Kostenloses eBook

LERNEN VBA

Free unaffiliated eBook created from
Stack Overflow contributors.

#vba

Inhaltsverzeichnis

Über

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

Kapitel 1: Erste Schritte mit VBA

Bemerkungen

Dieser Abschnitt bietet einen Überblick über vBA und warum ein Entwickler sie verwenden möchte.

Es sollte auch alle großen Themen in vba erwähnen und auf die verwandten Themen verweisen. Da die Dokumentation für vba neu ist, müssen Sie möglicherweise erste Versionen dieser verwandten Themen erstellen.

Versionen

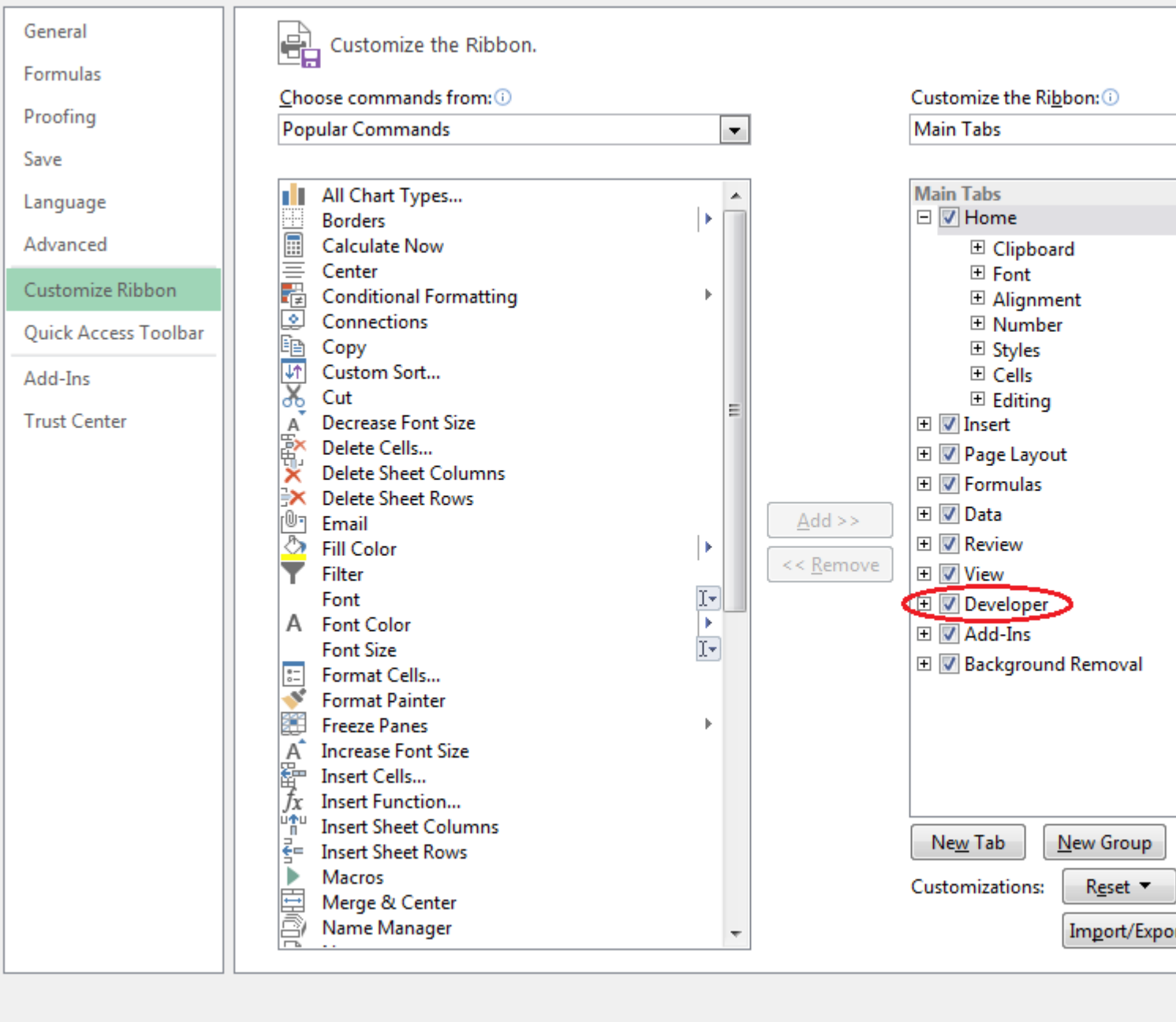
Ausführung	Office-Versionen	Veröffentlichungsdatum Hinweise	Veröffentlichungsdatum
Vba6	? - 2007	[Irgendwann danach] [1]	1992-06-30
Vba7	2010 - 2016	[blog.techkit.com] [2]	2010-04-15
VBA für Mac	2004, 2011 - 2016		2004-05-11

Examples

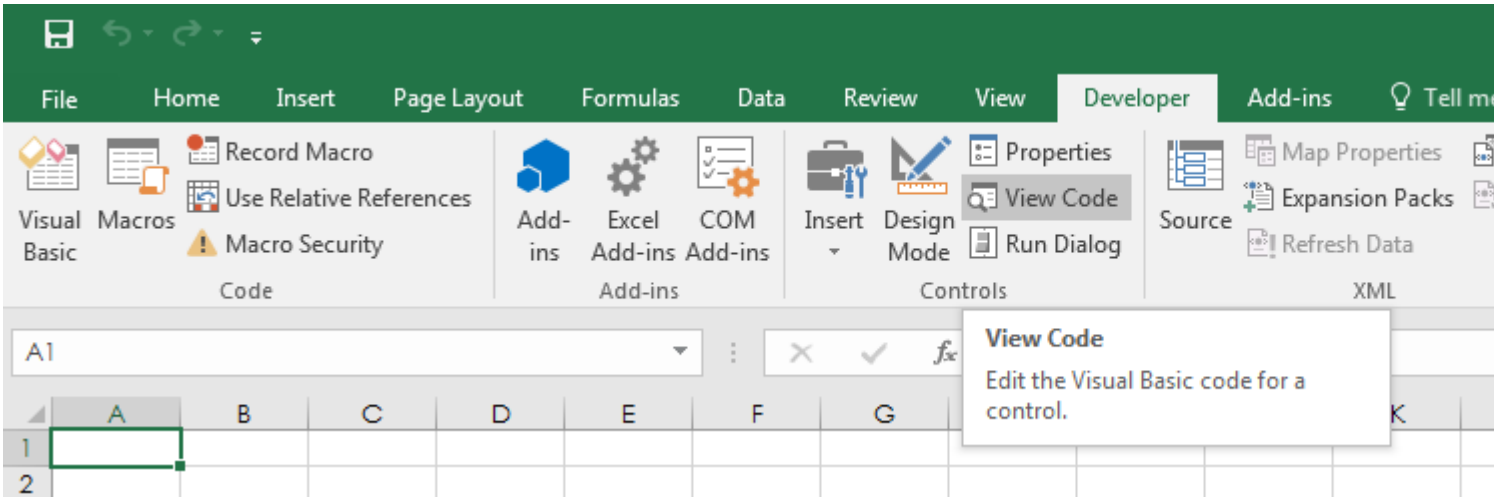
Zugriff auf den Visual Basic-Editor in Microsoft Office

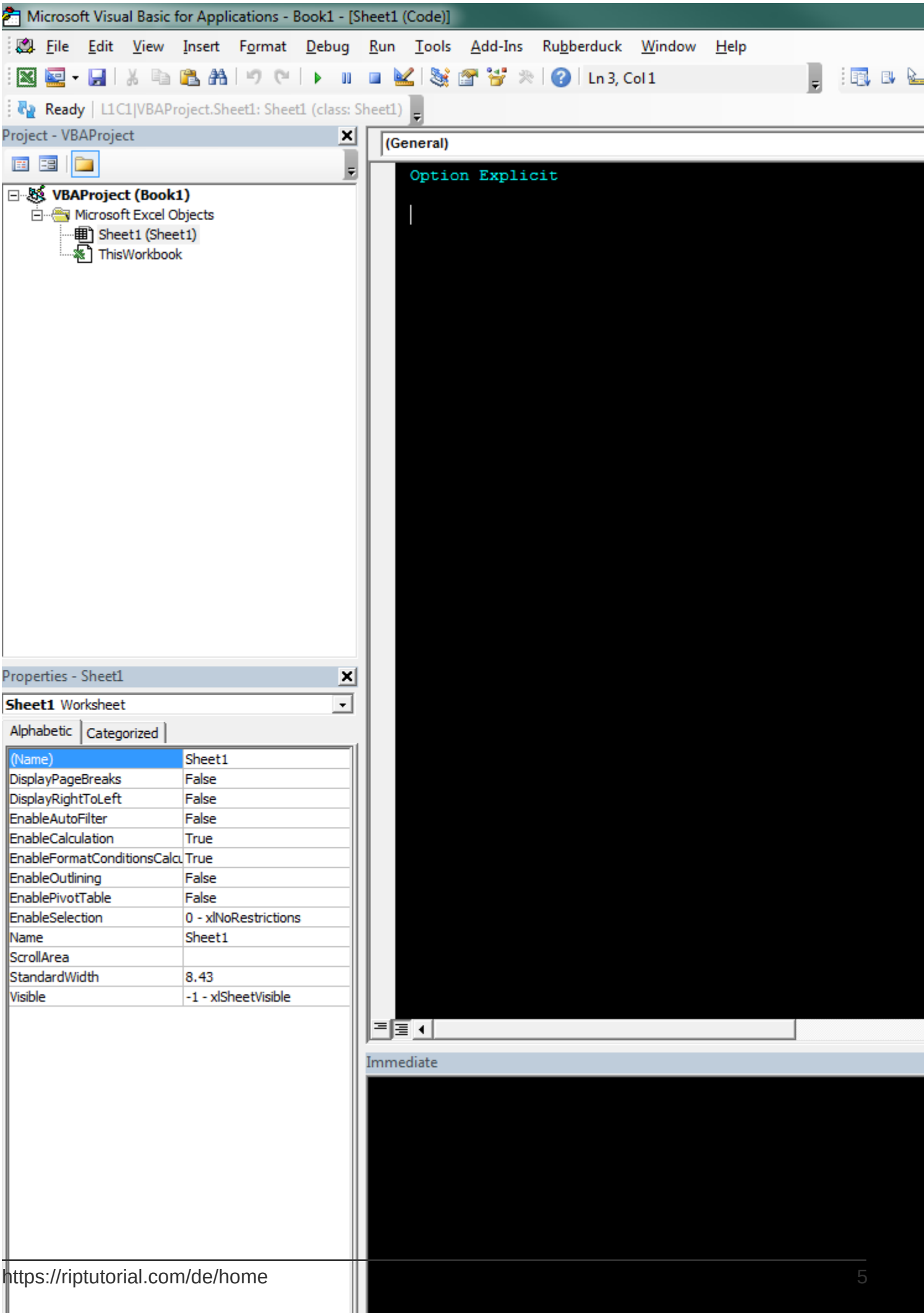
Sie können den VB-Editor in einer beliebigen Microsoft Office-Anwendung öffnen, indem Sie die Tastenkombination `Alt` + `F11` drücken oder auf die Registerkarte "Entwickler" gehen und auf die Schaltfläche "Visual Basic" klicken. Wenn die Registerkarte "Entwickler" nicht im Menüband angezeigt wird, überprüfen Sie, ob diese Option aktiviert ist.

Standardmäßig ist die Registerkarte "Entwickler" deaktiviert. Um die Registerkarte "Entwickler" zu aktivieren, gehen Sie zu "Datei -> Optionen" und wählen Sie in der Liste links die Option "Farbband anpassen". Suchen Sie in der rechten Baumansicht "Multifunktionsleiste anpassen" das Element der Entwickler-Struktur und aktivieren Sie das Kontrollkästchen "Entwickler". Klicken Sie auf OK, um das Dialogfeld "Optionen" zu schließen.



Die Registerkarte "Entwickler" ist jetzt in der Multifunktionsleiste sichtbar, auf der Sie auf "Visual Basic" klicken können, um den Visual Basic-Editor zu öffnen. Alternativ können Sie auf "Code anzeigen" klicken, um direkt den Codebereich des gerade aktiven Elements anzuzeigen, z. B. WorkSheet, Diagramm, Form.


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

Taste. Herzliche Glückwünsche! Sie haben Ihr erstes eigenes VBA-Modul gebaut.

Debuggen

Debugging ist eine sehr effektive Methode, um einen genaueren Blick auf fehlerhaft funktionierenden (oder nicht funktionierenden) Code zu werfen.

Führen Sie den Code Schritt für Schritt aus

Beim Debuggen müssen Sie zunächst den Code an bestimmten Stellen anhalten und dann Zeile für Zeile ausführen, um zu sehen, ob dies erwartungsgemäß geschieht.

- Haltepunkt (`F9` , Debug - Haltepunkt umschalten): Sie können jeder ausgeführten Zeile einen Haltepunkt hinzufügen (z. B. keine Deklarationen). Wenn die Ausführung diesen Punkt erreicht, stoppt sie und gibt dem Benutzer die Kontrolle.
- Sie können das `Stop` Schlüsselwort auch zu einer leeren Zeile hinzufügen, damit der Code zur Laufzeit an dieser Stelle angehalten wird. Dies ist nützlich, wenn Sie z. B. vor Deklarationszeilen keinen Haltepunkt mit `F9` hinzufügen können
- Step into (`F8` , Debug - Step into): Führt nur eine Codezeile aus. Wenn dies ein Aufruf einer benutzerdefinierten Funktion ist, wird dies Zeile für Zeile ausgeführt.
- Step over (`Umschalt` + `F8` , Debug - Step over): Führt eine Codezeile aus, gibt keine benutzerdefinierten Unterfunktionen ein.
- Step out (`Strg` + `Shift` + `F8` , Debug - Step out): Aktuelle Sub / Funktion beenden (Code bis zum Ende ausführen).
- Lauf zum Cursor (`Strg` + `F8` , Debug - Lauf zum Cursor): Code ausführen, bis die Zeile mit dem Cursor erreicht ist.
- Mit `Debug.Print` können `Debug.Print` zur Laufzeit Zeilen in das `Debug.Print` drucken. Sie können auch `Debug.?` als Abkürzung für `Debug.Print`

Uhrenfenster

Das Ausführen von Code Zeile für Zeile ist nur der erste Schritt. Wir müssen mehr Details kennen. Ein Werkzeug dafür ist das Watch-Fenster (View - Watch-Fenster). Hier können Sie Werte definierter Ausdrücke sehen. So fügen Sie dem Überwachungsfenster eine Variable hinzu:

- Klicken Sie mit der rechten Maustaste darauf und wählen Sie "Uhr hinzufügen".
- Klicken Sie mit der rechten Maustaste in das Watch-Fenster und wählen Sie "Watch hinzufügen".
- Gehe zu Debuggen - Watch hinzufügen.

Wenn Sie einen neuen Ausdruck hinzufügen, können Sie wählen, ob Sie nur den Wert anzeigen möchten oder die Codeausführung unterbrechen, wenn sie wahr ist oder wenn sich der Wert ändert.

Sofortiges Fenster

Im unmittelbaren Fenster können Sie beliebigen Code ausführen oder Elemente drucken, indem Sie ihnen entweder das Schlüsselwort " `Print` oder ein einzelnes Fragezeichen " `?` "

Einige Beispiele:

- `? ActiveSheet.Name` - gibt den Namen des aktiven Blattes zurück
- `Print ActiveSheet.Name` - gibt den Namen des aktiven Blattes zurück
- `? foo` - gibt den Wert von `foo` *
- `x = 10` Sätze `x` bis 10 *

* Das Abrufen / Setzen von Werten für Variablen über das Direktfenster ist nur zur Laufzeit möglich

Best Practices für das Debuggen

Wann immer Ihr Code nicht wie erwartet funktioniert, sollten Sie ihn zuerst sorgfältig lesen und nach Fehlern suchen.

Wenn das nicht hilft, beginnen Sie mit dem Debuggen. Bei kurzen Prozeduren kann es effizient sein, die Zeile nur Zeile für Zeile auszuführen. Bei längeren Prozeduren müssen Sie wahrscheinlich Haltepunkte oder Unterbrechungen für überwachte Ausdrücke festlegen. Das Ziel hier ist es, die Zeile nicht wie erwartet zu finden.

Wenn Sie die Zeile erhalten haben, die das falsche Ergebnis liefert, der Grund jedoch noch nicht klar ist, versuchen Sie, Ausdrücke zu vereinfachen oder Variablen durch Konstanten zu ersetzen. Dies hilft dabei zu verstehen, ob der Wert der Variablen falsch ist.

Wenn Sie es immer noch nicht lösen können, bitten Sie um Hilfe:

- Fügen Sie so wenig Code wie möglich ein, um Ihr Problem zu verstehen
- Wenn das Problem nicht mit dem Wert von Variablen zusammenhängt, ersetzen Sie sie durch Konstanten. (also anstelle von `Sheets(a*b*c+d^2).Range(addressOfRange)` `Sheets(4).Range("A2")` schreiben `Sheets(4).Range("A2")`)
- Beschreiben Sie, welche Zeile das falsche Verhalten angibt und was es ist (Fehler, falsches Ergebnis ...).

Erste Schritte mit VBA online lesen: https://riptutorial.com/de/vba/topic/802/erste-schritte-mit-vba

Kapitel 2: 2GB + -Dateien in binärer Form in VBA und File Hashes lesen

Einführung

Es gibt eine einfache Möglichkeit, Dateien in binärer Form in VBA zu lesen, es besteht jedoch eine Einschränkung von 2 GB (2.147.483.647 Bytes - max des Datentyps Long). Wenn sich die Technologie weiterentwickelt, wird diese Grenze von 2 GB leicht überschritten. zB ein ISO-Image der Betriebssystem-DVD. Microsoft bietet eine Möglichkeit, dies durch eine Windows-API auf niedriger Ebene zu beheben, und hier ist eine Sicherungskopie davon.

Demonstrieren Sie (Lese-Teil) auch die Berechnung von Datei-Hashes ohne externes Programm wie `fciv.exe` von Microsoft.

Bemerkungen

METHODEN FÜR DIE KLASSE VON MICROSOFT

Methodenname	Beschreibung
Ist offen	Gibt einen Booleschen Wert zurück, um anzuzeigen, ob die Datei geöffnet ist.
OpenFile (*sFileNam* As String)	Öffnet die durch das Argument sFileName angegebene Datei.
Datei schließen	Schließt die aktuell geöffnete Datei.
ReadBytes (*ByteCount* As Long)	Liest ByteCount-Bytes und gibt sie in einem Variant-Byte-Array zurück und verschiebt den Zeiger.
WriteBytes (*DataBytes ()* As Byte)	Schreibt den Inhalt des Bytearrays an die aktuelle Position in der Datei und verschiebt den Zeiger.
Spülen	Zwingt Windows, den Schreibcache zu leeren.
SeekAbsolute (*HighPos* so lang, *LowPos* so lang)	Verschiebt den Dateizeiger vom Anfang der Datei an die angegebene Position. Obwohl die DWORDS von VBA als signierte Werte behandelt werden, werden sie von der API als nicht signiert behandelt. Stellen Sie sicher, dass das Argument höherer Ordnung 4 GB überschreitet. Das niederwertige DWORD ist für Werte zwischen 2 GB und 4 GB negativ.
SeekRelative (	Verschiebt den Dateizeiger vom aktuellen Speicherort auf +/- 2 GB.

Methodenname	Beschreibung
Offset so lange)	Sie können diese Methode umschreiben, um Offsets von mehr als 2 GB zu berücksichtigen, indem Sie einen 64-Bit-Vorzeichenversatz in zwei 32-Bit-Werte konvertieren.

EIGENSCHAFTEN DER KLASSE VON MICROSOFT

Eigentum	Beschreibung
FileHandle	Das Dateihandle für die aktuell geöffnete Datei. Dies ist nicht kompatibel mit VBA-Dateihandles.
Dateiname	Der Name der aktuell geöffneten Datei.
AutoFlush	Legt fest, ob WriteBytes die Flush-Methode automatisch aufruft.

NORMALMODUL

Funktion	Anmerkungen
GetFileHash (*sFile* As String, *uBlockSize* As Double, *sHashType* As String)	Geben Sie einfach den vollständigen Pfad ein, der gehasht werden soll, verwenden Sie Blocksize (Anzahl Byte) und den Typ des zu verwendenden Hash - eine der privaten Konstanten: **HashTypeMD5** , **HashTypeSHA1** , **HashTypeSHA256** , **HashTypeSHA384** , **HashTypeSHA512** . Dies sollte so generisch wie möglich sein.

Sie sollten **uFileSize As Double** entsprechend **dekomprimieren** . Ich habe MD5 und SHA1 getestet.

Examples

Dies muss in einem Klassenmodul sein, Beispiele werden später als "Random" bezeichnet.

```
' 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
```

Code zum Berechnen von Datei-Hash in einem Standardmodul

```
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
```

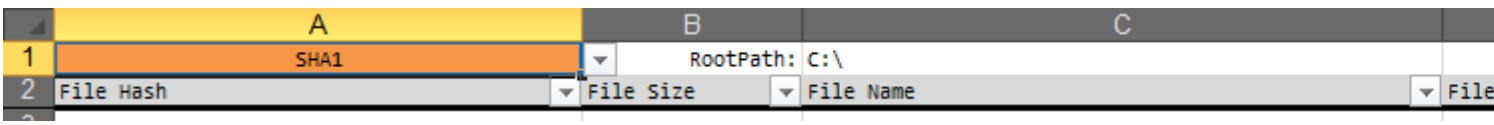
Die Ausgabe ist ziemlich interessant, meine Testdateien zeigen an, dass `BLOCKSIZE = 131071` **(2 ^ 17-1)** mit 32-Bit-Office 2010 unter Windows 7 x64 insgesamt die beste Leistung liefert, die zweitbeste ist *2 ^ 16-1 (65535)* . Hinweis `2^27-1` *zu wenig Speicher* .

Dateigröße (Bytes)	Dateiname
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_German.ISO
4.380.387.328	Win10_1607_German_x64.iso
5.400.115.200	SW_DVD9_Windows_Svr_Std_and_DataCtr_2012_R2_64Bit_English.ISO

Berechnung aller Dateien-Hash aus einem Stammordner

Eine andere Variante des obigen Codes bietet mehr Leistung, wenn Sie Hash-Codes aller Dateien aus einem Stammordner einschließlich aller Unterordner abrufen möchten.

Beispiel eines Arbeitsblatts:



Code

```
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
```

2GB + -Dateien in binärer Form in VBA und File Hashes lesen online lesen: https://riptutorial.com/de/vba/topic/8786/2gb-plus--dateien-in-binarer-form-in-vba-und-file-hashes-lesen

Kapitel 3: Andere Typen in Strings konvertieren

Bemerkungen

VBA konvertiert einige Typen implizit in eine Zeichenfolge, ohne dass zusätzliche Programmierarbeiten erforderlich sind. VBA bietet jedoch auch eine Reihe expliziter Zeichenfolgenkonvertierungsfunktionen. Sie können auch eigene Typen schreiben.

Drei der am häufigsten verwendeten Funktionen sind `CStr` , `Format` und `StrConv` .

Examples

Verwenden Sie CStr, um einen numerischen Typ in eine Zeichenfolge zu konvertieren

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

Verwenden Sie Format, um einen numerischen Typ als Zeichenfolge zu konvertieren und zu formatieren

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

Verwenden Sie StrConv, um ein Byte-Array aus Einzelbyte-Zeichen in eine Zeichenfolge zu konvertieren

```
'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"
```

Konvertieren Sie implizit ein Byte-Array mit Mehrbyte-Zeichen in einen String

```
'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"
```

Andere Typen in Strings konvertieren online lesen:
https://riptutorial.com/de/vba/topic/3467/andere-typen-in-strings-konvertieren

Kapitel 4: API-Aufrufe

Einführung

API steht für Application Programming Interface

APIs für VBA beinhalten eine Reihe von Methoden, die eine direkte Interaktion mit dem Betriebssystem ermöglichen

Systemaufrufe können durch Ausführen von in DLL-Dateien definierten Prozeduren ausgeführt werden

Bemerkungen

Übliche Bibliotheksdateien für die Betriebsumgebung (DLLs):

Dynamische Link Bibliothek	Beschreibung
Advapi32.dll	Erweiterte Servicebibliothek für APIs, einschließlich vieler Sicherheits- und Registrierungsaufrufe
Comdlg32.dll	Allgemeine Dialog-API-Bibliothek
Gdi32.dll	API-Bibliothek für Grafikgeräteschnittstellen
Kernel32.dll	Grundlegende 32-Bit-API-Unterstützung für Windows
Lz32.dll	32-Bit-Komprimierungsroutinen
Mpr.dll	Router-Bibliothek für mehrere Anbieter
Netapi32.dll	32-Bit-Netzwerk-API-Bibliothek
Shell32.dll	32-Bit-Shell-API-Bibliothek
User32.dll	Bibliothek für Benutzeroberflächenroutinen
Version.dll	Versionsbibliothek
Winmm.dll	Windows-Multimedia-Bibliothek
Winspool.drv	Schnittstelle des Druckerspoolers, die die Aufrufe der Druckerspooler-API enthält

Neue Argumente für das 64-System:

Art	Artikel	Beschreibung
Qualifikation	PtrSafe	Gibt an, dass die Declare-Anweisung mit 64 Bit kompatibel ist. Dieses Attribut ist auf 64-Bit-Systemen obligatorisch
Datentyp	LongPtr	Ein variabler Datentyp, der bei 32-Bit-Versionen ein 4-Byte-Datentyp ist, und bei 64-Bit-Versionen von Office 2010 ein 8-Byte-Datentyp. Dies ist die empfohlene Methode, um einen Zeiger oder Handle für neuen Code zu deklarieren für älteren Code, wenn er in der 64-Bit-Version von Office 2010 ausgeführt werden muss. Er wird nur in der VBA 7-Laufzeitumgebung auf 32-Bit und 64-Bit unterstützt. Beachten Sie, dass Sie ihm numerische Werte zuweisen können, jedoch keine numerischen Typen
Datentyp	Lang Lang	Dies ist ein 8-Byte-Datentyp, der nur in 64-Bit-Versionen von Office 2010 verfügbar ist. Sie können numerische Werte zuweisen, jedoch keine numerischen Typen (um das Abschneiden zu vermeiden).
Umwandlung	Operator	CLngPtr Konvertiert einen einfachen Ausdruck in einen LongPtr-Datentyp
Umwandlung	Operator	CLngLng Konvertiert einen einfachen Ausdruck in einen LongLong-Datentyp
Funktion	VarPtr	Variantenkonverter. Gibt einen LongPtr für 64-Bit-Versionen und einen Long für 32-Bit (4 Byte) zurück.
Funktion	ObjPtr	Objektkonverter Gibt einen LongPtr für 64-Bit-Versionen und einen Long für 32-Bit (4 Byte) zurück.
Funktion	StrPtr	String-Konverter Gibt einen LongPtr für 64-Bit-Versionen und einen Long für 32-Bit (4 Byte) zurück.

Vollständige Referenz der Anrufsignaturen:

- Win32api32.txt für Visual Basic 5.0 (alte API-Deklarationen, letzte Überprüfung im März 2005 von Microsoft)
- Win32API_PtrSafe mit 64-Bit-Unterstützung (Office 2010, Microsoft)

Examples

API-Deklaration und -Verwendung

So deklarieren Sie eine DLL-Prozedur , um mit verschiedenen VBA-Versionen zu arbeiten:

```
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
```

Die obige Deklaration teilt VBA mit, wie die in der Datei Kernel32.dll definierte Funktion "Sleep" aufgerufen wird

Win64 und Win32 sind vordefinierte Konstanten für die bedingte Kompilierung

---

Vordefinierte Konstanten

Einige Kompilierungskonstanten sind bereits vordefiniert. Welche davon existieren, hängt von der Bitness der Office-Version ab, in der Sie VBA ausführen. Beachten Sie, dass Vba7 neben Office 2010 zur Unterstützung von 64-Bit-Versionen von Office eingeführt wurde.

Konstante	16 bit	32 bit	64 bit
Vba6	Falsch	Wenn Vba6	Falsch
Vba7	Falsch	Wenn Vba7	Wahr
Win16	Wahr	Falsch	Falsch
Win32	Falsch	Wahr	Wahr
Win64	Falsch	Falsch	Wahr
Mac	Falsch	Wenn Mac	Wenn Mac

Diese Konstanten beziehen sich auf die Office-Version, nicht auf die Windows-Version. Beispiel: Win32 = TRUE in 32-Bit-Office, auch wenn das Betriebssystem eine 64-Bit-Version von Windows ist.

Der Hauptunterschied bei der Deklaration von APIs liegt zwischen 32-Bit- und 64-Bit-Office-Versionen, wodurch neue Parametertypen eingeführt wurden (weitere Informationen finden Sie im Abschnitt "Anmerkungen").

Anmerkungen:

- Deklarationen werden am oberen Rand des Moduls und außerhalb von Subs oder Funktionen platziert
- In Standardmodulen deklarierte Prozeduren sind standardmäßig öffentlich

- Um eine Prozedur für ein Modul als privat zu deklarieren, muss vor der Deklaration das Schlüsselwort `Private`
- In einem anderen Modultyp deklarierte DLL-Prozeduren sind für dieses Modul privat

---

Einfaches Beispiel für den Aufruf der Schlaf-API:

```
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
```

Es wird empfohlen, ein dediziertes API-Modul zu erstellen, um den einfachen Zugriff auf die Systemfunktionen von VBA-Wrappern zu ermöglichen - normalen VBA-Subs oder -Funktionen, die die für den eigentlichen Systemaufruf erforderlichen Details enthalten, z. B. in Bibliotheken verwendete Parameter und die Initialisierung dieser Parameter

Das Modul kann alle Deklarationen und Abhängigkeiten enthalten:

- Methodensignaturen und erforderliche Datenstrukturen
- Wrapper, die eine Eingabevalidierung durchführen und sicherstellen, dass alle Parameter wie erwartet übergeben werden

---

`Declare` dem Deklarationsabschnitt des `Declare` eine `Declare` Anweisung hinzu, um eine DLL-Prozedur zu deklarieren.

Wenn die Prozedur einen Wert zurückgibt, deklarieren Sie ihn als **Funktion** :

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

Wenn eine Prozedur keinen Wert zurückgibt, deklarieren Sie ihn als **Sub** :

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

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

  Bemerkenswert ist auch, dass die **meisten ungültigen Aufrufe der API Excel zum**

Absturz bringen und möglicherweise Datendateien beschädigen

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

Office 2011 für 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
```

Die folgenden Beispiele (Windows API - Dedicated Module (1 und 2)) zeigen ein API-Modul, das allgemeine Deklarationen für Win64 und Win32 enthält

Windows API - dediziertes Modul (1 von 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
```

(Fortsetzung im zweiten Beispiel)

Windows API - dediziertes Modul (2 von 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
```

Mac-APIs

Microsoft unterstützt APIs nicht offiziell, aber mit etwas Recherche können mehr Erklärungen online gefunden werden

Office 2016 für Mac ist in einer Sandbox

Im Gegensatz zu anderen Versionen von Office-Apps, die VBA unterstützen, sind Office 2016 für Mac-Apps in Sandbox-Formaten.

Durch Sandboxing wird verhindert, dass Apps auf Ressourcen außerhalb des App-Containers zugreifen. Dies betrifft alle Add-Ins oder Makros, die den Dateizugriff oder die Kommunikation zwischen Prozessen erfordern. Sie können die Auswirkungen des Sandboxing minimieren, indem

Sie die neuen Befehle verwenden, die im folgenden Abschnitt beschrieben werden. Neue VBA-Befehle für Office 2016 für Mac

Die folgenden VBA-Befehle sind neu und einzigartig in Office 2016 für Mac.

Befehl	Verwenden zu
GrantAccessToMultipleFiles	Fordern Sie die Berechtigung eines Benutzers an, auf mehrere Dateien gleichzeitig zuzugreifen
AppleScriptTask	Rufen Sie externe AppleScript-Skripts von VB auf
MAC_OFFICE_VERSION	IFDEF zwischen verschiedenen Mac Office-Versionen zur Kompilierzeit

---

Office 2011 für 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 für 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
```

Erhalten Sie Gesamtmonitore und Bildschirmauflösung

```
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- und regionale APIs

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
```

API-Aufrufe online lesen: https://riptutorial.com/de/vba/topic/10569/api-aufrufe

Kapitel 5: Arbeiten mit Dateien und Verzeichnissen ohne Verwendung von FileSystemObject

Bemerkungen

Das `Scripting.FileSystemObject` ist viel robuster als die in diesem Thema verwendeten Legacy-Methoden. Es sollte in fast allen Fällen bevorzugt werden.

Examples

Bestimmen, ob Ordner und Dateien vorhanden sind

Dateien:

Um festzustellen, ob eine Datei existiert, übergeben Sie einfach den Dateinamen an die Funktion `Dir$` und testen Sie, ob ein Ergebnis zurückgegeben wird. Beachten Sie, dass `Dir$ pathName` zu prüfen, ob eine *bestimmte* Datei `pathName` sollte der übergebene `pathName` getestet werden, um sicherzustellen, dass er keine `pathName` enthält. Das folgende Beispiel gibt einen Fehler aus. Wenn dies nicht das gewünschte Verhalten ist, kann die Funktion geändert werden, um einfach `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
```

Ordner (Dir $ -Methode):

Die Funktion `Dir$()` kann auch verwendet werden, um zu ermitteln, ob ein Ordner vorhanden ist, indem `vbDirectory` für den optionalen `attributes` angegeben wird. In diesem Fall muss der übergebene `pathName` Wert mit einem `pathName` (`\`) enden, da übereinstimmende *Dateinamen* zu falschen `pathName` führen. Beachten Sie, dass Platzhalterzeichen nur nach dem letzten Pfadtrennzeichen zulässig sind. Die Beispielfunktion unten verursacht einen Laufzeitfehler 52 - "Ungültiger Dateiname oder -nummer", wenn die Eingabe einen Platzhalter enthält. Wenn dies nicht das gewünschte Verhalten ist, entfernen `On Error Resume Next` Kommentar `On Error Resume Next` oben in der Funktion. Denken Sie auch daran, dass `Dir$` relative Dateipfade unterstützt (z. B. `..\Foo\Bar`). Daher sind die Ergebnisse nur garantiert gültig, solange das aktuelle Arbeitsverzeichnis nicht geändert wird.

```
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
```

Ordner (ChDir-Methode):

Die `ChDir` Anweisung kann auch verwendet werden, um zu testen, ob ein Ordner vorhanden ist. Beachten Sie, dass diese Methode die Umgebung, in der VBA ausgeführt wird, vorübergehend ändert. Wenn dies eine Überlegung ist, sollte stattdessen die `Dir$` -Methode verwendet werden. Es hat den Vorteil, dass es mit seinen Parametern viel weniger nachsichtig ist. Diese Methode unterstützt auch relative Dateipfade, hat also die gleiche Einschränkung wie die `Dir$` -Methode.

```
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
```

Dateiordner erstellen und löschen

HINWEIS: Der Kürze halber verwenden die folgenden Beispiele die Funktion FolderExists aus dem Beispiel zum **Bestimmen, ob Ordner und Dateien** in diesem Thema vorhanden sind.

Die `MkDir` Anweisung kann zum Erstellen eines neuen Ordners verwendet werden. Es akzeptiert Pfade mit Laufwerksbuchstaben (`C:\Foo`), UNC-Namen (`\\Server\Foo`), relativen Pfaden (`..\Foo`) oder dem aktuellen Arbeitsverzeichnis (`Foo`).

Wenn das Laufwerk oder der UNC-Name nicht angegeben wird (dh `\Foo`), wird der Ordner auf dem aktuellen Laufwerk erstellt. Dies ist möglicherweise das gleiche Laufwerk wie das aktuelle Arbeitsverzeichnis.

```
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
```

Die `RmDir` Anweisung kann verwendet werden, um vorhandene Ordner zu löschen. Es akzeptiert Pfade in den gleichen Formularen wie `MkDir` und verwendet dieselbe Beziehung zum aktuellen Arbeitsverzeichnis und Laufwerk. Beachten Sie, dass die Anweisung dem Windows-Befehl `rd` shell ähnelt. `rd` wird ein Laufzeitfehler 75 ausgegeben: "Pfad / Dateizugriffsfehler", wenn das Zielverzeichnis nicht leer ist.

```
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
```

Arbeiten mit Dateien und Verzeichnissen ohne Verwendung von FileSystemObject online lesen: https://riptutorial.com/de/vba/topic/5706/arbeiten-mit-dateien-und-verzeichnissen-ohne-verwendung-von-filesystemobject

Kapitel 6: Argumente übergeben ByRef oder ByVal

Einführung

Die Modifizierer `ByRef` und `ByVal` sind Teil der Signatur einer Prozedur und geben an, wie ein Argument an eine Prozedur übergeben wird. In VBA wird ein Parameter an `ByRef` sofern nichts anderes angegeben ist (dh `ByRef` ist implizit, falls nicht vorhanden).

Hinweis In vielen anderen Programmiersprachen (einschließlich VB.NET) werden Parameter implizit durch einen Wert übergeben, wenn kein Modifikator angegeben ist. Sie `ByRef` explizit `ByRef` Modifikatoren angeben, um mögliche Verwirrungen zu vermeiden.

Bemerkungen

Arrays übergeben

Arrays **müssen** als Referenz übergeben werden. Dieser Code wird kompiliert, verursacht jedoch den Laufzeitfehler 424 "Object Required":

```
Public Sub Test()
    DoSomething Array(1, 2, 3)
End Sub

Private Sub DoSomething(ByVal foo As Variant)
    foo.Add 42
End Sub
```

Dieser Code kompiliert nicht:

```
Private Sub DoSomething(ByVal foo() As Variant) 'ByVal is illegal for arrays
    foo.Add 42
End Sub
```

Examples

Einfache Variablen übergeben ByRef und ByVal

Übergeben von `ByRef` oder `ByVal` gibt an, ob der tatsächliche Wert eines Arguments von `CalledProcedure` an `CallingProcedure` wird oder ob eine Referenz (in anderen Sprachen als Zeiger bezeichnet) an `CalledProcedure` .

Wenn ein Argument `ByRef` , wird die Speicheradresse des Arguments an die `CalledProcedure` und alle Änderungen an diesem Parameter durch die `CalledProcedure` an dem Wert in der `CallingProcedure` .

Wenn ein Argument `ByVal` , wird der tatsächliche Wert, nicht ein Verweis auf die Variable, an die `CalledProcedure` .

Ein einfaches Beispiel veranschaulicht dies deutlich:

```
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
```

Ein anderes Beispiel:

```
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

Standardmodifikator

Wenn für einen Parameter kein Modifikator angegeben ist, wird dieser Parameter implizit als Referenz übergeben.

```
Public Sub DoSomething1(foo As Long)
End Sub
```

```
Public Sub DoSomething2(ByRef foo As Long)
End Sub
```

Der `foo` Parameter übergeben wird `ByRef` sowohl in `DoSomething1` und `DoSomething2` .

> **Achtung!** Wenn Sie mit Erfahrungen aus anderen Sprachen zu VBA kommen, ist dies sehr wahrscheinlich das genaue Gegenteil von dem, das Sie gewohnt sind. In vielen anderen Programmiersprachen (einschließlich VB.NET) übergibt der implizite / default-Modifikator Parameter nach Wert.

Übergabe als Referenz

- Wenn ein *Wert* `ByRef` , erhält die Prozedur **einen Verweis** auf den Wert.

```
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
```

Der Aufruf der oben `Test` - Prozedur Ausgänge 84. `DoSomething` ist gegeben `foo` und erhält einen *Verweis* auf den Wert, und deshalb arbeitet mit der gleichen Speicheradresse wie der Aufrufer.

- Wenn eine *Referenz* `ByRef` , erhält die Prozedur **eine Referenz** auf den Zeiger.

```
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
```

Der obige Code verursacht einen Laufzeitfehler 91 , da der Aufrufer das `Count DoSomething` eines Objekts aufruft, das nicht mehr vorhanden ist, weil `DoSomething` vor dem Zurückgeben einen *Verweis* auf den Objektzeiger erhalten und diesem `Nothing` zugewiesen hat.

ByVal am Anrufort erzwingen

Mit Klammern an der Aufrufstelle können Sie `ByRef` überschreiben und die `ByRef` eines Arguments für `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
```

Der obige Code gibt 42 aus, unabhängig davon, ob `ByRef` implizit oder explizit angegeben wird.

> **Achtung!** Aus diesem Grund kann die Verwendung fremder Klammern in Prozeduraufrufen leicht zu Fehlern führen. Achten Sie auf den Leerraum zwischen dem Prozedurnamen und der Argumentliste:
>
> ```
> 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

Übergabe nach Wert

- Wenn ein *Wert* `ByVal` , erhält die Prozedur **eine Kopie** des Werts.

  ```
  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
  ```

  Der Aufruf der oben `Test` - Prozedur Ausgänge 42. `DoSomething` ist gegeben `foo` und erhält **eine Kopie** des Wertes. Die Kopie wird mit 2 multipliziert und dann verworfen, wenn die Prozedur beendet wird. Die Kopie des Anrufers wurde nie geändert.

- Wenn eine *Referenz* `ByVal` , erhält die Prozedur **eine Kopie** des Zeigers.

