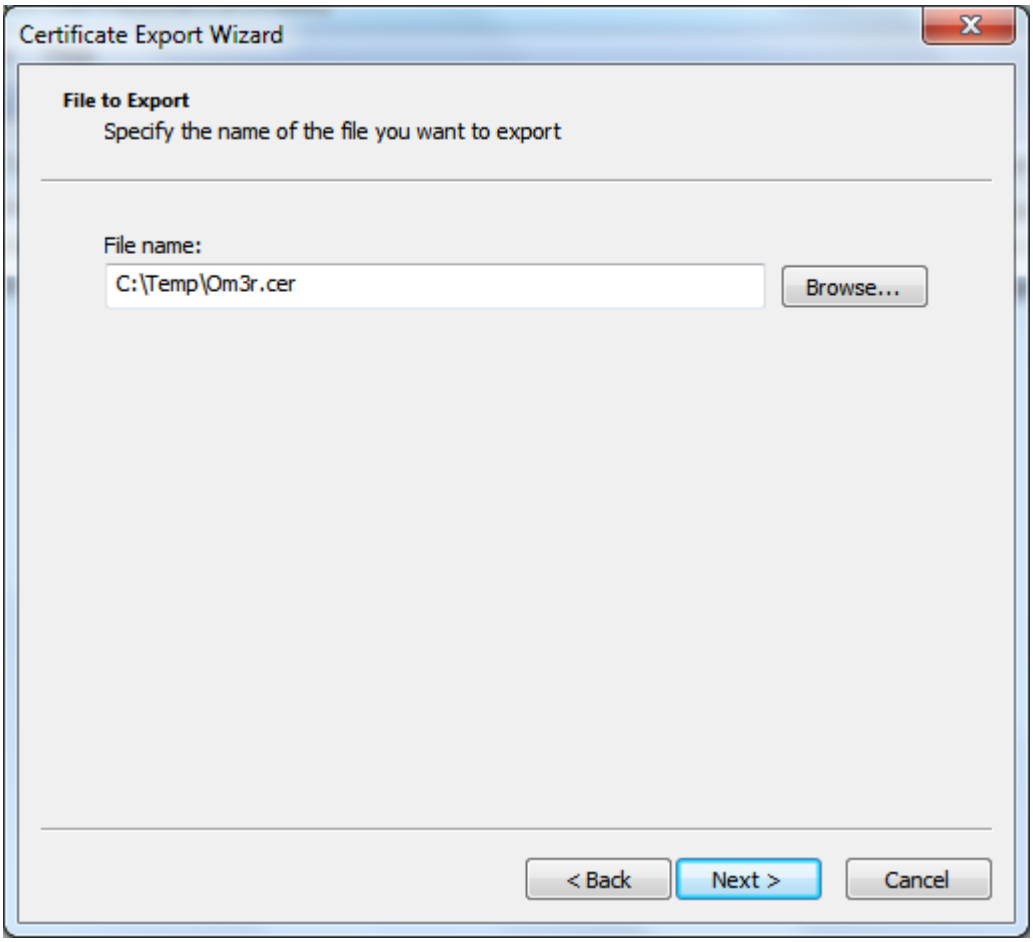
Haga clic en Siguiente



Sólo estará disponible una opción preseleccionada, por lo que haga clic en 'Siguiente' nuevamente:



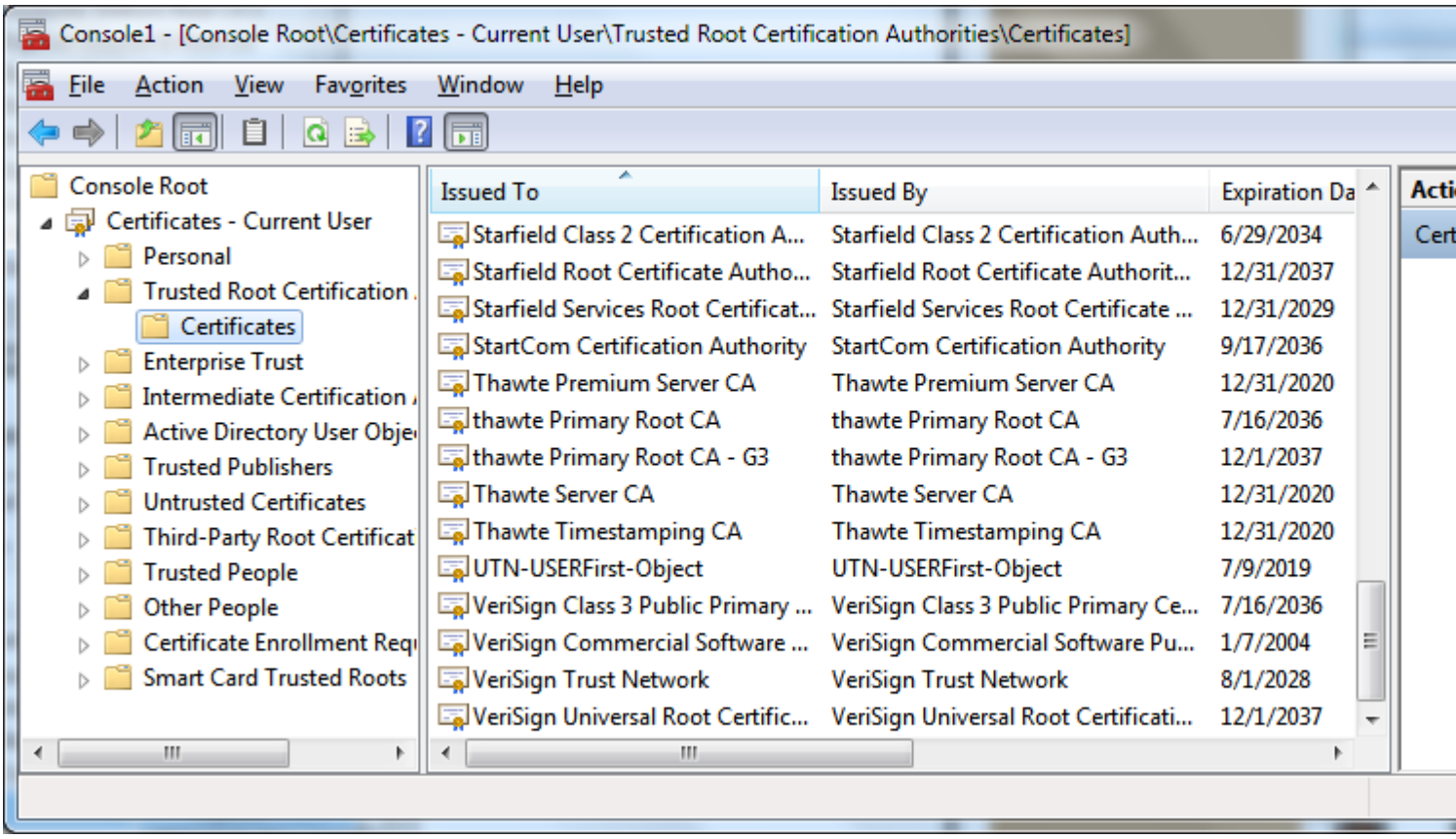
El elemento superior ya estará preseleccionado. Vuelva a hacer clic en Siguiente y elija un nombre y una ubicación para guardar el certificado exportado.



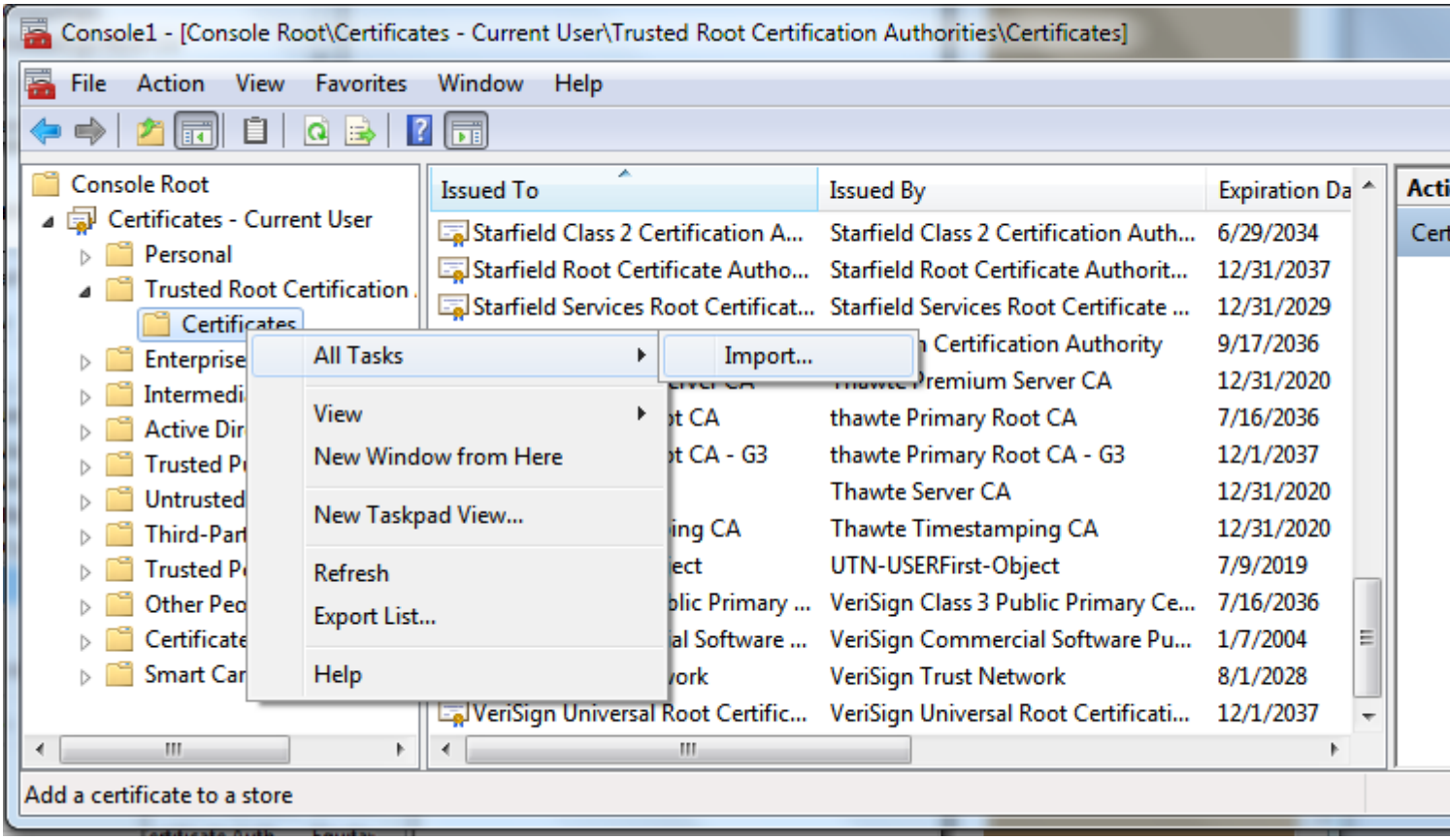
Haga clic en Siguiente nuevamente para guardar el certificado.

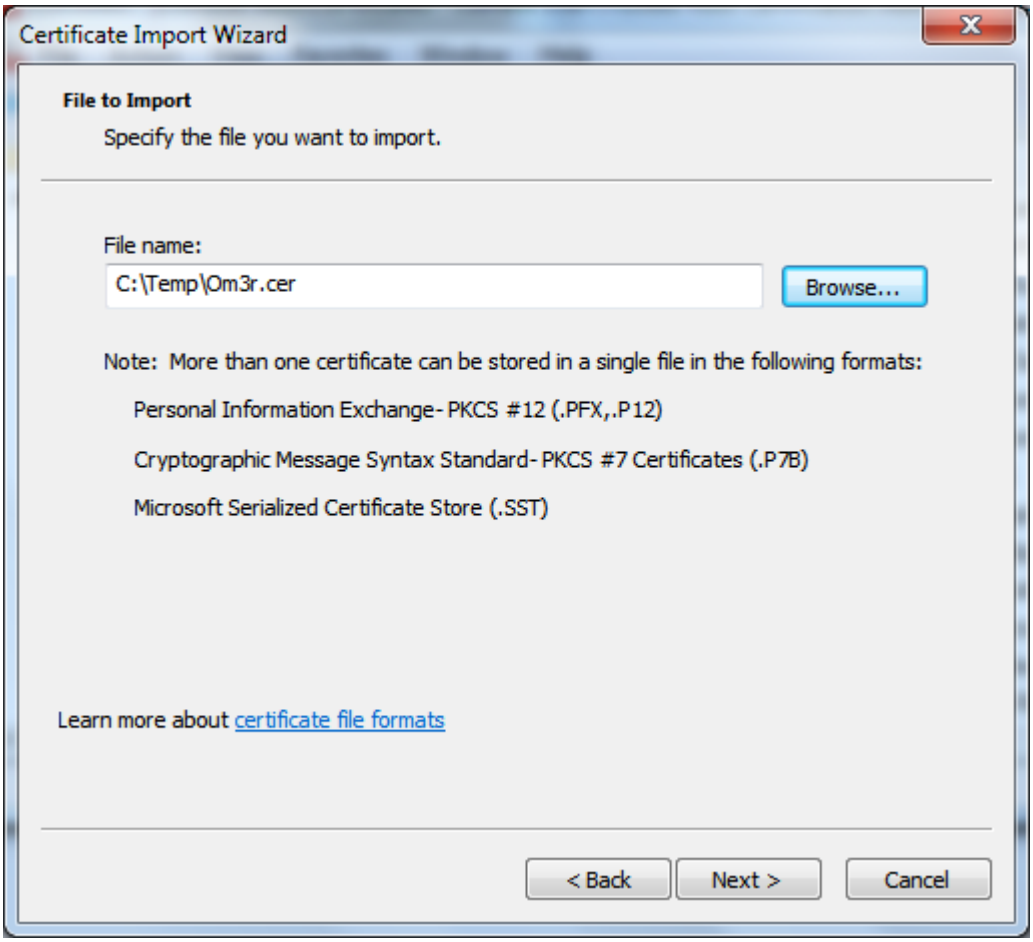
Una vez que el foco se devuelve a la consola de administración.

