Kostenloses eBook

LERNEN

excel-vba

Free unaffiliated eBook created from
Stack Overflow contributors.

#excel-vba

Inhaltsverzeichnis

Über ... 1

Kapitel 1: Erste Schritte mit Excel-VBA ... 2

Bemerkungen ... 2

Versionen ... 2

VB ... 2

Excel ... 3

Examples ... 3

Variablen deklarieren ... 3

Andere Arten, Variablen zu deklarieren, sind: ... 4

Öffnen des Visual Basic Editors (VBE) ... 5

Eine neue Objektbibliothek hinzufügen ... 6

Hallo Welt ... 11

Erste Schritte mit dem Excel-Objektmodell ... 13

Kapitel 2: Anwendungsobjekt ... 17

Bemerkungen ... 17

Examples ... 17

Beispiel für ein einfaches Anwendungsobjekt: Minimieren Sie das Excel-Fenster ... 17

Beispiel eines einfachen Anwendungsobjekts: Anzeige der Excel- und VBE-Version ... 17

Kapitel 3: Arbeiten mit Excel-Tabellen in VBA ... 18

Einführung ... 18

Examples ... 18

Instanziieren eines ListObjects ... 18

Mit ListRows / ListColumns arbeiten ... 18

Konvertieren einer Excel-Tabelle in einen normalen Bereich ... 18

Kapitel 4: Arbeitsmappen ... 20

Examples ... 20

Anwendungs-Arbeitsmappen ... 20

Wann Verwenden von ActiveWorkbook und ThisWorkbook ... 20

Öffnen einer (neuen) Arbeitsmappe, auch wenn sie bereits geöffnet ist ... 21

Speichern einer Arbeitsmappe, ohne den Benutzer zu fragen ... 22

Ändern der Standardanzahl von Arbeitsblättern in einer neuen Arbeitsmappe ... 22
Kapitel 5: Arrays ... 23
Examples ... 23
Felder füllen (Werte hinzufügen) ... 23
Direkt ... 23
Verwenden der Array () - Funktion ... 23
Aus Reichweite ... 23
2D mit Auswerten () ... 24
Verwenden der Funktion Split () ... 24
Dynamische Arrays (Größenanpassung von Arrays und dynamisches Handling) ... 24
Gezackte Arrays (Arrays von Arrays) ... 24
Prüfen Sie, ob das Array initialisiert ist (ob es Elemente enthält oder nicht) ... 25
Dynamische Arrays [Array-Deklaration, Größenänderung] ... 25
Kapitel 6: automatischer Filter ; Verwendungen und Best Practices ... 26
Einführung ... 26
Bemerkungen ... 26
Examples ... 26
Smartfilter! ... 26
Kapitel 7: Bedingte Anweisungen ... 32
Examples ... 32
Die If-Anweisung ... 32
Kapitel 8: Bedingte Formatierung mit VBA ... 34
Bemerkungen ... 34
Examples ... 34
FormatConditions.Add ... 34
Syntax: ... 34
Parameter: ... 34
XlFormatConditionType-Enumeration: ... 34
Formatierung nach Zellenwert: ... 35
Betreiber: ... 35
Die Formatierung nach Text enthält: ... 36

Betreiber: 36
Formatierung nach Zeitraum 36
Betreiber: 36
Bedingtes Format entfernen 37
Entfernen Sie alle bedingten Formate im Bereich: 37
Entfernen Sie alle bedingten Formate im Arbeitsblatt: 37
FormatConditions.AddUniqueValues 37
Doppelte Werte hervorheben 37
Einzigartige Werte hervorheben 37
FormatConditions.AddTop10 37
Top 5-Werte hervorheben 37
FormatConditions.AddAboveAverage 38
Betreiber: 38
FormatConditions.AddIconSetCondition 38
IconSet: 39
Art: 40
Operator: 41
Wert: 41
Kapitel 9: Benannte Bereiche 42
Einführung 42
Examples 42
Definieren Sie einen benannten Bereich 42
Benannte Bereiche in VBA verwenden 42
Verwalten Sie benannte Bereiche mit dem Namensmanager 43
Benannte Range Arrays 45
Kapitel 10: Benutzerdefinierte Funktionen (UDFs) 47
Syntax 47
Bemerkungen 47
Examples 47
UDF - Hallo Welt 48
Erlauben Sie vollständige Spaltenverweise ohne Strafe 49

Einzelne Werte in Bereich zählen ... 50
Kapitel 11: Bereiche und Zellen ... 52
Syntax ... 52
Bemerkungen ... 52
Examples ... 52
Erstellen eines Bereichs ... 52
Möglichkeiten, sich auf eine einzelne Zelle zu beziehen ... 54
Speichern eines Verweises auf eine Zelle in einer Variablen ... 55
Offset-Eigenschaft ... 55
Transponieren von Bereichen (horizontal in vertikal und umgekehrt) ... 55
Kapitel 12: Bindung ... 57
Examples ... 57
Early Binding vs. Late Binding ... 57
Kapitel 13: CustomDocumentProperties in der Praxis ... 59
Einführung ... 59
Examples ... 59
Neue Rechnungsnummern organisieren ... 59
Kapitel 14: Dateisystemobjekt ... 62
Examples ... 62
Datei, Ordner, Laufwerk ist vorhanden ... 62
Datei existiert: ... 62
Ordner existiert: ... 62
Laufwerk existiert: ... 62
Grundlegende Dateivorgänge ... 62
Kopieren: ... 62
Bewegung: ... 63
Löschen: ... 63
Grundlegende Ordnervorgänge ... 63
Erstellen: ... 63
Kopieren: ... 63
Bewegung: ... 63

Löschen: 64
Andere Operationen 64
Dateiname abrufen: 64
Basisnamen erhalten: 64
Erweiterungsname abrufen: 64
Laufwerksname abrufen: 65
Kapitel 15: Debugging und Fehlerbehebung 66
Syntax 66
Examples 66
Debug.Print 66
Halt 66
Sofortiges Fenster 66
Verwenden Sie den Timer, um Engpässe in der Leistung zu finden 67
Einen Haltepunkt zu Ihrem Code hinzufügen 68
Debugger-Fenster "Locals" 68
Kapitel 16: Diagramme und Diagramme 71
Examples 71
Erstellen eines Diagramms mit Bereichen und einem festen Namen 71
Erstellen eines leeren Diagramms 72
Erstellen Sie ein Diagramm, indem Sie die SERIES-Formel ändern 74
Charts in einem Raster anordnen 76
Kapitel 17: Doppelte Werte in einem Bereich suchen 80
Einführung 80
Examples 80
Duplikate in einem Bereich finden 80
Kapitel 18: Durchlaufen Sie alle Arbeitsblätter in der aktiven Arbeitsmappe 82
Examples 82
Rufen Sie alle Arbeitsblattnamen in Active Workbook ab 82
Alle Blätter in allen Dateien in einem Ordner durchlaufen 82
Kapitel 19: Erstellen eines Dropdown-Menüs im aktiven Arbeitsblatt mit einem Kombinationsf 84
Einführung 84

Examples ... 84
Jimi Hendrix Menü ... 84
Beispiel 2: Optionen nicht enthalten ... 85
Kapitel 20: Excel-VBA-Optimierung ... 88
Einführung ... 88
Bemerkungen ... 88
Examples ... 88
Arbeitsblattaktualisierung deaktivieren ... 88
Ausführungszeit prüfen ... 88
Verwendung mit Blöcken ... 89
Zeilenlöschung - Leistung ... 90
Deaktivieren aller Excel-Funktionen Vor dem Ausführen großer Makros ... 91
Optimierung der Fehlersuche durch erweitertes Debugging ... 92
Kapitel 21: Häufige Fehler ... 95
Examples ... 95
Qualifizierende Referenzen ... 95
Zeilen oder Spalten in einer Schleife löschen ... 96
ActiveWorkbook vs. ThisWorkbook ... 96
Schnittstelle für ein Dokument vs. mehrere Dokumentschnittstellen ... 97
Kapitel 22: Methoden zum Suchen der zuletzt verwendeten Zeile oder Spalte in einem Arbeits 100
Bemerkungen ... 100
Examples ... 100
Suchen Sie die letzte nicht leere Zelle in einer Spalte ... 100
Letzte Zeile mit benanntem Bereich suchen ... 101
Holen Sie sich die Zeile der letzten Zelle in einem Bereich ... 101
Suchen Sie die letzte nicht leere Spalte im Arbeitsblatt ... 102
Letzte Zelle in Range.CurrentRegion ... 102
Suchen Sie die letzte nicht leere Zeile im Arbeitsblatt ... 103
Finden Sie die letzte nicht leere Zelle in einer Reihe ... 103
Letzte nicht leere Zelle im Arbeitsblatt suchen - Leistung (Array) ... 104
Kapitel 23: Pivot-Tabellen ... 107
Bemerkungen ... 107

Examples 107
Erstellen einer Pivot-Tabelle 107
Pivot-Tabellenbereiche 109
Felder zu einer Pivot-Tabelle hinzufügen 109
Formatieren der Pivot-Tabellendaten 109
Kapitel 24: PowerPoint-Integration über VBA 111
Bemerkungen 111
Examples 111
Die Grundlagen: Starten von PowerPoint über VBA 111
Kapitel 25: SQL in Excel VBA - Best Practices 113
Examples 113
Wie verwende ich ADODB.Connection in VBA? 113
Bedarf: 113
Variablen deklarieren 113
Verbindung herstellen 113
ein. mit Windows-Authentifizierung 113
b. mit der SQL Server-Authentifizierung 114
Führen Sie den SQL-Befehl aus 114
Daten aus Datensatz lesen 114
Verbindung schließen 114
Wie benutze ich es? 114
Ergebnis 115
Kapitel 26: Tipps und Tricks zu Excel VBA 116
Bemerkungen 116
Examples 116
Verwenden von xlVeryHidden Sheets 116
Arbeitsblatt .Name, .Index oder .CodeName 117
Verwenden von Zeichenfolgen mit Trennzeichen anstelle von dynamischen Arrays 119
Doppelklicken Sie auf Ereignis für Excel-Shapes 120
Dateidialog öffnen - Mehrere Dateien 120
Kapitel 27: VBA-Best Practices 122

Bemerkungen ... 122
Examples ... 122
Verwenden Sie IMMER "Option Explicit" ... 122
Arbeit mit Arrays, nicht mit Ranges ... 125
Verwenden Sie, falls verfügbar, VB-Konstanten ... 125
Benennen Sie beschreibende Variablen ... 126
Fehlerbehandlung ... 127
On Error GoTo 0 ... 127
On Error Resume Next ... 128
On Error GoTo <Zeile> ... 128
Dokumentieren Sie Ihre Arbeit ... 129
Eigenschaften während der Makroausführung ausschalten ... 130
Vermeiden Sie die Verwendung von ActiveCell oder ActiveSheet in Excel ... 132
Nimm niemals das Arbeitsblatt an ... 133
Vermeiden Sie SELECT oder ACTIVATE ... 133
Definieren und setzen Sie immer Verweise auf alle Arbeitsmappen und Arbeitsblätter ... 135
Das WorksheetFunction-Objekt wird schneller als ein UDF-Äquivalent ausgeführt ... 136
Vermeiden Sie, die Namen von Eigenschaften oder Methoden als Variablen zu verwenden ... 137
Kapitel 28: VBA-Sicherheit ... 139
Examples ... 139
Passwort Schützen Sie Ihre VBA ... 139
Kapitel 29: Verwenden Sie ein Arbeitsblattobjekt und kein Blattobjekt ... 140
Einführung ... 140
Examples ... 140
Drucken Sie den Namen des ersten Objekts ... 140
Kapitel 30: Wie man ein Makro aufnimmt ... 141
Examples ... 141
Wie man ein Makro aufnimmt ... 141
Kapitel 31: Zusammengeführte Zellen / Bereiche ... 144
Examples ... 144
Überlegen Sie, bevor Sie zusammengefügte Zellen / Bereiche verwenden ... 144

Wo befinden sich die Daten in einem zusammengeführten Bereich? 144

Credits 145

Über

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

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

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

Kapitel 1: Erste Schritte mit Excel-VBA

Bemerkungen

Microsoft Excel enthält eine umfassende Makroprogrammiersprache namens VBA. Diese Programmiersprache bietet Ihnen mindestens drei zusätzliche Ressourcen:

1. Excel automatisch mit Makros vom Code entfernen. In der Regel kann alles, was der Benutzer durch Manipulieren von Excel über die Benutzeroberfläche tun kann, durch Schreiben von Code in Excel VBA erledigt werden.
2. Erstellen Sie neue benutzerdefinierte Arbeitsblattfunktionen.
3. Interagieren Sie Excel mit anderen Anwendungen wie Microsoft Word, PowerPoint, Internet Explorer, Notepad usw.

VBA steht für Visual Basic für Applikationen. Es ist eine benutzerdefinierte Version der alten Visual Basic-Programmiersprache, die seit Mitte der 1990er Jahre die Makros von Microsoft Excel unterstützt.

WICHTIG
Bitte stellen Sie sicher, dass alle Beispiele oder Themen, die mit dem Tag excel-vba erstellt wurden, **spezifisch** und **für** die Verwendung von VBA mit Microsoft Excel **relevant** sind. Alle vorgeschlagenen Themen oder Beispiele, die für die VBA-Sprache generisch sind, sollten abgelehnt werden, um Doppelarbeit zu vermeiden.

- On-Topic-Beispiele:

  ✓ *Erstellen und Interagieren mit Arbeitsblattobjekten*
  ✓ *Die* `WorksheetFunction` *Klasse und die entsprechenden Methoden*
  ✓ *Verwenden Sie die* `xlDirection` *Enumeration zum Navigieren in einem Bereich*

- Off-Topic-Beispiele:

  ✗ *So erstellen Sie eine 'für jede' Schleife*
  ✗ `MsgBox` *Klasse und wie eine Nachricht anzuzeigen* ,
  ✗ *Verwenden von WinAPI in VBA*

Versionen

VB

Ausführung	Veröffentlichungsdatum
VB6	1998-10-01

Ausführung	Veröffentlichungsdatum
VB7	2001-06-06
WIN32	1998-10-01
WIN64	2001-06-06
MAC	1998-10-01

Excel

Ausführung	Veröffentlichungsdatum
16	2016-01-01
fünfzehn	2013-01-01
14	2010-01-01
12	2007-01-01
11	2003-01-01
10	2001-01-01
9	1999-01-01
8	1997-01-01
7	1995-01-01
5	1993-01-01
2	1987-01-01

Examples

Variablen deklarieren

Um Variablen in VBA explizit zu deklarieren, verwenden Sie die `Dim` Anweisung, gefolgt von Variablenname und Typ. Wenn eine Variable ohne Deklaration verwendet wird oder kein Typ angegeben wird, wird der Typ `Variant` zugewiesen.

Verwenden Sie die `Option Explicit` Anweisung in der ersten Zeile eines Moduls, um die Deklaration aller Variablen vor der Verwendung zu erzwingen (siehe immer Verwenden Sie "Option Explicit").

Die `Option Explicit` immer zu verwenden ist sehr zu empfehlen, da Tippfehler / Rechtschreibfehler vermieden werden und sichergestellt wird, dass Variablen / Objekte ihren beabsichtigten Typ beibehalten.

```
Option Explicit

Sub Example()
    Dim a As Integer
    a = 2
    Debug.Print a
    'Outputs: 2

    Dim b As Long
    b = a + 2
    Debug.Print b
    'Outputs: 4

    Dim c As String
    c = "Hello, world!"
    Debug.Print c
    'Outputs: Hello, world!
End Sub
```

Mehrere Variablen können in einer einzigen Zeile mit Kommas als Trennzeichen **deklariert werden. Jeder Typ muss jedoch einzeln deklariert** werden. Andernfalls wird der `Variant` Typ verwendet.

```
Dim Str As String, IntOne, IntTwo As Integer, Lng As Long
Debug.Print TypeName(Str)    'Output: String
Debug.Print TypeName(IntOne) 'Output: Variant <--- !!!
Debug.Print TypeName(IntTwo) 'Output: Integer
Debug.Print TypeName(Lng)    'Output: Long
```

Variablen können auch mit Datentypzeichen-Suffixen ($% &! # @) Deklariert werden, von deren Verwendung wird jedoch zunehmend abgeraten.

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

Andere Arten, Variablen zu deklarieren, sind:

- `Static` wie: `Static CounterVariable as Integer`

  Wenn Sie die Static-Anweisung anstelle einer Dim-Anweisung verwenden, behält die deklarierte Variable ihren Wert zwischen den Aufrufen bei.

- `Public` wie: `Public CounterVariable as Integer`

  Öffentliche Variablen können in beliebigen Prozeduren im Projekt verwendet werden.

Wenn eine öffentliche Variable in einem Standardmodul oder einem Klassenmodul deklariert ist, kann sie auch in Projekten verwendet werden, die auf das Projekt verweisen, in dem die öffentliche Variable deklariert ist.

- `Private` wie: `Private CounterVariable as Integer`

  Private Variablen können nur von Prozeduren im selben Modul verwendet werden.

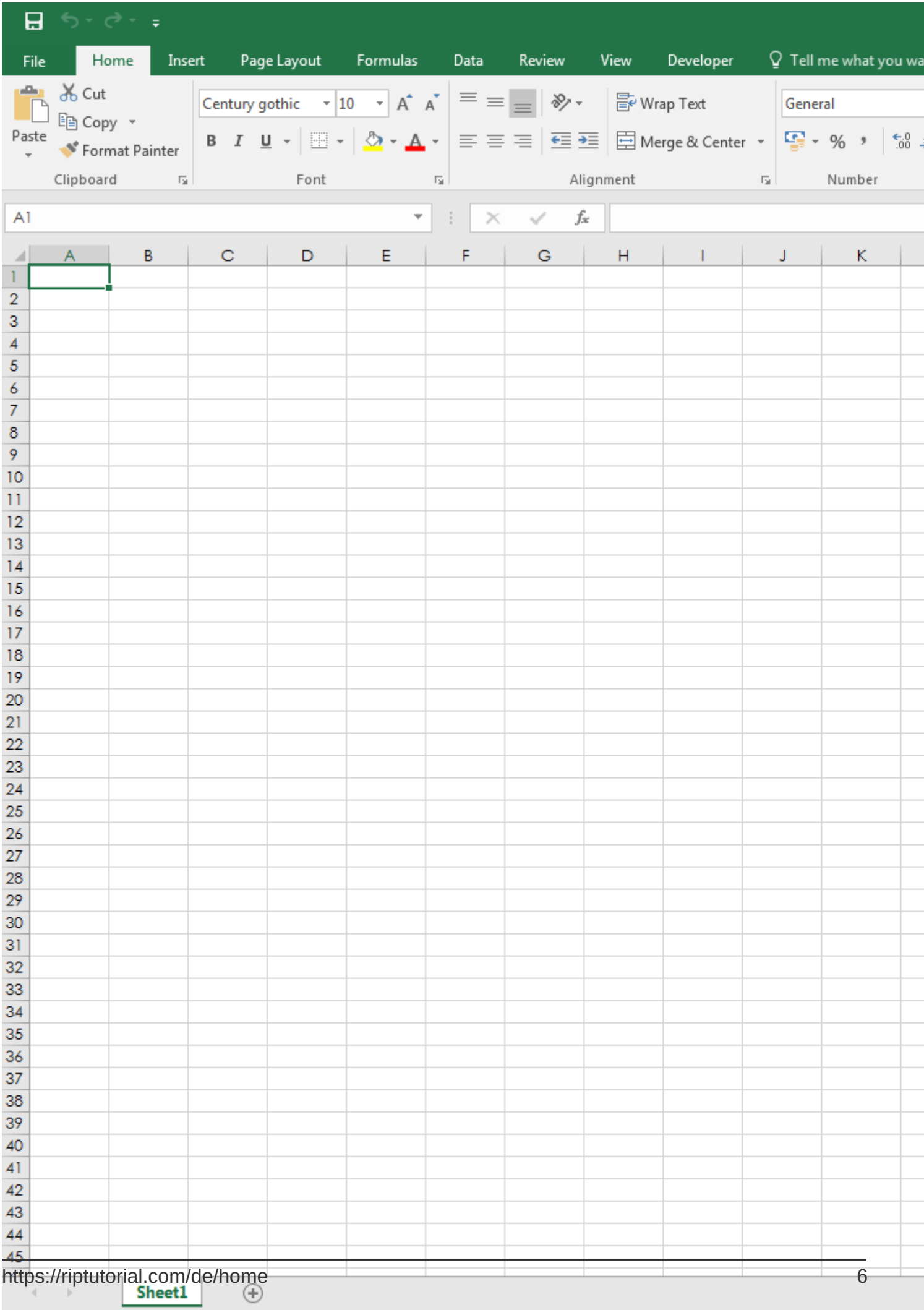
Quelle und weitere Informationen:

MSDN-deklarierende Variablen

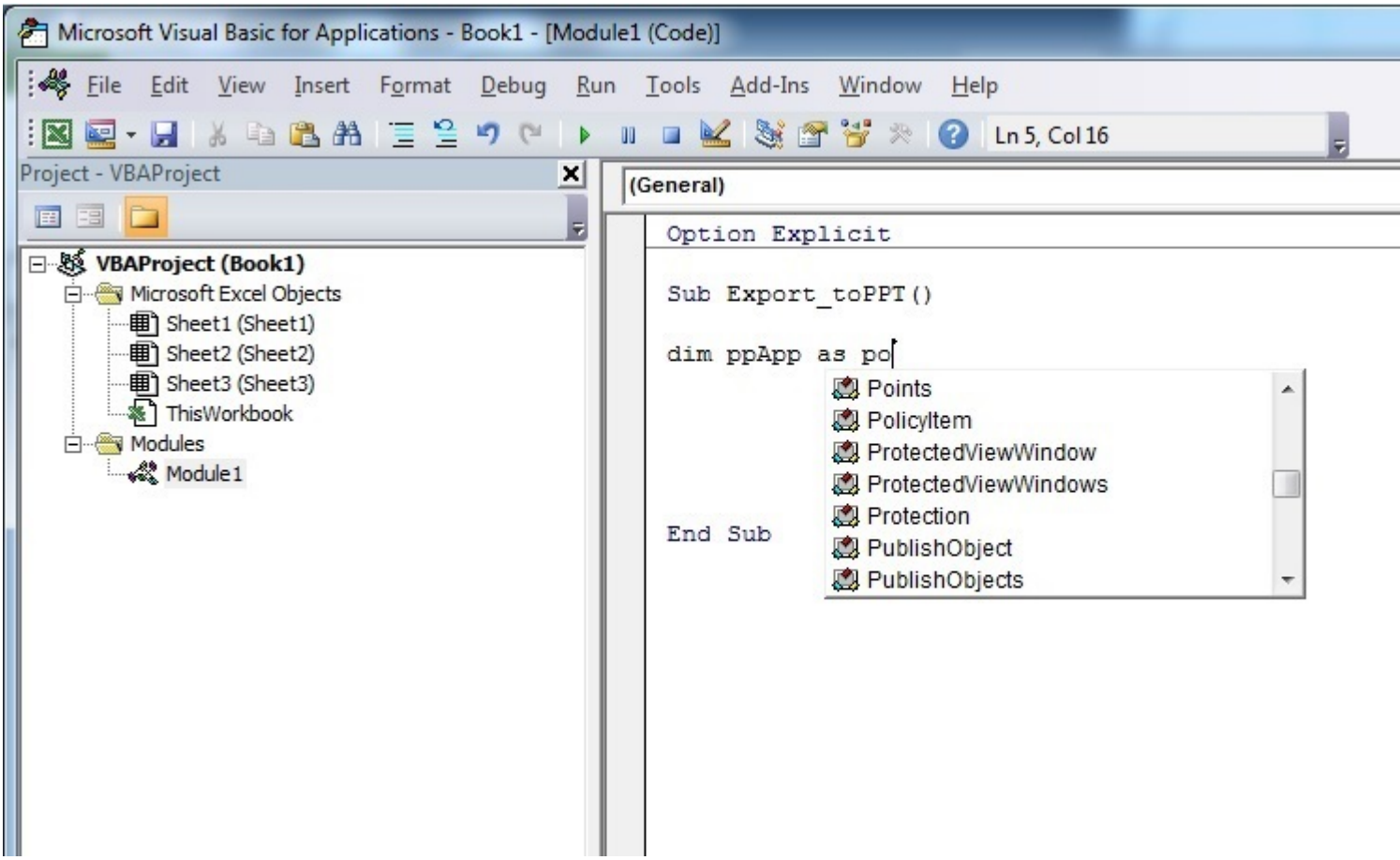
Zeichen eingeben (Visual Basic)

Öffnen des Visual Basic Editors (VBE)

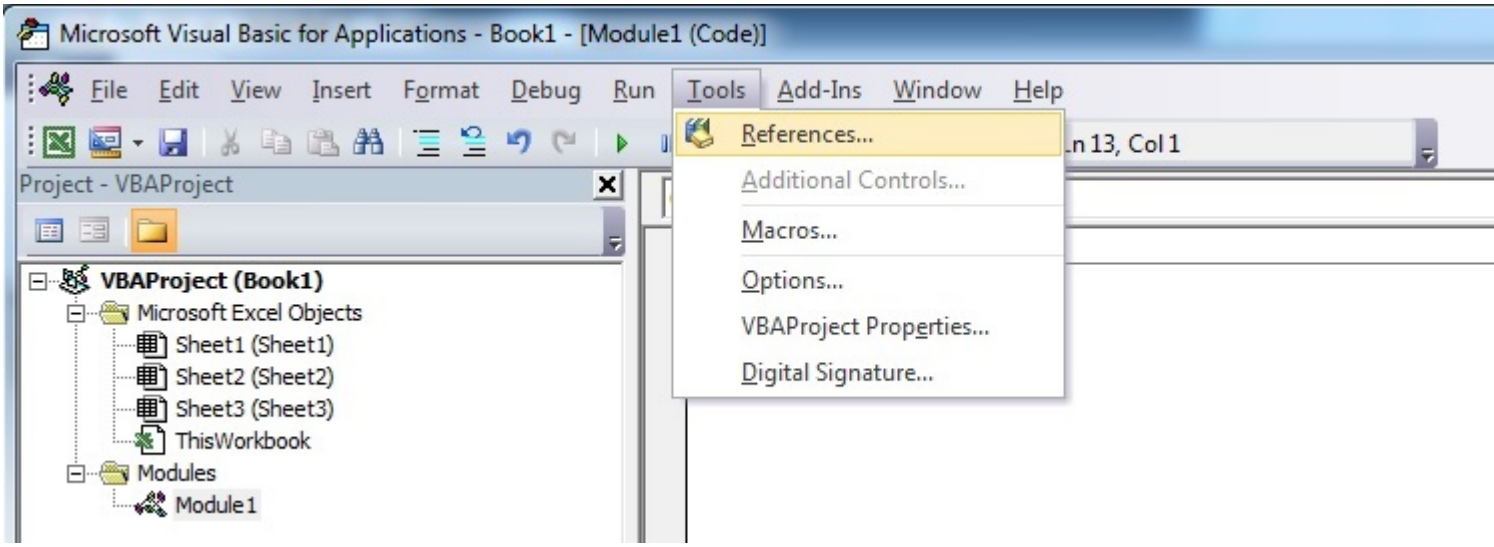
Schritt 1: Öffnen Sie eine Arbeitsmappe


File
Home
Insert
Page Layout
Formulas
Data
Review
View
Developer
Tell me what you wa
Cut
Copy
Format Painter
Paste
Century gothic
10
Wrap Text
General
Merge & Center
Clipboard
Font
Alignment
Number
A1
Sheet1

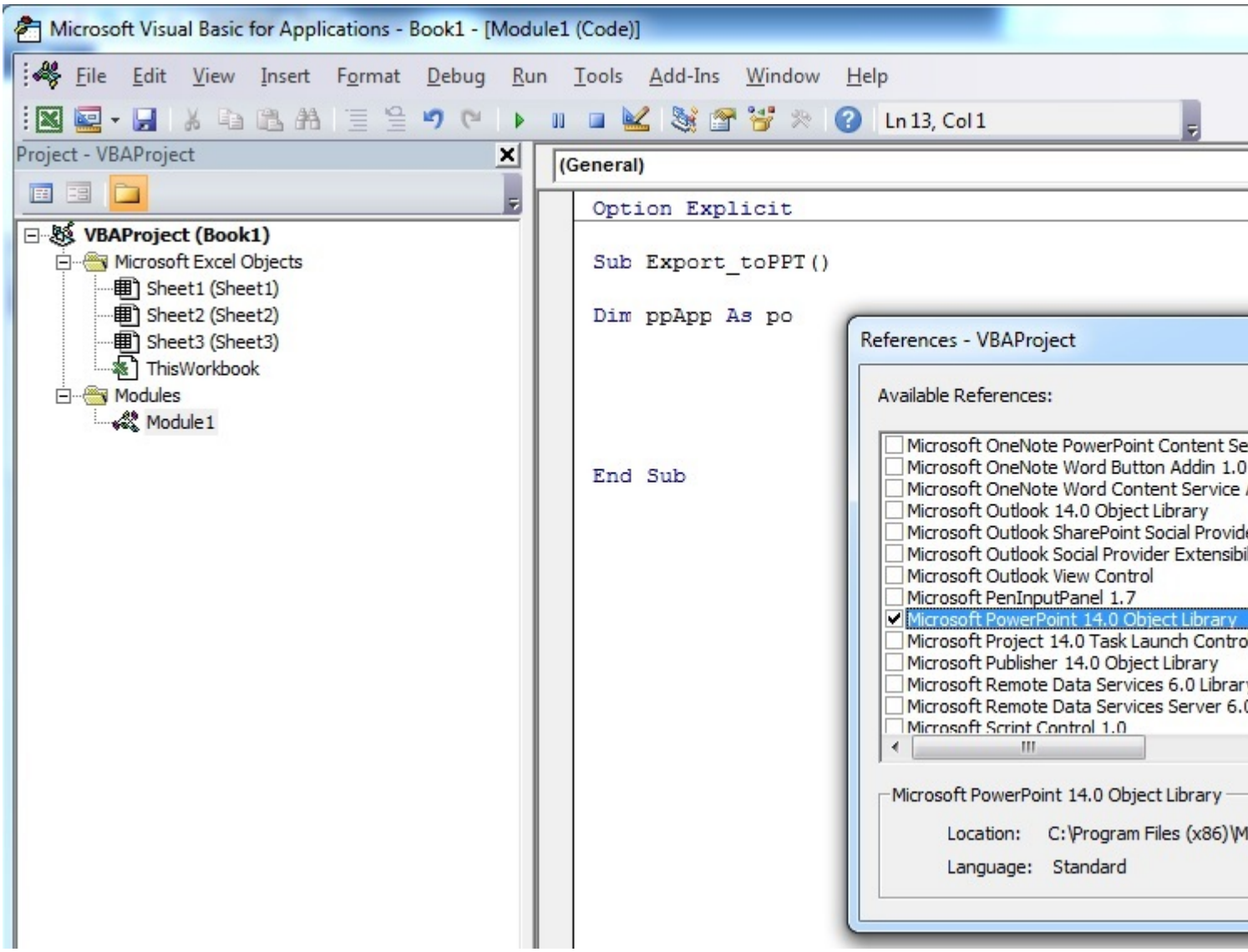
Bibliothek zum vorhandenen VB-Projekt hinzufügen. Wie zu sehen ist, ist derzeit die PowerPoint-Objektbibliothek nicht verfügbar.



Schritt 1 : Wählen Sie Menü ***Extras -> Referenzen…***

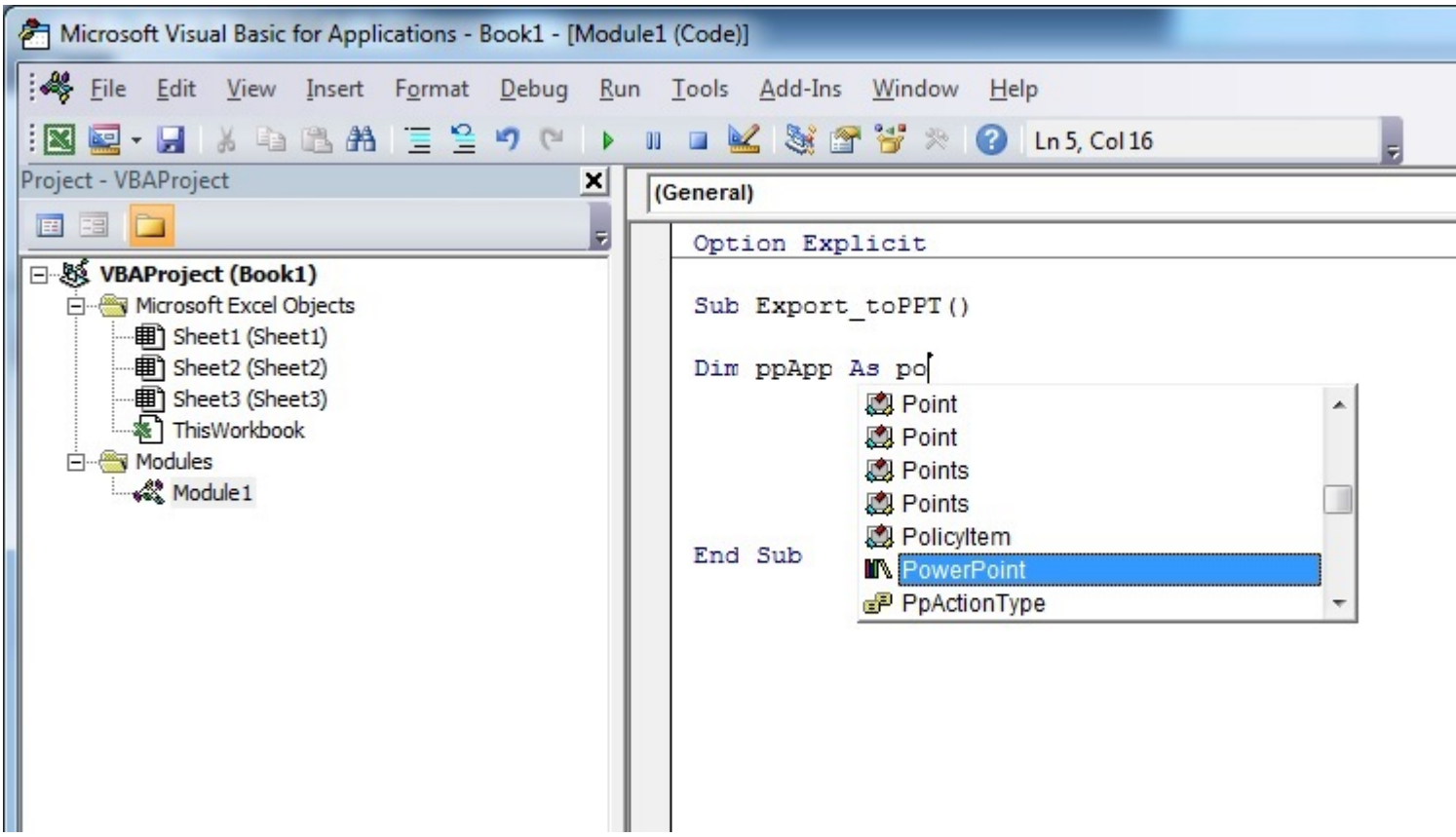


Schritt 2 : Wählen Sie die Referenz aus, die Sie hinzufügen möchten. In diesem Beispiel scrollen wir nach unten, um nach " ***Microsoft PowerPoint 14.0-Objektbibliothek*** " zu suchen, und klicken Sie auf " **OK** ".

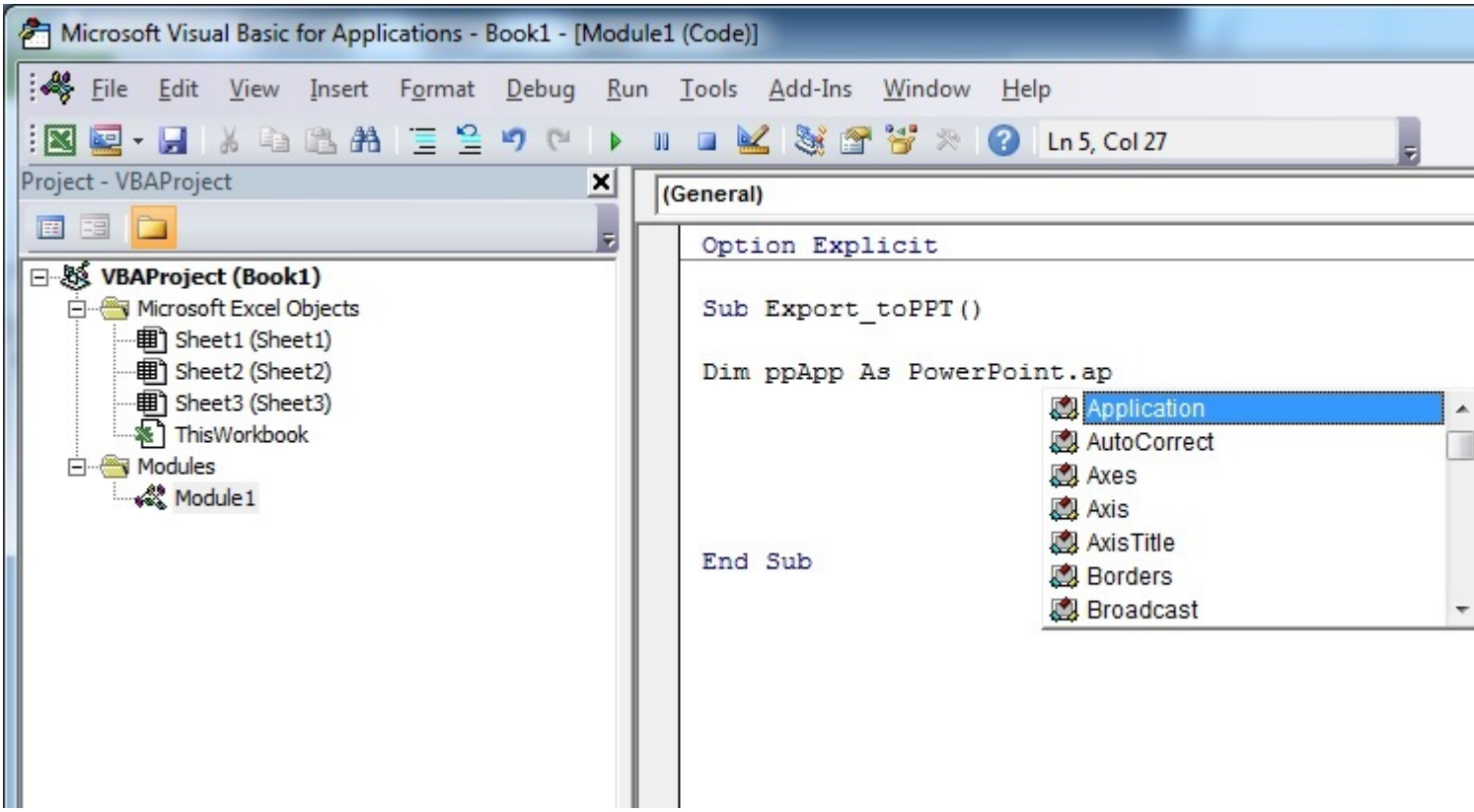


Hinweis: PowerPoint 14.0 bedeutet, dass die Office 2010-Version auf dem PC installiert ist.

Schritt 3 : Wenn Sie im VB-Editor die Tastenkombination **Strg + Leertaste gleichzeitig** drücken, erhalten Sie die Autocomplete-Option von PowerPoint.

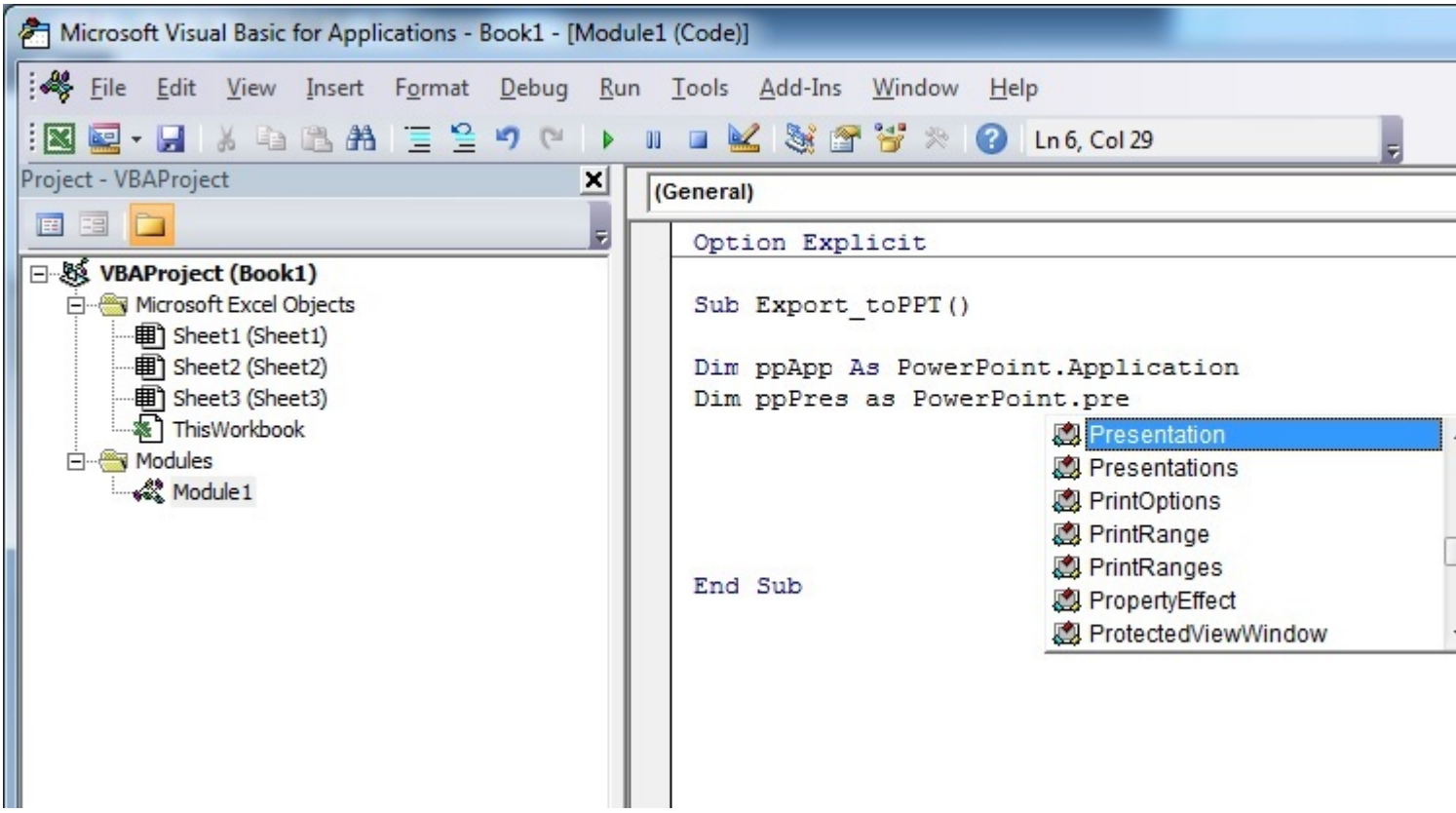


Nach Auswahl von `PowerPoint` und Drücken von `.` wird ein weiteres Menü mit allen Objektoptionen angezeigt, die sich auf die PowerPoint-Objektbibliothek beziehen. Dieses Beispiel zeigt, wie Sie das Objekt `Application` PowerPoint auswählen.