  ```
  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
```

Aufrufen der obigen `Test` - Verfahren 1. Ausgänge `DoSomething` gegeben `foo` und erhält *eine Kopie* **des Zeigers** auf die `Collection` Objekt. Da die `foo` Objektvariable in den `Test` - Umfang Punkte auf das gleiche Objekt, ein Element in das Hinzufügen `DoSomething` fügt das Element auf das gleiche Objekt. Da es sich um *eine Kopie* des Zeigers handelt, wirkt sich die Einstellung des Verweises auf `Nothing` nicht auf die eigene Kopie des Aufrufers aus.

Argumente übergeben ByRef oder ByVal online lesen:
https://riptutorial.com/de/vba/topic/7363/argumente-ubergeben-byref-oder-byval

Kapitel 7: Arrays

Examples

Ein Array in VBA deklarieren

Das Deklarieren eines Arrays ist dem Deklarieren einer Variablen sehr ähnlich, mit der Ausnahme, dass Sie die Dimension des Arrays direkt nach seinem Namen deklarieren müssen:

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

Standardmäßig werden Arrays in VBA **von NULL indiziert.** Daher bezieht sich die Zahl in Klammern nicht auf die Größe des Arrays, sondern auf **den Index des letzten Elements**

Zugriff auf Elemente

Der Zugriff auf ein Element des Arrays erfolgt über den Namen des Arrays, gefolgt vom Index des Elements in Klammern:

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

Array-Indizierung

Sie können die Indizierung der Arrays ändern, indem Sie diese Zeile oben in einem Modul platzieren:

```
Option Base 1
```

Mit dieser Zeile werden alle im Modul deklarierten Arrays **von ONE indiziert** .

Spezifischer Index

Sie können jedes Array auch mit seinem eigenen Index deklarieren, indem Sie das `To` Schlüsselwort und die Unter- und Obergrenze (= Index) verwenden:

```
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
```

Dynamische Deklaration

Wenn Sie die Größe Ihres Arrays vor der Deklaration nicht kennen, können Sie die dynamische

Deklaration und das Schlüsselwort `ReDim` :

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

Beachten Sie, dass mit dem Schlüsselwort `ReDim` alle vorherigen Inhalte Ihres Arrays `ReDim` werden. Um dies zu verhindern, können Sie nach `ReDim` das Schlüsselwort `Preserve 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
```

Verwendung von Split zum Erstellen eines Arrays aus einer Zeichenfolge

Split-Funktion

Gibt ein nullbasiertes, eindimensionales Array zurück, das eine angegebene Anzahl von Teilzeichenfolgen enthält.

Syntax

Split (Ausdruck [, **Trennzeichen** [, **Limit** [, **vergleichen**]]] **)**

Teil	Beschreibung
Ausdruck	Erforderlich. Zeichenfolgenausdruck, der Teilzeichenfolgen und Trennzeichen enthält. Wenn *expression* eine Zeichenfolge der Länge Null ist ("" oder vbNullString), gibt **Split** ein leeres Array zurück, das keine Elemente und keine Daten enthält. In diesem Fall hat das zurückgegebene Array eine LBound von 0 und eine UBound von -1.
Trennzeichen	Wahlweise. Zeichenfolge-Zeichen, das zum Identifizieren von Teilzeichenfolgengrenzen Wenn nicht angegeben, wird das Leerzeichen ("") als Trennzeichen angenommen. Wenn ***Trennzeichen*** eine Zeichenfolge der Länge Null ist, wird ein Array mit einem einzelnen Element zurückgegeben, das die gesamte ***Ausdruckszeichenfolge*** enthält.
Grenze	Wahlweise. Anzahl der zurückzugebenden Teilzeichenfolgen; -1 gibt an, dass alle Teilzeichenfolgen zurückgegeben werden.
vergleichen Sie	Wahlweise. Numerischer Wert, der angibt, welche Art von Vergleich bei der Auswertung von Teilzeichenfolgen verwendet werden soll. Siehe Abschnitt Einstellungen für Werte.

die Einstellungen

Das ***Compare-*** Argument kann folgende Werte annehmen:

Konstante	Wert	Beschreibung
Beschreibung	-1	Führt einen Vergleich mit der Einstellung der **Option Compare-** Anweisung durch.
vbBinaryCompare	0	Führt einen binären Vergleich durch.
vbTextCompare	1	Führt einen Textvergleich durch.
vbDatabaseCompare	2	Nur Microsoft Access. Führt einen Vergleich basierend auf Informationen in Ihrer Datenbank durch.

Beispiel

In diesem Beispiel wird gezeigt, wie Split durch das Anzeigen mehrerer Stile funktioniert. In den Kommentaren wird die Ergebnismenge für die verschiedenen durchgeführten Split-Optionen angezeigt. Schließlich wird gezeigt, wie das zurückgegebene String-Array durchlaufen wird.

```
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
```

Elemente eines Arrays iterieren

Fürs nächste

Die Iterator-Variable als Indexnummer ist der schnellste Weg, um die Elemente eines Arrays zu iterieren:

```
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
```

Verschachtelte Schleifen können verwendet werden, um mehrdimensionale Arrays zu iterieren:

```
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
```

Für jeden ... Weiter

A `For Each...Next` Schleife kann auch zum Durchlaufen von Arrays verwendet werden, wenn die Leistung keine Rolle spielt:

```
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` Schleife durchläuft alle Dimensionen von außen nach innen (die Reihenfolge, in der die Elemente im Speicher angeordnet sind), sodass keine geschachtelten Schleifen erforderlich sind:

```
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
```

Beachten Sie, dass `For Each` Schleifen am besten zum Durchlaufen von `Collection` Objekten verwendet werden, wenn die Leistung von Bedeutung ist.

Alle 4 Ausschnitte oben erzeugen die gleiche Ausgabe:

```
 0
 1
 2
 3
```

Dynamische Arrays (Größenanpassung von Arrays und dynamisches Handling)

Dynamische Arrays

Das dynamische Hinzufügen und Reduzieren von Variablen in einem Array ist ein großer Vorteil, wenn die von Ihnen behandelten Informationen nicht über eine bestimmte Anzahl von Variablen verfügen.

Werte dynamisch hinzufügen

Sie können das Array einfach mit der `ReDim` Anweisung in der Größe ändern. `ReDim` wird das Array jedoch verkleinert. Wenn Sie jedoch die Informationen beibehalten `ReDim` , die bereits im Array gespeichert sind, benötigen Sie den Teil `Preserve` .

Im folgenden Beispiel erstellen wir ein Array und erhöhen es in jeder Iteration um eine weitere Variable, während die Werte, die sich bereits im Array befinden, beibehalten werden.

```
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
```

Werte dynamisch entfernen

Wir können dieselbe Logik verwenden, um das Array zu verkleinern. Im Beispiel wird der Wert "last" aus dem Array entfernt.

```
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
```

Array zurücksetzen und dynamisch wiederverwenden

Wir können die von uns erstellten Arrays genauso gut wiederverwenden, als nicht viele im Speicher vorhanden sind, was die Laufzeit verlangsamen würde. Dies ist nützlich für Arrays verschiedener Größen. Ein Ausschnitt, den Sie zur erneuten Verwendung des Arrays verwenden könnten, besteht `ReDim` das Array wieder auf `(0)` zu `ReDim` dem Array eine Variable zuzuordnen und das Array frei zu vergrößern.

In dem folgenden Snippet konstruiere ich ein Array mit den Werten 1 bis 40, leeren Sie das Array und füllen Sie das Array mit den Werten 40 bis 100 auf. Dies geschieht alles dynamisch.

```
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
```

Gezackte Arrays (Arrays von Arrays)

Gezackte Arrays NICHT multidimensionale Arrays

Arrays von Arrays (gezackte Arrays) sind nicht identisch mit mehrdimensionalen Arrays, wenn Sie visuell darüber nachdenken. Multidimensionale Arrays sehen wie Matrizen (rechteckig) mit einer definierten Anzahl von Elementen auf ihren Dimensionen (innerhalb von Arrays) aus, während ein Jagged-Array wie ein Jahr aussehen würde Kalender, wobei die inneren Arrays eine unterschiedliche Anzahl von Elementen aufweisen, z. B. Tage in verschiedenen Monaten.

Obwohl Jagged Arrays aufgrund ihrer verschachtelten Ebenen recht unübersichtlich und schwierig zu verwenden sind, bieten sie nicht viel Sicherheit. Sie sind jedoch sehr flexibel und ermöglichen es Ihnen, verschiedene Datentypen auf einfache Weise zu manipulieren und müssen nicht verwendet werden leere Elemente.

Erstellen eines gezackten Arrays

Im folgenden Beispiel werden wir ein gezacktes Array initialisieren, das zwei Arrays enthält, eines für Names und eines für Numbers, und dann auf ein Element von jedem zugreifen

```
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
```

Gepackte Arrays dynamisch erstellen und lesen

Wir können auch dynamischer sein, um die Arrays zu erstellen, stellen wir uns vor, wir haben ein Kundendatenblatt in Excel und wir möchten ein Array erstellen, um die Kundendaten auszugeben.

```
   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
```

Wir werden dynamisch ein Header-Array und ein Customers-Array erstellen, der Header enthält die Spaltentitel und das Customers-Array enthält die Informationen der einzelnen Kunden / Zeilen als Arrays.

```
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.

Das Ergebnis des obigen Snippets ist ein Jagged-Array mit zwei Arrays, eines dieser Arrays mit 4 Elementen, 2 Eindrückungsstufen und das andere selbst, ein weiteres Jagged-Array, das 5 Arrays mit jeweils 4 Elementen und 3 Eindrückungsstufen enthält. Siehe folgende Struktur:

```
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)
```

Um auf die Informationen zuzugreifen, müssen Sie die Struktur des von Ihnen erstellten Jagged-Arrays berücksichtigen. Im obigen Beispiel sehen Sie, dass das `Main Array` ein Array von `Headers` und ein Array von Arrays (`Customers`) enthält, mit unterschiedlichen Möglichkeiten Zugriff auf die Elemente.

Nun lesen wir die Informationen des `Main Array` und drucken jede Kundeninformation als `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
```

Denken Sie daran, die Struktur Ihres Jagged-Arrays zu verfolgen. In dem Beispiel oben, um auf den Namen eines Kunden zuzugreifen, `Main_Array -> Customers -> CustomerNumber -> Name` der aus drei Ebenen besteht, um `"Person4"` die Position der Kunden im Main_Array, dann die Position des Kunden vier im Arrays Customers Jagged und zuletzt die Position des benötigten Elements, in diesem Fall `Main_Array(1)(3)(0)` wobei es sich um `Main_Array(Customers)(CustomerNumber)(Name)` .

Mehrdimensionale Arrays

Mehrdimensionale Arrays

Wie der Name schon sagt, sind mehrdimensionale Arrays Arrays, die mehr als eine Dimension enthalten, normalerweise zwei oder drei, aber sie können bis zu 32 Dimensionen haben.

Ein Multi-Array arbeitet wie eine Matrix mit verschiedenen Ebenen, nehmen Sie zum Beispiel einen Vergleich zwischen einer, zwei und drei Dimensionen.

Eine Dimension ist Ihr typisches Array, es sieht aus wie eine Liste von Elementen.

```
Dim 1D(3) as Variant

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

```
(1)
(2)
```

Zwei Dimensionen würden wie ein Sudoku-Grid oder eine Excel-Tabelle aussehen. Beim Initialisieren des Arrays würden Sie festlegen, wie viele Zeilen und Spalten das Array enthalten würde.

```
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)
```

Drei Dimensionen würden wie Rubiks Würfel aussehen. Beim Initialisieren des Arrays würden Sie Zeilen und Spalten und Layer / Tiefen definieren, die das Array hätte.

```
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)
```

Weitere Dimensionen könnten als Multiplikation der 3D-Technik betrachtet werden, sodass eine 4D (1,3,3,3) zwei nebeneinander liegende 3D-Arrays sein würde.

Zwei-Dimension-Array

Erstellen

Im folgenden Beispiel wird eine Liste von Mitarbeitern zusammengestellt. Jeder Mitarbeiter verfügt über eine Reihe von Informationen (Vorname, Nachname, Adresse, E-Mail, Telefon ...). Das Beispiel wird im Wesentlichen im Array gespeichert (Mitarbeiter, Information) (0,0) ist der Vorname des ersten Mitarbeiters.

```
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
```

Größenänderung

Größenänderung oder `ReDim Preserve` Wenn `ReDim Preserve` ein Multi-Array wie die Norm für ein One-Dimension-Array `ReDim Preserve` , wird ein Fehler `ReDim Preserve` Stattdessen müssen die Informationen in ein temporäres Array mit derselben Größe wie das Original plus der Anzahl der hinzuzufügenden Zeilen / Spalten übertragen werden. Im folgenden Beispiel sehen Sie, wie Sie ein Temp-Array initialisieren, die Informationen aus dem ursprünglichen Array übernehmen, die restlichen leeren Elemente füllen und das temporäre Array durch das ursprüngliche Array ersetzen.

```
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
```

Elementwerte ändern

Zum Ändern / Ändern der Werte in einem bestimmten Element können Sie einfach die Koordinate aufrufen, die geändert werden soll, und einen neuen Wert `Employees(0, 0) = "NewValue"` : `Employees(0, 0) = "NewValue"`

Alternativ können Sie die Koordinaten verwenden, um die Werte entsprechend den erforderlichen Parametern abgleichen zu können:

```
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
```

lesen

Der Zugriff auf die Elemente im Array kann mit einer geschachtelten Schleife (durchlaufen jedes Element), mit Schleife und Koordinate (durch wiederholte Zeilen und direktem Zugriff auf Spalten) oder direkt mit beiden Koordinaten erfolgen.

```
'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)
```

***Denken** Sie **daran** , dass es immer praktisch ist, eine Array-Map beizubehalten, wenn Sie multidimensionale Arrays verwenden.*

Drei-Dimension-Array

Für das 3D-Array verwenden wir dieselbe Prämisse wie das 2D-Array. Zusätzlich werden nicht nur der Mitarbeiter und die Informationen gespeichert, sondern auch das Gebäude, in dem sie arbeiten.

Das 3D-Array enthält die Angestellten (können als Zeilen betrachtet werden), die Informationen (Spalten) und das Gebäude, die als unterschiedliche Tabellen in einem Excel-Dokument betrachtet werden können. Sie haben die gleiche Größe zwischen ihnen, aber jede Tabelle hat ein unterschiedliche Informationen in den Zellen / Elementen. Das 3D-Array enthält ***n*** Anzahl von 2D-Arrays.

Erstellen

Ein 3D-Array benötigt 3 Koordinaten, um initialisiert zu werden. `Dim 3Darray(2,5,5) As Variant` die erste Koordinate des Arrays die Anzahl der Gebäude / Pläne (verschiedene Reihen- und Spaltensätze) sein, die zweite Koordinate definiert die Zeilen und die dritte Säulen. Das oben genannte `Dim` ergibt ein 3D-Array mit 108 Elementen (`3*6*6`), das effektiv 3 verschiedene Sätze von 2D-Arrays enthält.

```
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
```

Größenänderung

Das Ändern der Größe eines 3D-Arrays ähnelt dem Ändern der Größe eines 2D-Arrays. Erstellen

Sie zunächst ein temporäres Array mit der gleichen Größe des Originals, indem Sie eins in der Koordinate des Parameters hinzufügen, um zu erhöhen. Die erste Koordinate erhöht die Anzahl der Sätze im Array, die zweite und die dritte Koordinaten erhöhen die Anzahl der Zeilen oder Spalten in jedem Satz.

Das folgende Beispiel erhöht die Anzahl der Zeilen in jedem Satz um eine und füllt die zuletzt hinzugefügten Elemente mit neuen Informationen.

```
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
```

Elementwerte ändern und lesen

Das Lesen und Ändern der Elemente auf dem 3D-Array kann auf ähnliche Weise erfolgen wie das 2D-Array. Passen Sie einfach die zusätzliche Ebene in den Schleifen und Koordinaten an.

```
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)
```

Arrays online lesen: https://riptutorial.com/de/vba/topic/3064/arrays

Kapitel 8: Arrays kopieren, zurückgeben und übergeben

Examples

Arrays kopieren

Sie können ein VBA-Array mit dem Operator = in ein Array desselben Typs kopieren. Die Arrays müssen vom gleichen Typ sein, andernfalls löst der Code einen Kompilierungsfehler "Array nicht zuordnen" aus.

```
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
```

Das Quellarray kann fest oder dynamisch sein, das Zielarray muss jedoch dynamisch sein. Wenn Sie versuchen, in ein festes Array zu kopieren, wird ein Kompilierungsfehler "Kann Array nicht zuweisen" ausgelöst. Alle bereits vorhandenen Daten im empfangenden Array gehen verloren, und ihre Grenzen und Dimensionen werden in das Quellarray geändert.

```
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)
```

Sobald die Kopie erstellt wurde, sind die beiden Arrays im Speicher getrennt, dh die beiden Variablen sind keine Verweise auf dieselben zugrunde liegenden Daten. Daher werden die an einem Array vorgenommenen Änderungen nicht im anderen angezeigt.

```
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
```

Arrays von Objekten kopieren

Bei Arrays von Objekten werden die *Referenzen* auf diese Objekte kopiert, nicht die Objekte selbst. Wenn eine Änderung an einem Objekt in einem Array vorgenommen wird, scheint es auch in dem anderen Array geändert zu werden - beide referenzieren auf dasselbe Objekt. Wenn Sie ein Element in einem Array auf ein anderes Objekt setzen, wird dieses Objekt jedoch nicht auf das andere Array gesetzt.

```
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
```

Varianten, die ein Array enthalten

Sie können ein Array auch in und aus einer Variantenvariablen kopieren. Beim Kopieren aus einer Variante muss ein Array desselben Typs wie das empfangende Array enthalten sein. Andernfalls wird ein Laufzeitfehler "Typkonflikt" ausgelöst.

```
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
```

Arrays aus Funktionen zurückgeben

Eine Funktion in einem normalen Modul (aber nicht einem Class-Modul) kann ein Array zurückgeben, indem Sie `()` hinter den Datentyp stellen.

```
Function arrayOfPiDigits() As Long()
```

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

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

    arrayOfPiDigits = outputArray
End Function
```

Das Ergebnis der Funktion kann dann in einem dynamischen Array desselben Typs oder einer Variante abgelegt werden. Auf die Elemente kann auch direkt mit einem zweiten Satz von Klammern zugegriffen werden. Die Funktion wird jedoch jedes Mal aufgerufen. Daher empfiehlt es sich, die Ergebnisse in einem neuen Array zu speichern, wenn sie mehrmals verwendet werden sollen

```
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
```

Beachten Sie, dass das, was zurückgegeben wird, tatsächlich eine Kopie des Arrays innerhalb der Funktion ist, keine Referenz. Wenn die Funktion also den Inhalt eines statischen Arrays zurückgibt, können ihre Daten nicht von der aufrufenden Prozedur geändert werden.

Ausgabe eines Arrays über ein Ausgabeargument

Es ist normalerweise eine gute Kodierungspraxis, wenn die Argumente einer Prozedur eingegeben und über den Rückgabewert ausgegeben werden. Die Einschränkungen von VBA machen es jedoch manchmal erforderlich, dass eine Prozedur Daten über ein `ByRef` Argument `ByRef` .

Ausgabe in ein festes Array

```
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
```

Ausgabe eines Arrays aus einer Class-Methode

Ein Ausgabeargument kann auch zur Ausgabe eines Arrays aus einer Methode / einem Verfahren in einem Klassenmodul verwendet werden

```
' 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
```

Arrays an Vorgänge übergeben

Arrays können an verfahren übergeben werden, indem nach dem Namen der Arrayvariablen `()` wird.

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

Arrays *müssen* als Referenz übergeben werden. Wenn kein Übergabemechanismus angegeben wird, z. B. `myFunction(arr())` , nimmt VBA standardmäßig `ByRef` an. Es ist jedoch `ByRef` , dies explizit zu machen. Beim Versuch, ein Array über einen Wert zu übergeben, z. B. führt `myFunction(ByVal arr())` zu einem Kompilierungsfehler "Array-Argument muss ByRef sein" (oder zu einem Kompilierungsfehler "Syntaxfehler", wenn die `Auto Syntax Check` Syntaxprüfung in den VBE-Optionen nicht geprüft wird) .

Übergeben als Verweis bedeutet, dass alle Änderungen am Array im aufrufenden Vorgang beibehalten werden.

```
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
```

Wenn Sie das ursprüngliche Array nicht ändern möchten, schreiben Sie die Funktion so, dass keine Elemente geändert werden.

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

Alternativ können Sie eine Arbeitskopie des Arrays erstellen und mit der Kopie arbeiten.

```
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
```

Arrays kopieren, zurückgeben und übergeben online lesen:
https://riptutorial.com/de/vba/topic/9069/arrays-kopieren--zuruckgeben-und-ubergeben

Kapitel 9: Attribute

Syntax

- Attribut VB_Name = "ClassOrModuleName"
- Attribut VB_GlobalNameSpace = False 'wird ignoriert
- Attribut VB_Creatable = False 'Ignoriert
- Attribut VB_PredeclaredId = {True | Falsch}
- Attribut VB_Exposed = {True | Falsch}
- Attribut variableName.VB_VarUserMemId = 0 'Null bedeutet, dass dies das Standardmitglied der Klasse ist.
- Attribut variableName.VB_VarDescription = "some string" 'Fügt den Text zu den Objektbrowserinformationen für diese Variable hinzu.
- Attribut procName.VB_Description = "some string" 'Fügt den Text der Object Browser-Information für die Prozedur hinzu.
- Attribut procName.VB_UserMemId = {0 | -4}
    - '0: Macht die Funktion zum Standardmitglied der Klasse.
    - '-4: Gibt an, dass die Funktion einen Enumerator zurückgibt.

Examples

VB_Name

VB_Name gibt den Klassen- oder Modulnamen an.

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

Eine neue Instanz dieser Klasse würde mit erstellt

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

VB_GlobalNameSpace

In VBA wird dieses Attribut ignoriert. Es wurde nicht von VB6 übertragen.

In VB6 wird eine globale Standardinstanz der Klasse erstellt (eine "Verknüpfung"), sodass auf Klassenmitglieder ohne Verwendung des Klassennamens zugegriffen werden kann. Beispielsweise ist `DateTime` (wie in `DateTime.Now`) tatsächlich Teil der `VBA.Conversion` Klasse.

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

VB_Createable

Dieses Attribut wird ignoriert. Es wurde nicht von VB6 übertragen.

In VB6 wurde es in Verbindung mit dem Attribut `VB_Exposed` , um die Zugänglichkeit von Klassen außerhalb des aktuellen Projekts zu steuern.

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

Dies würde zu einer `Public Class` , auf die von anderen Projekten aus zugegriffen werden kann. Diese Funktionalität ist jedoch in VBA nicht vorhanden.

VB_PredeclaredId

Erstellt eine globale Standardinstanz einer Klasse. Auf die Standardinstanz wird über den Namen der Klasse zugegriffen.

Erklärung

```
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
```

Anruf

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

In gewisser Weise simuliert dies das Verhalten statischer Klassen in anderen Sprachen. Im Gegensatz zu anderen Sprachen können Sie jedoch immer noch eine Instanz der Klasse erstellen.

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

VB_Exposed

Steuert die Instanzierungsmerkmale einer Klasse.

```
Attribute VB_Exposed = False
```

Macht die Klasse `Private` . Es kann nicht außerhalb des aktuellen Projekts zugegriffen werden.

```
Attribute VB_Exposed = True
```

Macht die Klasse `Public` außerhalb des Projekts verfügbar. Da `VB_Createable` jedoch in VBA ignoriert wird, können Instanzen der Klasse nicht direkt erstellt werden. Dies entspricht einer folgenden VB.Net-Klasse.

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

Um eine Instanz von außerhalb des Projekts zu erhalten, müssen Sie eine Fabrik zum Erstellen von Instanzen verfügbar machen. Eine Möglichkeit hierzu ist ein reguläres `Public` Modul.

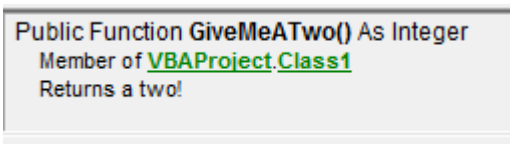
```
Public Function CreateFoo() As Foo
    CreateFoo = New Foo
End Function
```

Da öffentliche Module von anderen Projekten aus zugänglich sind, können wir neue Instanzen unserer `Public - Not Createable` Klassen `Public - Not Createable` .

VB_Beschreibung

Fügt einer Klasse oder einem Modulmitglied eine Textbeschreibung hinzu, die im Objekt-Explorer sichtbar wird. Idealerweise sollten alle öffentlichen Mitglieder einer öffentlichen Schnittstelle / API eine Beschreibung haben.

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



Hinweis: Alle Accessor-Mitglieder einer Eigenschaft (`Get` , `Let` , `Set`) verwenden dieselbe Beschreibung.

VB_ [Var] UserMemId

`VB_VarUserMemId` (für `VB_VarUserMemId`) und `VB_UserMemId` (für Prozeduren) werden in VBA hauptsächlich für zwei `VB_UserMemId` verwendet.

Angeben des Standardmitglieds einer Klasse

Eine `List` , die eine `Collection` kapseln würde, würde eine `Item` Eigenschaft haben, sodass der Clientcode Folgendes tun kann:

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

`VB_UserMemId` ein `VB_UserMemId` Attribut für die `Item` Eigenschaft auf 0 gesetzt ist, kann der Clientcode `VB_UserMemId` tun:

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

Nur ein Member kann in einer bestimmten Klasse legal `VB_UserMemId = 0` haben. Geben Sie für Eigenschaften das Attribut im `Get` Accessor an:

```
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
```

Eine Klasse mit einem `For Each` Schleifenkonstrukt iterierbar machen

Mit dem magischen Wert `-4` teilt das `VB_UserMemId` Attribut VBA mit, dass dieses Member einen Enumerator liefert - was dem Client-Code `VB_UserMemId` ermöglicht:

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

Die einfachste Methode zum Implementieren dieser Methode ist das Aufrufen des hidden-`[_NewEnum]` Eigenschafts für eine interne / gekapselte `Collection` . Der Bezeichner muss in eckige Klammern eingeschlossen werden, da der führende Unterstrich ihn zu einem unzulässigen VBA-Bezeichner macht:

```
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
```

Attribute online lesen: https://riptutorial.com/de/vba/topic/5321/attribute

Kapitel 10: Automatisierung oder Verwendung anderer Anwendungsbibliotheken

Einführung

Wenn Sie die Objekte in anderen Anwendungen als Teil Ihrer Visual Basic-Anwendung verwenden, möchten Sie möglicherweise einen Verweis auf die Objektbibliotheken dieser Anwendungen einrichten. Diese Dokumentation enthält eine Liste, Quellen und Beispiele für die Verwendung von Bibliotheken verschiedener Software, z. B. Windows Shell, Internet Explorer, XML HttpRequest und andere.

Syntax

- expression.CreateObject (ObjectName)
- Ausdruck; Erforderlich. Ein Ausdruck, der ein Anwendungsobjekt zurückgibt.
- Objektname; Erforderliche Zeichenfolge Der Klassenname des zu erstellenden Objekts. Informationen zu gültigen Klassennamen finden Sie unter Programmierbare OLE-Bezeichner.

Bemerkungen

- MSDN-Automatisierung

  Wenn eine Anwendung die Automatisierung unterstützt, kann auf die von der Anwendung bereitgestellten Objekte von Visual Basic zugegriffen werden. Verwenden Sie Visual Basic, um diese Objekte zu bearbeiten, indem Sie Methoden für das Objekt aufrufen oder die Eigenschaften des Objekts abrufen und festlegen.

- MSDN-Check oder Hinzufügen einer Objektbibliothek

  Wenn Sie die Objekte in anderen Anwendungen als Teil Ihrer Visual Basic-Anwendung verwenden, möchten Sie möglicherweise einen Verweis auf die Objektbibliotheken dieser Anwendungen einrichten. Bevor Sie dies tun können, müssen Sie zunächst sicherstellen, dass die Anwendung eine Objektbibliothek bereitstellt.

- MSDN-Verweise-Dialogfeld

  Ermöglicht das Auswählen von Objekten einer anderen Anwendung, die in Ihrem Code verfügbar sein sollen, indem Sie einen Verweis auf die Objektbibliothek dieser Anwendung festlegen.

- MSDN-CreateObject-Methode

Erstellt ein Automatisierungsobjekt der angegebenen Klasse. Wenn die Anwendung bereits läuft, erstellt CreateObject eine neue Instanz.

Examples

VBScript-reguläre Ausdrücke

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

Tools> Referenzen> Regelmäßige Microsoft VBScript-Ausdrücke #. #
Zugehörige DLL: VBScript.dll
Quelle: Internet Explorer 1.0 und 5.5

- MSDN-Microsoft verbessert VBScript mit regulären Ausdrücken
- MSDN-Syntax für reguläre Ausdrücke (Skripting)
- Expertenaustausch - Verwenden regulärer Ausdrücke in Visual Basic für Applikationen und Visual Basic 6
- Verwendung von regulären Ausdrücken (Regex) in Microsoft Excel sowohl in der Zelle als auch in Schleifen in SO
- regular-expressions.info/vbscript
- regular-expressions.info/vbscriptexample
- WIKI-regulärer Ausdruck

Code

Sie können diese Funktionen verwenden, um RegEx-Ergebnisse zu erhalten, alle Übereinstimmungen (wenn mehr als 1) in einer Zeichenfolge zu verketten und das Ergebnis in einer Excel-Zelle anzuzeigen.

```
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
```

Skript-Dateisystemobjekt

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

Extras> Referenzen> Microsoft Scripting Runtime
Zugehörige DLL: ScrRun.dll
Quelle: Windows-Betriebssystem

MSDN-Zugriff auf Dateien mit FileSystemObject

> Das Dateisystemobjektmodell (FSO) bietet ein objektbasiertes Werkzeug zum Arbeiten mit Ordnern und Dateien. Es ermöglicht Ihnen die Verwendung der bekannten object.method-Syntax mit einem umfangreichen Satz von Eigenschaften, Methoden und Ereignissen zum Verarbeiten von Ordnern und Dateien. Sie können auch die herkömmlichen Visual Basic-Anweisungen und -Befehle verwenden.
>
> Mit dem FSO-Modell können Sie mit Ihrer Anwendung Ordner erstellen, ändern, verschieben und löschen oder feststellen, ob und wo bestimmte Ordner vorhanden sind. Außerdem können Sie Informationen zu Ordnern abrufen, z. B. deren Namen und das Datum, an dem sie erstellt oder zuletzt geändert wurden.

MSDN-Filesystem Themen : *„... erklären das Konzept des Filesystem und wie es zu benutzen.“*
Exceltrick-Filesystem in VBA - Erklärte
Scripting.FileSystemObject

Scripting Dictionary-Objekt

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

Extras> Referenzen> Microsoft Scripting Runtime
Zugehörige DLL: ScrRun.dll
Quelle: Windows-Betriebssystem

Scripting.Dictionary-Objekt
MSDN-Dictionary-Objekt

Internet Explorer-Objekt

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

Extras> Referenzen> Microsoft Internet Controls
Zugehörige DLL: ieframe.dll
Quelle: Internet Explorer-Browser

MSDN-InternetExplorer-Objekt

> Steuert eine Instanz von Windows Internet Explorer durch Automatisierung.

Grundlegende Mitglieder des Internet Explorer-Objekts

Der folgende Code sollte einführen, wie das IE-Objekt funktioniert und wie es über VBA bearbeitet wird. Ich empfehle einen Schritt durch, andernfalls kann es bei mehreren Navigationen zu Fehlern kommen.

```
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
```

Web Scraping

Die häufigste Sache, die mit IE zu tun ist, ist, einige Informationen einer Website zu kratzen oder ein Website-Formular auszufüllen und Informationen zu übermitteln. Wir werden sehen, wie es geht.

Betrachten wir den Quellcode von *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>
```

Wir können Code wie unten verwenden, um Informationen zu erhalten und einzustellen:

```
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
```

Was ist los? Der Schlüsselspieler ist hier das **.Document** , also der HTML-Quellcode. Wir können einige Abfragen durchführen, um die gewünschten Sammlungen oder Objekte zu erhalten. Zum Beispiel `IE.Document.GetElementsByTagName("title")(0).innerHtml` . `GetElementsByTagName` gibt eine **Sammlung** von HTML - Elemente, die den *„Titel“* Tag haben. Es gibt nur einen solchen Tag im Quellcode. Die **Collection** basiert auf 0. Um das erste Element zu erhalten, fügen wir `(0)` . In unserem Fall wollen wir nur die `innerHtml` (ein String), nicht das Element-Objekt selbst. Wir geben also die gewünschte Eigenschaft an.

Klicken

Um einem Link auf einer Site zu folgen, können wir mehrere Methoden verwenden:

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

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

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

        'Return Back
        .GoBack
        Stop 'VBE Stop.

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

```
        Stop 'VBE Stop.

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

Microsoft HTML Object Library oder IE Bester Freund

Um den in den IE geladenen HTML-Code optimal zu nutzen, können (oder sollten) Sie eine andere Bibliothek verwenden, z. B. die *Microsoft HTML Object Library* . Mehr dazu in einem anderen Beispiel.

IE Hauptprobleme

Das Hauptproblem bei IE besteht darin, zu überprüfen, ob die Seite vollständig geladen wurde und bereit ist, mit ihr zu interagieren. Die `Do While... Loop` hilft, ist aber nicht zuverlässig.

Die Verwendung von IE nur zum Abkratzen von HTML-Inhalten ist OVERKILL. Warum? Da der Browser zum Browsen gedacht ist, dh die Webseite mit allen CSS, JavaScripts, Bildern, Popups usw. anzeigen soll. Wenn Sie nur die Rohdaten benötigen, sollten Sie einen anderen Ansatz verwenden. ZB mit XML HTTPRequest . Mehr dazu in einem anderen Beispiel.

Automatisierung oder Verwendung anderer Anwendungsbibliotheken online lesen: https://riptutorial.com/de/vba/topic/8916/automatisierung-oder-verwendung-anderer-anwendungsbibliotheken

Kapitel 11: Bedingte Kompilierung

Examples

Ändern des Codeverhaltens zur Kompilierzeit

Die `#Const` Direktive wird verwendet, um eine benutzerdefinierte Präprozessorkonstante zu definieren. Diese können später von `#If` verwendet werden, `#If` zu steuern, welche Codeblöcke kompiliert und ausgeführt werden.

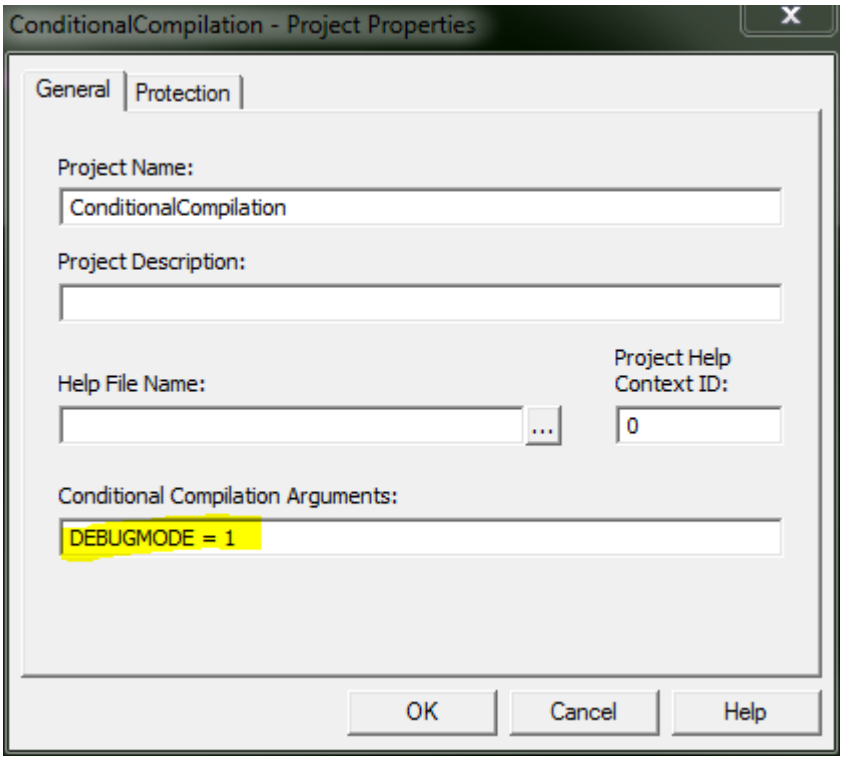
```
#Const DEBUGMODE = 1

#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
```

Dies führt dazu, dass der `filepath` auf `"C:\Users\UserName\Path\To\File.txt"` . `#Const` Zeile `#Const DEBUGMODE = 0` oder in `#Const DEBUGMODE = 0 filepath` wird der `filepath` auf `"\\server\share\path\to\file.txt"` .

#Const Scope

Die Direktive `#Const` ist nur für eine einzelne `#Const` (Modul oder Klasse) wirksam. Sie muss für jede einzelne Datei deklariert werden, in der Sie Ihre benutzerdefinierte Konstante verwenden möchten. Alternativ können Sie eine `#Const` für Ihr Projekt deklarieren, indem Sie auf Tools >> [Ihr Projektname] Projekteigenschaften klicken. Daraufhin wird das Dialogfeld Projekteigenschaften geöffnet, in das die Konstantendeklaration eingegeben wird. `[constName] = [value]` im Feld "Bedingte Kompilierungsargumente" `[constName] = [value]` . Sie können mehr als eine Konstante eingeben, indem Sie sie mit einem Doppelpunkt `[constName1] = [value1] : [constName2] = [value2]` , wie `[constName1] = [value1] : [constName2] = [value2]` .



Vordefinierte Konstanten

Einige Kompilierungskonstanten sind bereits vordefiniert. Welche davon existieren, hängt von der Bitness der Office-Version ab, in der Sie VBA ausführen. Beachten Sie, dass Vba7 neben Office 2010 zur Unterstützung von 64-Bit-Versionen von Office eingeführt wurde.

Konstante	16 bit	32 bit	64 bit
Vba6	Falsch	Wenn Vba6	Falsch
Vba7	Falsch	Wenn Vba7	Wahr
Win16	Wahr	Falsch	Falsch
Win32	Falsch	Wahr	Wahr
Win64	Falsch	Falsch	Wahr
Mac	Falsch	Wenn Mac	Wenn Mac

Beachten Sie, dass sich Win64 / Win32 auf die Office-Version und nicht auf die Windows-Version beziehen. Beispiel: Win32 = TRUE in 32-Bit-Office, auch wenn das Betriebssystem eine 64-Bit-Version von Windows ist.

Verwenden von Declare Importiert alle Office-Versionen

```
#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
```

Dies kann etwas vereinfacht werden, je nachdem, welche Office-Versionen Sie unterstützen müssen. Zum Beispiel unterstützen nicht viele Leute immer noch 16-Bit-Versionen von Office. Die letzte Version von 16 Bit Office war Version 4.3, die 1994 veröffentlicht wurde. Daher ist die folgende Erklärung für fast alle modernen Fälle (einschließlich Office 2007) ausreichend.

```
#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
```

Wenn Sie nichts älteres als Office 2010 unterstützen müssen, funktioniert diese Deklaration einwandfrei.

```
' 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
```

Bedingte Kompilierung online lesen: https://riptutorial.com/de/vba/topic/3364/bedingte-kompilierung

Kapitel 12: Bemerkungen

Bemerkungen

Kommentarblöcke

Wenn Sie mehrere Zeilen auf einmal kommentieren oder auskommentieren müssen, können Sie die Schaltflächen der Symbolleiste " **Bearbeiten"** der IDE verwenden:

> **Kommentarblock** - Fügt dem Anfang aller ausgewählten Zeilen einen einzelnen Apostroph hinzu

> **Blockieren ohne Kommentar** - Entfernt den ersten Apostroph vom Anfang aller ausgewählten Zeilen

Mehrzeilige Kommentare Viele andere Sprachen unterstützen mehrzeilige Blockkommentare, VBA erlaubt jedoch nur einzeilige Kommentare.

Examples

Apostrophe Kommentare

Ein Kommentar wird durch einen Apostroph (') markiert und bei der Ausführung des Codes ignoriert. Kommentare helfen zukünftigen Lesern, sich selbst, Ihren Code zu erklären.

Da alle Zeilen, die mit einem Kommentar beginnen, ignoriert werden, können sie auch verwendet werden, um die Ausführung von Code zu verhindern (während Sie debuggen oder umgestalten). Platzieren Sie einen Apostroph ' , bevor Sie Ihren Code verwandelt es in einen Kommentar. (Dies wird als *Kommentieren* der Zeile bezeichnet.)

```
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-Kommentare

```
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
```

Bemerkungen online lesen: https://riptutorial.com/de/vba/topic/2059/bemerkungen

Kapitel 13: Benutzerformulare

Examples

Best Practices

Eine `UserForm` ist ein Klassenmodul mit einem Designer und einer **Standardinstanz** . Auf den *Designer* kann durch Drücken von `UMSCHALT` + `F7` zugegriffen werden, während der *hintere Code* angezeigt wird. Auf den `nachfolgenden` *Code* kann durch Drücken von `F7` zugegriffen werden, während der *Designer angezeigt wird* .

Jedes Mal mit einer neuen Instanz arbeiten.