Expanda el menú *Certificados* y, en el menú Entidades de certificación raíz de confianza, seleccione *Certificados* .

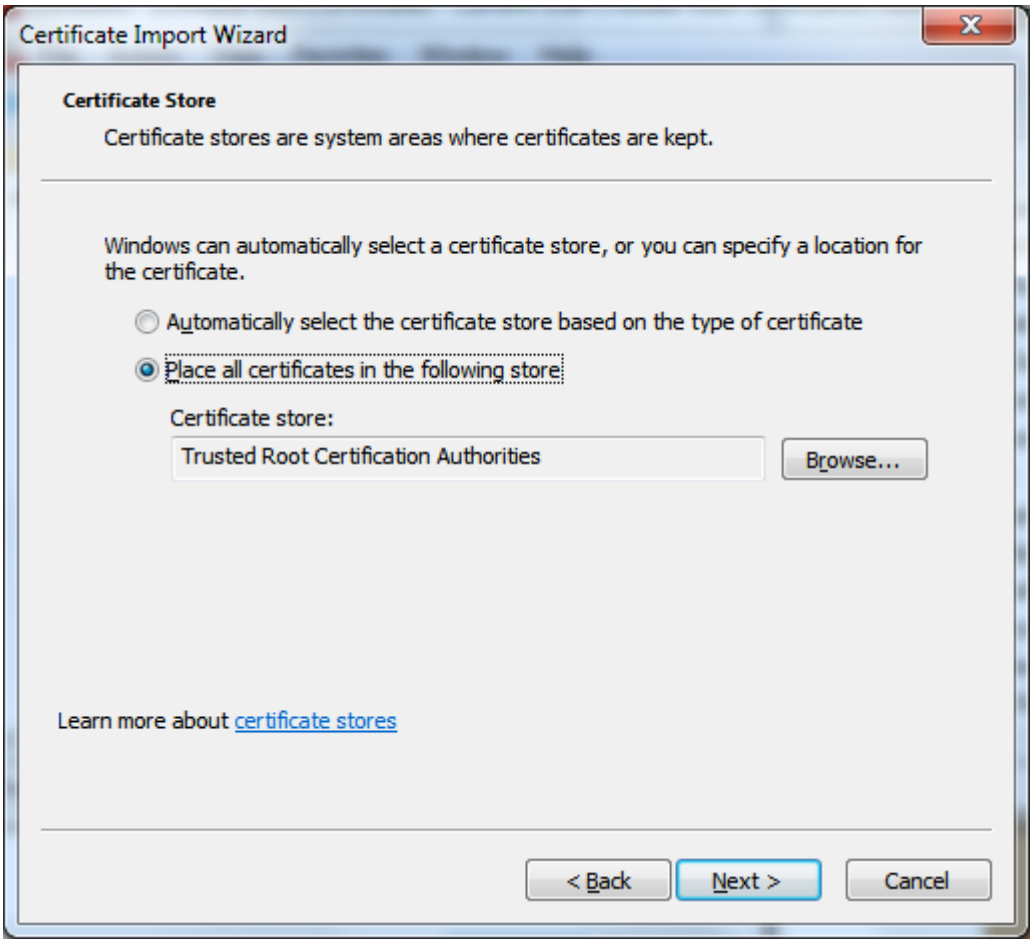


Botón derecho del ratón. Seleccionar *todas las tareas* e *importar*



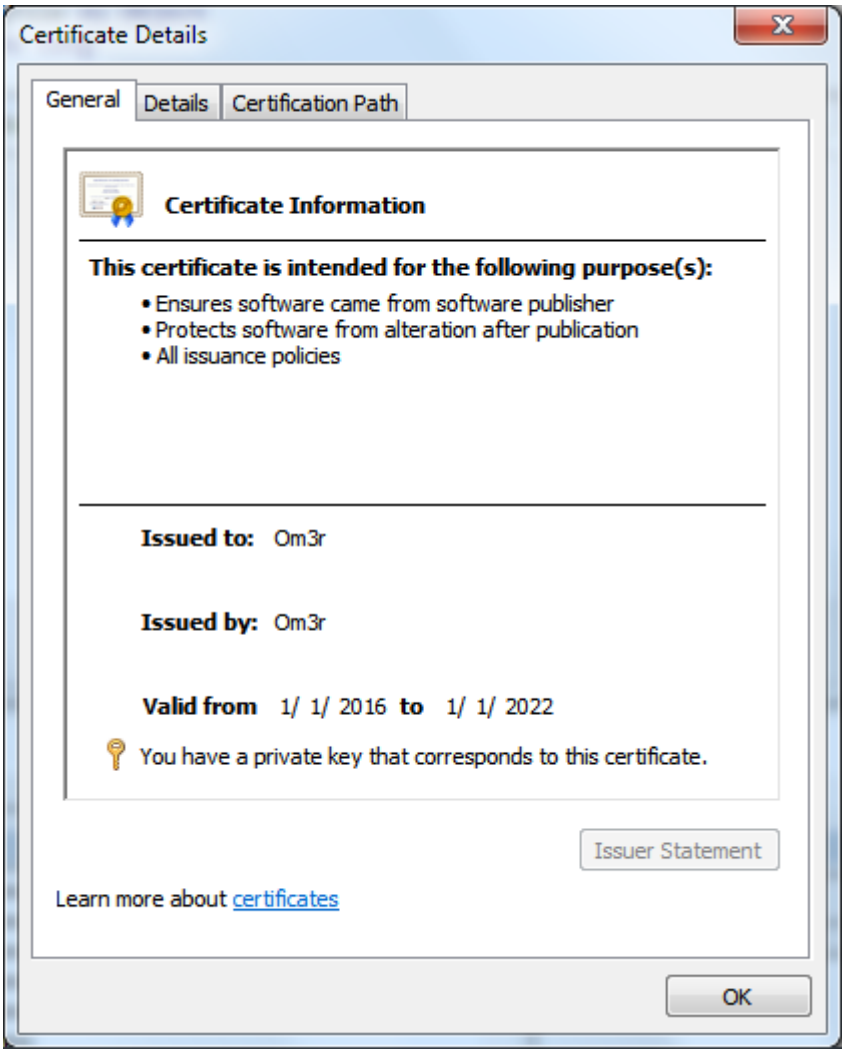


Haga clic en siguiente y guardar en la *tienda Trusted Root Certification Authorities* :



Luego Siguiente> Finalizar, ahora cierra la Consola.

Si ahora usa el certificado y verifica sus propiedades, verá que es un certificado de confianza y puede usarlo para firmar su proyecto:



Lea Macro seguridad y firma de proyectos / módulos VBA. en línea: https://riptutorial.com/es/vba/topic/7733/macro-seguridad-y-firma-de-proyectos---modulos-vba-

Capítulo 31: Manejo de errores

Examples

Evitando condiciones de error.

Cuando se produce un error de tiempo de ejecución, un buen código debe manejarlo. La mejor estrategia de manejo de errores es escribir código que verifique las condiciones de error y simplemente evite ejecutar código que genere un error de tiempo de ejecución.

Un elemento clave para reducir los errores de tiempo de ejecución, es escribir pequeños procedimientos que *hacen una cosa* . Cuantas menos razones tengan los procedimientos para fallar, más fácil será la depuración del código en su totalidad.

Evitando el error de tiempo de ejecución 91 - Objeto o con variable de bloque no establecida:

Este error se generará cuando se utilice un objeto antes de que se asigne su referencia. Uno podría tener un procedimiento que recibe un parámetro de objeto:

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

Si el `target` no tiene asignada una referencia, el código anterior generará un error que se evitará fácilmente al verificar si el objeto contiene una referencia de objeto real:

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

Si el `target` no tiene una referencia asignada, la referencia no asignada nunca se usa y no se produce ningún error.

Esta forma de salir temprano de un procedimiento cuando uno o más parámetros no es válido, se denomina *cláusula de guarda* .

Evitar el error de tiempo de ejecución 9 - Subíndice fuera de rango:

Este error se produce cuando se accede a una matriz fuera de sus límites.

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

Dado un índice mayor que el número de hojas de trabajo en `ActiveWorkbook` , el código anterior generará un error de tiempo de ejecución. Una simple cláusula de guardia puede evitar que:

```
Private Sub DoSomething(ByVal index As Integer)
    If index > ActiveWorkbook.Worksheets.Count Or index <= 0 Then Exit Sub
    Debug.Print ActiveWorkbook.Worksheets(index)
End Sub
```

La mayoría de los errores de tiempo de ejecución se pueden evitar verificando cuidadosamente los valores que usamos *antes de* usarlos, y bifurcándonos en otra ruta de ejecución en consecuencia usando una simple sentencia `If` : en cláusulas de seguridad que no hacen suposiciones y validan los parámetros de un procedimiento, o incluso en el Cuerpo de procedimientos más grandes.

Declaración de error

Incluso con *las cláusulas de guardia,* no se puede dar cuenta de manera realista *siempre* para todas las posibles condiciones de error que podrían ser planteadas en el cuerpo de un procedimiento. La instrucción `On Error GoTo` indica a VBA que salte a una *etiqueta de línea* e ingrese al "modo de manejo de errores" siempre que ocurra un error inesperado en el tiempo de ejecución. Después de manejar un error, el código puede *reanudar* de nuevo en la ejecución "normal" con el `Resume` de palabras clave.

Las etiquetas de línea denotan *subrutinas* : como las subrutinas se originan a partir del código BASIC heredado y `GoSub` saltos `GoTo` y `GoSub` y declaraciones de `Return` para saltar a la rutina "principal", es bastante fácil escribir *código espagueti* difícil de seguir si las cosas no están estructuradas de manera rigurosa . Por esta razón, es mejor que:

- un procedimiento tiene **una y solo una** subrutina de manejo de errores
- la subrutina de manejo de errores **solo se ejecuta en un estado de error**

Esto significa que un procedimiento que maneja sus errores, debe estar estructurado así:

```
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
```

Estrategias de manejo de errores

A veces quieres manejar diferentes errores con diferentes acciones. En ese caso, inspeccionará el objeto `Err` global, que contendrá información sobre el error que se generó, y actuará en consecuencia:

```
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
```

Como una guía general, considere activar el manejo de errores para una subrutina o función completa, y manejar todos los errores que puedan ocurrir dentro de su alcance. Si solo necesita manejar los errores en la sección pequeña del código, active y desactive el manejo de errores al mismo nivel:

```
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
```

Línea de números

VBA admite números de línea de estilo heredado (por ejemplo, QBASIC). La propiedad oculta de `Erl` se puede usar para identificar el número de línea que generó el último error. Si no estás usando números de línea, `Erl` solo devolverá 0.

```
Sub DoSomething()
10 On Error GoTo 50
20 Debug.Print 42 / 0
30 Exit Sub
40
50 Debug.Print "Error raised on line " & Erl ' returns 20
End Sub
```

Si *está* usando números de línea, pero no de manera consistente, entonces `Erl` devolverá *el último número de línea antes de la instrucción que generó el error* .

```
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
```

Tenga en cuenta que `Erl` también solo tiene precisión `Integer` y se desbordará silenciosamente. Esto significa que los números de línea fuera del rango de enteros darán resultados incorrectos:

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

El número de línea no es tan relevante como la declaración que causó el error, y las líneas de numeración rápidamente se vuelven tediosas y no son fáciles de mantener.

Reanudar palabra clave

Una subrutina de manejo de errores puede:

- ejecutar hasta el final del procedimiento, en cuyo caso la ejecución se reanuda en el procedimiento de llamada.
- o, use la palabra clave `Resume` para *reanudar la* ejecución dentro del mismo procedimiento.

La palabra clave `Resume` solo se debe utilizar dentro de una subrutina de manejo de errores, porque si VBA encuentra `Resume` sin estar en un estado de error, se genera el error 20 "Reanudar sin error".