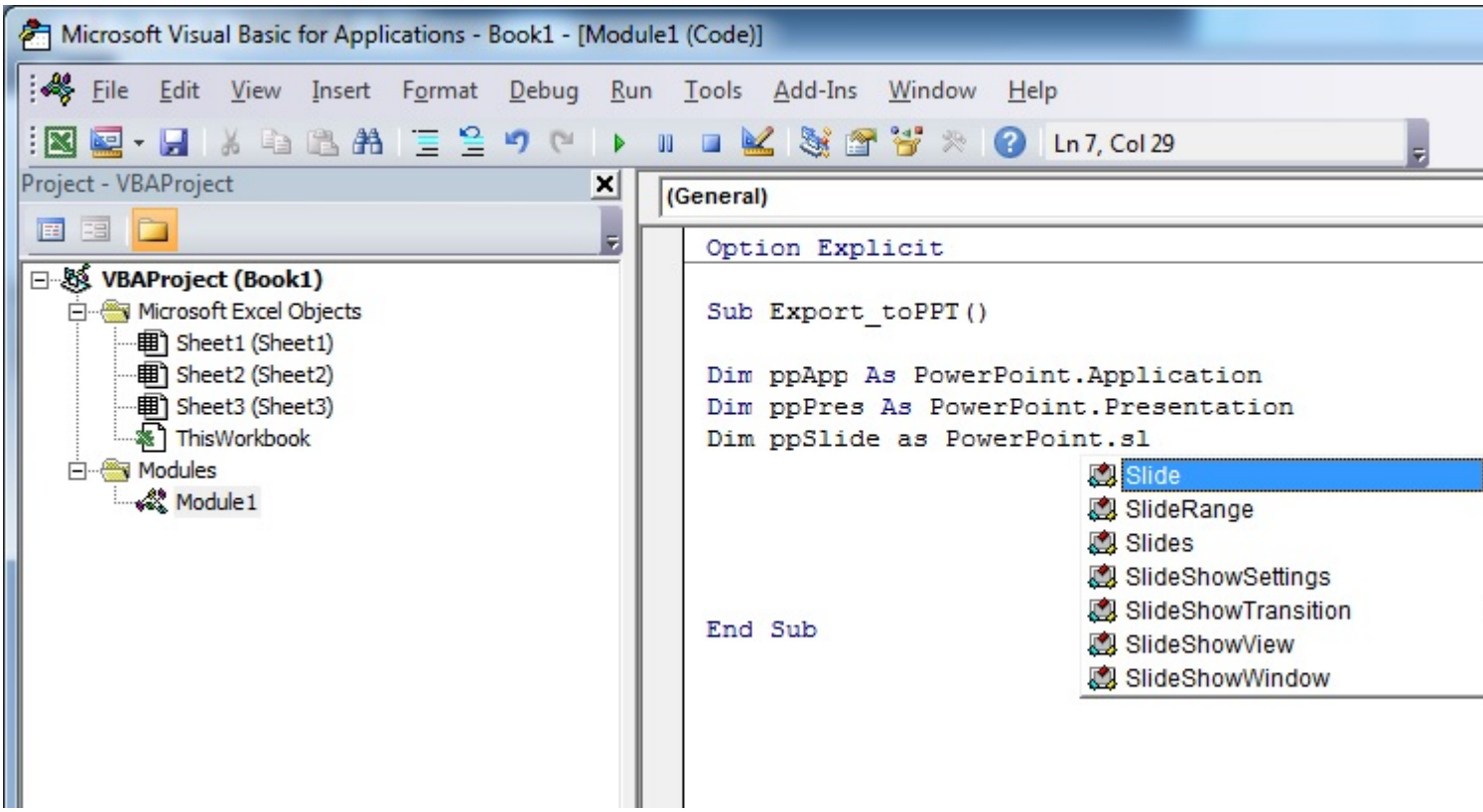


Schritt 4 : Jetzt kann der Benutzer mithilfe der PowerPoint-Objektbibliothek weitere Variablen deklarieren.

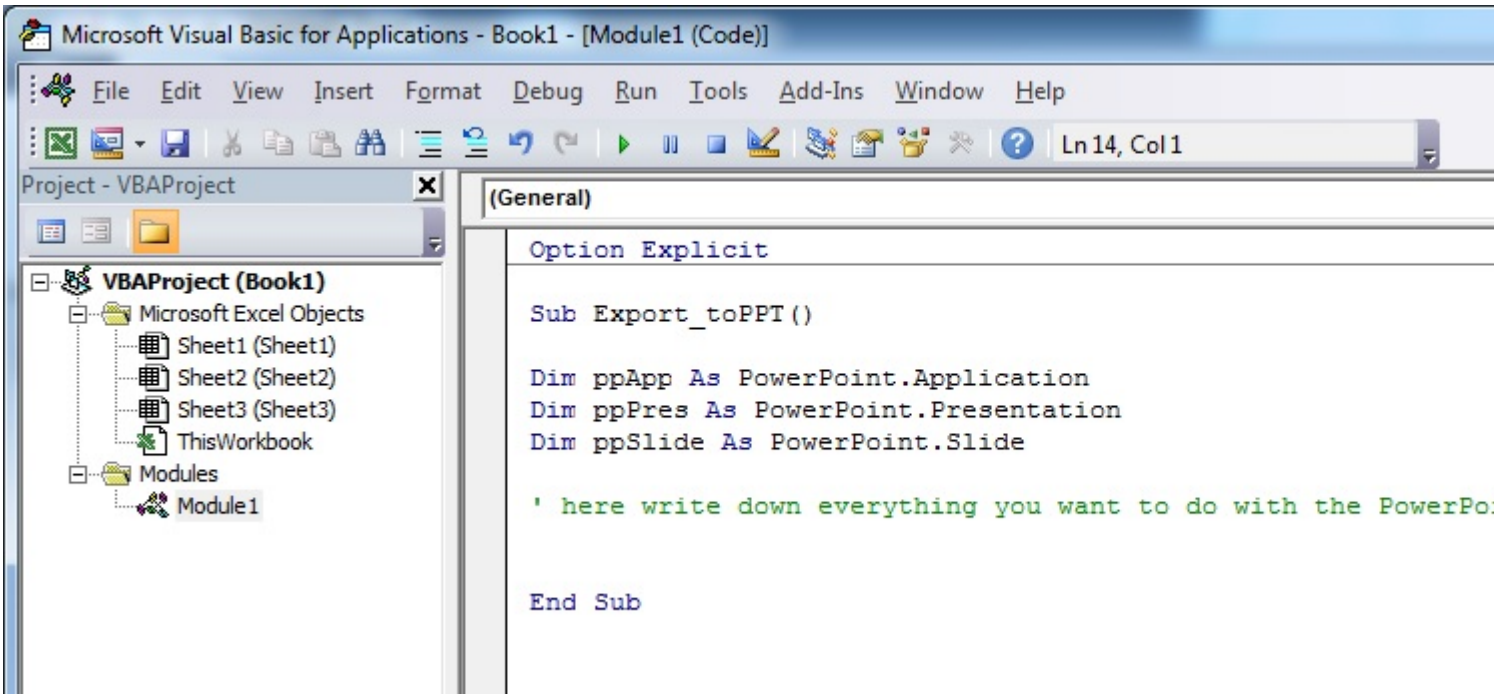
Deklarieren Sie eine Variable, die auf das `Presentation` Objekt der PowerPoint-Objektbibliothek verweist.



Deklarieren Sie eine andere Variable, die auf das `Slide` Objekt der PowerPoint-Objektbibliothek verweist.



Nun sieht der Variablendeklarationsabschnitt wie in der Abbildung unten aus, und der Benutzer kann diese Variablen in seinem Code verwenden.

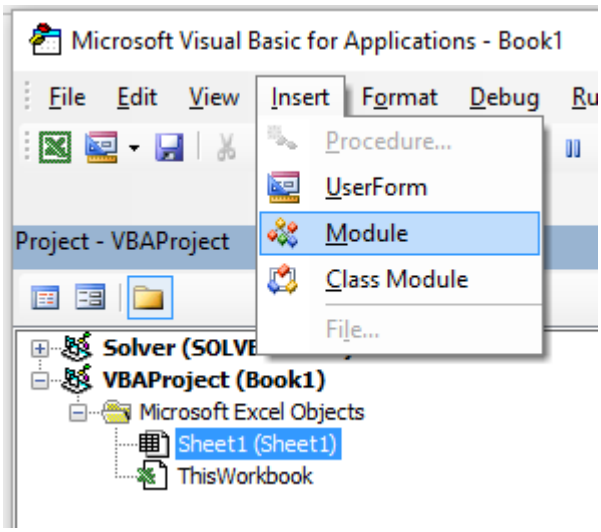


Codeversion dieses Tutorials:

```
Option Explicit

Sub Export_toPPT()

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

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


End Sub
```

Hallo Welt

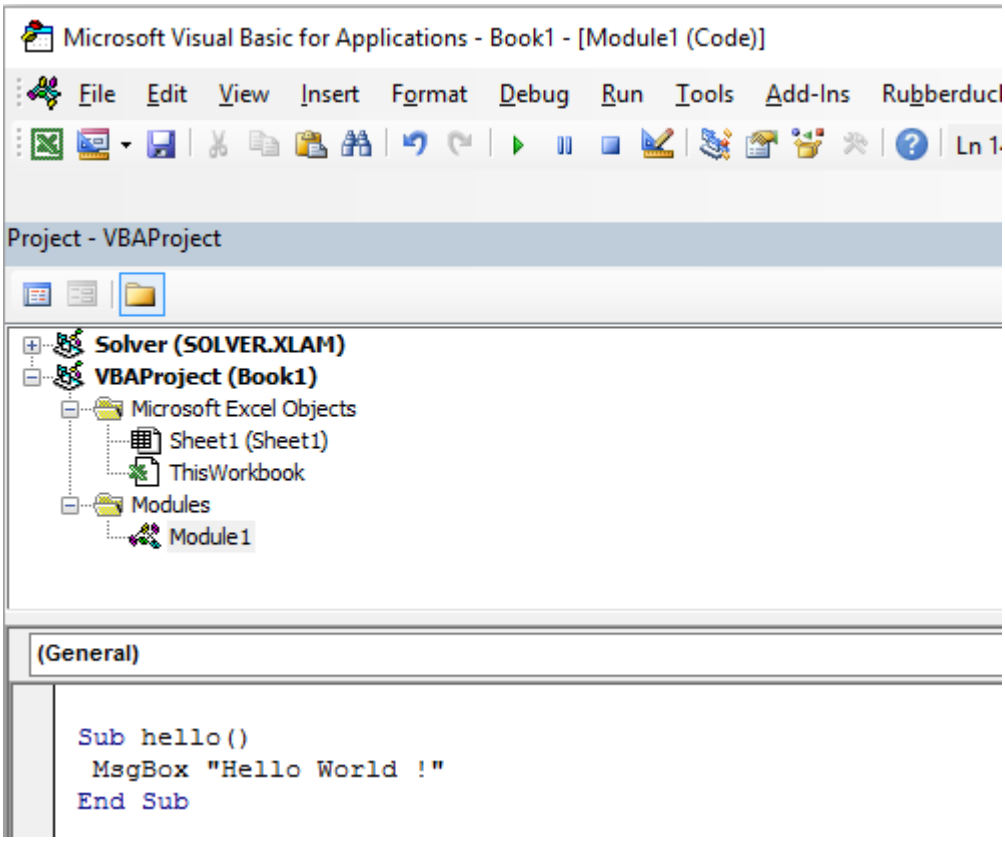
1. Öffnen Sie den Visual Basic-Editor (siehe Öffnen des Visual Basic-Editors).
2. Klicken Sie auf Einfügen -> Modul, um ein neues Modul hinzuzufügen:



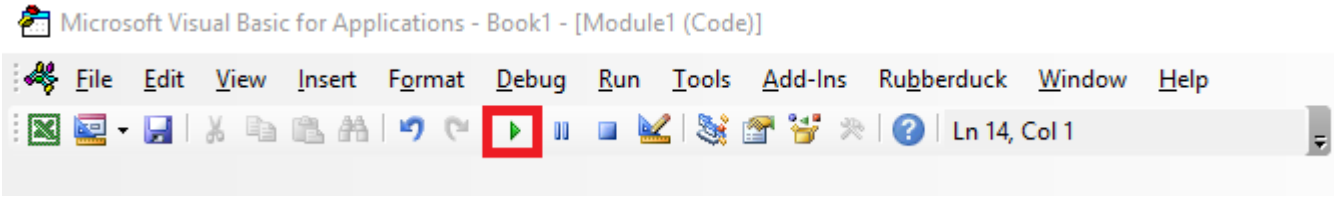
3. Kopieren Sie den folgenden Code, und fügen Sie ihn in das neue Modul ein:

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

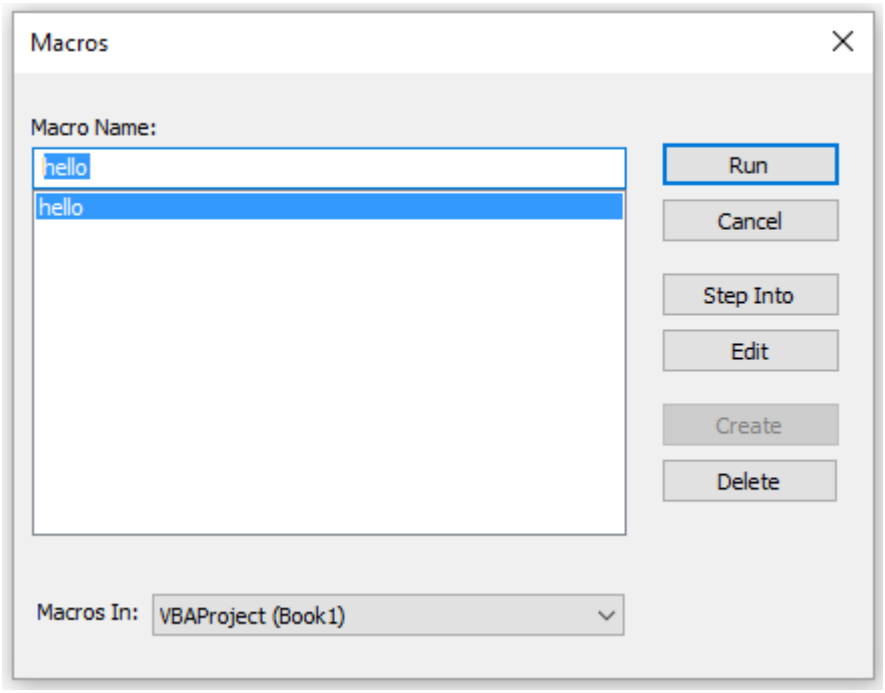
Erhalten :



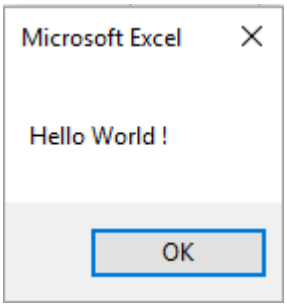
4. Klicken Sie auf den grünen "Play" -Pfeil (oder drücken Sie F5) in der Visual Basic-Symbolleiste, um das Programm auszuführen:



5. Wählen Sie das neu erstellte "Hallo" aus und klicken Sie auf " `Run` .



6. Fertig, Ihr sollte folgendes Fenster sehen:

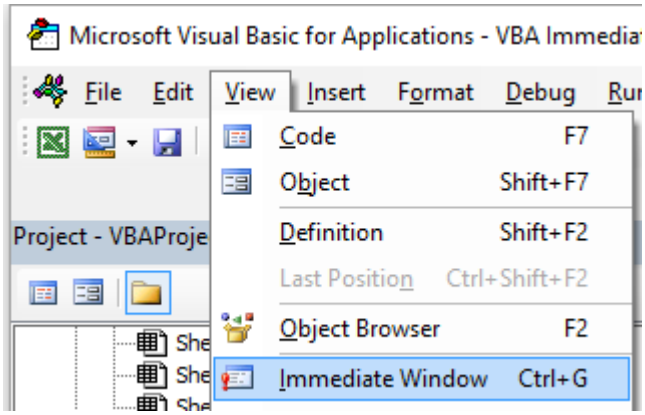


Erste Schritte mit dem Excel-Objektmodell

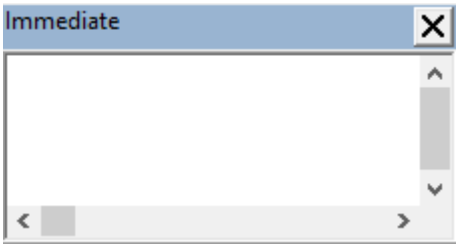
Dieses Beispiel soll eine sanfte Einführung in das Excel-Objektmodell **für Anfänger sein** .

---

1. Öffnen Sie den Visual Basic Editor (VBE)
2. Klicken Sie auf Ansicht -> `Direktfenster` , um das `Direktfenster` (oder `Strg` + `G`) zu öffnen:



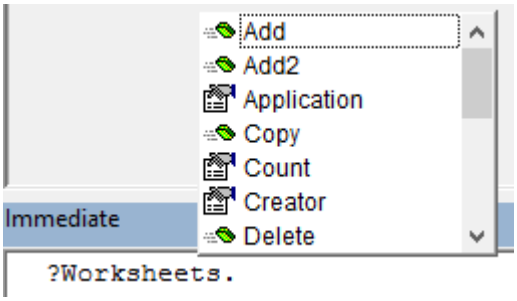
3. Das folgende Direktfenster sollte in VBE unten angezeigt werden:



In diesem Fenster können Sie den VBA-Code direkt testen. Beginnen wir also, geben Sie diese Konsole ein:

```
?Worksheets.
```

VBE hat Intellisense und dann sollte ein Tooltip wie in der folgenden Abbildung geöffnet werden:



Wählen Sie .Count in der Liste aus oder geben Sie direkt `.Cout` , um `.Cout` zu erhalten:

```
?Worksheets.Count
```

4. Drücken Sie dann die Eingabetaste. Der Ausdruck wird ausgewertet und es sollte 1 zurückgegeben werden. Dies gibt die Anzahl der Arbeitsblätter an, die aktuell in der Arbeitsmappe vorhanden sind. Das Fragezeichen (`?`) Ist ein Alias für Debug.Print.

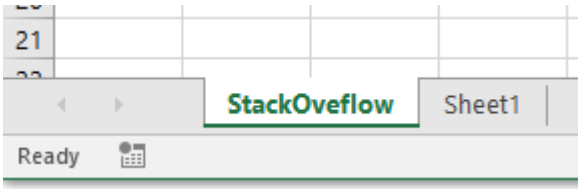
Arbeitsblätter sind ein **Objekt** und Count ist eine **Methode** . Excel verfügt über mehrere Objekte (`Workbook` , `Worksheet` , `Range` , `Chart` ...), und jedes enthält bestimmte Methoden und Eigenschaften. Die vollständige Liste der Objekte finden Sie in der Excel VBA-Referenz . Arbeitsblätter Objekt wird hier vorgestellt.

> Diese Excel VBA-Referenz sollte Ihre primäre Informationsquelle zum Excel-Objektmodell sein.

5. Versuchen wir nun einen anderen Ausdruck, Typ (ohne das Zeichen `?`):

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

6. Drücken Sie Enter. Dadurch sollte ein neues Arbeitsblatt mit dem Namen `StackOverflow.` :



Um diesen Ausdruck zu verstehen, müssen Sie die Funktion Hinzufügen in der oben genannten Excel-Referenz lesen. Sie werden folgendes finden:

```
Add:  Creates a new worksheet, chart, or macro sheet.
The new worksheet becomes the active sheet.
Return Value: An Object value that represents the new worksheet, chart,
 or macro sheet.
```

Die `Worksheets.Add()` erstellen also ein neues Arbeitsblatt und geben es zurück. Arbeitsblatt (**ohne s**) ist selbst ein Objekt, das in der Dokumentation zu finden ist, und `Name` gehört zu seinen **Eigenschaften** (siehe hier). Es ist definiert als:

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

Indem wir die verschiedenen Objektdefinitionen untersuchen, können wir diesen Code `Worksheets.Add().Name = "StackOveflow"` verstehen.

`Add()` erstellt und fügt ein neues Arbeitsblatt hinzu und gibt einen **Verweis** darauf zurück. Dann setzen wir seine Name- **Eigenschaft** auf "StackOverflow".

Lassen Sie uns formeller sein, Excel enthält mehrere Objekte. Diese Objekte können aus einer oder mehreren Sammlungen von Excel-Objekten derselben Klasse bestehen. `WorkSheets` ist der Fall für `WorkSheets` denen es sich um eine Auflistung von `Worksheet` . Jedes Objekt verfügt über einige Eigenschaften und Methoden, mit denen der Programmierer interagieren kann.

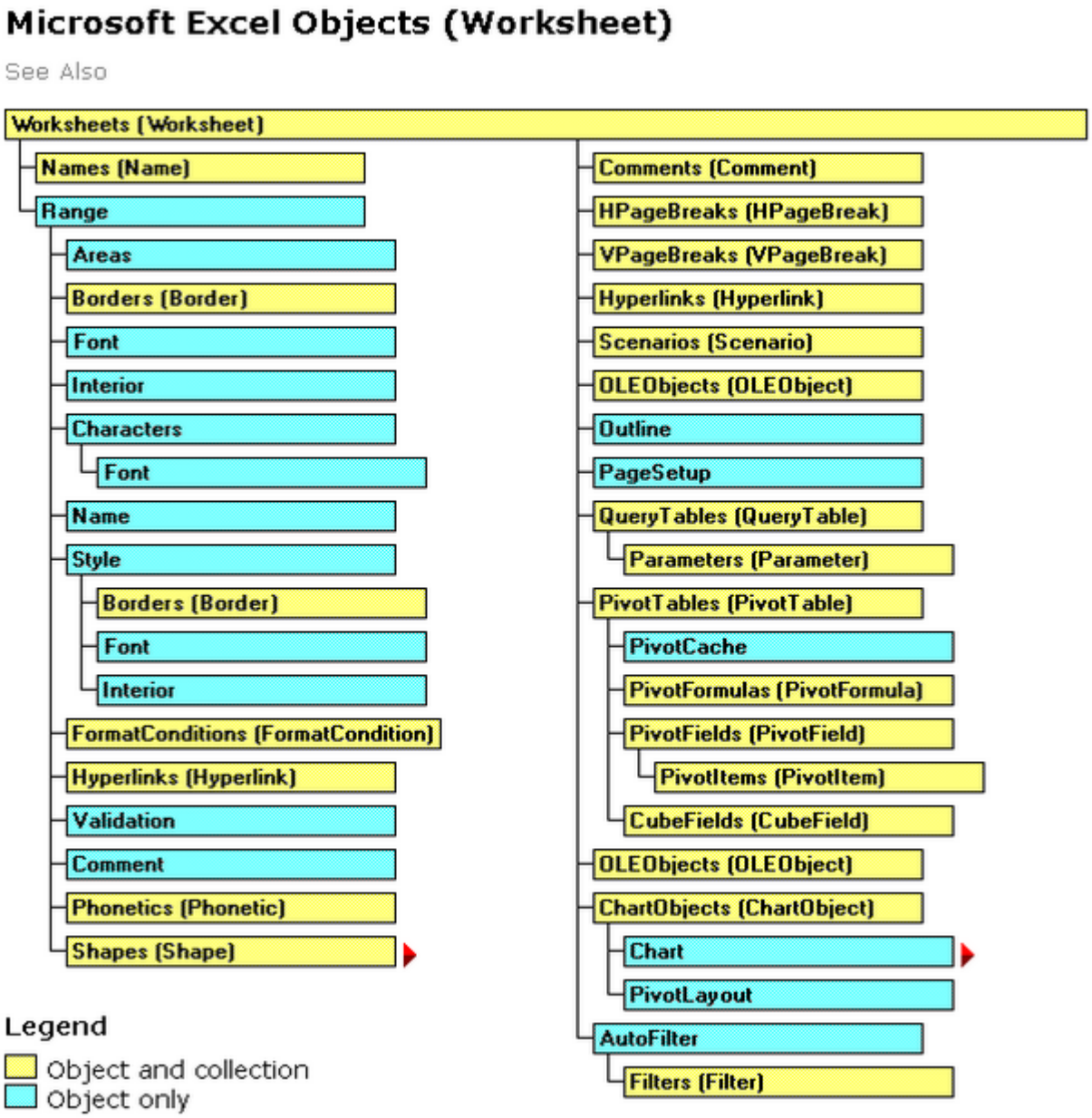
> Das Excel-Objektmodell bezieht sich auf die Excel- **Objekthierarchie**

An der Spitze aller Objekte befindet sich das `Application` , es repräsentiert die Excel-Instanz selbst. Die Programmierung in VBA erfordert ein gutes Verständnis dieser Hierarchie, da wir immer einen Verweis auf ein Objekt benötigen, um eine Methode aufrufen zu können oder eine Eigenschaft zu setzen / abzurufen.

Das (sehr vereinfachte) Excel-Objektmodell kann dargestellt werden als

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

Eine detailliertere Version für das Arbeitsblattobjekt (wie in Excel 2007) wird unten gezeigt.



Das vollständige Excel-Objektmodell finden Sie hier .

Schließlich können einige Objekte `events` (z. B. `Workbook.WindowActivate`) enthalten, die ebenfalls Teil des Excel-Objektmodells sind.

Erste Schritte mit Excel-VBA online lesen: https://riptutorial.com/de/excel-vba/topic/777/erste-schritte-mit-excel-vba

Kapitel 2: Anwendungsobjekt

Bemerkungen

Excel VBA verfügt über ein umfassendes *Objektmodell,* das Klassen und Objekte enthält, mit denen Sie einen beliebigen Teil der laufenden Excel-Anwendung bearbeiten können. Eines der am häufigsten verwendeten Objekte ist das **Application-** Objekt. Dies ist ein Catchall der obersten Ebene, das die aktuell ausgeführte Instanz von Excel darstellt. Fast alles, was nicht mit einer bestimmten Excel-Arbeitsmappe verbunden ist, befindet sich im **Application-** Objekt.

Das *Application-* Objekt verfügt als Top-Level-Objekt über hunderte von Eigenschaften, Methoden und Ereignissen, mit denen jeder Aspekt von Excel gesteuert werden kann.

Examples

Beispiel für ein einfaches Anwendungsobjekt: Minimieren Sie das Excel-Fenster

Dieser Code verwendet das **Application-** Objekt der obersten Ebene, um das Hauptfenster von Excel zu minimieren.

```
Sub MinimizeExcel()

    Application.WindowState = xlMinimized

End Sub
```

Beispiel eines einfachen Anwendungsobjekts: Anzeige der Excel- und VBE-Version

```
Sub DisplayExcelVersions()

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

End Sub
```

Die Verwendung der Application.Version-Eigenschaft ist hilfreich, um sicherzustellen, dass Code nur mit einer kompatiblen Version von Excel funktioniert.

Anwendungsobjekt online lesen: https://riptutorial.com/de/excel-vba/topic/5645/anwendungsobjekt

Kapitel 3: Arbeiten mit Excel-Tabellen in VBA

Einführung

Dieses Thema behandelt das Arbeiten mit Tabellen in VBA und setzt Kenntnisse in Excel-Tabellen voraus. In VBA bzw. Excel-Objektmodell werden Tabellen als ListObjects bezeichnet. Die am häufigsten verwendeten Eigenschaften eines ListObjects sind ListRow (s), ListColumn (s), DataBodyRange, Range und HeaderRowRange.

Examples

Instanziieren eines ListObjects

```
Dim lo as ListObject
Dim MyRange as Range

Set lo = Sheet1.ListObjects(1)

'or

Set lo = Sheet1.ListObjects("Table1")

'or

Set lo = MyRange.ListObject
```

Mit ListRows / ListColumns arbeiten

```
Dim lo as ListObject
Dim lr as ListRow
Dim lc as ListColumn

Set lr = lo.ListRows.Add
Set lr = lo.ListRows(5)

For Each lr in lo.ListRows
    lr.Range.ClearContents
    lr.Range(1, lo.ListColumns("Some Column").Index).Value = 8
Next

Set lc = lo.ListColumns.Add
Set lc = lo.ListColumns(4)
Set lc = lo.ListColumns("Header 3")

For Each lc in lo.ListColumns
    lc.DataBodyRange.ClearContents   'DataBodyRange excludes the header row
    lc.Range(1,1).Value = "New Header Name"    'Range includes the header row
Next
```

Konvertieren einer Excel-Tabelle in einen normalen Bereich

```
Dim lo as ListObject

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

Arbeiten mit Excel-Tabellen in VBA online lesen: https://riptutorial.com/de/excel-vba/topic/9753/arbeiten-mit-excel-tabellen-in-vba

Kapitel 4: Arbeitsmappen

Examples

Anwendungs-Arbeitsmappen

In vielen Excel-Anwendungen führt der VBA-Code Aktionen aus, die auf die Arbeitsmappe gerichtet sind, in der er enthalten ist. Sie speichern diese Arbeitsmappe mit der Erweiterung ".xlsm", und die VBA-Makros konzentrieren sich nur auf die Arbeitsblätter und die darin enthaltenen Daten. In manchen Fällen müssen Sie jedoch Daten aus anderen Arbeitsmappen kombinieren oder zusammenführen oder einige Ihrer Daten in eine separate Arbeitsmappe schreiben. Das Öffnen, Schließen, Speichern, Erstellen und Löschen anderer Arbeitsmappen ist für viele VBA-Anwendungen eine häufige Notwendigkeit.

Im VBA-Editor können Sie jederzeit alle Arbeitsmappen, die derzeit von dieser Instanz von Excel geöffnet werden, mit der `Workbooks` Eigenschaft des `Application` Objekts anzeigen und darauf zugreifen. Die MSDN-Dokumentation erläutert dies mit Referenzen.

Wann Verwenden von ActiveWorkbook und ThisWorkbook

Es ist eine bewährte Methode für VBA, immer anzugeben, auf welche Arbeitsmappe Ihr VBA-Code verweist. Wenn diese Angabe ausgelassen wird, geht VBA davon aus, dass der Code an die aktuell aktive Arbeitsmappe (`ActiveWorkbook`) gerichtet ist.

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

Wenn jedoch mehrere Arbeitsmappen gleichzeitig geöffnet sind, insbesondere wenn VBA-Code von einem Excel-Add-In `ActiveWorkbook` können Verweise auf das `ActiveWorkbook` verwirrt oder fehlgeleitet werden. Beispielsweise muss ein Add-In mit einer UDF, die die Tageszeit überprüft und mit einem Wert vergleicht, der in einem der Arbeitsblätter des Add-Ins (die normalerweise für den Benutzer nicht sichtbar sind) gespeichert ist, muss explizit angeben, um welche Arbeitsmappe es sich handelt referenziert werden. In unserem Beispiel hat unsere offene (und aktive) Arbeitsmappe eine Formel in der Zelle A1 `=EarlyOrLate()` und KEINE VBA, die für diese aktive Arbeitsmappe geschrieben wurde. In unserem Add-In haben wir folgende User Defined Function (UDF):

```
Public Function EarlyOrLate() As String
    If Hour(Now) > ThisWorkbook.Sheets("WatchTime").Range("A1") Then
        EarlyOrLate = "It's Late!"
    Else
        EarlyOrLate = "It's Early!"
    End If
End Function
```

Der Code für die UDF wird geschrieben und im installierten Excel-Add-In gespeichert. Es

verwendet Daten, die in einem Arbeitsblatt im Add-In namens "WatchTime" gespeichert sind. Wenn die UDF `ActiveWorkbook` anstelle von `ThisWorkbook` , kann sie niemals garantieren, welche Arbeitsmappe vorgesehen ist.

Öffnen einer (neuen) Arbeitsmappe, auch wenn sie bereits geöffnet ist

Wenn Sie auf eine bereits geöffnete Arbeitsmappe zugreifen möchten, ist das Abrufen der Zuweisung aus der `Workbooks` Sammlung unkompliziert:

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

Wenn Sie eine neue Arbeitsmappe erstellen möchten, verwenden Sie dann die `Workbooks` Sammlung Objekt `Add` einen neuen Eintrag.

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

Es kann vorkommen, dass Sie die Arbeitsmappe, die Sie benötigen, nicht bereits geöffnet haben oder nicht (oder nicht) oder nicht oder nicht vorhanden ist. Die Beispielfunktion zeigt, wie immer ein gültiges Arbeitsmappenobjekt zurückgegeben wird.

```
Option Explicit
Function GetWorkbook(ByVal wbFilename As String) As Workbook
    '--- returns a workbook object for the given filename, including checks
    '    for when the workbook is already open, exists but not open, or
    '    does not yet exist (and must be created)
    '    ***  wbFilename must be a fully specified pathname
    Dim folderFile As String
    Dim returnedWB As Workbook

    '--- check if the file exists in the directory location
    folderFile = File(wbFilename)
    If folderFile = "" Then
        '--- the workbook doesn't exist, so create it
        Dim pos1 As Integer
        Dim fileExt As String
        Dim fileFormatNum As Long
        '--- in order to save the workbook correctly, we need to infer which workbook
        '    type the user intended from the file extension
        pos1 = InStrRev(sFullName, ".", , vbTextCompare)
        fileExt = Right(sFullName, Len(sFullName) - pos1)
        Select Case fileExt
            Case "xlsx"
                fileFormatNum = 51
            Case "xlsm"
                fileFormatNum = 52
            Case "xls"
                fileFormatNum = 56
            Case "xlsb"
                fileFormatNum = 50
            Case Else
                Err.Raise vbObjectError + 1000, "GetWorkbook function", _
                        "The file type you've requested (file extension) is not recognized. "
& _
```

```
                        "Please use a known extension: xlsx, xlsm, xls, or xlsb."
        End Select
        Set returnedWB = Workbooks.Add
        Application.DisplayAlerts = False
        returnedWB.SaveAs filename:=wbFilename, FileFormat:=fileFormatNum
        Application.DisplayAlerts = True
        Set GetWorkbook = returnedWB
    Else
        '--- the workbook exists in the directory, so check to see if
        '    it's already open or not
        On Error Resume Next
        Set returnedWB = Workbooks(sFile)
        If returnedWB Is Nothing Then
            Set returnedWB = Workbooks.Open(sFullName)
        End If
    End If
End Function
```

Speichern einer Arbeitsmappe, ohne den Benutzer zu fragen

Beim Speichern neuer Daten in einer vorhandenen Arbeitsmappe mit VBA wird häufig eine Popup-Frage angezeigt, die darauf hinweist, dass die Datei bereits vorhanden ist.

Um diese Popup-Frage zu verhindern, müssen Sie diese Arten von Warnungen unterdrücken.

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

Ändern der Standardanzahl von Arbeitsblättern in einer neuen Arbeitsmappe

Die "werkseitige" Anzahl von Arbeitsblättern, die in einer neuen Excel-Arbeitsmappe erstellt werden, ist im Allgemeinen auf drei festgelegt. Ihr VBA-Code kann die Anzahl der Arbeitsblätter in einer neuen Arbeitsmappe explizit festlegen.

```
'--- save the current Excel global setting
With Application
    Dim oldSheetsCount As Integer
    oldSheetsCount = .SheetsInNewWorkbook
    Dim myNewWB As Workbook
    .SheetsInNewWorkbook = 1
    Set myNewWB = .Workbooks.Add
    '--- restore the previous setting
    .SheetsInNewWorkbook = oldsheetcount
End With
```

Arbeitsmappen online lesen: https://riptutorial.com/de/excel-vba/topic/2969/arbeitsmappen

Kapitel 5: Arrays

Examples

Felder füllen (Werte hinzufügen)

Es gibt mehrere Möglichkeiten, ein Array zu füllen.

Direkt

```
'one-dimensional
Dim arrayDirect1D(2) As String
arrayDirect(0) = "A"
arrayDirect(1) = "B"
arrayDirect(2) = "C"

'multi-dimensional (in this case 3D)
Dim arrayDirectMulti(1, 1, 2)
arrayDirectMulti(0, 0, 0) = "A"
arrayDirectMulti(0, 0, 1) = "B"
arrayDirectMulti(0, 0, 2) = "C"
arrayDirectMulti(0, 1, 0) = "D"
'...
```

Verwenden der Array () - Funktion

```
'one-dimensional only
Dim array1D As Variant 'has to be type variant
array1D = Array(1, 2, "A")
'-> array1D(0) = 1, array1D(1) = 2, array1D(2) = "A"
```

Aus Reichweite

```
Dim arrayRange As Variant 'has to be type variant

'putting ranges in an array always creates a 2D array (even if only 1 row or column)
'starting at 1 and not 0, first dimension is the row and the second the column
arrayRange = Range("A1:C10").Value
'-> arrayRange(1,1) = value in A1
'-> arrayRange(1,2) = value in B1
'-> arrayRange(5,3) = value in C5
'...

'Yoo can get an one-dimensional array from a range (row or column)
'by using the worksheet functions index and transpose:
```

```
'one row from range into 1D-Array:
arrayRange = Application.WorksheetFunction.Index(Range("A1:C10").Value, 3, 0)
'-> row 3 of range into 1D-Array
'-> arrayRange(1) = value in A3, arrayRange(2) = value in B3, arrayRange(3) = value in C3

'one column into 1D-Array:
'limited to 65536 rows in the column, reason: limit of .Transpose
arrayRange = Application.WorksheetFunction.Index( _
Application.WorksheetFunction.Transpose(Range("A1:C10").Value), 2, 0)
'-> column 2 of range into 1D-Array
'-> arrayRange(1) = value in B1, arrayRange(2) = value in B2, arrayRange(3) = value in B3
'...

'By using Evaluate() - shorthand [] - you can transfer the
'range to an array and change the values at the same time.
'This is equivalent to an array formula in the sheet:
arrayRange = [(A1:C10*3)]
arrayRange = [(A1:C10&"_test")]
arrayRange = [(A1:B10*C1:C10)]
'...
```

2D mit Auswerten ()

```
Dim array2D As Variant
'[] ist a shorthand for evaluate()
'Arrays defined with evaluate start at 1 not 0
array2D = [{"1A","1B","1C";"2A","2B","3B"}]
'-> array2D(1,1) = "1A", array2D(1,2) = "1B", array2D(2,1) = "2A" ...