Ein Formular ist daher als *Klassenmodul* ein *Entwurf* für ein *Objekt* . Da ein Formular Status und Daten enthalten kann, empfiehlt es sich, mit einer neuen *Instanz* der Klasse zu arbeiten, anstatt mit der standardmäßigen / globalen *Instanz* :

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

Anstatt:

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

Das Arbeiten mit der Standardinstanz kann zu subtilen Fehlern führen, wenn das Formular mit der roten "X" -Taste geschlossen wird und / oder wenn `Unload Me` im Code- `Unload Me` verwendet wird.

Implementieren Sie die Logik an anderer Stelle.

Ein Formular sollte sich nur mit der *Darstellung* befassen: Ein `Click` Handler, der eine Verbindung zu einer Datenbank herstellt und eine parametrisierte Abfrage basierend auf Benutzereingaben ausführt, führt **zu viele Aufgaben aus** .

Implementieren Sie stattdessen die *anwendbare Logik* in dem Code, der für die Anzeige des Formulars oder besser in dedizierten Modulen und Prozeduren verantwortlich ist.

Schreiben Sie den Code so, dass die UserForm nur dafür verantwortlich ist, zu wissen, wie Daten angezeigt und erfasst werden sollen: woher die Daten kommen oder was mit den Daten danach

passiert, ist nicht von Belang.

Der Anrufer sollte sich nicht mit den Bedienelementen beschäftigen.

Erstellen Sie ein gut definiertes *Modell,* mit dem das Formular arbeiten kann, entweder in einem eigenen dedizierten Klassenmodul oder gekapselt im Code-Behind des Formulars. Setzen Sie das *Modell* mit den `Property Get` Prozeduren zur Verfügung, und lassen Sie den Client-Code damit arbeiten Sie bilden eine *Abstraktion* über die Steuerelemente und ihre winzigen Details, wobei nur die relevanten Daten dem Clientcode angezeigt werden.

Dies bedeutet Code, der so aussieht:

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

An Stelle von:

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

Behandeln Sie das QueryClose-Ereignis.

Formulare haben in der Regel die Schaltfläche `Schließen` und Eingabeaufforderungen / Dialogfelder die Schaltflächen `OK` und `Abbrechen` . Der Benutzer kann das Formular mithilfe des *Kontrollkästchens* des Formulars schließen (die rote Schaltfläche "X"), wodurch standardmäßig die Formularinstanz zerstört wird (ein weiterer guter Grund, *jedes Mal mit einer neuen Instanz zu arbeiten*).

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

Die einfachste Möglichkeit, das `QueryClose` Ereignis zu behandeln, besteht darin, den Parameter `Cancel` auf `True` und dann das Formular *auszublenden* , anstatt es zu *schließen* :

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

```
End Sub
```

Auf diese Weise wird die Instanz niemals durch die Schaltfläche "X" zerstört, und der Anrufer kann sicher auf alle öffentlichen Mitglieder zugreifen.

Verstecken, nicht schließen.

Der Code, der ein Objekt erstellt, sollte für die Zerstörung verantwortlich sein. Es ist nicht die Aufgabe des Formulars, sich selbst zu entladen und zu beenden.

Vermeiden Sie die Verwendung von `Unload Me` im Code- `Unload Me` eines Formulars. Rufen `Me.Hide` stattdessen `Me.Hide` , damit der aufrufende Code das Objekt, das er erstellt hat, beim Schließen des Formulars weiterhin verwenden kann.

Nennen Sie die Dinge.

Verwenden Sie das *Eigenschaftenwerkzeugfenster* (`F4`), um jedes Steuerelement in einem Formular sorgfältig zu benennen. Der Name eines Steuerelements wird im Code-Behind verwendet. Wenn Sie also kein Refactoring-Tool verwenden, das damit umgehen kann, wird das **Umbenennen eines Steuerelements den Code** beschädigen. Daher ist es viel einfacher, Dinge richtig zu machen, als zu versuchen um herauszufinden, für welches der 20 `TextBox12` steht.

Normalerweise werden UserForm-Steuerelemente mit ungarischen Präfixen benannt:

- `lblUserName` für ein `Label` Steuerelement, das einen Benutzernamen angibt.
- `txtUserName` für ein `TextBox` Steuerelement, in das der Benutzer einen Benutzernamen eingeben kann.
- `cboUserName` für ein `ComboBox` Steuerelement, in dem der Benutzer einen Benutzernamen eingeben oder auswählen kann.
- `lstUserName` für ein `ListBox` Steuerelement, bei dem der Benutzer einen Benutzernamen auswählen kann.
- `btnOk` oder `cmdOk` für ein `Button` Steuerelement mit der Bezeichnung "Ok".

Das Problem ist, dass, wenn beispielsweise die Benutzeroberfläche umgestaltet wird und eine `ComboBox` in eine `ListBox` , der Name geändert werden muss, um den neuen Steuerelementtyp widerzuspiegeln: Es ist besser, Steuerelemente für das, was sie repräsentieren, zu benennen, als nach ihrem Steuerelementtyp - um sie zu *entkoppeln* Code von der Benutzeroberfläche so viel wie möglich.

- `UserNameLabel` für ein schreibgeschütztes Label, das einen Benutzernamen angibt.
- `UserNameInput` für ein Steuerelement, bei dem der Benutzer einen Benutzernamen eingeben oder auswählen kann.
- `OkButton` für eine Befehlsschaltfläche mit der Bezeichnung "Ok".

Unabhängig davon, welcher Stil gewählt wird, ist alles besser, als alle Steuerelemente ihren Standardnamen zu belassen. Ideal ist auch die Konsistenz im Benennungsstil.

Umgang mit QueryClose

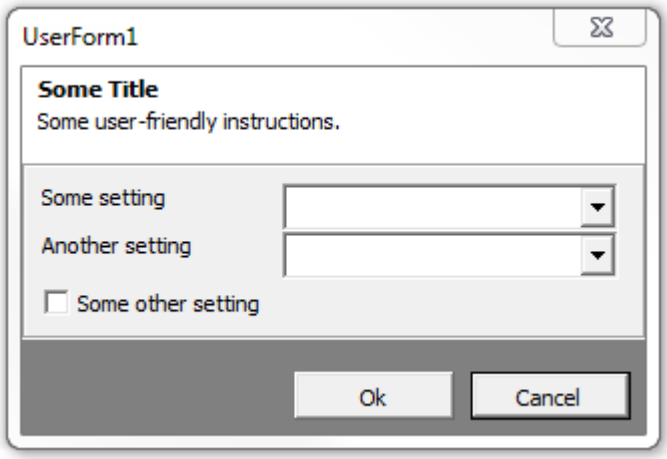
Das `QueryClose` Ereignis wird immer dann `QueryClose` , wenn ein Formular geschlossen wird, entweder über Benutzeraktionen oder programmgesteuert. Der Parameter `CloseMode` enthält einen `VbQueryClose` , der angibt, wie das Formular geschlossen wurde:

Konstante	Beschreibung	Wert
`vbFormControlMenu`	Das Formular wird als Reaktion auf eine Benutzeraktion geschlossen	0
`vbFormCode`	Das Formular wird als Antwort auf eine `Unload` Anweisung geschlossen	1
`vbAppWindows`	Windows-Sitzung wird beendet	2
`vbAppTaskManager`	Windows Task Manager schließt die Hostanwendung	3
`vbFormMDIForm`	Wird in VBA nicht unterstützt	4

Für eine bessere Lesbarkeit empfiehlt es sich, diese Konstanten zu verwenden, anstatt deren Werte direkt zu verwenden.

Eine stornierbare UserForm

Ein Formular mit einer `Abbrechen-` Schaltfläche erhalten



Der Code-Behind des Formulars könnte folgendermaßen aussehen:

```
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
```

Der aufrufende Code könnte dann das Formular anzeigen und wissen, ob es storniert wurde:

```
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
```

Die `IsCancelled` Eigenschaft gibt `True` wenn auf die Schaltfläche `Cancel` geklickt wird oder wenn der Benutzer das Formular mithilfe des *Steuerelements* schließt.

Benutzerformulare online lesen: https://riptutorial.com/de/vba/topic/5351/benutzerformulare

Kapitel 14: CreateObject vs. GetObject

Bemerkungen

Im einfachsten `CreateObject` erstellt `CreateObject` eine Instanz eines Objekts, während `GetObject` eine vorhandene Instanz eines Objekts `GetObject` . Die Bestimmung, ob ein Objekt erstellt oder abgerufen werden kann, hängt von seiner Instanzeigenschaft ab . Einige Objekte sind SingleUse (z. B. WMI) und können nicht erstellt werden, wenn sie bereits vorhanden sind. Andere Objekte (z. B. Excel) sind MultiUse und ermöglichen die gleichzeitige Ausführung mehrerer Instanzen. Wenn eine Instanz eines Objekts noch nicht vorhanden ist und Sie `GetObject` versuchen, erhalten Sie die folgende abfangbare Meldung: `Run-time error '429': ActiveX component can't create object` .

Für GetObject muss mindestens einer dieser beiden optionalen Parameter vorhanden sein:

1. *Pfadname* - Variante (String): Der vollständige Pfad (einschließlich Dateiname) der Datei, die das Objekt enthält. Dieser Parameter ist optional, aber *Class* ist erforderlich, wenn *Pfadname* weggelassen wird.
2. *Class* - Variant (String): Ein String, der die formale Definition (Application und ObjectType) des Objekts darstellt. *Klasse* ist erforderlich, wenn *Pfadname* weggelassen wird.

CreateObject hat einen erforderlichen Parameter und einen optionalen Parameter:

1. *Class* - Variant (String): Ein String, der die formale Definition (Application und ObjectType) des Objekts darstellt. *Klasse* ist ein erforderlicher Parameter.
2. *Servername* - Variant (String): Der Name des Remote-Computers, auf dem das Objekt erstellt wird. Wenn nicht angegeben, wird das Objekt auf dem lokalen Computer erstellt.

Class besteht immer aus zwei Teilen in Form von `Application.ObjectType` :

1. *Anwendung* - Der Name der Anwendung, zu der das Objekt gehört. |
2. *Objekttyp* - Der Typ des Objekts erstellt wird. |

Einige Beispielklassen sind:

1. Word.Anwendung
2. Excel-Tabelle
3. Scripting.FileSystemObject

Examples

Demonstration von GetObject und CreateObject

MSDN-GetObject-Funktion

> Gibt eine Referenz auf ein Objekt zurück, das von einer ActiveX-Komponente

bereitgestellt wird.

Verwenden Sie die GetObject-Funktion, wenn eine aktuelle Instanz des Objekts vorhanden ist oder wenn Sie das Objekt mit einer bereits geladenen Datei erstellen möchten. Wenn keine aktuelle Instanz vorhanden ist und das Objekt nicht mit einer geladenen Datei gestartet werden soll, verwenden Sie die Funktion 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
```

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

Kapitel 15: Datenstrukturen

Einführung

[TODO: Dieses Thema sollte ein Beispiel für alle grundlegenden Datenstrukturen des CS 101 sein, zusammen mit einigen Erläuterungen als Überblick, wie Datenstrukturen in VBA implementiert werden können. Dies wäre eine gute Gelegenheit, Konzepte, die in klassenbezogenen Themen in der VBA-Dokumentation eingeführt wurden, zu knüpfen und zu verstärken.]

Examples

Verknüpfte Liste

Dieses Beispiel für eine verknüpfte Liste implementiert abstrakte Datentypvorgänge .

SinglyLinkedNode- Klasse

```
Option Explicit

Private Value As Variant
Private NextNode As SinglyLinkedNode '"Next" is a keyword in VBA and therefore is not a valid
variable name
```

LinkedList- Klasse

```
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
```

Binärer Baum

Dies ist ein Beispiel für einen unsymmetrischen binären Suchbaum . Ein binärer Baum ist konzeptionell als Hierarchie von Knoten strukturiert, die von einer gemeinsamen Wurzel abwärts absteigen, wobei jeder Knoten zwei untergeordnete Elemente hat: links und rechts. Angenommen, die Zahlen 7, 5, 9, 3, 11, 6, 12, 14 und 15 wurden in einen BinaryTree eingefügt. Die Struktur wäre wie folgt. Beachten Sie, dass dieser binäre Baum nicht abgeglichen ist. Dies kann eine wünschenswerte Eigenschaft sein, um die Leistung von Suchvorgängen zu gewährleisten. Ein Beispiel für einen selbstausgleichenden binären Suchbaum finden Sie in AVL-Bäume .

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

```
                        \
                         15
```

BinaryTreeNode- Klasse

```
Option Explicit

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

BinaryTree- Klasse

[MACHEN]

Datenstrukturen online lesen: https://riptutorial.com/de/vba/topic/8628/datenstrukturen

Kapitel 16: Datentypen und Grenzwerte

Examples

Byte

```
Dim Value As Byte
```

Ein Byte ist ein vorzeichenloser 8-Bit-Datentyp. Es kann ganze Zahlen zwischen 0 und 255 darstellen. Wenn Sie versuchen, einen Wert außerhalb dieses Bereichs zu speichern, führt dies zu Laufzeitfehler 6: `Overflow` . Byte ist der einzige intrinsische vorzeichenlose Typ, der in VBA verfügbar ist.

Die Casting-Funktion, die in ein Byte konvertiert werden soll, ist `CByte()` . Bei Umsetzungen von Fließkommatypen wird das Ergebnis auf den nächsten ganzzahligen Wert mit einer Rundung von 0,5 gerundet.

Byte-Arrays und Strings

Strings und Byte-Arrays können durch einfache Zuweisung gegeneinander ausgetauscht werden (keine Konvertierungsfunktionen erforderlich).

Zum Beispiel:

```
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
```

Um Unicode- Zeichen kodieren zu können, belegt jedes Zeichen in der Zeichenfolge zwei Bytes im Array, wobei das niedrigstwertige Byte zuerst steht. Zum Beispiel:

```
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
```

Ganze Zahl

```
Dim Value As Integer
```

Eine Ganzzahl ist ein vorzeichenbehafteter 16-Bit-Datentyp. Es kann ganzzahlige Zahlen im Bereich von -32.768 bis 32.767 speichern. Wenn Sie versuchen, einen Wert außerhalb dieses Bereichs zu speichern, wird dies zu Laufzeitfehler 6: Überlauf.

Integer - Werte werden als Little-Endian- Werte gespeichert, wobei Negative als Zweierkomplement dargestellt werden .

Beachten Sie, dass es im Allgemeinen besser ist, Long anstelle von Integer zu verwenden, es sei denn, der kleinere Typ ist ein Member eines Typs oder muss (entweder aufgrund einer API-Aufrufkonvention oder aus einem anderen Grund) 2 Byte groß sein. In den meisten Fällen behandelt VBA Integer-Werte intern als 32-Bit. Daher hat die Verwendung des kleineren Typs normalerweise keinen Vorteil. Darüber hinaus tritt bei jeder Verwendung eines Integer-Typs ein Performance-Nachteil auf, da dieser lautlos als Long geworfen wird.

Die Casting-Funktion, die in eine Ganzzahl konvertiert werden soll, ist `CInt()` . Bei Umsetzungen von Fließkommatypen wird das Ergebnis auf den nächsten ganzzahligen Wert mit einer Rundung von 0,5 gerundet.

Boolean

```
Dim Value As Boolean
```

Ein Boolean-Wert wird zum Speichern von Werten verwendet, die als "True" oder "False" dargestellt werden können. Intern wird der Datentyp als 16-Bit-Wert gespeichert, wobei 0 False und jeder andere Wert True darstellt.

Wenn ein boolescher Wert in einen numerischen Typ umgewandelt wird, werden alle Bits auf 1 gesetzt. Dies führt zu einer internen Darstellung von -1 für vorzeichenbehaftete Typen und dem Maximalwert für einen vorzeichenlosen Typ (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
```

Die Casting-Funktion, die in ein Boolean konvertiert werden soll, ist `CBool()` . Obwohl es intern als 16-Bit-Zahl dargestellt wird, ist das Umsetzen von Werten außerhalb dieses Bereichs auf einen Booleschen Wert vor einem Überlauf sicher, obwohl alle 16 Bits auf 1 gesetzt werden:

```
Dim Example As Boolean
Example = CBool(2 ^ 17)
Debug.Print CInt(Example)   'Prints -1
Debug.Print CByte(Example)  'Prints 255
```

Lange

```
Dim Value As Long
```

Ein Long ist ein vorzeichenbehafteter 32-Bit-Datentyp. Es kann ganzzahlige Zahlen im Bereich von -2.147.483.648 bis 2.147.483.647 speichern. Wenn Sie versuchen, einen Wert außerhalb dieses Bereichs zu speichern, führt dies zu Laufzeitfehler 6: Überlauf.

Longs werden als Little-Endian- Werte gespeichert, wobei Negative als Zweierkomplement dargestellt werden .

Beachten Sie, dass Longs allgemein zum Speichern und Weiterleiten von Zeigern zu und von API-Funktionen verwendet werden, da Long mit der Breite eines Zeigers in einem 32-Bit-Betriebssystem übereinstimmt.

Die Casting-Funktion, die in ein Long konvertiert werden soll, ist `CLng()` . Bei Umsetzungen von Fließkommatypen wird das Ergebnis auf den nächsten ganzzahligen Wert mit einer Rundung von 0,5 gerundet.

Single

```
Dim Value As Single
```

Ein Single ist ein vorzeichenbehafteter 32-Bit-Gleitkomma-Datentyp. Es wird intern mit einem Little-Endian- Speicherlayout nach IEEE 754 gespeichert. Daher gibt es keinen festen Wertebereich, der durch den Datentyp dargestellt werden kann - begrenzt ist die Genauigkeit der gespeicherten Werte. Ein Single kann ***ganzzahlige*** Werte im Bereich von -16.777.216 bis 16.777.216 ohne Genauigkeitsverlust speichern. Die Genauigkeit der Gleitkommazahlen hängt vom Exponenten ab.

Ein Single wird überlaufen, wenn ein Wert größer als ungefähr 2^{128} zugewiesen wird. Bei negativen Exponenten wird es nicht überlaufen, obwohl die nutzbare Genauigkeit fraglich ist, bevor die Obergrenze erreicht wird.

Wie bei allen Fließkommazahlen sollte beim Vergleich der Gleichheit sorgfältig vorgegangen werden. Am besten verwenden Sie einen Deltawert, der der erforderlichen Genauigkeit entspricht.

Die Casting-Funktion, die in ein Single konvertiert werden soll, ist `CSng()` .

Doppelt

```
Dim Value As Double
```

Ein Double ist ein 64-Bit-Gleitkommadatentyp mit Vorzeichen. Wie das Single wird es intern unter Verwendung eines Little-Endian- IEEE 754- Speicherlayouts gespeichert, und es sollten die gleichen Vorsichtsmaßnahmen hinsichtlich der Genauigkeit getroffen werden. Ein Double kann ***ganzzahlige*** Werte im Bereich von -9.007.199.254.740.992 bis 9.007.199.254.740.992 ohne Genauigkeitsverlust speichern. Die Genauigkeit der Gleitkommazahlen hängt vom Exponenten ab.

Ein Double wird überlaufen, wenn ein Wert größer als 2^{1024} zugewiesen wird. Bei negativen Exponenten wird es nicht überlaufen, obwohl die nutzbare Genauigkeit fraglich ist, bevor die Obergrenze erreicht wird.

Die Casting-Funktion zum Konvertieren in ein Double ist `CDbl()` .

Währung

```
Dim Value As Currency
```

Eine Währung ist ein 64-Bit-Gleitkomma-Datentyp mit Vorzeichen, der einem Double ähnelt, jedoch um 10.000 skaliert ist, um den vier Stellen rechts vom Dezimalpunkt eine höhere Genauigkeit zu geben. Eine Currency-Variable kann Werte von -922.337.203.685.477.5808 bis 922.337.203.685.477.5807 speichern, wodurch sie die größte Kapazität eines beliebigen intrinsischen Typs in einer 32-Bit-Anwendung darstellt. Wie der Name des Datentyps impliziert, wird empfohlen, diesen Datentyp bei der Darstellung von monetären Berechnungen zu verwenden, da durch die Skalierung Rundungsfehler vermieden werden.

Die Casting-Funktion, die in eine Währung konvertiert werden soll, ist `CCur()` .

Datum

```
Dim Value As Date
```

Ein Date-Typ wird intern als vorzeichenbehafteter 64-Bit-Gleitkomma-Datentyp dargestellt, wobei der Wert links von der Dezimalzahl die Anzahl der Tage ab dem Epochendatum des 30. Dezember 1899 darstellt (siehe jedoch den Hinweis unten). Der Wert rechts von der Dezimalstelle steht für die Uhrzeit als gebrochenen Tag. Ein ganzzahliges Datum hätte also eine Zeitkomponente von 12:00:00 Uhr und x.5 eine Zeitkomponente von 12:00:00 Uhr.

Gültige Werte für Termine sind zwischen dem 1. Januar und dem 31. Dezember 100 st 9999. Da ein Doppel eine größere Reichweite hat, ist es möglich , ein Datum zu überfluten von Werten außerhalb dieses Bereichs zuweisen.

Als solches kann es austauschbar mit einer Double for Date-Berechnung verwendet werden:

```
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.
```

Die Casting-Funktion, die in ein Datum konvertiert werden soll, ist `CDate()` , die eine beliebige Datums- / `CDate()` numerischen Typ akzeptiert. Es ist wichtig zu beachten, dass Zeichenfolgendarstellungen von Datumsangaben basierend auf der aktuell verwendeten Gebietsschemaeinstellung konvertiert werden. Daher sollten direkte Casts vermieden werden, wenn der Code portierbar sein soll.

String

Ein String steht für eine Folge von Zeichen und ist in zwei Ausführungen erhältlich:

Variable Länge

```
Dim Value As String
```

Ein String mit variabler Länge ermöglicht das Anhängen und Abschneiden und wird als COM-BSTR im Speicher gespeichert . Diese besteht aus einer vorzeichenlosen 4-Byte-Ganzzahl, die die Länge des Strings in Bytes speichert, gefolgt von den String-Daten selbst als Breitzeichen (2 Bytes pro Zeichen) und mit 2 Null-Bytes abgeschlossen. Daher beträgt die maximale Zeichenfolgenlänge, die von VBA verarbeitet werden kann, 2.147.483.647 Zeichen.

Der interne Zeiger auf die Struktur (abrufbar mit der Funktion `StrPtr()`) zeigt auf den Speicherplatz der *Daten* , nicht auf das Längenpräfix. Dies bedeutet, dass ein VBA-String direkt an API-Funktionen übergeben werden kann, für die ein Zeiger auf ein Zeichenarray erforderlich ist.

Da sich die Länge ändern kann, *ordnet* VBA bei *jeder Variablenzuweisung den* Speicher für einen String neu zu. *Dies* kann zu Performance-Strafen für Prozeduren führen, die sie wiederholt ändern.

Feste Länge

```
Dim Value As String * 1024    'Declares a fixed length string of 1024 characters.
```

Zeichenfolgen fester Länge werden 2 Byte für jedes Zeichen zugewiesen und als einfaches Byte-Array im Speicher abgelegt. Nach der Zuweisung ist die Länge der Zeichenfolge unveränderlich. Sie werden im Arbeitsspeicher ***nicht mit*** Null abgeschlossen, daher ist eine Zeichenfolge, die den mit Nicht-Null-Zeichen belegten Speicher füllt, nicht für die Weitergabe an API-Funktionen geeignet, die eine mit Null endende Zeichenfolge erwarten.

Zeichenfolgen mit fester Länge haben eine ältere 16-Bit-Indexbegrenzung und können daher nur bis zu 65.535 Zeichen lang sein. Wenn Sie versuchen, einen Wert länger als den verfügbaren Speicherplatz zuzuweisen, führt dies nicht zu einem Laufzeitfehler. Stattdessen wird der resultierende Wert einfach abgeschnitten:

```
Dim Foobar As String * 5
Foobar = "Foo" & "bar"
```

```
Debug.Print Foobar          'Prints "Fooba"
```

Die Casting-Funktion, die in einen String eines der beiden Typen konvertiert werden soll, ist `CStr()` .

Lang Lang

```
Dim Value As LongLong
```

Ein LongLong ist ein signierter 64-Bit-Datentyp und ist nur in 64-Bit-Anwendungen verfügbar. Es ist **nicht** in 32-Bit-Anwendungen verfügbar, die auf 64-Bit-Betriebssystemen ausgeführt werden. Es kann ganzzahlige Werte im Bereich von -9,223,372,036,854,775,808 bis 9,223,372,036,854,775,807 speichern, und der Versuch, einen Wert außerhalb dieses Bereichs zu speichern, führt zu Laufzeitfehler 6: Überlauf.

LongLongs werden als Little-Endian- Werte gespeichert, wobei Negative als Zweierkomplement dargestellt werden .

Der LongLong-Datentyp wurde als Teil der 64-Bit-Betriebssystemunterstützung von VBA eingeführt. In 64-Bit-Anwendungen kann dieser Wert verwendet werden, um Zeiger zu speichern und an 64-Bit-APIs zu übergeben.

Die Casting-Funktion zum Konvertieren in LongLong ist `CLngLng()` . Bei Umsetzungen von Fließkommatypen wird das Ergebnis auf den nächsten ganzzahligen Wert mit einer Rundung von 0,5 gerundet.

Variante

```
Dim Value As Variant    'Explicit
Dim Value               'Implicit
```

Eine Variante ist ein COM-Datentyp, der zum Speichern und Austauschen von Werten beliebiger Typen verwendet wird. Jeder andere Typ in VBA kann einer Variante zugewiesen werden. Variablen, die ohne expliziten Typ deklariert wurden, der mit `As [Type]` Standard angegeben ist, sind Variant.

Varianten werden als VARIANT-Struktur im Speicher abgelegt, die aus einem Byte-Deskriptor (VARTYPE) gefolgt von 6 reservierten Bytes und einem 8-Byte-Datenbereich besteht. Bei numerischen Typen (einschließlich Date und Boolean) wird der zugrunde liegende Wert in der Variante selbst gespeichert. Bei allen anderen Typen enthält der Datenbereich einen Zeiger auf den zugrunde liegenden Wert.

VARTYPE		Reserved						Data area			
0	1	2	3	4	5	6	7	8	9	10	11

Der zugrunde liegende Typ einer Variante kann entweder mit der `VarType()` Funktion, die den im `VarType()` gespeicherten numerischen Wert zurückgibt, oder mit der `TypeName()` Funktion, die die

String-Darstellung zurückgibt, bestimmt werden:

```
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"
```

Da Varianten Werte eines beliebigen Typs speichern können, werden Zuweisungen aus Literalen ohne Typhinweise implizit in eine Variante des entsprechenden Typs gemäß der nachstehenden Tabelle umgewandelt. Literale mit Typhinweisen werden in eine Variante des angedeuteten Typs umgewandelt.

Wert	Resultierender Typ
String-Werte	String
Nicht-Fließkommazahlen im Integer-Bereich	Ganze Zahl
Nicht-Fließkommazahlen in großer Reichweite	Lange
Nicht-Fließkommazahlen außerhalb des Long-Bereichs	Doppelt
Alle Gleitkommazahlen	Doppelt

Anmerkung: Wenn es keinen bestimmten Grund gibt, eine Variante zu verwenden (dh einen Iterator in einer For Each-Schleife oder eine API-Anforderung), sollte der Typ aus folgenden Gründen für Routineaufgaben generell vermieden werden:

- Sie sind nicht typsicher, was die Möglichkeit von Laufzeitfehlern erhöht. Eine Variante, die einen Integer-Wert enthält, ändert sich beispielsweise automatisch in einen Long-Wert, anstatt überzulaufen.
- Sie führen zu einem Verarbeitungsaufwand, indem sie mindestens eine zusätzliche Zeigerdereferenzierung erfordern.
- Der Speicherbedarf für eine Variante ist immer ***mindestens*** 8 Byte höher als zum Speichern des zugrunde liegenden Typs erforderlich.

Die Casting-Funktion, die in eine Variante konvertiert werden soll, ist `CVar()` .

LongPtr

```
Dim Value As LongPtr
```

Der LongPtr wurde in VBA eingeführt, um 64-Bit-Plattformen zu unterstützen. Bei einem 32-Bit-System wird es als Long und bei 64-Bit-Systemen als LongLong behandelt.

Sie wird hauptsächlich zum Bereitstellen einer tragbaren Methode zum Speichern und Übergeben

von Zeigern auf beiden Architekturen verwendet (siehe Ändern des Codeverhaltens zur Kompilierzeit) .

Obwohl es vom Betriebssystem als Speicheradresse bei der Verwendung in API-Aufrufen behandelt wird, ist zu beachten, dass es von VBA als vorzeichenbehafteter Typ behandelt wird (und daher einem vorzeichenlosen Überlauf ohne Vorzeichen unterliegt). Aus diesem Grund sollte für eine Zeigerarithmetik, die mit LongPtrs ausgeführt wird, kein Vergleich mit `>` oder `<` . Diese "Eigenart" macht es außerdem möglich, dass das Hinzufügen einfacher Offsets, die auf gültige Adressen im Speicher verweisen, Überlauffehler verursachen kann. Daher ist bei der Arbeit mit Zeigern in VBA Vorsicht geboten.

Die Casting-Funktion zum Konvertieren in einen LongPtr ist `CLngPtr()` . Bei Umsetzungen von Fließkommatypen wird das Ergebnis auf den nächsten ganzzahligen Wert gerundet, wobei aufgerundet wird. (Da es sich jedoch normalerweise um eine Speicheradresse handelt, ist die Verwendung als Zuweisungsziel für eine Fließkommaberechnung bestenfalls gefährlich).

Dezimal

```
Dim Value As Variant
Value = CDec(1.234)

'Set Value to the smallest possible Decimal value
Value = CDec("0.0000000000000000000000000001")
```

Der `Decimal` Datentyp ist *nur* als `Variant` -Subtyp verfügbar. Sie müssen daher jede Variable, die ein `Decimal` enthalten muss, als `Variant` deklarieren und *anschließend* mit der `CDec` Funktion einen `Decimal` `CDec` . Das Schlüsselwort `Decimal` ist ein reserviertes Wort (was darauf hindeutet, dass VBA schließlich erstklassige Unterstützung für den Typ hinzufügte). `Decimal` kann `Decimal` nicht als Variablen- oder Prozedurname verwendet werden.

Der `Decimal` Typ benötigt 14 Byte Speicher (zusätzlich zu den Bytes, die von der übergeordneten Variante benötigt werden) und kann Zahlen mit bis zu 28 Dezimalstellen speichern. Für Zahlen ohne Nachkommastellen liegt der zulässige Wertebereich bei -79.228.162.514.264.337.593.543.950.335 bis +79.228.162.514.264.337.593.543.950.335. Bei Zahlen mit maximal 28 Dezimalstellen liegt der zulässige Wertebereich zwischen -7,9228162514264337593543950335 und +7,9228162514264337593543950335.

Datentypen und Grenzwerte online lesen: https://riptutorial.com/de/vba/topic/3418/datentypen-und-grenzwerte

Kapitel 17: Datums-Uhrzeit-Manipulation

Examples

Kalender

VBA unterstützt zwei Kalender: Gregorian und Hijri

Die `Calendar` wird verwendet, um den aktuellen Kalender zu ändern oder anzuzeigen.

Die zwei Werte für den Kalender sind:

Wert	Konstante	Beschreibung
0	vbCalGreg	Gregorianischer Kalender (Standard)
1	vbCalHijri	Hijri-Kalender

Beispiel

```
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
```

Dieses Sub druckt folgendes aus:

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

Rufen Sie System DateTime ab

VBA unterstützt 3 integrierte Funktionen, um Datum und / oder Uhrzeit von der Systemuhr abzurufen.

Funktion	Rückgabetyp	Rückgabewert
Jetzt	Datum	Gibt das aktuelle Datum und die aktuelle Uhrzeit zurück
Datum	Datum	Gibt den Datumsteil des aktuellen Datums und der aktuellen Uhrzeit zurück
Zeit	Datum	Gibt den Zeitabschnitt des aktuellen Datums und der aktuellen Uhrzeit zurück

```
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
```

Timerfunktion

Die `Timer` Funktion gibt eine Single zurück, die die Anzahl der seit Mitternacht verstrichenen Sekunden angibt. Die Genauigkeit beträgt eine Hundertstelsekunde.

```
Sub TimerExample()

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

End Sub
```

Da die `Now` und `Time` Funktionen nur sekundengenau sind, bietet `Timer` eine bequeme Möglichkeit, die Genauigkeit der Zeitmessung zu erhöhen:

```
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 () prüft, ob ein Ausdruck ein gültiges Datum ist oder nicht. Gibt einen `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
```

Extraktionsfunktionen

Diese Funktionen verwenden eine `Variant` , die in ein `Date` werden kann, und geben eine `Integer` die einen Teil eines Datums oder einer Uhrzeit darstellt. Wenn der Parameter nicht in ein `Date` , führt dies zu einem Laufzeitfehler 13: Typenkonflikt.

Funktion	Beschreibung	Rückgabewert
Jahr()	Gibt den Jahresanteil des Datumsarguments zurück.	Ganzzahl (100 bis 9999)

Funktion	Beschreibung	Rückgabewert
Monat()	Gibt den Monatsteil des Datumsarguments zurück.	Ganzzahl (1 bis 12)
Tag()	Gibt den Tagteil des Datumsarguments zurück.	Ganzzahl (1 bis 31)
Wochentag()	Gibt den Wochentag des Datumsarguments zurück. Akzeptiert ein optionales zweites Argument, das den ersten Tag der Woche definiert	Ganzzahl (1 bis 7)
Stunde()	Gibt den Stundenteil des Datumsarguments zurück.	Ganzzahl (0 bis 23)
Minute()	Gibt den Minutenteil des Datumsarguments zurück.	Ganzzahl (0 bis 59)
Zweite()	Gibt den zweiten Teil des Datumsarguments zurück.	Ganzzahl (0 bis 59)

Beispiele:

```
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
```

DatePart () - Funktion

`DatePart()` ist auch eine Funktion, die einen Teil eines Datums `DatePart()` funktioniert jedoch anders und lässt mehr Möglichkeiten zu als die obigen Funktionen. Es kann zum Beispiel das Quartal des Jahres oder die Woche des Jahres zurückgeben.

Syntax:

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

Intervall Argument kann sein:

Intervall	Beschreibung
"JJJJ"	Jahr (100 bis 9999)
"y"	Tag des Jahres (1 bis 366)
"m"	Monat (1 bis 12)
"q"	Quartal (1 bis 4)
"ww"	Woche (1 bis 53)
"w"	Wochentag (1 bis 7)
"d"	Tag des Monats (1 bis 31)
"h"	Stunde (0 bis 23)
"n"	Minute (0 bis 59)
"s"	Zweite (0 bis 59)

firstdayofweek ist optional. Es ist eine Konstante, die den ersten Tag der Woche angibt. Wenn nicht angegeben, wird `vbSunday` angenommen.

firstweekofyear ist optional. Es ist eine Konstante, die die erste Woche des Jahres angibt. Wenn nicht angegeben, wird davon ausgegangen, dass die erste Woche die Woche ist, in der der 1. Januar stattfindet.

Beispiele:

```
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
```

Berechnungsfunktionen

DateDiff ()

`DateDiff()` gibt ein `Long` das die Anzahl der Zeitintervalle zwischen zwei angegebenen Daten angibt.

Syntax

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

- *Intervall* kann ein beliebiges Intervall sein, das in der `DatePart()` Funktion definiert ist
- *Datum1* und *Datum2* sind die beiden Datumsangaben, die Sie bei der Berechnung verwenden möchten
- *firstdayofweek* und *firstweekofyear* sind optional. Erläuterungen dazu finden Sie unter `DatePart()`

Beispiele

```
Sub DateDiffExamples()

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

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

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

End Sub
```

DateAdd ()

`DateAdd()` gibt ein `Date` zu dem ein angegebenes Datum oder Zeitintervall hinzugefügt wurde.

Syntax

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

- *Intervall* kann ein beliebiges Intervall sein, das in der `DatePart()` Funktion definiert ist
- *number* Numerischer Ausdruck, der die Anzahl der Intervalle angibt, die Sie hinzufügen

möchten. Es kann positiv sein (um Datumsangaben in der Zukunft zu erhalten) oder negativ (um Datumsangaben in der Vergangenheit zu erhalten).
- *Datum* ist ein `Date` oder ein Literal, das das Datum darstellt, zu dem das Intervall hinzugefügt wird

Beispiele:

```
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
```

Umwandlung und Schöpfung

CDate ()

`CDate()` konvertiert etwas von einem beliebigen Datentyp in einen `Date` Datentyp

```
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
```

Beachten Sie, dass VBA hat auch eine lose typisierte `CVDate()` , die Funktionen in der gleichen Weise wie die `CDate()` andere Funktion als die Rückkehr ein Datum eingegeben `Variant` anstelle eines stark typisierten `Date` . Die `CDate()` -Version sollte bevorzugt werden, wenn an einen `Date` Parameter übergeben oder einer `Date` Variablen `CVDate()` , und die `CVDate()` -Version sollte bevorzugt werden, wenn an einen `Variant` Parameter übergeben oder einer `Variant` Variablen `CVDate()` wird. Dies vermeidet ein implizites Casting.

DateSerial ()

`DateSerial()` Funktion dient zum Erstellen eines Datums. Es gibt ein `Date` für ein bestimmtes Jahr, einen Monat und einen Tag zurück.

Syntax:

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

Mit Argumenten für Jahr, Monat und Tag sind Ganzzahlen gültig (Jahr von 100 bis 9999, Monat von 1 bis 12, Tag von 1 bis 31).

Beispiele

```
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
```

Beachten Sie, dass `DateSerial()` "ungültige" Daten akzeptiert und daraus ein gültiges Datum berechnet. Dies kann kreativ für immer genutzt werden:

Positives Beispiel

```
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
```

Es ist jedoch wahrscheinlicher, dass Trauer verursacht wird, wenn versucht wird, ein Datum aus nicht validierten Benutzereingaben zu erstellen:

Negatives Beispiel

```
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
```

Datums-Uhrzeit-Manipulation online lesen: https://riptutorial.com/de/vba/topic/4452/datums-uhrzeit-manipulation

Kapitel 18: Eine benutzerdefinierte Klasse erstellen

Bemerkungen

Dieser Artikel zeigt, wie Sie eine vollständige benutzerdefinierte Klasse in VBA erstellen. Es verwendet das Beispiel eines `DateRange` Objekts, da Start- und Enddatum häufig gemeinsam an Funktionen übergeben werden.

Examples

Hinzufügen einer Eigenschaft zu einer Klasse

Eine `Property` Prozedur ist eine Reihe von Anweisungen, mit denen eine benutzerdefinierte Eigenschaft eines Moduls abgerufen oder geändert wird.

Es gibt drei Arten von Eigenschaftszugängen:

1. Eine `Get` Prozedur, die den Wert einer Eigenschaft zurückgibt.
2. Eine `Let` Prozedur, die einem `Object` einen (Nicht- `Object`) Wert zuweist.
3. Eine `Set` Prozedur, die eine `Object` zuweist.

Eigenschafts-Accessoren werden häufig paarweise definiert, wobei für jede Eigenschaft sowohl `Get` als auch `Let` / `Set` werden. Eine Eigenschaft mit nur einer `Get` Prozedur wäre schreibgeschützt, während eine Eigenschaft mit nur einer `Let` / `Set` Prozedur nur schreibgeschützt wäre.

Im folgenden Beispiel sind vier Eigenschaftszugriffe für die `DateRange` Klasse definiert:

1. `StartDate` (*Lesen / Schreiben*). Datumswert, der das frühere Datum in einem Bereich darstellt. Jede Prozedur verwendet den Wert der `mStartDate` .
2. `EndDate` (*Lesen / Schreiben*). Datumswert, der das spätere Datum in einem Bereich darstellt. Jede Prozedur verwendet den Wert der `mEndDate` .
3. `DaysBetween` (*schreibgeschützt*). Berechneter Integer-Wert, der die Anzahl der Tage zwischen den beiden Daten angibt. Da es nur eine `Get` Prozedur gibt, kann diese Eigenschaft nicht direkt geändert werden.
4. `RangeToCopy` (`RangeToCopy` *Schreiben*). Eine `Set` Prozedur, die zum Kopieren der Werte eines vorhandenen `DateRange` Objekts verwendet wird.

```
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
```

Funktionalität zu einer Klasse hinzufügen

Jede öffentliche `Sub` , `Function` oder `Property` innerhalb eines Klassenmoduls kann aufgerufen werden, indem dem Aufruf eine Objektreferenz vorangestellt wird:

```
Object.Procedure
```

In einer `DateRange` Klasse kann ein `Sub` verwendet werden, um dem Enddatum eine Anzahl von Tagen hinzuzufügen:

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

Eine `Function` könnte den letzten Tag des nächsten Monatsendes zurückgeben (beachten Sie, dass `GetFirstDayOfMonth` außerhalb der Klasse nicht sichtbar ist, da es privat ist):

```
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
```

Prozeduren können Argumente eines beliebigen Typs akzeptieren, einschließlich Referenzen auf Objekte der definierten Klasse.

Im folgenden Beispiel wird getestet, ob das aktuelle `DateRange` Objekt ein Start- und Enddatum hat, das das Start- und Enddatum eines anderen `DateRange` Objekts enthält.

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

Beachten Sie die Verwendung der `Me` Notation, um auf den Wert des Objekts zuzugreifen, das den Code ausführt.

Klassenmodulumfang, Instanziierung und Wiederverwendung

Standardmäßig ist ein neues Klassenmodul eine Private-Klasse. Daher ist es *nur* für die Instantiierung und Verwendung innerhalb des VBProject verfügbar, in dem es definiert ist. Sie können die Klasse überall im *selben* Projekt deklarieren, instanziieren und verwenden:

```
'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
```

Oft schreiben Sie Klassen, die Sie in anderen Projekten verwenden möchten, *ohne* das Modul zwischen den Projekten *zu* kopieren. Wenn Sie in `ProjectA` eine Klasse mit dem Namen `List` definieren und diese Klasse in `ProjectB` möchten, müssen Sie 4 Aktionen ausführen:

1. Ändern Sie die instancing-Eigenschaft der `List` Klasse in `ProjectA` im Eigenschaftenfenster von `Private` in `PublicNotCreatable`

2. Erstellen Sie eine öffentliche "Factory" -Funktion in `ProjectA` , die eine Instanz einer `List` Klasse erstellt und zurückgibt. Normalerweise enthält die Factory-Funktion Argumente für die Initialisierung der Klasseninstanz. Die Fabrik - Funktion ist erforderlich , da die Klasse kann verwendet werden `ProjectB` aber `ProjectB` kann nicht direkt eine Instanz erstellen `ProjectA` ,s - Klasse.

   ```
   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. `Tools..References...` in `ProjectB` über das Menü `Tools..References...` einen Verweis auf `ProjectA` `Tools..References...` .