Hay varias formas en que una subrutina de manejo de errores puede usar la palabra clave `Resume` :

- `Resume` utilizado solo, la ejecución continúa **en la declaración que causó el error** . Si el error no se maneja *realmente* antes de hacer eso, entonces el mismo error volverá a aparecer, y la ejecución podría entrar en un bucle infinito.
- `Resume Next` continúa la ejecución **en la instrucción inmediatamente después de** la

instrucción que causó el error. Si el error no se maneja *realmente* antes de hacer eso, entonces se permite que la ejecución continúe con datos potencialmente inválidos, lo que puede resultar en errores lógicos y un comportamiento inesperado.

- `Resume [line label]` continúa la ejecución **en la etiqueta de línea especificada** (o número de línea, si está usando números de línea de estilo heredado). Por lo general, esto permitiría ejecutar algún código de limpieza antes de salir del procedimiento de forma limpia, como asegurarse de que la conexión de la base de datos se cierre antes de regresar a la persona que llama.

En error reanudar siguiente

La propia declaración de `On Error` puede usar la palabra clave `Resume` para indicar al tiempo de ejecución de VBA que **ignore** efectivamente **todos los errores** .

> *Si el error no se **maneja realmente** antes de hacer eso, entonces se permite que la ejecución continúe con datos potencialmente inválidos, lo que puede resultar en **errores lógicos y un comportamiento inesperado** .*

El énfasis anterior no se puede enfatizar lo suficiente. ***On Error Resume Next* ignora efectivamente todos los errores y los empuja debajo de la alfombra** . Un programa que explota con un error de tiempo de ejecución dado una entrada no válida es mejor que uno que se sigue ejecutando con datos desconocidos / no deseados, ya sea porque el error es mucho más fácil de identificar. `On Error Resume Next` puede **ocultar errores** fácilmente.

La declaración `On Error` tiene un ámbito de procedimiento, por eso *normalmente* debería haber solo **una** , tal como la declaración `On Error` en un procedimiento determinado.

Sin embargo, a *veces* no se puede evitar una condición de error, y saltar a una subrutina de manejo de errores solo para `Resume Next` simplemente no se siente bien. En este caso específico, lo conocido a fallar posiblemente-afirmación puede ser **envuelto** entre dos `On Error` declaraciones:

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

La instrucción `On Error GoTo 0` restablece el manejo de errores en el procedimiento actual, de manera que cualquier instrucción adicional que cause un error en tiempo de ejecución *se manejaría dentro de ese procedimiento* y en lugar de eso pasaría la pila de llamadas hasta que la detecte un controlador de errores activo. Si no hay un controlador de errores activo en la pila de llamadas, se tratará como una excepción no controlada.

```
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
```

Errores personalizados

A menudo, al escribir una clase especializada, querrá que genere sus propios errores específicos, y querrá una forma limpia para que el usuario / código de llamada maneje estos errores personalizados. Una buena forma de lograrlo es mediante la definición de un tipo de `Enum` dedicado:

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

El uso de la `vbObjectError` incorporada `vbObjectError` garantiza que los códigos de error personalizados no se superpongan con los códigos de error reservados / existentes. Solo el primer valor de enumeración debe especificarse explícitamente, ya que el valor subyacente de cada miembro de `Enum` es `1` mayor que el miembro anterior, por lo que el valor subyacente de `Err_BarNotInitialized` es implícitamente `vbObjectError + 1025` .

Elevando tus propios errores de ejecución.

Se puede `Err.Raise` un error de tiempo de ejecución utilizando la declaración `Err.Raise` , por lo que el error personalizado `Err_FooWasNotBarred` se puede `Err_FooWasNotBarred` siguiente manera:

```
Err.Raise Err_FooWasNotBarred
```

El método `Err.Raise` también puede tomar parámetros personalizados de `Description` y `Source` ; por esta razón, es una buena idea también definir constantes para contener la descripción de cada error personalizado:

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

Y luego crea un método privado dedicado para elevar cada error:

```
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
```

La implementación de la clase puede simplemente llamar a estos procedimientos especializados para generar el error:

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

El código del cliente puede manejar `Err_BarNotInitialized` como lo haría con cualquier otro error, dentro de su propia subrutina de manejo de errores.

Nota: la palabra clave de `Error` heredada también se puede usar en lugar de `Err.Raise` , pero está obsoleta / obsoleta.

Lea Manejo de errores en línea: https://riptutorial.com/es/vba/topic/3211/manejo-de-errores

Capítulo 32: Manipulación de cuerdas de uso frecuente.

Introducción

Ejemplos rápidos para las funciones de cadena MID LEFT y RIGHT utilizando INSTR FIND y LEN.

¿Cómo encuentra el texto entre dos términos de búsqueda (por ejemplo, después de dos puntos y antes de una coma)? ¿Cómo se obtiene el resto de una palabra (usando MID o usando DERECHA)? ¿Cuál de estas funciones utiliza parámetros basados en cero y códigos de retorno frente a uno basado? ¿Qué pasa cuando las cosas van mal? ¿Cómo manejan las cadenas vacías, los resultados no encontrados y los números negativos?

Examples

Manipulación de cuerdas de uso frecuente.

Mejor MID () y otros ejemplos de extracción de cadenas, que actualmente faltan en la web. Por favor, ayúdame a dar un buen ejemplo, o completa este aquí. Algo como esto:

```
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) '...
```

Por favor, siéntase libre de editar este ejemplo y hacerlo mejor. Siempre que quede claro, y tenga en él prácticas de uso comunes.

Lea Manipulación de cuerdas de uso frecuente. en línea: https://riptutorial.com/es/vba/topic/8890/manipulacion-de-cuerdas-de-uso-frecuente-

Capítulo 33: Manipulación de fecha y hora

Examples

Calendario

VBA soporta 2 calendarios: gregoriano y hijri

La propiedad `Calendar` se utiliza para modificar o mostrar el calendario actual.

Los 2 valores para el calendario son:

Valor	Constante	Descripción
0	vbCalGreg	Calendario gregoriano (predeterminado)
1	vbCalHijri	Calendario hijri

Ejemplo

```
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
```

Este Sub imprime lo siguiente;

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

Recuperar sistema DateTime

VBA admite 3 funciones incorporadas para recuperar la fecha y / o la hora del reloj del sistema.

Función	Tipo de retorno	Valor de retorno
Ahora	Fecha	Devuelve la fecha y hora actual.
Fecha	Fecha	Devuelve la parte de fecha de la fecha y hora actual
Hora	Fecha	Devuelve la parte de tiempo de la fecha y hora actuales

```
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
```

Función de temporizador

La función de `Timer` devuelve un solo que representa el número de segundos transcurridos desde la medianoche. La precisión es una centésima de segundo.

```
Sub TimerExample()

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

End Sub
```

Debido a que las funciones `Now` y `Time` son solo precisas a segundos, el `Timer` ofrece una manera conveniente de aumentar la precisión de la medición del tiempo:

```
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 () comprueba si una expresión es una fecha válida o no. Devuelve 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
```

Funciones de extraccion

Estas funciones toman una `Variant` que se puede convertir en una `Date` como parámetro y devuelven un `Integer` representa una parte de una fecha u hora. Si el parámetro no se puede convertir en una `Date` , se producirá un error de tiempo de ejecución 13: No coincide el tipo.

Función	Descripción	Valor devuelto
Año()	Devuelve la parte del año del argumento de fecha.	Entero (100 a 9999)
Mes()	Devuelve la parte del mes del argumento de fecha.	Entero (1 a

Función	Descripción	Valor devuelto
		12)
Día()	Devuelve la parte del día del argumento de fecha.	Entero (1 a 31)
Día laborable()	Devuelve el día de la semana del argumento fecha. Acepta un segundo argumento opcional que define el primer día de la semana.	Entero (1 a 7)
Hora()	Devuelve la parte de hora del argumento de fecha.	Entero (0 a 23)
Minuto()	Devuelve la porción de minutos del argumento de fecha.	Entero (0 a 59)
Segundo()	Devuelve la segunda parte del argumento de fecha.	Entero (0 a 59)

Ejemplos:

```
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
```

Función DatePart ()

`DatePart()` es también una función que devuelve una parte de una fecha, pero funciona de manera diferente y permite más posibilidades que las funciones anteriores. Puede, por ejemplo, devolver el trimestre del año o la semana del año.

Sintaxis:

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

argumento de *intervalo* puede ser:

Intervalo	Descripción
"yyyy"	Año (100 a 9999)
"y"	Día del año (1 a 366).
"metro"	Mes (1 a 12)
"q"	Trimestre (1 a 4)
"ww"	Semana (1 a 53)
"w"	Día de la semana (1 a 7)
"re"	Día del mes (1 a 31)
"h"	Hora (0 a 23)
"norte"	Minuto (0 a 59)
"s"	Segundo (0 a 59)

Firstdayofweek es opcional. Es una constante que especifica el primer día de la semana. Si no se especifica, se asume `vbSunday` .

Firstweekofyear es opcional. Es una constante que especifica la primera semana del año. Si no se especifica, se supone que la primera semana es la semana en que ocurre el 1 de enero.

Ejemplos:

```
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
```

DateDiff ()

`DateDiff()` devuelve un valor `Long` representa el número de intervalos de tiempo entre dos fechas especificadas.

Sintaxis

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

- *intervalo* puede ser cualquiera de los intervalos definidos en la función `DatePart()`
- *fecha1* y *fecha2* son las dos fechas que desea utilizar en el cálculo
- *primerdíadelasemana* y *firstweekofyear* son opcionales. Consulte la función `DatePart()` para obtener explicaciones.

Ejemplos

```
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
```

FechaAgregar ()

`DateAdd()` devuelve una `Date` a la que se ha agregado una fecha o un intervalo de tiempo específico.

Sintaxis

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

- *intervalo* puede ser cualquiera de los intervalos definidos en la función `DatePart()`
- *número* Expresión numérica que es el número de intervalos que desea agregar. Puede ser positivo (para obtener fechas en el futuro) o negativo (para obtener fechas en el pasado).

- *fecha* es una `Date` o literal que representa la fecha a la que se agrega el intervalo

Ejemplos:

```
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
```

Conversión y Creación

CDate ()

`CDate()` convierte algo de cualquier tipo de datos a un tipo de `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
```

Tenga en cuenta que VBA también tiene un `CVDate()` escrito de forma `CVDate()` que funciona de la misma manera que la función `CDate()` , aparte de devolver una `Variant` tipo de fecha en lugar de una `Date` tipo muy fuerte. La versión `CDate()` debe ser preferida cuando se pasa a un parámetro de `Date` o se asigna a una variable de `Date` , y la versión `CVDate()` debe preferir cuando se pasa a un parámetro de `Variant` o se asigna a una variable de `Variant` . Esto evita la conversión implícita de tipos.

DateSerial ()

`DateSerial()` función `DateSerial()` se utiliza para crear una fecha. Devuelve una `Date` para un año, mes y día especificados.

Sintaxis:

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

Con año, mes y día los argumentos son enteros válidos (año de 100 a 9999, mes de 1 a 12, día de 1 a 31).

Ejemplos

```
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
```

Tenga en cuenta que `DateSerial()` aceptará fechas "no válidas" y calculará una fecha válida a partir de ellas. Esto puede ser usado creativamente para bien:

Ejemplo 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
```

Sin embargo, es más probable que cause dolor al intentar crear una fecha a partir de una entrada de usuario no validada:

Ejemplo 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
```

Lea Manipulación de fecha y hora en línea: https://riptutorial.com/es/vba/topic/4452/manipulacion-de-fecha-y-hora

Capítulo 34: Midiendo la longitud de las cuerdas

Observaciones

La longitud de una cadena se puede medir de dos maneras: la medida de longitud más utilizada es el número de caracteres que usan las funciones de `Len` , pero VBA también puede revelar el número de bytes que usan `LenB` funciones de `LenB` . Un carácter de doble byte o Unicode tiene más de un byte de longitud.

Examples

Usa la función de Len para determinar el número de caracteres en una cadena

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

Dim charLength As Long

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

Utilice la función LenB para determinar el número de bytes en una cadena

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

Dim byteLength As Long

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

Prefiero `If Len (myString) = 0 Then` sobre` If myString = "" Then`

Cuando se comprueba si una cadena es de longitud cero, es una mejor práctica y más eficiente, inspeccionar la longitud de la cadena en lugar de comparar la cadena con una cadena vacía.