'if you want to use a string to fill the 2D-Array:
Dim strValues As String
strValues = "{""1A"",""1B"",""1C"";""2A"",""2B"",""2C""}"
array2D = Evaluate(strValues)
```

Verwenden der Funktion Split ()

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

Dynamische Arrays (Größenanpassung von Arrays und dynamisches Handling)

Da es sich nicht um exklusive Excel-VBA-Inhalte handelt, wurde dieses Beispiel in die VBA-Dokumentation verschoben.

Link: Dynamische Arrays (Größenanpassung von Arrays und dynamisches Handling)

Gezackte Arrays (Arrays von Arrays)

Da es sich nicht um exklusive Excel-VBA-Inhalte handelt, wurde dieses Beispiel in die VBA-

Dokumentation verschoben.

Link: Gezackte Arrays (Arrays von Arrays)

Prüfen Sie, ob das Array initialisiert ist (ob es Elemente enthält oder nicht).

Ein häufiges Problem ist der Versuch, über ein Array zu iterieren, das keine Werte enthält. Zum Beispiel:

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

Um dieses Problem zu vermeiden und zu überprüfen, ob ein Array Elemente enthält, verwenden Sie diesen *Oneliner* :

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

Dynamische Arrays [Array-Deklaration, Größenänderung]

```
Sub Array_clarity()

Dim arr() As Variant  'creates an empty array
Dim x As Long
Dim y As Long

x = Range("A1", Range("A1").End(xlDown)).Cells.Count
y = Range("A1", Range("A1").End(xlToRight)).Cells.Count

ReDim arr(0 To x, 0 To y) 'fixing the size of the array

For x = LBound(arr, 1) To UBound(arr, 1)
    For y = LBound(arr, 2) To UBound(arr, 2)
        arr(x, y) = Range("A1").Offset(x, y) 'storing the value of Range("A1:E10") from
activesheet in x and y variables
    Next
Next

'Put it on the same sheet according to the declaration:
Range("A14").Resize(UBound(arr, 1), UBound(arr, 2)).Value = arr

End Sub
```

Arrays online lesen: https://riptutorial.com/de/excel-vba/topic/2027/arrays

Kapitel 6: automatischer Filter ; Verwendungen und Best Practices

Einführung

Das ultimative Ziel des ***Autofilters*** ist es, auf möglichst ***schnelle*** Art und Weise Data Mining aus Hunderten oder Tausenden von ***Zeilendaten*** bereitzustellen, um die Aufmerksamkeit auf die Elemente zu lenken, auf die wir uns konzentrieren möchten. Es kann Parameter wie "Text / Werte / Farben" empfangen und sie können zwischen Spalten angeordnet werden. Sie können bis zu 2 Kriterien pro Spalte basierend auf logischen Konnektoren und Regelwerken verbinden.
Anmerkung: Autofilter filtert Zeilen, es gibt keinen Autofilter, um Spalten zu filtern (zumindest nicht nativ).

Bemerkungen

'Um Autofilter innerhalb von VBA verwenden zu können, müssen wir mindestens die folgenden Parameter aufrufen:

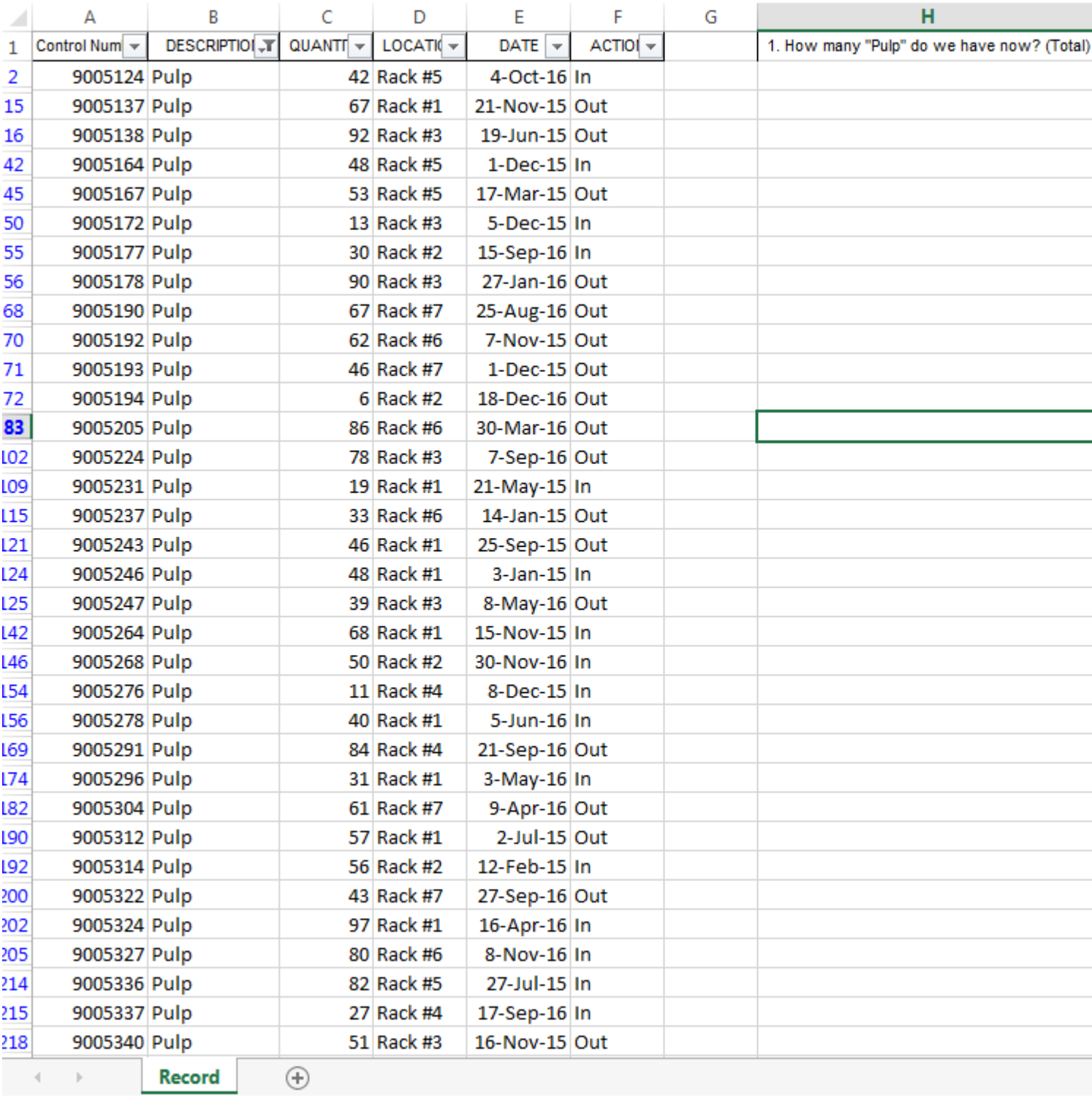
Sheet ("MySheet"). Range ("MyRange"). Autofilter Field = (ColumnNumberWithin "MyRange" ToBeFilteredInNumericValue)

„Es gibt viele Beispiele, entweder im Web oder hier bei stackoverflow

Examples

Smartfilter!

Problemsituation
Der Lagerverwalter verfügt über ein Blatt ("Record"), in dem jede von der Einrichtung ausgeführte Logistikbewegung gespeichert wird. Er kann nach Bedarf filtern. Dies ist jedoch sehr zeitaufwändig und er möchte den Prozess verbessern, um Anfragen schneller berechnen zu können Beispiel: Wieviel "Fruchtfleisch" haben wir jetzt (in allen Racks)? Wie viel Fruchtfleisch haben wir jetzt (in Gestell Nr. 5)? Filter sind ein großartiges Werkzeug, aber sie sind etwas eingeschränkt, um diese Art von Frage in Sekundenschnelle zu beantworten.



	A	B	C	D	E	F	G	H
1	Control Num	DESCRIPTIO	QUANTI	LOCATI	DATE	ACTIO		1. How many "Pulp" do we have now? (Total)
2	9005124	Pulp	42	Rack #5	4-Oct-16	In		
15	9005137	Pulp	67	Rack #1	21-Nov-15	Out		
16	9005138	Pulp	92	Rack #3	19-Jun-15	Out		
42	9005164	Pulp	48	Rack #5	1-Dec-15	In		
45	9005167	Pulp	53	Rack #5	17-Mar-15	Out		
50	9005172	Pulp	13	Rack #3	5-Dec-15	In		
55	9005177	Pulp	30	Rack #2	15-Sep-16	In		
56	9005178	Pulp	90	Rack #3	27-Jan-16	Out		
68	9005190	Pulp	67	Rack #7	25-Aug-16	Out		
70	9005192	Pulp	62	Rack #6	7-Nov-15	Out		
71	9005193	Pulp	46	Rack #7	1-Dec-15	Out		
72	9005194	Pulp	6	Rack #2	18-Dec-16	Out		
83	9005205	Pulp	86	Rack #6	30-Mar-16	Out		
102	9005224	Pulp	78	Rack #3	7-Sep-16	Out		
109	9005231	Pulp	19	Rack #1	21-May-15	In		
115	9005237	Pulp	33	Rack #6	14-Jan-15	Out		
121	9005243	Pulp	46	Rack #1	25-Sep-15	Out		
124	9005246	Pulp	48	Rack #1	3-Jan-15	In		
125	9005247	Pulp	39	Rack #3	8-May-16	Out		
142	9005264	Pulp	68	Rack #1	15-Nov-15	In		
146	9005268	Pulp	50	Rack #2	30-Nov-16	In		
154	9005276	Pulp	11	Rack #4	8-Dec-15	In		
156	9005278	Pulp	40	Rack #1	5-Jun-16	In		
169	9005291	Pulp	84	Rack #4	21-Sep-16	Out		
174	9005296	Pulp	31	Rack #1	3-May-16	In		
182	9005304	Pulp	61	Rack #7	9-Apr-16	Out		
190	9005312	Pulp	57	Rack #1	2-Jul-15	Out		
192	9005314	Pulp	56	Rack #2	12-Feb-15	In		
200	9005322	Pulp	43	Rack #7	27-Sep-16	Out		
202	9005324	Pulp	97	Rack #1	16-Apr-16	In		
205	9005327	Pulp	80	Rack #6	8-Nov-16	In		
214	9005336	Pulp	82	Rack #5	27-Jul-15	In		
215	9005337	Pulp	27	Rack #4	17-Sep-16	In		
218	9005340	Pulp	51	Rack #3	16-Nov-15	Out		

Record

Makrolösung:

Der Codierer weiß, dass ***Autofilter die beste, schnellste und zuverlässigste Lösung*** in solchen Szenarien sind, da ***die Daten bereits im Arbeitsblatt vorhanden sind*** und die ***Eingabe für sie einfach*** in diesem Fall durch Benutzereingaben abgerufen werden kann.

Der verwendete Ansatz besteht darin, ein Blatt mit dem Namen "SmartFilter" zu erstellen, in dem der Administrator mehrere Daten nach Bedarf filtern kann und die Berechnung sofort ausgeführt wird.

Er verwendet für diese Angelegenheit 2 Module und das Ereignis `Worksheet_Change`

Code für SmartFilter-Arbeitsblatt:

```
Private Sub Worksheet_Change(ByVal Target As Range)
Dim ItemInRange As Range
Const CellsFilters As String = "C2,E2,G2"
    Call ExcelBusy
    For Each ItemInRange In Target
    If Not Intersect(ItemInRange, Range(CellsFilters)) Is Nothing Then Call Inventory_Filter
    Next ItemInRange
    Call ExcelNormal
End Sub
```

Code für Modul 1 mit dem Namen "General_Functions"

```
Sub ExcelNormal()
        With Excel.Application
        .EnableEvents = True
        .Cursor = xlDefault
        .ScreenUpdating = True
        .DisplayAlerts = True
        .StatusBar = False
        .CopyObjectsWithCells = True
        End With
End Sub
Sub ExcelBusy()
        With Excel.Application
        .EnableEvents = False
        .Cursor = xlWait
        .ScreenUpdating = False
        .DisplayAlerts = False
        .StatusBar = False
        .CopyObjectsWithCells = True
        End With
End Sub
Sub Select_Sheet(NameSheet As String, Optional VerifyExistanceOnly As Boolean)
    On Error GoTo Err01Select_Sheet
    Sheets(NameSheet).Visible = True
    If VerifyExistanceOnly = False Then ' 1. If VerifyExistanceOnly = False
    Sheets(NameSheet).Select
    Sheets(NameSheet).AutoFilterMode = False
    Sheets(NameSheet).Cells.EntireRow.Hidden = False
    Sheets(NameSheet).Cells.EntireColumn.Hidden = False
    End If ' 1. If VerifyExistanceOnly = False
    If 1 = 2 Then '99. If error
Err01Select_Sheet:
    MsgBox "Err01Select_Sheet: Sheet " & NameSheet & " doesn't exist!", vbCritical: Call
ExcelNormal: On Error GoTo -1: End
    End If '99. If error
End Sub
Function General_Functions_Find_Title(InSheet As String, TitleToFind As String, Optional
InRange As Range, Optional IsNeededToExist As Boolean, Optional IsWhole As Boolean) As Range
Dim DummyRange As Range
    On Error GoTo Err01General_Functions_Find_Title
    If InRange Is Nothing Then ' 1. If InRange Is Nothing
    Set DummyRange = IIf(IsWhole = True, Sheets(InSheet).Cells.Find(TitleToFind,
LookAt:=xlWhole), Sheets(InSheet).Cells.Find(TitleToFind, LookAt:=xlPart))
    Else ' 1. If InRange Is Nothing
    Set DummyRange = IIf(IsWhole = True,
Sheets(InSheet).Range(InRange.Address).Find(TitleToFind, LookAt:=xlWhole),
```

```
Sheets(InSheet).Range(InRange.Address).Find(TitleToFind, LookAt:=xlPart))
    End If ' 1. If InRange Is Nothing
    Set General_Functions_Find_Title = DummyRange
    If 1 = 2 Or DummyRange Is Nothing Then '99. If error
Err01General_Functions_Find_Title:
    If IsNeededToExist = True Then MsgBox "Err01General_Functions_Find_Title: Ttile '" &
TitleToFind & "' was not found in sheet '" & InSheet & "'", vbCritical: Call ExcelNormal: On
Error GoTo -1: End
    End If '99. If error
End Function
```

Code für Modul 2 mit dem Namen "Inventory_Handling"

```
Const TitleDesc As String = "DESCRIPTION"
Const TitleLocation As String = "LOCATION"
Const TitleActn As String = "ACTION"
Const TitleQty As String = "QUANTITY"
Const SheetRecords As String = "Record"
Const SheetSmartFilter As String = "SmartFilter"
Const RowFilter As Long = 2
Const ColDataToPaste As Long = 2
Const RowDataToPaste As Long = 7
Const RangeInResult As String = "K1"
Const RangeOutResult As String = "K2"
Sub Inventory_Filter()
Dim ColDesc As Long: ColDesc = General_Functions_Find_Title(SheetSmartFilter, TitleDesc,
IsNeededToExist:=True, IsWhole:=True).Column
Dim ColLocation As Long: ColLocation = General_Functions_Find_Title(SheetSmartFilter,
TitleLocation, IsNeededToExist:=True, IsWhole:=True).Column
Dim ColActn As Long: ColActn = General_Functions_Find_Title(SheetSmartFilter, TitleActn,
IsNeededToExist:=True, IsWhole:=True).Column
Dim ColQty As Long: ColQty = General_Functions_Find_Title(SheetSmartFilter, TitleQty,
IsNeededToExist:=True, IsWhole:=True).Column
Dim CounterQty As Long
Dim TotalQty As Long
Dim TotalIn As Long
Dim TotalOut As Long
Dim RangeFiltered As Range
    Call Select_Sheet(SheetSmartFilter)
    If Cells(Rows.Count, ColDataToPaste).End(xlUp).Row > RowDataToPaste - 1 Then
Rows(RowDataToPaste & ":" & Cells(Rows.Count, "B").End(xlUp).Row).Delete
    Sheets(SheetRecords).AutoFilterMode = False
    If Cells(RowFilter, ColDesc).Value <> "" Or Cells(RowFilter, ColLocation).Value <> "" Or
Cells(RowFilter, ColActn).Value <> "" Then ' 1. If Cells(RowFilter, ColDesc).Value <> "" Or
Cells(RowFilter, ColLocation).Value <> "" Or Cells(RowFilter, ColActn).Value <> ""
    With Sheets(SheetRecords).UsedRange
    If Sheets(SheetSmartFilter).Cells(RowFilter, ColDesc).Value <> "" Then .AutoFilter
Field:=General_Functions_Find_Title(SheetRecords, TitleDesc, IsNeededToExist:=True,
IsWhole:=True).Column, Criteria1:=Sheets(SheetSmartFilter).Cells(RowFilter, ColDesc).Value
    If Sheets(SheetSmartFilter).Cells(RowFilter, ColLocation).Value <> "" Then .AutoFilter
Field:=General_Functions_Find_Title(SheetRecords, TitleLocation, IsNeededToExist:=True,
IsWhole:=True).Column, Criteria1:=Sheets(SheetSmartFilter).Cells(RowFilter, ColLocation).Value
    If Sheets(SheetSmartFilter).Cells(RowFilter, ColActn).Value <> "" Then .AutoFilter
Field:=General_Functions_Find_Title(SheetRecords, TitleActn, IsNeededToExist:=True,
IsWhole:=True).Column, Criteria1:=Sheets(SheetSmartFilter).Cells(RowFilter, ColActn).Value
    'If we don't use a filter we would need to use a cycle For/to or For/Each Cell in range
    'to determine whether or not the row meets the criteria that we are looking and then
    'save it on an array, collection, dictionary, etc
    'IG: For CounterRow = 2 To TotalRows
    'If Sheets(SheetSmartFilter).Cells(RowFilter, ColDesc).Value <> "" and
```

```
Sheets(SheetRecords).cells(CounterRow,ColDescInRecords).Value=
Sheets(SheetSmartFilter).Cells(RowFilter, ColDesc).Value then
    'Redim Preserve MyUnecessaryArray(UnecessaryNumber) ''Save to array:
(UnecessaryNumber)=MyUnecessaryArray. Or in a dictionary, etc. At the end, we would transpose
this values into the sheet, at the end
    'both are the same, but, just try to see the time invested on each logic.
    If .Cells(1, 1).End(xlDown).Value <> "" Then Set RangeFiltered = .Rows("2:" &
Sheets(SheetRecords).Cells(Rows.Count, "A").End(xlUp).Row).SpecialCells(xlCellTypeVisible)
    'If it is not <>"" means that there was not filtered data!
    If RangeFiltered Is Nothing Then MsgBox "Err01Inventory_Filter: No data was found with the
given criteria!", vbCritical: Call ExcelNormal: End
    RangeFiltered.Copy Destination:=Cells(RowDataToPaste, ColDataToPaste)
    TotalQty = Cells(Rows.Count, ColQty).End(xlUp).Row
    For CounterQty = RowDataToPaste + 1 To TotalQty
    If Cells(CounterQty, ColActn).Value = "In" Then ' 2. If Cells(CounterQty, ColActn).Value =
"In"
    TotalIn = Cells(CounterQty, ColQty).Value + TotalIn
    ElseIf Cells(CounterQty, ColActn).Value = "Out" Then ' 2. If Cells(CounterQty,
ColActn).Value = "In"
    TotalOut = Cells(CounterQty, ColQty).Value + TotalOut
    End If ' 2. If Cells(CounterQty, ColActn).Value = "In"
    Next CounterQty
    Range(RangeInResult).Value = TotalIn
    Range(RangeOutResult).Value = -(TotalOut)
    End With
    End If ' 1. If Cells(RowFilter, ColDesc).Value <> "" Or Cells(RowFilter,
ColLocation).Value <> "" Or Cells(RowFilter, ColActn).Value <> ""
End Sub
```

Tests und Ergebnisse:

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

SmartFilter | Record

Wie wir im vorherigen Bild gesehen haben, wurde diese Aufgabe leicht gelöst. Durch die Verwendung von ***Autofiltern wurde*** eine Lösung bereitgestellt, die nur wenige ***Sekunden zur Berechnung benötigt.*** Dies ist ***für den Benutzer einfach zu erklären,*** da er / sie mit diesem Befehl vertraut ist, und ***nahm einige Zeilen zum Codierer auf.***

automatischer Filter ; Verwendungen und Best Practices online lesen: https://riptutorial.com/de/excel-vba/topic/8645/automatischer-filter---verwendungen-und-best-practices

Kapitel 7: Bedingte Anweisungen

Examples

Die If-Anweisung

Die `If` Steueranweisung ermöglicht die Ausführung von unterschiedlichem Code, abhängig von der Auswertung einer bedingten (booleschen) Anweisung. Eine Bedingungsanweisung ist eine, die entweder `True` oder `False` ergibt, z. B. `x > 2` .

Bei der Implementierung einer `If` Anweisung können drei Muster verwendet werden, die im Folgenden beschrieben werden. Beachten Sie, dass auf eine bedingte `If` Bewertung immer ein `Then` folgt.

1. Eine auswerten `If` bedingte Anweisung und etwas tun, wenn es `True`

Einzeilige `If` Anweisung

Dies ist der kürzeste Weg, ein `If` und es ist nützlich, wenn bei einer `True` Auswertung nur eine Anweisung ausgeführt werden muss. Bei Verwendung dieser Syntax muss sich der gesamte Code in einer einzelnen Zeile befinden. Fügen Sie am Ende der Zeile kein `End If` .

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

`If` blockieren

Wenn bei einer `True` Auswertung mehrere Codezeilen ausgeführt werden müssen, kann ein `If` Block verwendet werden.

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

Beachten Sie, dass bei Verwendung eines mehrzeiligen `If` Blocks ein entsprechendes `End If` erforderlich ist.

2. Eine bedingte `If` Anweisung auswerten, eine Sache tun, wenn sie `True` und etwas anderes, wenn sie `False`

Einzelne Zeile `If` , `Else` Anweisung

Dies kann verwendet werden, wenn eine Anweisung bei einer `True` Auswertung und eine andere Anweisung bei einer `False` Auswertung ausgeführt werden soll. Seien Sie vorsichtig mit dieser Syntax, da den Lesern oftmals weniger klar ist, dass es eine `Else` Anweisung gibt. Bei Verwendung dieser Syntax muss sich der gesamte Code in einer einzelnen Zeile befinden. Fügen Sie am Ende der Zeile kein `End If` .

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

`If` , `Else` blockieren

Verwenden Sie einen `If` , `Else` Block, um den Code klarer zu machen, oder wenn mehrere Codezeilen unter einer `True` oder einer `False` Auswertung ausgeführt werden müssen.

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

Beachten Sie, dass bei Verwendung eines mehrzeiligen `If` Blocks ein entsprechendes `End If` erforderlich ist.

3. Viele bedingte Anweisungen auswerten, wenn die vorangegangenen Aussagen alle `False` , und für jede einzelne etwas anderes tun

Dieses Muster ist die allgemeinste Verwendung von `If` und würde verwendet, wenn viele nicht überlappende Bedingungen vorliegen, die eine andere Behandlung erfordern. Im Gegensatz zu den ersten beiden Mustern erfordert dieser Fall die Verwendung eines `If` Blocks, auch wenn nur eine Codezeile für jede Bedingung ausgeführt wird.

`If` , `ElseIf` , `...` , `Else` Block

Anstatt viele `If` Blöcke untereinander `ElseIf` kann ein `ElseIf` verwendet werden, um eine zusätzliche Bedingung auszuwerten. Die `ElseIf` wird nur ausgewertet, wenn eine vorangehende `If` Bewertung `False` .

```
If [Some condition is True] Then
   [Do some thing(s)]
ElseIf [Some other condition is True] Then
   [Do some different thing(s)]
Else    'Everything above has evaluated to False
   [Do some other thing(s)]
End If
```

`ElseIf` können beliebig viele `ElseIf` Steueranweisungen zwischen einem `If` und einem `End If` `ElseIf` werden. Eine `Else` Steueranweisung ist nicht erforderlich, wenn `ElseIf` (obwohl dies empfohlen wird). Wenn sie jedoch enthalten ist, muss es die letzte Steueranweisung vor dem `End If` .

Bedingte Anweisungen online lesen: https://riptutorial.com/de/excel-vba/topic/9632/bedingte-anweisungen

Kapitel 8: Bedingte Formatierung mit VBA

Bemerkungen

Sie können nicht mehr als drei bedingte Formate für einen Bereich definieren. Verwenden Sie die Modify-Methode, um ein vorhandenes bedingtes Format zu ändern, oder löschen Sie ein vorhandenes Format mit der Delete-Methode, bevor Sie ein neues hinzufügen.

Examples

FormatConditions.Add

Syntax:

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

Parameter:

Name	Erforderlich / optional	Datentyp
Art	Erforderlich	XlFormatConditionType
Operator	Wahlweise	Variante
Formel 1	Wahlweise	Variante
Formula2	Wahlweise	Variante

XlFormatConditionType-Enumeration:

Name	Beschreibung
xlAboveAverageCondition	Überdurchschnittlicher Zustand
xlBlanksCondition	Leere Bedingung
xlCellValue	Zellenwert
xlColorScale	Farbskala
xlDatabar	Databar

Name	Beschreibung
xlErrorsCondition	Fehlerbedingung
xlExpression	Ausdruck
XlIconSet	Icon-Set
xlNoBlanksCondition	Keine Leerzeichen
xlNoErrorsCondition	Keine Fehlerbedingung
xlTextString	Textzeichenfolge
xlTimePeriod	Zeitperiode
xlTop10	Top 10 Werte
xlUniqueValues	Einzigartige Werte

Formatierung nach Zellenwert:

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

Betreiber:

Name
xlBetween
xlEqual
xlGreater
xlGreaterEqual
xlLess
xlLessEqual
xlNotBetween
xlNotEqual

Wenn Typ xlExpression ist, wird das Operator-Argument ignoriert.

Die Formatierung nach Text enthält:

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

Betreiber:

Name	Beschreibung
xlBeginsWith	Beginnt mit einem angegebenen Wert.
xlContains	Enthält einen angegebenen Wert.
xlDoesNotContain	Enthält nicht den angegebenen Wert.
xlEndsWith	Ende mit dem angegebenen Wert

Formatierung nach Zeitraum

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

Betreiber:

Name
Gestern
xl Morgen
xlLast7Days
xlLastWeek
xlThisWeek

Name
xlNextWeek
xlLastMonth
xlThisMonth
xlNextMonth

Bedingtes Format entfernen

Entfernen Sie alle bedingten Formate im Bereich:

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

Entfernen Sie alle bedingten Formate im Arbeitsblatt:

```
Cells.FormatConditions.Delete
```

FormatConditions.AddUniqueValues

Doppelte Werte hervorheben

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

Einzigartige Werte hervorheben

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

FormatConditions.AddTop10

Top 5-Werte hervorheben

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

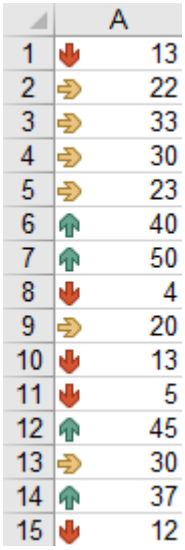
FormatConditions.AddAboveAverage

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

Betreiber:

Name	Beschreibung
XlAboveAverage	Überdurchschnittlich
XlAboveStdDev	Oberhalb der Standardabweichung
XlBelowAverage	Unterdurchschnittlich
XlBelowStdDev	Unter Standardabweichung
XlEqualAboveAverage	Gleich überdurchschnittlich
XlEqualBelowAverage	Gleich unterdurchschnittlich

FormatConditions.AddIconSetCondition



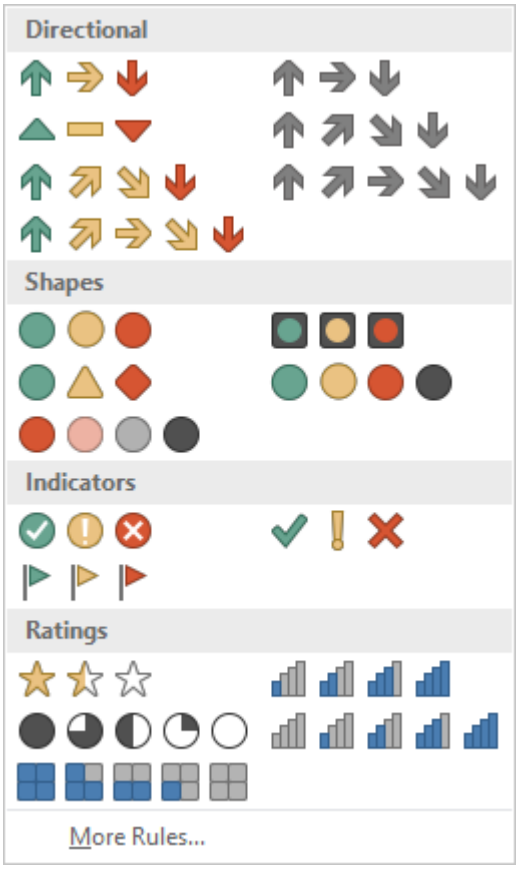
	A
1	13
2	22
3	33
4	30
5	23
6	40
7	50
8	4
9	20
10	13
11	5
12	45
13	30
14	37
15	12

```
Range("a1:a10").FormatConditions.AddIconSetCondition
With Selection.FormatConditions(1)
    .ReverseOrder = False
    .ShowIconOnly = False
    .IconSet = ActiveWorkbook.IconSets(xl3Arrows)
End With

With Selection.FormatConditions(1).IconCriteria(2)
    .Type = xlConditionValuePercent
    .Value = 33
    .Operator = 7
End With

With Selection.FormatConditions(1).IconCriteria(3)
    .Type = xlConditionValuePercent
    .Value = 67
    .Operator = 7
End With
```

IconSet:

Name
xl3-pfeile
xl3ArrowsGray
xl3Flags
xl3Signs
xl3Stars
xl3Symbole
xl3Symbols2
xl3TrafficLights1

Name
xl3TrafficLights2
xl3Dreieck
xl4-pfeile
xl4ArrowsGray
xl4CRV
xl4RedToBlack
xl4TrafficLights
xl5Pfeile
xl5ArrowsGray
xl5Boxen
xl5CRV
xl5Quarters



Art:

Name
xlConditionValuePercent
xlConditionValueNumber
xlConditionValuePercentile
xlConditionValueFormula

Operator:

Name	Wert
xlGreater	5
xlGreaterEqual	7

Wert:

Gibt den Schwellenwert für ein Symbol in einem bedingten Format zurück oder legt diesen fest.

Bedingte Formatierung mit VBA online lesen: https://riptutorial.com/de/excel-vba/topic/9912/bedingte-formatierung-mit-vba

Kapitel 9: Benannte Bereiche

Einführung

Das Thema sollte Informationen enthalten, die sich speziell auf benannte Bereiche in Excel beziehen, einschließlich Methoden zum Erstellen, Ändern, Löschen und Zugreifen auf definierte benannte Bereiche.

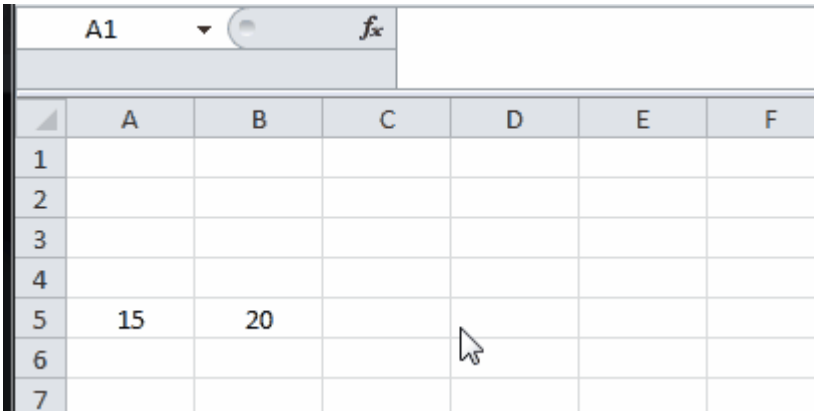
Examples

Definieren Sie einen benannten Bereich

Mit benannten Bereichen können Sie die Bedeutung des Zellinhalts beschreiben und diesen definierten Namen anstelle einer tatsächlichen Zelladresse verwenden.

Zum Beispiel kann formula `=A5*B5` durch `=Width*Height` , um das Verständnis der Formel zu erleichtern.

Um einen neuen benannten Bereich zu definieren, wählen Sie die zu benennenden Zellen oder Zellen aus, und geben Sie den neuen Namen in das Namensfeld neben der Formelleiste ein.



> Hinweis: Benannte Bereiche sind standardmäßig auf den globalen Bereich eingestellt. Dies bedeutet, dass von überall in der Arbeitsmappe darauf zugegriffen werden kann. Ältere Versionen von Excel lassen doppelte Namen zu. Daher muss vermieden werden, dass doppelte Namen des globalen Gültigkeitsbereichs verwendet werden, da sonst die Ergebnisse unvorhersehbar sind. Verwenden Sie den Namensmanager auf der Registerkarte Formeln, um den Bereich zu ändern.

Benannte Bereiche in VBA verwenden

Erstellen Sie einen neuen benannten Bereich mit dem Namen 'MyRange', der Zelle `A1` zugewiesen ist

```
ThisWorkbook.Names.Add Name:="MyRange", _
```

```
    RefersTo:=Worksheets("Sheet1").Range("A1")
```

Definierten benannten Bereich nach Namen **löschen**

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

Zugriff auf benannten Bereich über den Namen

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

Greifen Sie mit einer Verknüpfung auf einen benannten Bereich zu

Wie auf alle anderen Bereiche kann auf benannte Bereiche direkt mit einer Kurzbefehlsnotation zugegriffen werden, für die kein `Range` Objekt erstellt werden muss. Die drei Zeilen des obigen Codeausschnitts können durch eine einzige Zeile ersetzt werden:

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

> Hinweis: Die Standardeigenschaft für einen Bereich ist der Wert. `[MyRange]` `[MyRange].Value`

Sie können auch Methoden für den Bereich aufrufen. Folgendes wählt `MyRange` :

```
[MyRange].Select
```

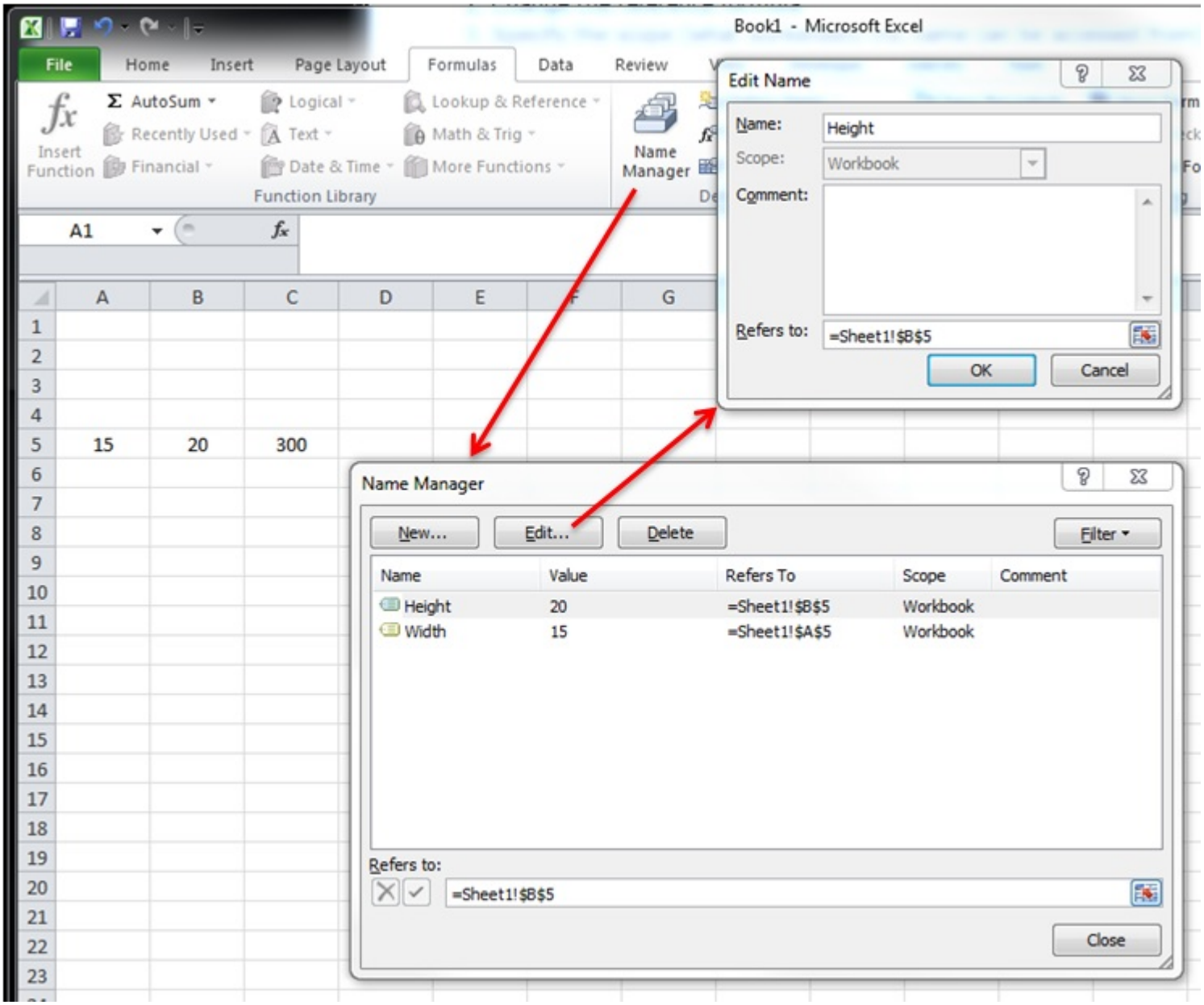
> Hinweis: Eine Einschränkung ist, dass die Kurzbefehlsnotation nicht mit Wörtern funktioniert, die an anderer Stelle in der VBA-Bibliothek verwendet werden. Ein Bereich mit dem Namen `Width` wäre beispielsweise nicht als `[Width]` `ThisWorkbook.Worksheets("Sheet1").Range("Width")` , würde jedoch wie erwartet funktionieren, wenn der Zugriff über `ThisWorkbook.Worksheets("Sheet1").Range("Width")`

Verwalten Sie benannte Bereiche mit dem Namensmanager

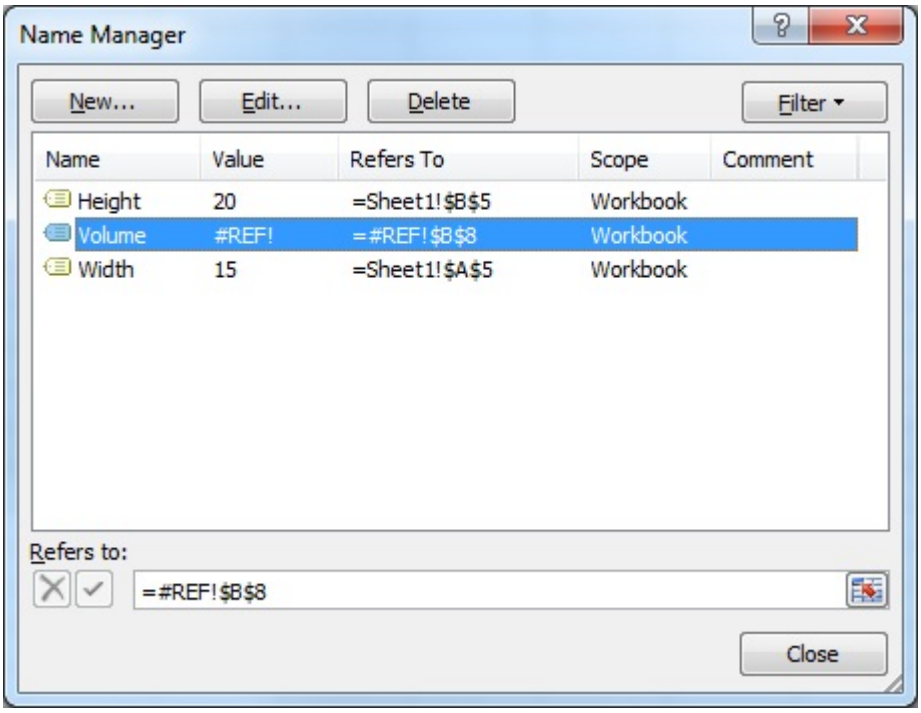
Registerkarte "Formeln"> Gruppe "Definierte Namen"> Schaltfläche "Namensmanager"

Mit dem benannten Manager können Sie:

1. Namen erstellen oder ändern
2. Zellreferenz erstellen oder ändern
3. Bereich erstellen oder ändern
4. Vorhandenen benannten Bereich löschen

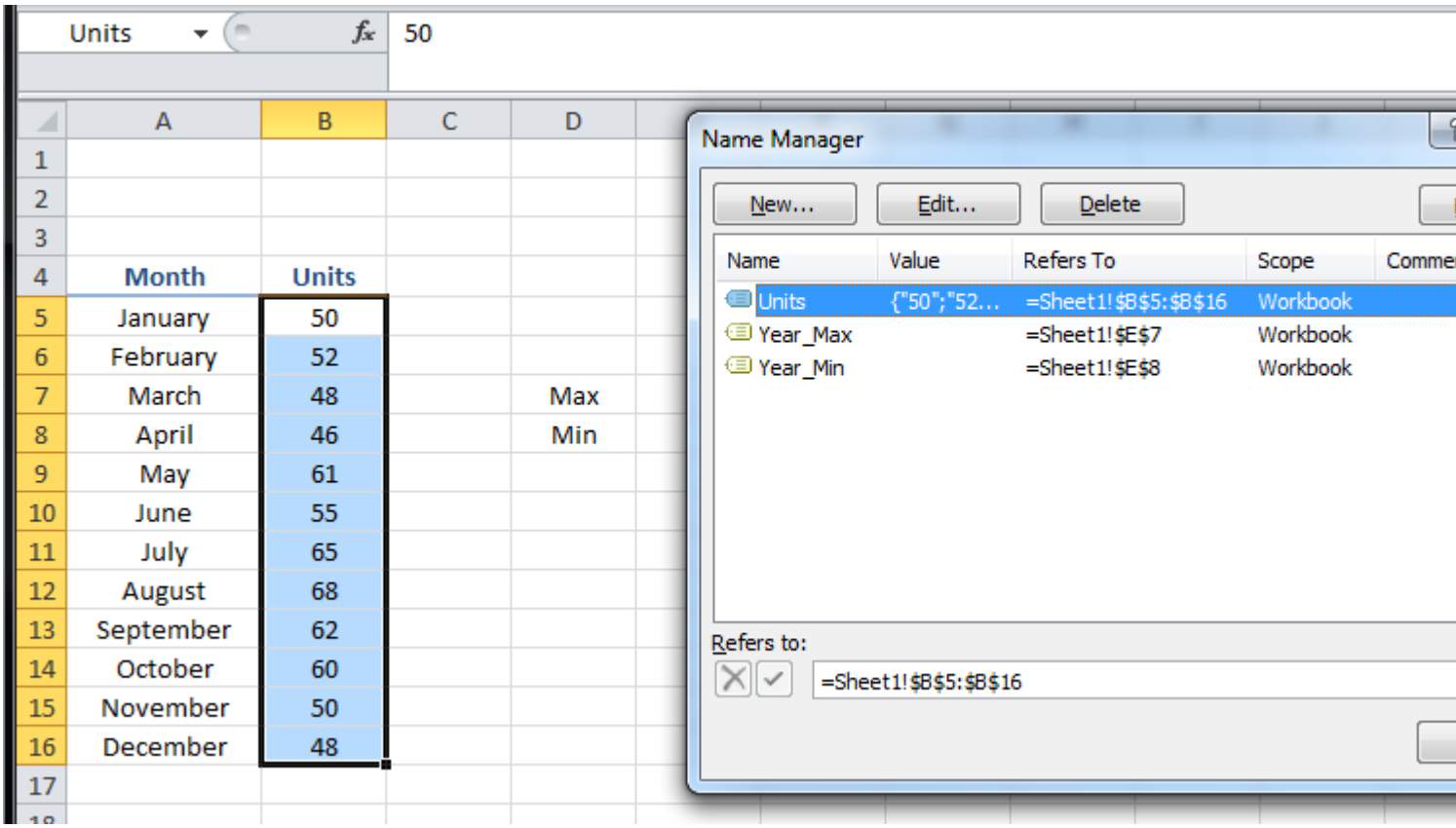


Named Manager bietet eine schnelle Suche nach defekten Links.



Benannte Range Arrays

Beispielblatt



Code

```
Sub Example()
    Dim wks As Worksheet
```

```
    Set wks = ThisWorkbook.Worksheets("Sheet1")