4. `ProjectB` in `ProjectB` eine Variable, und weisen Sie ihr mithilfe der Factory-Funktion von `ProjectA` eine Instanz von `List`

```
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
```

Eine benutzerdefinierte Klasse erstellen online lesen:
https://riptutorial.com/de/vba/topic/4464/eine-benutzerdefinierte-klasse-erstellen

Kapitel 19: Fehlerbehandlung

Examples

Fehlerzustände vermeiden

Wenn ein Laufzeitfehler auftritt, sollte guter Code damit umgehen. Die beste Strategie zur Fehlerbehandlung besteht darin, Code zu schreiben, der auf Fehlerbedingungen prüft und einfach die Ausführung von Code vermeidet, der zu einem Laufzeitfehler führt.

Ein Schlüsselelement bei der Reduzierung von Laufzeitfehlern ist das Schreiben kleiner Prozeduren, *die eine Sache tun* . Je weniger Gründe für das Scheitern von Prozeduren erforderlich sind, desto einfacher ist es, den gesamten Code zu debuggen.

Laufzeitfehler vermeiden 91 - Objekt oder Mit Blockvariable nicht gesetzt:

Dieser Fehler wird ausgelöst, wenn ein Objekt verwendet wird, bevor seine Referenz zugewiesen wird. Möglicherweise verfügt eine Prozedur über einen Objektparameter:

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

Wenn dem `target` keine Referenz zugewiesen wurde, löst der obige Code einen Fehler aus, der leicht vermieden werden kann, indem geprüft wird, ob das Objekt eine tatsächliche Objektreferenz enthält:

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

Wenn dem `target` keine Referenz zugewiesen ist, wird die nicht zugewiesene Referenz niemals verwendet und es tritt kein Fehler auf.

Diese Methode, eine Prozedur vorzeitig zu beenden, wenn ein oder mehrere Parameter nicht gültig sind, wird als *Guard-Klausel bezeichnet* .

Laufzeitfehler 9 vermeiden - Index außerhalb des gültigen Bereichs:

Dieser Fehler wird ausgelöst, wenn auf ein Array außerhalb seiner Grenzen zugegriffen wird.

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

Bei einem Index, der größer als die Anzahl der Arbeitsblätter im `ActiveWorkbook` , führt der obige Code zu einem Laufzeitfehler. Eine einfache Schutzklausel kann das vermeiden:

```
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
```

Die meisten Laufzeitfehler können vermieden werden, indem die Werte, die wir verwenden, sorgfältig überprüft werden, *bevor* wir sie verwenden, und entsprechend in einen anderen Ausführungspfad verzweigt werden. Verwenden Sie dazu eine einfache `If` Anweisung, die keine Annahmen macht und die Parameter einer Prozedur oder sogar die der Prozeduren validiert Körper der größeren Verfahren.

On Error-Anweisung

Selbst mit *Wachklauseln* kann man nicht realistisch *immer* alle möglichen Fehlerbedingungen berücksichtigen, die im Körper einer Prozedur auftreten können. Die `On Error GoTo` Anweisung weist VBA an, zu einer *Zeilenbezeichnung* zu springen und den "Fehlerbehandlungsmodus" aufzurufen, wenn zur Laufzeit ein unerwarteter Fehler auftritt. Einen Fehler nach dem Umgang mit Code kann mit dem in „normale“ Ausführung *wieder* zurück `Resume` Schlüsselwort.

Linienbezeichnungen Subroutinen bezeichnen: da Subroutinen von Legacy - BASIC - Code stammt und verwendet `GoTo` und `GoSub` Sprünge und `Return` Aussagen zurück auf die „main“ Routine zu springen, es ist ziemlich einfach schwer zu folgen *Spaghetti - Code* zu schreiben , wenn die Dinge nicht streng strukturiert . Aus diesem Grund ist es am besten:

- Eine Prozedur hat **nur ein** Unterprogramm zur Fehlerbehandlung
- Das Unterprogramm zur Fehlerbehandlung **läuft immer nur in einem Fehlerzustand**

Dies bedeutet, dass eine Prozedur, die ihre Fehler behandelt, folgendermaßen strukturiert ist:

```
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
```

Fehlerbehandlungsstrategien

Manchmal möchten Sie verschiedene Fehler mit unterschiedlichen Aktionen behandeln. In diesem Fall werden Sie das globale `Err` Objekt untersuchen, das Informationen zu dem aufgetretenen Fehler enthält - und entsprechend handeln:

```
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
```

Als allgemeine Richtlinie sollten Sie die Fehlerbehandlung für die gesamte Subroutine oder Funktion aktivieren und alle Fehler behandeln, die in ihrem Gültigkeitsbereich auftreten können. Wenn Sie nur Fehler im kleinen Codeabschnitt behandeln müssen, aktivieren Sie die Fehlerbehandlung auf derselben Ebene:

```
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
```

Linien Nummern

VBA unterstützt Zeilennummern (z. B. QBASIC) im Legacy-Stil. Die ausgeblendete `Erl` Eigenschaft kann verwendet werden, um die Zeilennummer zu identifizieren, die den letzten Fehler ausgelöst hat. Wenn Sie keine Zeilennummern verwenden, gibt `Erl` nur 0 zurück.

```
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
```

Wenn Sie Zeilennummern verwenden, aber nicht konsequent, dann `Erl` wird wieder *die letzte Zeilennummer vor der Anweisung, die den Fehler ausgelöst.*

```
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
```

Denken Sie daran, dass `Erl` auch nur eine `Integer` Genauigkeit hat und lautlos überläuft. Dies bedeutet, dass Zeilennummern außerhalb des Ganzzahlbereichs falsche Ergebnisse liefern:

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

Die Zeilennummer ist nicht ganz so relevant wie die Aussage, die den Fehler verursacht hat, und Nummerierungszeilen werden schnell langweilig und nicht sehr wartungsfreundlich.

Schlüsselwort fortsetzen

Eine Fehlerbehandlungs-Subroutine wird entweder:

- bis zum Ende der Prozedur ausführen. In diesem Fall wird die Ausführung in der aufrufenden Prozedur fortgesetzt.
- oder verwenden Sie das Schlüsselwort `Resume` , *um die* Ausführung innerhalb derselben Prozedur fortzusetzen.

Das `Resume` Schlüsselwort sollte nur innerhalb einer Fehlerbehandlungs-Subroutine verwendet werden, da der VBA-Fehler 20 "Resume ohne Fehler" auslöst, wenn VBA auf `Resume` stößt, ohne sich in einem Fehlerzustand zu befinden.

Es gibt mehrere Möglichkeiten, wie ein Unterprogramm zur Fehlerbehandlung das Schlüsselwort `Resume` :

- `Resume` für den `Resume` verwendet, wird die Ausführung **mit der Anweisung** fortgesetzt **, die den Fehler verursacht hat** . Wenn der Fehler *tatsächlich* , bevor Sie das gehandhabt wird , nicht, dann wird der gleiche Fehler wieder angehoben werden, und die Ausführung könnte

eine Endlosschleife eingeben.

- `Resume Next` setzt die Ausführung **der Anweisung unmittelbar nach** der Anweisung, die den Fehler verursacht hat, fort. Wenn der Fehler zuvor nicht *wirklich* behandelt wird, kann die Ausführung mit möglicherweise ungültigen Daten fortgesetzt werden, was zu logischen Fehlern und unerwartetem Verhalten führen kann.
- `Resume [line label]` setzt die Ausführung **an der angegebenen Zeilenbeschriftung** (oder Zeilennummer, wenn Sie Zeilennummern im Legacy-Stil verwenden) fort. Dies würde normalerweise die Ausführung eines Bereinigungscodes ermöglichen, bevor die Prozedur sauber beendet wird, z. B. das Sicherstellen, dass eine Datenbankverbindung geschlossen wird, bevor zum Aufrufer zurückgekehrt wird.

On Error Resume Next

Die `On Error` Anweisung selbst kann das Schlüsselwort `Resume` , um die VBA-Laufzeitumgebung anzuweisen **, alle Fehler** effektiv zu **ignorieren** .

> *Wenn der Fehler zuvor nicht **wirklich behandelt wird** , kann die Ausführung mit möglicherweise ungültigen Daten fortgesetzt werden, was zu **logischen Fehlern und unerwartetem Verhalten führen kann** .*

Die Betonung oben kann nicht genug betont werden. ***On Error Resume Next* ignoriert effektiv alle Fehler und schiebt sie unter den Teppich** . Ein Programm, das mit einem Laufzeitfehler bei ungültiger Eingabe explodiert, ist ein besseres Programm als eines, das mit unbekannten / unbeabsichtigten Daten weiterläuft - sei es nur, weil der Fehler viel einfacher zu identifizieren ist. `On Error Resume Next` kann `On Error Resume Next` leicht **ausblenden** .

Die `On Error` Anweisung hat einen prozeduralen Geltungsbereich. Aus diesem Grund sollte es in einer bestimmten Prozedur *normalerweise* nur **eine** solche `On Error` Anweisung geben.

Manchmal kann jedoch eine Fehlerbedingung nicht ganz vermieden werden, und ein Sprung in eine Fehlerbehandlungs-Subroutine, nur um `Resume Next` fühlt sich einfach nicht richtig an. In diesem speziellen Fall ist die bekannte zu möglicherweise fail - Anweisung kann zwischen zwei **eingewickelt** werden `On Error` - Anweisungen:

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

Die `On Error GoTo 0` Anweisung setzt die Fehlerbehandlung in der aktuellen Prozedur zurück, sodass alle weiteren Anweisungen, die einen Laufzeitfehler verursachen *, in dieser Prozedur nicht behandelt werden* und stattdessen den Aufrufstapel übergeben, bis er von einem aktiven Fehlerhandler abgefangen wird. Wenn der Aufrufstack keinen aktiven Fehlerhandler enthält, wird er als nicht behandelte Ausnahme behandelt.

```
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
```

Benutzerdefinierte Fehler

Wenn Sie eine spezialisierte Klasse schreiben, müssen Sie häufig spezifische Fehler auslösen, und Sie möchten eine saubere Methode für Benutzer- / Aufrufcode, um diese benutzerdefinierten Fehler zu behandeln. Eine gute Möglichkeit, dies zu erreichen, ist die Definition eines speziellen `Enum` Typs:

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

Die Verwendung der `vbObjectError` Konstante `vbObjectError` stellt sicher, dass die benutzerdefinierten Fehlercodes nicht mit reservierten / vorhandenen Fehlercodes überlappen. Es muss nur der erste Aufzählungswert explizit angegeben werden, da der zugrunde liegende Wert jedes `Enum` `Err_BarNotInitialized` um `1` größer ist als der vorherige Member, sodass der zugrunde liegende Wert von `Err_BarNotInitialized` implizit `vbObjectError + 1025` .

Erhöhen Sie Ihre eigenen Laufzeitfehler

Mit der `Err.Raise` Anweisung kann ein Laufzeitfehler `Err.Raise` werden, sodass der benutzerdefinierte `Err_FooWasNotBarred` Fehler wie folgt `Err_FooWasNotBarred` werden kann:

```
Err.Raise Err_FooWasNotBarred
```

Die `Err.Raise` Methode kann auch benutzerdefinierte Parameter für `Description` und `Source` übernehmen. Aus diesem Grund ist es eine gute Idee, auch Konstanten für die Beschreibung der

benutzerdefinierten Fehler zu definieren:

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

Erstellen Sie dann eine dedizierte private Methode, um jeden Fehler zu melden:

```
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
```

Die Klassenimplementierung kann dann einfach diese speziellen Prozeduren aufrufen, um den Fehler zu melden:

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

Der Client-Code kann dann `Err_BarNotInitialized` wie jeden anderen Fehler in seiner eigenen Fehlerbehandlungs-Subroutine behandeln.

Hinweis: Das ältere Schlüsselwort `Error` kann auch anstelle von `Err.Raise` ist jedoch veraltet / veraltet.

Fehlerbehandlung online lesen: https://riptutorial.com/de/vba/topic/3211/fehlerbehandlung

Kapitel 20: Flusssteuerungsstrukturen

Examples

Fall auswählen

`Select Case` kann verwendet werden, wenn viele verschiedene Bedingungen möglich sind. Die Bedingungen werden von oben nach unten geprüft und nur der erste übereinstimmende Fall wird ausgeführt.

```
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
```

Die Logik des `Select Case` Blocks kann invertiert werden, um das Testen verschiedener Variablen zu unterstützen. In diesem Szenario können wir auch logische Operatoren verwenden:

```
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
```

Case-Anweisungen können auch arithmetische Operatoren verwenden. Wenn ein arithmetischer Operator für den Wert " `Select Case` auswählen" verwendet wird, sollte das Schlüsselwort " `Is` vorangestellt werden:

```
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
```

Für jede Schleife

Das `For Each` Schleifenkonstrukt ist ideal für das Durchlaufen aller Elemente einer Auflistung.

```
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
```

Verwenden Sie `For Each` Element, wenn Sie Objektauflistungen durchlaufen:

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

Vermeiden Sie `For Each` beim Durchlaufen von Arrays; Eine `For` Schleife bietet eine wesentlich bessere Leistung bei Arrays. Umgekehrt bietet eine `For Each` Schleife eine bessere Leistung, wenn eine `Collection` durchlaufen wird.

Syntax

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

Auf das `Next` Schlüsselwort kann optional die Iterator-Variable folgen. Dies kann helfen, geschachtelte Schleifen zu klären, obwohl es bessere Möglichkeiten gibt, geschachtelten Code zu klären, beispielsweise das Extrahieren der inneren Schleife in eine eigene Prozedur.

```
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
```

Machen Sie eine Schleife

```
 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
```

While-Schleife

```
'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
```

Für Schleife

Die `For` Schleife wird verwendet, um den eingeschlossenen Codeabschnitt eine bestimmte Anzahl von Malen zu wiederholen. Das folgende einfache Beispiel veranschaulicht die grundlegende Syntax:

```
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
```

Der obige Code deklariert eine Ganzzahl `i` . Die `For` Schleife weist `i` jeden Wert zwischen 1 und 10 zu und führt dann `Debug.Print i` - dh der Code gibt die Zahlen 1 bis 10 in das unmittelbare Fenster aus. Beachten Sie, dass die Schleifenvariable durch die `Next` Anweisung inkrementiert wird, d. H. Nachdem der eingeschlossene Code ausgeführt wurde, bevor er ausgeführt wird.

Standardmäßig wird der Zähler bei jeder Ausführung der Schleife um 1 erhöht. Es kann jedoch ein `Step` angegeben werden, um den Betrag des Inkrements entweder als Literal oder als Rückgabewert einer Funktion zu ändern. Wenn der Startwert, der Endwert oder der `Step` eine Gleitkommazahl ist, wird diese auf den nächsten ganzzahligen Wert gerundet. `Step` kann entweder ein positiver oder ein negativer Wert sein.

```
Dim i As Integer
For i = 1 To 10 Step 2
    Debug.Print i       'Prints 1, 3, 5, 7, and 9
Next
```

Im Allgemeinen wird eine `For` Schleife in Situationen verwendet, in denen vor Beginn der Schleife bekannt ist, wie oft der eingeschlossene Code ausgeführt werden soll (andernfalls ist eine `Do` oder

`While` Schleife möglicherweise geeigneter). Dies liegt daran, dass die Beendigungsbedingung nach dem ersten Eintritt in die Schleife festgelegt ist, wie der folgende Code zeigt:

```
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
```

Eine `For` Schleife kann mit der `Exit For` Anweisung vorzeitig beendet werden:

```
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
```

Flusssteuerungsstrukturen online lesen:
https://riptutorial.com/de/vba/topic/1873/flusssteuerungsstrukturen

Kapitel 21: Häufig verwendete String-Manipulation

Einführung

Schnelle Beispiele für MID LEFT- und RIGHT-Stringfunktionen mit INSTR FIND und LEN.

Wie finden Sie den Text zwischen zwei Suchbegriffen (sprich: nach einem Doppelpunkt und vor einem Komma)? Wie erhalten Sie den Rest eines Wortes (mit MID oder mit RIGHT)? Welche dieser Funktionen verwenden nullbasierte Params und Rückkehrcodes im Vergleich zu One-based? Was passiert, wenn etwas schief geht? Wie gehen sie mit leeren Strings, unbefundeten Ergebnissen und negativen Zahlen um?

Examples

String-Manipulation häufig verwendete Beispiele

Besseres MID () und andere Beispiele für die String-Extraktion, die derzeit nicht im Web verfügbar sind. Bitte helfen Sie mir, ein gutes Beispiel zu machen, oder ergänzen Sie dieses hier. Etwas wie das:

```
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) '...
```

Fühlen Sie sich frei, dieses Beispiel zu bearbeiten und es besser zu machen. Solange es klar bleibt und allgemeine Gebrauchspraktiken hat.

Häufig verwendete String-Manipulation online lesen:
https://riptutorial.com/de/vba/topic/8890/haufig-verwendete-string-manipulation

Kapitel 22: Länge der Saiten messen

Bemerkungen

Die Länge einer Zeichenfolge kann auf zwei Arten gemessen werden: Das am häufigsten verwendete Längenmaß ist die Anzahl der Zeichen, die die `Len` Funktionen verwenden. VBA kann jedoch auch die Anzahl der Bytes mithilfe der `LenB` Funktionen `LenB` . Ein Doppelbyte- oder Unicode-Zeichen ist mehr als ein Byte lang.

Examples

Verwenden Sie die Len-Funktion, um die Anzahl der Zeichen in einer Zeichenfolge zu bestimmen

```
Const baseString As String = "Hello World"

Dim charLength As Long

charLength = Len(baseString)
'charlength = 11
```

Verwenden Sie die LenB-Funktion, um die Anzahl der Bytes in einer Zeichenfolge zu bestimmen

```
Const baseString As String = "Hello World"

Dim byteLength As Long

byteLength = LenB(baseString)
'byteLength = 22
```

Bevorzugen Sie, wenn Len (myString) = 0 Then` über `If myString =" "Then"

Bei der Prüfung, ob eine Zeichenfolge eine Länge von Null hat, ist es besser und effizienter, die Länge der Zeichenfolge zu prüfen, anstatt die Zeichenfolge mit einer leeren Zeichenfolge zu vergleichen.

```
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
```

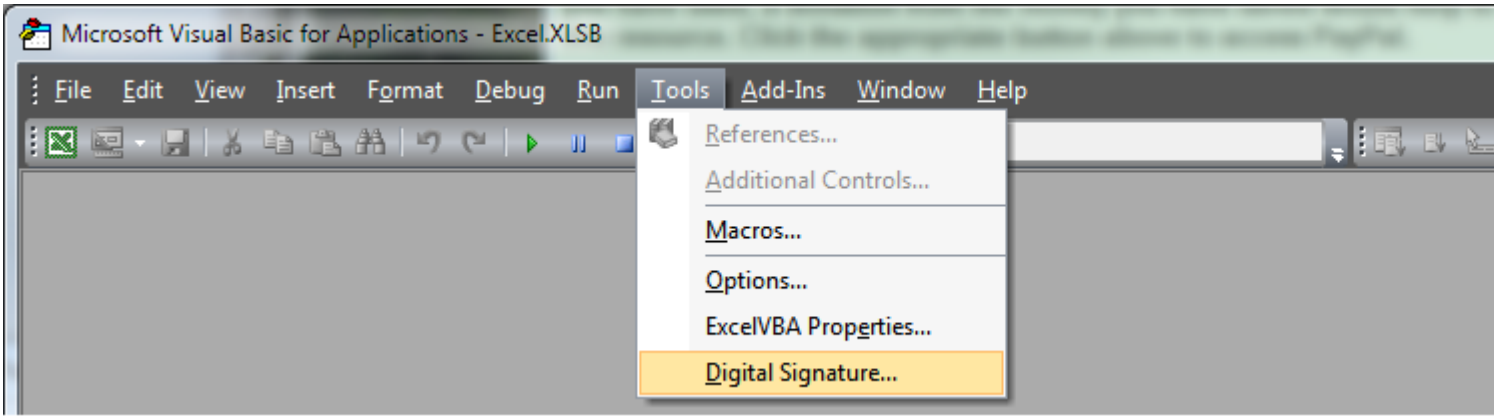
Länge der Saiten messen online lesen: https://riptutorial.com/de/vba/topic/3576/lange-der-saiten-messen

Kapitel 23: Makrosicherheit und Signatur von VBA-Projekten / -Modulen

Examples

Erstellen Sie ein gültiges digitales selbstsigniertes Zertifikat SELFCERT.EXE

Um Makros auszuführen und die Sicherheit zu gewährleisten, die Office-Anwendungen gegen schädlichen Code bereitstellen, ist es erforderlich, das VBAProject.OTM über den *VBA-Editor> Tools> Digitale Signatur* digital zu signieren.

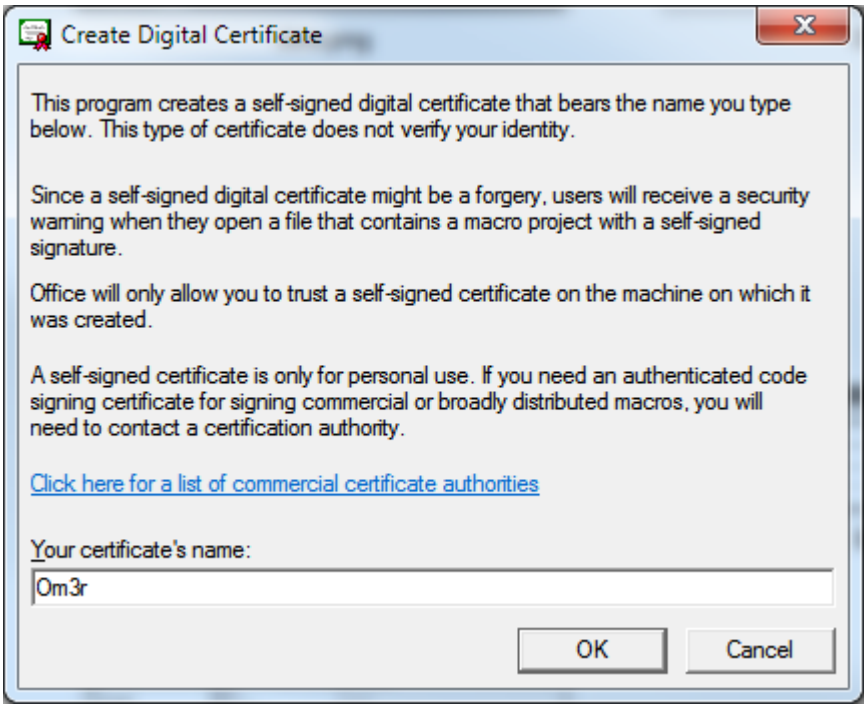


Office enthält ein Dienstprogramm zum Erstellen eines selbstsignierten digitalen Zertifikats, das Sie auf dem PC zum Signieren Ihrer Projekte verwenden können.

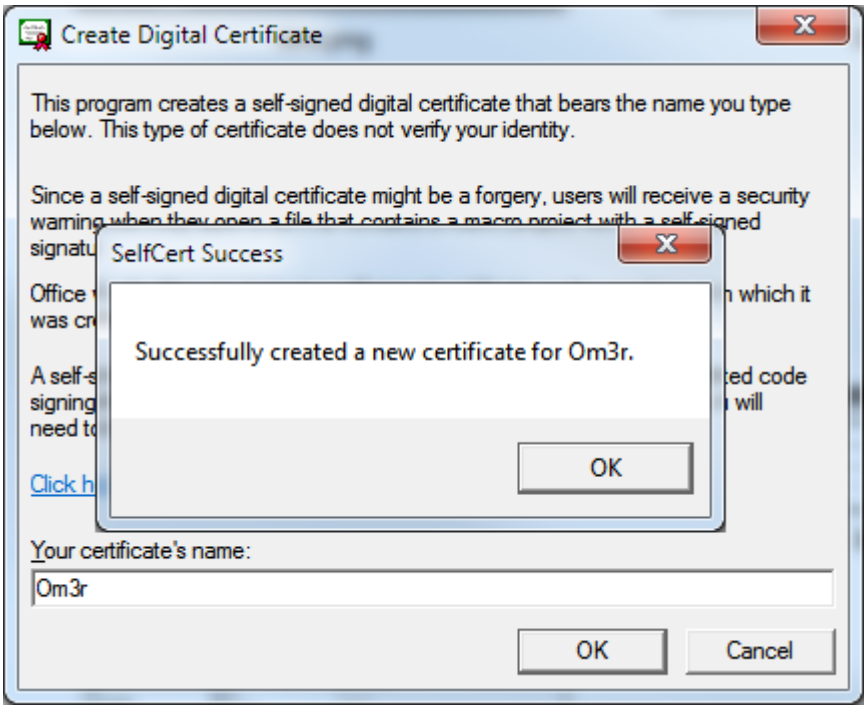
Dieses Dienstprogramm **SELFCERT.EXE** befindet sich im Office-Programmordner.

Klicken Sie auf Digitales Zertifikat für VBA-Projekte, um den Zertifikat- *Assistenten* zu öffnen.

Geben Sie im Dialogfeld einen geeigneten Namen für das Zertifikat ein und klicken Sie auf OK.

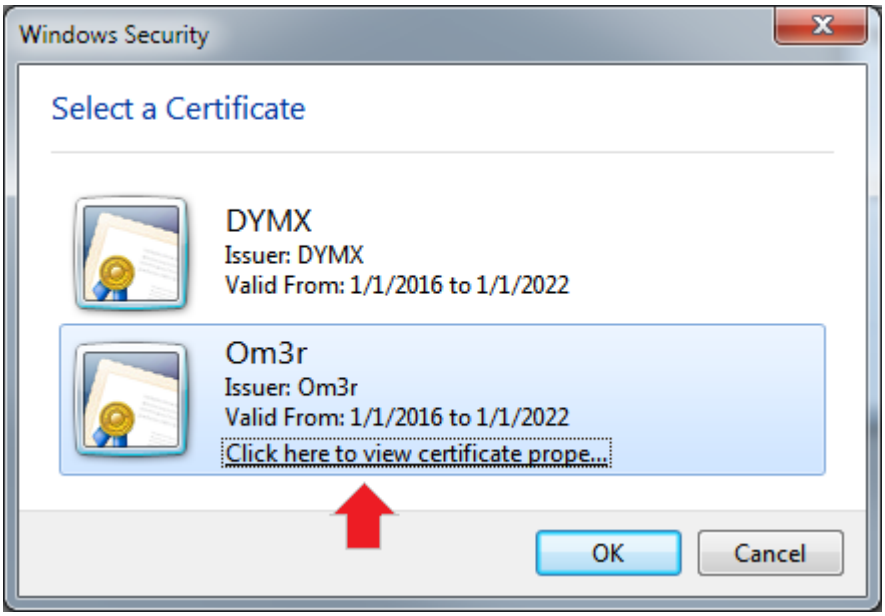


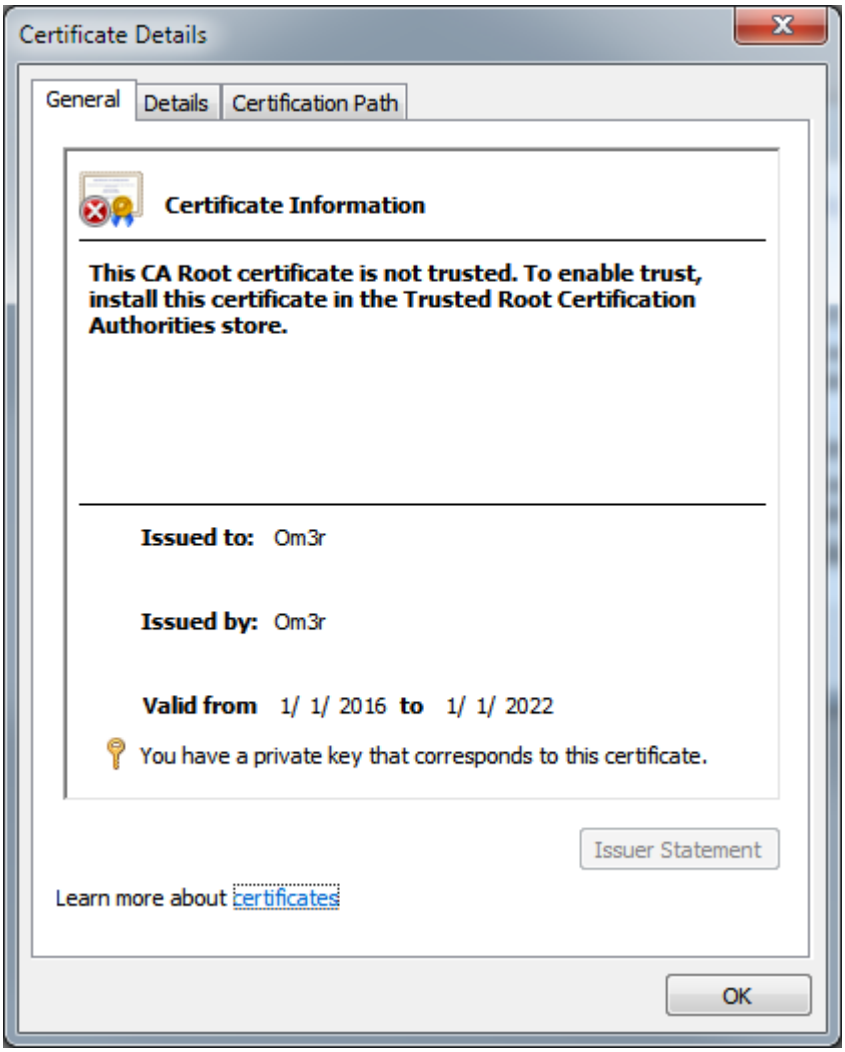
Wenn alles gut geht, sehen Sie eine Bestätigung:



Sie können nun den **SELFCERT-** Assistenten schließen und Ihre Aufmerksamkeit auf das von Ihnen erstellte Zertifikat **richten** .

Wenn Sie versuchen, das soeben erstellte Zertifikat zu verwenden, überprüfen Sie dessen Eigenschaften

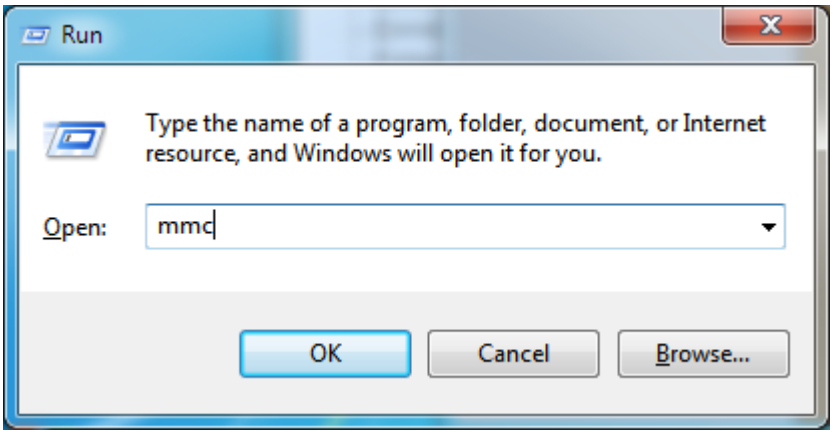




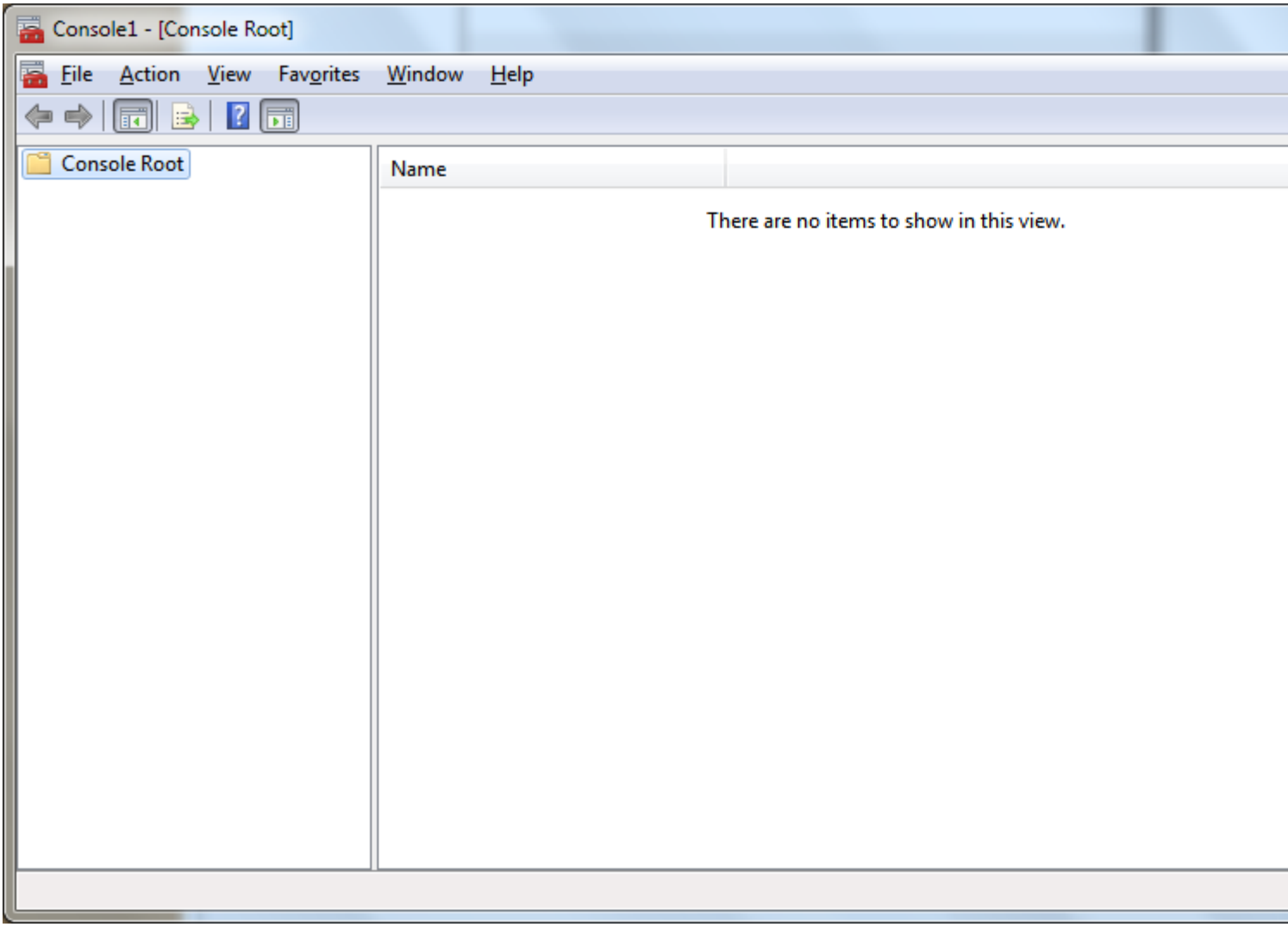
Sie werden feststellen, dass das Zertifikat nicht vertrauenswürdig ist und der Grund im Dialogfeld angegeben wird.

Das Zertifikat wurde im Speicher Aktueller Benutzer> Persönlich> Zertifikate erstellt. Es muss unter Lokaler Computer> Vertrauenswürdige Stammzertifizierungsstellen> Zertifikatsspeicher gehen. Sie müssen also den ersten exportieren und in den letzteren importieren.

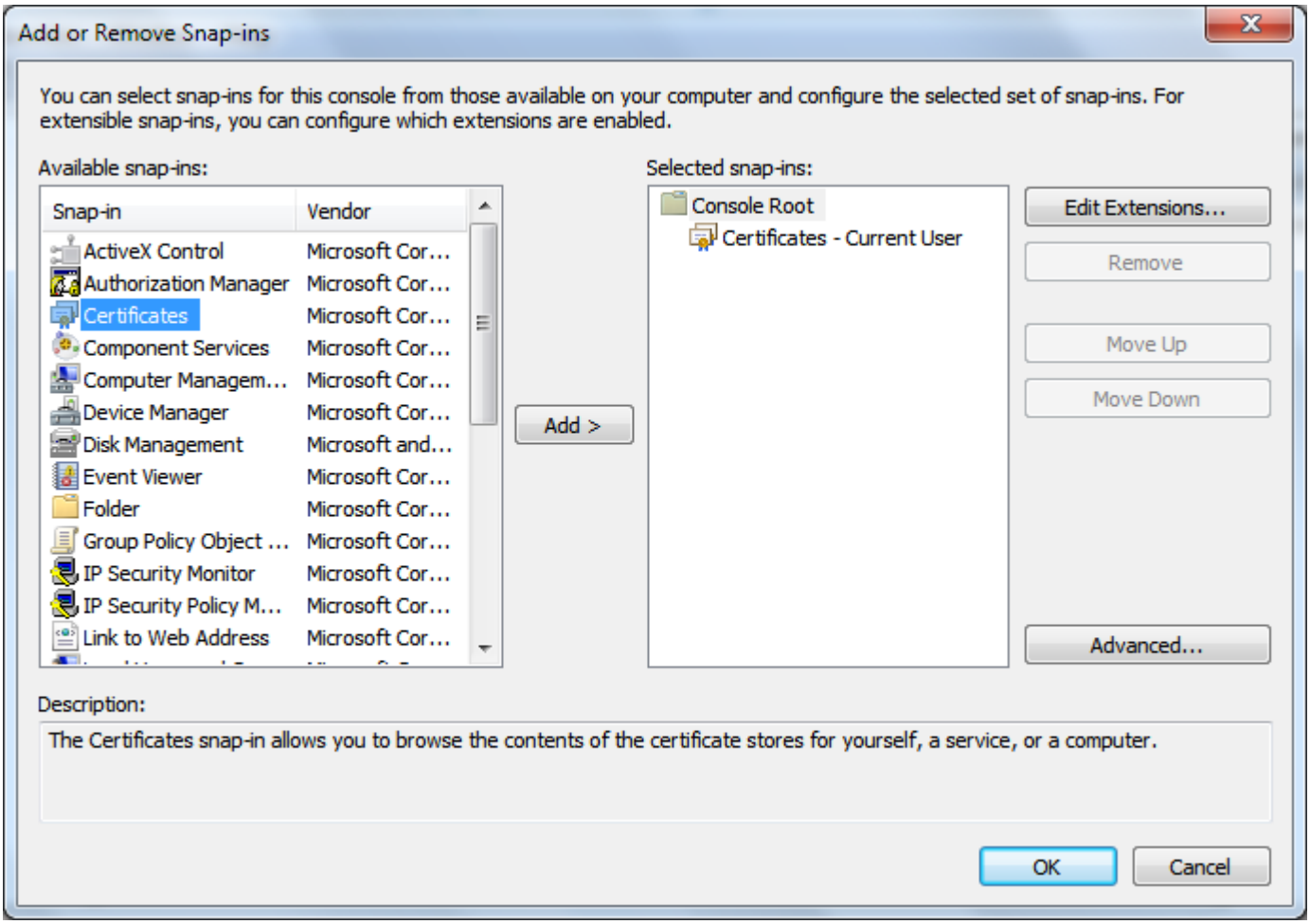
Drücken Sie die Windows- `Taste + R` , um das 'Run'-Fenster zu öffnen. Geben Sie dann 'mmc' in das Fenster ein und klicken Sie auf 'OK'.