```
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
```

Lea Midiendo la longitud de las cuerdas en línea:
https://riptutorial.com/es/vba/topic/3576/midiendo-la-longitud-de-las-cuerdas

Capítulo 35: Objeto Scripting.Dictionary

Observaciones

Debe agregar Microsoft Scripting Runtime al proyecto VBA a través del comando Herramientas → Referencias de VBE para implementar el enlace anticipado del objeto Scripting Dictionary. Esta referencia de la biblioteca se lleva con el proyecto; no es necesario volver a referenciarlo cuando el proyecto de VBA se distribuye y ejecuta en otra computadora.

Examples

Propiedades y metodos

Un objeto del Diccionario de Scripting almacena información en pares de Clave / Elemento. Las Claves deben ser únicas y no una matriz, pero los Elementos asociados se pueden repetir (su singularidad se mantiene en la Clave complementaria) y puede ser de cualquier tipo de variante u objeto.

Un diccionario puede considerarse como una base de datos de dos campos en la memoria con un índice único primario en el primer "campo" (la *clave*). Este índice único en la propiedad Claves permite "búsquedas" muy rápidas para recuperar el valor del elemento asociado de una Clave.

Propiedades

nombre	leer escribir	tipo	descripción
CompareMode	*leer escribir*	Constante de modo de comparacion	La configuración del modo de comparación solo se puede realizar en un diccionario vacío. Los valores aceptados son 0 (vbBinaryCompare), 1 (vbTextCompare), 2 (vbDatabaseCompare).
Contar	*solo lectura*	entero largo sin signo	Un recuento basado en uno de los pares de clave / elemento en el objeto de diccionario de scripting.
Llave	*leer escribir*	variante no matricial	Cada clave única individual en el diccionario.
Artículo (*Clave*)	*leer escribir*	cualquier variante	Propiedad por defecto. Cada elemento individual asociado con una clave en el diccionario. Tenga en cuenta que el intento de recuperar un elemento con una clave que no existe en el diccionario agregará *implícitamente*

nombre	leer escribir	tipo	descripción
			la clave pasada.

Métodos

nombre	descripción
Añadir (*clave* , *elemento*)	Agrega una nueva clave y elemento al diccionario. La nueva clave no debe existir en la colección de claves actual del diccionario, pero un elemento puede repetirse entre muchas claves únicas.
Existe (*clave*)	Prueba booleana para determinar si ya existe una clave en el diccionario.
Llaves	Devuelve la matriz o colección de claves únicas.
Artículos	Devuelve la matriz o colección de elementos asociados.
Eliminar (*clave*)	Elimina una clave de diccionario individual y su elemento asociado.
Eliminar todo	Borra todas las claves y elementos de un objeto de diccionario.

Código de muestra

```
'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
```

Agregando datos con Scripting.Dictionary (Maximum, Count)

Los diccionarios son excelentes para administrar información donde se producen entradas múltiples, pero solo le preocupa un valor único para cada conjunto de entradas: el primer o último valor, el mínimo o el valor máximo, un promedio, una suma, etc.

Considere un libro de trabajo que contenga un registro de la actividad del usuario, con un script que inserte el nombre de usuario y la fecha de edición cada vez que alguien edite el libro de trabajo:

Hoja de `Log`

UNA	segundo
mover	12/12/2016 9:00
Alicia	13/10/2016 13:00
mover	13/10/2016 13:30
Alicia	13/10/2016 14:00
Alicia	14/10/2016 13:00

Supongamos que desea generar la última hora de edición para cada usuario en una hoja de cálculo llamada `Summary` .

Notas:

1. Se asume que los datos están en `ActiveWorkbook` .
2. Estamos utilizando una matriz para extraer los valores de la hoja de trabajo; esto es más eficiente que iterar sobre cada celda.
3. El `Dictionary` se crea mediante el enlace temprano.

```
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
```

y la salida se verá así:

Hoja de trabajo de `Summary`

UNA	segundo
mover	13/10/2016 13:30
Alicia	14/10/2016 13:00

Si, por otro lado, desea mostrar cuántas veces ha editado el libro cada usuario, el cuerpo del bucle `For` debe tener este aspecto:

```
        '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
```

y la salida se verá así:

Hoja de trabajo de `Summary`

UNA	segundo
mover	2
Alicia	3

Obteniendo valores únicos con Scripting.Dictionary

El `Dictionary` permite obtener un conjunto único de valores muy simple. Considera la siguiente función:

```
Function Unique(values As Variant) As Variant()
    'Put all the values as keys into a dictionary
    Dim dict As New Scripting.Dictionary
```

```
    Dim val As Variant
    For Each val In values
        dict(val) = 1 'The value doesn't matter here
    Next
    Unique = dict.Keys
End Function
```

que entonces podrías llamar así:

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

y `uniqueVals` contendría solo `{1,2,3}` .

Nota: esta función se puede utilizar con cualquier objeto enumerable.

Lea Objeto Scripting.Dictionary en línea: https://riptutorial.com/es/vba/topic/3667/objeto-scripting-dictionary

Capítulo 36: Opción VBA Palabra clave

Sintaxis

- Opción optionName [valor]
- Opción explícita
- Opción Comparar {Texto | Binario Base de datos}
- Opción Módulo Privado
- Base de opciones {0 | 1}

Parámetros

Opción	Detalle
Explícito	*Requerir declaración de variable* en el módulo en el que se especifica (idealmente todas); con esta opción especificada, el uso de una variable no declarada (/ mal escrito) se convierte en un error de compilación.
Comparar texto	Hace que las comparaciones de cadenas del módulo no distingan entre mayúsculas y minúsculas, según la configuración regional del sistema, dando prioridad a la equivalencia alfabética (por ejemplo, "a" = "A").
Comparar binario	Modo de comparación de cadena predeterminado. Hace que las comparaciones de cadenas del módulo distingan entre mayúsculas y minúsculas, comparando cadenas utilizando la representación binaria / valor numérico de cada carácter (por ejemplo, ASCII).
Comparar base de datos	(Solo MS-Access) Hace que las comparaciones de cadenas del módulo funcionen como lo harían en una declaración SQL.
Módulo privado	Impide que se acceda al miembro `Public` del módulo desde fuera del proyecto en el que reside el módulo, ocultando efectivamente los procedimientos de la aplicación host (es decir, no está disponible para usar como macros o funciones definidas por el usuario).
Opción Base 0	Configuración predeterminada. Establece el límite inferior de la matriz implícita en `0` en un módulo. Cuando se declara una matriz sin un valor límite inferior explícito, se utilizará `0` .
Opción Base 1	Establece el límite inferior de la matriz implícita en `1` en un módulo. Cuando se declara una matriz sin un valor explícito límite inferior, `1` se utilizará.

Observaciones

Es mucho más fácil controlar los límites de las matrices declarando los límites explícitamente en lugar de dejar que el compilador retroceda en una declaración de `Option Base {0|1}` . Esto se puede hacer así:

```
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)
```

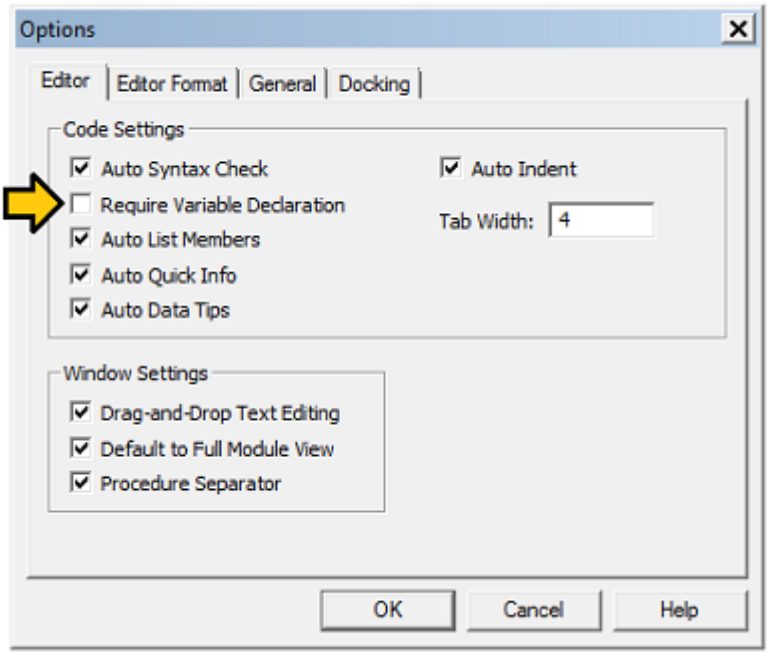
Examples

Opción explícita

Se considera la mejor práctica usar siempre `Option Explicit` en VBA, ya que obliga al desarrollador a declarar todas sus variables antes de usar. Esto también tiene otros beneficios, como el uso de mayúsculas automáticas para los nombres de variables declarados e IntelliSense.

```
Option Explicit

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

La configuración de **Requerir declaración de variable** dentro de las Herramientas de VBE ▸ `Opciones ▸ página de propiedades del editor colocará la declaración` **Option Explicit** en la parte superior de cada hoja de códigos recién creada.



Esto evitará errores de codificación tontos, como errores ortográficos, y también influirá en el uso del tipo de variable correcto en la declaración de la variable. (Algunos ejemplos más se dan en SIEMPRE use "Opción explícita" .)

Opción Comparar Binario

La comparación binaria hace que todas las comprobaciones de igualdad de cadenas dentro de un módulo / clase *distingan entre* mayúsculas y *minúsculas* . Técnicamente, con esta opción, las comparaciones de cadenas se realizan utilizando el orden de clasificación de las representaciones binarias de cada carácter.

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

Si no se especifica ninguna opción de comparación en un módulo, se usa Binary de forma predeterminada.

```
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
```

Opción Comparar texto

Option Compare Text hace que todas las comparaciones de cadenas dentro de un módulo / clase use una comparación que no *distinga* mayúsculas y *minúsculas* .

(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
```

Opción Comparar base de datos

La base de datos de comparación de opciones solo está disponible dentro de MS Access. Establece el módulo / clase para usar la configuración actual de la base de datos para determinar si usar el modo Texto o Binario.

Nota: se desaconseja el uso de esta configuración a menos que el módulo se use para escribir UDF (funciones definidas por el usuario) personalizadas de Access que deban tratar las comparaciones de texto de la misma manera que las consultas SQL en esa base de datos.

Base de opciones {0 | 1}

`Option Base` se utiliza para declarar el límite inferior predeterminado de los elementos de la **matriz** . Se declara a nivel de módulo y solo es válido para el módulo actual.

De forma predeterminada (y, por lo tanto, si no se especifica una Base de Opción), la Base es 0. Lo que significa que el primer elemento de cualquier matriz declarada en el módulo tiene un índice de 0.

Si se especifica la `Option Base 1` , el primer elemento de la matriz tiene el índice 1

Ejemplo en Base 0:

```
Option Base 0
```

```
Sub BaseZero()

    Dim myStrings As Variant

    ' Create an array out of the Variant, having 3 fruits elements
    myStrings = Array("Apple", "Orange", "Peach")

    Debug.Print LBound(myStrings) ' This Prints "0"
    Debug.Print UBound(myStrings) ' This print "2", because we have 3 elements beginning at 0
-> 0,1,2

    For i = 0 To UBound(myStrings)

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

    Next i

End Sub
```

Mismo ejemplo 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
```

El segundo ejemplo generó un Subíndice fuera de rango (Error 9) en la primera etapa del bucle porque se realizó un intento de acceder al índice 0 de la matriz, y este índice no existe ya que el módulo se declara con `Base 1`

El código correcto con Base 1 es:

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

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

    Next i
```

Debe tenerse en cuenta que la función Dividir **siempre** crea una matriz con un índice de elemento

basado en cero, independientemente de la configuración de `Option Base` . Los ejemplos sobre cómo usar la función **Split** se pueden encontrar aquí.

> Función de división
>
> > Devuelve una matriz unidimensional basada en cero que contiene un número específico de subcadenas.

En Excel, las propiedades `Range.Value` y `Range.Formula` para un rango de varias celdas *siempre* devuelven una matriz de Variante 2D basada en 1.

Del mismo modo, en ADO, el método `Recordset.GetRows` *siempre* devuelve una matriz 2D basada en 1.

Una de las mejores prácticas recomendadas es usar siempre las funciones LBound y UBound para determinar las extensiones de una matriz.

```
'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
```

La `Option Base 1` debe estar en la parte superior de cada módulo de código donde se crea una matriz o se redimensiona para que las matrices se creen de manera consistente con un límite inferior de 1.

Lea Opción VBA Palabra clave en línea: https://riptutorial.com/es/vba/topic/3992/opcion-vba-palabra-clave

Capítulo 37: Pasando Argumentos ByRef o ByVal

Introducción

Los modificadores `ByRef` y `ByVal` son parte de la firma de un procedimiento e indican cómo se pasa un argumento a un procedimiento. En VBA, se pasa un parámetro `ByRef` menos que se especifique lo contrario (es decir, `ByRef` está implícito si está ausente).

Nota En muchos otros lenguajes de programación (incluido VB.NET), los parámetros se pasan implícitamente por valor si no se especifica ningún modificador: considere la posibilidad de especificar los modificadores `ByRef` explícitamente para evitar posibles confusiones.

Observaciones

Pasando matrices

Las matrices **deben** ser pasadas por referencia. Este código se compila, pero genera un error de tiempo de ejecución 424 "Objeto requerido":

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

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

Este código no se compila:

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

Examples

Pasando variables simples ByRef y ByVal

Pasar `ByRef` o `ByVal` indica si el valor real de un argumento se pasa al `CalledProcedure` mediante el `CallingProcedure` , o si una referencia (llamada puntero en otros idiomas) se pasa al `CalledProcedure` .

Si se pasa un argumento `ByRef` , la dirección de memoria del argumento se pasa al `CalledProcedure` y cualquier modificación a ese parámetro por el `CalledProcedure` se hace al valor en el `CallingProcedure` .

Si se pasa un argumento `ByVal` , el valor real, no una referencia a la variable, se pasa a `CalledProcedure` .

Un ejemplo simple ilustrará esto claramente:

```
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
```

Otro ejemplo:

```
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

Modificador por defecto

Si no se especifica ningún modificador para un parámetro, ese parámetro se pasa implícitamente por referencia.