    Dim units As Range
    Set units = ThisWorkbook.Names("Units").RefersToRange

    Worksheets("Sheet1").Range("Year_Max").Value = WorksheetFunction.Max(units)
    Worksheets("Sheet1").Range("Year_Min").Value = WorksheetFunction.Min(units)
End Sub
```

Ergebnis

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

Benannte Bereiche online lesen: https://riptutorial.com/de/excel-vba/topic/8360/benannte-bereiche

Kapitel 10: Benutzerdefinierte Funktionen (UDFs)

Syntax

1. **Funktion functionName (ArgumentVariable As Datentyp, ArgumentVariable2 Als Datentyp, Optionales ArgumentVariable3 Als Datentyp) Als FunktionReturnDataType**
   Grunddeklaration einer Funktion. Jede Funktion benötigt einen Namen, muss jedoch keine Argumente enthalten. Es können 0 Argumente oder eine bestimmte Anzahl von Argumenten verwendet werden. Sie können ein Argument auch als optional deklarieren (dh es spielt keine Rolle, ob Sie es beim Aufruf der Funktion angeben). Es ist empfehlenswert, den Variablentyp für jedes Argument anzugeben und auch zurückzugeben, welchen Datentyp die Funktion selbst zurückgeben wird.

2. **functionName = theVariableOrValueBeingReturned**
   Wenn Sie aus anderen Programmiersprachen kommen, sind Sie möglicherweise an das `Return` Schlüsselwort gewöhnt. Dies wird in VBA nicht verwendet - stattdessen verwenden wir den Funktionsnamen. Sie können den Inhalt einer Variablen oder einen direkt bereitgestellten Wert festlegen. Wenn Sie einen Datentyp für die Rückgabe der Funktion festgelegt haben, müssen die Variablen oder Daten, die Sie zu diesem Zeitpunkt bereitstellen, diesen Datentyp haben.

3. **Funktion beenden**
   Verpflichtend. Bezeichnet das Ende des `Function` Codeblocks und muss somit am Ende sein. Die VBE liefert dies normalerweise automatisch, wenn Sie eine neue Funktion erstellen.

Bemerkungen

Eine benutzerdefinierte Funktion (UDF) bezieht sich auf eine aufgabenspezifische Funktion, die vom Benutzer erstellt wurde. Sie kann als Arbeitsblattfunktion (ex: `=SUM(...)`) aufgerufen werden oder verwendet werden, um einen Wert an einen laufenden Prozess in einer Sub-Prozedur zurückzugeben. Eine UDF gibt einen Wert zurück, normalerweise aus Informationen, die als ein oder mehrere Parameter an sie übergeben werden.

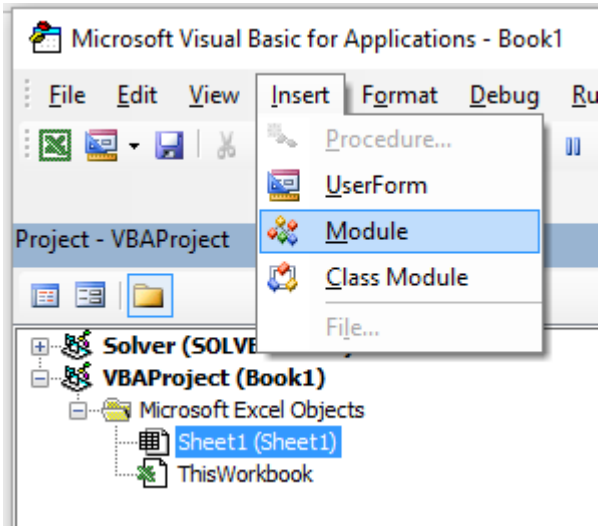
Es kann erstellt werden von:

1. mit VBA.
2. using Excel C API - Durch Erstellen einer XLL, die kompilierte Funktionen nach Excel exportiert.
3. über die COM-Schnittstelle.

Examples

UDF - Hallo Welt

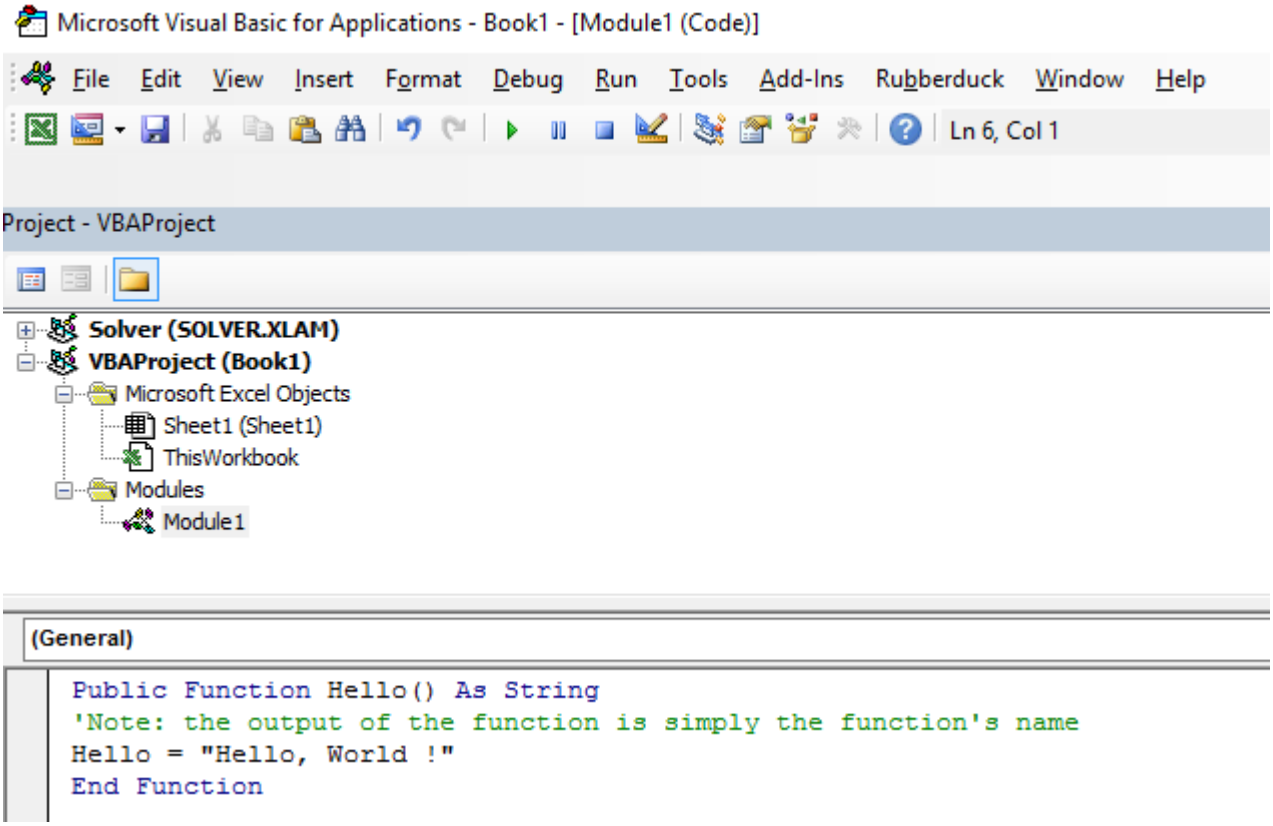
1. Öffnen Sie Excel
2. Öffnen Sie den Visual Basic-Editor (siehe Öffnen des Visual Basic-Editors).
3. Fügen Sie ein neues Modul hinzu, indem Sie auf Einfügen -> Modul klicken:



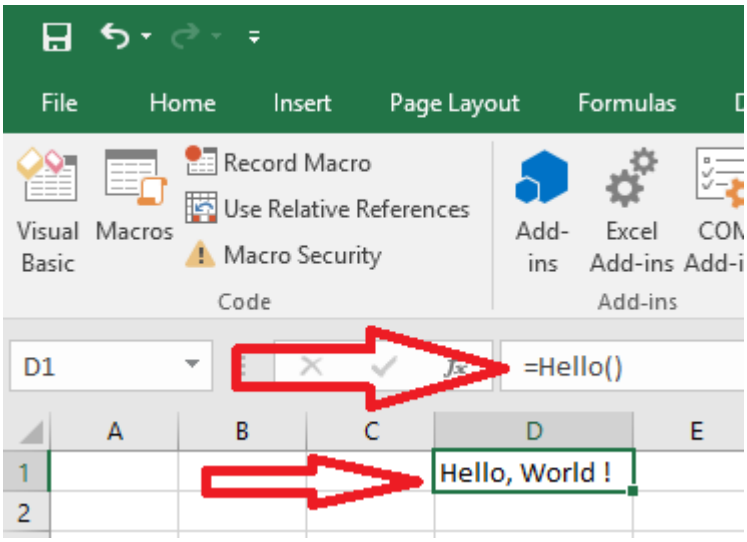
4. Kopieren Sie den folgenden Code, und fügen Sie ihn in das neue Modul ein:

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

Erhalten :



5. Gehen Sie zurück zu Ihrer Arbeitsmappe und geben Sie "= Hello ()" in eine Zelle ein, um "Hello World" anzuzeigen.



Erlauben Sie vollständige Spaltenverweise ohne Strafe

Es ist einfacher, einige UDFs im Arbeitsblatt zu implementieren, wenn vollständige Spaltenverweise als Parameter übergeben werden können. Aufgrund der expliziten Art der Codierung kann jedoch jede Schleife, die diese Bereiche umfasst, Hunderttausende von Zellen verarbeiten, die vollständig leer sind. Dies reduziert Ihr VBA-Projekt (und Ihre Arbeitsmappe) auf ein eingefrorenes Durcheinander, während unnötige Nichtwerte verarbeitet werden.

Das Durchlaufen der Zellen eines Arbeitsblatts ist eine der langsamsten Methoden, um eine Aufgabe auszuführen, aber manchmal ist dies unvermeidlich. Es ist absolut sinnvoll, die geleistete Arbeit auf das Notwendige zu reduzieren.

Die Lösung besteht darin, mit der Intersect-Methode die vollständigen Spalten- oder vollständigen Zeilenverweise auf die Worksheet.UsedRange-Eigenschaft abzuschneiden . Das folgende Beispiel wird lose ein Arbeitsblatt nativen SUMIF Funktion repliziert , so dass die *Kriterien_Bereich* auch die *Summe_Bereich* anpassen Größe verändert werden , da jeder Wert in der *Summe_Bereich* muss von einem Wert in der *Kriterien_Bereich* begleitet werden.

Der Application.Caller für eine in einem Arbeitsblatt verwendete UDF ist die Zelle, in der sie sich befindet. Die .Parent- Eigenschaft der Zelle ist das Arbeitsblatt. Dies wird zur Definition des .UsedRange verwendet.

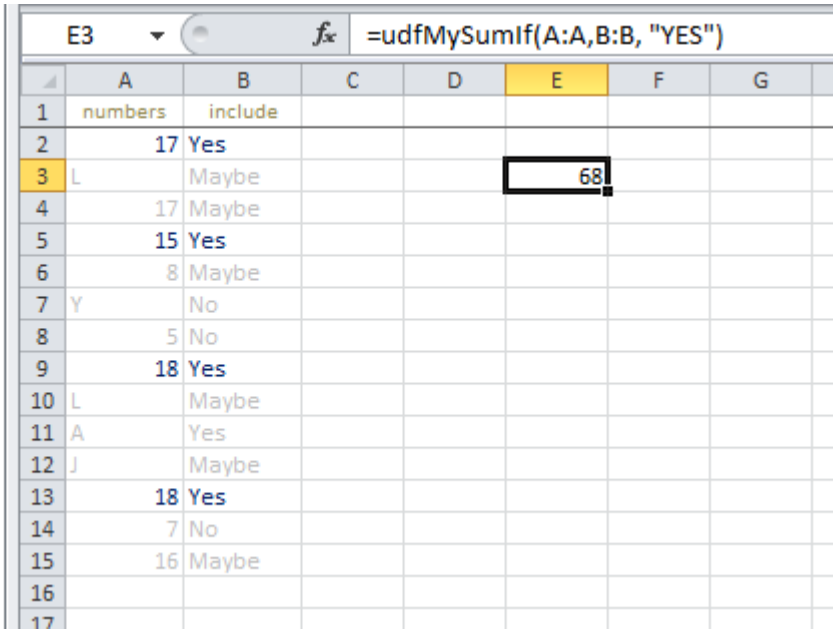
In einem Modul-Codeblatt:

```
Option Explicit

Function udfMySumIf(rngA As Range, rngB As Range, _
                    Optional crit As Variant = "yes")
    Dim c As Long, ttl As Double

    With Application.Caller.Parent
        Set rngA = Intersect(rngA, .UsedRange)
        Set rngB = rngB.Resize(rngA.Rows.Count, rngA.Columns.Count)
```

```
    End With

    For c = 1 To rngA.Cells.Count
        If IsNumeric(rngA.Cells(c).Value2) Then
            If LCase(rngB(c).Value2) = LCase(crit) Then
                ttl = ttl + rngA.Cells(c).Value2
            End If
        End If
    Next c

    udfMySumIf = ttl

End Function
```

Syntax:

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



Dies ist zwar ein ziemlich simpeles Beispiel, zeigt jedoch das Übergeben von zwei vollständigen Spaltenreferenzen (jeweils 1.048.576 Zeilen), wobei jedoch nur 15 Zeilen mit Daten und Kriterien verarbeitet werden.

Verknüpfte offizielle MSDN-Dokumentation einzelner Methoden und Eigenschaften mit freundlicher Genehmigung von Microsoft ™.

Einzelne Werte in Bereich zählen

```
Function countUnique(r As range) As Long
    'Application.Volatile False ' optional
    Set r = Intersect(r, r.Worksheet.UsedRange) ' optional if you pass entire rows or columns
to the function
    Dim c As New Collection, v
    On Error Resume Next   ' to ignore the Run-time error 457: "This key is already associated
with an element of this collection".
    For Each v In r.Value  ' remove .Value for ranges with more than one Areas
        c.Add 0, v & ""
    Next
```

```
    c.Remove "" ' optional to exclude blank values from the count
    countUnique = c.Count
End Function
```

Sammlungen

Benutzerdefinierte Funktionen (UDFs) online lesen: https://riptutorial.com/de/excel-vba/topic/1070/benutzerdefinierte-funktionen--udfs-

Kapitel 11: Bereiche und Zellen

Syntax

- **Set** - Der Operator, mit dem ein Verweis auf ein Objekt festgelegt wird, beispielsweise ein Bereich
- **For Each** - Der Operator, mit dem alle Elemente in einer Sammlung durchlaufen wurden

Bemerkungen

Beachten Sie, dass die Variablennamen `r` , `cell` und andere beliebig benannt werden können, sie sollten jedoch entsprechend benannt werden, damit der Code für Sie und andere leichter verständlich ist.

Examples

Erstellen eines Bereichs

Ein Bereich kann nicht auf die gleiche Weise erstellt oder gefüllt werden, wie eine Zeichenfolge:

```
Sub RangeTest()
    Dim s As String
    Dim r As Range 'Specific Type of Object, with members like Address, WrapText, AutoFill,
etc.

    ' This is how we fill a String:
    s = "Hello World!"

    ' But we cannot do this for a Range:
    r = Range("A1") '//Run. Err.:  91 Object variable or With block variable not set//

    ' We have to use the Object approach, using keyword Set:
    Set r = Range("A1")
End Sub
```

Es gilt als bewährte Methode, Ihre Referenzen zu qualifizieren. Daher verwenden wir ab jetzt denselben Ansatz.
Weitere Informationen zum Erstellen von Objektvariablen (z. B. Range) in MSDN . Weitere Informationen zum Set-Statement in MSDN .

Es gibt verschiedene Möglichkeiten, denselben Bereich zu erstellen:

```
Sub SetRangeVariable()
    Dim ws As Worksheet
    Dim r As Range

    Set ws = ThisWorkbook.Worksheets(1) ' The first Worksheet in Workbook with this code in it

    ' These are all equivalent:
```

```
    Set r = ws.Range("A2")
    Set r = ws.Range("A" & 2)
    Set r = ws.Cells(2, 1) ' The cell in row number 2, column number 1
    Set r = ws.[A2] 'Shorthand notation of Range.
    Set r = Range("NamedRangeInA2") 'If the cell A2 is named NamedRangeInA2. Note, that this
is Sheet independent.
    Set r = ws.Range("A1").Offset(1, 0) ' The cell that is 1 row and 0 columns away from A1
    Set r = ws.Range("A1").Cells(2,1) ' Similar to Offset. You can "go outside" the original
Range.

    Set r = ws.Range("A1:A5").Cells(2) 'Second cell in bigger Range.
    Set r = ws.Range("A1:A5").Item(2) 'Second cell in bigger Range.
    Set r = ws.Range("A1:A5")(2) 'Second cell in bigger Range.
End Sub
```

Beachten Sie im Beispiel, dass Zellen (2, 1) dem Bereich ("A2") entspricht. Dies liegt daran, dass Cells ein Range-Objekt zurückgibt.
Einige Quellen: Chip Pearson-Cells in Ranges ; MSDN-Bereichsobjekt ; John Walkenback-Verweise auf Bereiche in Ihrem VBA-Code .

Beachten Sie außerdem, dass Sie in allen Fällen, in denen eine Nummer in der Deklaration des Bereichs verwendet wird und die Nummer selbst außerhalb von Anführungszeichen steht (z. B. Range ("A" & 2)), diese Nummer gegen eine Variable austauschen können, die ein Ganzzahl / lang. Zum Beispiel:

```
Sub RangeIteration()
    Dim wb As Workbook, ws As Worksheet
    Dim r As Range

    Set wb = ThisWorkbook
    Set ws = wb.Worksheets(1)

    For i = 1 To 10
        Set r = ws.Range("A" & i)
        ' When i = 1, the result will be Range("A1")
        ' When i = 2, the result will be Range("A2")
        ' etc.
        ' Proof:
        Debug.Print r.Address
    Next i
End Sub
```

Wenn Sie Doppelschleifen verwenden, ist Cells besser:

```
Sub RangeIteration2()
    Dim wb As Workbook, ws As Worksheet
    Dim r As Range

    Set wb = ThisWorkbook
    Set ws = wb.Worksheets(1)

    For i = 1 To 10
        For j = 1 To 10
            Set r = ws.Cells(i, j)
            ' When i = 1 and j = 1, the result will be Range("A1")
            ' When i = 2 and j = 1, the result will be Range("A2")
```

```
            ' When i = 1 and j = 2, the result will be Range("B1")
            ' etc.
            ' Proof:
            Debug.Print r.Address
        Next j
    Next i
End Sub
```

Möglichkeiten, sich auf eine einzelne Zelle zu beziehen

Der einfachste Weg, auf eine einzelne Zelle im aktuellen Excel-Arbeitsblatt zu verweisen, besteht einfach darin, die A1-Form ihrer Referenz in eckige Klammern zu setzen:

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

Beachten Sie, dass eckige Klammern nur zweckmäßiger syntaktischer Zucker für die `Evaluate` Methode des `Application` Objekts sind. Technisch ist dies mit dem folgenden Code identisch:

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

Sie können auch die `Cells` Methode aufrufen, die eine Zeile und eine Spalte übernimmt und eine Zellreferenz zurückgibt.

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

Denken Sie daran, dass immer, wenn Sie von VBA aus eine Zeile und eine Spalte an Excel übergeben, die Zeile immer zuerst steht, gefolgt von der Spalte. Dies ist verwirrend, da die Spalte zuerst in der üblichen `A1` Notation steht.

In beiden Beispielen haben wir kein Arbeitsblatt angegeben, daher verwendet Excel das aktive Blatt (das Blatt, das sich in der Benutzeroberfläche befindet). Sie können das aktive Blatt explizit angeben:

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

Oder Sie können den Namen eines bestimmten Blattes angeben:

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

Es gibt eine Vielzahl von Methoden, um von einem Bereich zum anderen zu gelangen. Beispielsweise kann die `Rows` Methode verwendet werden, um zu den einzelnen Zeilen eines beliebigen Bereichs zu gelangen, und die `Cells` Methode kann verwendet werden, um zu einzelnen Zellen einer Zeile oder Spalte zu gelangen. Daher bezieht sich der folgende Code auf Zelle C1:

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

Speichern eines Verweises auf eine Zelle in einer Variablen

Um einen Verweis auf eine Zelle in einer Variablen zu speichern, müssen Sie die `Set` Syntax verwenden. Beispiel:

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

später...

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

Warum ist das `Set` Schlüsselwort erforderlich? `Set` teilt Visual Basic mit, dass der Wert auf der rechten Seite von `=` ein Objekt sein soll.

Offset-Eigenschaft

- **Offset (Zeilen, Spalten)** - Der Operator, mit dem ein anderer Punkt der aktuellen Zelle statisch referenziert wird. Wird häufig in Schleifen verwendet. Es versteht sich, dass sich positive Zahlen im Zeilenabschnitt nach rechts bewegen, während Negative sich nach links bewegen. Bei den Spalten bewegen sich die Positiven nach unten und die Negativen nach oben.

dh

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

Dieser Code wählt B2 aus, fügt dort eine neue Zeichenfolge ein und verschiebt die Zeichenfolge anschließend wieder nach A1 und löscht anschließend B2.

Transponieren von Bereichen (horizontal in vertikal und umgekehrt)

```
Sub TransposeRangeValues()
    Dim TmpArray() As Variant, FromRange as Range, ToRange as Range

    set FromRange = Sheets("Sheet1").Range("a1:a12")         'Worksheets(1).Range("a1:p1")
    set ToRange = ThisWorkbook.Sheets("Sheet1").Range("a1")
'ThisWorkbook.Sheets("Sheet1").Range("a1")

    TmpArray = Application.Transpose(FromRange.Value)
    FromRange.Clear
    ToRange.Resize(FromRange.Columns.Count,FromRange.Rows.Count).Value2 = TmpArray
End Sub
```

Hinweis: Copy / PasteSpecial enthält auch die Option Transpose einfügen, mit der auch die

Formeln der transponierten Zellen aktualisiert werden.

Bereiche und Zellen online lesen: https://riptutorial.com/de/excel-vba/topic/1503/bereiche-und-zellen

Kapitel 12: Bindung

Examples

Early Binding vs. Late Binding

Bindung ist der Vorgang, bei dem ein Objekt einem Bezeichner oder Variablennamen zugewiesen wird. Eine frühe Bindung (auch als statische Bindung bezeichnet) liegt vor, wenn ein in Excel deklariertes Objekt einen bestimmten Objekttyp aufweist, beispielsweise ein Arbeitsblatt oder eine Arbeitsmappe. Eine späte Bindung tritt auf, wenn allgemeine Objektzuordnungen vorgenommen werden, z. B. die Deklarationstypen Object und Variant.

Frühes Binden von Referenzen einige Vorteile gegenüber dem späten Binden.

- Die frühe Bindung ist betriebsbereit schneller als die späte Bindung während der Laufzeit. Das Erstellen des Objekts mit einer späten Bindung zur Laufzeit erfordert Zeit, die durch die frühe Bindung beim ersten Laden des VBA-Projekts erreicht wird.
- Die frühe Bindung bietet zusätzliche Funktionen durch die Identifizierung von Schlüssel- / Artikelpaaren anhand ihrer Ordinalposition.
- Je nach Codestruktur kann das frühe Binden eine zusätzliche Ebene der Typprüfung bieten und Fehler reduzieren.
- Die Kapitalisierungskorrektur der VBE beim Eintippen der Eigenschaften und Methoden eines gebundenen Objekts ist mit einer frühen Bindung aktiv, jedoch mit einer späten Bindung nicht verfügbar.

  Hinweis: Sie müssen dem VBA-Projekt die entsprechende Referenz über den Befehl Tools → References der VBE hinzufügen, um eine frühe Bindung zu implementieren.

  Diese Bibliotheksreferenz wird dann mit dem Projekt mitgeführt; Es muss nicht erneut referenziert werden, wenn das VBA-Projekt auf einem anderen Computer verteilt und ausgeführt wird.

```
'Looping through a dictionary that was created with late binding¹
Sub iterateDictionaryLate()
    Dim k As Variant, dict As Object

    Set dict = CreateObject("Scripting.Dictionary")
    dict.comparemode = vbTextCompare          'non-case sensitive compare model

    'populate the dictionary
    dict.Add Key:="Red", Item:="Balloon"
    dict.Add Key:="Green", Item:="Balloon"
    dict.Add Key:="Blue", Item:="Balloon"

    'iterate through the keys
    For Each k In dict.Keys
        Debug.Print k & " - " & dict.Item(k)
    Next k

    dict.Remove "blue"      'remove individual key/item pair by key
    dict.RemoveAll          'remove all remaining key/item pairs

End Sub
```

```
'Looping through a dictionary that was created with early binding¹
Sub iterateDictionaryEarly()
    Dim d As Long, k As Variant
    Dim dict As New Scripting.Dictionary

    dict.CompareMode = vbTextCompare          'non-case sensitive compare model

    'populate the dictionary
    dict.Add Key:="Red", Item:="Balloon"
    dict.Add Key:="Green", Item:="Balloon"
    dict.Add Key:="Blue", Item:="Balloon"
    dict.Add Key:="White", Item:="Balloon"

    'iterate through the keys
    For Each k In dict.Keys
        Debug.Print k & " - " & dict.Item(k)
    Next k

    'iterate through the keys by the count
    For d = 0 To dict.Count - 1
        Debug.Print dict.Keys(d) & " - " & dict.Items(d)
    Next d

    'iterate through the keys by the boundaries of the keys collection
    For d = LBound(dict.Keys) To UBound(dict.Keys)
        Debug.Print dict.Keys(d) & " - " & dict.Items(d)
    Next d

    dict.Remove "blue"                          'remove individual key/item pair by key
    dict.Remove dict.Keys(0)                    'remove first key/item by index position
    dict.Remove dict.Keys(UBound(dict.Keys))    'remove last key/item by index position
    dict.RemoveAll                              'remove all remaining key/item pairs

End Sub
```

Wenn Sie jedoch die vorzeitige Bindung verwenden und das Dokument auf einem System ausgeführt wird, dem eine der Bibliotheken fehlt, auf die Sie verwiesen haben, treten Probleme auf. Die Routinen, die die fehlende Bibliothek verwenden, funktionieren nicht nur nicht ordnungsgemäß, sondern das Verhalten des gesamten Codes innerhalb des Dokuments wird unregelmäßig. Es ist wahrscheinlich, dass auf dem Computer kein Code des Dokuments funktioniert.

Hier ist spätes Binden von Vorteil. Wenn Sie das Late Binding verwenden, müssen Sie den Verweis nicht im Menü Extras> Verweise hinzufügen. Auf Computern, die über die entsprechende Bibliothek verfügen, funktioniert der Code weiterhin. Auf Computern ohne diese Bibliothek funktionieren die Befehle, die auf die Bibliothek verweisen, nicht, aber der gesamte andere Code in Ihrem Dokument funktioniert weiterhin.

Wenn Sie mit der Bibliothek, auf die Sie verweisen, nicht genau vertraut sind, kann es hilfreich sein, die frühe Bindung beim Schreiben des Codes zu verwenden und dann vor der Bereitstellung auf die späte Bindung umzustellen. Auf diese Weise können Sie während der Entwicklung den IntelliSense und den Objektbrowser der VBE nutzen.

Bindung online lesen: https://riptutorial.com/de/excel-vba/topic/3811/bindung

Kapitel 13: CustomDocumentProperties in der Praxis

Einführung

Die Verwendung von CustomDocumentProperties (CDPs) ist eine gute Methode, um benutzerdefinierte Werte relativ sicher im selben Arbeitsbuch zu speichern, wobei jedoch vermieden wird, verwandte Zellenwerte einfach in einem ungeschützten Arbeitsblatt anzuzeigen *).

Hinweis: CDPs stellen eine separate Collection dar, die mit BuiltInDocumentProperties vergleichbar ist. Sie können jedoch benutzerdefinierte Eigenschaftsnamen anstelle einer festen Collection erstellen.

*) Alternativ können Sie Werte auch in einer verborgenen oder "sehr versteckten" Arbeitsmappe eingeben.

Examples

Neue Rechnungsnummern organisieren

Das Erhöhen einer Rechnungsnummer und das Speichern ihres Wertes ist eine häufige Aufgabe. Die Verwendung von CustomDocumentProperties (CDPs) ist eine gute Methode, um solche Zahlen auf relativ sichere Weise innerhalb desselben Arbeitsbuchs zu speichern, wobei jedoch vermieden wird, verwandte Zellwerte einfach in einem ungeschützten Arbeitsblatt anzuzeigen.

Zusätzlicher Hinweis:

Alternativ können Sie Werte auch in ein verstecktes Arbeitsblatt oder sogar in ein sogenanntes "very hidden" Arbeitsblatt eingeben (siehe Verwenden von xlVeryHidden Sheets . Selbstverständlich können Sie Daten auch in externen Dateien speichern (z. B. ini-Datei, csv oder einem anderen Typ) oder die Registry.

Beispielinhalt :

Das Beispiel unten zeigt

- eine Funktion NextInvoiceNo, die die nächste Rechnungsnummer einstellt und zurückgibt,
- eine Prozedur DeleteInvoiceNo, die den Rechnungs-CDP vollständig löscht, sowie
- Eine Prozedur showAllCDPs, in der die gesamte CDPs-Sammlung mit allen Namen aufgeführt ist. Wenn Sie VBA nicht verwenden, können Sie sie auch über die Informationen der Arbeitsmappe auflisten: Info | Eigenschaften [DropDown:] | Erweiterte Eigenschaften | Brauch

Sie können die nächste Rechnungsnummer (die letzte keine plus eine) erhalten und festlegen,

indem Sie einfach die oben genannte Funktion aufrufen und einen Zeichenfolgenwert zurückgeben, um das Hinzufügen von Präfixen zu erleichtern. "InvoiceNo" wird in allen Verfahren implizit als CDP-Name verwendet.

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

Beispielcode:

```
Option Explicit

Sub Test()
  Dim sNumber As String
  sNumber = NextInvoiceNo()
  MsgBox "New Invoice No: " & sNumber, vbInformation, "New Invoice Number"
End Sub

Function NextInvoiceNo() As String
' Purpose: a) Set Custom Document Property (CDP) "InvoiceNo" if not yet existing
'          b) Increment CDP value and return new value as string
' Declarations
  Dim prop As Object
  Dim ret  As String
  Dim wb   As Workbook
' Set workbook and CDPs
  Set wb = ThisWorkbook
  Set prop = wb.CustomDocumentProperties

  ' ------------------------------------------------------
  ' Generate new CDP "InvoiceNo" if not yet existing
  ' ------------------------------------------------------
    If Not CDPExists("InvoiceNo") Then
    '  set temporary starting value "0"
       prop.Add "InvoiceNo", False, msoPropertyTypeString, "0"
    End If

  ' ------------------------------------------------------
  ' Increment invoice no and return function value as string
  ' ------------------------------------------------------
       ret = Format(Val(prop("InvoiceNo")) + 1, "0")
  ' a) Set CDP "InvoiceNo" = ret
       prop("InvoiceNo").value = ret
  ' b) Return function value
       NextInvoiceNo = ret
End Function

Private Function CDPExists(sCDPName As String) As Boolean
' Purpose: return True if custom document property (CDP) exists
' Method: loop thru CustomDocumentProperties collection and check if name parameter exists
' Site: cf. http://stackoverflow.com/questions/23917977/alternatives-to-public-variables-in-
vba/23918236#23918236
' vgl.: https://answers.microsoft.com/en-us/msoffice/forum/msoffice_word-mso_other/using-
customdocumentproperties-with-vba/91ef15eb-b089-4c9b-a8a7-1685d073fb9f
' Declarations
  Dim cdp As Variant      ' element of CustomDocumentProperties Collection
  Dim boo As Boolean      ' boolean value showing element exists
  For Each cdp In ThisWorkbook.CustomDocumentProperties
    If LCase(cdp.Name) = LCase(sCDPName) Then
       boo = True      ' heureka
```

```
       Exit For        ' exit loop
    End If
  Next
  CDPExists = boo          ' return value to function
End Function


Sub DeleteInvoiceNo()
' Declarations
  Dim wb     As Workbook
  Dim prop   As Object
' Set workbook and CDPs
  Set wb = ThisWorkbook
  Set prop = wb.CustomDocumentProperties

' ----------------------
' Delete CDP "InvoiceNo"
' ----------------------
 If CDPExists("InvoiceNo") Then
    prop("InvoiceNo").Delete
 End If
```

End Sub

```
Sub showAllCDPs()
' Purpose: Show all CustomDocumentProperties (CDP) and values (if set)
' Declarations
  Dim wb      As Workbook
  Dim cdp     As Object

  Dim i       As Integer
  Dim maxi    As Integer
  Dim s       As String
' Set workbook and CDPs
  Set wb = ThisWorkbook
  Set cdp = wb.CustomDocumentProperties
' Loop thru CDP getting name and value
  maxi = cdp.Count
  For i = 1 To maxi
    On Error Resume Next    ' necessary in case of unset value
    s = s & Chr(i + 96) & ") " & _
            cdp(i).Name & "=" & cdp(i).value & vbCr
  Next i
' Show result string
  Debug.Print s
End Sub
```

CustomDocumentProperties in der Praxis online lesen: https://riptutorial.com/de/excel-vba/topic/10932/customdocumentproperties-in-der-praxis

Kapitel 14: Dateisystemobjekt

Examples

Datei, Ordner, Laufwerk ist vorhanden

Datei existiert:

```
Sub FileExists()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    If fso.FileExists("D:\test.txt") = True Then
       MsgBox "The file is exists."
    Else
       MsgBox "The file isn't exists."
    End If
End Sub
```

Ordner existiert:

```
Sub FolderExists()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    If fso.FolderExists("D:\testFolder") = True Then
       MsgBox "The folder is exists."
    Else
       MsgBox "The folder isn't exists."
    End If
End Sub
```

Laufwerk existiert:

```
Sub DriveExists()
    Dim fso as Scripting.FileSystemObject
    Set fso = CreateObject("Scripting.FileSystemObject")
    If fso.DriveExists("D:\") = True Then
       MsgBox "The drive is exists."
    Else
       MsgBox "The drive isn't exists."
    End If
End Sub
```

Grundlegende Dateivorgänge

Kopieren:

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

Bewegung:

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

Löschen:

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

Grundlegende Ordnervorgänge

Erstellen:

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

Kopieren:

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

Bewegung:

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

Löschen:

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

Andere Operationen

Dateiname abrufen:

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

Ergebnis: *Makro.txt*

Basisnamen erhalten:

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

Ergebnis: *Makro*

Erweiterungsname abrufen:

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

```
End Sub
```

Ergebnis: *txt*

Laufwerksname abrufen:

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

Ergebnis: *c:*

Dateisystemobjekt online lesen: https://riptutorial.com/de/excel-vba/topic/9933/dateisystemobjekt

Kapitel 15: Debugging und Fehlerbehebung

Syntax

- Debug.Print (Zeichenfolge)
- Halt halt

Examples

Debug.Print

Um eine Liste der Fehlercode-Beschreibungen an das `Debug.Print` zu drucken, übergeben Sie sie an die `Debug.Print` Funktion:

```
Private Sub ListErrCodes()
    Debug.Print "List Error Code Descriptions"
    For i = 0 To 65535
        e = Error(i)
        If e <> "Application-defined or object-defined error" Then Debug.Print i & ": " & e
    Next i
End Sub
```

Sie können das Direktfenster anzeigen, indem Sie:

- Auswählen von **V** iew | **Ich** verwende Fenster aus der Menüleiste
- Verwenden der Tastenkombination **Strg-G**

Halt

Der Befehl Stop unterbricht die Ausführung, wenn er aufgerufen wird. Von dort kann der Prozess fortgesetzt oder schrittweise ausgeführt werden.

```
Sub Test()
    Dim TestVar as String
    TestVar = "Hello World"
    Stop                    'Sub will be executed to this point and then wait for the user
    MsgBox TestVar
End Sub
```

Sofortiges Fenster

Wenn Sie eine Zeile mit Makrocode testen möchten, ohne ein komplettes Sub-Element ausführen zu müssen, können Sie Befehle direkt in das `ENTER` eingeben und die `ENTER` , um die Zeile auszuführen.

Um die Ausgabe einer Zeile zu testen, können Sie ihr ein Fragezeichen voranstellen `?` um direkt in das Direktfenster zu drucken. Alternativ können Sie auch den `print` , um die Ausgabe drucken zu

lassen.

Drücken Sie in dem Visual Basic-Editor `CTRL + G` , um das `CTRL + G` zu öffnen. So benennen Sie Ihre aktuell ausgewählte Blatt „ExampleSheet“, geben Sie Folgendes in das Direktfenster ein und drücken `ENTER`

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

So drucken Sie den Namen des aktuell ausgewählten Blatts direkt im Direktfenster

```
? ActiveSheet.Name
ExampleSheet
```

Diese Methode kann sehr nützlich sein, um die Funktionalität von integrierten oder benutzerdefinierten Funktionen zu testen, bevor sie in Code implementiert werden. Das folgende Beispiel veranschaulicht, wie das Direktfenster verwendet werden kann, um die Ausgabe einer Funktion oder einer Reihe von Funktionen zu testen, um ein erwartetes Ergebnis zu bestätigen.

```
'In this example, the Immediate Window was used to confirm that a series of Left and Right
'string methods would return the desired string

'expected output: "value"
print Left(Right("1111value1111",9),5) ' <---- written code here, ENTER pressed
value                                  ' <---- output
```

Das Direktfenster kann auch zum Festlegen oder Zurücksetzen von Application, Workbook oder anderen erforderlichen Eigenschaften verwendet werden. Dies kann nützlich sein, wenn sich in einer Subroutine `Application.EnableEvents = False` , die unerwartet einen Fehler auslöst, ohne dass der Wert auf `True` (was frustrierende und unerwartete Funktionen verursachen kann. In diesem Fall können die Befehle direkt eingegeben werden in das Direktfenster und starte:

```
? Application.EnableEvents       ' <---- Testing the current state of "EnableEvents"
False                            ' <---- Output
Application.EnableEvents = True  ' <---- Resetting the property value to True
? Application.EnableEvents       ' <---- Testing the current state of "EnableEvents"
True                             ' <---- Output
```

Für fortgeschrittenere Debugging-Techniken kann ein Doppelpunkt `:` als Trennzeichen verwendet werden. Dies kann für mehrzeilige Ausdrücke verwendet werden, z. B. für das Schleifen im folgenden Beispiel.

```
x = Split("a,b,c",","): For i = LBound(x,1) to UBound(x,1): Debug.Print x(i): Next i '<----
Input this and press enter
a '<----Output
b '<----Output
c '<----Output
```

Verwenden Sie den Timer, um Engpässe in der Leistung zu finden

Der erste Schritt bei der Optimierung der Geschwindigkeit besteht darin, die langsamsten

Codeabschnitte zu finden. Die `Timer` VBA-Funktion gibt die Anzahl der seit Mitternacht verstrichenen Sekunden mit einer Genauigkeit von 1/56 Sekunde (3,90625 Millisekunden) auf Windows-basierten PCs zurück. Die VBA-Funktionen `Now` und `Time` sind nur auf eine Sekunde genau.

```
Dim start As Double       ' Timer returns Single, but converting to Double to avoid
start = Timer             ' scientific notation like 3.90625E-03 in the Immediate window
' ... part of the code
Debug.Print Timer - start; "seconds in part 1"

start = Timer
' ... another part of the code
Debug.Print Timer - start; "seconds in part 2"
```

Einen Haltepunkt zu Ihrem Code hinzufügen

Sie können Ihrem Code einfach einen Haltepunkt hinzufügen, indem Sie auf die graue Spalte links neben der Zeile Ihres VBA-Codes klicken, an der die Ausführung angehalten werden soll. In der Spalte wird ein roter Punkt angezeigt, und der Haltepunktcode wird ebenfalls rot hervorgehoben.

Sie können im gesamten Code mehrere Haltepunkte hinzufügen. Die Fortsetzung der Ausführung wird durch Drücken des Symbols "Wiedergabe" in der Menüleiste erreicht. Nicht jeder Code kann ein Haltepunkt als Variablendefinitionszeile sein, die erste oder letzte Zeile einer Prozedur und Kommentarzeilen können nicht als Haltepunkt ausgewählt werden.

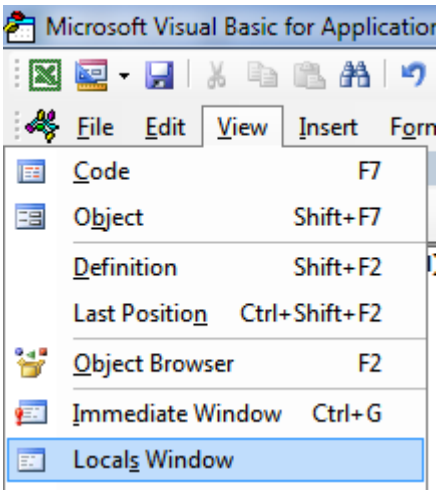
Debugger-Fenster "Locals"

Das Fenster "Locals" bietet einen einfachen Zugriff auf den aktuellen Wert von Variablen und Objekten im Rahmen der Funktion oder Subroutine, die Sie ausführen. Es ist ein unverzichtbares Werkzeug, um Ihren Code zu debuggen und Änderungen durchzugehen, um Probleme zu finden. Sie können damit auch Eigenschaften erkunden, die Sie möglicherweise nicht kennen.

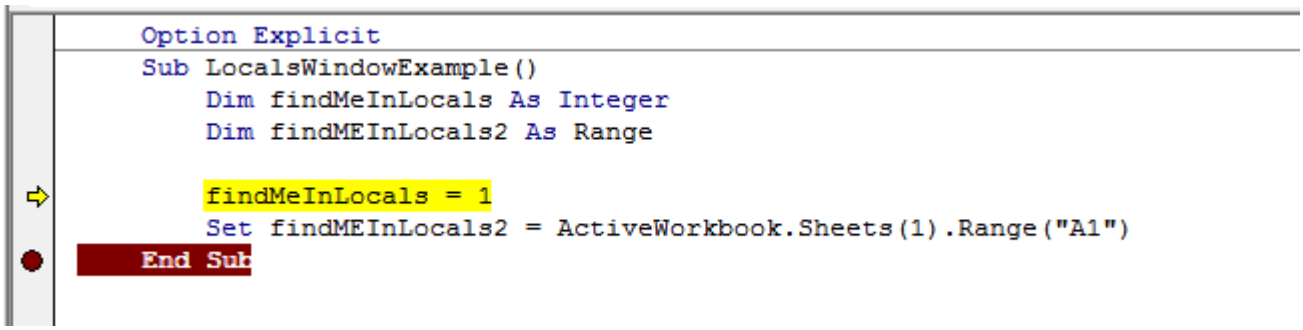
Nehmen Sie das folgende Beispiel:

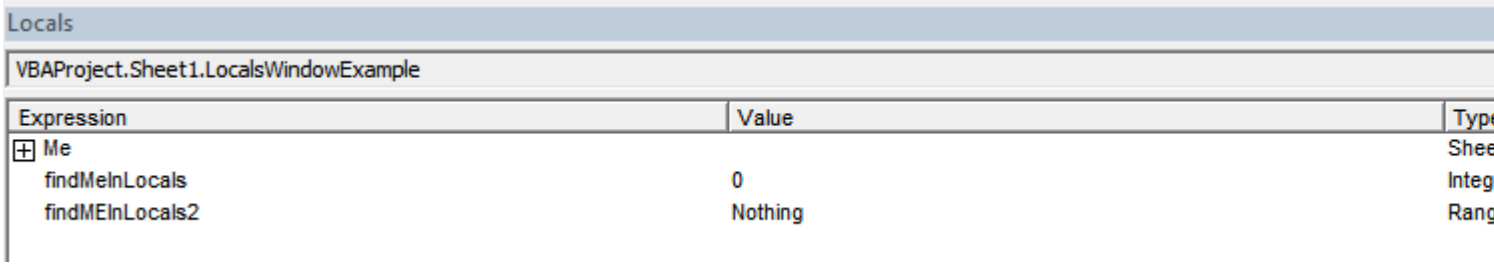
```
Option Explicit
Sub LocalsWindowExample()
    Dim findMeInLocals As Integer
    Dim findMEInLocals2 As Range

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

Klicken Sie im VBA-Editor auf Ansicht -> Lokalfenster



Nachdem wir den Code mit F8 durchlaufen haben, nachdem wir in die Subroutine geklickt haben, haben wir angehalten, bevor wir findMeinLocals zuweisen konnten. Unten sehen Sie den Wert 0 - und dies würde verwendet werden, wenn Sie ihm niemals einen Wert zugewiesen hätten. Das Bereichsobjekt lautet 'Nothing'.