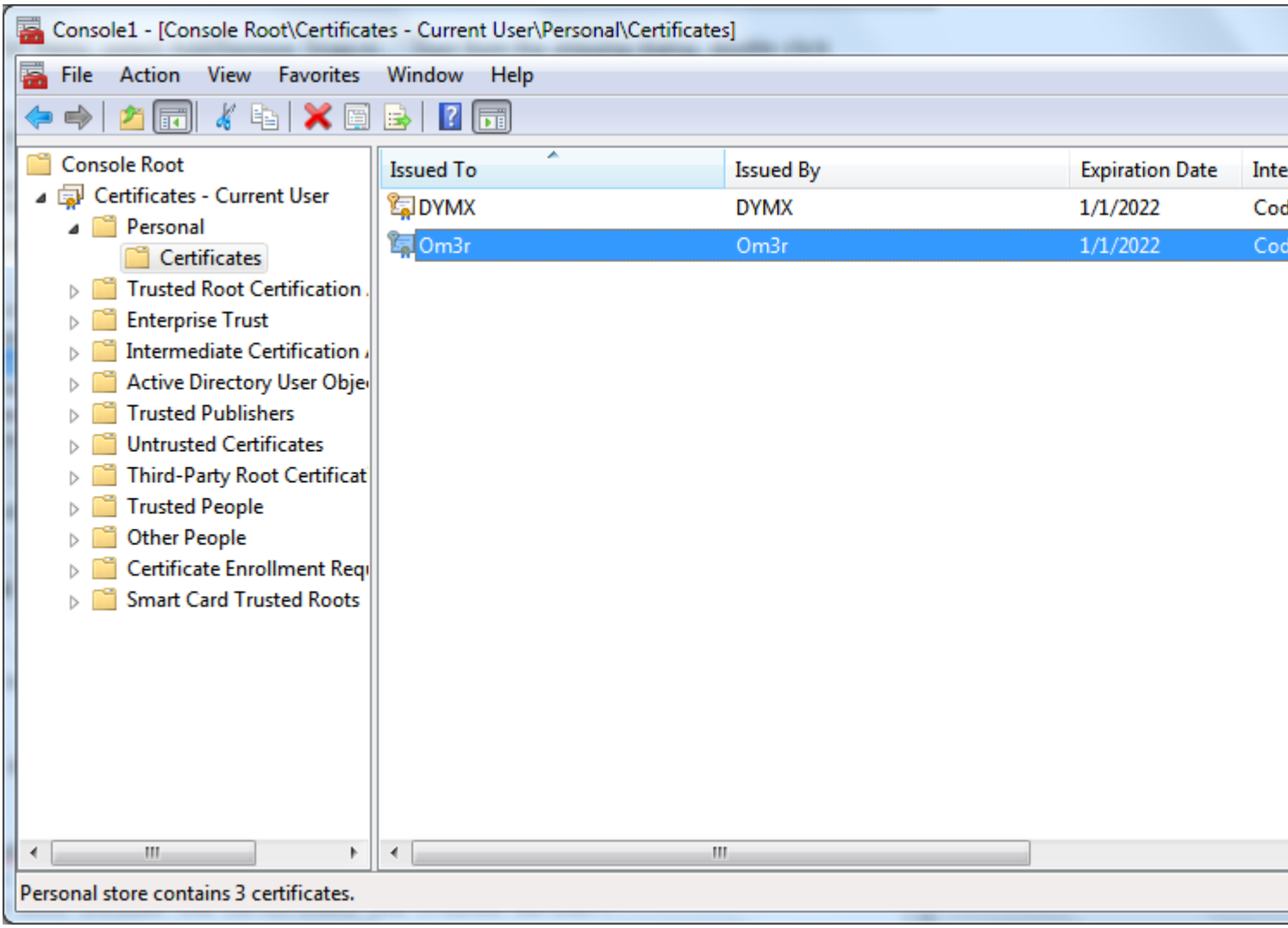
Die Microsoft Management Console wird geöffnet und sieht wie folgt aus.



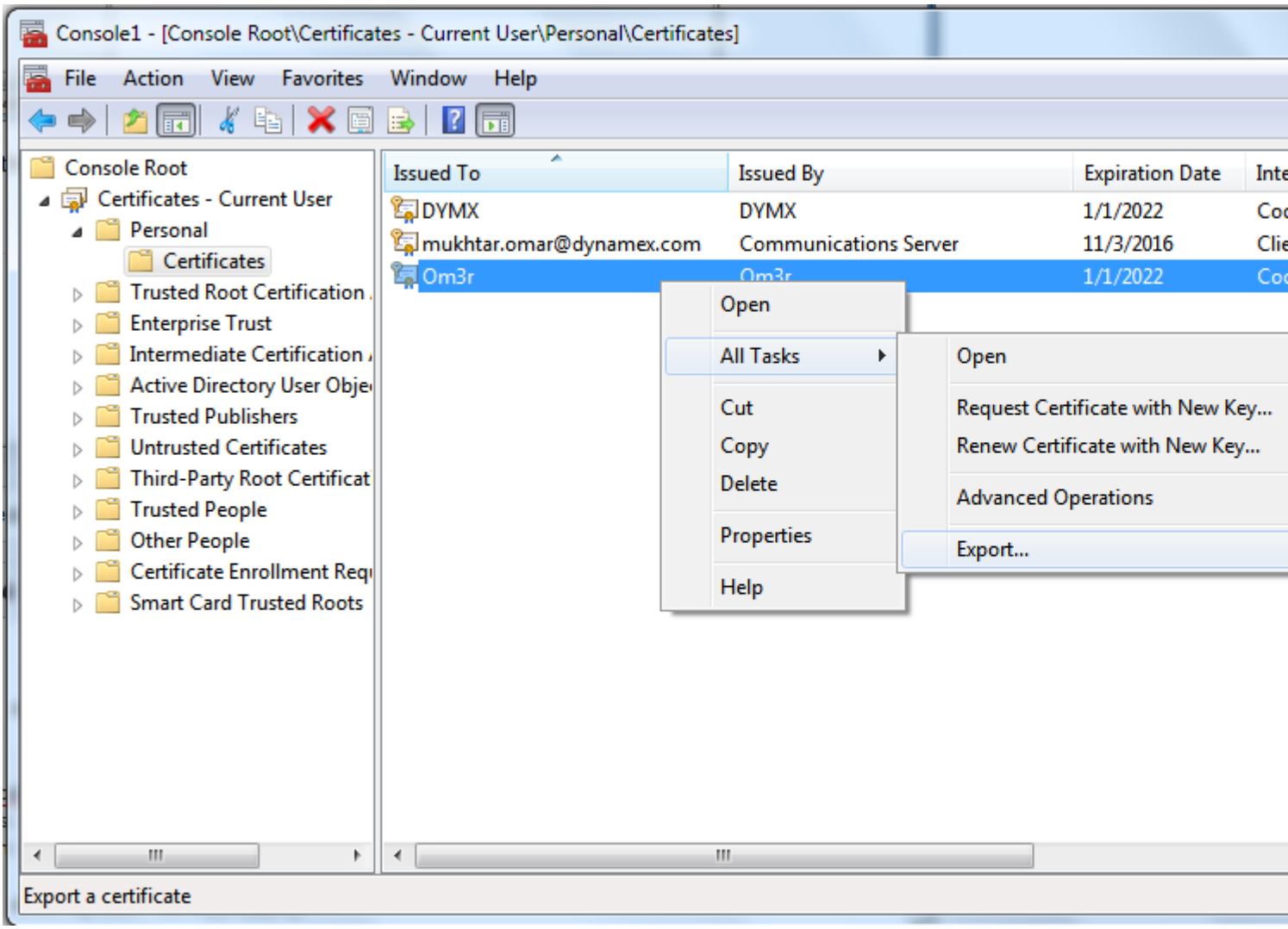
Wählen Sie im Menü Datei die Option Snap-In hinzufügen / entfernen aus. Doppelklicken Sie dann im folgenden Dialogfeld auf Zertifikate und klicken Sie dann auf OK



Erweitern Sie die Dropdown-Liste im linken Fenster für *Zertifikate - Aktueller Benutzer* , und wählen Sie Zertifikate wie unten gezeigt aus. Im mittleren Bereich werden dann die Zertifikate an diesem Speicherort angezeigt, die das zuvor erstellte Zertifikat enthalten:

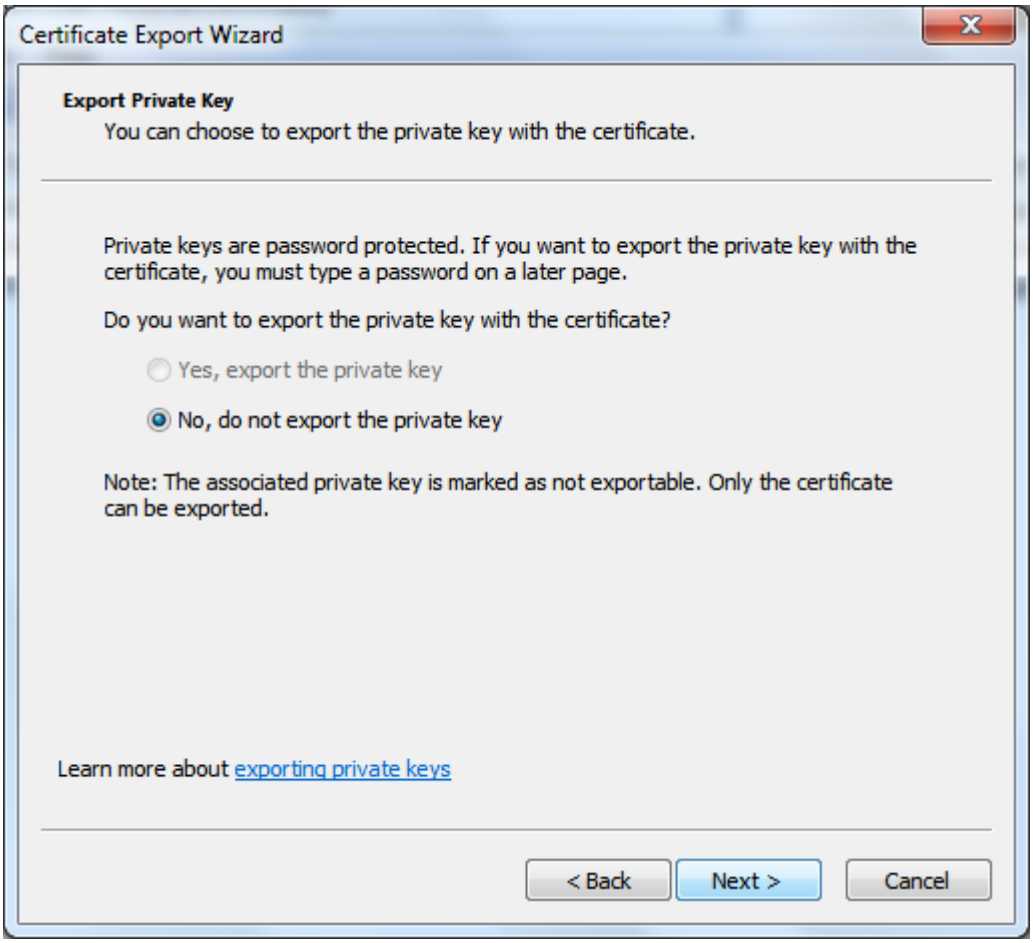


Klicken Sie mit der rechten Maustaste auf das Zertifikat, und wählen Sie Alle Aufgaben> Exportieren aus.

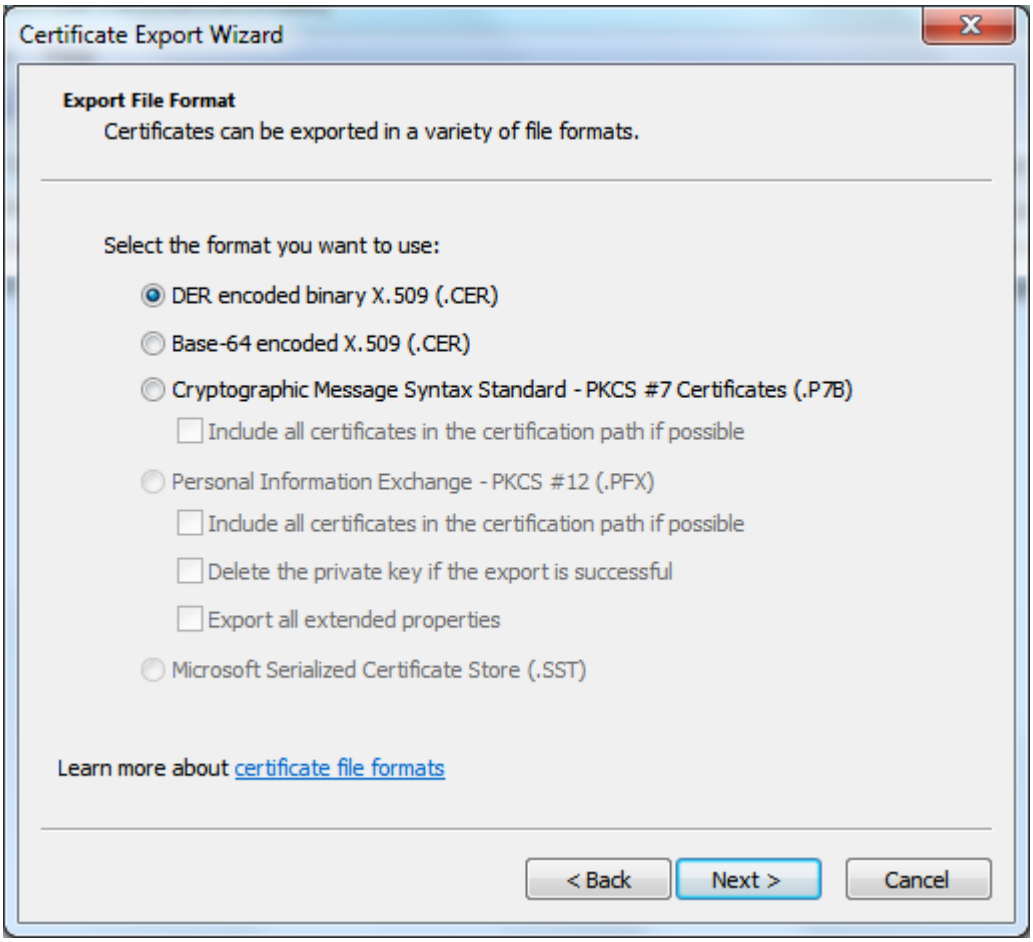


Export-Assistent

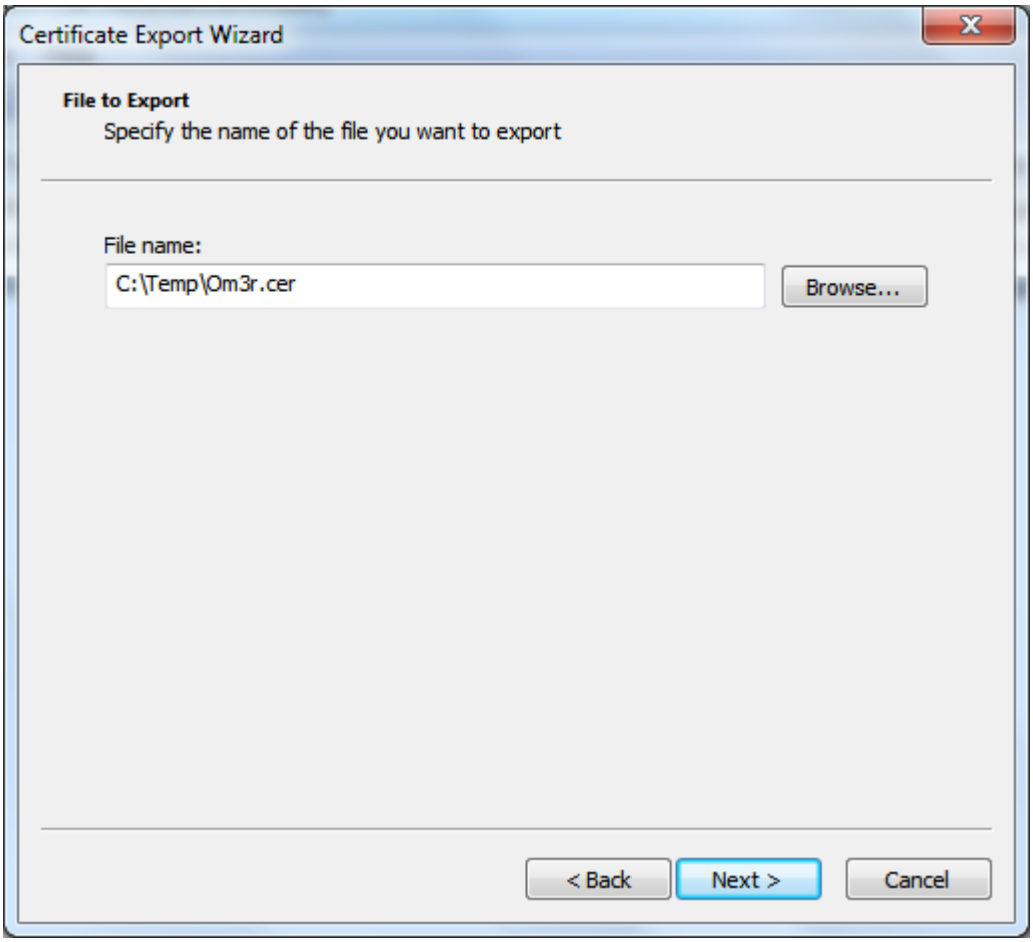
Weiter klicken



Die Option Nur eine vorgewählte Option ist verfügbar. Klicken Sie erneut auf "Weiter".



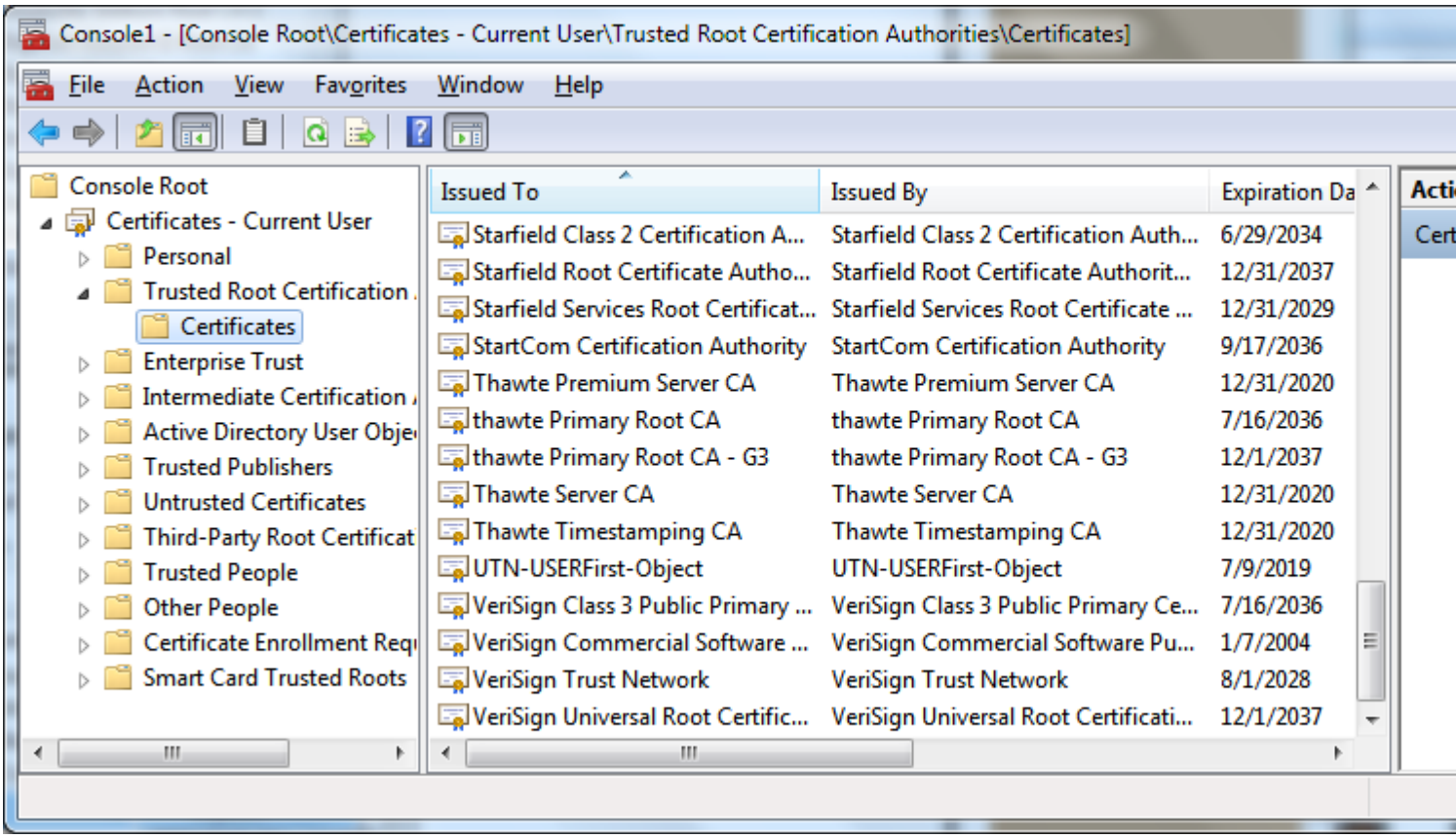
Das oberste Element ist bereits vorgewählt. Klicken Sie erneut auf Weiter, und wählen Sie einen Namen und einen Ort aus, um das exportierte Zertifikat zu speichern.



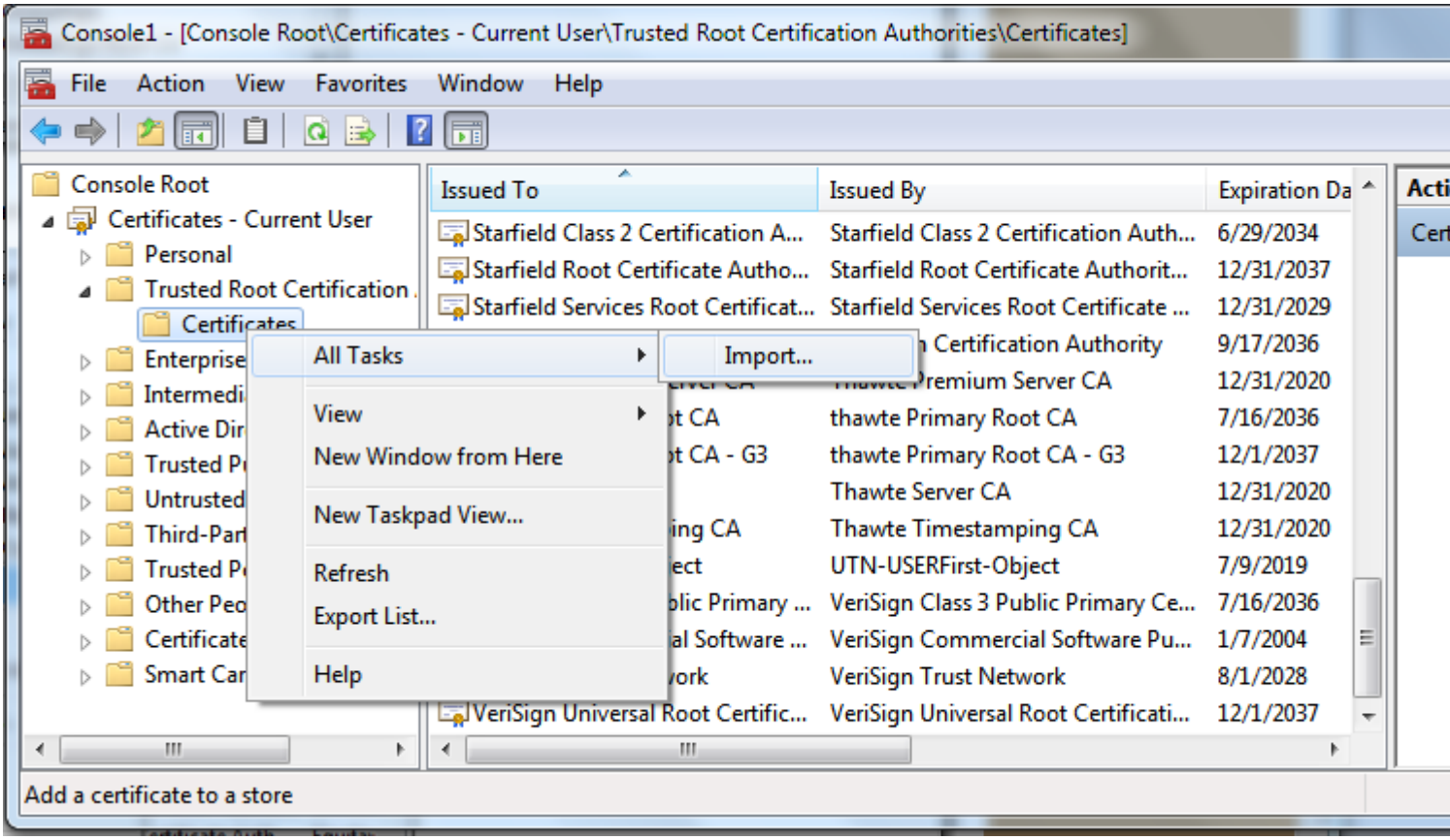
Klicken Sie erneut auf Weiter, um das Zertifikat zu speichern

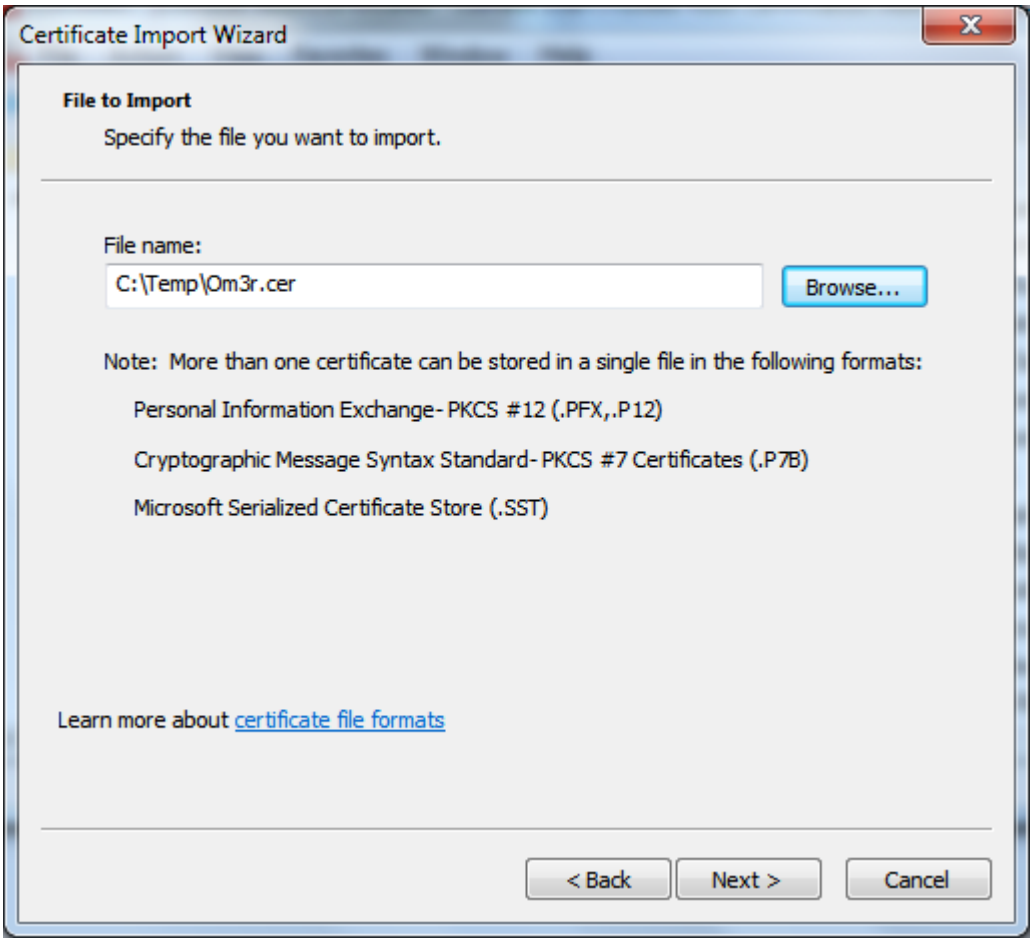
Sobald der Fokus wieder auf der Management Console liegt.

Erweitern Sie das *Zertifikate* - Menü und vom Vertrauenswürdige Stammzertifizierungsstellen - Menü wählen Sie *Zertifikate.*

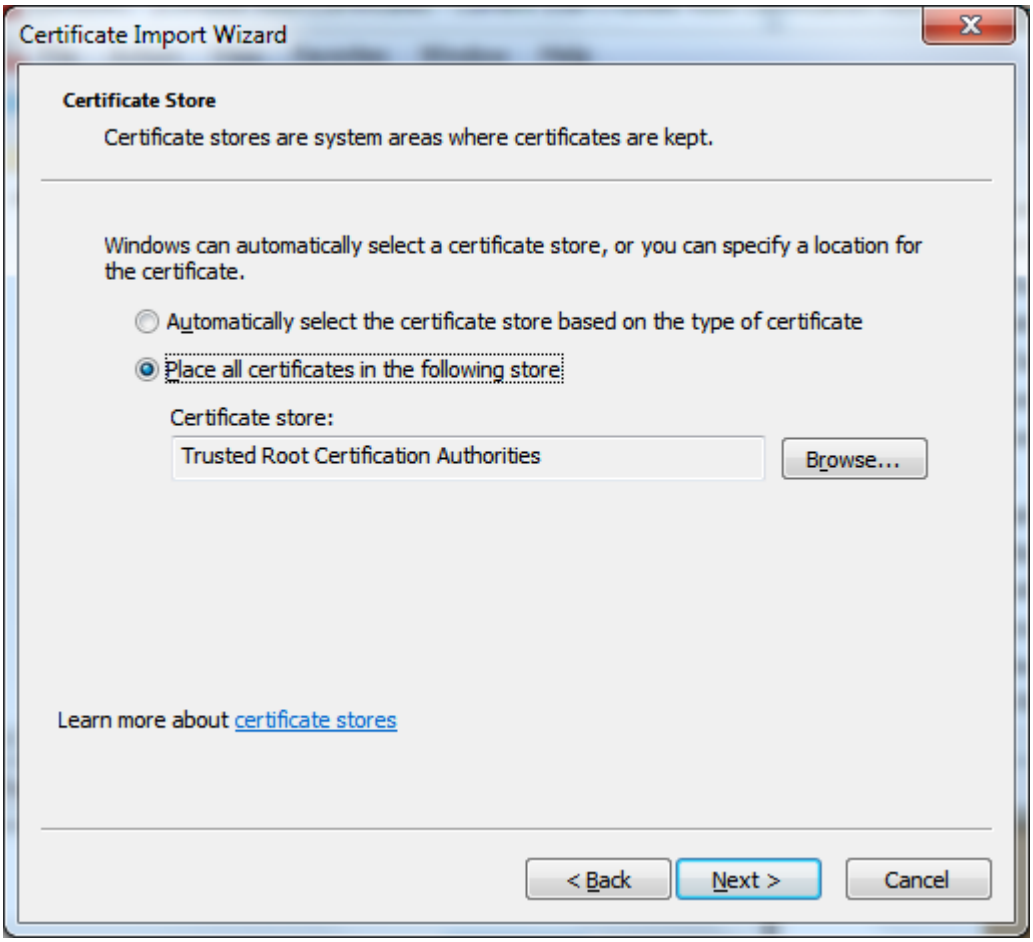


Rechtsklick. Wählen Sie *Alle Aufgaben* und *Importieren aus*



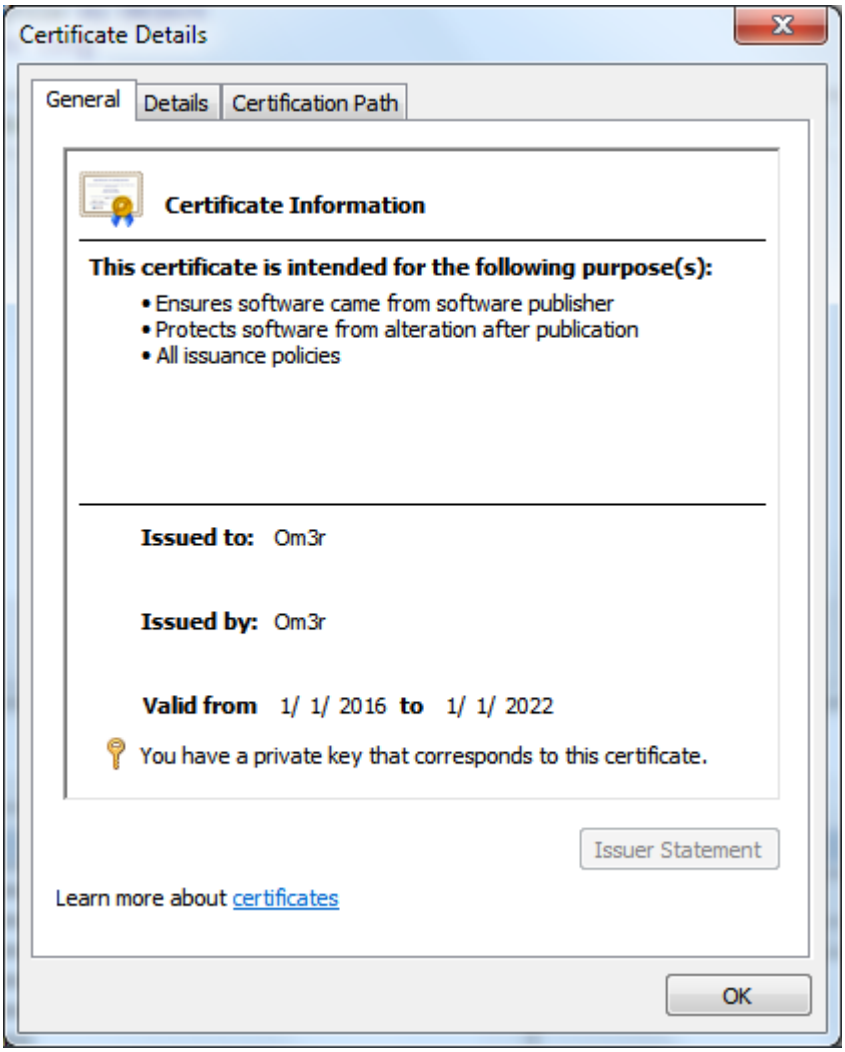


Klicken Sie auf " *Weiter"* und speichern Sie im *Store "Vertrauenswürdige Stammzertifizierungsstellen"* :



Dann Weiter> Fertig stellen, schließen Sie nun die Konsole.

Wenn Sie das Zertifikat jetzt verwenden und seine Eigenschaften überprüfen, werden Sie feststellen, dass es sich um ein vertrauenswürdiges Zertifikat handelt, und Sie können es zum Signieren Ihres Projekts verwenden:



Makrosicherheit und Signatur von VBA-Projekten / -Modulen online lesen: https://riptutorial.com/de/vba/topic/7733/makrosicherheit-und-signatur-von-vba-projekten----modulen

Kapitel 24: Mit ADO arbeiten

Bemerkungen

In den in diesem Thema gezeigten Beispielen wird zur Vereinfachung die frühe Bindung verwendet, und es ist ein Verweis auf die Microsoft ActiveX Data Object xx-Bibliothek erforderlich. Sie können in eine späte Bindung konvertiert werden, indem Sie die stark typisierten Verweise durch `Object` ersetzen und die Objekterstellung ggf. mit `New` durch `CreateObject` .

Examples

Herstellen einer Verbindung zu einer Datenquelle

Der erste Schritt beim Zugriff auf eine Datenquelle über ADO ist das Erstellen eines ADO-`Connection` . In der Regel wird dazu eine Verbindungszeichenfolge verwendet, um die Datenquellenparameter anzugeben. Es ist jedoch auch möglich, eine DSN-Verbindung zu öffnen, indem DSN, Benutzer-ID und Kennwort an die `.Open` Methode übergeben werden.

Beachten Sie, dass ein DSN nicht erforderlich ist, um eine Verbindung zu einer Datenquelle über ADO herzustellen. Jede Datenquelle, die über einen ODBC-Provider verfügt, kann mit der entsprechenden Verbindungszeichenfolge verbunden werden. Während bestimmte Verbindungszeichenfolgen für verschiedene Anbieter außerhalb des Bereichs dieses Themas liegen, ist ConnectionStrings.com eine hervorragende Referenz, um die geeignete Zeichenfolge für Ihren Anbieter zu finden.

```
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
```

Beachten Sie, dass das Datenbankkennwort im obigen Beispiel nur aus Gründen der Übersichtlichkeit in der Verbindungszeichenfolge enthalten ist. Best Practices setzen voraus **,** ***dass*** Datenbankkennwörter ***nicht*** im Code gespeichert werden. Dies kann erreicht werden, indem das Kennwort über die Benutzereingabe oder die Windows-Authentifizierung verwendet wird.

Datensätze mit einer Abfrage abrufen

Abfragen können auf zwei Arten ausgeführt werden. Beide geben ein ADO- `Recordset` Objekt zurück, bei dem es sich um eine Sammlung von zurückgegebenen Zeilen handelt. Beachten Sie, dass die beiden folgenden Beispiele aus Gründen der Kürze die `OpenDatabaseConnection` Funktion aus dem Beispiel Herstellen einer Verbindung mit einer Datenquelle verwenden . Denken Sie daran, dass die an die Datenquelle übergebene Syntax des SQL anbieterspezifisch ist.

Die erste Methode besteht darin, die SQL-Anweisung direkt an das Connection-Objekt zu übergeben. Dies ist die einfachste Methode zum Ausführen einfacher Abfragen:

```
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
```

Die zweite Methode ist das Erstellen eines ADO- `Command` für die Abfrage, die Sie ausführen möchten. Dies erfordert etwas mehr Code, ist aber erforderlich, um parametrisierte Abfragen verwenden zu können:

```
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
```

Beachten Sie, dass an die Datenquelle gesendete Befehle **anfällig für die SQL-Injektion sind** , entweder absichtlich oder unbeabsichtigt. Im Allgemeinen sollten Abfragen nicht durch Verketten von Benutzereingaben jeglicher Art erstellt werden. Sie sollten stattdessen parametrisiert werden (siehe Erstellen parametrisierter Befehle).

Nicht-Skalar-Funktionen ausführen

Mit ADO-Verbindungen können nahezu alle Datenbankfunktionen ausgeführt werden, die der Provider über SQL unterstützt. In diesem Fall ist es nicht immer erforderlich, das von der `Execute` Funktion zurückgegebene `Recordset` zu verwenden, obwohl es nützlich sein kann, Schlüsselzuweisungen nach INSERT-Anweisungen mit @@ Identity- oder ähnlichen SQL-Befehlen abzurufen. Beachten Sie, dass im folgenden Beispiel die `OpenDatabaseConnection` Funktion aus dem Beispiel Herstellen einer Verbindung zu einer Datenquelle verwendet wird .

```
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
```

Beachten Sie, dass an die Datenquelle gesendete Befehle **anfällig für die SQL-Injektion sind** , entweder absichtlich oder unbeabsichtigt. Im Allgemeinen sollten SQL-Anweisungen nicht durch Verketten von Benutzereingaben jeglicher Art erstellt werden. Sie sollten stattdessen parametrisiert werden (siehe Erstellen parametrisierter Befehle).

Parametrisierte Befehle erstellen

Immer wenn SQL, die über eine ADO-Verbindung ausgeführt werden, Benutzereingaben enthalten muss, gilt es als bewährte Methode, diese zu parametrisieren, um die Wahrscheinlichkeit einer SQL-Injection zu minimieren. Diese Methode ist auch lesbarer als lange Verkettungen und ermöglicht robusteren und wartbaren Code (z. B. durch Verwendung einer Funktion, die ein Array von `Parameter` zurückgibt).

In der Standard-ODBC-Syntax werden Parameter angegeben `?` "Platzhalter" im Abfragetext und die Parameter werden dann in derselben Reihenfolge an den `Command` angehängt, in der sie in der Abfrage angezeigt werden.

Beachten Sie, dass das folgende Beispiel die `OpenDatabaseConnection` Funktion aus Herstellen einer Verbindung zu einer Datenquelle verwendet .

```
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
```

Hinweis: Das obige Beispiel veranschaulicht eine parametrisierte UPDATE-Anweisung, es können jedoch beliebige SQL-Anweisungen angegeben werden.

Mit ADO arbeiten online lesen: https://riptutorial.com/de/vba/topic/3578/mit-ado-arbeiten

Kapitel 25: Nicht-lateinische Zeichen

Einführung

VBA kann mit Unicode Zeichenfolgen in jeder Sprache und jedem Skript lesen und schreiben. Es gibt jedoch strengere Regeln für Identifier-Token .

Examples

Nicht-lateinischer Text im VBA-Code

In der Tabellenzelle A1 haben wir das folgende arabische Pangram:

ف خَ لَقَ خَ دوِ كَ مِ ثِ لِ الشَ مسِ إِ ذ بَ زَ غَ تَ - يَ حظى الضَ جيعُ بِ اه نَ جلاءَ مِ عطارِ

VBA bietet die Funktionen `AscW` und `ChrW` , um mit Multi-Byte-Zeichencodes zu arbeiten. Wir können `Byte` Arrays auch verwenden, um die String-Variable direkt zu bearbeiten:

```
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
```

Dies erzeugt die folgende Ausgabe für den arabischen Buchstaben Sad :

> Erstes Zeichen (ChrW): 1589
> Erstes Zeichen (binär): 00011000110101
> Zeichenfolgewert (Text):?
> Stringwert (AscW): 1589
> Byte-Werte (dezimal): 53 | 6
> Byte-Werte (binär): 00110101 | 00000110
>
> Ganze Zeichenfolge
> ??? ????? ????? ??????? ??????? ??? ??????? - ????? ???????? ???? ???????
> ???????

Beachten Sie, dass VBA nicht-lateinischen Text nicht in das unmittelbare Fenster drucken kann, obwohl die Zeichenfolgenfunktionen ordnungsgemäß funktionieren. Dies ist eine Einschränkung der IDE und nicht der Sprache.

Nicht-lateinische Bezeichner und Sprachabdeckung

VBA-Bezeichner (Variablen- und Funktionsnamen) können das lateinische Skript und möglicherweise auch japanische , koreanische , vereinfachtes Chinesisch und traditionelles Chinesisch verwenden .