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

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

El parámetro `foo` se pasa `ByRef` tanto en `DoSomething1` como en `DoSomething2` .

> **¡Cuidado!** Si viene a VBA con experiencia de otros idiomas, es muy probable que este sea el comportamiento opuesto al que está acostumbrado. En muchos otros lenguajes de programación (incluido VB.NET), el modificador implícito / predeterminado pasa los parámetros por valor.

Pasando por referencia

- Cuando se pasa un *valor* `ByRef` , el procedimiento recibe **una referencia** al valor.

```
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
```

  Llamar a las salidas del procedimiento de `Test` 84. `DoSomething` se da `foo` y recibe una *referencia* al valor y, por lo tanto, funciona con la misma dirección de memoria que la persona que llama.

- Cuando se pasa una *referencia* `ByRef` , el procedimiento recibe **una referencia** al puntero.

```
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
```

  El código anterior genera el error de tiempo de ejecución 91 , porque el autor de la llamada está llamando al miembro `Count` de un objeto que ya no existe, porque `DoSomething` recibió una *referencia* al puntero del objeto y lo asignó a `Nothing` antes de regresar.

Forzar ByVal en el sitio de la llamada

Al usar paréntesis en el sitio de la llamada, puede anular `ByRef` y forzar la aprobación de un argumento `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
```

El código anterior genera 42, independientemente de si `ByRef` se especifica implícita o explícitamente.

> **¡Cuidado!** Debido a esto, el uso de paréntesis extraños en las llamadas a procedimientos puede introducir errores fácilmente. Preste atención al espacio en blanco entre el nombre del procedimiento y la lista de argumentos:
>
> ```
> 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

Pasando por valor

- Cuando se pasa un *valor* `ByVal` , el procedimiento recibe **una copia** del valor.

  ```
  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
  ```

  Llamando a las salidas del procedimiento de `Test` 42. `DoSomething` se da `foo` y recibe **una copia** del valor. La copia se multiplica por 2 y luego se desecha cuando finaliza el procedimiento; La copia de la persona que llama nunca fue alterada.

- Cuando se pasa una *referencia* `ByVal` , el procedimiento recibe **una copia** del puntero.

```
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
```

Llamando a la anterior `Test` salidas de procedimiento 1. `DoSomething` es dada `foo` y recibe *una copia* **del puntero** a la `Collection` de objetos. Debido a que la variable de objeto `foo` en el alcance de `Test` apunta al mismo objeto, agregar un elemento en `DoSomething` agrega el elemento al mismo objeto. Debido a que es *una copia* del puntero, establecer su referencia en `Nothing` no afecta a la propia copia de la persona que llama.

Lea Pasando Argumentos ByRef o ByVal en línea:
https://riptutorial.com/es/vba/topic/7363/pasando-argumentos-byref-o-byval

Capítulo 38: Personajes no latinos

Introducción

VBA puede leer y escribir cadenas en cualquier idioma o script utilizando Unicode . Sin embargo, existen reglas más estrictas para los identificadores de identificadores .

Examples

Texto no latino en código VBA

En la celda A1 de la hoja de cálculo, tenemos el siguiente pangram árabe:

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

VBA proporciona las funciones `AscW` y `ChrW` para trabajar con códigos de caracteres de múltiples bytes. También podemos usar matrices de `Byte` para manipular la variable de cadena directamente:

```
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
```

Esto produce la siguiente salida para la letra árabe Sad :

> Primera char (ChrW): 1589
> Primer char (binario): 00011000110101
> Valor de cadena (texto):?
> Valor de cadena (AscW): 1589
> Valores de byte (en decimal): 53 | 6
> Valores de bytes (binarios): 00110101 | 00000110
>
> Cadena entera
> ??? ????? ????? ??????? ??????? ??? ??????? - ????? ???????? ???? ??????? ???????

Tenga en cuenta que VBA no puede imprimir texto no latino en la ventana inmediata a pesar de que las funciones de cadena funcionan correctamente. Esta es una limitación del IDE y no del idioma.

Identificadores no latinos y cobertura de idiomas

Los identificadores de VBA (nombres de variables y funciones) pueden usar el script latino y también pueden usar scripts en japonés , coreano , chino simplificado y chino tradicional .

El script latino extendido tiene cobertura completa para muchos idiomas:
Inglés, francés, español, alemán, italiano, bretón, catalán, danés, estonio, finlandés, islandés, indonesio, irlandés, Lojban, mapudungun, noruego, portugués, gaélico escocés, sueco, tagalo

Algunos idiomas solo están cubiertos parcialmente:
Azerí, croata, checo, esperanto, húngaro, letón, lituano, polaco, rumano, serbio, eslovaco, esloveno, turco, yoruba, galés

Algunos idiomas tienen poca o ninguna cobertura:
Árabe, búlgaro, cherokee, dzongkha, griego, hindi, macedonio, malayalam, mongol, ruso, sánscrito, tailandés, tibetano, urdu, uigur

Las siguientes declaraciones de variables son todas válidas:

```
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
```

Tenga en cuenta que en el IDE de VBA, un solo apóstrofe dentro de un nombre de variable no convierte la línea en un comentario (como lo hace en Stack Overflow).

Además, los idiomas que usan dos ángulos para indicar una cita «» pueden usarlos en nombres de variables a pesar de que las comillas de tipo "" no lo son.

Lea Personajes no latinos en línea: https://riptutorial.com/es/vba/topic/10555/personajes-no-latinos

Capítulo 39: Procedimiento de llamadas

Sintaxis

- IdentifierName [*argumentos*]
- Call IdentifierName [*(argumentos)*]
- [Let | Set] *expresión* = IdentifierName [*(argumentos)*]
- [Let | Set] IdentifierName [*(argumentos)*] = *expresión*

Parámetros

Parámetro	Información
Nombre del identificador	El nombre del procedimiento a llamar.
argumentos	Una lista de argumentos separados por comas para pasar al procedimiento.

Observaciones

Las dos primeras sintaxis son para llamar a los procedimientos `Sub` ; Note que la primera sintaxis no implica paréntesis.

Ver Esto es confuso. ¿Por qué no usar siempre paréntesis? para una explicación detallada de las diferencias entre las dos primeras sintaxis.

La tercera sintaxis es para llamar a los procedimientos de `Function` y `Function Property Get` ; Cuando hay parámetros, los paréntesis son siempre obligatorios. La palabra clave `Let` es opcional cuando se asigna un *valor* , pero se **requiere la** palabra clave `Set` cuando se asigna una *referencia* .

La cuarta sintaxis es para llamar a los procedimientos `Property Let` y `Property Set` ; la *`expression`* en el lado derecho de la asignación se pasa al parámetro de valor de la propiedad.

Examples

Sintaxis de llamada implícita

```
ProcedureName
ProcedureName argument1, argument2
```

Llame a un procedimiento por su nombre sin paréntesis.

Caso extremo

La palabra clave de `Call` solo se requiere en un caso de borde:

```
Call DoSomething : DoSomethingElse
```

`DoSomething` y `DoSomethingElse` son procedimientos a los que se llama. Si se eliminara la palabra clave de `Call` , `DoSomething` se analizaría como una *etiqueta de línea en* lugar de una llamada de procedimiento, lo que rompería el código:

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

Valores de retorno

Para recuperar el resultado de una llamada de procedimiento (por ejemplo, los procedimientos de `Function` o `Property Get`), coloque la llamada en el lado derecho de una asignación:

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

Los paréntesis deben estar presentes si hay parámetros. Si el procedimiento no tiene parámetros, los paréntesis son redundantes.

Esto es confuso. ¿Por qué no usar siempre paréntesis?

Los paréntesis se utilizan para encerrar los argumentos de *las llamadas* de *función* . Su uso para *llamadas de procedimiento* puede causar problemas inesperados.

Porque pueden introducir errores, tanto en tiempo de ejecución al pasar un valor posiblemente no deseado al procedimiento, como en tiempo de compilación simplemente por ser una sintaxis no válida.

Tiempo de ejecución

Los paréntesis redundantes pueden introducir errores. Dado un procedimiento que toma una referencia de objeto como parámetro ...

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

... y llama con paréntesis:

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

Esto generará un error de tiempo de ejecución "Objeto requerido" # 424. Otros errores son posibles en otras circunstancias: aquí la referencia del objeto `Application.ActiveCell Range` se *evalúa* y se pasa por valor, **independientemente** de la firma del procedimiento que especifique que `ByRef` pasaría el `target` . El valor real pasado de `ByVal` a `DoSomething` en el fragmento anterior, es `Application.ActiveCell.Value` .

Los paréntesis obligan a VBA a evaluar el valor de la expresión entre corchetes y pasar el resultado `ByVal` al procedimiento llamado. Cuando el tipo del resultado evaluado no coincide con el tipo esperado del procedimiento y no se puede convertir implícitamente, se genera un error de tiempo de ejecución.

Tiempo de compilación

Este código no podrá compilar:

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

Debido a que la expresión `("Invalid Code!", vbCritical)` no se puede *evaluar* a un valor.

Esto compilaría y funcionaría:

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

Pero definitivamente se vería tonto. Evite los paréntesis redundantes.

Sintaxis explícita de llamadas

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

La sintaxis de llamada explícita requiere la palabra clave de `Call` y paréntesis alrededor de la lista de argumentos; los paréntesis son redundantes si no hay parámetros. Esta sintaxis se volvió obsoleta cuando se agregó a VB la sintaxis de llamada implícita más moderna.

Argumentos opcionales

Algunos procedimientos tienen argumentos opcionales. Los argumentos opcionales siempre vienen después de los argumentos requeridos, pero el procedimiento se puede llamar sin ellos.

Por ejemplo, si la función, `ProcedureName` tuviera dos argumentos obligatorios (`argument1` , `argument2`) y un argumento opcional, `optArgument3` , podría llamarse al menos cuatro formas:

```
' 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 estructura del encabezado de la función que se llama aquí sería algo así:

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

La palabra clave `Optional` indica que este argumento se puede omitir. Como se mencionó anteriormente, todos los argumentos opcionales introducidos en el encabezado **deben** aparecer al final, después de los argumentos requeridos.

También puede proporcionar un valor *predeterminado* para el argumento en el caso de que un valor no se pase a la función:

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

En esta función, si no se proporciona el argumento para `c` , su valor será por defecto `"C"` . Si *se* proporciona un valor, esto anulará el valor predeterminado.

Lea Procedimiento de llamadas en línea: https://riptutorial.com/es/vba/topic/1179/procedimiento-de-llamadas

Capítulo 40: Recursion

Introducción

Una función que se llama a sí misma se dice que es *recursiva* . La lógica recursiva a menudo también se puede implementar como un bucle. La recursión debe controlarse con un parámetro, de modo que la función sepa cuándo dejar de recurrir y profundizar la pila de llamadas. *La recursión infinita* eventualmente causa un error en el tiempo de ejecución '28': "Fuera de espacio de pila".

Ver Recursión .

Observaciones

La recursión permite llamadas repetidas y autorreferencia de un procedimiento.