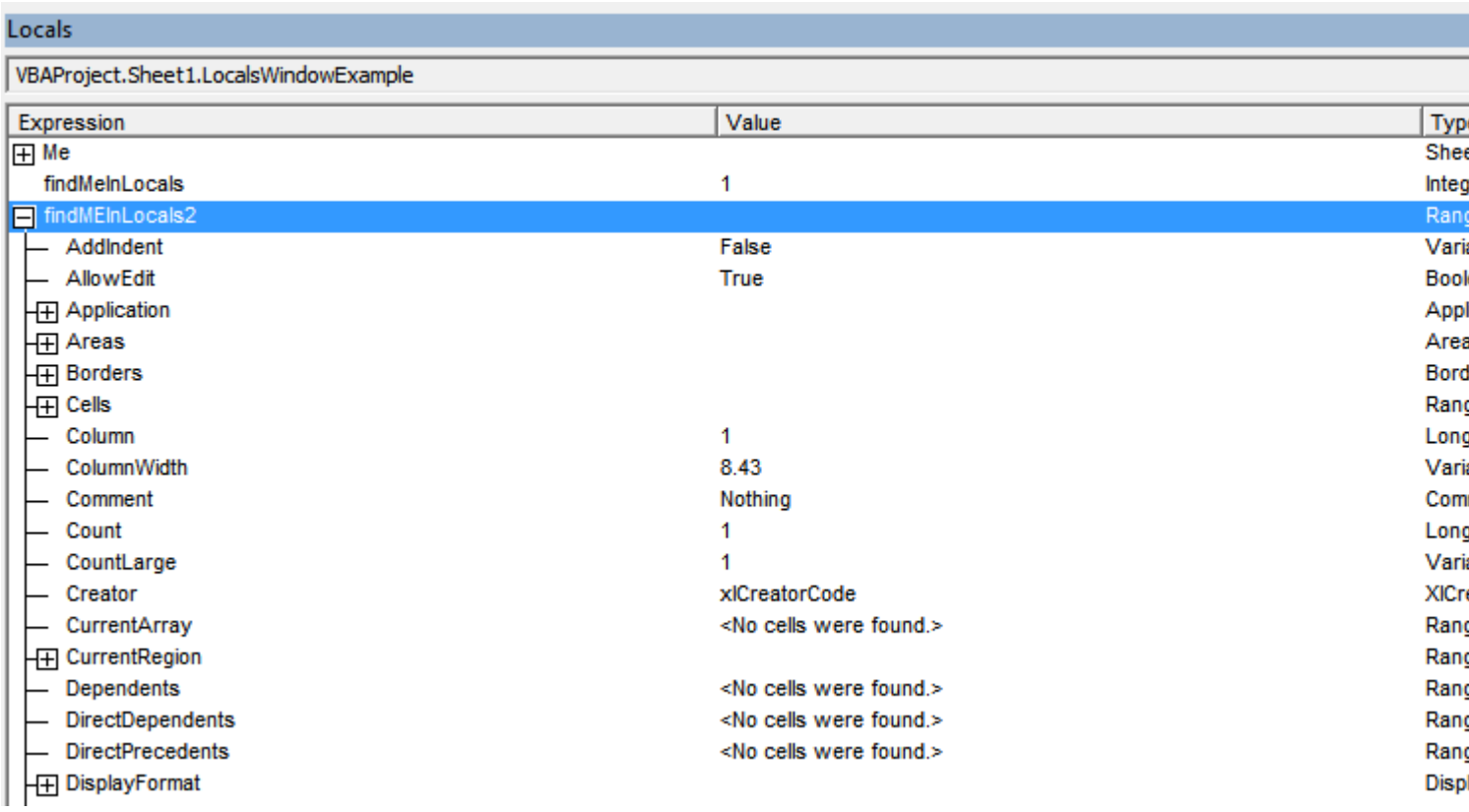




Wenn wir kurz vor dem Ende der Subroutine anhalten, können wir die endgültigen Werte der Variablen sehen.

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

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

Wir können findMeInLocals mit dem Wert 1 und den Typ Integer sowie FindMeInLocals2 mit einem Typ von Range / Range sehen. Wenn Sie auf das Pluszeichen klicken, können Sie das Objekt erweitern und seine Eigenschaften anzeigen, wie z. B. Anzahl oder Spalte.



Debugging und Fehlerbehebung online lesen: https://riptutorial.com/de/excel-vba/topic/861/debugging-und-fehlerbehebung

Kapitel 16: Diagramme und Diagramme

Examples

Erstellen eines Diagramms mit Bereichen und einem festen Namen

Diagramme können erstellt werden, indem direkt mit dem `Series` Objekt gearbeitet wird, das die Diagrammdaten definiert. Um ohne ein vorhandenes Diagramm zur `Series` zu gelangen, erstellen Sie ein `ChartObject` in einem bestimmten `Worksheet` und `ChartObject` dann das `Chart` Objekt ab. Die Arbeit mit dem `Series` Objekt hat den `XValues` dass Sie die `Values` und `XValues` `Values` `XValues` indem Sie auf `Range` Objekte verweisen. Diese Dateneigenschaften definieren die `Series` ordnungsgemäß mit Verweisen auf diese Bereiche. Der Nachteil dieses Ansatzes ist, dass beim Festlegen des `Name` nicht dieselbe Konvertierung ausgeführt wird. es ist ein fester Wert. Es wird nicht mit den zugrunde liegenden Daten im ursprünglichen Bereich `Range` . Wenn Sie die `SERIES` Formel `SERIES` , ist es offensichtlich, dass der Name festgelegt ist. Dies muss durch direktes Erstellen der `SERIES` Formel erfolgen.

Code zum Erstellen eines Diagramms

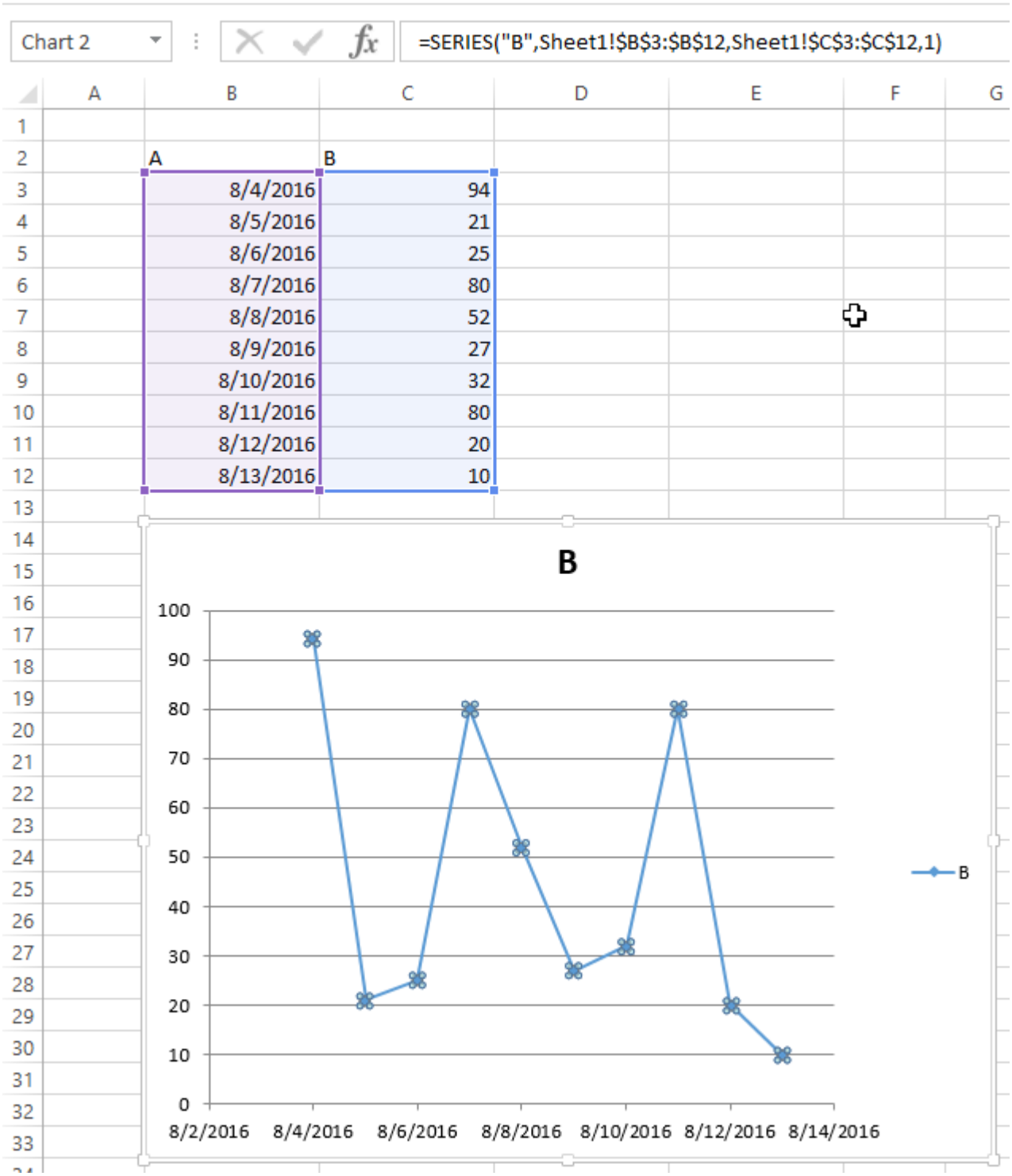
Beachten Sie, dass dieser Code zusätzliche Variablendeklarationen für das `Chart` und das `Worksheet` . Diese können weggelassen werden, wenn sie nicht verwendet werden. Sie können jedoch nützlich sein, wenn Sie den Stil oder andere Diagrammeigenschaften ändern.

```
Sub CreateChartWithRangesAndFixedName()

    Dim xData As Range
    Dim yData As Range
    Dim serName As Range

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

    'get reference to ActiveSheet
    Dim sht As Worksheet
    Set sht = ActiveSheet

    'create a new ChartObject at position (48, 195) with width 400 and height 300
    Dim chtObj As ChartObject
    Set chtObj = sht.ChartObjects.Add(48, 195, 400, 300)

    'get reference to chart object
    Dim cht As Chart
    Set cht = chtObj.Chart

    'create the new series
    Dim ser As Series
    Set ser = cht.SeriesCollection.NewSeries

    ser.Values = yData
    ser.XValues = xData
```

```
    ser.Name = serName

    ser.ChartType = xlXYScatterLines

End Sub
```

Ursprüngliche Daten / Bereiche und resultierendes `Chart` nach Code-Ausführung

Beachten Sie, dass die `SERIES` Formel anstelle des Verweises auf den `Range` , in dem sie erstellt wurde, ein `"B"` für den `SERIES` enthält.



Erstellen eines leeren Diagramms

Der Ausgangspunkt für die überwiegende Mehrheit des Diagrammcodes ist das Erstellen eines leeren `Chart` . Beachten Sie, dass dieses `Chart` der aktiven Standardvorlagenvorlage unterliegt und möglicherweise nicht leer ist (wenn die Vorlage geändert wurde).

Der Schlüssel für das `ChartObject` ist das Bestimmen seines Standorts. Die Syntax für den Aufruf lautet `ChartObjects.Add(Left, Top, Width, Height)` . Nachdem das `ChartObject` erstellt wurde, können Sie das `Chart` Objekt verwenden, um das Diagramm tatsächlich zu ändern. Das `ChartObject` verhält sich eher wie eine `Shape` um das Diagramm auf dem `ChartObject` zu positionieren.

Code zum Erstellen eines leeren Diagramms

```
Sub CreateEmptyChart()

    'get reference to ActiveSheet
    Dim sht As Worksheet
    Set sht = ActiveSheet

    'create a new ChartObject at position (0, 0) with width 400 and height 300
    Dim chtObj As ChartObject
    Set chtObj = sht.ChartObjects.Add(0, 0, 400, 300)

    'get refernce to chart object
    Dim cht As Chart
    Set cht = chtObj.Chart

    'additional code to modify the empty chart
    '...

End Sub
```

Ergebnisdiagramm

Erstellen Sie ein Diagramm, indem Sie die SERIES-Formel ändern

Für die vollständige Kontrolle über ein neues `Chart` und `Series` - Objekt (vor allem für einen dynamischen `Series` müssen Sie die zu modifizierenden greifen `SERIES` Formel direkt. Der Vorgang zum Einrichten der `Range` Objekte ist unkompliziert und die Haupthürde besteht einfach aus der Zeichenfolge für die `SERIES` Formel.

Die `SERIES` Formel verwendet die folgende Syntax:

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

Diese Inhalte können als Referenzen oder als Array-Werte für die Datenelemente bereitgestellt werden. `Order` für die Reihenposition innerhalb des Diagramms. Beachten Sie, dass die Verweise auf die Daten nur funktionieren, wenn sie mit dem Blattnamen vollständig qualifiziert sind. Klicken Sie für ein Beispiel einer Arbeitsformel auf eine vorhandene Serie, und überprüfen Sie die Formelleiste.

Code zum Erstellen eines Diagramms und Einrichten von Daten mithilfe der `SERIES` Formel

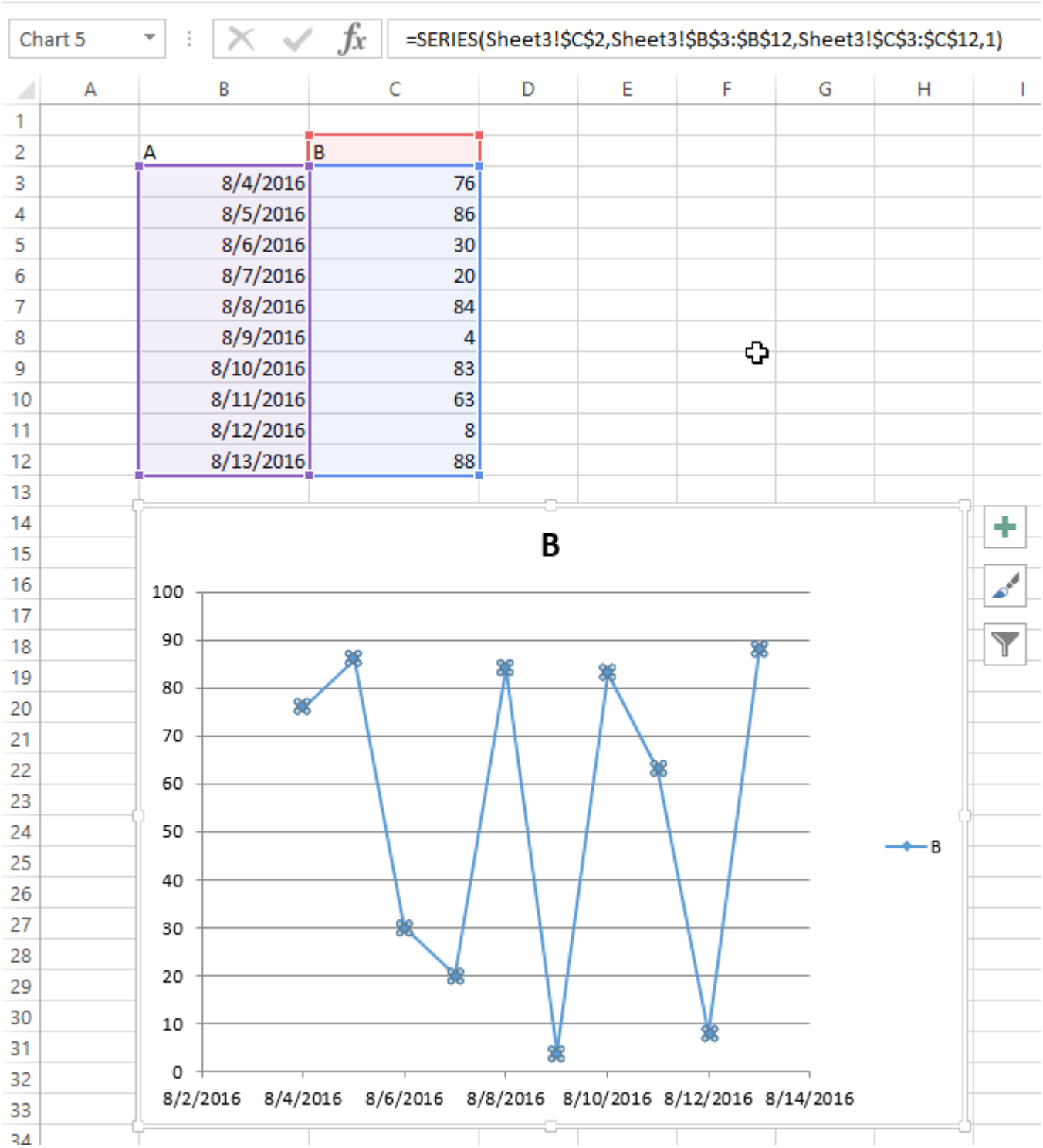
Beachten Sie, dass beim Erstellen der Zeichenfolge zur Erstellung der `SERIES` Formel `.Address(,,,True)` . Dadurch wird sichergestellt, dass die *externe* Bereichsreferenz verwendet wird, sodass eine vollständig qualifizierte Adresse mit dem Blattnamen angegeben wird. Sie **erhalten eine Fehlermeldung, wenn der Blattname ausgeschlossen ist** .

```
Sub CreateChartUsingSeriesFormula()
```

```
    Dim xData As Range
    Dim yData As Range
    Dim serName As Range

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

    'get reference to ActiveSheet
    Dim sht As Worksheet
    Set sht = ActiveSheet

    'create a new ChartObject at position (48, 195) with width 400 and height 300
    Dim chtObj As ChartObject
    Set chtObj = sht.ChartObjects.Add(48, 195, 400, 300)

    'get refernce to chart object
    Dim cht As Chart
    Set cht = chtObj.Chart

    'create the new series
    Dim ser As Series
    Set ser = cht.SeriesCollection.NewSeries

    'set the SERIES formula
    '=SERIES(name, xData, yData, plotOrder)

    Dim formulaValue As String
    formulaValue = "=SERIES(" & _
        serName.Address(, , , True) & "," & _
        xData.Address(, , , True) & "," & _
        yData.Address(, , , True) & ",1)"

    ser.Formula = formulaValue
    ser.ChartType = xlXYScatterLines

End Sub
```

Originaldaten und Ergebnisdiagramm

Beachten Sie, dass für dieses Diagramm der Serienname richtig mit einem Bereich zur gewünschten Zelle festgelegt ist. Dies bedeutet, dass Aktualisierungen auf das `Chart` .



Charts in einem Raster anordnen

Eine übliche Aufgabe bei Diagrammen in Excel ist die Standardisierung der Größe und des Layouts mehrerer Diagramme auf einem einzelnen Blatt. Wenn Sie dies manuell tun, können Sie die `ALT-` Taste gedrückt halten, während Sie die Größe des Diagramms ändern oder verschieben, um an den Zellgrenzen zu bleiben. Dies funktioniert für ein paar Diagramme, aber ein VBA-Ansatz ist viel einfacher.

Code zum Erstellen eines Gitters

Mit diesem Code wird ein Diagrammraster erstellt, das an einer bestimmten Position (oben, links)

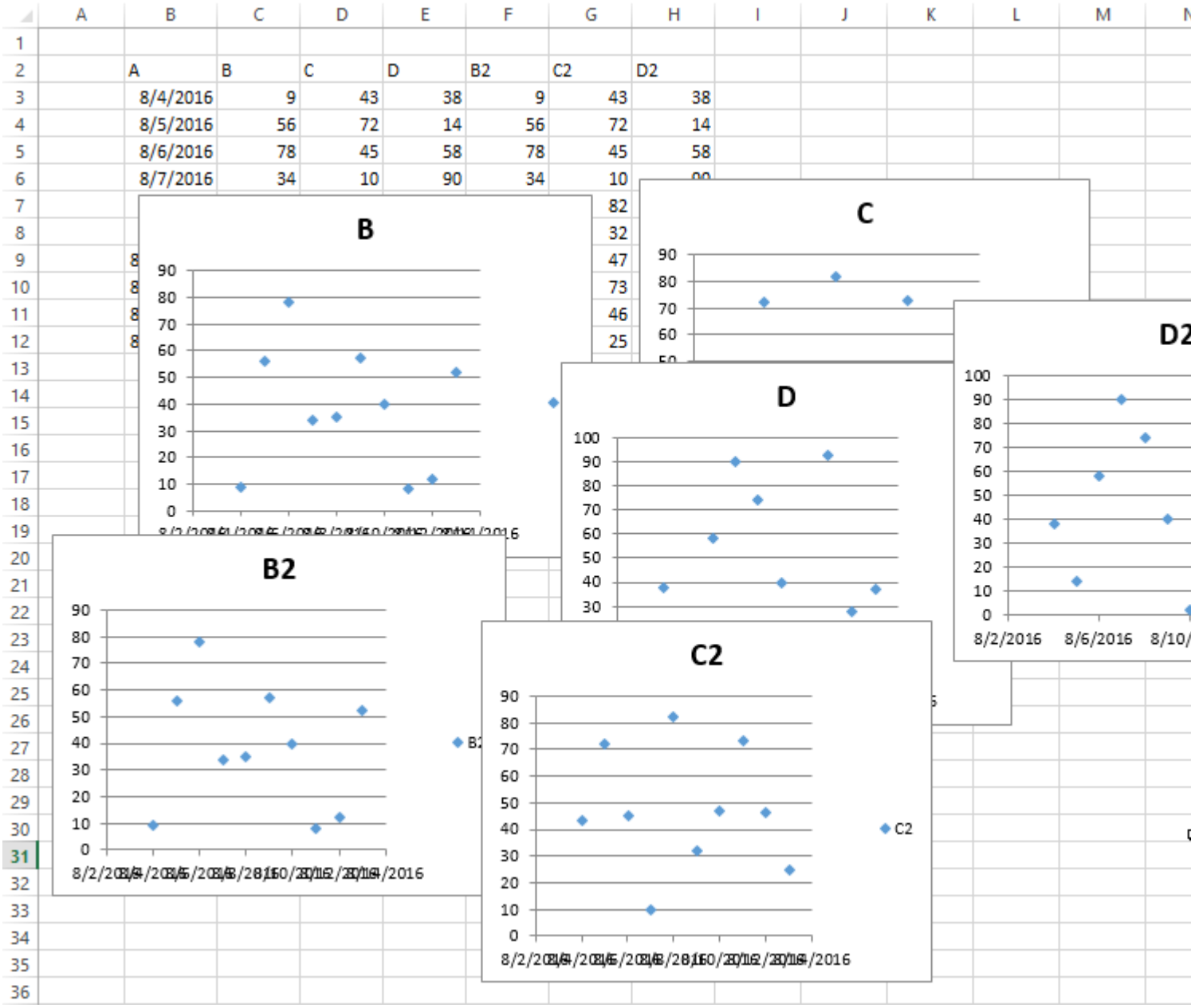
mit einer definierten Anzahl von Spalten und einer definierten allgemeinen Diagrammgröße beginnt. Die Diagramme werden in der Reihenfolge platziert, in der sie erstellt wurden, und werden um den Rand herumgelegt, um eine neue Zeile zu bilden.

```
Sub CreateGridOfCharts()

    Dim int_cols As Integer
    int_cols = 3

    Dim cht_width As Double
    cht_width = 250

    Dim cht_height As Double
    cht_height = 200

    Dim offset_vertical As Double
    offset_vertical = 195

    Dim offset_horz As Double
    offset_horz = 40

    Dim sht As Worksheet
    Set sht = ActiveSheet

    Dim count As Integer
    count = 0

    'iterate through ChartObjects on current sheet
    Dim cht_obj As ChartObject
    For Each cht_obj In sht.ChartObjects

        'use integer division and Mod to get position in grid
        cht_obj.Top = (count \ int_cols) * cht_height + offset_vertical
        cht_obj.Left = (count Mod int_cols) * cht_width + offset_horz
        cht_obj.Width = cht_width
        cht_obj.Height = cht_height

        count = count + 1

    Next cht_obj
End Sub
```

Ergebnis mit mehreren Diagrammen

Diese Bilder zeigen das ursprüngliche zufällige Layout der Diagramme und das resultierende Raster, wenn der obige Code ausgeführt wird.

Vor


A
B
C
D
E
F
G
H
I
J
K
L
M
1
2
A
B
C
D
B2
C2
D2
3
8/4/2016
9
43
38
9
43
38
4
8/5/2016
56
72
14
56
72
14
5
8/6/2016
78
45
58
78
45
58
6
8/7/2016
34
10
90
34
10
B
C
D
D2
B2
C2
8/2/2016
8/6/2016
8/10/
B2
C2

Nach dem

	A	B	C	D	E	F	G	H
1								
2		A	B	C	D	B2	C2	D2
3		8/4/2016	9	43	38	9	43	38
4		8/5/2016	56	72	14	56	72	14
5		8/6/2016	78	45	58	78	45	58
6		8/7/2016	34	10	90	34	10	90
7		8/8/2016	35	82	74	35	82	74
8		8/9/2016	57	32	40	57	32	40
9		8/10/2016	40	47	2	40	47	2
10		8/11/2016	8	73	93	8	73	93
11		8/12/2016	12	46	28	12	46	28
12		8/13/2016	52	25	37	52	25	37

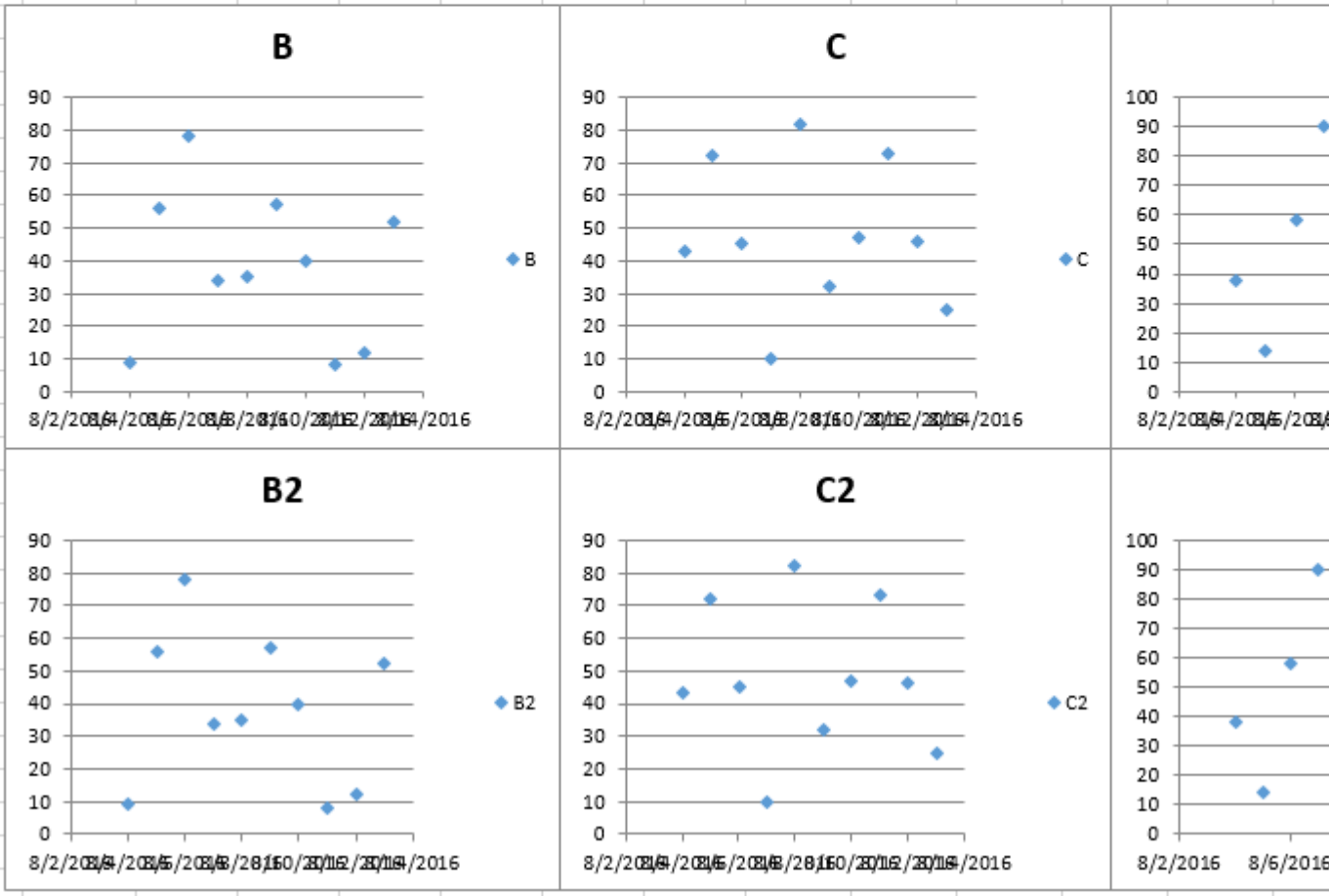


Diagramme und Diagramme online lesen: https://riptutorial.com/de/excel-vba/topic/4968/diagramme-und-diagramme

Kapitel 17: Doppelte Werte in einem Bereich suchen

Einführung

An bestimmten Stellen werden Sie eine Reihe von Daten auswerten, und Sie müssen die Duplikate darin finden. Bei größeren Datensätzen gibt es verschiedene Ansätze, die entweder VBA-Code oder bedingte Funktionen verwenden. In diesem Beispiel wird eine einfache if-then-Bedingung in zwei verschachtelten for-next-Schleifen verwendet, um zu testen, ob jede Zelle im Bereich gleichwertig ist mit einer anderen Zelle im Bereich.

Examples

Duplikate in einem Bereich finden

Die folgenden Tests reichen von A2 bis A7 für doppelte Werte. **Anmerkung:** Dieses Beispiel zeigt eine mögliche Lösung als ersten Lösungsansatz. Es ist schneller, ein Array als einen Bereich zu verwenden, und Sie könnten Sammlungen, Wörterbücher oder XML-Methoden verwenden, um nach Duplikaten zu suchen.

```
    Sub find_duplicates()
' Declare variables
  Dim ws     As Worksheet               ' worksheet
  Dim cell   As Range                   ' cell within worksheet range
  Dim n      As Integer                 ' highest row number
  Dim bFound As Boolean                 ' boolean flag, if duplicate is found
  Dim sFound As String: sFound = "|"    ' found duplicates
  Dim s      As String                  ' message string
  Dim s2     As String                  ' partial message string
' Set Sheet to memory
  Set ws = ThisWorkbook.Sheets("Duplicates")

' loop thru FULLY QUALIFIED REFERENCE
  For Each cell In ws.Range("A2:A7")
    bFound = False: s2 = ""             ' start each cell with empty values
 '  Check if first occurrence of this value as duplicate to avoid further searches
    If InStr(sFound, "|" & cell & "|") = 0 Then

      For n = cell.Row + 1 To 7           ' iterate starting point to avoid REDUNDANT SEARCH
        If cell = ws.Range("A" & n).Value Then
           If cell.Row <> n Then        ' only other cells, as same cell cannot be a duplicate
                 bFound = True             ' boolean flag
              '  found duplicates in cell A{n}
                 s2 = s2 & vbNewLine & " -> duplicate in A" & n
           End If
        End If
       Next
     End If
   ' notice all found duplicates
     If bFound Then
```

```
          ' add value to list of all found duplicate values
          ' (could be easily split to an array for further analyze)
            sFound = sFound & cell & "|"
            s = s & cell.Address & " (value=" & cell & ")" & s2 & vbNewLine & vbNewLine
      End If
    Next
' Messagebox with final result
  MsgBox "Duplicate values are " & sFound & vbNewLine & vbNewLine & s, vbInformation, "Found
duplicates"
End Sub
```

Je nach Ihren Anforderungen kann das Beispiel geändert werden. Beispielsweise kann die obere Grenze von n der Zeilenwert der letzten Zelle mit Daten im Bereich sein oder die Aktion im Fall einer True-If-Bedingung kann bearbeitet werden, um das Duplikat zu extrahieren Wert woanders. Die Mechanik der Routine würde sich jedoch nicht ändern.

Doppelte Werte in einem Bereich suchen online lesen: https://riptutorial.com/de/excel-vba/topic/8295/doppelte-werte-in-einem-bereich-suchen

Kapitel 18: Durchlaufen Sie alle Arbeitsblätter in der aktiven Arbeitsmappe

Examples

Rufen Sie alle Arbeitsblattnamen in Active Workbook ab

```
Option Explicit

Sub LoopAllSheets()

Dim sht As Excel.Worksheet
' declare an array of type String without committing to maximum number of members
Dim sht_Name() As String
Dim i As Integer

' get the number of worksheets in Active  Workbook , and put it as the maximum number of
members in the array
ReDim sht_Name(1 To ActiveWorkbook.Worksheets.count)

i = 1

' loop through all worksheets in Active Workbook
For Each sht In ActiveWorkbook.Worksheets
    sht_Name(i) = sht.Name ' get the name of each worksheet and save it in the array
    i = i + 1
Next sht

End Sub
```

Alle Blätter in allen Dateien in einem Ordner durchlaufen

```
 Sub Theloopofloops()

 Dim wbk As Workbook
 Dim Filename As String
 Dim path As String
 Dim rCell As Range
 Dim rRng As Range
 Dim wsO As Worksheet
 Dim sheet As Worksheet


 path = "pathtofile(s)" & "\"
 Filename = Dir(path & "*.xl??")
 Set wsO = ThisWorkbook.Sheets("Sheet1") 'included in case you need to differentiate_
             between workbooks i.e currently opened workbook vs workbook containing code

 Do While Len(Filename) > 0
     DoEvents
     Set wbk = Workbooks.Open(path & Filename, True, True)
         For Each sheet In ActiveWorkbook.Worksheets  'this needs to be adjusted for
specifiying sheets. Repeat loop for each sheet so thats on a per sheet basis
```

```
                Set rRng = sheet.Range("a1:a1000") 'OBV needs to be changed
                For Each rCell In rRng.Cells
                If rCell <> "" And rCell.Value <> vbNullString And rCell.Value <> 0 Then

                   'code that does stuff

                End If
                Next rCell
        Next sheet
    wbk.Close False
    Filename = Dir
Loop
End Sub
```

Durchlaufen Sie alle Arbeitsblätter in der aktiven Arbeitsmappe online lesen: https://riptutorial.com/de/excel-vba/topic/1144/durchlaufen-sie-alle-arbeitsblatter-in-der-aktiven-arbeitsmappe

Kapitel 19: Erstellen eines Dropdown-Menüs im aktiven Arbeitsblatt mit einem Kombinationsfeld

Einführung

Dies ist ein einfaches Beispiel, das zeigt, wie Sie ein Dropdown-Menü im Active Sheet Ihrer Arbeitsmappe erstellen, indem Sie ein ActiveX-Objekt in einem Kombinationsfeld in das Arbeitsblatt einfügen. Sie können einen von fünf Jimi Hendrix-Songs in jede aktivierte Zelle des Arbeitsblatts einfügen und entsprechend löschen.

Examples

Jimi Hendrix Menü

Im Allgemeinen wird der Code in das Modul eines Arbeitsblatts eingefügt.

Dies ist das Worksheet_SelectionChange-Ereignis, das jedes Mal ausgelöst wird, wenn eine andere Zelle im aktiven Arbeitsblatt ausgewählt wird. Sie können "Arbeitsblatt" aus dem ersten Dropdown-Menü über dem Codefenster und "Selection_Change" aus dem Dropdown-Menü daneben auswählen. In diesem Fall wird der Code bei jeder Aktivierung einer Zelle in den Code der Combo Box umgeleitet.

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

   ComboBox1_Change

End Sub
```

Hier ist die für die ComboBox bestimmte Routine standardmäßig für das Change-Ereignis codiert. Darin befindet sich ein festes Array, das alle Optionen enthält. Nicht die CLEAR-Option an der letzten Position, mit der der Inhalt einer Zelle gelöscht wird. Das Array wird dann an die Combo-Box übergeben und an die Routine übergeben, die die Arbeit erledigt.

```
Private Sub ComboBox1_Change()

Dim myarray(0 To 5)
    myarray(0) = "Hey Joe"
    myarray(1) = "Little Wing"
    myarray(2) = "Voodoo Child"
    myarray(3) = "Purple Haze"
    myarray(4) = "The Wind Cries Mary"
    myarray(5) = "CLEAR"

    With ComboBox1
        .List = myarray()
```

```
    End With

    FillACell myarray()

End Sub
```

Das Array wird an die Routine übergeben, in der die Zellen mit dem Songnamen oder dem Nullwert gefüllt werden, um sie zu leeren. Zunächst erhält eine Integer-Variable den Wert der Position der Wahl, die der Benutzer trifft. Anschließend wird die Combo-Box in die TOP LEFT-Ecke der vom Benutzer aktivierten Zelle verschoben und ihre Abmessungen angepasst, um das Erlebnis flüssiger zu gestalten. Der aktiven Zelle wird dann der Wert in der Position in der Ganzzahlvariablen zugewiesen, die die Auswahl des Benutzers verfolgt. Falls der Benutzer CLEAR aus den Optionen auswählt, wird die Zelle geleert.

Die gesamte Routine wird für jede ausgewählte Zelle wiederholt.

```
Sub FillACell(MyArray As Variant)

Dim n As Integer

n = ComboBox1.ListIndex

ComboBox1.Left = ActiveCell.Left
ComboBox1.Top = ActiveCell.Top
Columns(ActiveCell.Column).ColumnWidth = ComboBox1.Width * 0.18

ActiveCell = MyArray(n)

If ComboBox1 = "CLEAR" Then
    Range(ActiveCell.Address) = ""
End If

End Sub
```

Beispiel 2: Optionen nicht enthalten

Dieses Beispiel wird zum Angeben von Optionen verwendet, die möglicherweise nicht in einer Datenbank verfügbarer Wohnungen und der zugehörigen Annehmlichkeiten enthalten sind.

Es baut auf dem vorherigen Beispiel auf, mit einigen Unterschieden:

1. Zwei Prozeduren sind für ein einzelnes Kombinationsfeld nicht mehr erforderlich, indem der Code in einer einzigen Prozedur kombiniert wird.
2. Die Verwendung der LinkedCell-Eigenschaft, um jedes Mal die korrekte Eingabe der Benutzerauswahl zu ermöglichen
3. Die Einfügung einer Sicherungsfunktion zum Sicherstellen der aktiven Zelle befindet sich in der richtigen Spalte und ein Fehlervermeidungscode, der auf früheren Erfahrungen basiert, wobei numerische Werte als Zeichenfolgen formatiert würden, wenn sie in die aktive Zelle eingefügt werden.

```
Private Sub cboNotIncl_Change()
```

```
Dim n As Long
Dim notincl_array(1 To 9) As String

n = myTarget.Row

    If n >= 3 And n < 10000 Then

        If myTarget.Address = "$G$" & n Then

            'set up the array elements for the not included services
            notincl_array(1) = "Central Air"
            notincl_array(2) = "Hot Water"
            notincl_array(3) = "Heater Rental"
            notincl_array(4) = "Utilities"
            notincl_array(5) = "Parking"
            notincl_array(6) = "Internet"
            notincl_array(7) = "Hydro"
            notincl_array(8) = "Hydro/Hot Water/Heater Rental"
            notincl_array(9) = "Hydro and Utilities"

            cboNotIncl.List = notincl_array()

        Else

            Exit Sub

        End If

        With cboNotIncl

            'make sure the combo box moves to the target cell
            .Left = myTarget.Left
            .Top = myTarget.Top

            'adjust the size of the cell to fit the combo box
            myTarget.ColumnWidth = .Width * 0.18

            'make it look nice by editing some of the font attributes
            .Font.Size = 11
            .Font.Bold = False

            'populate the cell with the user choice, with a backup guarantee that it's in
column G

            If myTarget.Address = "$G$" & n Then

                    .LinkedCell = myTarget.Address

                    'prevent an error where a numerical value is formatted as text
                    myTarget.EntireColumn.TextToColumns

            End If

        End With

    End If 'ensure that the active cell is only between rows 3 and 1000

End Sub
```

Das obige Makro wird jedes Mal initiiert, wenn eine Zelle mit dem SelectionChange-Ereignis im

Arbeitsblattmodul aktiviert wird:

```
Public myTarget As Range

Private Sub Worksheet_SelectionChange(ByVal Target As Range)

    Set myTarget = Target

    'switch for Not Included
    If Target.Column = 7 And Target.Cells.Count = 1 Then

        Application.Run "Module1.cboNotIncl_Change"

    End If

End Sub
```

Erstellen eines Dropdown-Menüs im aktiven Arbeitsblatt mit einem Kombinationsfeld online lesen: https://riptutorial.com/de/excel-vba/topic/8929/erstellen-eines-dropdown-menus-im-aktiven-arbeitsblatt-mit-einem-kombinationsfeld

Kapitel 20: Excel-VBA-Optimierung

Einführung

Die Excel-VBA-Optimierung bezieht sich auch auf die Kodierung einer besseren Fehlerbehandlung durch Dokumentation und zusätzliche Details. Dies wird hier gezeigt.

Bemerkungen

*) Zeilennummern sind ganze Zahlen, d. H. Ein vorzeichenbehafteter 16-Bit-Datentyp im Bereich von -32.768 bis 32.767, ansonsten erzeugen Sie einen Überlauf. Üblicherweise werden Zeilennummern in Schritten von 10 über einen Teil des Codes oder alle Prozeduren eines Moduls insgesamt eingefügt.

Examples

Arbeitsblattaktualisierung deaktivieren

Durch das Deaktivieren der Berechnung des Arbeitsblatts kann die Laufzeit des Makros erheblich verringert werden. Außerdem wäre das Deaktivieren von Ereignissen, Bildschirmaktualisierungen und Seitenumbrüchen von Vorteil. Folgender `Sub` kann zu diesem Zweck in einem beliebigen Makro verwendet werden.