Das erweiterte lateinische Skript deckt viele Sprachen ab:
Englisch, Französisch, Spanisch, Deutsch, Italienisch, Bretonisch, Katalanisch, Dänisch, Estnisch, Finnisch, Isländisch, Indonesisch, Irisch, Lojban, Mapudungun, Norwegisch, Portugiesisch, Schottisches Gälisch, Schwedisch, Tagalog

Einige Sprachen sind nur teilweise abgedeckt:
Azeri, Kroatisch, Tschechisch, Esperanto, Ungarisch, Lettisch, Litauisch, Polnisch, Rumänisch, Serbisch, Slowakisch, Slowenisch, Türkisch, Yoruba, Walisisch

Einige Sprachen haben wenig oder keine Abdeckung:
Arabisch, Bulgarisch, Cherokee, Dzongkha, Griechisch, Hindi, Mazedonisch, Malayalam, Mongolisch, Russisch, Sanskrit, Thailändisch, Tibetisch, Urdu, Uiguren

Die folgenden Variablendeklarationen sind alle gültig:

```
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
```

Beachten Sie, dass in der VBA-IDE ein einzelner Apostroph innerhalb eines Variablennamens die Zeile nicht in einen Kommentar umwandelt (wie bei Stack Overflow).

Sprachen, die zwei Anführungszeichen verwenden, um ein Zitat zu kennzeichnen, dürfen auch in Variablennamen verwendet werden, ohne dass die Anführungszeichen "" verwendet werden.

Nicht-lateinische Zeichen online lesen: https://riptutorial.com/de/vba/topic/10555/nicht-lateinische-zeichen

Kapitel 26: Objektorientierte VBA

Examples

Abstraktion

Abstraktionsebenen bestimmen, wann die Dinge aufgeteilt werden müssen.

Die Abstraktion wird durch die Implementierung von Funktionen mit immer detaillierterem Code erreicht. Der Einstiegspunkt eines Makros sollte eine kleine Prozedur mit einem *hohen Abstraktionsgrad sein* , die es ermöglicht, auf einen Blick zu erfassen, was los ist:

```
Public Sub DoSomething()
    With New SomeForm
        Set .Model = CreateViewModel
        .Show vbModal
        If .IsCancelled Then Exit Sub
        ProcessUserData .Model
    End With
End Sub
```

Die `DoSomething` Prozedur hat einen hohen *Abstraktionsgrad* : Wir können feststellen, dass sie ein Formular anzeigt und ein Modell erstellt und dieses Objekt an eine `ProcessUserData` Prozedur `ProcessUserData` , die weiß, was mit ihr zu tun ist - wie das Modell erstellt wird, ist Aufgabe einer anderen Prozedur:

```
Private Function CreateViewModel() As ISomeModel
    Dim result As ISomeModel
    Set result = SomeModel.Create(Now, Environ$("UserName"))
    result.AvailableItems = GetAvailableItems
    Set CreateViewModel = result
End Function
```

Die Funktion `CreateViewModel` ist nur für das Erstellen einer `ISomeModel` Instanz verantwortlich. Zu dieser Verantwortung gehört auch der Erwerb einer Reihe *verfügbarer Elemente.* Wie diese Elemente erworben werden, ist ein Implementierungsdetail, das hinter der `GetAvailableItems` Prozedur `GetAvailableItems` wird:

```
Private Function GetAvailableItems() As Variant
    GetAvailableItems = DataSheet.Names("AvailableItems").RefersToRange
End Function
```

Hier liest die Prozedur die verfügbaren Werte aus einem benannten Bereich in einem `DataSheet` Arbeitsblatt. Es könnte genauso gut aus einer Datenbank gelesen werden, oder die Werte könnten hart codiert sein: Dies ist ein *Implementierungsdetail* , das für keine der höheren Abstraktionsstufen von Belang ist.

Verkapselung

Encapsulation verbirgt Implementierungsdetails vor Clientcode.

Das Handling QueryClose- Beispiel veranschaulicht die Kapselung: Das Formular verfügt über ein Kontrollkästchen, der Clientcode funktioniert jedoch nicht direkt damit. Das Kontrollkästchen enthält *Implementierungsdetails.* Der Clientcode muss wissen, ob die Einstellung aktiviert ist oder nicht.

Wenn sich der Kontrollkästchenwert ändert, weist der Handler ein privates Feldmitglied zu:

```
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
```

Wenn der Clientcode diesen Wert lesen möchte, muss er sich nicht um ein Kontrollkästchen kümmern, sondern verwendet stattdessen die `SomeOtherSetting` Eigenschaft:

```
Public Property Get SomeOtherSetting() As Boolean
    SomeOtherSetting = this.SomeOtherSetting
End Property
```

Die `SomeOtherSetting` Eigenschaft *kapselt* den `SomeOtherSetting` des Kontrollkästchens. Client-Code muss nicht wissen, dass ein Kontrollkästchen vorhanden ist, nur dass es eine Einstellung mit einem booleschen Wert gibt. Durch das *Einkapseln* des `Boolean` Werts haben wir dem Kontrollkästchen eine *Abstraktionsebene* hinzugefügt.

Verwendung von Schnittstellen zur Durchsetzung der Unveränderlichkeit

Lassen Sie uns das noch einen Schritt weiter gehen, indem Sie das *Modell* des Formulars in einem dedizierten Klassenmodul *kapseln* . Wenn wir jedoch eine `Public Property` für den `UserName` und den `Timestamp` `UserName` , müssten wir die `Property Let` `UserName` , wodurch die Eigenschaften veränderbar werden. Wir möchten nicht, dass der Clientcode diese Werte nach dem `UserName` ändern kann.

Die `CreateViewModel` Funktion im **Abstraction-** Beispiel gibt eine `ISomeModel` Klasse zurück: `ISomeModel` ist unsere *Schnittstelle* und sieht etwa so aus:

```
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
```

Hinweis `Timestamp` und `UserName` Eigenschaften aussetzen nur ein `Property Get` Accessor. Nun kann die `SomeModel` Klasse diese Schnittstelle implementieren:

```
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
```

Die Schnittstellenmitglieder sind alle `Private` , und alle Mitglieder der Schnittstelle müssen implementiert werden, damit der Code kompiliert werden kann. Die `Public` Member sind nicht Teil der Schnittstelle und unterliegen daher nicht dem für die `ISomeModel` Schnittstelle geschriebenen Code.

Verwenden einer Factory-Methode zum Simulieren eines Konstruktors

Mit einem VB_PredeclaredId- Attribut können wir der `SomeModel` Klasse eine *Standardinstanz* `SomeModel` und eine Funktion schreiben, die wie ein `SomeModel` der `SomeModel` (`Shared` in VB.NET,

`static` in C #) funktioniert, das der Clientcode aufrufen kann, ohne dass er zuerst erstellt werden muss ein Beispiel, wie wir es hier gemacht haben:

```
Private Function CreateViewModel() As ISomeModel
    Dim result As ISomeModel
    Set result = SomeModel.Create(Now, Environ$("UserName"))
    result.AvailableItems = GetAvailableItems
    Set CreateViewModel = result
End Function
```

Diese *Factory-Methode* weist die Eigenschaftswerte zu, die schreibgeschützt sind, wenn von der `ISomeModel` Schnittstelle aus `ISomeModel` wird, hier `Timestamp` und `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
```

Und jetzt können wir gegen die `ISomeModel` Schnittstelle `ISomeModel` , die `Timestamp` und `UserName` als schreibgeschützte Eigenschaften `UserName` , die niemals neu zugewiesen werden können (solange der Code gegen die Schnittstelle geschrieben wird).

Polymorphismus

Polymorphismus ist die Fähigkeit, dieselbe Schnittstelle für verschiedene zugrunde liegende Implementierungen bereitzustellen.

Durch die Möglichkeit, Schnittstellen zu implementieren, kann die Anwendungslogik vollständig von der Benutzeroberfläche oder von der Datenbank oder von diesem oder diesem Arbeitsblatt entkoppelt werden.

`ISomeView` , Sie haben eine `ISomeView` Schnittstelle, die das Formular selbst implementiert:

```
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
```

Der Code-Behind des Formulars könnte folgendermaßen aussehen:

```
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
```

Allerdings verbietet nichts, ein anderes Klassenmodul zu erstellen, das die `ISomeView` Schnittstelle implementiert, *ohne ein Benutzerformular zu sein* - dies könnte eine `SomeViewMock` Klasse sein:

```
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
```

Und jetzt können wir den Code ändern, der mit einer `UserForm` und die `ISomeView` Schnittstelle zum `ISomeView` , z. B. indem Sie die Form als Parameter `ISomeView` , anstatt sie zu instanziieren:

```
Public Sub DoSomething(ByVal view As ISomeView)
    With view
        Set .Model = CreateViewModel
        .Show
        If .IsCancelled Then Exit Sub
        ProcessUserData .Model
    End With
End Sub
```

Da die `DoSomething` Methode von einer Schnittstelle (z. B. einer *Abstraktion*) und nicht von einer *konkreten Klasse* (z. B. einer bestimmten `UserForm`) `UserForm` , können wir einen automatisierten `UserForm` schreiben, der gewährleistet, dass `ProcessUserData` nicht ausgeführt wird, wenn `view.IsCancelled` `True` hat test Erstellen Sie eine `SomeViewMock` Instanz, setzen Sie die `IsCancelled` Eigenschaft auf `True` und übergeben Sie sie an `DoSomething` .

Testbarer Code hängt von Abstraktionen ab

Unit-Tests können in VBA geschrieben werden, es gibt Add-Ins, die es sogar in die IDE integrieren. Wenn Code jedoch *eng* mit einem Arbeitsblatt, einer Datenbank, einem Formular oder dem Dateisystem gekoppelt ist, erfordert der Komponententest ein tatsächliches Arbeitsblatt, eine Datenbank, ein Formular oder ein Dateisystem. Diese *Abhängigkeiten* sind ein neuer Out-of-Control-Fehler Punkte, die der zu testende Code isolieren sollte, sodass für Einzeltests *kein* Arbeitsblatt, keine Datenbank, *kein* Formular oder *kein* Dateisystem erforderlich ist.

Durch das Schreiben von Code gegen Schnittstellen, sodass `SomeViewMock` Stub / Mock-Implementierungen (wie das obige `SomeViewMock` Beispiel) *injizieren* kann, können Sie Tests in einer "kontrollierten Umgebung" schreiben und simulieren, was passiert, wenn jede einzelne der 42 möglich ist Permutationen von Benutzerinteraktionen auf die Daten des Formulars, ohne auch nur ein Formular anzuzeigen und manuell auf ein Formularsteuerelement zu klicken.

Objektorientierte VBA online lesen: https://riptutorial.com/de/vba/topic/5357/objektorientierte-vba

Kapitel 27: Operatoren

Bemerkungen

Operatoren werden in der folgenden Reihenfolge bewertet:

- Mathematische Operatoren
- Bitweise Operatoren
- Verkettungsoperatoren
- Vergleichsoperatoren
- Logische Operatoren

Operatoren mit übereinstimmender Priorität werden von links nach rechts ausgewertet. Die Standardreihenfolge kann durch Verwendung von Klammern `(` und `)` zum Gruppieren von Ausdrücken überschrieben werden.

Examples

Mathematische Operatoren

Nach Priorität sortiert:

Zeichen	Name	Beschreibung
`^`	Potenzierung	Bringen Sie das Ergebnis des Anhebens des linken Operanden auf die Leistung des rechten Operanden zurück. Beachten Sie, dass der von der Exponentiation zurückgegebene Wert *immer* ein `Double` , unabhängig von den Wertetypen, die geteilt werden. Eine Erzwingung eines Ergebnisses in einen Variablentyp findet ***nach*** der Berechnung statt.
`/`	Abteilung [1]	Gibt das Ergebnis der Division des linken Operanden durch den rechten Operanden zurück. Beachten Sie, dass der von Division zurückgegebene Wert *immer* ein `Double` , unabhängig von den Wertetypen, die geteilt werden. Eine Erzwingung eines Ergebnisses in einen Variablentyp findet ***nach*** der Berechnung statt.
`*`	Multiplikation [1]	Gibt das Produkt von 2 Operanden zurück.
`\`	Integer Division	Gibt das ganzzahlige Ergebnis der Division des linken Operanden durch den rechten Operanden zurück, ***nachdem*** beide Seiten mit 0,5 abgerundet wurden. Der Rest der Abteilung wird ignoriert. Wenn der rechte Operand (der Divisor) `0` , führt dies zu einem Laufzeitfehler 11: Division durch Null. Beachten Sie, dass dies der

Zeichen	Name	Beschreibung
		Fall ist, ***nachdem*** alle Rundungen ausgeführt wurden. Ausdrücke wie `3 \ 0.4` führen auch zu einem Fehler durch Division durch Null.
`Mod`	Modulo	Gibt den ganzzahligen Rest der Division des linken Operanden durch den rechten Operanden zurück. Der Operand auf jeder Seite wird *vor* der Division auf eine ganze Zahl gerundet, 0,5 wird abgerundet. Beispielsweise führen sowohl `8.6 Mod 3` als auch `12 Mod 2.6` zu `0` . Wenn der rechte Operand (der Divisor) `0` , führt dies zu einem Laufzeitfehler 11: Division durch Null. Beachten Sie, dass dies der Fall ist, ***nachdem*** alle Rundungen ausgeführt wurden - Ausdrücke wie `3 Mod 0.4` führen auch zu einer Division durch Null Fehler.
`-`	Subtraktion [2]	Gibt das Ergebnis der Subtraktion des rechten Operanden vom linken Operanden zurück.
`+`	Zusatz [2]	Gibt die Summe von 2 Operanden zurück. Beachten Sie, dass dieses Token auch als Verkettungsoperator behandelt wird, wenn es auf einen `String` angewendet wird. Siehe **Verkettungsoperatoren** .

[1] Multiplikation und Division werden als gleichrangig behandelt.

[2] Addition und Subtraktion werden als gleichrangig behandelt.

Verkettungsoperatoren

VBA unterstützt 2 verschiedene Verknüpfungsoperatoren, `+` und `&` und beide führen die gleiche Funktion , wenn sie mit verwendet `String` - Typen - der rechte `String` wird an das Ende des linken beigefügten `String` .

Wenn der Operator `&` mit einem anderen Variablentyp als `String` , wird er implizit in einen `String` bevor er verkettet wird.

Beachten Sie, dass der `+` Verkettungsoperator eine Überlastung des `+` Additionsoperators ist. Das Verhalten von `+` wird durch die Variablentypen der Operanden und die Rangfolge der Operatortypen bestimmt. Wenn beide Operanden als `String` oder `Variant` mit einem Untertyp von `String` eingegeben werden, werden sie verkettet:

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

    left = "5"
    right = "5"

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

Wenn *eine* Seite ein numerischer Typ ist und die andere Seite eine `String` , die in eine Zahl umgewandelt werden kann, führt der Typvorrang von mathematischen Operatoren dazu, dass der Operator als Additionsoperator behandelt wird und die numerischen Werte hinzugefügt werden:

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

    left = 5
    right = "5"

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

Dieses Verhalten kann zu subtilen, schwer zu debuggenden Fehlern führen - insbesondere wenn `Variant` Typen verwendet werden. Daher sollte normalerweise nur der Operator `&` für die Verkettung verwendet werden.

Vergleichsoperatoren

Zeichen	Name	Beschreibung
`=`	Gleich	Gibt " `True` wenn die Operanden links und rechts gleich sind. Beachten Sie, dass dies eine Überlastung des Zuweisungsoperators ist.
`<>`	Nicht gleichzusetzen mit	Gibt " `True` wenn die Operanden links und rechts nicht gleich sind.
`>`	Größer als	Gibt `True` wenn der linke Operand größer ist als der rechte.
`<`	Weniger als	Gibt `True` wenn der linke Operand kleiner als der rechte Operand ist.
`>=`	Größer als oder gleich	Gibt `True` wenn der linke Operand größer oder gleich dem rechten Operanden ist.
`<=`	Weniger als oder gleich	Gibt " `True` wenn der linke Operand kleiner oder gleich dem rechten Operanden ist.
`Is`	Referenz-Eigenkapital	Gibt " `True` wenn die linke Objektreferenz dieselbe Instanz wie die rechte Objektreferenz ist. Sie kann auch mit `Nothing` (der Nullobjektreferenz) auf beiden Seiten verwendet werden. **Hinweis:** Der `Is` Operator versucht, beide Operanden in ein `Object` zu zwingen, bevor der Vergleich ausgeführt wird. Wenn eine Seite ein primitiver Typ *oder* eine `Variant` , die kein Objekt enthält (entweder ein Nicht-Objekt-Subtyp oder `vtEmpty`), führt der Vergleich zu einem Laufzeitfehler 424 - "Objekt erforderlich". Wenn einer der Operanden zu einer anderen *Schnittstelle*

Zeichen	Name	Beschreibung
		desselben Objekts gehört, wird der Vergleich `True` . Wenn Sie sowohl für die Instanz *als auch für* die Schnittstelle auf Gerechtigkeit prüfen müssen, verwenden `ObjPtr(left) = ObjPtr(right)` stattdessen `ObjPtr(left) = ObjPtr(right)` .

Anmerkungen

Die VBA-Syntax erlaubt "Ketten" von Vergleichsoperatoren, diese Konstrukte sollten jedoch generell vermieden werden. Vergleiche werden immer von links nach rechts an jeweils nur 2 Operanden durchgeführt, und jeder Vergleich führt zu einem `Boolean` . Zum Beispiel der Ausdruck ...

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

... kann in einigen Zusammenhängen gelesen werden, um zu prüfen, ob `b` zwischen `a` und `c` . In VBA wird dies wie folgt bewertet:

```
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
```

Jeder Vergleichsoperator andere als `Is` mit einem verwendeten `Object` als Operand wird auf dem Rückgabewert des durchgeführt wird `Object` ,s Standardelement . Wenn das Objekt über kein Standardelement verfügt, führt der Vergleich zu einem Laufzeitfehler. 438 - "Objekt unterstützt nicht seine Eigenschaft oder Methode".

Wenn das `Object` initialisiert ist, führt der Vergleich zu einem Laufzeitfehler 91 - "Objektvariable oder Mit nicht gesetzte Blockvariable".

Wenn das Literal `Nothing` mit einem anderen Vergleichsoperator als " `Is` , führt dies zu einem Kompilierungsfehler - "Ungültige Verwendung des Objekts".

Wenn das Standardmitglied des `Object` ein *anderes* *`Object`* , ruft VBA das Standardmitglied jedes nachfolgenden Rückgabewerts kontinuierlich auf, bis ein primitiver Typ zurückgegeben wird oder ein Fehler ausgegeben wird. `SomeClass` , `SomeClass` hat ein Standardmitglied von `Value` , bei dem es sich um eine Instanz von `ChildClass` mit einem Standardmitglied von `ChildValue` . Der Vergleich...

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

... wird bewertet als:

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

Wenn einer der Operanden ein numerischer Typ ist und der *andere* Operand ein `String` oder eine `Variant` des Untertyps `String` , wird ein numerischer Vergleich durchgeführt. Wenn der `String` nicht in eine Zahl umgewandelt werden kann, führt dies zu einem Laufzeitfehler 13 - "Typenkonflikt".

Wenn ***beide*** Operanden ein `String` oder eine `Variant` des Subtyps `String` , wird ein Stringvergleich basierend auf der Option Compare- Einstellung des Codemoduls durchgeführt. Diese Vergleiche werden zeichenweise durchgeführt. Beachten Sie, dass die *Zeichendarstellung* eines `String` mit einer Zahl als Vergleich der numerischen Werte ***nicht*** gleich ist:

```
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
```

Stellen Sie daher sicher, dass `String` oder `Variant` Variablen in Zahlen umgewandelt werden, bevor Sie Vergleiche mit numerischen Ungleichungen durchführen.

Wenn ein Operand ein `Date` , wird ein numerischer Vergleich des zugrunde liegenden Double- Werts durchgeführt, wenn der andere Operand numerisch ist oder in einen numerischen Typ umgewandelt werden kann.

Wenn der andere Operand ein `String` oder eine `Variant` des Subtyps `String` , der mit dem aktuellen Gebietsschema in ein `Date` werden kann, wird der `String` in ein `Date` . Wenn es im aktuellen Gebietsschema nicht in ein `Date` , führt dies zu einem Laufzeitfehler 13 - "Typenkonflikt".

Beim Vergleich zwischen `Double` oder `Single` und Booleschen Werten ist Vorsicht geboten . Im Gegensatz zu anderen numerischen Typen, Nicht-Null - Werte können nicht angenommen werden `True` aufgrund VBA das Verhalten von den Datentyp eines Vergleichs der Förderung einer Gleitkommazahl zu denen `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
```

Bitweise \ logische Operatoren

Alle logischen Operatoren in VBA können als "Überschreibungen" der bitweisen Operatoren desselben Namens betrachtet werden. Technisch werden sie *immer* als bitweise Operatoren behandelt. Alle Vergleichsoperatoren in VBA geben einen Booleschen Wert zurück, bei dem immer keines der Bits (`False`) oder *alle* Bits (`True`) gesetzt sind. Ein Wert wird jedoch mit *einem* als `True` festgelegten Bit behandelt. Dies bedeutet, dass das Ergebnis der Umwandlung des bitweisen Ergebnisses eines Ausdrucks in einen `Boolean` (siehe Vergleichsoperatoren) immer das gleiche ist, als würde er als logischer Ausdruck behandelt.

Wenn Sie das Ergebnis eines Ausdrucks mit einem dieser Operatoren zuweisen, erhalten Sie das bitweise Ergebnis. Beachten Sie, dass in den folgenden Wahrheitstabellen `0` gleich `False` und `1` gleich `True` .

`And`

Gibt `True` wenn die Ausdrücke auf beiden Seiten als `True` ausgewertet werden.

Linkshändiger Operand	Rechtshänder-Operand	Ergebnis
0	0	0
0	1	0
1	0	0
1	1	1

`Or`

Gibt `True` wenn eine Seite des Ausdrucks als `True` ausgewertet wird.

Linkshändiger Operand	Rechtshänder-Operand	Ergebnis
0	0	0
0	1	1
1	0	1
1	1	1

`Not`

Gibt " `True` wenn der Ausdruck " `False` und " `False` ausgewertet wird, wenn der Ausdruck " `True` .

Rechtshänder-Operand	Ergebnis
0	1
1	0

`Not` ist der einzige Operand ohne einen linken Operanden. Der Visual Basic-Editor vereinfacht Ausdrücke automatisch mit einem Argument für die linke Hand. Wenn du schreibst ...

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

... die VBE ändert die Zeile in:

```
Debug.Print Not x
```

Ähnliche Vereinfachungen gelten für alle Ausdrücke, die einen linken Operanden (einschließlich Ausdrücken) für `Not` .

`Xor`

Auch als "exklusiv oder" bezeichnet. Gibt " `True` wenn beide Ausdrücke unterschiedliche Ergebnisse ergeben.

Linkshändiger Operand	Rechtshänder-Operand	Ergebnis
0	0	0
0	1	1
1	0	1
1	1	0

Beachten Sie, dass , obwohl der `Xor` Bediener kann wie ein logischer Operator *verwendet* wird, gibt es absolut so zu tun , keinen Grund ist , wie es das gleiche Ergebnis wie der Vergleichsoperator gibt `<>` .

`Eqv`

Auch als "Äquivalenz" bezeichnet. Gibt " `True` wenn beide Ausdrücke dasselbe Ergebnis ergeben.

Linkshändiger Operand	Rechtshänder-Operand	Ergebnis
0	0	1
0	1	0
1	0	0

Linkshändiger Operand	Rechtshänder-Operand	Ergebnis
1	1	1

Beachten Sie, dass die `Eqv` Funktion *sehr* selten verwendet wird, da `x Eqv y` dem viel besser lesbaren `Not (x Xor y)` .

`Imp`

Auch als "Implikation" bezeichnet. Gibt `True` wenn beide Operanden gleich sind *oder* der zweite Operand `True` .

Linkshändiger Operand	Rechtshänder-Operand	Ergebnis
0	0	1
0	1	1
1	0	0
1	1	1

Beachten Sie, dass die `Imp` Funktion sehr selten verwendet wird. Eine gute Faustregel lautet: Wenn Sie nicht erklären können, was es bedeutet, sollten Sie ein anderes Konstrukt verwenden.

Operatoren online lesen: https://riptutorial.com/de/vba/topic/5813/operatoren

Kapitel 28: Prozedur erstellen

Examples

Einführung in die Prozeduren

Ein `Sub` ist eine Prozedur, die eine bestimmte Aufgabe ausführt, aber keinen bestimmten Wert zurückgibt.

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

Wenn kein Zugriffsmodifizierer angegeben wird, ist standardmäßig eine Prozedur `Public` .

Eine `Function` ist eine Prozedur, die Daten erhält und einen Wert zurückgibt, im Idealfall ohne globale oder Moduleffekte.

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

Eine `Property` ist eine Prozedur, die Moduldaten *einkapselt* . Eine Eigenschaft kann bis zu 3 Zugriffsmethoden haben: `Get` , um einen Wert oder eine Objektreferenz zurückzugeben, `Let` einen Wert zuweisen und / oder `Set` , um eine Objektreferenz zuzuweisen.

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

Eigenschaften werden normalerweise in Klassenmodulen verwendet (obwohl sie auch in Standardmodulen zulässig sind), wodurch der Zugriff auf Daten verfügbar gemacht wird, auf die der aufrufende Code sonst nicht zugreifen kann. Eine Eigenschaft, die nur einen `Get` Accessor verfügbar macht, ist "schreibgeschützt". Eine Eigenschaft, die nur einen `Let` und / oder `Set` freigeben würde, ist "Nur-Schreiben". Nur Schreiben Eigenschaften sind keine gute Programmierpraxis - wenn der Client - Code einen Wert *schreiben* kann, sollte es in der Lage sein , es zu *lesen* zurück. Erwägen Sie, eine `Sub` Prozedur zu implementieren, anstatt eine schreibgeschützte Eigenschaft zu erstellen.

Rückgabe eines Wertes

Eine `Function` oder `Property Get` Prozedur kann (und sollte!) Einen Wert an ihren Aufrufer zurückgeben. Dies geschieht durch Zuweisung der Kennung der Prozedur:

```
Property Get Foo() As Integer
```

```
    Foo = 42
End Property
```

Funktion mit Beispielen

Wie bereits erwähnt, handelt es sich bei Funktionen um kleinere Prozeduren, die kleine Codeteile enthalten, die sich innerhalb einer Prozedur wiederholen können.

Funktionen werden verwendet, um Redundanz im Code zu reduzieren.

Ähnlich wie bei einer Prozedur kann eine Funktion mit oder ohne Argumentliste deklariert werden.

Funktion wird als Rückgabetyp deklariert, da alle Funktionen einen Wert zurückgeben. Der Name und die Rückgabevariable einer Funktion sind gleich.

1. Funktion mit Parameter:

```
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. Funktion ohne Parameter:

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

Die Funktion kann auf verschiedene Arten innerhalb einer Funktion aufgerufen werden. Da eine mit einem Rückgabetyp deklarierte Funktion grundsätzlich eine Variable ist. es wird ähnlich wie eine Variable verwendet.

Funktionale Anrufe:

```
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
```

Weiterhin kann die Funktion auch als Bedingung für if- und andere Bedingungsanweisungen verwendet werden.

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

Weitere Funktionen können für ihre Argumente Modifikatoren wie By ref und By val haben.

Prozedur erstellen online lesen: https://riptutorial.com/de/vba/topic/1474/prozedur-erstellen

Kapitel 29: Prozeduraufrufe

Syntax

- IdentifierName [*Argumente*]
- Call IdentifierName [*(Argumente)*]
- [Let | Set] *expression* = IdentifierName [*(Argumente)*]
- [Let | Set] IdentifierName [*(Argumente)*] = *Ausdruck*

Parameter

Parameter	Info
IdentifierName	Der Name der aufzurufenden Prozedur.
Argumente	Eine durch Kommas getrennte Liste von Argumenten, die an die Prozedur übergeben werden sollen.

Bemerkungen

Die ersten beiden Syntaxarten dienen zum Aufrufen von `Sub` Prozeduren. Beachten Sie, dass die erste Syntax keine Klammern enthält.

Siehe das ist verwirrend. Warum nicht immer immer Klammern verwenden? Eine ausführliche Erklärung der Unterschiede zwischen den ersten beiden Syntaxen.

Die dritte Syntax dient zum Aufrufen von `Function` und `Property Get` Prozeduren. Wenn es Parameter gibt, sind die Klammern immer obligatorisch. Das `Let` Schlüsselwort ist optional, wenn ein *Wert* zugewiesen *wird* , das `Set` Schlüsselwort ist jedoch **erforderlich,** wenn eine *Referenz* zugewiesen wird.

Die vierte Syntax dient zum Aufrufen von `Property Let` und `Property Set` Prozeduren. Der *`expression`* auf der rechten Seite der Zuweisung wird an den Werteparameter der Eigenschaft übergeben.

Examples

Implizite Aufrufsyntax

```
ProcedureName
ProcedureName argument1, argument2
```

Rufen Sie eine Prozedur mit ihrem Namen ohne Klammern auf.

Randfall

Das Schlüsselwort `Call` wird nur in einem Randfall benötigt:

```
Call DoSomething : DoSomethingElse
```

`DoSomething` und `DoSomethingElse` sind Prozeduren, die aufgerufen werden. Wenn der `Call` Stichwort entfernt wurde, dann `DoSomething` würde als *Zeilenmarke* analysiert werden , anstatt ein Prozeduraufruf, der den Code brechen würde:

```
DoSomething: DoSomethingElse 'only DoSomethingElse will run
```

Rückgabewerte

Um das Ergebnis eines Prozeduraufrufs abzurufen (z. B. `Function` oder `Property Get` Prozeduren), setzen Sie den Aufruf auf die rechte Seite einer Zuweisung:

```
result = ProcedureName
result = ProcedureName(argument1, argument2)
```

Bei Parametern müssen Klammern vorhanden sein. Wenn die Prozedur keine Parameter hat, sind die Klammern überflüssig.

Das ist verwirrend. Warum nicht immer immer Klammern verwenden?

In Klammern werden die Argumente von *Funktionsaufrufen eingeschlossen* . Die Verwendung für *Prozeduraufrufe* kann unerwartete Probleme verursachen.

Weil sie Fehler einführen können, sowohl zur Laufzeit durch Übergeben eines möglicherweise unbeabsichtigten Werts an die Prozedur, als auch zur Kompilierzeit, indem sie einfach eine ungültige Syntax sind.

Laufzeit

Redundante Klammern können Fehler verursachen. Bei einer Prozedur, die eine Objektreferenz als Parameter verwendet ...

```
Sub DoSomething(ByRef target As Range)
End Sub
```

... und mit Klammern aufgerufen:

```
DoSomething (Application.ActiveCell) 'raises an error at runtime
```

Dies führt zu einem Laufzeitfehler "Object Required" # 424. Andere Fehler sind unter anderen Umständen möglich: Hier wird die `Application.ActiveCell Range` Objektreferenz *ausgewertet* und als Wert übergeben, **unabhängig** von der Signatur der Prozedur, die angibt, dass das `target ByRef` . Der tatsächliche Wert, der von `ByVal` an `DoSomething` im obigen Snippet übergeben wurde, lautet `Application.ActiveCell.Value` .

Klammern zwingen VBA, den Wert des Klammerausdrucks auszuwerten, und übergeben das Ergebnis `ByVal` an die aufgerufene Prozedur. Wenn der Typ des ausgewerteten Ergebnisses nicht mit dem erwarteten Typ der Prozedur übereinstimmt und nicht implizit konvertiert werden kann, wird ein Laufzeitfehler ausgelöst.

Kompilierzeit

Dieser Code kann nicht kompiliert werden:

```
MsgBox ("Invalid Code!", vbCritical)
```

Der Ausdruck `("Invalid Code!", vbCritical)` kann nicht als Wert *ausgewertet* werden.

Das würde kompilieren und funktionieren:

```
MsgBox ("Invalid Code!"), (vbCritical)
```

Würde aber auf jeden Fall dumm aussehen. Vermeiden Sie redundante Klammern.

Explizite Aufrufsyntax

```
Call ProcedureName
Call ProcedureName(argument1, argument2)
```

Die explizite Aufrufsyntax erfordert das Schlüsselwort `Call` und Klammern um die Argumentliste. Klammern sind redundant, wenn keine Parameter vorhanden sind. Diese Syntax wurde überholt, als die modernere implizite Aufrufsyntax zu VB hinzugefügt wurde.

Optionale Argumente

Einige Verfahren haben optionale Argumente. Optionale Argumente kommen immer nach erforderlichen Argumenten, aber die Prozedur kann ohne sie aufgerufen werden.

Wenn die Funktion `ProcedureName` über zwei erforderliche Argumente (`argument1` , `argument2`) und ein optionales Argument, `optArgument3` , `optArgument3` , könnte dies auf vier Arten aufgerufen werden:

```
' 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")
```

Die Struktur des Funktions-Headers, der hier aufgerufen wird, würde ungefähr so aussehen:

```
Function ProcedureName(argument1 As String, argument2 As String, Optional optArgument3 As
String) As String
```

Das `Optional` Schlüsselwort gibt an, dass dieses Argument weggelassen werden kann. Wie bereits erwähnt, **müssen** optionale Argumente, die in die Kopfzeile **eingefügt werden** , am Ende nach den erforderlichen Argumenten angezeigt werden.

Sie können auch einen *Standardwert* für das Argument angeben, falls kein Wert an die Funktion übergeben wird:

```
Function ProcedureName(argument1 As String, argument2 As String, Optional optArgument3 As
String = "C") As String
```

Wenn in dieser Funktion das Argument für `c` nicht angegeben wird, ist der Wert standardmäßig `"C"` . Wenn ein Wert angegeben *wird,* wird der Standardwert überschrieben.

Prozeduraufrufe online lesen: https://riptutorial.com/de/vba/topic/1179/prozeduraufrufe

Kapitel 30: Regeln der Namensgebung

Examples

Variablennamen

Variablen enthalten Daten. Benennen Sie sie nach dem Verwendungszweck, **nicht nach ihrem Datentyp** oder Gültigkeitsbereich, und verwenden Sie dabei ein **Nomen** . Wenn Sie sich gezwungen fühlen, Ihre Variablen zu *nummerieren* (z. B. `thing1, thing2, thing3`), sollten Sie stattdessen eine geeignete Datenstruktur (z. B. ein Array, eine `Collection` oder ein `Dictionary`) verwenden.

Namen von Variablen, die einen iterierbaren *Satz* von Werten darstellen - z. B. ein Array, eine `Collection` , ein `Dictionary` oder ein `Range` von Zellen - sollten Plural sein.

Einige gängige VBA-Namenskonventionen gelten daher:

Für Variablen auf Prozedurebene :

`camelCase`

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

    Dim procedureVariable As Long
    Dim someOtherVariable As String

End Sub
```

Für Variablen auf Modulebene:

`PascalCase`

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

Für Konstanten:

`SHOUTY_SNAKE_CASE` wird häufig verwendet, um Konstanten von Variablen zu unterscheiden:

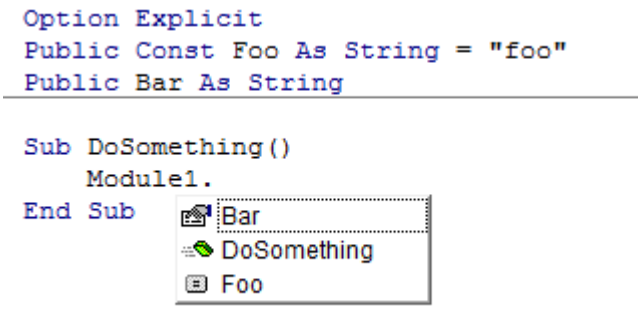
```
Public Const GLOBAL_CONSTANT As String = "Project Version #1.000.000.001"
Private Const MODULE_CONSTANT As String = "Something relevant to this Module"

Public Sub SomeProcedure()

    Const PROCEDURE_CONSTANT As Long = 10

End Sub
```

`PascalCase` Namen machen jedoch saubereren Code und sind genauso gut, da IntelliSense unterschiedliche Symbole für Variablen und Konstanten verwendet:



Ungarische Notation

> Benennen Sie sie nach dem Verwendungszweck, **nicht nach ihrem Datentyp** oder Bereich.

"Ungarische Notation macht es einfacher, den Typ einer Variablen zu erkennen"

Wenn Sie Ihren Code schreiben, z. B. wenn Prozeduren dem *Prinzip* der *einheitlichen Verantwortung entsprechen* (wie es sollte), sollten Sie niemals einen Bildschirm mit Variablendeklarationen oben in einer Prozedur betrachten. Deklarieren Sie Variablen so nahe wie möglich an ihrer ersten Verwendung. Der Datentyp ist immer sichtbar, wenn Sie sie mit einem expliziten Typ deklarieren. Mit der Tastenkombination `Strg` + `i der` VBE können Sie auch den Typ einer Variablen in einer QuickInfo anzeigen.

Wofür eine Variable verwendet wird, sind viel mehr nützliche Informationen als ihr Datentyp, *insbesondere* in einer Sprache wie VBA, die einen Typ nach Bedarf glücklich und implizit in eine andere konvertiert.

Betrachten `iFile` in diesem Beispiel `iFile` und `strFile` :

```
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
```

Vergleichen mit:

```
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` geleitet wird `ByRef` im oberen Beispiel, aber neben der Tatsache , dass wir das Glück zu sehen sind , dass sie *ausdrücklich* als solche übergeben wird , gibt es keinen Hinweis darauf , dass `strData` tatsächlich von der Funktion *zurückgegeben* wird.

Das untere Beispiel nennt es `outContent` . Dieses `out` Präfix ist das, wofür die Ungarische Notation erfunden wurde: um zu klären, *wofür eine Variable verwendet wird* , in diesem Fall, um sie eindeutig als "Out" -Parameter zu identifizieren.

Dies ist nützlich, da IntelliSense selbst keine `ByRef` , selbst wenn der Parameter *explizit* als Referenz übergeben wird:

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

Was dazu führt...

Ungarisch richtig gemacht

Die ungarische Notation hatte ursprünglich nichts mit Variablentypen zu tun . In der Tat ist die Ungarische Notation *richtig gemacht* . Betrachten Sie dieses kleine Beispiel (`ByVal` und `As Integer` wurden aus Gründen der Übersichtlichkeit entfernt):

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

Vergleichen mit:

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

`src` und `dst` sind hier die Präfixe `dst` die *ungarische Notation* und vermitteln *nützliche* Informationen, die ansonsten nicht bereits aus den Parameternamen oder IntelliSense abgeleitet werden können, die den deklarierten Typ `dst` .

Natürlich gibt es eine bessere Möglichkeit, alles zu vermitteln, indem die richtige *Abstraktion* und echte Wörter verwendet werden, die laut und sinnvoll ausgesprochen werden können - als ein erfundenes Beispiel:

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

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

Prozedurnamen

Prozeduren *machen etwas* . Benennen Sie sie nach dem, was sie tun, und verwenden Sie ein **Verb** . Wenn eine genaue Benennung einer Prozedur nicht möglich ist, führt die Prozedur wahrscheinlich *zu viele Dinge aus* und muss in kleinere, speziellere Prozeduren unterteilt werden.

Einige gängige VBA-Namenskonventionen gelten daher:

Für alle Verfahren:

`PascalCase`

```
Public Sub DoThing()

End Sub

Private Function ReturnSomeValue() As [DataType]

End Function
```

Für Event-Handler-Prozeduren:

`ObjectName_EventName`

```
Public Sub Workbook_Open()

End Sub

Public Sub Button1_Click()

End Sub
```

Event-Handler werden normalerweise automatisch von der VBE benannt. Wenn Sie sie

umbenennen, ohne das Objekt und / oder das behandelte Ereignis umzubenennen, wird der Code beschädigt. Der Code wird ausgeführt und kompiliert. Die Prozedur wird jedoch verwaist und wird niemals ausgeführt.

Boolesche Mitglieder

Betrachten Sie eine Boolesche Rückkehrfunktion:

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

Vergleichen mit:

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

Der `Can` - Präfix den gleichen Zweck wie die *nicht* dienen `b` Präfix: Sie identifiziert die Rückgabewert der Funktion als `Boolean` . `Can` besser lesen als `b` :

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

Verglichen mit:

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

Erwägen Sie die Verwendung von Präfixen wie `Can` , `Is` oder `Has` vor Boolean-wiederkehrenden Elementen (Funktionen und Eigenschaften), jedoch nur, wenn dies einen Mehrwert ergibt. Dies entspricht den aktuellen Benennungsrichtlinien von Microsoft .

Regeln der Namensgebung online lesen: https://riptutorial.com/de/vba/topic/1184/regeln-der-namensgebung

Kapitel 31: Rekursion

Einführung

Eine Funktion, die sich selbst aufruft, wird als *rekursiv bezeichnet* . Rekursive Logik kann oft auch als Schleife implementiert werden. Die Rekursion muss mit einem Parameter gesteuert werden, damit die Funktion weiß, wann die Rekursion und Vertiefung des Aufrufstapels unterbrochen werden muss. *Unbegrenzte Rekursion führt* schließlich zu einem Laufzeitfehler "28": "Nicht genügend Stapelspeicher".

Siehe Rekursion .

Bemerkungen

Rekursion ermöglicht wiederholte, selbstreferenzierende Aufrufe einer Prozedur.

Examples

Faktoren

```
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
```

Folder Rekursion

Early Bound (mit einem Verweis auf `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
```

Rekursion online lesen: https://riptutorial.com/de/vba/topic/3236/rekursion

Kapitel 32: Sammlungen

Bemerkungen

Eine `Collection` ist ein Containerobjekt, das in der VBA-Laufzeitumgebung enthalten ist. Zur Verwendung sind keine zusätzlichen Referenzen erforderlich. Eine `Collection` kann zum Speichern von Elementen eines beliebigen Datentyps verwendet werden und ermöglicht das Abrufen entweder über den Ordnungsindex des Elements oder über einen optionalen eindeutigen Schlüssel.