Examples

Factoriales

```
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
```

Carpeta Recursion

Early Bound (con una referencia 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
```

Lea Recursion en línea: https://riptutorial.com/es/vba/topic/3236/recursion

Capítulo 41: Scripting.FileSystemObject

Examples

Creando un 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
```

Leyendo un archivo de texto usando 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
```

Creando un archivo de texto 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
```

Escribir en un archivo existente 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
```

Enumerar archivos en un directorio usando FileSystemObject

Límite temprano (requiere una referencia 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
```

Límite tardío:

```
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
```

Enumerar recursivamente carpetas y archivos

Early Bound (con una referencia 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
```

Salida cuando se llama con argumentos como: `EnumerateFilesAndFolders "C:\Test"`

```
C:\Test
  Documents
    Personal
      Budget.xls
      Recipes.doc
    Work
      Planning.doc
  Downloads
    FooBar.exe
  ReadMe.txt
```

Salida cuando se llama con argumentos como: `EnumerateFilesAndFolders "C:\Test", 0`

```
C:\Test
  Documents
  Downloads
  ReadMe.txt
```

Salida cuando se llama con argumentos como: `EnumerateFilesAndFolders "C:\Test", 1, 4`

```
C:\Test
    Documents
        Personal
        Work
    Downloads
        FooBar.exe
    ReadMe.txt
```

Tira la extensión de archivo de un nombre de archivo

```
Dim fso As New Scripting.FileSystemObject
Debug.Print fso.GetBaseName("MyFile.something.txt")
```

Imprime `MyFile.something`

Tenga en cuenta que el método `GetBaseName()` ya maneja varios períodos en un nombre de archivo.

Recupera solo la extensión de un nombre de archivo

```
Dim fso As New Scripting.FileSystemObject
Debug.Print fso.GetExtensionName("MyFile.something.txt")
```

Imprime `txt` Tenga en cuenta que el método `GetExtensionName()` ya maneja múltiples períodos en un nombre de archivo.

Recuperar solo la ruta de acceso de un archivo

El método GetParentFolderName devuelve la carpeta principal para cualquier ruta. Si bien esto también se puede usar con carpetas, podría decirse que es más útil para extraer la ruta de una ruta de archivo absoluta:

```
Dim fso As New Scripting.FileSystemObject
Debug.Print fso.GetParentFolderName("C:\Users\Me\My Documents\SomeFile.txt")
```

Imprime `C:\Users\Me\My Documents`

Tenga en cuenta que el separador de ruta de acceso no se incluye en la cadena devuelta.

Uso de FSO.BuildPath para crear una ruta completa desde la ruta de la carpeta y el nombre del archivo

Si está aceptando la entrada del usuario para las rutas de las carpetas, es posible que deba verificar si hay barras diagonales (`\`) antes de crear una ruta de archivo. El método `FSO.BuildPath` hace esto más simple:

```
  Const sourceFilePath As String = "C:\Temp"  '<-- Without trailing backslash
  Const targetFilePath As String = "C:\Temp\" '<-- With trailing backslash

  Const fileName As String = "Results.txt"

  Dim FSO As FileSystemObject
  Set FSO = New FileSystemObject

  Debug.Print FSO.BuildPath(sourceFilePath, fileName)
  Debug.Print FSO.BuildPath(targetFilePath, fileName)
```

Salida:

```
C:\Temp\Results.txt
C:\Temp\Results.txt
```

Lea Scripting.FileSystemObject en línea: https://riptutorial.com/es/vba/topic/990/scripting-filesystemobject

Capítulo 42: Subcadenas

Observaciones

VBA tiene funciones incorporadas para extraer partes específicas de cadenas, que incluyen:

- `Left` / `Left$`
- `Right` / `Right$`
- `Mid` / `Mid$`
- `Trim` / `Trim$`

Para evitar la conversión de tipo implícita en la cabecera (y, por lo tanto, para un mejor rendimiento), use la versión con el sufijo $ de la función cuando se pasa una variable de cadena a la función, y / o si el resultado de la función se asigna a una variable de cadena.

Pasar un valor de parámetro `Null` a una función $ -suffixed generará un error de tiempo de ejecución ("uso no válido de nulo"), esto es especialmente relevante para el código que involucra una base de datos.

Examples

Use Left o Left $ para obtener los 3 caracteres más a la izquierda en una cadena

```
Const baseString As String = "Foo Bar"

Dim leftText As String
leftText = Left$(baseString, 3)
'leftText = "Foo"
```

Use Right o Right $ para obtener los 3 caracteres más a la derecha en una cadena

```
Const baseString As String = "Foo Bar"
Dim rightText As String
rightText = Right$(baseString, 3)
'rightText = "Bar"
```

Use Mid o Mid $ para obtener caracteres específicos dentro de una cadena

```
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"
```

Utilice Recortar para obtener una copia de la cadena sin espacios iniciales ni finales

```
'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"
```

Lea Subcadenas en línea: https://riptutorial.com/es/vba/topic/3481/subcadenas

Capítulo 43: Tipos de datos y límites

Examples

Byte

```
Dim Value As Byte
```

Un byte es un tipo de datos de 8 bits sin firmar. Puede representar números enteros entre 0 y 255 y tratar de almacenar un valor fuera de ese rango dará como resultado un error de tiempo de ejecución 6: `Overflow` . Byte es el único tipo sin signo intrínseco disponible en VBA.

La función de conversión para convertir a un byte es `CByte()` . Para las conversiones de tipos de punto flotante, el resultado se redondea al valor entero más cercano con .5 redondeo hacia arriba.

Byte Arrays y cadenas

Las cadenas y las matrices de bytes se pueden sustituir entre sí mediante una asignación simple (no se requieren funciones de conversión).

Por ejemplo:

```
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
```

Para poder codificar caracteres Unicode , cada carácter de la cadena ocupa dos bytes en la matriz, con el byte menos significativo primero. Por ejemplo:

```
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
```

Entero

```
Dim Value As Integer
```

Un entero es un tipo de datos firmado de 16 bits. Puede almacenar números enteros en el rango de -32,768 a 32,767 y tratar de almacenar un valor fuera de ese rango resultará en un error de tiempo de ejecución 6: Desbordamiento.

Los enteros se almacenan en la memoria como valores little-endian con negativos representados como un complemento de dos .

Tenga en cuenta que, en general, es una mejor práctica utilizar un tipo Long en lugar de un Integer, a menos que el tipo más pequeño sea miembro de un Tipo o sea requerido (ya sea por una convención de llamada de API o por alguna otra razón) para que sea de 2 bytes. En la mayoría de los casos, VBA trata a los enteros como 32 bits internamente, por lo que generalmente no hay ninguna ventaja al usar el tipo más pequeño. Además, hay una penalización en el rendimiento en la que se incurre cada vez que se usa un tipo Integer, ya que se lanza silenciosamente como un Long.

La función de conversión para convertir a un entero es `CInt()` . Para las conversiones de tipos de punto flotante, el resultado se redondea al valor entero más cercano con .5 redondeo hacia arriba.

Booleano

```
Dim Value As Boolean
```

Un booleano se utiliza para almacenar valores que pueden representarse como Verdadero o Falso. Internamente, el tipo de datos se almacena como un valor de 16 bits con 0 que representa Falso y cualquier otro valor que representa Verdadero.

Se debe tener en cuenta que cuando un valor booleano se convierte en un tipo numérico, todos los bits se establecen en 1. Esto da como resultado una representación interna de -1 para los tipos con signo y el valor máximo para un tipo sin signo (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 función de conversión para convertir a un Booleano es `CBool()` . Aunque se representa internamente como un número de 16 bits, la conversión a un valor booleano desde valores fuera de ese rango está a salvo del desbordamiento, aunque establece todos los 16 bits en 1:

```
Dim Example As Boolean
```

```
Example = CBool(2 ^ 17)
Debug.Print CInt(Example)   'Prints -1
Debug.Print CByte(Example)  'Prints 255
```

Largo

```
Dim Value As Long
```

A Long es un tipo de datos firmado de 32 bits. Puede almacenar números enteros en el rango de -2,147,483,648 a 2,147,483,647 y tratar de almacenar un valor fuera de ese rango resultará en un error de tiempo de ejecución 6: Desbordamiento.

Los largos se almacenan en la memoria como valores little-endian con negativos representados como un complemento de dos .

Tenga en cuenta que dado que un Largo coincide con el ancho de un puntero en un sistema operativo de 32 bits, los Largos se usan comúnmente para almacenar y pasar punteros hacia y desde las funciones API.

La función de conversión para convertir a un largo es `CLng()` . Para las conversiones de tipos de punto flotante, el resultado se redondea al valor entero más cercano con .5 redondeo hacia arriba.

Soltero

```
Dim Value As Single
```

Un Single es un tipo de datos de punto flotante de 32 bits firmado. Se almacena internamente utilizando un diseño de memoria IEEE 754 little-endian . Como tal, no hay un rango fijo de valores que pueda representarse por el tipo de datos; lo que está limitado es la precisión del valor almacenado. Un Single puede almacenar valores de valores ***enteros*** en el rango de -16,777,216 a 16,777,216 sin pérdida de precisión. La precisión de los números de punto flotante depende del exponente.

Un solo se desbordará si se le asigna un valor mayor que aproximadamente 2^{128} . No se desbordará con exponentes negativos, aunque la precisión utilizable será cuestionable antes de que se alcance el límite superior.

Al igual que con todos los números de punto flotante, se debe tener cuidado al hacer comparaciones de igualdad. La mejor práctica es incluir un valor delta apropiado para la precisión requerida.

La función de conversión para convertir a un solo es `CSng()` .

Doble

```
Dim Value As Double
```

Un doble es un tipo de datos de punto flotante de 64 bits firmado. Al igual que el Single , se almacena internamente utilizando un diseño de memoria IEEE 754 little-endian y se deben tomar las mismas precauciones con respecto a la precisión. Un Double puede almacenar valores ***enteros*** en el rango de -9,007,199,254,740,992 a 9,007,199,254,740,992 sin pérdida de precisión. La precisión de los números de punto flotante depende del exponente.

Un doble se desbordará si se le asigna un valor mayor que aproximadamente 2 ^{1024} . No se desbordará con exponentes negativos, aunque la precisión utilizable será cuestionable antes de que se alcance el límite superior.

La función de conversión para convertir a un doble es `CDbl()` .

Moneda

```
Dim Value As Currency
```

Una moneda es un tipo de datos de punto flotante con signo de 64 bits similar a un doble , pero escalado en 10,000 para dar una mayor precisión a los 4 dígitos a la derecha del punto decimal. Una variable de moneda puede almacenar valores desde -922,337,203,685,477.5808 hasta 922,337,203,685,477.5807, lo que le otorga la mayor capacidad de cualquier tipo intrínseco en una aplicación de 32 bits. Como lo indica el nombre del tipo de datos, se considera la mejor práctica usar este tipo de datos cuando se representan cálculos monetarios, ya que la escala ayuda a evitar errores de redondeo.

La función de conversión para convertir a una moneda es `CCur()` .

Fecha

```
Dim Value As Date
```

Un tipo de fecha se representa internamente como un tipo de datos de coma flotante de 64 bits con signo con el valor a la izquierda del punto decimal que representa el número de días desde la fecha de época de 30 de diciembre de 1899 (aunque véase la nota a continuación). El valor a la derecha del decimal representa la hora como un día fraccionario. Por lo tanto, una fecha entera tendría un componente de hora de 12:00:00 AM y x.5 tendría un componente de hora de 12:00:00 PM.

Los valores válidos para fechas están entre el 1 de enero de 100 y 31 de diciembre de 9999. Desde una doble tiene un alcance más amplio, es posible desbordamiento de una fecha mediante la asignación de valores fuera de ese rango.

Como tal, se puede usar indistintamente con los cálculos de Doble para la fecha:

```
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 función de conversión para convertir a una Fecha es `CDate()` , que acepta cualquier tipo de representación numérica de fecha / hora de cadena. Es importante tener en cuenta que las representaciones de cadena de las fechas se convertirán en función de la configuración regional actual en uso, por lo que se deben evitar los lanzamientos directos si se pretende que el código sea portátil.

Cuerda

Una cadena representa una secuencia de caracteres y viene en dos sabores:

Longitud variable

```
Dim Value As String
```

Una cadena de longitud variable permite agregar y truncar y se almacena en la memoria como un COM BSTR . Consiste en un entero sin signo de 4 bytes que almacena la longitud de la cadena en bytes seguidos por los propios datos de cadena como caracteres anchos (2 bytes por carácter) y terminados con 2 bytes nulos. Por lo tanto, la longitud máxima de la cadena que puede ser manejada por VBA es 2,147,483,647 caracteres.

El puntero interno a la estructura (recuperable por la función `StrPtr()`) apunta a la ubicación de la memoria de los *datos* , no al prefijo de longitud. Esto significa que una cadena de VBA se puede pasar directamente a las funciones de la API que requieren un puntero a una matriz de caracteres.

Debido a que la longitud puede cambiar, VBA reasigna la memoria para una Cadena *cada vez que se asigna la variable* , lo que puede imponer penalizaciones de rendimiento para los procedimientos que las alteran repetidamente.

Longitud fija

```
Dim Value As String * 1024    'Declares a fixed length string of 1024 characters.
```

Las cadenas de longitud fija se asignan a 2 bytes para cada carácter y se almacenan en la memoria como una simple matriz de bytes. Una vez asignada, la longitud de la Cadena es inmutable. ***No*** están terminación nula en la memoria, por lo que una cadena que se llena la memoria asignada con caracteres que no son nulos no es adecuado para pasar a funciones API esperan una cadena terminada en nulo.

Las cadenas de longitud fija superan una limitación del índice de 16 bits heredado, por lo que solo pueden tener hasta 65.535 caracteres de longitud. El intento de asignar un valor más largo que el espacio de memoria disponible no generará un error de tiempo de ejecución; en su lugar, el valor resultante simplemente se truncará:

```
Dim Foobar As String * 5
Foobar = "Foo" & "bar"
```

```
Debug.Print Foobar          'Prints "Fooba"
```

La función de conversión para convertir a una cadena de cualquier tipo es `CStr()` .

Largo largo

```
Dim Value As LongLong
```

A LongLong es un tipo de datos firmado de 64 bits y solo está disponible en aplicaciones de 64 bits. **No** está disponible en aplicaciones de 32 bits que se ejecutan en sistemas operativos de 64 bits. Puede almacenar valores enteros en el rango de -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807 y tratar de almacenar un valor fuera de ese rango resultará en un error de tiempo de ejecución 6: Desbordamiento.

Los LongLongs se almacenan en la memoria como valores little-endian con negativos representados como un complemento de dos .