```
Sub OptimizeVBA(isOn As Boolean)
    Application.Calculation = IIf(isOn, xlCalculationManual, xlCalculationAutomatic)
    Application.EnableEvents = Not(isOn)
    Application.ScreenUpdating = Not(isOn)
    ActiveSheet.DisplayPageBreaks = Not(isOn)
End Sub
```

Zur Optimierung folgen Sie dem unten stehenden Pseudo-Code:

```
Sub MyCode()

    OptimizeVBA True

    'Your code goes here

    OptimizeVBA False

End Sub
```

Ausführungszeit prüfen

Unterschiedliche Prozeduren können dasselbe Ergebnis liefern, sie würden jedoch andere Verarbeitungszeiten verwenden. Um herauszufinden, welcher schneller ist, kann ein Code wie folgt verwendet werden:

```
time1 = Timer

For Each iCell In MyRange
   iCell = "text"
Next iCell

time2 = Timer

For i = 1 To 30
   MyRange.Cells(i) = "text"
Next i

time3 = Timer

debug.print "Proc1 time: " & cStr(time2-time1)
debug.print "Proc2 time: " & cStr(time3-time2)
```

MicroTimer :

```
Private Declare PtrSafe Function getFrequency Lib "Kernel32" Alias "QueryPerformanceFrequency"
(cyFrequency As Currency) As Long
Private Declare PtrSafe Function getTickCount Lib "Kernel32" Alias "QueryPerformanceCounter"
(cyTickCount As Currency) As Long

Function MicroTimer() As Double
    Dim cyTicks1 As Currency
    Static cyFrequency As Currency

    MicroTimer = 0
    If cyFrequency = 0 Then getFrequency cyFrequency        'Get frequency
    getTickCount cyTicks1                                   'Get ticks
    If cyFrequency Then MicroTimer = cyTicks1 / cyFrequency 'Returns Seconds
End Function
```

Verwendung mit Blöcken

Die Verwendung mit Blöcken kann die Ausführung eines Makros beschleunigen. Statt einen Bereich, einen Diagrammnamen, ein Arbeitsblatt usw. zu schreiben, können Sie With-Blocks wie folgt verwenden.

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

Welches ist schneller als das:

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

Anmerkungen:

- Sobald ein With-Block eingegeben wurde, kann das Objekt nicht geändert werden. Daher können Sie keine einzelne With-Anweisung verwenden, um eine Reihe verschiedener Objekte zu beeinflussen
- **Springen Sie nicht in With-Blöcke** . Wenn Anweisungen in einem With-Block ausgeführt werden, die With- oder End With-Anweisung jedoch nicht ausgeführt wird, **bleibt eine temporäre Variable, die einen Verweis auf das Objekt enthält, im Speicher, bis Sie die Prozedur beenden**
- Keine Schleife mit With-Anweisungen, insbesondere wenn das zwischengespeicherte Objekt als Iterator verwendet wird
- Sie können With-Anweisungen verschachteln, indem Sie einen With-Block in einen anderen einfügen. Da jedoch Mitglieder der äußeren With-Blöcke innerhalb der inneren With-Blöcke maskiert werden, müssen Sie jedem Member eines Objekts in einem äußeren With-Block eine vollständig qualifizierte Objektreferenz in einem inneren With-Block bereitstellen.

Verschachtelungsbeispiel:

In diesem Beispiel wird mit der With-Anweisung eine Reihe von Anweisungen für ein einzelnes Objekt ausgeführt.
Das Objekt und seine Eigenschaften sind generische Namen, die nur zu Illustrationszwecken verwendet werden.

```
With MyObject
    .Height = 100               'Same as MyObject.Height = 100.
    .Caption = "Hello World"    'Same as MyObject.Caption = "Hello World".
    With .Font
        .Color = Red            'Same as MyObject.Font.Color = Red.
        .Bold = True            'Same as MyObject.Font.Bold = True.
        MyObject.Height = 200   'Inner-most With refers to MyObject.Font (must be qualified
    End With
End With
```

Weitere Informationen zu MSDN

Zeilenlöschung - Leistung

- Das Löschen von Zeilen ist langsam, insbesondere wenn Sie die Zellen durchlaufen und nacheinander Zeilen löschen
- Ein anderer Ansatz ist die Verwendung eines Autofilters zum Ausblenden der zu löschenden Zeilen
- Kopieren Sie den sichtbaren Bereich und fügen Sie ihn in ein neues Arbeitsblatt ein
- Entfernen Sie das Ausgangsblatt vollständig
- Je mehr Zeilen gelöscht werden, desto schneller wird diese Methode

Beispiel:

```
Option Explicit

'Deleted rows: 775,153, Total Rows: 1,000,009, Duration: 1.87 sec

Public Sub DeleteRows()
    Dim oldWs As Worksheet, newWs As Worksheet, wsName As String, ur As Range

    Set oldWs = ThisWorkbook.ActiveSheet
    wsName = oldWs.Name
    Set ur = oldWs.Range("F2", oldWs.Cells(oldWs.Rows.Count, "F").End(xlUp))

    Application.ScreenUpdating = False
    Set newWs = Sheets.Add(After:=oldWs)    'Create a new WorkSheet

    With ur    'Copy visible range after Autofilter (modify Criteria1 and 2 accordingly)
        .AutoFilter Field:=1, Criteria1:="<>0", Operator:=xlAnd, Criteria2:="<>"
        oldWs.UsedRange.Copy
    End With
    'Paste all visible data into the new WorkSheet (values and formats)
    With newWs.Range(oldWs.UsedRange.Cells(1).Address)
        .PasteSpecial xlPasteColumnWidths
        .PasteSpecial xlPasteAll
        newWs.Cells(1, 1).Select: newWs.Cells(1, 1).Copy
    End With

    With Application
        .CutCopyMode = False
        .DisplayAlerts = False
            oldWs.Delete
        .DisplayAlerts = True
        .ScreenUpdating = True
    End With
    newWs.Name = wsName
End Sub
```

Deaktivieren aller Excel-Funktionen Vor dem Ausführen großer Makros

In den folgenden Abschnitten werden alle Excel-Funktionen auf WorkBook- und WorkSheet-Ebene vorübergehend deaktiviert

- FastWB () ist ein Umschalter, der On- oder Off-Flags akzeptiert
- FastWS () akzeptiert ein optionales WorkSheet-Objekt oder keines
- Wenn der Parameter ws fehlt, werden alle Funktionen für alle Arbeitsblätter in der Auflistung ein- und ausgeschaltet
    - Mit einem benutzerdefinierten Typ können Sie alle Einstellungen erfassen, bevor Sie sie deaktivieren
    - Am Ende des Prozesses können die ursprünglichen Einstellungen wiederhergestellt werden

---

```
Public Sub FastWB(Optional ByVal opt As Boolean = True)
```

```
    With Application
        .Calculation = IIf(opt, xlCalculationManual, xlCalculationAutomatic)
        If .DisplayAlerts <> Not opt Then .DisplayAlerts = Not opt
        If .DisplayStatusBar <> Not opt Then .DisplayStatusBar = Not opt
        If .EnableAnimations <> Not opt Then .EnableAnimations = Not opt
        If .EnableEvents <> Not opt Then .EnableEvents = Not opt
        If .ScreenUpdating <> Not opt Then .ScreenUpdating = Not opt
    End With
    FastWS , opt
End Sub
```

```
Public Sub FastWS(Optional ByVal ws As Worksheet, Optional ByVal opt As Boolean = True)
    If ws Is Nothing Then
        For Each ws In Application.ThisWorkbook.Sheets
            OptimiseWS ws, opt
        Next
    Else
        OptimiseWS ws, opt
    End If
End Sub
Private Sub OptimiseWS(ByVal ws As Worksheet, ByVal opt As Boolean)
    With ws
        .DisplayPageBreaks = False
        .EnableCalculation = Not opt
        .EnableFormatConditionsCalculation = Not opt
        .EnablePivotTable = Not opt
    End With
End Sub
```

Setzen Sie alle Excel-Einstellungen auf die Standardeinstellungen zurück

```
Public Sub XlResetSettings()    'default Excel settings
    With Application
        .Calculation = xlCalculationAutomatic
        .DisplayAlerts = True
        .DisplayStatusBar = True
        .EnableAnimations = False
        .EnableEvents = True
        .ScreenUpdating = True
        Dim sh As Worksheet
        For Each sh In Application.ThisWorkbook.Sheets
            With sh
                .DisplayPageBreaks = False
                .EnableCalculation = True
                .EnableFormatConditionsCalculation = True
                .EnablePivotTable = True
            End With
        Next
    End With
End Sub
```

Optimierung der Fehlersuche durch erweitertes Debugging

Zeilennummern verwenden ... und im Fehlerfall dokumentieren ("Wie wichtig es ist, Erl zu sehen")

Das Erkennen der Zeile, die einen Fehler auslöst, ist ein wesentlicher Bestandteil des Debugging und begrenzt die Suche nach der Ursache. Die Dokumentation der identifizierten Fehlerzeilen mit einer kurzen Beschreibung vervollständigt eine erfolgreiche Fehlerverfolgung, am besten zusammen mit den Namen des Moduls und der Prozedur. Das folgende Beispiel speichert diese Daten in einer Protokolldatei.

Hintergrund

Das Fehlerobjekt gibt Fehlernummer (Err.Number) und Fehlerbeschreibung (Err.Description) zurück, antwortet jedoch nicht explizit auf die Frage, wo der Fehler zu finden ist. Die **Erl-** Funktion tut dies jedoch, vorausgesetzt, Sie fügen * (*Zeilennummern)* dem Code hinzu (BTW eine von mehreren anderen Zugeständnisse an frühere Basiszeiten).

Wenn keine Fehlerzeilen vorhanden sind, gibt die Erl-Funktion 0 zurück. Wenn die Nummerierung unvollständig ist, wird die letzte vorangehende Zeilennummer der Prozedur angezeigt.

```
Option Explicit


Public Sub MyProc1()
Dim i As Integer
Dim j As Integer
On Error GoTo LogErr
10     j = 1 / 0    ' raises an error
okay:
Debug.Print "i=" & i
Exit Sub

LogErr:
MsgBox LogErrors("MyModule", "MyProc1", Err), vbExclamation, "Error " & Err.Number
Stop
Resume Next
End Sub

Public Function LogErrors( _
           ByVal sModule As String, _
           ByVal sProc As String, _
           Err As ErrObject) As String
' Purpose: write error number, description and Erl to log file and return error text
  Dim sLogFile As String: sLogFile = ThisWorkbook.Path & Application.PathSeparator &
"LogErrors.txt"
  Dim sLogTxt  As String
  Dim lFile    As Long

' Create error text
  sLogTxt = sModule & "|" & sProc & "|Erl " & Erl & "|Err " & Err.Number & "|" &
Err.Description

  On Error Resume Next
  lFile = FreeFile

  Open sLogFile For Append As lFile
  Print #lFile, Format$(Now(), "yy.mm.dd hh:mm:ss "); sLogTxt
      Print #lFile,
  Close lFile
' Return error text
  LogErrors = sLogTxt
```

```
End Function
```

' Zusätzlicher Code zum Anzeigen der Protokolldatei

```
Sub ShowLogFile()
Dim sLogFile As String: sLogFile = ThisWorkbook.Path & Application.PathSeparator &
"LogErrors.txt"

On Error GoTo LogErr
Shell "notepad.exe " & sLogFile, vbNormalFocus

okay:
On Error Resume Next
Exit Sub

LogErr:
MsgBox LogErrors("MyModule", "ShowLogFile", Err), vbExclamation, "Error No " & Err.Number
Resume okay
End Sub
```

Excel-VBA-Optimierung online lesen: https://riptutorial.com/de/excel-vba/topic/9798/excel-vba-optimierung

Kapitel 21: Häufige Fehler

Examples

Qualifizierende Referenzen

Wenn Sie sich auf ein `worksheet` , einen `range` oder einzelne `cells` beziehen, ist es wichtig, die Referenz vollständig zu qualifizieren.

Zum Beispiel:

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

Ist nicht vollständig qualifiziert: Den `Cells` sind keine Arbeitsmappe und kein Arbeitsblatt zugeordnet. Ohne explizite Referenz verweist Cells standardmäßig auf das `ActiveSheet` . Daher `Sheet1` dieser Code fehl (falsche Ergebnisse), wenn ein anderes Arbeitsblatt als `Sheet1` das aktuelle `ActiveSheet` .

Die einfachste Möglichkeit, dies zu korrigieren, besteht darin, eine `With` Anweisung wie folgt zu verwenden:

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

Alternativ können Sie eine Arbeitsblattvariable verwenden. (Diese Methode wird höchstwahrscheinlich bevorzugt, wenn Ihr Code auf mehrere Arbeitsblätter verweisen muss, z. B. das Kopieren von Daten von einem Arbeitsblatt auf ein anderes.)

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

Ein weiteres häufiges Problem besteht darin, auf die Worksheets-Auflistung zu verweisen, ohne die Arbeitsmappe zu qualifizieren. Zum Beispiel:

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

Das Arbeitsblatt `Sheet1` ist nicht vollständig qualifiziert und es fehlt eine Arbeitsmappe. Dies kann fehlschlagen, wenn auf mehrere Arbeitsmappen im Code verwiesen wird. Verwenden Sie stattdessen eine der folgenden Möglichkeiten:

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

Vermeiden Sie jedoch Folgendes:

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

Ähnliches gilt für `range` Objekte, wenn nicht ausdrücklich qualifiziert, der `range` wird auf das aktuell aktive Blatt verweisen:

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

Ist das gleiche wie:

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

Zeilen oder Spalten in einer Schleife löschen

Wenn Sie Zeilen (oder Spalten) in einer Schleife löschen möchten, sollten Sie die Schleife immer am Ende des Bereichs beginnen und in jedem Schritt zurückgehen. Bei Verwendung des Codes:

```
Dim i As Long
With Workbooks("Book1").Worksheets("Sheet1")
    For i = 1 To 4
        If IsEmpty(.Cells(i, 1)) Then .Rows(i).Delete
    Next i
End With
```

Sie werden einige Reihen vermissen. Wenn der Code beispielsweise Zeile 3 löscht, wird Zeile 4 zu Zeile 3. Die Variable `i` ändert sich jedoch zu 4. In diesem Fall wird der Code eine Zeile übersehen und eine andere Zeile überprüfen, die zuvor nicht in Reichweite war.

Der richtige Code wäre

```
Dim i As Long
With Workbooks("Book1").Worksheets("Sheet1")
    For i = 4 To 1 Step -1
        If IsEmpty(.Cells(i, 1)) Then .Rows(i).Delete
    Next i
End With
```

ActiveWorkbook vs. ThisWorkbook

`ActiveWorkbook` und `ThisWorkbook` werden von neuen VBA-Benutzern manchmal austauschbar verwendet, ohne zu `ThisWorkbook` , worauf sich jedes Objekt bezieht. Beide Objekte gehören zum Anwendungsobjekt

---

Das `ActiveWorkbook` Objekt verweist auf die Arbeitsmappe, die sich zum Zeitpunkt der Ausführung derzeit in der obersten Ansicht des Excel-Anwendungsobjekts befindet. *(zB die Arbeitsmappe, mit der Sie an dem Punkt sehen und interagieren können, an dem auf dieses Objekt verwiesen wird)*

```
Sub ActiveWorkbookExample()

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

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

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

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

End Sub
```

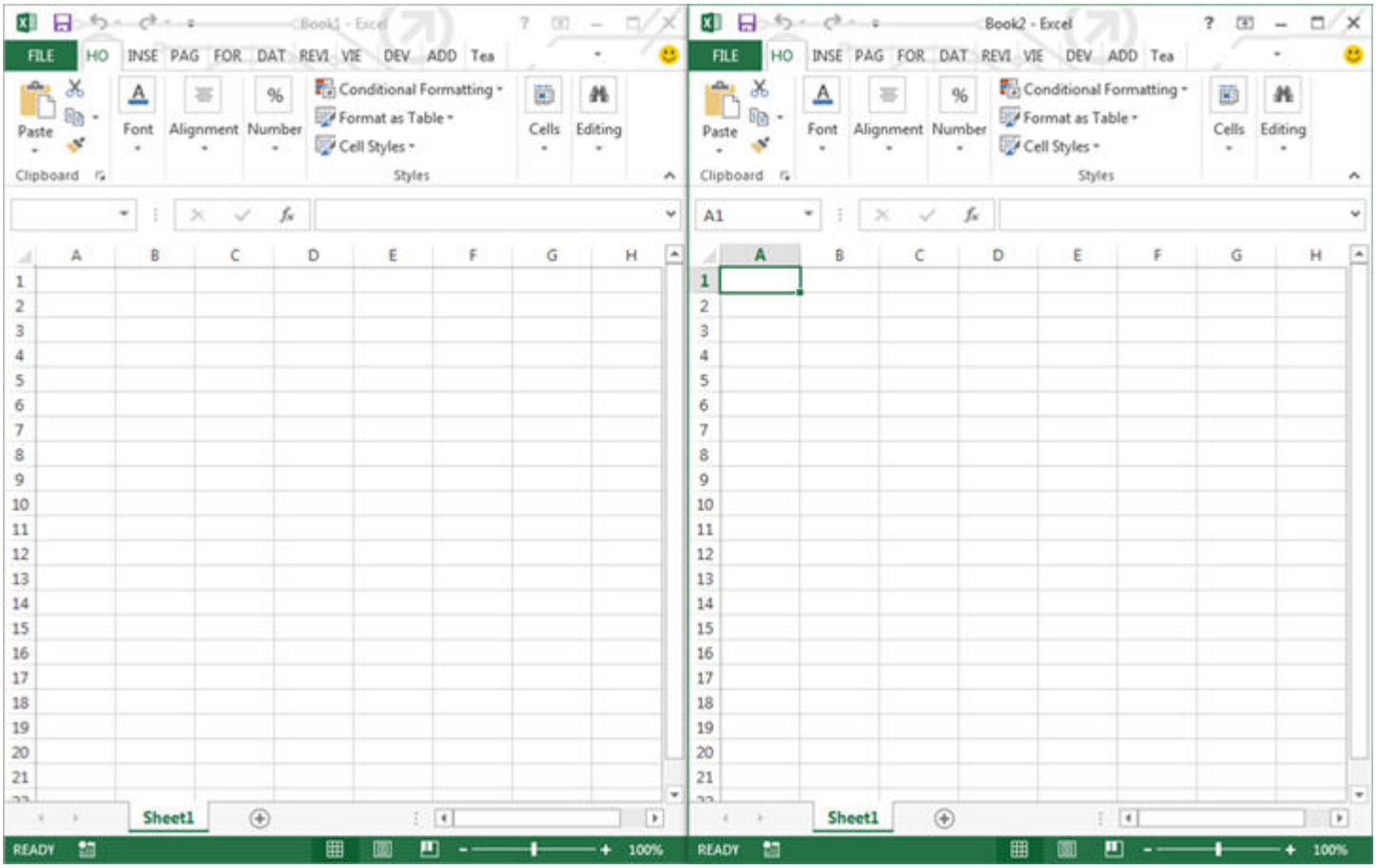
Das `ThisWorkbook` Objekt verweist auf die Arbeitsmappe, zu der der Code zum Zeitpunkt der Ausführung gehört.

```
Sub ThisWorkbookExample()

'// Let's assume to begin that this code is in the same workbook that is currently active

    ActiveWorkbook.Sheet1.Range("A1").Value = "Foo"
    Workbooks.Add 1
    ActiveWorkbook.ActiveSheet.Range("A1").Value = "Bar"

    Debug.Print ActiveWorkbook.ActiveSheet.Range("A1").Value '// Prints "Bar"
    Debug.Print ThisWorkbook.Sheet1.Range("A1").Value '// Prints "Foo"

End Sub
```

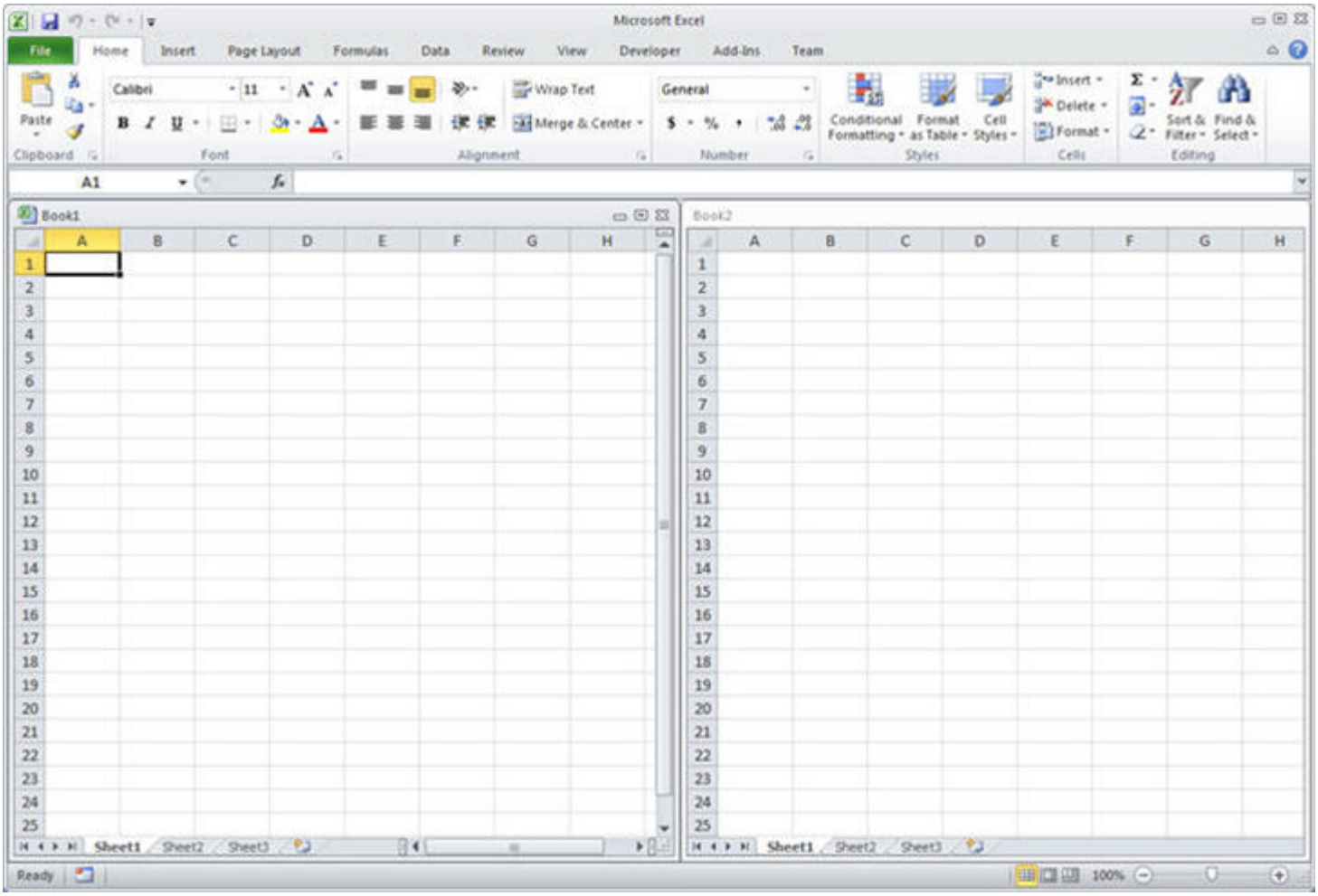
Schnittstelle für ein Dokument vs. mehrere Dokumentschnittstellen

> Beachten Sie, dass Microsoft Excel 2013 (und höher) Single Document Interface (SDI) und Excel 2010 (und darunter) Multiple Document Interfaces (MDI) verwenden.

Dies impliziert, dass für Excel 2013 (SDI) jede Arbeitsmappe in einer einzelnen Instanz von Excel eine **eigene** Menübandoberfläche enthält:



Im Gegensatz dazu wurde für Excel 2010 für jede Arbeitsmappe in einer einzelnen Instanz von Excel eine **gemeinsame** Ribbon-Benutzeroberfläche (MDI) verwendet:



Dies wirft einige wichtige Probleme auf, wenn Sie einen VBA-Code (2010 <-> 2013) migrieren möchten, der mit der Multifunktionsleiste interagiert.

> Es muss eine Prozedur erstellt werden, um die Ribbon-UI-Steuerelemente in allen Arbeitsmappen für Excel 2013 und höher im gleichen Status zu aktualisieren.

Beachten Sie, dass :

1. Alle Fenstermethoden, Ereignisse und Eigenschaften auf Excel-Anwendungsebene bleiben davon unberührt. (`Application.ActiveWindow` , `Application.Windows` ...)
2. In Excel 2013 und höher (SDI) werden jetzt alle Methoden, Ereignisse und Eigenschaften der Arbeitsmappen auf Fenstern der obersten Ebene ausgeführt. Das Handle dieses obersten Fensters kann mit `Application.Hwnd` abgerufen werden

Weitere Informationen finden Sie in der Quelle dieses Beispiels: MSDN .

Dies verursacht auch einige Probleme mit modelllosen Benutzerformularen. Hier finden Sie eine Lösung.

Häufige Fehler online lesen: https://riptutorial.com/de/excel-vba/topic/1576/haufige-fehler

Kapitel 22: Methoden zum Suchen der zuletzt verwendeten Zeile oder Spalte in einem Arbeitsblatt

Bemerkungen

Eine gute Erklärung, warum andere Methoden nicht empfohlen werden, ist hier zu finden: http://stackoverflow.com/a/11169920/4628637

Examples

Suchen Sie die letzte nicht leere Zelle in einer Spalte

In diesem Beispiel betrachten wir eine Methode zum Zurückgeben der letzten nicht leeren Zeile in einer Spalte für einen Datensatz.

Diese Methode funktioniert unabhängig von leeren Bereichen innerhalb des Datensatzes.

***Vorsicht** ist* jedoch ***geboten** , wenn verbundene **Zellen** beteiligt sind* , da die `End` Methode für einen verbundenen Bereich "angehalten" wird und die erste Zelle des zusammengeführten Bereichs zurückgibt.

Darüber hinaus werden nicht leere Zellen in ***ausgeblendeten Zeilen*** nicht berücksichtigt.

```
Sub FindingLastRow()
    Dim wS As Worksheet, LastRow As Long
    Set wS = ThisWorkbook.Worksheets("Sheet1")

    'Here we look in Column A
    LastRow = wS.Cells(wS.Rows.Count, "A").End(xlUp).Row
    Debug.Print LastRow
End Sub
```

Um die oben genannten Einschränkungen zu beheben, gilt folgende Zeile:
`LastRow = wS.Cells(wS.Rows.Count, "A").End(xlUp).Row`

kann ersetzt werden durch:

1. für die zuletzt verwendete Reihe von `"Sheet1"` :
   `LastRow = wS.UsedRange.Row - 1 + wS.UsedRange.Rows.Count .`
2. für die letzte nicht leere Zelle der Spalte `"A"` in `"Sheet1"` :

```
Dim i As Long
For i = LastRow To 1 Step -1
    If Not (IsEmpty(Cells(i, 1))) Then Exit For
```

```
Next i
LastRow = i
```

Letzte Zeile mit benanntem Bereich suchen

Falls Sie einen benannten Bereich in Ihrem Arbeitsblatt haben und die letzte Zeile dieses dynamischen benannten Bereichs dynamisch abrufen möchten. Behandelt auch Fälle, in denen der benannte Bereich nicht bei der ersten Reihe beginnt.

```
Sub FindingLastRow()

Dim sht As Worksheet
Dim LastRow As Long
Dim FirstRow As Long

Set sht = ThisWorkbook.Worksheets("form")

'Using Named Range "MyNameRange"
FirstRow = sht.Range("MyNameRange").Row

' in case "MyNameRange" doesn't start at Row 1
LastRow = sht.Range("MyNameRange").Rows.count + FirstRow - 1

End Sub
```

Aktualisieren:
@Jeeped hat auf ein mögliches Schlupfloch für einen benannten Bereich mit nicht zusammenhängenden Zeilen hingewiesen, da dies zu einem unerwarteten Ergebnis führt. Um dieses Problem zu beheben, wird der Code wie folgt überarbeitet.
Annahmen: targes sheet = `form` , benannter Bereich = `MyNameRange`

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

Holen Sie sich die Zeile der letzten Zelle in einem Bereich

```
'if only one area (not multiple areas):
With Range("A3:D20")
    Debug.Print .Cells(.Cells.CountLarge).Row
    Debug.Print .Item(.Cells.CountLarge).Row 'using .item is also possible
End With 'Debug prints: 20

'with multiple areas (also works if only one area):
Dim rngArea As Range, LastRow As Long
With Range("A3:D20, E5:I50, H20:R35")
    For Each rngArea In .Areas
        If rngArea(rngArea.Cells.CountLarge).Row > LastRow Then
            LastRow = rngArea(rngArea.Cells.CountLarge).Row
        End If
```

```
    Next
    Debug.Print LastRow 'Debug prints: 50
End With
```

Suchen Sie die letzte nicht leere Spalte im Arbeitsblatt

```
Private Sub Get_Last_Used_Row_Index()
    Dim wS As Worksheet

    Set wS = ThisWorkbook.Sheets("Sheet1")
    Debug.Print LastCol_1(wS)
    Debug.Print LastCol_0(wS)
End Sub
```

Sie können zwischen zwei Optionen wählen, ob Sie wissen möchten, ob das Arbeitsblatt keine Daten enthält:

- NO: Use LastCol_1: Sie können es direkt in `wS.Cells(...,LastCol_1(wS))`
- JA: Verwenden Sie LastCol_0: Sie müssen vor der Verwendung prüfen, ob das Ergebnis der Funktion 0 ist oder nicht

```
Public Function LastCol_1(wS As Worksheet) As Double
    With wS
        If Application.WorksheetFunction.CountA(.Cells) <> 0 Then
            LastCol_1 = .Cells.Find(What:="*", _
                                    After:=.Range("A1"), _
                                    Lookat:=xlPart, _
                                    LookIn:=xlFormulas, _
                                    SearchOrder:=xlByColumns, _
                                    SearchDirection:=xlPrevious, _
                                    MatchCase:=False).Column
        Else
            LastCol_1 = 1
        End If
    End With
End Function
```

Die Eigenschaften des Err-Objekts werden beim Beenden der Funktion automatisch auf Null zurückgesetzt.

```
Public Function LastCol_0(wS As Worksheet) As Double
    On Error Resume Next
    LastCol_0 = wS.Cells.Find(What:="*", _
                              After:=ws.Range("A1"), _
                              Lookat:=xlPart, _
                              LookIn:=xlFormulas, _
                              SearchOrder:=xlByColumns, _
                              SearchDirection:=xlPrevious, _
                              MatchCase:=False).Column
End Function
```

Letzte Zelle in Range.CurrentRegion

`Range.CurrentRegion` ist ein rechteckiger Bereich, der von leeren Zellen umgeben ist. Leere Zellen

mit Formeln wie `=""` oder `'` nicht leer betrachtet (auch durch die `ISBLANK` Excel - Funktion).

```
Dim rng As Range, lastCell As Range
Set rng = Range("C3").CurrentRegion       ' or Set rng = Sheet1.UsedRange.CurrentRegion
Set lastCell = rng(rng.Rows.Count, rng.Columns.Count)
```

Suchen Sie die letzte nicht leere Zeile im Arbeitsblatt

```
Private Sub Get_Last_Used_Row_Index()
    Dim wS As Worksheet

    Set wS = ThisWorkbook.Sheets("Sheet1")
    Debug.Print LastRow_1(wS)
    Debug.Print LastRow_0(wS)
End Sub
```

Sie können zwischen zwei Optionen wählen, ob Sie wissen möchten, ob das Arbeitsblatt keine Daten enthält:

- NEIN: LastRow_1 verwenden: Sie können es direkt in `wS.Cells(LastRow_1(wS),...)`
- JA: Verwenden Sie LastRow_0: Sie müssen vor der Verwendung prüfen, ob das Ergebnis der Funktion 0 ist oder nicht

```
Public Function LastRow_1(wS As Worksheet) As Double
    With wS
        If Application.WorksheetFunction.CountA(.Cells) <> 0 Then
            LastRow_1 = .Cells.Find(What:="*", _
                                    After:=.Range("A1"), _
                                    Lookat:=xlPart, _
                                    LookIn:=xlFormulas, _
                                    SearchOrder:=xlByRows, _
                                    SearchDirection:=xlPrevious, _
                                    MatchCase:=False).Row
        Else
            LastRow_1 = 1
        End If
    End With
End Function

Public Function LastRow_0(wS As Worksheet) As Double
    On Error Resume Next
    LastRow_0 = wS.Cells.Find(What:="*", _
                              After:=ws.Range("A1"), _
                              Lookat:=xlPart, _
                              LookIn:=xlFormulas, _
                              SearchOrder:=xlByRows, _
                              SearchDirection:=xlPrevious, _
                              MatchCase:=False).Row
End Function
```

Finden Sie die letzte nicht leere Zelle in einer Reihe

In diesem Beispiel betrachten wir eine Methode zum Zurückgeben der letzten nicht leeren Spalte in einer Zeile.

Diese Methode funktioniert unabhängig von leeren Bereichen innerhalb des Datensatzes.

***Vorsicht** ist* jedoch ***geboten** , wenn verbundene **Zellen** beteiligt sind* , da die `End` Methode für einen verbundenen Bereich "angehalten" wird und die erste Zelle des zusammengeführten Bereichs zurückgibt.

Darüber hinaus werden nicht leere Zellen in ***ausgeblendeten Spalten*** nicht berücksichtigt.

```
Sub FindingLastCol()
    Dim wS As Worksheet, LastCol As Long
    Set wS = ThisWorkbook.Worksheets("Sheet1")

    'Here we look in Row 1
    LastCol = wS.Cells(1, wS.Columns.Count).End(xlToLeft).Column
    Debug.Print LastCol
End Sub
```

Letzte nicht leere Zelle im Arbeitsblatt suchen - Leistung (Array)

- Die erste Funktion mit einem Array ist **viel schneller**
- Wird der Parameter ohne den optionalen Parameter aufgerufen, wird standardmäßig `.ThisWorkbook.ActiveSheet`
- Wenn der Bereich leer ist, wird `Cell( 1, 1 )` als Standard statt `Nothing`

Geschwindigkeit:

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

Gemessen mit MicroTimer

```
Public Function GetLastCell(Optional ByVal ws As Worksheet = Nothing) As Range
    Dim uRng As Range, uArr As Variant, r As Long, c As Long
    Dim ubR As Long, ubC As Long, lRow As Long

    If ws Is Nothing Then Set ws = Application.ThisWorkbook.ActiveSheet
    Set uRng = ws.UsedRange
    uArr = uRng
    If IsEmpty(uArr) Then
        Set GetLastCell = ws.Cells(1, 1):   Exit Function
    End If
    If Not IsArray(uArr) Then
        Set GetLastCell = ws.Cells(uRng.Row, uRng.Column):  Exit Function
    End If
    ubR = UBound(uArr, 1):  ubC = UBound(uArr, 2)
    For r = ubR To 1 Step -1    '------------------------------------------------ last row
        For c = ubC To 1 Step -1
            If Not IsError(uArr(r, c)) Then
                If Len(Trim$(uArr(r, c))) > 0 Then
                    lRow = r:   Exit For
                End If
            End If
        Next
        If lRow > 0 Then Exit For
    Next
```

```
    If lRow = 0 Then lRow = ubR
    For c = ubC To 1 Step -1    '------------------------------------------- last col
        For r = lRow To 1 Step -1
            If Not IsError(uArr(r, c)) Then
                If Len(Trim$(uArr(r, c))) > 0 Then
                    Set GetLastCell = ws.Cells(lRow + uRng.Row - 1, c + uRng.Column - 1)
                    Exit Function
                End If
            End If
        Next
    Next
End Function
```

```
'Returns last cell (max row & max col) using Find

Public Function GetMaxCell2(Optional ByRef rng As Range = Nothing) As Range 'Using Find

    Const NONEMPTY As String = "*"

    Dim lRow As Range, lCol As Range

    If rng Is Nothing Then Set rng = Application.ThisWorkbook.ActiveSheet.UsedRange

    If WorksheetFunction.CountA(rng) = 0 Then
        Set GetMaxCell2 = rng.Parent.Cells(1, 1)
    Else
        With rng
            Set lRow = .Cells.Find(What:=NONEMPTY, LookIn:=xlFormulas, _
                                   After:=.Cells(1, 1), _
                                   SearchDirection:=xlPrevious, _
                                   SearchOrder:=xlByRows)
            If Not lRow Is Nothing Then
                Set lCol = .Cells.Find(What:=NONEMPTY, LookIn:=xlFormulas, _
                                       After:=.Cells(1, 1), _
                                       SearchDirection:=xlPrevious, _
                                       SearchOrder:=xlByColumns)

                Set GetMaxCell2 = .Parent.Cells(lRow.Row, lCol.Column)
            End If
        End With
    End If
End Function
```

.

MicroTimer :

```
Private Declare PtrSafe Function getFrequency Lib "Kernel32" Alias "QueryPerformanceFrequency"
(cyFrequency As Currency) As Long
Private Declare PtrSafe Function getTickCount Lib "Kernel32" Alias "QueryPerformanceCounter"
(cyTickCount As Currency) As Long

Function MicroTimer() As Double
    Dim cyTicks1 As Currency
    Static cyFrequency As Currency