Funktionsvergleich mit Arrays und Wörterbüchern

	Sammlung	Array	Wörterbuch
Kann in der Größe verändert werden	Ja	Manchmal [1]	Ja
Artikel sind bestellt	Ja	Ja	Ja [2]
Elemente sind stark typisiert	Nein	Ja	Nein
Elemente können über die Ordinalzahl abgerufen werden	Ja	Ja	Nein
Neue Elemente können an der Ordinalzahl eingefügt werden	Ja	Nein	Nein
So können Sie feststellen, ob ein Artikel vorhanden ist	Alle Elemente iterieren	Alle Elemente iterieren	Alle Elemente iterieren
Elemente können per Schlüssel abgerufen werden	Ja	Nein	Ja
Bei den Tasten wird zwischen Groß- und Kleinschreibung unterschieden	Nein	N / A	Optional [3]
So stellen Sie fest, ob ein Schlüssel vorhanden ist	Fehlerhandler	N / A	`.Exists` Funktion
Entfernen Sie alle Elemente	Iterieren und `.Remove`	`Erase` , `ReDim`	`.RemoveAll` Funktion

[1] Nur dynamische Arrays können in der Größe geändert werden, und nur die letzte Dimension von mehrdimensionalen Arrays.

[2] Die zugrunde liegenden `.Keys` und `.Items` werden sortiert.

[3] Bestimmt durch die `.CompareMode` Eigenschaft.

Examples

Elemente zu einer Sammlung hinzufügen

Die Artikel werden zu einer hinzugefügt `Collection` durch seinen Aufruf `.Add` Methode:

Syntax:

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

Parameter	Beschreibung
Artikel	Der Artikel, der in der `Collection` gespeichert werden soll. Dies kann im Wesentlichen jeder Wert sein, dem eine Variable zugewiesen werden kann, einschließlich primitiver Typen, Arrays, Objekte und `Nothing` .
Schlüssel	Wahlweise. Ein `String` , der als eindeutiger Bezeichner für das Abrufen von Elementen aus der `Collection` . Wenn der angegebene Schlüssel bereits in der `Collection` , führt dies zu einem Laufzeitfehler 457: "Dieser Schlüssel ist bereits einem Element dieser Sammlung zugeordnet."
Vor	Wahlweise. Ein vorhandener Schlüssel (`String` Wert) *oder ein* Index (numerischer Wert), um das Element zuvor in die `Collection` einzufügen. Wenn ein Wert angegeben wird, ***muss*** der *nach* Parameter leer sein oder ein Laufzeitfehler 5: „ungültiger Prozedur - Aufruf oder Argument“ wird zur Folge haben . Wenn ein `String` Schlüssel übergeben wird, der in der `Collection` nicht vorhanden ist, führt dies zu einem Laufzeitfehler 5: "Ungültiger Prozeduraufruf oder -argument". Wenn ein numerischer Index übergeben wird, der in der `Collection` nicht vorhanden ist, führt dies zu einem Laufzeitfehler 9: "Index außerhalb des gültigen Bereichs".
nach dem	Wahlweise. Ein vorhandener Schlüssel (`String` Wert) *oder ein* Index (numerischer Wert), nach dem das Element in die `Collection` . Wenn ein Wert angegeben wird, ***muss*** der *vor* Parameter leer sein. Fehler sind identisch mit dem Parameter *before* .

Anmerkungen:

- Bei den Schlüsseln wird ***nicht zwischen*** Groß- und Kleinschreibung unterschieden. `.Add "Bar", "Foo"` und `.Add "Baz", "foo"` führt zu einer Schlüsselkollision.
- Wenn keine der optionalen *Vorher*- oder *Nachher*- Parameter angegeben ist, wird der Artikel nach dem letzten Artikel in der `Collection` hinzugefügt.
- Einfügungen, die durch Angabe eines *Vorher*- oder *Nachher*- Parameters vorgenommen

werden, ändern die numerischen Indizes der vorhandenen Elemente an ihre neue Position. Dies bedeutet, dass beim Einfügen in Schleifen mit numerischen Indizes Vorsicht geboten ist.

Beispielverwendung:

```
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
```

Elemente aus einer Sammlung entfernen

Elemente werden aus einer `Collection` indem sie ihre `.Remove` Methode aufrufen:

Syntax:

```
.Remove(index)
```

Parameter	Beschreibung
Index	Das Element, das aus der `Collection` . Wenn der übergebene Wert ein numerischer Typ oder eine `Variant` mit einem numerischen Untertyp ist, wird er als numerischer Index interpretiert. Wenn der übergebene Wert ein `String` oder eine `Variant` die einen String enthält, wird dieser als Schlüssel interpretiert. Wenn ein String-Schlüssel übergeben wird, der in der `Collection` nicht vorhanden ist, führt dies zu einem Laufzeitfehler 5: "Ungültiger Prozeduraufruf oder -argument". Wenn ein numerischer Index übergeben wird, der in der `Collection` nicht vorhanden ist, führt dies zu einem Laufzeitfehler 9: "Index außerhalb des gültigen Bereichs".

Anmerkungen:

- Durch das Entfernen eines Elements aus einer `Collection` werden die numerischen Indizes aller Elemente in der `Collection` geändert. `For` Schleifen , die numerische Indizes und Entfernen von Elementen verwenden sollte , *rückwärts* läuft (`Step -1`) Index Ausnahmen zu verhindern und übersprungenen Elemente.
- Elemente sollten im Allgemeinen ***nicht*** aus einer `Collection` aus einer `For Each` Schleife

entfernt werden, da dies zu unvorhersehbaren Ergebnissen führen kann.

Beispielverwendung:

```
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
```

Abrufen der Elementanzahl einer Sammlung

Die Anzahl der Elemente in einer `Collection` kann durch Aufruf der `.Count` Funktion abgerufen werden:

Syntax:

```
.Count()
```

Beispielverwendung:

```
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
```

Elemente aus einer Sammlung abrufen

Elemente können aus einer `Collection` abgerufen werden, indem die Funktion `.Item` wird.

Syntax:

```
.Item(index)
```

Parameter	Beschreibung
Index	Das Element, das aus der `Collection` abgerufen werden soll. Wenn der übergebene Wert ein numerischer Typ oder eine `Variant` mit einem numerischen Untertyp ist, wird er als numerischer Index interpretiert. Wenn der übergebene Wert ein `String` oder eine `Variant` die einen String enthält, wird dieser als Schlüssel interpretiert. Wenn ein String-Schlüssel übergeben wird, der in der `Collection` nicht vorhanden ist, führt dies zu einem Laufzeitfehler 5: "Ungültiger Prozeduraufruf oder -argument". Wenn ein numerischer Index übergeben wird, der in der `Collection` nicht vorhanden ist, führt dies zu einem Laufzeitfehler 9: "Index außerhalb des gültigen Bereichs".

Anmerkungen:

- `.Item` ist das Standardmitglied von `Collection` . Dies ermöglicht Flexibilität in der Syntax, wie im folgenden Beispiel gezeigt.
- Numerische Indizes basieren auf 1.
- Bei den Schlüsseln wird ***nicht zwischen*** Groß- und Kleinschreibung unterschieden. `.Item("Foo")` und `.Item("foo")` beziehen sich auf denselben Schlüssel.
- Der *Index*- Parameter wird ***nicht*** implizit in eine Zahl aus einem `String` oder umgekehrt umgewandelt. Es ist durchaus möglich, dass `.Item(1)` und `.Item("1")` auf verschiedene Elemente der `Collection` verweisen.

Beispielverwendung (Indizes):

```
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
```

Verwendungsbeispiel (Schlüssel):

```
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
```

Verwendungsbeispiel (alternative Syntax):

```
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
```

Beachten Sie, dass die Syntax bang (`!`) `.Item` ist, da `.Item` das Standardmitglied ist und ein einzelnes `String` Argument annehmen kann. Der Nutzen dieser Syntax ist fragwürdig.

Bestimmen, ob ein Schlüssel oder ein Element in einer Sammlung vorhanden ist

Schlüssel

Im Gegensatz zu einem Scripting.Dictionary verfügt eine `Collection` nicht über eine Methode zum Bestimmen, ob ein bestimmter Schlüssel vorhanden ist, *oder* eine Methode zum Abrufen von in der `Collection` vorhandenen Schlüsseln. Die einzige Methode, um festzustellen, ob ein Schlüssel vorhanden ist, ist die Verwendung des Fehlerhandlers:

```
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
```

Artikel

Die einzige Möglichkeit festzustellen, ob ein Element in einer `Collection` ist, besteht darin, die `Collection` zu durchlaufen, bis das Element gefunden wurde. Beachten Sie, dass eine `Collection` einige Grundelemente oder Objekte enthalten kann. Daher ist eine zusätzliche Behandlung erforderlich, um Laufzeitfehler während der Vergleiche zu vermeiden:

```
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
```

Alle Elemente aus einer Sammlung löschen

Die einfachste Möglichkeit, alle Elemente aus einer `Collection` zu löschen, besteht darin, sie einfach durch eine neue `Collection` zu ersetzen und die alten aus dem Geltungsbereich zu lassen:

```
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
```

Wenn jedoch mehrere Verweise auf die `Collection` , gibt Ihnen diese Methode nur eine leere `Collection` *für die zugewiesene Variable* .

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

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

    Set bar = foo
    Set foo = New Collection

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

In diesem Fall können Sie den Inhalt am einfachsten löschen, indem Sie die Anzahl der Elemente in der `Collection` durchlaufen und das unterste Element wiederholt entfernen:

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

Sammlungen online lesen: https://riptutorial.com/de/vba/topic/5838/sammlungen

Kapitel 33: Schnittstellen

Einführung

Eine **Schnittstelle** ist eine Möglichkeit, eine Gruppe von Verhalten zu definieren, die eine Klasse ausführt. Die Definition einer Schnittstelle ist eine Liste von Methodensignaturen (Name, Parameter und Rückgabetyp). Eine Klasse mit allen Methoden soll diese Schnittstelle "implementieren".

In VBA kann der Compiler mithilfe von Interfaces überprüfen, ob ein Modul alle seine Methoden implementiert. Eine Variable oder ein Parameter kann in Form einer Schnittstelle anstelle einer bestimmten Klasse definiert werden.

Examples

Einfache Schnittstelle - flugfähig

Die Schnittstelle `Flyable` ist ein Klassenmodul mit dem folgenden Code:

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

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

Ein Klassenmodul, `Airplane` , verwendet das `Implements` Schlüsselwort, um dem Compiler mitzuteilen, dass ein Fehler `Flyable_Fly()` es sei denn, es gibt zwei Methoden: ein `Flyable_Fly()` Sub und eine `Flyable_GetAltitude()` Funktion, die ein `Long` zurückgibt.

```
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
```

Ein zweites Klassenmodul, `Duck` , implementiert auch `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
```

Wir können eine Routine schreiben, die einen beliebigen `Flyable` Wert akzeptiert und weiß, dass er auf einen Befehl von `Fly` oder `GetAltitude` :

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

Da die Schnittstelle definiert ist, werden im IntelliSense-Popup-Fenster `Fly` und `GetAltitude` für `F GetAltitude` .

Wenn wir den folgenden Code ausführen:

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

FlyAndCheckAltitude MyDuck
FlyAndCheckAltitude MyAirplane
```

Die Ausgabe ist:

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

Obwohl das Unterprogramm in `Airplane` und `Duck` als `Flyable_Fly` , kann es als `Fly` wenn die Variable oder der Parameter als `Flyable` definiert `Flyable` . Wenn die Variable speziell als `Duck` , müsste sie als `Flyable_Fly` .

Mehrere Schnittstellen in einer Klasse - flugfähig und schwimmfähig

`Flyable` vom `Flyable` Beispiel können Sie eine zweite Schnittstelle, `Swimmable` , mit folgendem Code hinzufügen:

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

Die `Duck` Objekt kann `Implement` beide Fliegen und Schwimmen:

```
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
```

Ein `Fish` - Klasse kann implementieren `Swimmable` auch:

```
Implements Swimmable

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

Jetzt können wir sehen , dass die `Duck` Objekt kann auf einen Teil als übergeben werden `Flyable` einerseits und einer `Swimmable` auf der anderen Seite :

```
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
```

Die Ausgabe dieses Codes ist:

> Fly Check ...
>
> Mit Flügeln fliegen!
>
> 30
>
> Mit Jetmotoren fliegen!

10000

Schwimmcheck ...

Auf dem Wasser treiben

Schwimmen unter Wasser

Schnittstellen online lesen: https://riptutorial.com/de/vba/topic/8784/schnittstellen

Kapitel 34: Scripting.Dictionary-Objekt

Bemerkungen

Sie müssen dem VBA-Projekt Microsoft Scripting Runtime über den Befehl Tools → References der VBE hinzufügen, um das frühe Binden des Scripting Dictionary-Objekts zu implementieren. Diese Bibliotheksreferenz wird mit dem Projekt mitgeführt. Es muss nicht erneut referenziert werden, wenn das VBA-Projekt auf einem anderen Computer verteilt und ausgeführt wird.

Examples

Eigenschaften und Methoden

Ein Scripting Dictionary-Objekt speichert Informationen in Schlüssel- / Elementpaaren. Die Schlüssel müssen eindeutig sein und dürfen kein Array sein, aber die zugehörigen Elemente können wiederholt werden (ihre Eindeutigkeit wird vom zugehörigen Schlüssel gehalten) und können von jeder Art von Variante oder Objekt sein.

Ein Wörterbuch kann als eine In-Memory-Datenbank mit zwei Feldern und einem primären eindeutigen Index für das erste 'Feld' (den *Schlüssel*) betrachtet werden. Dieser eindeutige Index der Keys-Eigenschaft ermöglicht sehr schnelle "Lookups", um den mit einem Schlüssel verknüpften Elementwert abzurufen.

Eigenschaften

Name	lesen Schreiben	Art	Beschreibung
CompareMode	*lesen Schreiben*	CompareMode-Konstante	Das Setzen des CompareMode kann nur für ein leeres Wörterbuch durchgeführt werden. Zulässige Werte sind 0 (vbBinaryCompare), 1 (vbTextCompare), 2 (vbDatabaseCompare).
Anzahl	*schreibgeschützt*	vorzeichenlose lange ganze Zahl	Eine einseitige Anzahl der Schlüssel- / Elementpaare im Skriptwörterbuchobjekt.
Schlüssel	*lesen Schreiben*	Nicht-Array-Variante	Jeder einzelne eindeutige Schlüssel im Wörterbuch.
Artikel (*Schlüssel*)	*lesen Schreiben*	jede Variante	Standardeigenschaft Jedes einzelne Element ist einem Schlüssel im Wörterbuch zugeordnet. Wenn Sie versuchen, ein Element mit einem

Name	lesen Schreiben	Art	Beschreibung
			Schlüssel abzurufen, der nicht im Wörterbuch vorhanden ist, wird der übergebene Schlüssel *implizit hinzugefügt* .

Methoden

Name	Beschreibung
Hinzufügen (*Schlüssel* , *Element*)	Fügt dem Wörterbuch einen neuen Schlüssel und ein neues Element hinzu. Der neue Schlüssel darf nicht in der aktuellen Schlüsselauflistung des Wörterbuchs vorhanden sein. Ein Element kann jedoch unter vielen eindeutigen Schlüsseln wiederholt werden.
Existiert (*Schlüssel*)	Boolescher Test, um festzustellen, ob ein Schlüssel bereits im Wörterbuch vorhanden ist.
Schlüssel	Gibt das Array oder die Sammlung eindeutiger Schlüssel zurück.
Artikel	Gibt das Array oder die Sammlung zugehöriger Elemente zurück.
Entfernen (*Schlüssel*)	Entfernt einen einzelnen Wörterbuchschlüssel und das zugehörige Element.
Alles entfernen	Löscht alle Schlüssel und Elemente eines Wörterbuchobjekts.

Beispielcode

```
'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
```

Daten mit Scripting.Dictionary aggregieren (Maximum, Anzahl)

Wörterbücher eignen sich hervorragend zum Verwalten von Informationen, bei denen mehrere Einträge vorkommen, aber es geht nur um einen einzelnen Wert für jeden Eintragssatz - den ersten oder letzten Wert, den Mindest- oder Höchstwert, einen Durchschnitt, eine Summe usw.

Stellen Sie sich eine Arbeitsmappe vor, die ein Protokoll der Benutzeraktivität enthält, mit einem Skript, das den Benutzernamen und das Bearbeitungsdatum jedes Mal einfügt, wenn jemand die Arbeitsmappe bearbeitet:

`Log` - Arbeitsblatt

EIN	B
Bob	10/12/2016 9:00 Uhr
alice	13.10.2016 13:00 Uhr
Bob	13.10.2016 13:30 Uhr
alice	13.10.2016 14:00 Uhr
alice	14.10.2016 13:00 Uhr

Angenommen, Sie möchten die letzte Bearbeitungszeit für jeden Benutzer in ein Arbeitsblatt mit dem Namen `Summary` ausgeben.

Anmerkungen:

1. Es wird angenommen, dass sich die Daten in `ActiveWorkbook` .
2. Wir verwenden ein Array, um die Werte aus dem Arbeitsblatt zu ziehen. Dies ist effizienter als das Durchlaufen jeder Zelle.
3. Das `Dictionary` wird mithilfe einer frühen Bindung erstellt.

```
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
```

und die Ausgabe sieht so aus:

`Summary` Arbeitsblatt

EIN	B
Bob	13.10.2016 13:30 Uhr
alice	14.10.2016 13:00 Uhr

Wenn Sie dagegen ausgeben möchten, wie oft jeder Benutzer die Arbeitsmappe bearbeitet hat, sollte der Rumpf der `For` Schleife folgendermaßen aussehen:

```
        '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
```

und die Ausgabe sieht so aus:

`Summary` Arbeitsblatt

EIN	B
Bob	2
alice	3

Mit Scripting.Dictionary eindeutige Werte erhalten

Das `Dictionary` ermöglicht es sehr einfach, eine eindeutige Menge von Werten zu erhalten. Betrachten Sie die folgende Funktion:

```
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
```

was könnte man dann so nennen:

```
Dim duplicates() As Variant
duplicates = Array(1, 2, 3, 1, 2, 3)
Dim uniqueVals() As Variant
uniqueVals = Unique(duplicates)
```

und `uniqueVals` würde nur `{1,2,3}` .

Hinweis: Diese Funktion kann mit jedem aufzählbaren Objekt verwendet werden.

Scripting.Dictionary-Objekt online lesen: https://riptutorial.com/de/vba/topic/3667/scripting-dictionary-objekt

Kapitel 35: Scripting.FileSystemObject

Examples

Ein FileSystemObject erstellen

```
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
```

Lesen einer Textdatei mit einem 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
```

Erstellen einer Textdatei mit 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
```

Mit FileSystemObject in eine vorhandene Datei schreiben

```
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
```

Auflisten von Dateien in einem Verzeichnis mithilfe von FileSystemObject

Früh gebunden (erfordert einen Verweis auf 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
```

Spät gebunden:

```
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
```

Ordnen Sie Ordner und Dateien rekursiv auf

Early Bound (mit einem Verweis auf `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
```

Ausgabe beim Aufruf mit Argumenten wie: `EnumerateFilesAndFolders "C:\Test"`

```
C:\Test
  Documents
    Personal
      Budget.xls
      Recipes.doc
    Work
      Planning.doc
  Downloads
    FooBar.exe
  ReadMe.txt
```

Ausgabe bei Aufruf mit Argumenten wie: `EnumerateFilesAndFolders "C:\Test", 0`

```
C:\Test
  Documents
  Downloads
  ReadMe.txt
```

Ausgabe bei Aufruf mit Argumenten wie: `EnumerateFilesAndFolders "C:\Test", 1, 4`

```
C:\Test
    Documents
        Personal
        Work
    Downloads
        FooBar.exe
    ReadMe.txt
```

Dateierweiterung von einem Dateinamen entfernen

```
Dim fso As New Scripting.FileSystemObject
Debug.Print fso.GetBaseName("MyFile.something.txt")
```

```
MyFile.something
```

Beachten Sie, dass die `GetBaseName()` -Methode bereits mehrere `GetBaseName()` in einem Dateinamen verarbeitet.

Rufen Sie nur die Erweiterung von einem Dateinamen ab

```
Dim fso As New Scripting.FileSystemObject
Debug.Print fso.GetExtensionName("MyFile.something.txt")
```

`txt` Beachten Sie, dass die `GetExtensionName()` -Methode bereits mehrere `GetExtensionName()` in einem Dateinamen verarbeitet.

Rufen Sie nur den Pfad aus einem Dateipfad ab

Die GetParentFolderName-Methode gibt den übergeordneten Ordner für einen beliebigen Pfad zurück. Dies kann zwar auch für Ordner verwendet werden, ist aber für das Extrahieren des Pfads aus einem absoluten Dateipfad sinnvoller:

```
Dim fso As New Scripting.FileSystemObject
Debug.Print fso.GetParentFolderName("C:\Users\Me\My Documents\SomeFile.txt")
```

Druckt `C:\Users\Me\My Documents`

Beachten Sie, dass das nachgestellte Pfadtrennzeichen nicht in der zurückgegebenen Zeichenfolge enthalten ist.

Verwenden von FSO.BuildPath zum Erstellen eines vollständigen Pfads aus Ordnerpfad und Dateiname

Wenn Sie Benutzereingaben für Ordnerpfade akzeptieren, müssen Sie möglicherweise vor dem Erstellen eines Dateipfads nach nachfolgenden umgekehrten Schrägstrichen (`\`) suchen. Die `FSO.BuildPath` Methode macht dies einfacher:

```
  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)
```

Ausgabe:

```
C:\Temp\Results.txt
C:\Temp\Results.txt
```

Scripting.FileSystemObject online lesen: https://riptutorial.com/de/vba/topic/990/scripting-filesystemobject

Kapitel 36: Sortierung

Einführung

Im Gegensatz zum .NET-Framework enthält die Visual Basic für Applikationen-Bibliothek keine Routinen zum Sortieren von Arrays.

Es gibt zwei Arten von Problemumgehungen: 1) Einsetzen eines Sortieralgorithmus von Grund auf oder 2) Verwenden von Sortierroutinen in anderen allgemein verfügbaren Bibliotheken.

Examples

Algorithmusimplementierung - Schnelle Sortierung in einem eindimensionalen Array

Von der VBA-Array-Sortierfunktion?

```
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
```

Verwenden der Excel-Bibliothek zum Sortieren eines eindimensionalen Arrays

Dieser Code nutzt die `Sort` Klasse in der Microsoft Excel-Objektbibliothek.

Weitere Informationen finden Sie unter:

- Kopieren Sie einen Bereich in einen virtuellen Bereich
- Wie kopiere ich den ausgewählten Bereich in ein bestimmtes Array?

```
Sub testExcelSort()

Dim arr As Variant

InitArray arr
ExcelSort arr

End Sub

Private Sub InitArray(arr As Variant)

Const size = 10
ReDim arr(size)

Dim i As Integer

' Add descending numbers to the array to start
For i = 0 To size
    arr(i) = size - i
Next i

End Sub

Private Sub ExcelSort(arr As Variant)

' Ininitialize the Excel objects (required)
Dim xl As New Excel.Application
Dim wbk As Workbook
Set wbk = xl.Workbooks.Add
Dim sht As Worksheet
Set sht = wbk.ActiveSheet

' Copy the array to the Range object
Dim rng As Range
Set rng = sht.Range("A1")
Set rng = rng.Resize(UBound(arr, 1), 1)
rng.Value = xl.WorksheetFunction.Transpose(arr)

' Run the worksheet's sort routine on the Range
Dim MySort As Sort
Set MySort = sht.Sort

With MySort
    .SortFields.Clear
    .SortFields.Add rng, xlSortOnValues, xlAscending, xlSortNormal
    .SetRange rng
    .Header = xlNo
    .Apply
End With
```

```
' Copy the results back to the array
CopyRangeToArray rng, arr

' Clear the objects
Set rng = Nothing
wbk.Close False
xl.Quit

End Sub

Private Sub CopyRangeToArray(rng As Range, arr)

Dim i As Long
Dim c As Range

' Can't just set the array to Range.value (adds a dimension)
For Each c In rng.Cells
    arr(i) = c.Value
    i = i + 1
Next c

End Sub
```

Sortierung online lesen: https://riptutorial.com/de/vba/topic/8836/sortierung

Kapitel 37: String Literals - Escape-Zeichen, nicht druckbare Zeichen und Zeilenfortsetzungen

Bemerkungen

Die Zuweisung von String-Literalen in VBA ist durch die Einschränkungen der IDE und die Codepage der Spracheinstellungen des aktuellen Benutzers eingeschränkt. Die obigen Beispiele zeigen die Sonderfälle von Escape-Strings, speziellen, nicht druckbaren Strings und langen String-Literalen.

Wenn Sie Zeichenfolgenliterale zuweisen, die bestimmte Zeichen für eine bestimmte Codepage enthalten, müssen Sie möglicherweise Bedenken hinsichtlich der Internationalisierung berücksichtigen, indem Sie eine Zeichenfolge aus einer separaten Unicode-Ressourcendatei zuweisen.

Examples

Dem Charakter "entkommen"

VBA - Syntax erfordert , dass ein String-Literal erscheinen innerhalb von " Marken, so dass , wenn die Zeichenfolge muss in Anführungszeichen *enthalten,* werden Sie entkommen müssen / prepend die " Zeichen mit einem zusätzlichen " so dass VBA versteht , dass Sie die Absicht , "" zu sein als " String " interpretiert.

```
'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"
```

Lange String-Literale zuweisen

Der VBA-Editor lässt nur 1023 Zeichen pro Zeile zu, normalerweise sind jedoch nur die ersten 100-150 Zeichen ohne Bildlauf sichtbar. Wenn Sie lange Zeichenfolgenliterale zuweisen müssen, aber den Code lesbar halten möchten, müssen Sie für die Zuweisung der Zeichenfolge Zeilenfortsetzungen und Verkettungen verwenden.

```
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.
```

Mit VBA können Sie eine begrenzte Anzahl von Zeilenfortsätzen verwenden (die tatsächliche Anzahl hängt von der Länge jeder Zeile im Continuous-Block ab). Wenn Sie also sehr lange Zeichenfolgen haben, müssen Sie die Verkettung zuweisen und erneut zuweisen .

```
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.
```

Verwenden von VBA-Stringkonstanten

VBA definiert eine Reihe von String-Konstanten für Sonderzeichen wie:

- vbCr: Carriage-Return 'Entspricht "\ r" in C-Sprachen.
- vbLf: Line-Feed 'Entspricht "\ n" in C-Sprachen.
- vbCrLf: Wagenrücklauf und Zeilenvorschub (eine neue Zeile in Windows)
- vbTab: Tabulatorzeichen
- vbNullString: eine leere Zeichenfolge wie ""

Sie können diese Konstanten mit Verkettung und anderen Stringfunktionen verwenden, um String-Literale mit Sonderzeichen zu erstellen.

```
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
```

Die Verwendung von `vbNullString` wird aufgrund der unterschiedlichen Kompilierung des Codes als bessere Vorgehensweise als der entsprechende Wert von `""` angesehen. Auf Zeichenketten

wird über einen Zeiger auf einen zugewiesenen Speicherbereich zugegriffen, und der VBA-Compiler ist intelligent genug, um einen Nullzeiger zur Darstellung von `vbNullString` . Dem Literal `""` wird Speicher zugewiesen, als wäre es eine vom Typ String typisierte Variante, wodurch die Verwendung der Konstante wesentlich effizienter wird:

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

String Literals - Escape-Zeichen, nicht druckbare Zeichen und Zeilenfortsetzungen online lesen: https://riptutorial.com/de/vba/topic/3445/string-literals---escape-zeichen--nicht-druckbare-zeichen-und-zeilenfortsetzungen

Kapitel 38: Substrings

Bemerkungen

VBA verfügt über integrierte Funktionen zum Extrahieren bestimmter Teile von Zeichenfolgen, darunter:

- `Left` / `Left$`
- `Right` / `Right$`
- `Mid` / `Mid$`
- `Trim` / `Trim$`

Um eine implizite Typumwandlung onverhead (und damit eine bessere Leistung) zu vermeiden, verwenden Sie die $ -suffixed-Version der Funktion, wenn eine Zeichenfolgenvariable an die Funktion übergeben wird und / oder wenn das Ergebnis der Funktion einer Zeichenfolgenvariablen zugewiesen wird.

Das Übergeben eines `Null` Parameterwerts an eine $ -suffixed-Funktion führt zu einem Laufzeitfehler ("ungültige Verwendung von Null"). Dies ist insbesondere für Code relevant, der eine Datenbank beinhaltet.

Examples

Verwenden Sie Left oder Left $, um die 3 äußersten linken Zeichen einer Zeichenfolge abzurufen

```
Const baseString As String = "Foo Bar"

Dim leftText As String
leftText = Left$(baseString, 3)
'leftText = "Foo"
```

Verwenden Sie Right oder Right $, um die 3 Zeichen ganz rechts in einer Zeichenfolge zu erhalten

```
Const baseString As String = "Foo Bar"
Dim rightText As String
rightText = Right$(baseString, 3)
'rightText = "Bar"
```

Verwenden Sie Mid oder Mid $, um bestimmte Zeichen aus einer Zeichenfolge abzurufen

```
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"
```

Verwenden Sie Trim, um eine Kopie der Zeichenfolge ohne führende oder nachgestellte Leerzeichen abzurufen

```
'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"
```

Substrings online lesen: https://riptutorial.com/de/vba/topic/3481/substrings

Kapitel 39: Suche innerhalb von Strings nach dem Vorhandensein von Teilstrings

Bemerkungen

Wenn Sie nach dem Vorhandensein oder der Position eines Teilstrings innerhalb einer Zeichenfolge `InStrRev` müssen, bietet VBA die `InStr` und `InStrRev` Funktionen, die die Zeichenposition des Teilstrings in der Zeichenfolge zurückgeben, sofern vorhanden.

Examples

Verwenden Sie InStr, um festzustellen, ob eine Zeichenfolge eine Teilzeichenfolge enthält

```
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
```

Verwenden Sie InStr, um die Position der ersten Instanz einer Teilzeichenfolge zu ermitteln

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

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

Verwenden Sie InStrRev, um die Position der letzten Instanz einer Teilzeichenfolge zu ermitteln

```
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
```

Suche innerhalb von Strings nach dem Vorhandensein von Teilstrings online lesen: https://riptutorial.com/de/vba/topic/3480/suche-innerhalb-von-strings-nach-dem-vorhandensein-von-teilstrings

Kapitel 40: Variablen deklarieren

Examples

Implizite und explizite Erklärung

Wenn ein Codemodul keine `Option Explicit` am oberen `Option Explicit` des Moduls enthält, erstellt der Compiler automatisch (dh "implizit") Variablen, wenn Sie diese verwenden. Sie haben standardmäßig den Variablentyp `Variant` .

```
Public Sub ExampleDeclaration()

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

End Sub
```

Wenn im obigen Code `Option Explicit` angegeben ist, wird der Code unterbrochen, da die erforderlichen `Dim` Anweisungen für `someVariable` und `someOtherVariable` .

```
Option Explicit

Public Sub ExampleDeclaration()

    Dim someVariable As Long
    someVariable = 10

    Dim someOtherVariable As String
    someOtherVariable = "Hello World"

End Sub
```

Es wird empfohlen, Option Explicit in Codemodulen zu verwenden, um sicherzustellen, dass Sie alle Variablen deklarieren.

Informationen zum Festlegen dieser Option finden Sie unter VBA-Best Practices .

Variablen

Umfang

Eine Variable kann deklariert werden (um den Sichtbarkeitsgrad zu erhöhen):

- Verwenden Sie auf Prozedurebene das `Dim` Schlüsselwort in einer beliebigen Prozedur. eine *lokale Variable* .
- Auf Modulebene mit dem Schlüsselwort " `Private` in einem beliebigen Modultyp. ein *privates Feld* .
- Auf Instanzebene mit dem `Friend` Schlüsselwort in einem beliebigen Klassenmodul. ein

Freundfeld

- Auf Instanzebene: Verwenden Sie das Schlüsselwort `Public` in einem beliebigen Klassenmodul. ein *öffentliches Feld* .
- Verwenden Sie das `Public` Schlüsselwort global in einem *Standardmodul* . eine *globale Variable* .

Variablen sollten immer mit dem kleinstmöglichen Umfang deklariert werden: Bevorzugen Sie die Übergabe von Parametern an Prozeduren, anstatt globale Variablen zu deklarieren.

Weitere Informationen finden Sie unter Zugriffsmodifizierer .

Lokale Variablen

Verwenden Sie das `Dim` Schlüsselwort, um eine *lokale Variable* zu deklarieren:

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

Der Teil `[As Type]` der Deklarationssyntax ist optional. Wenn angegeben, wird der Datentyp der Variablen festgelegt, der bestimmt, wie viel Speicher dieser Variablen zugewiesen wird. Dies deklariert eine `String` Variable:

```
Dim identifierName As String
```

Wenn kein Typ angegeben wird, ist der Typ implizit `Variant` :

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

Die VBA-Syntax unterstützt auch das Deklarieren mehrerer Variablen in einer einzigen Anweisung:

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

Beachten Sie, dass für jede Variable (als 'Variant'-Variable) der `[As Type]` angegeben werden muss. Dies ist eine relativ häufige Falle:

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

Statische Variablen

Lokale Variablen können auch `Static` . In VBA wird das `Static` Schlüsselwort verwendet, um eine Variable an den Wert zu erinnern, den sie beim letzten Aufruf einer Prozedur hatte:

```
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
```

Hier wird die `values` als `Static` lokal deklariert. Da es sich um eine *Objektvariable handelt* , wird sie mit `Nothing` initialisiert. Die Bedingung , dass die Erklärung folgt überprüft , ob die Objektreferenz wurde `Set` vor - wenn es das erste Mal ist das Verfahren läuft, wird die Sammlung initialisiert. `DoSomethingElse` möglicherweise Elemente hinzu oder entfernt sie, und diese werden beim nächsten `DoSomething` weiterhin in der Auflistung `DoSomething` .

Alternative

> Das `Static` Schlüsselwort von VBA kann leicht missverstanden werden - *insbesondere* von erfahrenen Programmierern, die normalerweise in anderen Sprachen arbeiten. In vielen Sprachen wird `static` verwendet, um ein Klassenmitglied (Feld, Eigenschaft, Methode, ...) zum *Typ* und nicht zur *Instanz zu machen* . Code im `static` Kontext kann nicht auf Code im *Instanzkontext* verweisen. Das VBA-Schlüsselwort `Static` bedeutet etwas völlig anderes.

Eine `Static` lokale Variable kann häufig genauso gut als `Private` Variable auf Modulebene (Feld) implementiert werden. Dies stellt jedoch das Prinzip in Frage, nach dem eine Variable mit dem kleinstmöglichen Umfang deklariert werden sollte. Vertrauen Sie auf Ihre Instinkte, verwenden Sie, was Sie bevorzugen - beide werden funktionieren ... aber `Static` ohne zu verstehen, was es tut, kann zu interessanten Fehlern führen.

Dim vs. Private

Das Schlüsselwort `Dim` ist auf Verfahrens- und Modulebene legal. Die Verwendung auf Modulebene entspricht der Verwendung des `Private` Schlüsselworts:

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

Das Schlüsselwort " `Private` ist nur auf Modulebene zulässig. Dadurch werden `Dim` für lokale Variablen reserviert und Modulvariablen mit `Private` deklariert, insbesondere mit dem `Public` Schlüsselwort `Public` , das ohnehin verwendet werden müsste, um ein öffentliches Mitglied zu deklarieren. Alternativ kann `Dim` *überall verwendet werden* - worauf es ankommt, ist die *Konsistenz* :

"Private Felder"

- **DO** verwendet `Private` eine Variable auf Modulebene zu erklären.
- **DO** verwenden `Dim` eine lokale Variable zu deklarieren.
- Verwenden **Sie KEINE** `Dim` eine Variable auf Modulebene zu erklären.

"Überall verdunkeln"

- **DO** verwendet `Dim` alles privat / local zu erklären.
- Verwenden Sie **KEIN** `Private` , um eine Variable auf Modulebene zu deklarieren.
- **AVOID-** Deklaration für `Public` Felder. *

* Generell sollte man sowieso vermeiden, `Public` oder `Global` Felder zu deklarieren.

Felder

Eine auf Modulebene im *Deklarationsabschnitt* oben im Modulhauptteil deklarierte Variable ist ein *Feld* . Ein in einem *Standardmodul* deklariertes `Public` Feld ist eine *globale Variable* :

```
Public PublicField As Long
```

Auf eine Variable mit globalem Gültigkeitsbereich kann von überall aus zugegriffen werden, einschließlich anderer VBA-Projekte, die auf das Projekt verweisen, in dem es deklariert ist.

Verwenden Sie den `Friend` Modifizierer, um eine Variable global / public zu machen, jedoch nur innerhalb des Projekts sichtbar zu machen:

```
Friend FriendField As Long
```

Dies ist besonders nützlich in Add-Ins, bei denen die Absicht besteht, dass andere VBA-Projekte auf das Add-In-Projekt verweisen und die öffentliche API verwenden können.

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

Freundfelder sind in Standardmodulen nicht verfügbar.

Instanzfelder

Eine Variable auf Modulebene deklarierte, im *Deklarationsbereich* an der Oberseite des Körpers eines Klassenmodul (einschließlich `ThisWorkbook` , `ThisDocument` , `Worksheet` , `UserForm` und *Klasse - Module),* ist eine *Instanz Feld:* es existiert nur solange , wie es eine *Instanz* die Klasse um.

```
'> 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
```

Felder einkapseln

Instanzdaten werden häufig als `Private` gehalten und *gekapselt* überspielt. Ein privates Feld kann mithilfe einer `Property` Prozedur `Property` werden. Um eine private Variable öffentlich zugänglich zu machen, ohne dem Aufrufer Schreibzugriff zu gewähren, implementiert ein Klassenmodul (oder ein Standardmodul) ein `Property Get` Member:

```
Option Explicit
Private encapsulated As Long

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

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

Die Klasse selbst kann den gekapselten Wert ändern, der aufrufende Code kann jedoch nur auf die `Public` Mitglieder (und die `Friend` Mitglieder, wenn sich der Aufrufer im selben Projekt befindet) zugreifen.

Um dem Anrufer das Ändern zu ermöglichen:

- Ein gekapselter **Wert** , ein Modul macht ein `Property Let` .
- Ein gekapselter **Objektverweis** , ein Modul macht ein `Property Set` Mitglied `Property Set` .

Konstanten (Const)

Wenn Sie einen Wert haben, der in Ihrer Anwendung nie geändert wird, können Sie eine benannte Konstante definieren und anstelle eines Literalwerts verwenden.

Sie können Const nur auf Modul- oder Prozedurebene verwenden. Dies bedeutet, dass der Deklarationskontext für eine Variable eine Klasse, eine Struktur, ein Modul, eine Prozedur oder ein Block sein muss und keine Quelldatei, Namespace oder Schnittstelle sein kann.

```
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
```

Die Angabe von Konstantentypen kann zwar als bewährte Methode angesehen werden, ist jedoch nicht unbedingt erforderlich. Wenn Sie den Typ nicht angeben, führt dies immer noch zum richtigen Typ:

```
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
```

Beachten Sie, dass dies spezifisch für Konstanten ist und im Gegensatz zu Variablen, bei denen die Angabe des Typs nicht zu einem Variant-Typ führt.

Es ist zwar möglich, eine Konstante explizit als String zu deklarieren, es ist jedoch nicht möglich, eine Konstante mit einer String-Syntax mit fester Breite zu deklarieren

```
'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"
```

Zugriffsmodifizierer

Die `Dim` Anweisung sollte für lokale Variablen reserviert sein. Bevorzugen Sie auf Modulebene explizite Zugriffsmodifizierer:

- `Private` für private Felder, auf die nur innerhalb des Moduls zugegriffen werden kann, in dem sie deklariert sind.
- `Public` für öffentliche Felder und globale Variablen, auf die mit beliebigem aufrufendem Code zugegriffen werden kann.
- `Friend` für Variablen innerhalb des Projekts öffentlich, aber für andere referenzierende VBA-Projekte nicht zugänglich (relevant für Add-Ins)
- `Global` kann auch verwendet wird für `Public` Felder in Standardmodulen, ist aber illegal in Klassenmodulen und ist veraltet sowieso - das bevorzugt `Public` statt Modifikator. Dieser Modifikator ist auch für Verfahren nicht zulässig.

Zugriffsmodifizierer sind auf Variablen und Prozeduren gleichermaßen anwendbar.

```
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
```

Option Privates Modul

Öffentliche parameterlose `Sub` in Standardmodulen werden als Makros verfügbar gemacht und können an Steuerelemente und Tastenkombinationen im Hostdokument angehängt werden.