El tipo de datos LongLong se introdujo como parte del soporte del sistema operativo de 64 bits de VBA. En aplicaciones de 64 bits, este valor se puede utilizar para almacenar y pasar punteros a API de 64 bits.

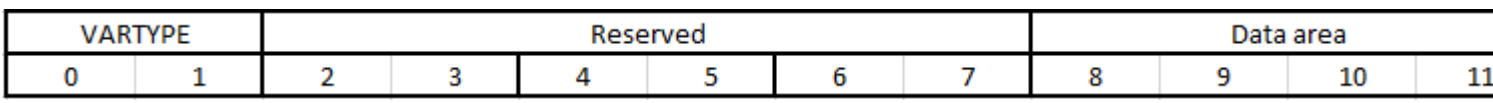
La función de conversión para convertir a un LongLong es `CLngLng()` . Para las conversiones de tipos de punto flotante, el resultado se redondea al valor entero más cercano con .5 redondeo hacia arriba.

Variante

```
Dim Value As Variant    'Explicit
Dim Value               'Implicit
```

Una variante es un tipo de datos COM que se utiliza para almacenar e intercambiar valores de tipos arbitrarios, y cualquier otro tipo en VBA se puede asignar a una variante. Las variables declaradas sin un tipo explícito especificado por `As [Type]` defecto a Variant.

Las variantes se almacenan en la memoria como una estructura VARIANTE que consta de un descriptor de tipo de byte (VARTYPE) seguido de 6 bytes reservados y luego un área de datos de 8 bytes. Para los tipos numéricos (incluidos Date y Boolean), el valor subyacente se almacena en la propia Variant. Para todos los demás tipos, el área de datos contiene un puntero al valor subyacente.



El tipo subyacente de una Variante se puede determinar con la función `VarType()` que devuelve el valor numérico almacenado en el descriptor de tipo, o la función `TypeName()` que devuelve la representación de la cadena:

```
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"
```

Debido a que las Variantes pueden almacenar valores de cualquier tipo, las asignaciones de literales sin sugerencias de tipo se convertirán implícitamente a una Variante del tipo apropiado de acuerdo con la siguiente tabla. Los literales con sugerencias de tipo se convertirán a una Variante del tipo indicado.

Valor	Tipo resultante
Valores de cadena	Cuerda
Números de punto no flotante en rango Integer	Entero
Números de punto no flotante en largo alcance	Largo
Números de punto no flotante fuera de largo alcance	Doble
Todos los números de punto flotante	Doble

Nota: A menos que haya una razón específica para usar una Variante (es decir, un iterador en un bucle For Each o un requisito de API), el tipo generalmente se debe evitar para las tareas de rutina por las siguientes razones:

- No son de tipo seguro, lo que aumenta la posibilidad de errores de tiempo de ejecución. Por ejemplo, una Variante que contiene un valor Integer se convertirá silenciosamente en un Largo en lugar de desbordarse.
- Introducen la sobrecarga de procesamiento al requerir al menos una desreferencia de puntero adicional.
- El requisito de memoria para una Variante es siempre ***al menos*** 8 bytes más alto que el necesario para almacenar el tipo subyacente.

La función de conversión para convertir a una variante es `CVar()` .

LongPtr

```
Dim Value As LongPtr
```

El LongPtr se introdujo en VBA para admitir plataformas de 64 bits. En un sistema de 32 bits, se trata como un sistema Long y en sistemas de 64 bits se trata como un LongLong .

Su uso principal es proporcionar una forma portátil para almacenar y pasar punteros en ambas arquitecturas (consulte Cambiar el comportamiento del código en tiempo de compilación .

Aunque el sistema operativo lo trata como una dirección de memoria cuando se usa en llamadas a la API, se debe tener en cuenta que VBA lo trata como un tipo firmado (y, por lo tanto, está sujeto a desbordamiento firmado y no firmado). Por esta razón, cualquier aritmética de punteros realizada con LongPtrs no debe usar comparaciones `>` o `<` . Esta "peculiaridad" también hace posible que agregar desplazamientos simples que apuntan a direcciones válidas en la memoria puede causar errores de desbordamiento, por lo que se debe tener cuidado al trabajar con punteros en VBA.

La función de conversión para convertir a un LongPtr es `CLngPtr()` . Para las conversiones de tipos de punto flotante, el resultado se redondea al valor entero más cercano con .5 redondeo hacia arriba (aunque como es generalmente una dirección de memoria, usarlo como un objetivo de asignación para un cálculo de punto flotante es, en el mejor de los casos, peligroso).

Decimal

```
Dim Value As Variant
Value = CDec(1.234)

'Set Value to the smallest possible Decimal value
Value = CDec("0.0000000000000000000000000001")
```

El tipo de datos `Decimal` *solo* está disponible como subtipo de `Variant` , por lo que debe declarar cualquier variable que deba contener un `Decimal` como `Variant` y *luego* asignar un valor `Decimal` mediante la función `CDec` . La palabra clave `Decimal` es una palabra reservada (lo que sugiere que VBA eventualmente agregará soporte de primera clase para el tipo), por lo que el `Decimal` no se puede usar como una variable o nombre de procedimiento.

El tipo `Decimal` requiere 14 bytes de memoria (además de los bytes requeridos por la variante principal) y puede almacenar números con hasta 28 decimales. Para los números sin ningún lugar decimal, el rango de valores permitidos es de -79,228,162,514,264,337,593,550,335 a +79,228,162,514,264,337,593,543,950,335 inclusive. Para los números con el máximo de 28 decimales, el rango de valores permitidos es de -7.9228162514264337593543950335 a +7.9228162514264337593543950335 inclusive.

Lea Tipos de datos y límites en línea: https://riptutorial.com/es/vba/topic/3418/tipos-de-datos-y-limites

Capítulo 44: Trabajando con ADO

Observaciones

Los ejemplos que se muestran en este tema utilizan el enlace inicial para mayor claridad y requieren una referencia a la biblioteca Microsoft ActiveX Data Object xx. Se pueden convertir a enlace tardío reemplazando las referencias fuertemente tipadas con `Object` y reemplazando la creación de objetos usando `New` con `CreateObject` cuando sea apropiado.

Examples

Haciendo una conexión a una fuente de datos

El primer paso para acceder a una fuente de datos a través de ADO es crear un objeto de `Connection` ADO. Normalmente, esto se hace usando una cadena de conexión para especificar los parámetros de la fuente de datos, aunque también es posible abrir una conexión DSN pasando el DSN, el ID de usuario y la contraseña al método `.Open` .

Tenga en cuenta que no se requiere un DSN para conectarse a una fuente de datos a través de ADO: cualquier fuente de datos que tenga un proveedor ODBC puede conectarse con la cadena de conexión apropiada. Si bien las cadenas de conexión específicas para diferentes proveedores están fuera del alcance de este tema, ConnectionStrings.com es una excelente referencia para encontrar la cadena adecuada para su proveedor.

```
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
```

Tenga en cuenta que la contraseña de la base de datos se incluye en la cadena de conexión en el ejemplo anterior solo por razones de claridad. Las mejores prácticas dictarían ***no*** almacenar las contraseñas de la base de datos en el código. Esto puede lograrse tomando la contraseña a través de la entrada del usuario o usando la autenticación de Windows.

Recuperando registros con una consulta

Las consultas se pueden realizar de dos formas, ambas de las cuales devuelven un objeto `Recordset` ADO que es una colección de filas devueltas. Tenga en cuenta que los dos ejemplos a continuación utilizan la función `OpenDatabaseConnection` del ejemplo de Conexión a un origen de datos con el propósito de ser breves. Recuerde que la sintaxis del SQL pasado al origen de datos es específica del proveedor.

El primer método es pasar la instrucción SQL directamente al objeto Connection, y es el método más sencillo para ejecutar consultas simples:

```
Public Sub DisplayDistinctItems()
    On Error GoTo Handler
    Dim database As ADODB.Connection
    Set database = OpenDatabaseConnection(SomeDSN)

    If Not database Is Nothing Then
        Dim records As ADODB.Recordset
        Set records = database.Execute("SELECT DISTINCT Item FROM Table")
        'Loop through the returned Recordset.
        Do While Not records.EOF      'EOF is false when there are more records.
            'Individual fields are indexed either by name or 0 based ordinal.
            'Note that this is using the default .Fields member of the Recordset.
            Debug.Print records("Item")
            'Move to the next record.
            records.MoveNext
        Loop
    End If
CleanExit:
    If Not records Is Nothing Then records.Close
    If Not database Is Nothing And database.State = adStateOpen Then
        database.Close
    End If
    Exit Sub
Handler:
    Debug.Print "Error " & Err.Number & ": " & Err.Description
    Resume CleanExit
End Sub
```

El segundo método es crear un objeto de `Command` ADO para la consulta que desea ejecutar. Esto requiere un poco más de código, pero es necesario para usar consultas parametrizadas:

```
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
```

Tenga en cuenta que los comandos enviados al origen de datos son **vulnerables a la inyección de SQL** , ya sea intencional o no intencional. En general, las consultas no deben crearse concatenando entradas de usuario de ningún tipo. En su lugar, deben estar parametrizados (ver Crear comandos parametrizados).

Ejecutando funciones no escalares

Las conexiones ADO se pueden usar para realizar prácticamente cualquier función de base de datos que el proveedor admita a través de SQL. En este caso, no siempre es necesario usar el `Recordset` devuelto por la función `Execute` , aunque puede ser útil para obtener asignaciones de claves después de las instrucciones INSERT con @@ Identity o comandos SQL similares. Tenga en cuenta que el siguiente ejemplo utiliza la función `OpenDatabaseConnection` del ejemplo de Conexión a un origen de datos con el propósito de brevedad.

```
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
```

Tenga en cuenta que los comandos enviados al origen de datos son **vulnerables a la inyección de SQL** , ya sea intencional o no intencional. En general, las sentencias de SQL no deben crearse concatenando entradas de usuario de ningún tipo. En su lugar, deben estar parametrizados (ver Crear comandos parametrizados).

Creando comandos parametrizados

Cada vez que el SQL ejecutado a través de una conexión ADO debe contener información del usuario, se considera la mejor práctica para parametrizarlo a fin de minimizar la posibilidad de inyección de SQL. Este método también es más legible que las concatenaciones largas y facilita un código más robusto y mantenible (es decir, mediante el uso de una función que devuelve una matriz de `Parameter`).

En la sintaxis estándar de ODBC, se dan los parámetros `?` los "marcadores de posición" en el texto de la consulta, y luego los parámetros se agregan al `Command` en el mismo orden en que aparecen en la consulta.

Tenga en cuenta que el ejemplo a continuación utiliza la función `OpenDatabaseConnection` de Hacer una conexión a una fuente de datos por brevedad.

```
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: el ejemplo anterior muestra una instrucción UPDATE parametrizada, pero a cualquier instrucción SQL se le pueden dar parámetros.

Lea Trabajando con ADO en línea: https://riptutorial.com/es/vba/topic/3578/trabajando-con-ado

Capítulo 45: Trabajar con archivos y directorios sin usar FileSystemObject

Observaciones

El `Scripting.FileSystemObject` es mucho más robusto que los métodos heredados en este tema. Se debe preferir en casi todos los casos.

Examples

Determinar si existen carpetas y archivos

Archivos:

Para determinar si existe un archivo, simplemente pase el nombre de archivo a la función `Dir$` y compruebe si devuelve un resultado. Tenga en cuenta que `Dir$` admite comodines, por lo que para probar un archivo *específico* , el nombre de `pathName` pasado se debe probar para asegurarse de que no los contiene. La muestra a continuación genera un error: si este no es el comportamiento deseado, la función se puede cambiar para que simplemente devuelva `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
```

Carpetas (método Dir $):

La función `Dir$()` también se puede usar para determinar si existe una carpeta al especificar `vbDirectory` para el parámetro de `attributes` opcional. En este caso, el pasado `pathName` valor debe terminar con un separador de ruta (`\`), como búsqueda de *nombres de archivos* hará que los falsos positivos. Tenga en cuenta que los comodines solo se permiten después del último separador de ruta, por lo que la siguiente función de ejemplo arrojará un error de tiempo de ejecución 52 - "Nombre o número de archivo incorrecto" si la entrada contiene un comodín. Si este no es el comportamiento deseado, elimine el comentario `On Error Resume Next` en la parte superior de la función. También recuerde que `Dir$` admite rutas de archivo relativas (es decir, `..\Foo\Bar`), por lo que se garantiza que los resultados solo serán válidos siempre que no se cambie el directorio de trabajo actual.

```
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
```

Carpetas (método ChDir):

La instrucción `ChDir` también se puede utilizar para probar si existe una carpeta. Tenga en cuenta que este método cambiará temporalmente el entorno en el que se está ejecutando VBA, por lo que si eso es una consideración, se debe usar el método `Dir$` su lugar. Tiene la ventaja de ser mucho menos tolerante con su parámetro. Este método también admite rutas de archivo relativas, por lo que tiene la misma advertencia que el método `Dir$` .

```
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
```

Creación y eliminación de carpetas de archivos

NOTA: para mayor brevedad, los siguientes ejemplos utilizan la función FolderExists del ejemplo de **Determinación de si existen carpetas y archivos** en este tema.