    MicroTimer = 0
```

```
    If cyFrequency = 0 Then getFrequency cyFrequency        'Get frequency
    getTickCount cyTicks1                                   'Get ticks
    If cyFrequency Then MicroTimer = cyTicks1 / cyFrequency 'Returns Seconds
End Function
```

Methoden zum Suchen der zuletzt verwendeten Zeile oder Spalte in einem Arbeitsblatt online lesen: https://riptutorial.com/de/excel-vba/topic/918/methoden-zum-suchen-der-zuletzt-verwendeten-zeile-oder-spalte-in-einem-arbeitsblatt

Kapitel 23: Pivot-Tabellen

Bemerkungen

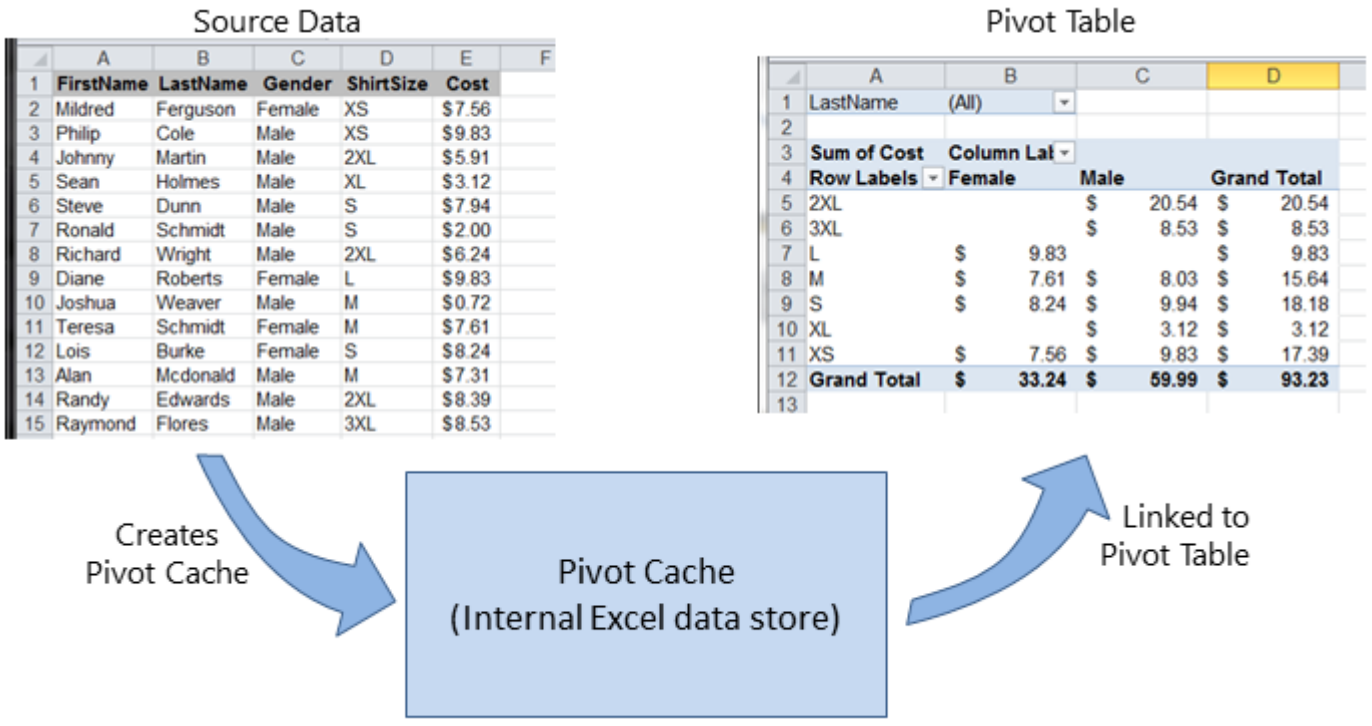
Es gibt viele ausgezeichnete Referenz- und Beispielquellen im Web. Einige Beispiele und Erklärungen werden hier als Sammelpunkt für schnelle Antworten erstellt. Detailliertere Abbildungen können mit externen Inhalten verknüpft werden (anstatt vorhandenes Originalmaterial zu kopieren).

Examples

Erstellen einer Pivot-Tabelle

Eine der leistungsfähigsten Funktionen in Excel ist die Verwendung von Pivot-Tabellen zum Sortieren und Analysieren von Daten. Die Verwendung von VBA zum Erstellen und Bearbeiten der Pivots ist einfacher, wenn Sie die Beziehung von Pivot-Tabellen zu Pivot-Caches und den Bezug und die Verwendung der verschiedenen Teile der Tabellen verstehen.

Im Grunde, die Quelldaten ist ein `Range` Bereich von Daten auf einem `Worksheet` . Dieser Datenbereich **MUSS** die Datenspalten mit einer Kopfzeile als erste Zeile im Bereich identifizieren. Sobald die Pivot-Tabelle erstellt wurde, kann der Benutzer die Quelldaten jederzeit anzeigen und ändern. Änderungen werden jedoch möglicherweise nicht automatisch oder sofort in der Pivot-Tabelle selbst widergespiegelt, da eine Zwischenspeicherstruktur mit dem Namen Pivot-Cache vorhanden ist, die direkt mit der Pivot-Tabelle selbst verbunden ist.

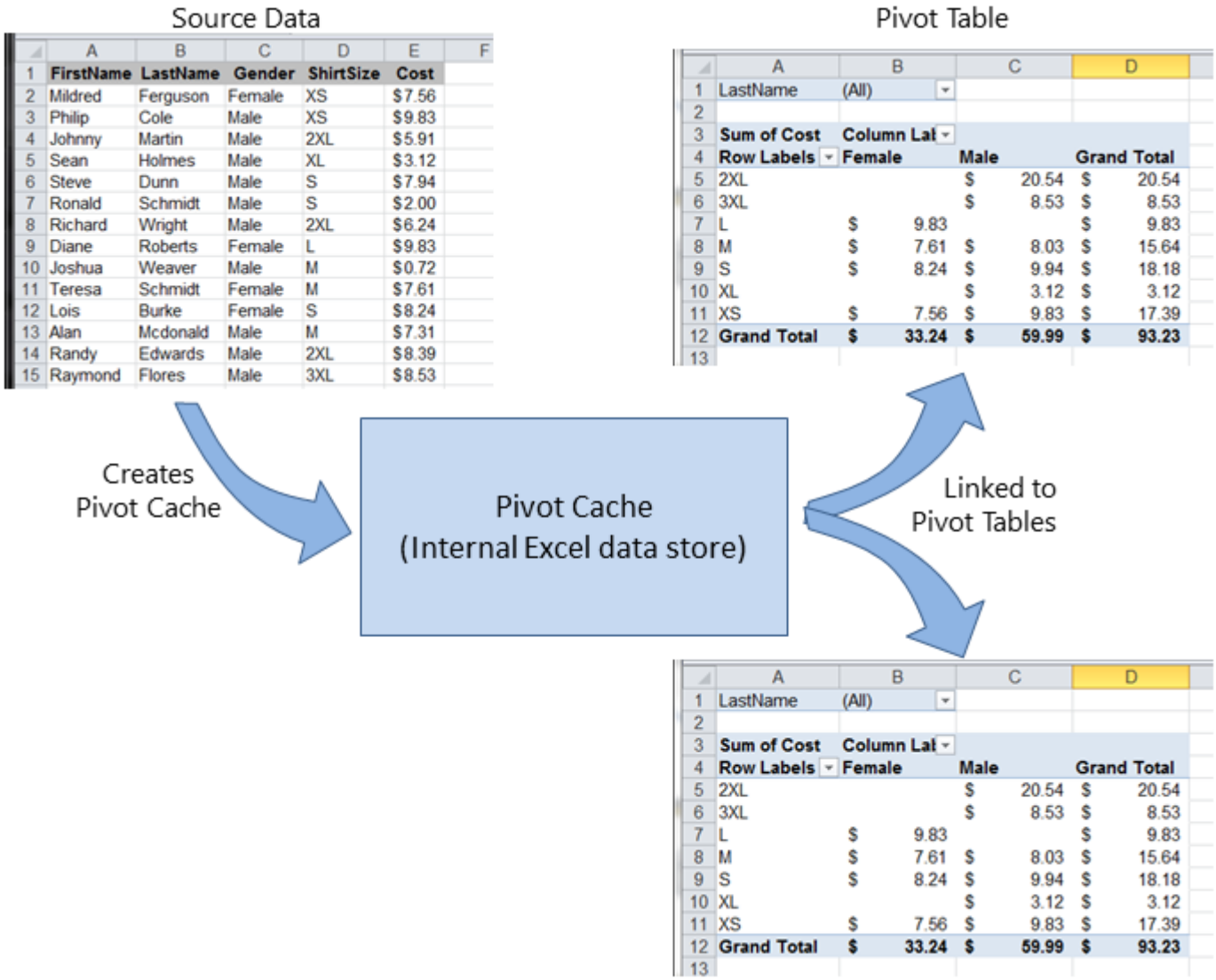


	A	B	C	D	E	F
1	FirstName	LastName	Gender	ShirtSize	Cost	
2	Mildred	Ferguson	Female	XS	$7.56	
3	Philip	Cole	Male	XS	$9.83	
4	Johnny	Martin	Male	2XL	$5.91	
5	Sean	Holmes	Male	XL	$3.12	
6	Steve	Dunn	Male	S	$7.94	
7	Ronald	Schmidt	Male	S	$2.00	
8	Richard	Wright	Male	2XL	$6.24	
9	Diane	Roberts	Female	L	$9.83	
10	Joshua	Weaver	Male	M	$0.72	
11	Teresa	Schmidt	Female	M	$7.61	
12	Lois	Burke	Female	S	$8.24	
13	Alan	Mcdonald	Male	M	$7.31	
14	Randy	Edwards	Male	2XL	$8.39	
15	Raymond	Flores	Male	3XL	$8.53	

	A	B	C	D
1	LastName	(All)		
2				
3	Sum of Cost	Column Lal		
4	Row Labels	Female	Male	Grand Total
5	2XL		$ 20.54	$ 20.54
6	3XL		$ 8.53	$ 8.53
7	L	$ 9.83		$ 9.83
8	M	$ 7.61	$ 8.03	$ 15.64
9	S	$ 8.24	$ 9.94	$ 18.18
10	XL		$ 3.12	$ 3.12
11	XS	$ 7.56	$ 9.83	$ 17.39
12	Grand Total	$ 33.24	$ 59.99	$ 93.23
13				

Wenn mehrere Pivot-Tabellen auf der Grundlage derselben Quelldaten benötigt werden, kann der

Pivot-Cache als interner Datenspeicher für jede der Pivot-Tabellen wiederverwendet werden. Dies ist eine bewährte Methode, da dadurch Speicher eingespart und die Größe der Excel-Datei zum Speichern reduziert wird.



	A	B	C	D	E	F
1	**FirstName**	**LastName**	**Gender**	**ShirtSize**	**Cost**	
2	Mildred	Ferguson	Female	XS	$7.56	
3	Philip	Cole	Male	XS	$9.83	
4	Johnny	Martin	Male	2XL	$5.91	
5	Sean	Holmes	Male	XL	$3.12	
6	Steve	Dunn	Male	S	$7.94	
7	Ronald	Schmidt	Male	S	$2.00	
8	Richard	Wright	Male	2XL	$6.24	
9	Diane	Roberts	Female	L	$9.83	
10	Joshua	Weaver	Male	M	$0.72	
11	Teresa	Schmidt	Female	M	$7.61	
12	Lois	Burke	Female	S	$8.24	
13	Alan	Mcdonald	Male	M	$7.31	
14	Randy	Edwards	Male	2XL	$8.39	
15	Raymond	Flores	Male	3XL	$8.53	

	A	B	C	D
1	LastName	(All)		
2				
3	**Sum of Cost**	**Column Lal**		
4	**Row Labels**	**Female**	**Male**	**Grand Total**
5	2XL		$ 20.54	$ 20.54
6	3XL		$ 8.53	$ 8.53
7	L	$ 9.83		$ 9.83
8	M	$ 7.61	$ 8.03	$ 15.64
9	S	$ 8.24	$ 9.94	$ 18.18
10	XL		$ 3.12	$ 3.12
11	XS	$ 7.56	$ 9.83	$ 17.39
12	**Grand Total**	**$ 33.24**	**$ 59.99**	**$ 93.23**
13				

So erstellen Sie eine Pivot-Tabelle basierend auf den in den Abbildungen oben dargestellten Quelldaten:

```
Sub test()
    Dim pt As PivotTable
    Set pt = CreatePivotTable(ThisWorkbook.Sheets("Sheet1").Range("A1:E15"))
End Sub

Function CreatePivotTable(ByRef srcData As Range) As PivotTable
    '--- creates a Pivot Table from the given source data and
    '    assumes that the first row contains valid header data
    '    for the columns
    Dim thisPivot As PivotTable
    Dim dataSheet As Worksheet
    Dim ptSheet As Worksheet
    Dim ptCache As PivotCache

    '--- the Pivot Cache must be created first...
    Set ptCache = ThisWorkbook.PivotCaches.Create(SourceType:=xlDatabase, _
```

```
                                                SourceData:=srcData)
    '--- ... then use the Pivot Cache to create the Table
    Set ptSheet = ThisWorkbook.Sheets.Add
    Set thisPivot = ptCache.CreatePivotTable(TableDestination:=ptSheet.Range("A3"))
    Set CreatePivotTable = thisPivot
End Function
```

Verweist auf das MSDN-Pivot-Tabellenobjekt

Pivot-Tabellenbereiche

Diese hervorragenden Referenzquellen bieten Beschreibungen und Abbildungen der verschiedenen Bereiche in Pivot-Tabellen.

Verweise

- Referenzieren von Pivot-Tabellenbereichen in VBA - aus Jon Peltiers Tech-Blog
- Referenzieren eines Excel-Pivot-Tabellenbereichs mit VBA - von globaliconnect Excel VBA

Felder zu einer Pivot-Tabelle hinzufügen

Beachten Sie beim Hinzufügen von Feldern zu einer Pivot-Tabelle zwei wichtige Punkte: Ausrichtung und Position. Manchmal kann ein Entwickler davon ausgehen, wo ein Feld platziert wird. Daher ist es immer klarer, diese Parameter explizit zu definieren. Diese Aktionen betreffen nur die angegebene Pivot-Tabelle, nicht den Pivot-Cache.

```
Dim thisPivot As PivotTable
Dim ptSheet As Worksheet
Dim ptField As PivotField

Set ptSheet = ThisWorkbook.Sheets("SheetNameWithPivotTable")
Set thisPivot = ptSheet.PivotTables(1)

With thisPivot
    Set ptField = .PivotFields("Gender")
    ptField.Orientation = xlRowField
    ptField.Position = 1
    Set ptField = .PivotFields("LastName")
    ptField.Orientation = xlRowField
    ptField.Position = 2
    Set ptField = .PivotFields("ShirtSize")
    ptField.Orientation = xlColumnField
    ptField.Position = 1
    Set ptField = .AddDataField(.PivotFields("Cost"), "Sum of Cost", xlSum)
    .InGridDropZones = True
    .RowAxisLayout xlTabularRow
End With
```

Formatieren der Pivot-Tabellendaten

In diesem Beispiel werden mehrere Formate im Datenbereich (`DataBodyRange`) der angegebenen Pivot-Tabelle geändert / festgelegt. Alle formatierbare Parameter in einem `Range` zur Verfügung.

Das Formatieren der Daten wirkt sich nur auf die Pivot-Tabelle selbst aus, nicht auf den Pivot-Cache.

HINWEIS: Die Eigenschaft hat den Namen `TableStyle2` da die `TableStyle` Eigenschaft kein Mitglied der `PivotTable` von `PivotTable` .

```
Dim thisPivot As PivotTable
Dim ptSheet As Worksheet
Dim ptField As PivotField

Set ptSheet = ThisWorkbook.Sheets("SheetNameWithPivotTable")
Set thisPivot = ptSheet.PivotTables(1)

With thisPivot
    .DataBodyRange.NumberFormat = "_($* #,##0.00_);_($* (#,##0.00);_($* "-"??_);_(@_)"
    .DataBodyRange.HorizontalAlignment = xlRight
    .ColumnRange.HorizontalAlignment = xlCenter
    .TableStyle2 = "PivotStyleMedium9"
End With
```

Pivot-Tabellen online lesen: https://riptutorial.com/de/excel-vba/topic/3797/pivot-tabellen

Kapitel 24: PowerPoint-Integration über VBA

Bemerkungen

In diesem Abschnitt werden verschiedene Möglichkeiten für die Interaktion mit PowerPoint über VBA beschrieben. Von der Anzeige von Daten auf Folien bis zur Erstellung von Diagrammen ist PowerPoint in Verbindung mit Excel ein sehr leistungsfähiges Werkzeug. In diesem Abschnitt soll daher gezeigt werden, wie VBA zur Automatisierung dieser Interaktion verwendet werden kann.

Examples

Die Grundlagen: Starten von PowerPoint über VBA

Es gibt zwar viele Parameter, die geändert werden können, und Variationen, die abhängig von der gewünschten Funktionalität hinzugefügt werden können. In diesem Beispiel wird das grundlegende Framework für das Starten von PowerPoint beschrieben.

> **Hinweis:** Dieser Code erfordert, dass die PowerPoint-Referenz zum aktiven VBA-Projekt hinzugefügt wurde. Siehe die Referenzen Dokumentation Eintrag zu lernen , wie die Referenz zu ermöglichen.

Definieren Sie zunächst Variablen für die Anwendungs-, Präsentations- und Folienobjekte. Dies kann zwar mit einer späten Bindung erfolgen, es ist jedoch immer am besten, die frühe Bindung zu verwenden, falls zutreffend.

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

Öffnen oder erstellen Sie eine neue Instanz der PowerPoint-Anwendung. Hier wird der Aufruf `On Error Resume Next` verwendet, um zu verhindern, dass ein Fehler von `GetObject` wenn PowerPoint noch nicht geöffnet wurde. Eine ausführlichere Erklärung finden Sie im Beispiel zur Fehlerbehandlung des Themas Best Practices.

```
'Open PPT if not running, otherwise select active instance
On Error Resume Next
Set PPApp = GetObject(, "PowerPoint.Application")
On Error GoTo ErrHandler
If PPApp Is Nothing Then
    'Open PowerPoint
    Set PPApp = CreateObject("PowerPoint.Application")
    PPApp.Visible = True
End If
```

Nachdem die Anwendung gestartet wurde, wird eine neue Präsentation und anschließend enthaltene Folie zur Verwendung generiert.

```
'Generate new Presentation and slide for graphic creation
Set PPPres = PPApp.Presentations.Add
Set PPSlide = PPPres.Slides.Add(1, ppLayoutBlank)

'Here, the slide type is set to the 4:3 shape with slide numbers enabled and the window
'maximized on the screen.  These properties can, of course, be altered as needed

PPApp.ActiveWindow.ViewType = ppViewSlide
PPPres.PageSetup.SlideOrientation = msoOrientationHorizontal
PPPres.PageSetup.SlideSize = ppSlideSizeOnScreen
PPPres.SlideMaster.HeadersFooters.SlideNumber.Visible = msoTrue
PPApp.ActiveWindow.WindowState = ppWindowMaximized
```

Nach Abschluss dieses Codes wird ein neues PowerPoint-Fenster mit einer leeren Folie geöffnet. Mithilfe der Objektvariablen können Formen, Text, Grafiken und Excel-Bereiche nach Wunsch hinzugefügt werden

PowerPoint-Integration über VBA online lesen: https://riptutorial.com/de/excel-vba/topic/2327/powerpoint-integration-uber-vba

Kapitel 25: SQL in Excel VBA - Best Practices

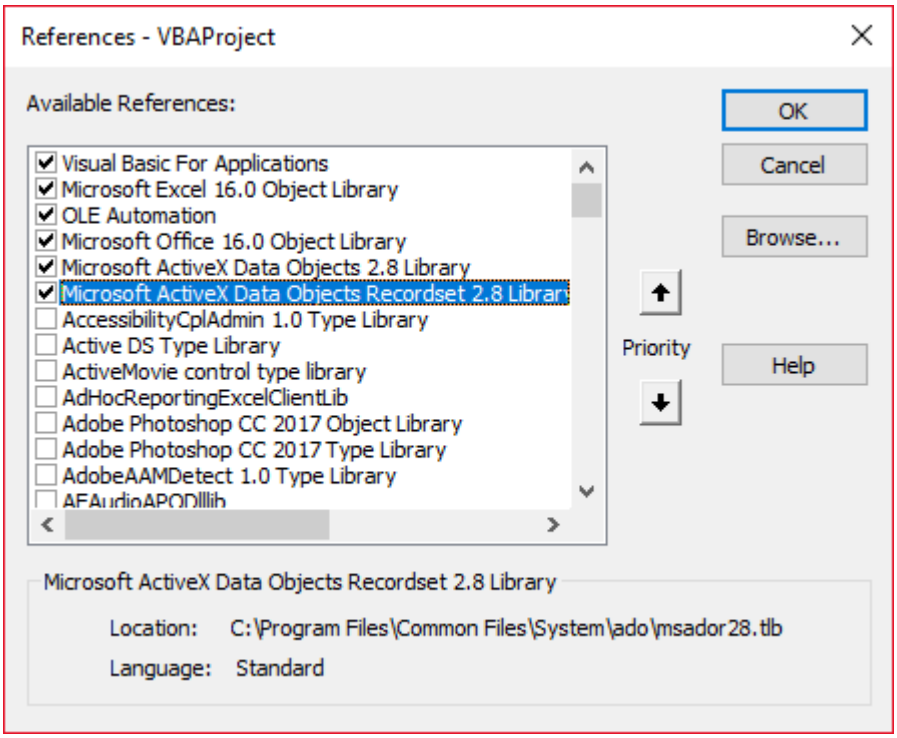
Examples

Wie verwende ich ADODB.Connection in VBA?

Bedarf:

Fügen Sie dem Projekt folgende Referenzen hinzu:

- Microsoft ActiveX Data Objects 2.8-Bibliothek
- Microsoft ActiveX Data Objects-Recordset 2.8-Bibliothek



Variablen deklarieren

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

Verbindung herstellen

ein. mit Windows-Authentifizierung

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

b. mit der SQL Server-Authentifizierung

```
Private Sub OpenConnection2(pServer As String, pCatalog As String, pUser As String, pPsw As
String)
    Call mDataBase.Open("Provider=SQLOLEDB;Initial Catalog=" & pCatalog & ";Data Source=" &
pServer & ";Integrated Security=SSPI;User ID=" & pUser & ";Password=" & pPsw)
    mCmd.ActiveConnection = mDataBase
End Sub
```

Führen Sie den SQL-Befehl aus

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

Daten aus Datensatz lesen

```
Private Sub ReadRS()
    Do While Not (mRS.EOF)
        Debug.Print "ShipperID: " & mRS.Fields("ShipperID").Value & " CompanyName: " &
mRS.Fields("CompanyName").Value & " Phone: " & mRS.Fields("Phone").Value
        Call mRS.MoveNext
    Loop
End Sub
```

Verbindung schließen

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

Wie benutze ich es?

```
Public Sub Program()
    Call OpenConnection("ServerName", "NORTHWND")
```

```
    Call ExecuteCmd("INSERT INTO [NORTHWND].[dbo].[Shippers]([CompanyName],[Phone]) Values
('speedy shipping','(503) 555-1234')")
    Call ExecuteCmd("SELECT * FROM [NORTHWND].[dbo].[Shippers]")
    Call ReadRS
    Call CloseConnection
End Sub
```

Ergebnis

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

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

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

ShipperID: 4 CompanyName: schneller Versand Telefon: (503) 555-1234

SQL in Excel VBA - Best Practices online lesen: https://riptutorial.com/de/excel-vba/topic/9958/sql-in-excel-vba---best-practices

Kapitel 26: Tipps und Tricks zu Excel VBA

Bemerkungen

Dieses Thema umfasst eine Vielzahl nützlicher Tipps und Tricks, die SO-Benutzer durch ihre Codierkenntnisse entdeckt haben. Dies sind oft Beispiele für Möglichkeiten, häufige Frustrationen zu umgehen oder Excel auf eine "intelligentere" Weise zu verwenden.

Examples

Verwenden von xlVeryHidden Sheets

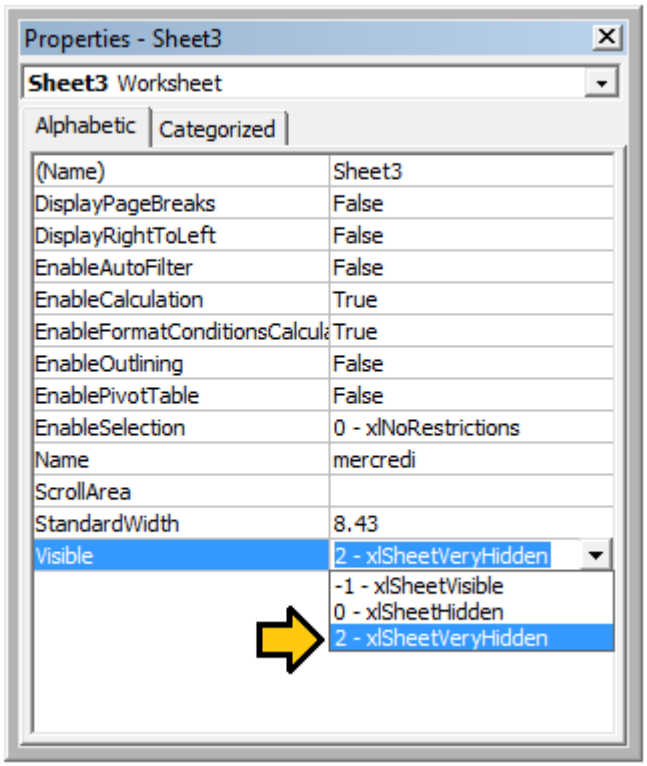
Arbeitsblätter in Excel haben drei Optionen für die Eigenschaft `Visible` . Diese Optionen werden durch Konstanten in der `xlSheetVisibility` Enumeration dargestellt und lauten wie folgt:

1. `xlVisible` oder `xlSheetVisible` Wert: `-1` (Standardeinstellung für neue `xlSheetVisible`)
2. `xlHidden` oder `xlSheetHidden` Wert: `0`
3. `xlVeryHidden` `xlSheetVeryHidden` Wert: `2`

Sichtbare Blätter repräsentieren die Standardsichtbarkeit für Blätter. Sie sind in der Tab-Leiste sichtbar und können frei ausgewählt und angezeigt werden. Ausgeblendete Blätter werden aus der Blatt-Registerkartenleiste ausgeblendet und können daher nicht ausgewählt werden. Ausgeblendete Blätter können jedoch aus dem Excel-Fenster ausgeblendet werden, indem Sie mit der rechten Maustaste auf die Blattregister klicken und "Einblenden" auswählen.

Sehr verborgene Tabellen sind dagegen *nur* über den Visual Basic-Editor zugänglich. Dies macht sie zu einem unglaublich nützlichen Werkzeug zum Speichern von Daten zwischen verschiedenen Instanzen von Excel sowie zum Speichern von Daten, die für Endbenutzer verborgen sein sollten. Der Zugriff auf die Tabellen ist über einen Namen innerhalb des VBA-Codes möglich, sodass die gespeicherten Daten einfach verwendet werden können.

Um die .Visible-Eigenschaft eines Arbeitsblatts manuell in xlSheetVeryHidden zu ändern, öffnen Sie das Eigenschaftenfenster des VBE (`F4`), wählen Sie das zu ändernde Arbeitsblatt aus und verwenden Sie das Dropdown-Menü in der dreizehnten Zeile, um Ihre Auswahl zu treffen.



Um die .Visible-Eigenschaft eines Arbeitsblatts im Code in xlSheetVeryHidden[1] zu ändern, greifen Sie in ähnlicher Weise auf die .Visible-Eigenschaft zu und weisen Sie einen neuen Wert zu.

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

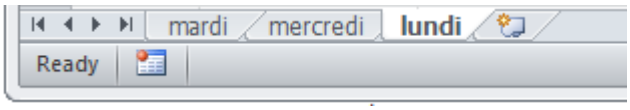
---

[1] Sowohl **xlVeryHidden** als auch **xlSheetVeryHidden** geben den numerischen Wert **2 zurück** (sie sind austauschbar).

Arbeitsblatt .Name, .Index oder .CodeName

Wir wissen, dass 'Best Practice' vorschreibt, dass das übergeordnete Arbeitsblatt eines Bereichsobjekts explizit referenziert werden sollte. Ein Arbeitsblatt kann über seine .Name-Eigenschaft, seine numerische .Index-Eigenschaft oder seine .CodeName-Eigenschaft referenziert werden. Ein Benutzer kann jedoch die Arbeitsblattwarteschlange neu ordnen, indem er einfach eine Namensregisterkarte zieht oder das Arbeitsblatt mit einem Doppelklick auf dieselbe Registerkarte und einige umbenennt Eingeben einer ungeschützten Arbeitsmappe.

Betrachten Sie ein Standard-drei-Arbeitsblatt. Sie haben die drei Arbeitsblätter Montag, Dienstag und Mittwoch in dieser Reihenfolge umbenannt und VBA-Unterprozeduren codiert, die auf diese verweisen. Angenommen, ein Benutzer kommt und entscheidet, dass Montag am Ende der Arbeitsblattwarteschlange steht, dann kommt ein anderer und entscheidet, dass die Namen der Arbeitsblätter auf Französisch besser aussehen. Sie haben jetzt eine Arbeitsmappe mit einer Warteschlange für die Registerkarte "Name des Arbeitsblatts", die etwa wie folgt aussieht.



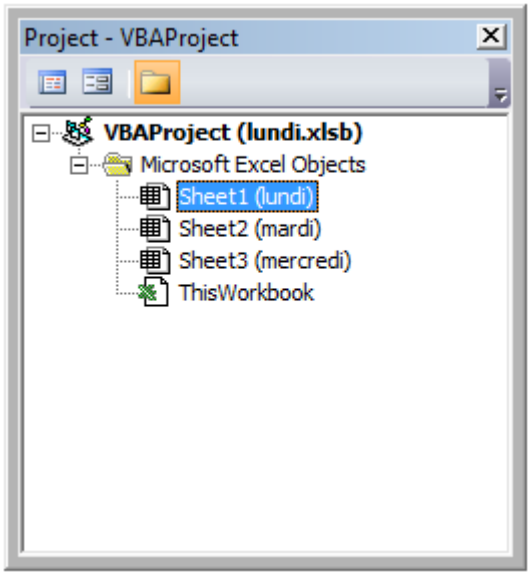
Wenn Sie eine der folgenden Arbeitsblatt-Referenzmethoden verwendet hätten, wäre Ihr Code jetzt beschädigt.

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

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

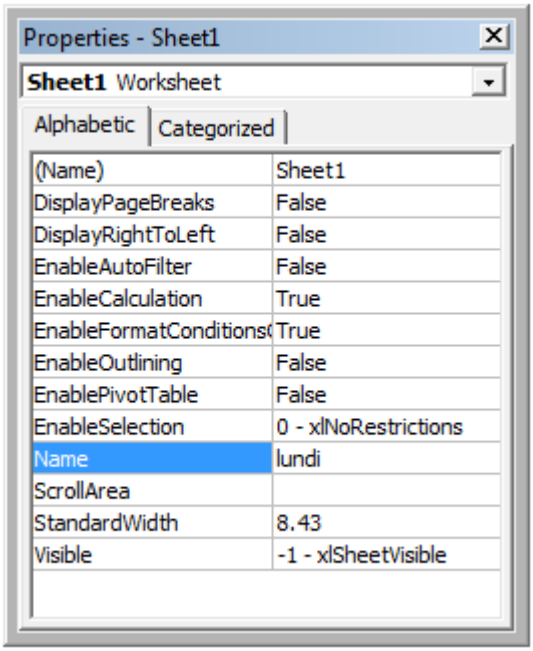
Sowohl die ursprüngliche Reihenfolge als auch der Name des ursprünglichen Arbeitsblatts wurden beeinträchtigt. Wenn Sie jedoch die .CodeName-Eigenschaft des Arbeitsblatts verwendet hätten, wäre Ihre Subprozedur weiterhin betriebsbereit

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

Das folgende Bild zeigt das VBA-Projektfenster ([Strg] + R), in dem die Arbeitsblätter nach .CodeName und dann nach .Name (in Klammern) aufgeführt sind. Die Reihenfolge, in der sie angezeigt werden, ändert sich nicht. Der Ordnungsindex .Index wird anhand der Reihenfolge ermittelt, in der sie in der Warteschlange der Registerkarte "Name" im Arbeitsblattfenster angezeigt werden.



Das Umbenennen eines .CodeName ist zwar ungewöhnlich, aber es ist nicht unmöglich. Öffnen Sie einfach das Eigenschaftenfenster der VBE ([F4]).



Das Arbeitsblatt .CodeName befindet sich in der ersten Zeile. Das Arbeitsblatt .Name befindet sich im zehnten. Beide sind editierbar.

Verwenden von Zeichenfolgen mit Trennzeichen anstelle von dynamischen Arrays

Die Verwendung dynamischer Arrays in VBA kann bei sehr großen Datensätzen recht unhandlich und zeitaufwändig sein. Wenn Sie einfache Datentypen in einem dynamischen Array (Strings, Numbers, Booleans usw.) speichern, können Sie die `ReDim Preserve` Anweisungen vermeiden, die für dynamische Arrays in VBA erforderlich sind, indem Sie die `Split()` Funktion mit einigen cleveren String-Prozeduren verwenden. Zum Beispiel betrachten wir eine Schleife, die basierend auf einigen Bedingungen eine Reihe von Werten aus einem Bereich zu einer Zeichenfolge hinzufügt, und verwendet dann diese Zeichenfolge, um die Werte einer ListBox aufzufüllen.

```
Private Sub UserForm_Initialize()

Dim Count As Long, DataString As String, Delimiter As String

For Count = 1 To ActiveSheet.UsedRows.Count
    If ActiveSheet.Range("A" & Count).Value <> "Your Condition" Then
        RowString = RowString & Delimiter & ActiveSheet.Range("A" & Count).Value
        Delimiter = "><" 'By setting the delimiter here in the loop, you prevent an extra
occurance of the delimiter within the string
    End If
Next Count

ListBox1.List = Split(DataString, Delimiter)

End Sub
```

Die `Delimiter` selbst kann auf einen beliebigen Wert gesetzt werden, es ist jedoch ratsam, einen Wert zu wählen, der in der Menge nicht natürlich vorkommt. Angenommen, Sie haben beispielsweise eine Spalte mit Datumsangaben verarbeitet. In diesem Fall verwenden Sie `.` , `-` oder `/` wäre unklug als Trennzeichen, da die Datumsangaben so formatiert werden könnten, dass

eines dieser Werte verwendet wird und mehr Datenpunkte generiert werden, als Sie erwartet hatten.

Hinweis: Die Verwendung dieser Methode unterliegt Einschränkungen (insbesondere der maximalen Länge von Zeichenfolgen). Daher sollte sie bei sehr großen Datensätzen mit Vorsicht verwendet werden. Dies ist nicht unbedingt die schnellste oder effektivste Methode zum Erstellen dynamischer Arrays in VBA, aber es ist eine praktikable Alternative.

Doppelklicken Sie auf Ereignis für Excel-Shapes

Standardmäßig haben Formen in Excel keine bestimmte Methode für die Verarbeitung von Einzel- oder Doppelklicks. Sie enthalten nur die Eigenschaft "OnAction", mit der Sie Klicks verarbeiten können. Es kann jedoch Fälle geben, in denen Ihr Code Sie dazu zwingt, bei einem Doppelklick anders (oder ausschließlich) zu handeln. Die folgende Subroutine kann zu Ihrem VBA-Projekt hinzugefügt werden. Wenn Sie als `OnAction` Routine für Ihre Form festgelegt ist, können Sie Doppelklicks ausführen.

```
Public Const DOUBLECLICK_WAIT as Double = 0.25 'Modify to adjust click delay
Public LastClickObj As String, LastClickTime As Date

Sub ShapeDoubleClick()

    If LastClickObj = "" Then
        LastClickObj = Application.Caller
        LastClickTime = CDbl(Timer)
    Else
        If CDbl(Timer) - LastClickTime > DOUBLECLICK_WAIT Then
            LastClickObj = Application.Caller
            LastClickTime = CDbl(Timer)
        Else
            If LastClickObj = Application.Caller Then
                'Your desired Double Click code here
                LastClickObj = ""
            Else
                LastClickObj = Application.Caller
                LastClickTime = CDbl(Timer)
            End If
        End If
    End If

End Sub
```

Diese Routine bewirkt, dass die Form den ersten Klick funktionell ignoriert und nur den gewünschten Code innerhalb des angegebenen Zeitraums auf den zweiten Klick ausführt.

Dateidialog öffnen - Mehrere Dateien

Diese Subroutine ist ein schnelles Beispiel, wie einem Benutzer ermöglicht wird, mehrere Dateien auszuwählen und dann mit diesen Dateipfaden etwas zu tun, z.

```
Option Explicit

Sub OpenMultipleFiles()
```

```
    Dim fd As FileDialog
    Dim fileChosen As Integer
    Dim i As Integer
    Dim basename As String
    Dim fso As Variant
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set fd = Application.FileDialog(msoFileDialogFilePicker)
    basename = fso.getBaseName(ActiveWorkbook.Name)
    fd.InitialFileName = ActiveWorkbook.Path ' Set Default Location to the Active Workbook
Path
    fd.InitialView = msoFileDialogViewList
    fd.AllowMultiSelect = True

    fileChosen = fd.Show
    If fileChosen = -1 Then
        'open each of the files chosen
        For i = 1 To fd.SelectedItems.Count
            Debug.Print (fd.SelectedItems(i))
            Dim fileName As String
            ' do something with the files.
            fileName = fso.getFileName(fd.SelectedItems(i))
            Debug.Print (fileName)
        Next i
    End If

End Sub
```

Tipps und Tricks zu Excel VBA online lesen: https://riptutorial.com/de/excel-vba/topic/2240/tipps-und-tricks-zu-excel-vba

Kapitel 27: VBA-Best Practices

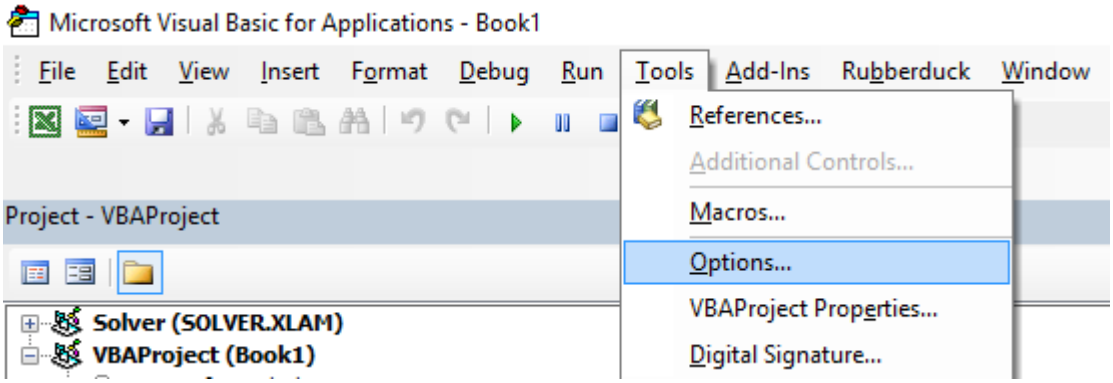
Bemerkungen

Wir kennen sie alle, aber diese Praktiken sind für jemanden, der mit dem Programmieren in VBA beginnt, weitaus weniger offensichtlich.

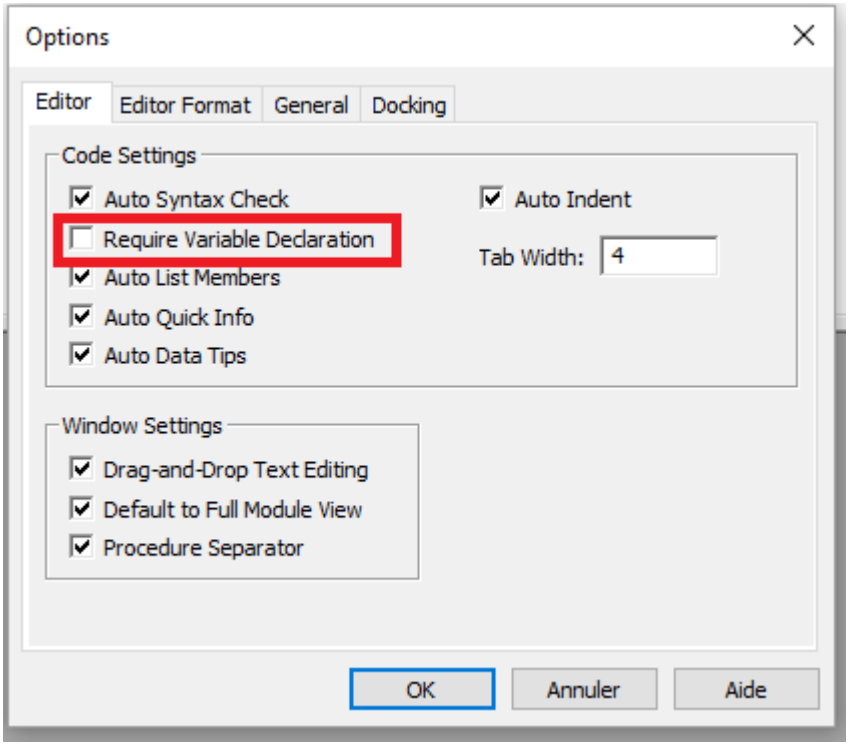
Examples

Verwenden Sie IMMER "Option Explicit"

Wählen Sie im VBA-Editor-Fenster im Menü "Extras" die Option "Optionen":



Stellen Sie dann auf der Registerkarte "Editor" sicher, dass "Variablendeklaration erforderlich" aktiviert ist:



Wenn Sie diese Option auswählen, wird die `Option Explicit` automatisch an der Spitze jedes VBA-Moduls angezeigt.

> **Kleiner Hinweis:** Dies gilt für die bisher noch nicht geöffneten Module, Klassenmodule usw. Wenn Sie sich beispielsweise bereits den Code von `Sheet1` bevor Sie die Option "Deklaration der Variablen erforderlich" aktivieren, wird `Option Explicit` nicht hinzugefügt!

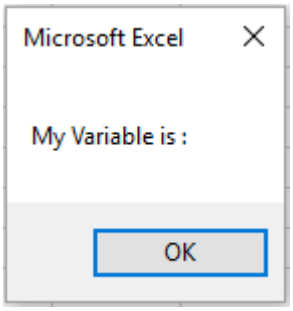
`Option Explicit` muss jede Variable vor der Verwendung definiert werden, z. B. mit einer `Dim` Anweisung. Wenn die `Option Explicit` aktiviert ist, nimmt der VBA-Compiler an, dass ein nicht erkanntes Wort eine neue Variable des `Variant` Typs ist. Dies führt zu äußerst schwer zu findenden Fehlern im Zusammenhang mit Tippfehlern. `Option Explicit` die `Option Explicit` aktiviert ist, wird bei nicht erkannten Wörtern ein Kompilierungsfehler ausgegeben, der die fehlerhafte Zeile anzeigt.

Beispiel:

Wenn Sie den folgenden Code ausführen:

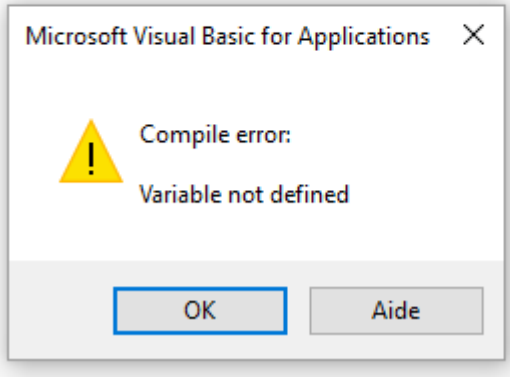
```
Sub Test()
  my_variable = 12
  MsgBox "My Variable is : " & myvariable
End Sub
```

Sie erhalten folgende Nachricht:



Sie haben einen Fehler gemacht, indem Sie `myvariable` anstelle von `my_variable` haben. Das Meldungsfeld zeigt dann eine leere Variable an. Wenn Sie die `Option Explicit` , ist dieser Fehler nicht möglich, da eine Fehlermeldung zum Kompilieren angezeigt wird.

```
Option Explicit

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

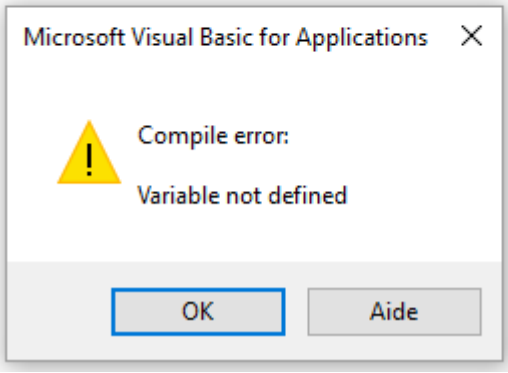


Wenn Sie jetzt die korrekte Deklaration hinzufügen:

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

Mit `myvariable` erhalten Sie eine Fehlermeldung, die den Fehler genau `myvariable` :

```
Option Explicit

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



Hinweis zu Option Explicit und Arrays (Deklarieren eines dynamischen Arrays):

Mit der ReDim-Anweisung können Sie ein Array implizit innerhalb einer Prozedur deklarieren.

- Achten Sie darauf, dass Sie den Namen des Arrays nicht falsch eingeben, wenn

Sie die ReDim-Anweisung verwenden

- Auch wenn die Option Explicit-Anweisung im Modul enthalten ist, wird ein neues Array erstellt

```
Dim arr() as Long

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

Arbeit mit Arrays, nicht mit Ranges

Office-Blog - Best Practices für die Excel VBA-Leistungscodierung

Oft wird die beste Leistung erzielt, indem der Einsatz von `Range` so weit wie möglich vermieden wird. In diesem Beispiel lesen wir ein gesamtes `Range` Objekt in ein Array ein, quadrieren jede Zahl im Array und bringen das Array dann wieder zum `Range` . Dies greift nur zweimal auf `Range` , wohingegen eine Schleife für das Lesen / Schreiben 20-mal darauf zugreifen würde.

```
Option Explicit
Sub WorkWithArrayExample()

Dim DataRange As Variant
Dim Irow As Long
Dim Icol As Integer
DataRange = ActiveSheet.Range("A1:A10").Value ' read all the values at once from the Excel
grid, put into an array

For Irow = LBound(DataRange,1) To UBound(DataRange, 1) ' Get the number of rows.
  For Icol = LBound(DataRange,2) To UBound(DataRange, 2) ' Get the number of columns.
    DataRange(Irow, Icol) = DataRange(Irow, Icol) * DataRange(Irow, Icol) ' cell.value^2
  Next Icol
Next Irow
ActiveSheet.Range("A1:A10").Value = DataRange ' writes all the results back to the range at
once

End Sub
```

Weitere Tipps und Informationen zu zeitgesteuerten Beispielen finden Sie in Charles Williams 'VBA-UDFs für Schreiben (Teil 1) und anderen Artikeln der Serie .

Verwenden Sie, falls verfügbar, VB-Konstanten

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

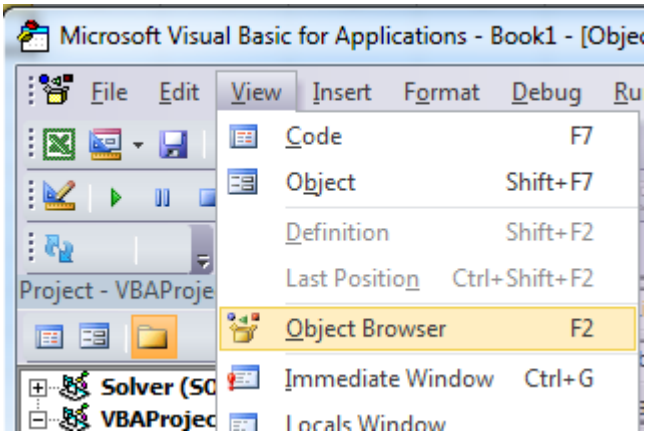
kann anstelle von verwendet werden

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

um die Lesbarkeit zu verbessern.