Umgekehrt werden öffentliche `Function` in Standardmodulen als benutzerdefinierte Funktionen (UDFs) in der Hostanwendung verfügbar gemacht.

Wenn Sie das `Option Private Module` oben in einem Standardmodul angeben, wird verhindert, dass seine Mitglieder als Makros und UDFs für die Hostanwendung angezeigt werden.

Typ Hinweise

Typhinweise werden **stark** entmutigt. Sie existieren und werden hier aus historischen Gründen und aus Gründen der Rückwärtskompatibilität dokumentiert. Verwenden Sie stattdessen die Syntax `As [DataType]` .

```
Public Sub ExampleDeclaration()

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

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

Durch Typhinweise wird die Lesbarkeit von Code erheblich verringert und eine ungarische Notation unterstützt, die *auch* die Lesbarkeit behindert:

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

Deklarieren Sie stattdessen Variablen näher an ihrer Verwendung und benennen Sie die Dinge nach dem Verwendungszweck, nicht nach ihrem Typ:

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

Typhinweise können auch für Literale verwendet werden, um einen bestimmten Typ durchzusetzen. Standardmäßig wird ein numerisches Literal, das kleiner als 32.768 ist, als `Integer` Literal interpretiert. Mit einem Typhinweis können Sie Folgendes steuern:

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

Typhinweise werden für Literale normalerweise nicht benötigt, da sie einer mit einem expliziten Typ deklarierten Variablen zugewiesen werden oder implizit in den entsprechenden Typ konvertiert werden, wenn sie als Parameter übergeben werden. Implizite Konvertierungen können mit einer der expliziten Typkonvertierungsfunktionen vermieden werden:

```
'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)
```

Integrierte Funktionen mit String-Rückgabe

Die Mehrheit der integrierten Funktionen, die Strings verarbeiten, gibt es in zwei Versionen: Eine lose typisierte Version, die eine `Variant` zurückgibt, und eine stark typisierte Version (die mit `$` endet), die einen `String` zurückgibt. Wenn Sie den Rückgabewert nicht einer `Variant` zuweisen, sollten Sie die Version vorziehen, die einen `String` zurückgibt. Andernfalls erfolgt eine implizite Konvertierung des Rückgabewerts.

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

Diese Funktionen sind:

- 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 $

Beachten Sie, dass diese Funktion *Aliase* sind, nicht ganz *Type Hints.* Die `Left` Funktion entspricht der verborgenen `B_Var_Left` Funktion, während die `Left$` -Version der verborgenen `B_Str_Left` Funktion entspricht.

In sehr frühen Versionen von VBA ist das Zeichen `$` kein erlaubtes Zeichen und der Funktionsname musste in eckige Klammern gesetzt werden. In Word Basic gab es viele, viele weitere Funktionen, die Zeichenfolgen zurückgaben, die mit $ endeten.

Zeichenfolgen mit fester Länge deklarieren

In VBA können Strings mit einer bestimmten Länge deklariert werden. Sie werden automatisch aufgefüllt oder abgeschnitten, um die angegebene Länge beizubehalten.

```
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
```

Wann wird eine statische Variable verwendet?

Eine lokal deklarierte statische Variable wird nicht zerstört und verliert ihren Wert nicht, wenn die Sub-Prozedur beendet wird. Nachfolgende Aufrufe der Prozedur erfordern keine Neuinitialisierung oder Zuweisung, obwohl Sie eventuell gespeicherte Werte auf Null setzen möchten.

Dies ist besonders nützlich, wenn ein Objekt in einem "Helfer" -Unter, das wiederholt aufgerufen wird, zu spät gebunden wird.

Ausschnitt 1: Verwenden Sie ein Scripting.Dictionary-Objekt in vielen Arbeitsblättern erneut

```
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
```

Ausschnitt 2: Erstellen Sie eine Arbeitsblatt-UDF, die das VBScript.RegExp-Objekt später bindet

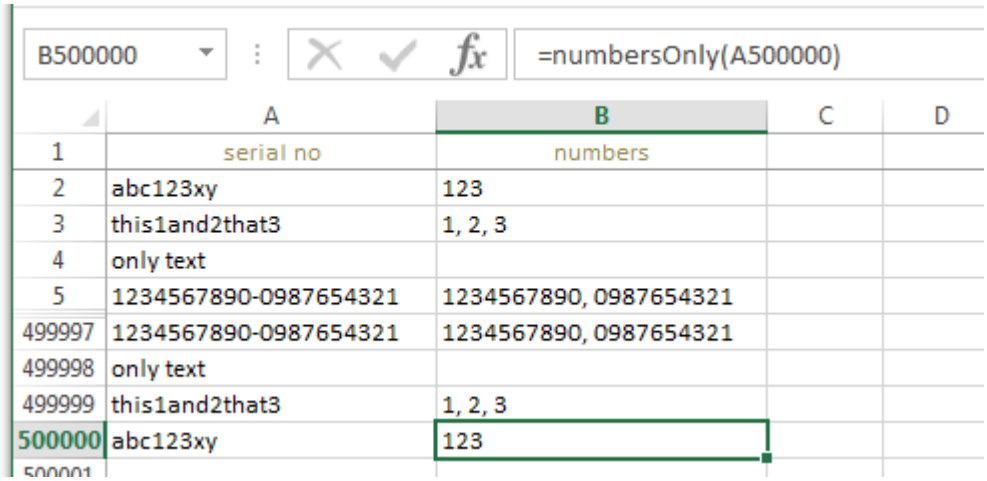
```
Option Explicit

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
```



Beispiel für eine UDF mit statischem Objekt, das über eine halbe Million Zeilen gefüllt ist

* Verstrichene Zeiten zum Füllen von 500.000 Zeilen mit UDF:
- mit **Dim rgx As Object** : 148,74 Sekunden
- mit **Static rgx As Object** : 26,07 Sekunden

* Diese sollten nur für den relativen Vergleich berücksichtigt werden. Ihre eigenen Ergebnisse variieren je nach Komplexität und
Umfang der durchgeführten Operationen.

Denken Sie daran, dass eine UDF während der Lebensdauer einer Arbeitsmappe nicht einmal berechnet wird. Sogar eine nichtflüchtige UDF wird neu berechnet, wenn sich die Werte in dem Bereich bzw. den Bereichen ändern, auf die sie verweisen. Jedes nachfolgende Neuberechnungsereignis erhöht nur die Vorteile einer statisch deklarierten Variablen.

- Eine statische Variable steht für die Lebensdauer des Moduls zur Verfügung, nicht die Prozedur oder Funktion, in der es deklariert und zugewiesen wurde.
- Statische Variablen können nur lokal deklariert werden.
- Statische Variablen enthalten viele der gleichen Eigenschaften wie eine private Modulebenenvariable, jedoch mit eingeschränktem Umfang.

Zugehörige Verweise: Statisch (Visual Basic)

Variablen deklarieren online lesen: https://riptutorial.com/de/vba/topic/877/variablen-deklarieren

Kapitel 41: VBA-Laufzeitfehler

Einführung

Code, der kompiliert wird, kann zur Laufzeit noch fehlerhaft sein. In diesem Thema werden die häufigsten Ursachen, ihre Ursachen und Möglichkeiten zur Vermeidung aufgeführt.

Examples

Laufzeitfehler '3': Rückgabe ohne GoSub

Falscher Code

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

Warum funktioniert das nicht?

Die Ausführung tritt in die `DoSomething` Prozedur ein, springt zum `DoThis` Label und gibt "Hi!" Aus. *kehrt* der Befehl unmittelbar nach dem Aufruf von `GoSub` zur Anweisung zurück und druckt "Hi!" wieder, und trifft dann auf eine `Return` - Anweisung, aber es gibt nirgendwo jetzt *zurück,* weil wir hier nicht mit bekommen haben `GoSub` - Anweisung.

Code korrigieren

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

Warum funktioniert das?

Durch die Einführung eines `Exit Sub` Anweisung *vor* der `DoThis` Label Linie haben wir die getrennt `DoThis` Unterprogramm von dem Rest des Verfahrens Körper - der einzige Weg , die auszuführen `DoThis` Unterprogramm ist über den `GoSub` Sprung.

Weitere Hinweise

`GoSub` / `Return` ist veraltet und sollte zu Gunsten von Prozeduraufrufen vermieden werden. Eine Prozedur sollte keine anderen Unterprogramme als Fehlerbehandlungsroutinen enthalten.

Dies ist dem Laufzeitfehler '20' sehr ähnlich : Ohne Fehler fortfahren ; In beiden Situationen besteht die Lösung darin, sicherzustellen, dass der *normale Ausführungspfad* nicht ohne expliziten Sprung in eine Subroutine (durch ein Leitungsetikett gekennzeichnet) einsteigen kann (vorausgesetzt, `On Error GoTo` wird als *expliziter Sprung betrachtet*).

Laufzeitfehler '6': Überlauf

Falscher Code

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

Warum funktioniert das nicht?

Der `Integer` Datentyp ist eine vorzeichenbehaftete 16-Bit-Ganzzahl mit einem Maximalwert von 32.767. Wenn Sie es einer größeren Größe zuweisen, wird der Typ *überlaufen* und dieser Fehler wird *ausgelöst* .

Korrigieren Sie den Code

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

Warum funktioniert das?

Durch die Verwendung einer `Long` -Ganzzahl (32-Bit) können wir jetzt eine Schleife erstellen, die mehr als 32.767-mal durchläuft, ohne den Typ der Zählervariable zu überlaufen.

Weitere Hinweise

Weitere Informationen finden Sie unter Datentypen und Grenzwerte .

Laufzeitfehler '9': Index außerhalb des gültigen Bereichs

Falscher Code

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

Warum funktioniert das nicht?

`foo` ist ein Array, das 10 Elemente enthält. Wenn der `i` Schleifenzähler einen Wert von 11 erreicht, ist `foo(i)` *außerhalb des Bereichs* . Dieser Fehler tritt auf, wenn auf ein Array oder eine Sammlung mit einem Index zugegriffen wird, der in diesem Array oder dieser Sammlung nicht vorhanden ist.

Korrigieren Sie den Code

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

Warum funktioniert das?

Verwenden `LBound` Funktionen `LBound` und `UBound` , um die Unter- bzw. Obergrenze eines Arrays zu bestimmen.

Weitere Hinweise

Wenn der Index eine Zeichenfolge ist, z. B. `ThisWorkbook.Worksheets("I don't exist")` , bedeutet dieser Fehler, dass der angegebene Name in der abgefragten Sammlung nicht vorhanden ist.

Der tatsächliche Fehler ist jedoch implementierungsspezifisch. `Collection` wird stattdessen der Laufzeitfehler 5 "Ungültiger Prozeduraufruf oder -argument" ausgelöst:

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

Laufzeitfehler '13': Typenkonflikt

Falscher Code

```
Public Sub DoSomething()
```

```
    DoSomethingElse "42?"
End Sub

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

Warum funktioniert das nicht?

VBA ist sehr bemüht, die `"42?"` Argument in einen `Date` . Wenn es fehlschlägt, wird der Anruf an `DoSomethingElse` kann nicht ausgeführt werden, weil VBA nicht weiß , zu welchem Zeitpunkt zu passieren, so dass es wirft *Laufzeitabgleichfehler* 13 - *Typ,* da der Typ des Arguments nicht den erwarteten Typ übereinstimmt (und kann (kann nicht implizit konvertiert werden).

Korrigieren Sie den Code

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

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

Warum funktioniert das?

Durch das Übergeben eines `Date` Arguments an eine Prozedur, die einen `Date` Parameter erwartet, kann der Aufruf erfolgreich sein.

Laufzeitfehler '91': Objektvariable oder Mit Blockvariable nicht gesetzt

Falscher Code

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

Warum funktioniert das nicht?

Objektvariablen halten einen *Verweis,* und Referenzen müssen die *gesetzt* werden mit `Set` - Schlüsselwort. Dieser Fehler tritt immer dann auf, wenn ein Member-Aufruf für ein Objekt ausgeführt wird, dessen Referenz `Nothing` . In diesem Fall ist `foo` eine `Collection` , die jedoch nicht initialisiert ist. `.Add` enthält die Referenz `Nothing` - und wir können `.Add` on `Nothing` nicht aufrufen.

Korrigieren Sie den Code

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

Warum funktioniert das?

Durch das Zuweisen der Objektvariablen mit dem Schlüsselwort `Set` eine gültige Referenz, sind die Aufrufe von `.Add` erfolgreich.

Weitere Hinweise

Häufig kann eine Funktion oder Eigenschaft eine Objektreferenz zurückgeben. Ein häufiges Beispiel ist die `Range.Find` Methode von Excel, die ein `Range` Objekt zurückgibt:

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

Die Funktion kann jedoch sehr gut `Nothing` (wenn der Suchbegriff nicht gefunden wird), so dass der verkettete `.Row` Member-Aufruf wahrscheinlich fehlschlägt.

Stellen Sie vor dem Aufrufen von Objektmitgliedern sicher, dass der Verweis mit der Bedingung "`If Not xxxx Is Nothing` ist:

```
Dim result As Range
Set result = SomeSheet.Cells.Find("Something")

Dim resultRow As Long
If Not result Is Nothing Then resultRow = result.Row
```

Laufzeitfehler '20': Ohne Fehler fortsetzen

Falscher Code

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

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

Warum funktioniert das nicht?

Wenn die `DoSomethingElse` Prozedur einen Fehler `DoSomethingElse` , springt die Ausführung zur `CleanFail` Zeilenbeschriftung, gibt die Fehlernummer aus und die Anweisung `Resume Next` springt zurück zu der Anweisung, die unmittelbar auf die Zeile folgt, in der der Fehler aufgetreten ist, in diesem Fall `Debug.Print` Anweisung: Die Fehlerbehandlungs-Subroutine wird ohne einen Fehlerkontext ausgeführt. Wenn die Anweisung `Resume Next` erreicht wird, wird der Laufzeitfehler 20 ausgelöst, da nirgends weitergegangen werden kann.

Code korrigieren

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

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

Warum funktioniert das?

Durch die Einführung einer `Exit Sub` Anweisung vor der `CleanFail` Zeilenbezeichnung haben wir die `CleanFail` Fehlerbehandlungs-Subroutine vom Rest des Prozedurenkörpers getrennt. Die einzige Möglichkeit, die Fehlerbehandlungs-Subroutine auszuführen, ist über einen `On Error` Sprung. Daher erreicht kein Ausführungspfad die `Resume` Anweisung außerhalb eines Fehlerkontexts, wodurch der Laufzeitfehler 20 vermieden wird.

Weitere Hinweise

Dies ist dem Laufzeitfehler '3' sehr ähnlich : Return ohne GoSub ; In beiden Situationen besteht die Lösung darin, sicherzustellen, dass der *normale Ausführungspfad* nicht ohne expliziten Sprung in eine Subroutine (durch ein Leitungsetikett gekennzeichnet) einsteigen kann (vorausgesetzt, `On Error GoTo` wird als *expliziter Sprung betrachtet*).

VBA-Laufzeitfehler online lesen: https://riptutorial.com/de/vba/topic/8917/vba-laufzeitfehler

Kapitel 42: VBA-Optionsschlüsselwort

Syntax

- Option optionName [Wert]
- Option explizit
- Option Vergleichen {Text | Binär | Datenbank}
- Option Privates Modul
- Optionsbasis {0 | 1}

Parameter

Möglichkeit	Detail
Explizit	*Fordern Sie eine Variablendeklaration* in dem Modul an, in dem es angegeben ist (idealerweise alle). Wenn diese Option angegeben ist, wird die Verwendung einer nicht deklarierten (/ falsch geschriebenen) Variablen zu einem Kompilierungsfehler.
Text vergleichen	Macht die String-Vergleiche des Moduls unabhängig von der Groß- und Kleinschreibung, basierend auf der Systemumgebung, und priorisiert die alphabetische Äquivalenz (z. B. "a" = "A").
Vergleichen Sie binär	Standard-Stringvergleichsmodus Die String-Vergleiche des Moduls müssen zwischen Groß- und Kleinschreibung unterschieden werden. Dabei werden Zeichenfolgen anhand des binären Repräsentations- / Zahlenwerts jedes Zeichens (z. B. ASCII) verglichen.
Vergleichen Sie die Datenbank	(Nur MS-Access) Lässt die String-Vergleiche des Moduls wie in einer SQL-Anweisung funktionieren.
Privatmodul	Verhindert, dass auf das `Public` Mitglied des Moduls von außerhalb des Projekts, in dem sich das Modul befindet, zugegriffen wird, wodurch Prozeduren effektiv von der Hostanwendung verborgen werden (dh nicht als Makros oder benutzerdefinierte Funktionen verwendet werden können).
Optionsbasis 0	Voreinstellung. Legt das implizite Array in einem Modul auf `0` . Wenn ein Array ohne explizit unteren Grenzwert deklariert wird, wird `0` verwendet.
Optionsbasis 1	Legt das implizite Array in einem Modul auf `1` . Wenn ein Array ohne explizit unteren Grenzwert deklariert wird, wird `1` verwendet.

Bemerkungen

Es ist viel einfacher, die Grenzen von Arrays zu steuern, indem Sie die Grenzen explizit deklarieren, anstatt den Compiler auf eine `Option Base {0|1}` Deklaration `Option Base {0|1}` zurückgreifen zu lassen. Dies kann wie folgt gemacht werden:

```
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

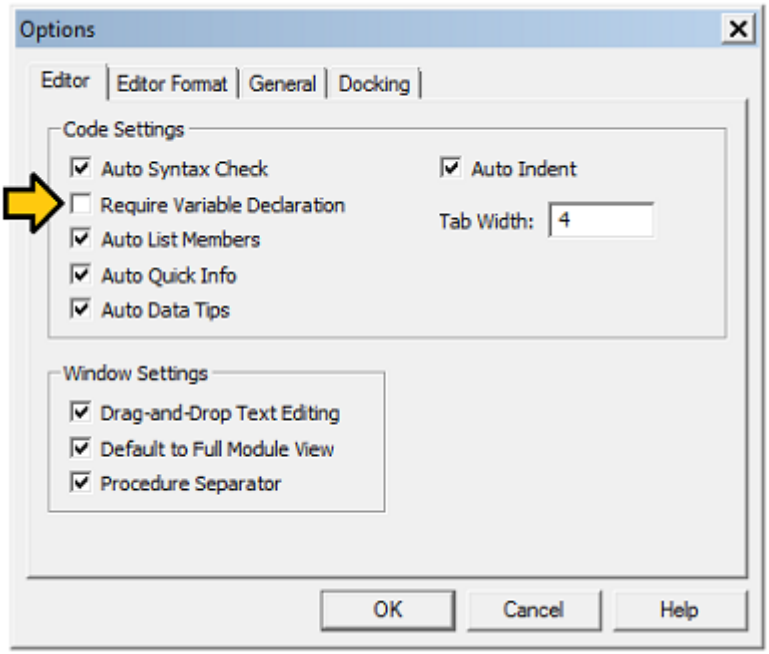
Option explizit

Es gilt als bewährte Methode, `Option Explicit` in VBA zu verwenden, da der Entwickler dazu gezwungen wird, alle Variablen vor der Verwendung zu deklarieren. Dies hat auch andere Vorteile, z. B. die automatische Aktivierung von deklarierten Variablennamen und IntelliSense.

```
Option Explicit

Sub OptionExplicit()
    Dim a As Integer
    a = 5
    b = 10 '// Causes compile error as 'b' is not declared
End Sub
```

Wenn **Sie** die **Option Require Variablendeklaration** in den Tools der VBE ▸ `Extras` ▸ `Editor-Eigenschaftsseite festlegen, wird die` **Option Explicit-** Anweisung oben in jedem neu erstellten Codeblatt angezeigt.



Dadurch werden dumme Codierungsfehler wie Rechtschreibfehler vermieden und Sie können den korrekten Variablentyp in der Variablendeklaration verwenden. (Einige weitere Beispiele finden Sie unter IMMER Verwenden Sie "Option Explicit" .)

Option Compare Binary

Durch den binären Vergleich werden alle Prüfungen auf String-Gleichheit innerhalb eines Moduls / einer Klasse zwischen Groß- und *Kleinschreibung unterschieden* . Mit dieser Option werden Zeichenfolgenvergleiche technisch unter Verwendung der Sortierreihenfolge der binären Repräsentationen jedes Zeichens durchgeführt.

A <B <E <Z <a <b <e <z

Wenn in einem Modul kein Option Compare angegeben ist, wird standardmäßig Binary verwendet.

```
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
```

Option Text vergleichen

Die Option Text vergleichen bewirkt, dass alle Zeichenfolgenvergleiche innerhalb eines Moduls / einer Klasse einen von Groß- und *Kleinschreibung abhängigen* Vergleich verwenden.

(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
```

Option Compare Database

Die Option Compare Database ist nur in MS Access verfügbar. Hiermit wird festgelegt, dass das Modul / die Klasse die aktuellen Datenbankeinstellungen verwendet, um zu bestimmen, ob der Text- oder Binärmodus verwendet werden soll.

Hinweis: Die Verwendung dieser Einstellung wird nicht empfohlen, es sei denn, das Modul wird zum Schreiben von benutzerdefinierten Access-UDFs (benutzerdefinierte Funktionen) verwendet, die Textvergleiche auf dieselbe Weise wie SQL-Abfragen in dieser Datenbank behandeln sollen.

Optionsbasis {0 | 1}

`Option Base` wird verwendet, um die voreingestellte untere Grenze von **Arrayelementen** zu deklarieren. Es wird auf Modulebene deklariert und gilt nur für das aktuelle Modul.

Standardmäßig (und wenn keine Optionsbasis angegeben ist) ist die Basis 0. Dies bedeutet, dass das erste Element eines im Modul deklarierten Arrays den Index 0 hat.

Wenn `Option Base 1` angegeben ist, hat das erste Array-Element den Index 1

Beispiel in Basis 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
```

Gleiches Beispiel mit Basis 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
```

Das zweite Beispiel hat in der ersten Schleifenphase ein Subscript außerhalb des Bereichs (Fehler 9) generiert, da versucht wurde, auf den Index 0 des Arrays zuzugreifen, und dieser Index ist nicht vorhanden, da das Modul mit `Base 1` deklariert ist

Der korrekte Code für Base 1 lautet:

```
    For i = 1 To UBound(myStrings)

        Debug.Print myStrings(i) ' This will print "Apple", then "Orange", then "Peach"

    Next i
```

Es sollte beachtet werden, dass die Split-Funktion unabhängig von einer `Option Base` Einstellung **immer** ein Array mit einem nullbasierten Elementindex erstellt. Beispiele zur Verwendung der

Split- Funktion finden Sie hier

> Split-Funktion
>
> > Gibt ein nullbasiertes, eindimensionales Array mit einer angegebenen Anzahl von Teilzeichenfolgen zurück.

In Excel geben die Eigenschaften `Range.Value` und `Range.Formula` für einen mehrzelligen Bereich *immer* ein 1-basiertes 2D-Variant-Array zurück.

In ähnlicher Weise gibt die `Recordset.GetRows` Methode in ADO *immer* ein 1-basiertes 2D-Array zurück.

Es wird empfohlen, immer die Funktionen LBound und UBound zu verwenden, um die Ausdehnungen eines Arrays zu bestimmen.

```
'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
```

Die `Option Base 1` muss an der Spitze jedes Codemoduls stehen, in dem ein Array erstellt oder neu dimensioniert wird, wenn Arrays konsistent mit einer unteren Grenze von 1 erstellt werden sollen.

VBA-Optionsschlüsselwort online lesen: https://riptutorial.com/de/vba/topic/3992/vba-optionsschlusselwort

Kapitel 43: Veranstaltungen

Syntax

- **Quellmodul** : `[Public] Event [identifier]([argument_list])`
- **Handler-Modul** : `Dim|Private|Public WithEvents [identifier] As [type]`

Bemerkungen

- Eine Veranstaltung kann nur `Public` . Der Modifizierer ist optional, da die Mitglieder des Klassenmoduls (einschließlich der Ereignisse) standardmäßig implizit `Public` .
- Eine `WithEvents` Variable kann `Private` oder `Public` , nicht jedoch `Friend` . Der Modifikator ist obligatorisch, da `WithEvents` kein Schlüsselwort ist, das eine Variable deklariert, sondern ein Modifikatorschlüsselwort, das Teil der Variablendeklarationssyntax ist. Daher muss das `Dim` Schlüsselwort verwendet werden, wenn kein Zugriffsmodifizierer vorhanden ist.

Examples

Quellen und Handler

Was sind Ereignisse?

VBA ist *ereignisgesteuert* : VBA-Code wird als Reaktion auf Ereignisse ausgeführt, die von der Hostanwendung oder dem Hostdokument ausgelöst werden. Das Verständnis von Ereignissen ist für das Verständnis von VBA von grundlegender Bedeutung.

APIs setzen häufig Objekte frei, die eine Reihe von *Ereignissen* als Reaktion auf verschiedene Zustände auslösen. Ein `Excel.Application` Objekt löst beispielsweise ein Ereignis aus, wenn eine neue Arbeitsmappe erstellt, geöffnet, aktiviert oder geschlossen wird. Oder wann immer ein Arbeitsblatt berechnet wird. Oder kurz bevor eine Datei gespeichert wird. Oder gleich danach. Eine Schaltfläche in einem Formular löst ein `Click` Ereignis aus, wenn der Benutzer darauf klickt, das Benutzerformular selbst ein Ereignis auslöst, unmittelbar nachdem es aktiviert wurde, und ein anderes unmittelbar vor dem Schließen.

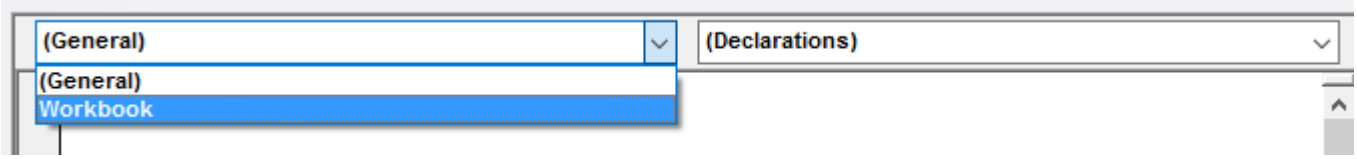
Aus API-Sicht sind Ereignisse *Erweiterungspunkte* : Der Clientcode kann auswählen, dass Code implementiert wird, *der* diese Ereignisse verarbeitet, und benutzerdefinierten Code ausführen, wenn diese Ereignisse ausgelöst werden. Auf diese Weise können Sie Ihren benutzerdefinierten Code bei jeder Änderung der Auswahl in einem Arbeitsblatt automatisch ausführen - durch Behandeln des Ereignisses, das ausgelöst wird, wenn sich die Auswahl in einem Arbeitsblatt ändert.

Ein Objekt, das Ereignisse verfügbar macht, ist eine *Ereignisquelle* . Eine Methode, die ein

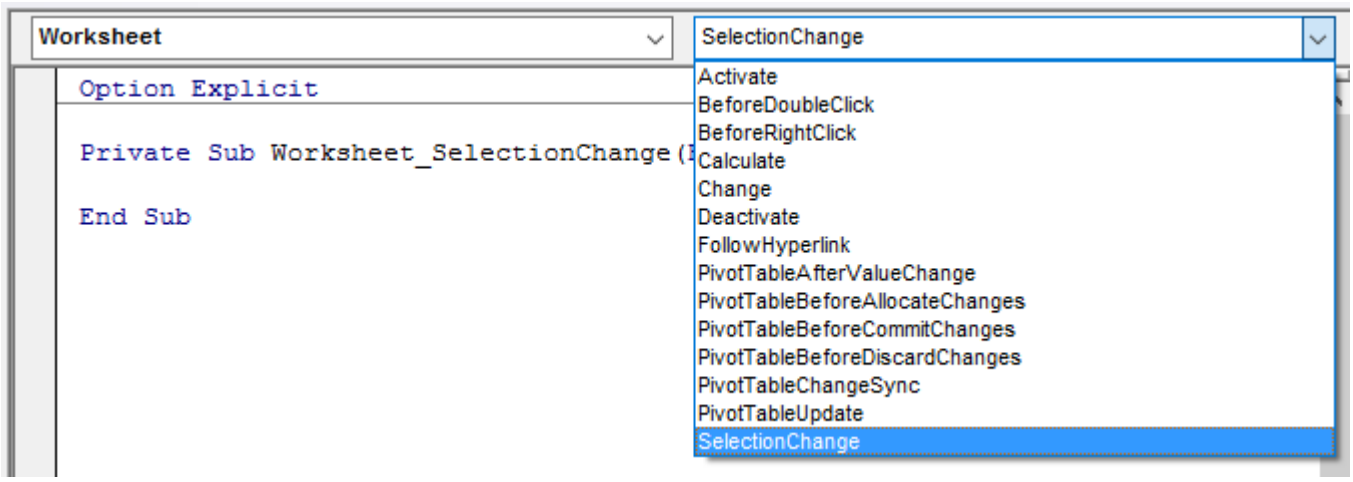
Ereignis behandelt, ist ein *Handler* .

Handler

VBA-Dokumentmodule (z. B. `ThisDocument` , `ThisWorkbook` , `Sheet1` usw.) und `UserForm` Module sind *Klassenmodule* , *die* spezielle Schnittstellen *implementieren* , die eine Reihe von *Ereignissen* `UserForm` . Sie können diese Schnittstellen in der linken Dropdown-Liste oben im Codefenster durchsuchen:



In der rechten Dropdownliste werden die Mitglieder der in der linken Dropdownliste ausgewählten Benutzeroberfläche aufgelistet:



Die VBE generiert automatisch einen Event-Handler-Stub, wenn ein Element in der rechten Liste ausgewählt wird, oder navigiert dorthin, wenn der Handler vorhanden ist.

In jedem Modul können Sie eine `WithEvents` Variable mit `WithEvents` definieren:

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

Jede `WithEvents` Deklaration kann aus der Dropdown-Liste auf der linken Seite ausgewählt werden. Wenn ein Ereignis in der rechten Dropdown-Liste ausgewählt wird, generiert die VBE einen Ereignishandler-Stub, der nach dem `WithEvents` Objekt und dem Namen des Ereignisses benannt ist, verbunden mit einem Unterstrich:

```
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
```

Mit `WithEvents` können nur Typen verwendet werden, die mindestens ein Ereignis `WithEvents` . Mit `WithEvents` Deklarationen kann vor Ort keine Referenz mit dem Schlüsselwort `New` zugewiesen werden. Dieser Code ist illegal:

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

Die Objektreferenz muss `Set` explizit; In einem Klassenmodul befindet sich ein guter Ort dafür häufig im `Class_Initialize` Handler, da die Klasse dann die Ereignisse dieses Objekts so lange behandelt, wie ihre Instanz vorhanden ist.

Quellen

Jedes Klassenmodul (oder Dokumentmodul oder Benutzerformular) kann eine Ereignisquelle sein. Verwenden Sie das `Event` , um die *Signatur* für das Ereignis im *Deklarationsabschnitt* des Moduls zu definieren:

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

Die Signatur des Ereignisses bestimmt, wie das Ereignis ausgelöst wird und wie die Ereignishandler aussehen.

Ereignisse können nur innerhalb der Klasse ausgelöst werden , in der sie definiert sind - Client-Code kann sie nur *verarbeiten* . Ereignisse werden mit dem Schlüsselwort `RaiseEvent` . Die Argumente des Ereignisses werden an dieser Stelle bereitgestellt:

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

Ohne Code, der das `SomethingHappened` Ereignis behandelt, wird das Ausführen der `DoSomething` Prozedur immer noch das Ereignis `DoSomething` , aber nichts passiert. Angenommen, die Ereignisquelle ist der obige Code in einer Klasse mit dem Namen `Something` . Dieser Code in `ThisWorkbook` zeigt ein Meldungsfeld mit der Aufschrift "Hallo" an, wenn `test.DoSomething` aufgerufen wird:

```
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
```

Zurückgeben von Daten an die Ereignisquelle

Verwenden von Parametern, die als Referenz übergeben werden

Ein Ereignis kann einen `ByRef` Parameter definieren, der an den Aufrufer zurückgegeben werden soll:

```
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
```

Wenn das `BeforeSomething` Ereignis hat einen Handler, der seine Sets `cancel` Parameter auf `True` , dann , wenn die Ausführung kehrt aus dem Handler, `cancel` wird `True` und `AfterSomething` nie angehoben werden.

```
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
```

Angenommen, die `foo` Objektreferenz wird irgendwo zugewiesen, wenn `foo.DoSomething` , werden Sie in einem Meldungsfeld gefragt, ob Sie den `foo.DoSomething` abbrechen `foo.DoSomething` , und in einem zweiten Meldungsfeld wird die Meldung "nicht abgebrochen" nur `foo.DoSomething` , wenn `Nein` ausgewählt wurde.

Verwenden von veränderlichen Objekten

Sie können auch eine Kopie des veränderlichen Objekts `ByVal` und Handler die Eigenschaften dieses Objekts ändern lassen. Der Aufrufer kann dann die geänderten Eigenschaftswerte lesen und entsprechend handeln.

```
'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
```

Kombiniert mit dem `Variant` Typ kann dies verwendet werden, um auf nicht offensichtliche Weise einen Wert an den Aufrufer zurückzugeben:

```
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
```

Der Handler würde so aussehen:

```
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
```

Veranstaltungen online lesen: https://riptutorial.com/de/vba/topic/5278/veranstaltungen

Kapitel 44: Zeichenketten deklarieren und zuweisen

Bemerkungen

Zeichenfolgen sind ein Referenztyp und für die meisten Programmieraufgaben von zentraler Bedeutung. Zeichenfolgen wird Text zugewiesen, auch wenn der Text numerisch ist. Zeichenfolgen können eine Länge von null oder eine beliebige Länge von bis zu 2 GB haben. Moderne Versionen von VBA speichern Strings intern mit einem Byte-Array aus Bytes mit mehreren Byte-Zeichensätzen (Alternative zu Unicode).

Examples

Deklarieren Sie eine String-Konstante

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

Deklarieren Sie eine Stringvariable mit variabler Breite

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

Deklarieren und weisen Sie eine Zeichenfolge mit fester Breite zu

```
'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"
```

Deklarieren Sie und weisen Sie ein String-Array zu

```
'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"
```

Weisen Sie mithilfe der Mid-Anweisung bestimmte Zeichen innerhalb einer Zeichenfolge zu

VBA bietet eine Mid-Funktion für die *Rückgabe von* Teilstrings innerhalb eines Strings, aber auch das Mid- *Statement, mit* dem Sie Teilstrings oder einzelne Zeichen mit einem String zuweisen können.

Die `Mid` Funktion wird normalerweise auf der rechten Seite einer Zuweisungsanweisung oder in einer Bedingung angezeigt. Die `Mid` Anweisung wird jedoch normalerweise auf der linken Seite einer Zuweisungsanweisung angezeigt.

```
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
```

Hinweis: Wenn Sie einzelnen *Bytes* in einer Zeichenfolge anstelle von einzelnen *Zeichen* in einer Zeichenfolge zuweisen müssen (siehe Anmerkungen zum Multi-Byte-Zeichensatz), kann die `MidB` Anweisung verwendet werden. In diesem Fall ist das zweite Argument für die `MidB` Anweisung die 1-basierte Position des Bytes, an dem die Ersetzung beginnen soll. Die entsprechende Zeile zum obigen Beispiel wäre also `MidB(surname, 5, 2) = "y"` .

Zuordnung zu und von einem Bytearray

Zeichenfolgen können Byte-Arrays direkt zugewiesen werden und umgekehrt. Denken Sie daran, dass Strings in einem Multi-Byte-Zeichensatz gespeichert werden (siehe Anmerkungen unten), sodass nur jeder andere Index des resultierenden Arrays den Teil des Zeichens darstellt, der in den ASCII-Bereich fällt.

```
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."
```

Zeichenketten deklarieren und zuweisen online lesen:
https://riptutorial.com/de/vba/topic/3446/zeichenketten-deklarieren-und-zuweisen

Kapitel 45: Zeichenketten verketten

Bemerkungen

Zeichenfolgen können mithilfe eines oder mehrerer Verkettungsoperatoren `&` verkettet oder zusammengefügt werden.

String-Arrays können auch mit der `Join` Funktion verkettet werden und bieten einen String (der null sein kann) für die Verwendung zwischen jedem Array-Element.

Examples

Verketten Sie Zeichenfolgen mit dem Operator &

```
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"
```

Verketten Sie ein String-Array mit der Join-Funktion

```
'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"
```

Zeichenketten verketten online lesen: https://riptutorial.com/de/vba/topic/3580/zeichenketten-verketten

Kapitel 46: Zuweisen von Zeichenfolgen mit wiederholten Zeichen

Bemerkungen

Manchmal müssen Sie eine Zeichenfolgenvariable mit einem bestimmten Zeichen zuweisen, das eine bestimmte Anzahl von Malen wiederholt wird. VBA bietet zu diesem Zweck zwei Hauptfunktionen:

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

Examples

Verwenden Sie die String-Funktion, um eine Zeichenfolge mit n wiederholten Zeichen zuzuweisen

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

Verwenden Sie die Funktionen String und Space, um eine Zeichenfolge mit n Zeichen zuzuweisen

```
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, " ")
```

Zuweisen von Zeichenfolgen mit wiederholten Zeichen online lesen:
https://riptutorial.com/de/vba/topic/3581/zuweisen-von-zeichenfolgen-mit-wiederholten-zeichen

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit 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	2GB + -Dateien in binärer Form in VBA und File Hashes lesen	PatricK
3	Andere Typen in Strings konvertieren	ThunderFrame
4	API-Aufrufe	paul bica
5	Arbeiten mit Dateien und Verzeichnissen ohne Verwendung von FileSystemObject	Comintern, Macro Man, SandPiper
6	Argumente übergeben ByRef oder ByVal	Branislav Kollár, Comintern, Mat's Mug, R3uK, RamenChef, ZygD
7	Arrays	Comintern, Dave, Hubisan, jamheadart, Josan Iracheta, Maarten van Stam, Mark.R, Mat's Mug, Miguel_Ryu, Tazaf
8	Arrays kopieren, zurückgeben und übergeben	Mark.R
9	Attribute	hymced, Mat's Mug, RamenChef, RubberDuck
10	Automatisierung oder Verwendung anderer Anwendungsbibliotheken	Branislav Kollár
11	Bedingte Kompilierung	Macro Man, Mat's Mug, RubberDuck, Steve Rindsberg
12	Bemerkungen	Comintern, Hosch250, Johnny C, litelite, Macro Man, Nijin22, Shawn V. Wilson, ThunderFrame
13	Benutzerformulare	Mat's Mug
14	CreateObject vs.	Branislav Kollár, Dave, Tim

	GetObject	
15	Datenstrukturen	Blackhawk
16	Datentypen und Grenzwerte	Comintern, FreeMan, Neil Mussett, StackzOfZtuff, Stephen Leppik, ThunderFrame
17	Datums-Uhrzeit-Manipulation	Comintern, FreeMan, Thomas G
18	Eine benutzerdefinierte Klasse erstellen	Branislav Kollár, Macro Man, Mat's Mug, Neil Mussett, ThunderFrame
19	Fehlerbehandlung	Comintern, Logan Reed, Mat's Mug
20	Flusssteuerungsstrukturen	Benno Grimm, Comintern, Kelly Tessena Keck, Leviathan, litelite, Macro Man, Martin, Mat's Mug, Roland, Siva, ThunderFrame
21	Häufig verwendete String-Manipulation	pashute
22	Länge der Saiten messen	Steve Rindsberg, ThunderFrame
23	Makrosicherheit und Signatur von VBA-Projekten / -Modulen	0m3r
24	Mit ADO arbeiten	Comintern, SandPiper, Tazaf
25	Nicht-lateinische Zeichen	Neil Mussett
26	Objektorientierte VBA	IvenBach, Mat's Mug
27	Operatoren	Comintern, Macro Man
28	Prozedur erstellen	Comintern, LiamH, Mat's Mug, Sivaprasath Vadivel, Tomas Zubiri
29	Prozeduraufrufe	Macro Man, Mat's Mug, Neil Mussett, Sam Johnson
30	Regeln der Namensgebung	FreeMan, Kaz, Mat's Mug, Victor Moraes
31	Rekursion	Mat's Mug, ThunderFrame
32	Sammlungen	Comintern
33	Schnittstellen	Neil Mussett
34	Scripting.Dictionary-Objekt	Comintern, Jeeped, Kyle, RamenChef, Tim, Wolf, Zev

		Spitz
35	Scripting.FileSystemObject	Comintern, Dave, Macro Man, Mikegrann, RubberDuck, Siva, Steve Rindsberg, ThunderFrame
36	Sortierung	Neil Mussett
37	String Literals - Escape-Zeichen, nicht druckbare Zeichen und Zeilenfortsetzungen	Comintern, ThunderFrame
38	Substrings	Mat's Mug, ThunderFrame
39	Suche innerhalb von Strings nach dem Vorhandensein von Teilstrings	ThunderFrame
40	Variablen deklarieren	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
41	VBA-Laufzeitfehler	Branislav Kollár, Macro Man, Mat's Mug
42	VBA-Optionsschlüsselwort	Jeeped, Maarten van Stam, Macro Man, Mat's Mug, RamenChef, RubberDuck, Stefan Pinnow, Thomas G, ThunderFrame
43	Veranstaltungen	Mat's Mug
44	Zeichenketten deklarieren und zuweisen	Comintern, ThunderFrame
45	Zeichenketten verketten	ThunderFrame
46	Zuweisen von Zeichenfolgen mit wiederholten Zeichen	ThunderFrame