La declaración `MkDir` se puede utilizar para crear una nueva carpeta. Acepta rutas que contienen letras de unidad (`C:\Foo`), nombres UNC (`\\Server\Foo`), rutas relativas (`..\Foo`) o el directorio de trabajo actual (`Foo`).

Si se omite la unidad o el nombre UNC (es decir, `\Foo`), la carpeta se crea en la unidad actual. Este puede o no ser el mismo disco que el directorio de trabajo actual.

```
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
```

La instrucción `RmDir` se puede utilizar para eliminar carpetas existentes. Acepta rutas en las mismas formas que `MkDir` y usa la misma relación con el directorio y la unidad de trabajo actuales. Tenga en cuenta que la instrucción es similar al comando de Windows `rd` shell, por lo que arrojará

un error de tiempo de ejecución 75: "Error de acceso a la ruta / archivo" si el directorio de destino no está vacío.

```
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
```

Lea Trabajar con archivos y directorios sin usar FileSystemObject en línea: https://riptutorial.com/es/vba/topic/5706/trabajar-con-archivos-y-directorios-sin-usar-filesystemobject

Capítulo 46: VBA orientado a objetos

Examples

Abstracción

Los niveles de abstracción ayudan a determinar cuándo dividir las cosas.

La abstracción se logra mediante la implementación de la funcionalidad con un código cada vez más detallado. El punto de entrada de una macro debe ser un procedimiento pequeño con un *alto nivel de abstracción* que facilite comprender de un vistazo lo que está sucediendo:

```
Public Sub DoSomething()
    With New SomeForm
        Set .Model = CreateViewModel
        .Show vbModal
        If .IsCancelled Then Exit Sub
        ProcessUserData .Model
    End With
End Sub
```

El procedimiento `DoSomething` tiene un alto *nivel de abstracción* : podemos decir que está mostrando un formulario y creando algún modelo, y pasar ese objeto a algún procedimiento `ProcessUserData` que sabe qué hacer con él; cómo se crea el modelo es el trabajo de otro procedimiento:

```
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 función `CreateViewModel` solo es responsable de crear algunas instancias de `ISomeModel` . Parte de esa responsabilidad es adquirir una serie de *elementos disponibles* ; la forma en que se adquieren estos elementos es un detalle de implementación que se `GetAvailableItems` procedimiento `GetAvailableItems` :

```
Private Function GetAvailableItems() As Variant
    GetAvailableItems = DataSheet.Names("AvailableItems").RefersToRange
End Function
```

Aquí, el procedimiento es leer los valores disponibles de un rango con nombre en una hoja de cálculo `DataSheet` . Igualmente, podría ser leerlos desde una base de datos, o los valores podrían estar codificados: es un *detalle de la implementación* que no es motivo de preocupación para ninguno de los niveles de abstracción más altos.

Encapsulacion

La encapsulación oculta los detalles de implementación del código del cliente.

El ejemplo de Handling QueryClose muestra la encapsulación: el formulario tiene un control de casilla de verificación, pero su código de cliente no funciona directamente con ella: la casilla de verificación es un *detalle de implementación* , lo que el código de cliente debe saber es si la configuración está habilitada o no.

Cuando el valor de la casilla de verificación cambia, el controlador asigna un miembro de campo privado:

```
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
```

Y cuando el código del cliente quiere leer ese valor, no necesita preocuparse por una casilla de verificación; en su lugar, simplemente utiliza la propiedad `SomeOtherSetting` :

```
Public Property Get SomeOtherSetting() As Boolean
    SomeOtherSetting = this.SomeOtherSetting
End Property
```

La propiedad `SomeOtherSetting` *encapsula* el estado de la casilla de verificación; el código del cliente no necesita saber que hay una casilla de verificación involucrada, solo que hay una configuración con un valor booleano. Al *encapsular* el valor `Boolean` , hemos agregado una *capa de abstracción* alrededor de la casilla de verificación.

Usando interfaces para imponer la inmutabilidad.

Vayamos un paso más allá *encapsulando* el *modelo* del formulario en un módulo de clase dedicado. Pero si hiciéramos una `Public Property` para el nombre de `UserName` y la `Timestamp` , tendríamos que exponer los accesores de `Property Let` , haciendo que las propiedades sean mutables, y no queremos que el código del cliente tenga la capacidad de cambiar estos valores una vez que estén establecidos.

La función `CreateViewModel` en el ejemplo de **Abstraction** devuelve una clase de `ISomeModel` : esa es nuestra *interfaz* , y se parece a esto:

```
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
```

Observe que las propiedades de `Timestamp` y nombre de `UserName` solo exponen una `Property Get` acceso. Ahora la clase `SomeModel` puede implementar esa interfaz:

```
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
```

Los miembros de la interfaz son todos `Private` , y todos los miembros de la interfaz deben implementarse para que el código se compile. Los miembros `Public` no forman parte de la interfaz y, por lo tanto, no están expuestos a código escrito contra la interfaz de `ISomeModel` .

Usando un método de fábrica para simular un constructor

Usando un atributo VB_PredeclaredId , podemos hacer que la clase `SomeModel` tenga una *instancia*

predeterminada y escribir una función que funcione como un miembro de nivel de tipo (`Shared` en VB.NET, `static` en C #) al que el código de cliente pueda llamar sin necesidad de crear primero Una instancia, como hicimos aquí:

```
Private Function CreateViewModel() As ISomeModel
    Dim result As ISomeModel
    Set result = SomeModel.Create(Now, Environ$("UserName"))
    result.AvailableItems = GetAvailableItems
    Set CreateViewModel = result
End Function
```

Este *método de fábrica* asigna los valores de propiedad que son de solo lectura cuando se accede desde la interfaz de `ISomeModel` , aquí `Timestamp` y `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
```

Y ahora podemos codificar en contra de la `ISomeModel` interfaz, que expone `Timestamp` y `UserName` como propiedades de sólo lectura que no se pueden reasignar (siempre y cuando el código está escrito en contra de la interfaz).

Polimorfismo

El polimorfismo es la capacidad de presentar la misma interfaz para diferentes implementaciones subyacentes.

La capacidad de implementar interfaces permite desacoplar completamente la lógica de la aplicación de la interfaz de usuario, o de la base de datos, o de esta o aquella hoja de trabajo.

Supongamos que tiene una interfaz `ISomeView` que el formulario 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
```

El código subyacente del formulario podría verse así:

```
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
```

Pero entonces, nada prohíbe la creación de otro módulo de clase que implemente la interfaz `ISomeView` *sin ser un formulario de usuario* ; esto podría ser una clase `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
```

Y ahora podemos cambiar el código que funciona con un `UserForm` y hacerlo funcionar fuera de la interfaz `ISomeView` , por ejemplo, dándole el formulario como un parámetro en lugar de crear una instancia de él:

```
Public Sub DoSomething(ByVal view As ISomeView)
    With view
        Set .Model = CreateViewModel
        .Show
        If .IsCancelled Then Exit Sub
        ProcessUserData .Model
    End With
End Sub
```

Debido a que el `DoSomething` método depende de una interfaz (es decir, una *abstracción)* y no una *clase concreta* (por ejemplo, un determinado `UserForm`), podemos escribir una prueba unitaria automatizado que asegura que `ProcessUserData` no se ejecuta cuando `view.IsCancelled` es `True` , haciendo que nuestra prueba crear una instancia de `SomeViewMock` , establecer su propiedad `IsCancelled` en `True` y pasarla a `DoSomething` .

Código verificable depende de abstracciones

Se pueden hacer pruebas unitarias de escritura en VBA, hay complementos que incluso se integran en el IDE. Pero cuando el código se *combina estrechamente* con una hoja de trabajo, una base de datos, un formulario o el sistema de archivos, entonces la prueba de la unidad comienza a requerir una hoja de trabajo, una base de datos, un formulario o un sistema de archivos reales, y estas *dependencias* son nuevas fallas fuera de control puntos que el código

comprobable debe aislar, por lo que las pruebas unitarias *no* requieren una hoja de trabajo, base de datos, formulario o sistema de archivos real.

Al escribir código contra interfaces, de una manera que permite que el código de prueba *inyecte* implementaciones de código auxiliar / simulacro (como el ejemplo anterior de `SomeViewMock`), puede escribir pruebas en un "entorno controlado", y simular lo que sucede cuando cada uno de los 42 posibles permutaciones de las interacciones del usuario en los datos del formulario, sin siquiera mostrar una vez el formulario y hacer clic manualmente en un control de formulario.

Lea VBA orientado a objetos en línea: https://riptutorial.com/es/vba/topic/5357/vba-orientado-a-objetos

Creditos

S. No	Capítulos	Contributors
1	Empezando 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	Arrays	Comintern, Dave, Hubisan, jamheadart, Josan Iracheta, Maarten van Stam, Mark.R, Mat's Mug, Miguel_Ryu, Tazaf
3	Asignando cadenas con caracteres repetidos	ThunderFrame
4	Atributos	hymced, Mat's Mug, RamenChef, RubberDuck
5	Automatización o uso de otras bibliotecas de aplicaciones.	Branislav Kollár
6	Buscando dentro de las cadenas la presencia de subcadenas.	ThunderFrame
7	Clasificación	Neil Mussett
8	Colecciones	Comintern
9	Comentarios	Comintern, Hosch250, Johnny C, litelite, Macro Man, Nijin22, Shawn V. Wilson, ThunderFrame
10	Compilación condicional	Macro Man, Mat's Mug, RubberDuck, Steve Rindsberg
11	Convenciones de nombres	FreeMan, Kaz, Mat's Mug, Victor Moraes
12	Convertir otros tipos a cadenas	ThunderFrame
13	Copiando, devolviendo y pasando matrices.	Mark.R
14	Creación de una clase personalizada	Branislav Kollár, Macro Man, Mat's Mug, Neil Mussett, ThunderFrame
15	Creando un procedimiento	Comintern, LiamH, Mat's Mug, Sivaprasath Vadivel, Tomas Zubiri

16	CreateObject vs. GetObject	Branislav Kollár, Dave, Tim
17	Cuerdas de concatenacion	ThunderFrame
18	Declarando variables	Comintern, dadde, Dave, Franck Dernoncourt, Jeeped, Kaz, lfrandom, litelite, Macro Man, Mark.R, Mat's Mug, Neil Mussett, RubberDuck, Shawn V. Wilson, SWa, Thierry Dalon, ThunderFrame, Tom, Victor Moraes, Zaider
19	Declarar y asignar cadenas	Comintern, ThunderFrame
20	Errores en tiempo de ejecución de VBA	Branislav Kollár, Macro Man, Mat's Mug
21	Estructuras de control de flujo	Benno Grimm, Comintern, Kelly Tessena Keck, Leviathan, litelite, Macro Man, Martin, Mat's Mug, Roland, Siva, ThunderFrame
22	Estructuras de datos	Blackhawk
23	Eventos	Mat's Mug
24	Formularios de usuario	Mat's Mug
25	Interfaces	Neil Mussett
26	Lectura de 2GB + archivos en binario en VBA y File Hashes	PatricK
27	Literales de cadenas - Escape, caracteres no imprimibles y continuaciones de línea	Comintern, ThunderFrame
28	Llamadas API	paul bica
29	Los operadores	Comintern, Macro Man
30	Macro seguridad y firma de proyectos / módulos VBA.	0m3r
31	Manejo de errores	Comintern, Logan Reed, Mat's Mug
32	Manipulación de cuerdas de uso frecuente.	pashute

33	Manipulación de fecha y hora	Comintern, FreeMan, Thomas G
34	Midiendo la longitud de las cuerdas	Steve Rindsberg, ThunderFrame
35	Objeto Scripting.Dictionary	Comintern, Jeeped, Kyle, RamenChef, Tim, Wolf, Zev Spitz
36	Opción VBA Palabra clave	Jeeped, Maarten van Stam, Macro Man, Mat's Mug, RamenChef, RubberDuck, Stefan Pinnow, Thomas G, ThunderFrame
37	Pasando Argumentos ByRef o ByVal	Branislav Kollár, Comintern, Mat's Mug, R3uK, RamenChef, ZygD
38	Personajes no latinos	Neil Mussett
39	Procedimiento de llamadas	Macro Man, Mat's Mug, Neil Mussett, Sam Johnson
40	Recursion	Mat's Mug, ThunderFrame
41	Scripting.FileSystemObject	Comintern, Dave, Macro Man, Mikegrann, RubberDuck, Siva, Steve Rindsberg, ThunderFrame
42	Subcadenas	Mat's Mug, ThunderFrame
43	Tipos de datos y límites	Comintern, FreeMan, Neil Mussett, StackzOfZtuff, Stephen Leppik, ThunderFrame
44	Trabajando con ADO	Comintern, SandPiper, Tazaf
45	Trabajar con archivos y directorios sin usar FileSystemObject	Comintern, Macro Man, SandPiper
46	VBA orientado a objetos	IvenBach, Mat's Mug