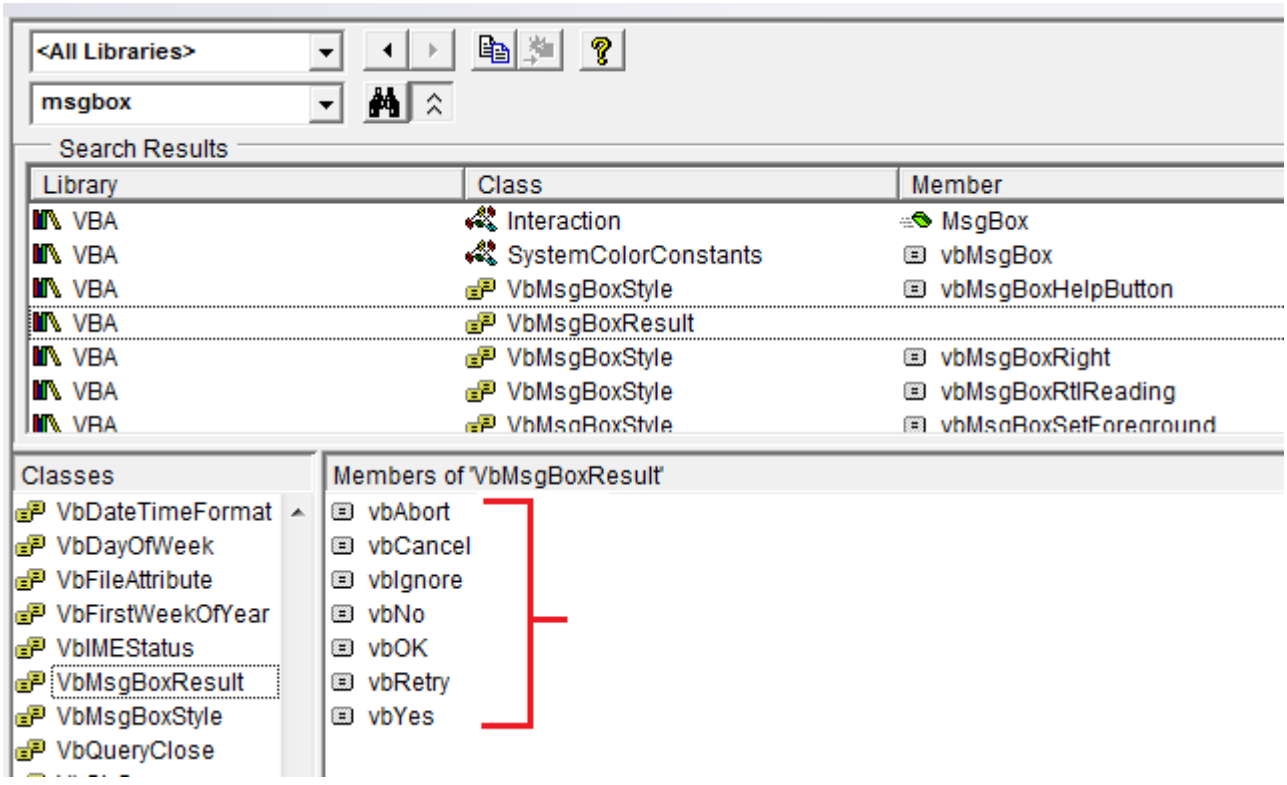
Verwenden Sie den *Objektbrowser* , um verfügbare VB-Konstanten zu finden. *Ansicht* →

Objektbrowser oder `F2 im` VB-Editor.



Geben Sie die zu suchende Klasse ein

Mitglieder anzeigen



Benennen Sie beschreibende Variablen

Beschreibende Namen und Strukturen in Ihrem Code helfen, Kommentare zu machen

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

ductArea = ductWidth * ductHeight
```

ist besser als

```
Dim a, w, h

a = w * h
```

Dies ist besonders hilfreich, wenn Sie Daten von einem Ort an einen anderen kopieren, egal ob es sich um eine Zelle, einen Bereich, ein Arbeitsblatt oder eine Arbeitsmappe handelt. Helfen Sie sich, indem Sie Namen wie diese verwenden:

```
Dim myWB As Workbook
Dim srcWS As Worksheet
Dim destWS As Worksheet
Dim srcData As Range
Dim destData As Range

Set myWB = ActiveWorkbook
Set srcWS = myWB.Sheets("Sheet1")
Set destWS = myWB.Sheets("Sheet2")
Set srcData = srcWS.Range("A1:A10")
Set destData = destWS.Range("B11:B20")
destData = srcData
```

Wenn Sie mehrere Variablen in einer Zeile deklarieren, müssen Sie für *jede* Variable einen Typ angeben, wie:

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

Im Folgenden wird nur die letzte Variable deklariert, und die erste Variable bleibt `Variant` :

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

Fehlerbehandlung

Eine gute Fehlerbehandlung verhindert, dass Endbenutzer VBA-Laufzeitfehler erkennen, und hilft dem Entwickler, Fehler leicht zu diagnostizieren und zu beheben.

Es gibt drei Hauptmethoden für die Fehlerbehandlung in VBA, von denen zwei für verteilte Programme vermieden werden sollten, sofern dies nicht ausdrücklich im Code erforderlich ist.

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

oder

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

Lieber verwenden:

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

On Error GoTo 0

Wenn in Ihrem Code keine Fehlerbehandlung festgelegt ist, ist `On Error GoTo 0` der Standardfehlerhandler. In diesem Modus wird bei Laufzeitfehlern die typische VBA-Fehlermeldung angezeigt, sodass Sie den Code entweder beenden oder in den `debug` Modus wechseln können, um die Quelle zu identifizieren. Beim Schreiben von Code ist diese Methode die einfachste und nützlichste Methode. Sie sollte jedoch immer für Code vermieden werden, der an Endbenutzer verteilt wird, da diese Methode für Endbenutzer sehr unansehnlich und schwer verständlich ist.

On Error Resume Next

`On Error Resume Next` bewirkt, dass VBA alle Fehler ignoriert, die zur Laufzeit für alle Zeilen nach dem Fehleraufruf ausgegeben werden, bis der Fehlerbehandler geändert wurde. In sehr speziellen Fällen kann diese Zeile nützlich sein, sollte aber außerhalb dieser Fälle vermieden werden. Wenn Sie beispielsweise ein separates Programm aus einem Excel-Makro starten, kann der Aufruf von `On Error Resume Next` hilfreich sein, wenn Sie nicht sicher sind, ob das Programm bereits geöffnet ist oder nicht:

```
'In this example, we open an instance of Powerpoint using the On Error Resume Next call
Dim PPApp As PowerPoint.Application
Dim PPPres As PowerPoint.Presentation
Dim PPSlide As PowerPoint.Slide

'Open PPT if not running, otherwise select active instance
On Error Resume Next
Set PPApp = GetObject(, "PowerPoint.Application")
On Error GoTo ErrHandler
If PPApp Is Nothing Then
    'Open PowerPoint
    Set PPApp = CreateObject("PowerPoint.Application")
    PPApp.Visible = True
End If
```

Wenn wir den Aufruf `On Error Resume Next` nicht verwendet hätten und die Powerpoint-Anwendung nicht bereits geöffnet war, würde die `GetObject` Methode einen Fehler `GetObject` . Daher war `On Error Resume Next` erforderlich, um zu vermeiden, dass zwei Instanzen der Anwendung erstellt werden.

Hinweis: Es empfiehlt sich auch, den Fehlerbehandler *sofort* zurückzusetzen, sobald Sie den Aufruf `On Error Resume Next` nicht mehr benötigen

---

On Error GoTo <Zeile>

Diese Methode zur Fehlerbehandlung wird für den gesamten Code empfohlen, der an andere Benutzer verteilt wird. Dadurch kann der Programmierer genau steuern, wie VBA einen Fehler

behandelt, indem er den Code an die angegebene Zeile sendet. Das Tag kann mit einer beliebigen Zeichenfolge (einschließlich numerischer Zeichenfolgen) gefüllt werden und sendet den Code an die entsprechende Zeichenfolge, auf die ein Doppelpunkt folgt. Mehrere Fehlerbehandlungsblöcke können verwendet werden, indem `On Error GoTo <line>` . Die folgende Subroutine veranschaulicht die Syntax eines `On Error GoTo <line>` -Aufrufs.

Hinweis: Es ist wichtig, dass die `Exit Sub` Zeile über dem ersten Error-Handler und vor jedem nachfolgenden Error-Handler platziert wird, um zu verhindern, dass der Code in den Block eintritt, *ohne dass* ein Fehler aufgerufen wird. Daher ist es für Funktion und Lesbarkeit empfehlenswert, Fehlerbehandlungsroutinen am Ende eines Codeblocks zu platzieren.

```
Sub YourMethodName()
    On Error GoTo errorHandler
    ' Insert code here
    On Error GoTo secondErrorHandler

    Exit Sub 'The exit sub line is essential, as the code will otherwise
             'continue running into the error handling block, likely causing an error

errorHandler:
    MsgBox "Error " & Err.Number & ": " & Err.Description & " in " & _
        VBE.ActiveCodePane.CodeModule, vbOKOnly, "Error"
    Exit Sub

secondErrorHandler:
    If Err.Number = 424 Then 'Object not found error (purely for illustration)
        Application.ScreenUpdating = True
        Application.EnableEvents = True
        Exit Sub
    Else
        MsgBox "Error " & Err.Number & ": " & Err.Desctription
        Application.ScreenUpdating = True
        Application.EnableEvents = True
        Exit Sub
    End If
    Exit Sub

End Sub
```

Wenn Sie Ihre Methode mit Ihrem Fehlerbehandlungscode beenden, stellen Sie sicher, dass Sie bereinigen:

- Machen Sie alles rückgängig, was teilweise abgeschlossen ist
- Dateien schließen
- Bildschirmaktualisierung zurücksetzen
- Berechnungsmodus zurücksetzen
- Ereignisse zurücksetzen
- Setzen Sie den Mauszeiger zurück
- Ruft die Unload-Methode für Instanzen von Objekten auf, die nach dem `End Sub` bestehen bleiben
- Statusleiste zurücksetzen

Dokumentieren Sie Ihre Arbeit

Es empfiehlt sich, Ihre Arbeit für die spätere Verwendung zu dokumentieren, insbesondere wenn Sie für eine dynamische Arbeitslast programmieren. Gute Kommentare sollten erklären, warum der Code etwas tut, nicht was er tut.

```
Function Bonus(EmployeeTitle as String) as Double
    If EmployeeTitle = "Sales" Then
        Bonus = 0    'Sales representatives receive commission instead of a bonus
    Else
        Bonus = .10
    End If
End Function
```

Wenn Ihr Code so dunkel ist, dass er Kommentare benötigt, um zu erklären, was er tut, sollten Sie ihn umschreiben. Zum Beispiel anstelle von:

```
Sub CopySalesNumbers
    Dim IncludeWeekends as Boolean

    'Boolean values can be evaluated as an integer, -1 for True, 0 for False.
    'This is used here to adjust the range from 5 to 7 rows if including weekends.
    Range("A1:A" & 5 - (IncludeWeekends * 2)).Copy
    Range("B1").PasteSpecial
End Sub
```

Verdeutlichen Sie den Code, um leichter zu folgen, z.

```
Sub CopySalesNumbers
    Dim IncludeWeekends as Boolean
    Dim DaysinWeek as Integer

    If IncludeWeekends Then
        DaysinWeek = 7
    Else
        DaysinWeek = 5
    End If
    Range("A1:A" & DaysinWeek).Copy
    Range("B1").PasteSpecial
End Sub
```

Eigenschaften während der Makroausführung ausschalten

In jeder Programmiersprache wird **empfohlen, vorzeitige Optimierung** zu **vermeiden.** Wenn sich jedoch beim Testen herausstellt, dass Ihr Code zu langsam ausgeführt wird, können Sie etwas Geschwindigkeit erreichen, indem Sie einige Eigenschaften der Anwendung deaktivieren, während sie ausgeführt wird. Fügen Sie diesen Code einem Standardmodul hinzu:

```
Public Sub SpeedUp( _
    SpeedUpOn As Boolean, _
    Optional xlCalc as XlCalculation = xlCalculationAutomatic _
)
    With Application
        If SpeedUpOn Then
            .ScreenUpdating = False
            .Calculation = xlCalculationManual
```

```
            .EnableEvents = False
            .DisplayStatusBar = False 'in case you are not showing any messages
            ActiveSheet.DisplayPageBreaks = False 'note this is a sheet-level setting
        Else
            .ScreenUpdating = True
            .Calculation = xlCalc
            .EnableEvents = True
            .DisplayStatusBar = True
            ActiveSheet.DisplayPageBreaks = True
        End If
    End With
End Sub
```

Weitere Informationen finden Sie im Office Blog - Best Practices für die Excel VBA-Leistungscodierung

Und nennen Sie es einfach am Anfang und Ende von Makros:

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

    SpeedUp True

    'code here ...

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

Während diese weitestgehend als "Verbesserungen" für reguläre `Public Sub` Prozeduren betrachtet werden können, sollte das Deaktivieren der Ereignisbehandlung mit `Application.EnableEvents = False` für obligatorische private Ereignismakros `Worksheet_Change` und `Workbook_SheetChange` , die Werte in einem oder mehreren Arbeitsblättern ändern. Wenn Sie die Ereignisauslöser nicht deaktivieren, wird das Ereignismakro rekursiv über sich selbst ausgeführt, wenn sich ein Wert ändert. Dies kann zu einer "eingefrorenen" Arbeitsmappe führen. Denken Sie daran, Ereignisse wieder zu aktivieren, bevor Sie das Ereignismakro verlassen, möglicherweise durch einen Fehlerhandler "Safe Exit".

```
Option Explicit

Private Sub Worksheet_Change(ByVal Target As Range)
    If Not Intersect(Target, Range("A:A")) Is Nothing Then
        On Error GoTo bm_Safe_Exit
        Application.EnableEvents = False

        'code that may change a value on the worksheet goes here

    End If
bm_Safe_Exit:
    Application.EnableEvents = True
End Sub
```

Ein Nachteil: Durch das Deaktivieren dieser Einstellungen wird zwar die Laufzeit verbessert, das Debugging der Anwendung kann jedoch schwieriger werden. Wenn Ihr Code *nicht* ordnungsgemäß funktioniert, kommentieren Sie den `SpeedUp True` Aufruf aus, bis Sie das Problem gefunden haben.

Dies ist besonders wichtig, wenn Sie in Zellen in einem Arbeitsblatt schreiben und dann die berechneten Ergebnisse von Arbeitsblattfunktionen `xlCalculationManual` da `xlCalculationManual` die Berechnung der `xlCalculationManual` verhindert. Um dies zu `SpeedUp` , ohne `SpeedUp` zu deaktivieren, sollten Sie `Application.Calculate` einschließen, um eine Berechnung an bestimmten Punkten auszuführen.

HINWEIS: Da es sich um Eigenschaften der `Application` selbst handelt, müssen Sie sicherstellen, dass sie erneut aktiviert werden, bevor Ihr Makro beendet wird. Dies macht es besonders wichtig, Fehlerbehandlungsroutinen zu verwenden und mehrere Ausstiegspunkte (z. B. `End` oder `Unload Me`) zu vermeiden.

Mit Fehlerbehandlung:

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

    On Error GoTo Handler
    SpeedUp True

    'code here ...
    i = 1 / 0
CleanExit:
    SpeedUp False, xlCalc
    Exit Sub
Handler:
    'handle error
    Resume CleanExit
End Sub
```

Vermeiden Sie die Verwendung von ActiveCell oder ActiveSheet in Excel

Die Verwendung von `ActiveCell` oder `ActiveSheet` kann Fehler verursachen, wenn (aus irgendeinem Grund) der Code an der falschen Stelle ausgeführt wird.

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

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

- Die Verwendung von `Active*` kann zu Problemen in lang laufenden Makros führen, wenn sich Ihr Benutzer langweilt und auf ein anderes Arbeitsblatt klickt oder eine andere

Arbeitsmappe öffnet.
- Es kann Probleme verursachen, wenn Ihr Code eine andere Arbeitsmappe öffnet oder erstellt.
- Wenn Ihr Code `Sheets("MyOtherSheet").Select` verwendet, kann dies zu Problemen führen `Sheets("MyOtherSheet").Select` Sie aus, und Sie haben vergessen, auf welchem Blatt Sie sich befanden, bevor Sie mit dem Lesen beginnen oder darauf schreiben.

Nimm niemals das Arbeitsblatt an

Selbst wenn Ihre gesamte Arbeit auf ein einziges Arbeitsblatt gerichtet ist, empfiehlt es sich, das Arbeitsblatt explizit in Ihrem Code anzugeben. Diese Gewohnheit macht es viel einfacher, Ihren Code später zu erweitern oder Teile (oder alle) eines `Sub` oder einer `Function` anzuheben, um sie an einem anderen Ort wiederzuverwenden. Viele Entwickler machen es sich zur Gewohnheit, denselben lokalen Variablennamen für ein Arbeitsblatt in ihrem Code (erneut) zu verwenden, wodurch die Wiederverwendung dieses Codes noch einfacher wird.

Der folgende Code ist zum Beispiel mehrdeutig - funktioniert aber! - solange der Entwickler kein anderes Arbeitsblatt aktiviert oder geändert hat:

```
Option Explicit
Sub ShowTheTime()
    '--- displays the current time and date in cell A1 on the worksheet
    Cells(1, 1).Value = Now()  ' don't refer to Cells without a sheet reference!
End Sub
```

Wenn `Sheet1` aktiv ist, wird Zelle A1 in Sheet1 mit dem aktuellen Datum und der aktuellen Uhrzeit gefüllt. Wenn der Benutzer jedoch Arbeitsblätter aus irgendeinem Grund ändert, wird der Code aktualisiert, unabhängig davon, welches Arbeitsblatt gerade aktiv ist. Das Zielarbeitsblatt ist mehrdeutig.

Es empfiehlt sich, immer zu ermitteln, auf welches Arbeitsblatt sich Ihr Code bezieht:

```
Option Explicit
Sub ShowTheTime()
    '--- displays the current time and date in cell A1 on the worksheet
    Dim myWB As Workbook
    Set myWB = ThisWorkbook
    Dim timestampSH As Worksheet
    Set timestampSH = myWB.Sheets("Sheet1")
    timestampSH.Cells(1, 1).Value = Now()
End Sub
```

Der obige Code ist eindeutig bei der Identifizierung der Arbeitsmappe und des Arbeitsblatts. Obwohl es wie ein Overkill klingt, werden Sie sich vor zukünftigen Problemen bewahren, wenn Sie sich eine gute Gewohnheit in Bezug auf die Zielreferenzen schaffen.

Vermeiden Sie SELECT oder ACTIVATE

Es ist **sehr** selten, dass Sie `Select` oder `Activate` in Ihrem Code verwenden möchten. Einige Excel-Methoden erfordern jedoch die Aktivierung eines Arbeitsblatts oder einer Arbeitsmappe, bevor sie

wie erwartet funktionieren.

Wenn Sie gerade erst anfangen, VBA zu lernen, wird Ihnen oft empfohlen, Ihre Aktionen mit dem Makro-Recorder aufzuzeichnen, und schauen Sie sich dann den Code an. Ich habe beispielsweise Aktionen aufgezeichnet, die zur Eingabe eines Werts in Zelle D3 in Sheet2 ausgeführt wurden, und der Makrocode sieht folgendermaßen aus:

```
Option Explicit
Sub Macro1()
'
' Macro1 Macro
'

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

Denken Sie jedoch daran, dass der Makrorecorder für jede Ihrer (Benutzer-) Aktionen eine Codezeile erstellt. Dazu müssen Sie auf die Registerkarte des Arbeitsblatts klicken, um Sheet2 (`Sheets("Sheet2").Select`) auszuwählen `Sheets("Sheet2").Select` Klicken Sie auf Zelle D3, bevor Sie den Wert (`Range("D3").Select Sheets("Sheet2").Select Range("D3").Select`) und die Eingabetaste (was effektiv ist). Auswahl "der Zelle unter der aktuell ausgewählten Zelle: `Range("D4").Select`).

Es gibt mehrere Probleme bei der Verwendung von `.Select` hier:

- **Das Arbeitsblatt ist nicht immer angegeben.** Dies geschieht, wenn Sie während der Aufnahme nicht zwischen Arbeitsblättern wechseln. Dies bedeutet, dass der Code für verschiedene aktive Arbeitsblätter zu unterschiedlichen Ergebnissen führt.
- **`.Select()` ist langsam.** Auch wenn `Application.ScreenUpdating` auf `False` , handelt es sich um einen nicht mehr zu verarbeitenden Vorgang.
- **`.Select()` ist nicht `.Select()` .** Wenn `Application.ScreenUpdating` auf `True` belassen wird, werden in Excel tatsächlich die Zellen, das Arbeitsblatt, das Formular usw. ausgewählt. Das ist anstrengend für die Augen und wirklich unangenehm anzusehen.
- **`.Select()` löst Listener aus.** Dies ist bereits ein wenig fortgeschritten, aber wenn dies nicht `Worksheet_SelectionChange()` wird, werden Funktionen wie `Worksheet_SelectionChange()` ausgelöst.

Wenn Sie in VBA codieren, sind alle "Typisierungsaktionen" (dh `Select` Anweisungen) nicht mehr erforderlich. Ihr Code kann auf eine einzige Anweisung reduziert werden, um den Wert in die Zelle einzufügen:

```
'--- GOOD
ActiveWorkbook.Sheets("Sheet2").Range("D3").Value = 3.1415

'--- BETTER
Dim myWB      As Workbook
Dim myWS      As Worksheet
Dim myCell    As Range
```

```
Set myWB = ThisWorkbook             '*** see NOTE2
Set myWS = myWB.Sheets("Sheet2")
Set myCell = myWS.Range("D3")

myCell.Value = 3.1415
```

(Das BETTER-Beispiel oben zeigt die Verwendung von Zwischenvariablen zum Trennen verschiedener Teile der Zellreferenz. Das GOOD-Beispiel funktioniert immer einwandfrei, kann jedoch bei viel längeren Codemodulen sehr umständlich sein und ist schwieriger zu debuggen, wenn eine der Referenzen falsch eingegeben wird.)

** HINWEIS: Der Makrorekorder nimmt viele Annahmen über den Datentyp an, den Sie eingeben. In diesem Fall geben Sie einen Zeichenfolgenwert als Formel ein, um den Wert zu erstellen. Ihr Code muss dies nicht tun und kann der Zelle einfach einen numerischen Wert wie oben gezeigt direkt zuweisen.

** HINWEIS2: Die empfohlene Vorgehensweise besteht darin, Ihre lokale Arbeitsmappenvariable auf `ThisWorkbook` statt auf `ActiveWorkbook` (sofern Sie dies nicht ausdrücklich benötigen). Der Grund dafür ist, dass Ihr Makro im Allgemeinen Ressourcen in der Arbeitsmappe benötigt bzw. verwendet, die der VBA-Code erzeugt, und NICHT außerhalb der Arbeitsmappe aussieht - auch wenn Sie Ihren Code nicht ausdrücklich dazu auffordern, mit einer anderen Arbeitsmappe zu arbeiten. Wenn Sie mehrere Arbeitsmappen in Excel geöffnet haben, hat `ActiveWorkbook` den Fokus, *der sich von der Arbeitsmappe unterscheiden kann, die in Ihrem VBA-Editor angezeigt wird* . Sie glauben also, Sie führen eine Arbeitsmappe aus, wenn Sie wirklich auf eine andere Arbeitsmappe verweisen. `ThisWorkbook` Arbeitsbuch bezieht sich auf die Arbeitsmappe, die den ausgeführten Code enthält.

Definieren und setzen Sie immer Verweise auf alle Arbeitsmappen und Arbeitsblätter

Wenn Sie mit mehreren offenen Arbeitsmappen arbeiten, von denen jede mehrere Arbeitsblätter haben kann, ist es am sichersten, alle Arbeitsmappen und Arbeitsblätter zu definieren und einen Verweis darauf zu setzen.

`ActiveWorkbook` **Sie sich nicht** auf `ActiveWorkbook` oder `ActiveSheet` da diese möglicherweise vom Benutzer geändert werden.

Das folgende Codebeispiel zeigt , wie eine Reihe von *„RAW_DATA“* Blatt in dem *„Data.xlsx“* Arbeitsmappe *„Refined_Data“* Blatt in der *„Results.xlsx“* Arbeitsmappe kopieren.

Das Verfahren veranschaulicht auch, wie Sie ohne die `Select` Methode kopieren und einfügen.

```
Option Explicit

Sub CopyRanges_BetweenShts()


    Dim wbSrc                           As Workbook
    Dim wbDest                          As Workbook
    Dim shtCopy                         As Worksheet
```

```
    Dim shtPaste                        As Worksheet

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

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

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

End Sub
```

Das WorksheetFunction-Objekt wird schneller als ein UDF-Äquivalent ausgeführt

VBA wird zur Laufzeit kompiliert, was die Leistung erheblich beeinträchtigt. Alles integrierte wird schneller. Versuchen Sie, sie zu verwenden.

Als Beispiel vergleiche ich die SUM- und COUNTIF-Funktionen, aber Sie können sie verwenden, wenn Sie etwas mit WorkSheetFunctions lösen können.

Ein erster Versuch für diese wäre, den Bereich zu durchlaufen und ihn Zelle für Zelle (mit einem Bereich) zu verarbeiten:

```
Sub UseRange()
    Dim rng as Range
    Dim Total As Double
    Dim CountLessThan01 As Long

    Total = 0
    CountLessThan01 = 0
    For Each rng in Sheets(1).Range("A1:A100")
        Total = Total + rng.Value2
        If rng.Value < 0.1 Then
            CountLessThan01 = CountLessThan01 + 1
        End If
    Next rng
    Debug.Print Total & ", " & CountLessThan01
End Sub
```

Eine Verbesserung kann darin bestehen, die Bereichswerte in einem Array zu speichern und Folgendes zu verarbeiten:

```
Sub UseArray()
    Dim DataToSummarize As Variant
    Dim i As Long
    Dim Total As Double
    Dim CountLessThan01 As Long

    DataToSummarize = Sheets(1).Range("A1:A100").Value2 'faster than .Value
    Total = 0
```

```
    CountLessThan01 = 0
    For i = 1 To 100
        Total = Total + DataToSummarize(i, 1)
        If DataToSummarize(i, 1) < 0.1 Then
            CountLessThan01 = CountLessThan01 + 1
        End If
    Next i
    Debug.Print Total & ", " & CountLessThan01
End Sub
```

Anstatt jedoch eine Schleife zu schreiben, können Sie die Anwendung `Application.Worksheetfunction` der Sie einfache Formeln ausführen können:

```
Sub UseWorksheetFunction()
    Dim Total As Double
    Dim CountLessThan01 As Long

    With Application.WorksheetFunction
        Total = .Sum(Sheets(1).Range("A1:A100"))
        CountLessThan01 = .CountIf(Sheets(1).Range("A1:A100"), "<0.1")
    End With

    Debug.Print Total & ", " & CountLessThan01
End Sub
```

Bei komplexeren Berechnungen können Sie sogar `Application.Evaluate` :

```
Sub UseEvaluate()
    Dim Total As Double
    Dim CountLessThan01 As Long

    With Application
        Total = .Evaluate("SUM(" & Sheet1.Range("A1:A100").Address( _
            external:=True) & ")")
        CountLessThan01 = .Evaluate("COUNTIF('Sheet1'!A1:A100,""<0.1"")")
    End With

    Debug.Print Total & ", " & CountLessThan01
End Sub
```

Und schließlich läuft man über 25.000 Mal über Subs. Hier ist die durchschnittliche Zeit (5 Tests) in Millisekunden (natürlich ist dies auf jedem PC anders, aber im Vergleich zueinander verhalten sie sich ähnlich)

1. UseWorksheetFunction: 2156 ms
2. UseArray: 2219 ms (+ 3%)
3. UseEvaluate: 4693 ms (+ 118%)
4. Nutzungsbereich: 6530 ms (+ 203%)

Vermeiden Sie, die Namen von Eigenschaften oder Methoden als Variablen zu verwenden

Es wird im Allgemeinen nicht als "bewährte Methode" betrachtet, die reservierten Namen von Eigenschaften oder Methoden als Namen Ihrer eigenen Prozeduren und Variablen zu verwenden.

Bad Form - Während die folgende ist (streng genommen) legal, Code arbeiten , um die Umwidmung der Suche Methode sowie die Zeile , Spalte und Adress Eigenschaften können Probleme / Konflikte mit Namen Mehrdeutigkeit verursacht und ist einfach nur verwirrend im Allgemeinen.

```
Option Explicit

Sub find()
    Dim row As Long, column As Long
    Dim find As String, address As Range

    find = "something"

    With ThisWorkbook.Worksheets("Sheet1").Cells
        Set address = .SpecialCells(xlCellTypeLastCell)
        row = .find(what:=find, after:=address).row        '< note .row not capitalized
        column = .find(what:=find, after:=address).column  '< note .column not capitalized

        Debug.Print "The first 'something' is in " & .Cells(row, column).address(0, 0)
    End With
End Sub
```

Gute Form : Da alle reservierten Wörter in nahe, aber eindeutige Näherungen der Originale umbenannt wurden, wurden mögliche Namenskonflikte vermieden.

```
Option Explicit

Sub myFind()
    Dim rw As Long, col As Long
    Dim wht As String, lastCell As Range

    wht = "something"

    With ThisWorkbook.Worksheets("Sheet1").Cells
        Set lastCell = .SpecialCells(xlCellTypeLastCell)
        rw = .Find(What:=wht, After:=lastCell).Row         '◂ note .Find and .Row
        col = .Find(What:=wht, After:=lastCell).Column     '◂ .Find and .Column

        Debug.Print "The first 'something' is in " & .Cells(rw, col).Address(0, 0)
    End With
End Sub
```

Es kann vorkommen, dass Sie eine Standardmethode oder -eigenschaft absichtlich nach Ihren eigenen Vorstellungen neu schreiben möchten, doch es gibt nur wenige Situationen. Bleiben Sie größtenteils fern von reservierten Namen für Ihre eigenen Konstrukte.

VBA-Best Practices online lesen: https://riptutorial.com/de/excel-vba/topic/1107/vba-best-practices

Kapitel 28: VBA-Sicherheit

Examples

Passwort Schützen Sie Ihre VBA

In Ihrem VBA befinden sich manchmal vertrauliche Informationen (z. B. Kennwörter), auf die Benutzer keinen Zugriff haben sollen. Sie können eine grundlegende Sicherheit für diese Informationen erreichen, indem Sie Ihr VBA-Projekt mit einem Kennwort schützen.

Folge diesen Schritten:

1. Öffnen Sie Ihren Visual Basic-Editor (Alt + F11)
2. Navigieren Sie zu Extras -> VBAProject-Eigenschaften ...
3. Navigieren Sie zur Registerkarte Schutz
4. Aktivieren Sie das Kontrollkästchen "Projekt für Ansicht sperren"
5. Geben Sie Ihr gewünschtes Passwort in die Textfelder Passwort und Passwort bestätigen ein

Wenn nun jemand in einer Office-Anwendung auf Ihren Code zugreifen möchte, muss er zunächst das Kennwort eingeben. Seien Sie sich jedoch bewusst, dass selbst ein starkes VBA-Projektkennwort zu trivial ist.

VBA-Sicherheit online lesen: https://riptutorial.com/de/excel-vba/topic/7642/vba-sicherheit

Kapitel 29: Verwenden Sie ein Arbeitsblattobjekt und kein Blattobjekt

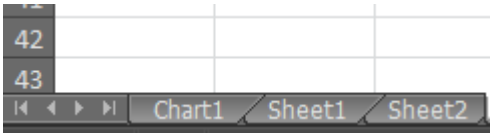
Einführung

Viele VBA-Benutzer betrachten Objekte für Arbeitsblätter und Arbeitsblätter als Synonyme. Sie sind nicht.

Das Objekt "Blätter" besteht aus Arbeitsblättern und Diagrammen. Wenn wir also Diagramme in unserer Excel-Arbeitsmappe haben, sollten wir vorsichtig sein und keine `Sheets` und `Worksheets` als Synonyme verwenden.

Examples

Drucken Sie den Namen des ersten Objekts




```
Option Explicit

Sub CheckWorksheetsDiagram()

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

End Sub
```

Das Ergebnis:

```
Sheet1
Chart1
Chart1
```

Verwenden Sie ein Arbeitsblattobjekt und kein Blattobjekt online lesen: https://riptutorial.com/de/excel-vba/topic/9996/verwenden-sie-ein-arbeitsblattobjekt-und-kein-blattobjekt

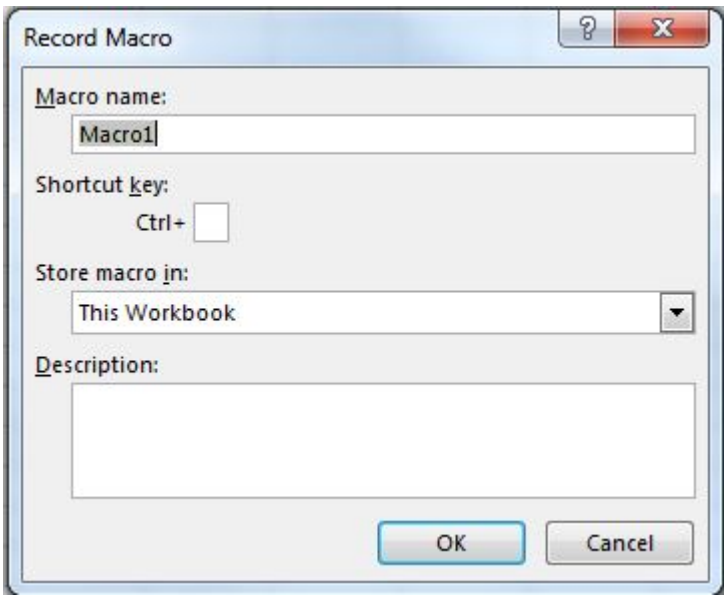
Kapitel 30: Wie man ein Makro aufnimmt

Examples

Wie man ein Makro aufnimmt

Die einfachste Möglichkeit zum Aufzeichnen eines Makros besteht darin, dass die Schaltfläche in der linken unteren Ecke von Excel folgendermaßen aussieht: 

Wenn Sie darauf klicken, werden Sie in einem Popup gefragt, ob Sie das Makro benennen und entscheiden möchten, ob Sie eine Tastenkombination benötigen. Fragt außerdem, wo das Makro gespeichert werden soll und eine Beschreibung. Sie können einen beliebigen Namen auswählen, Leerzeichen sind nicht erlaubt.



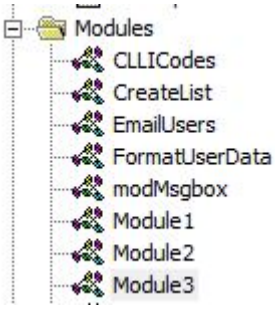
Wenn Sie Ihrem Makro eine Tastenkombination zuweisen möchten, wählen Sie einen Buchstaben, den Sie sich merken werden, damit Sie das Makro schnell und einfach wieder verwenden können.

Sie können das Makro in "Diese Arbeitsmappe", "Neue Arbeitsmappe" oder "Persönliche Makro-Arbeitsmappe" speichern. Wenn das Makro, das Sie gerade aufnehmen möchten, nur in der aktuellen Arbeitsmappe verfügbar sein soll, wählen Sie "Diese Arbeitsmappe". Wenn Sie es in einer neuen Arbeitsmappe speichern möchten, wählen Sie "Neue Arbeitsmappe". Wenn das Makro für jede geöffnete Arbeitsmappe verfügbar sein soll, wählen Sie "Persönliche Makro-Arbeitsmappe".

Nachdem Sie dieses Popup ausgefüllt haben, klicken Sie auf "Ok".

Führen Sie dann alle Aktionen aus, die Sie mit dem Makro wiederholen möchten. Wenn Sie fertig sind, klicken Sie auf die gleiche Schaltfläche, um die Aufnahme zu beenden. Es sieht jetzt so aus:

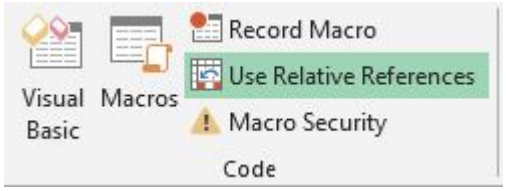
Jetzt können Sie zur Registerkarte "Entwickler" gehen und Visual Basic öffnen. (oder verwenden Sie Alt + F11)

Sie haben jetzt ein neues Modul im Ordner Modules. 

Das neueste Modul enthält das gerade aufgezeichnete Makro. Doppelklicken Sie darauf, um es aufzurufen.

Ich habe ein einfaches Kopieren und Einfügen vorgenommen:

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

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

Wenn Sie nicht möchten, dass es immer in "A12" eingefügt wird, können Sie relative Verweise verwenden, indem Sie auf der Registerkarte "Entwickler" das Kontrollkästchen "Relative Verweise verwenden" aktivieren. 

Wenn Sie die gleichen Schritte wie zuvor ausführen, wird das Makro jetzt folgendermaßen:

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

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

Der Wert wird immer noch von "A1" in eine Zelle 11 Zeilen nach unten kopiert. Jetzt können Sie dasselbe Makro mit einer Startzelle ausführen, und der Wert aus dieser Zelle wird in die Zelle 11 Zeilen nach unten kopiert.

Wie man ein Makro aufnimmt online lesen: https://riptutorial.com/de/excel-vba/topic/8204/wie-man-ein-makro-aufnimmt

Kapitel 31: Zusammengeführte Zellen / Bereiche

Examples

Überlegen Sie, bevor Sie zusammengefügte Zellen / Bereiche verwenden

In erster Linie sind zusammengefügte Zellen nur dazu da, das Aussehen Ihrer Blätter zu verbessern.

Es ist also buchstäblich das letzte, was Sie tun sollten, sobald Ihr Arbeitsblatt und Ihre Arbeitsmappe voll funktionsfähig sind!

Wo befinden sich die Daten in einem zusammengeführten Bereich?

Wenn Sie einen Bereich zusammenführen, wird nur ein Block angezeigt.

Die Daten befinden sich in der allerersten **Zelle dieses Bereichs** und die **anderen sind leere Zellen** !

Ein guter Punkt dabei: Sie müssen nicht alle Zellen oder den Bereich nach dem Zusammenfügen füllen, sondern nur die erste Zelle! ;)

Die anderen Aspekte dieses zusammengeführten Bereichs sind global negativ:

- Wenn Sie eine Methode zum Suchen der letzten Zeile oder Spalte verwenden , riskieren Sie einige Fehler
- Wenn Sie Zeilen durchlaufen und zur besseren Lesbarkeit einige Bereiche zusammengeführt haben, werden leere Zellen angezeigt und nicht der Wert, der vom zusammengeführten Bereich angezeigt wird

Zusammengeführte Zellen / Bereiche online lesen: https://riptutorial.com/de/excel-vba/topic/7308/zusammengefuhrte-zellen---bereiche

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit Excel-VBA	Branislav Kollár, chris neilsen, Cody G., Comintern, Community, Doug Coats, EEM, Gordon Bell, Jeeped, Joel Spolsky, Kaz, Laurel, LucyMarieJ, Macro Man, Malick, Maxime Porté, Regis, RGA, Ron McMahon, SandPiper, Shai Rado, Taylor Ostberg, whytheq
2	Anwendungsobjekt	Captain Grumpy, Joel Spolsky
3	Arbeiten mit Excel-Tabellen in VBA	Excel Developers
4	Arbeitsmappen	PeterT
5	Arrays	Alon Adler, Hubisan, Miguel_Ryu, Shahin
6	automatischer Filter ; Verwendungen und Best Practices	Sgdva
7	Bedingte Anweisungen	SteveES
8	Bedingte Formatierung mit VBA	Zsmaster
9	Benannte Bereiche	Andre Terra, Portland Runner
10	Benutzerdefinierte Funktionen (UDFs)	Jeeped, Malick, Slai, user3561813, Vegard
11	Bereiche und Zellen	Adam, Branislav Kollár, Doug Coats, Gregor y, Jbjstam, Joel Spolsky, Julian Kuchlbauer, Máté Juhász, Miguel_Ryu, Patrick Wynne, Vegard
12	Bindung	Captain Grumpy, EEM, Jeeped, jlookup, Malick, Raystafarian
13	CustomDocumentProperties in der Praxis	T.M.
14	Dateisystemobjekt	Zsmaster
15	Debugging und Fehlerbehebung	Cody G., Etheur, Gregor y, Julian Kuchlbauer, Kyle, Malick, Michael Russo, RGA, Ron McMahon, Slai,

		Steven Schroeder, Taylor Ostberg
16	Diagramme und Diagramme	Byron Wall
17	Doppelte Werte in einem Bereich suchen	quadrature, T.M.
18	Durchlaufen Sie alle Arbeitsblätter in der aktiven Arbeitsmappe	Doug Coats, Shai Rado
19	Erstellen eines Dropdown-Menüs im aktiven Arbeitsblatt mit einem Kombinationsfeld	Macro Man, quadrature, R3uK
20	Excel-VBA-Optimierung	Masoud, paul bica, T.M.
21	Häufige Fehler	Egan Wolf, Gordon Bell, Macro Man, Malick, Peh, SWa, Taylor Ostberg
22	Methoden zum Suchen der zuletzt verwendeten Zeile oder Spalte in einem Arbeitsblatt	curious, Hubisan, Máté Juhász, Michael Russo, Miqi180, paul bica, R3uK, Raystafarian, RGA, Shai Rado, Slai, Thomas Inzina, YowE3K
23	Pivot-Tabellen	PeterT
24	PowerPoint-Integration über VBA	mnoronha, RGA
25	SQL in Excel VBA - Best Practices	Zsmaster
26	Tipps und Tricks zu Excel VBA	Andre Terra, Cody G., Jeeped, Kumar Sourav, Macro Man, RGA
27	VBA-Best Practices	Alexis Olson, Branislav Kollár, Chel, Cody G., Comintern, EEM, FreeMan, genespos, Hubisan, Huzaifa Essajee, Jeeped, JKAbrams, Kumar Sourav, Kyle, Macro Man, Malick, Máté Juhász, Munkeeface, paul bica, Peh, PeterT, Portland Runner, RGA, Shai Rado, Stefan Pinnow, Steven Schroeder, Taylor Ostberg, ThunderFrame, Verzweifler, Vityata
28	VBA-Sicherheit	Chel, TheGuyThatDoesn'tKnowMuch
29	Verwenden Sie ein Arbeitsblattobjekt und kein Blattobjekt	Vityata

30	Wie man ein Makro aufnimmt	Mike, Robby
31	Zusammengeführte Zellen / Bereiche	R3uK