AUSGABE 04-05/2022

VISUAL BASIC ENTWICKLER

MAGAZIN FÜR DIE PROGRAMMIERUNG VON MICROSOFT OFFICE UND ANDEREN ANWENDUNGEN MIT VB.NET, VBA UND TWINBASIC

IN DIESEM HEFT:

André Minhorst Verlag

Ran an die Tabellenkalkulation!

In dieser Doppelausgabe steigen wir in die VBA-Programmierung von Excel ein. Nachdem wir in der letzten Ausgabe den Start mit Outlook hinter uns gebracht haben, geht es nun an die Tabellenkalkulation von Microsoft Office. Dabei schauen wir uns erst einmal grundlegende Techniken an wie beispielsweise das Aufzeichnen von Makros. Außerdem liest Du in diesem Heft alles, was Du für den Zugriff auf Workbooks, Worksheets, Bereiche und Zellen unter Excel benötigst. Und wir sehen direkt ein tolles Beispiel, nämlich das Übersetzen von in Excel-Tabellen enthaltenen Texten in eine andere Sprache!

Genau wie die erste Ausgabe ist auch die nächste Ausgabe wieder eine Doppelausgabe. Auf diese Weise können wir uns ausgiebig der zweiten Office-Anwendung zuwenden, um die wir uns in diesem Magazin kümmern wollen. Nachdem der Schwerpunkt in der ersten Doppelausgabe auf der Programmierung von Outlook lag, geht es in dieser Ausgabe primär um die Programmierung von Microsoft Excel.

Den Einstieg machen wir allerdings mit einem Artikel, indem wir uns das Thema Makroaufzeichnung ansehen (**VBA-Basics: Makros aufzeichnen**, ab Seite 4). Diese Technik können wir unter Excel und Word einsetzen, um einfache Abläufe zu automatisieren und zu reproduzieren.

Die VBA-Grundlagen bauen wir in den Artikeln **VBA Basics: Module, Klassen und Co.** (ab Seite 14), **VBA Basics: Makros, Prozeduren, Funktionen und Co.** (ab Seite 18) und **VBA Basics: Bedingungen** (ab Seite 27) aus. Außerdem lernen wir die VBA-Möglichkeiten der Interaktion mit dem Benutzer kennen (**VBA Basics: MsgBox- und InputBox-Funktion**, ab Seite 34).

Einiges von diesem Know-how nutzen wir im Artikel **Umfangreiche Texte in Code integrieren** (ab Seite 40), um längere Zeichenketten, die man per VBA weiterverarbeiten möchte, in den VBA-Code zu integrieren.

Wo wir schon mit der Excel-Programmierung beginnen, kümmern wir uns auch gleich darum, wir wir Ribbon-Anpassungen in den Office-Anwendungen Excel, Word und PowerPoint vornehmen können (**Ribbons in Office-Dokumenten**, ab Seite 45).

Die Outlook-Kenntnisse aus der letzten Ausgabe vertiefen wir in dieser Ausgabe mit den Artikeln **Outlook: Application_Startup feuert nicht** (ab Seite 51), **Outlook: E-Mails nach Eingang verarbeiten** (ab Seite 54) und **Outlook: E-Mail per Drag and Drop nach Access** (ab Seite 62).

Und schließlich folgt der große Excel-Block. Hier schauen wir uns zuerst ein Nicht-VBA-Thema an. Der Artikel **Einfache Excel-Formulare erstellen** zeigt ab Seite 73, wie man mit wenigen Handgriffen ein rudimentäres Formular zur Dateneingabe aufruft. Wenn Du Prozeduren per Schaltfläche direkt vom Tabellenblatt aus aufrufen möchtest, findest Du das notwendige Rüstzeug ab Seite 78 im Artikel **Buttons in Excel**. Und die Einführung in die Programmierung der verschiedenen Excel-Objekte liefern die beiden Artikel **Excel: Workbooks und Worksheets per VBA** (ab Seite 84) und **Excel: Zellen und Bereiche per VBA** (ab Seite 97).

Und damit wir auch gleich eine praktische Umsetzung haben, schauen wir uns im Artikel **Texte übersetzen mit DeepL** ab Seite 115 zuerst an, wir wie den Übersetzungs-Webservice **DeepL** steuern können. Diese Techniken setzen wir dann in einem Excel-Worksheet ein, um die in den Zellen einer Spalte enthaltenen Texte daneben in einer anderen Sprache auszugeben (**Excel: Übersetzungen mit DeepL**, ab Seite 123).

Nun viel Spaß beim Lesen!

Ihr André Minhorst

VBA-Basics: Makros aufzeichnen

Das Programmieren von Office-Automatisierungen beispielsweise in der Sprache VBA kann für Einsteiger herausfordernd sein. Allerdings gibt es Möglichkeiten, sich hier und da zu behelfen: Die Anwendungen Excel, Word und PowerPoint bieten nämlich einen sogenannten Makro-Rekorder, mit dem man einfache Abläufe innerhalb der Anwendung aufzeichnen kann. Das Ergebnis ist eine VBA-Prozedur, die Du anschließend erneut aufrufen kannst. In vielen Fällen reicht dies bereits aus, um die gewünschten Schritte zu automatisieren, in anderen Fällen möchtest Du das Ergebnis des Makro-Recordsets vielleicht noch anpassen. Wie Du diesen Makro-Rekorder einsetzt und wie Du die Ergebnisse anpassen kannst, zeigt der vorliegende Artikel.

Werkzeuge zum Aufzeichnen von Makros einblenden

Bevor wir das erste Makro aufzeichnen können, ist zunächst eine kleine Anpassung erforderlich. Die Werkzeuge zum Aufzeichnen von Makros sind nämlich standardmäßig gar nicht sichtbar. Also schauen wir uns anhand einer neuen, leeren Excel-Arbeitsmappe an, wie wir diese einblenden.

Diese Einstellung nehmen wir im Optionen-Dialog der jeweiligen Anwendung vor. Unter Excel klicken wir dazu auf den Registerreiter **Datei** und dann auf

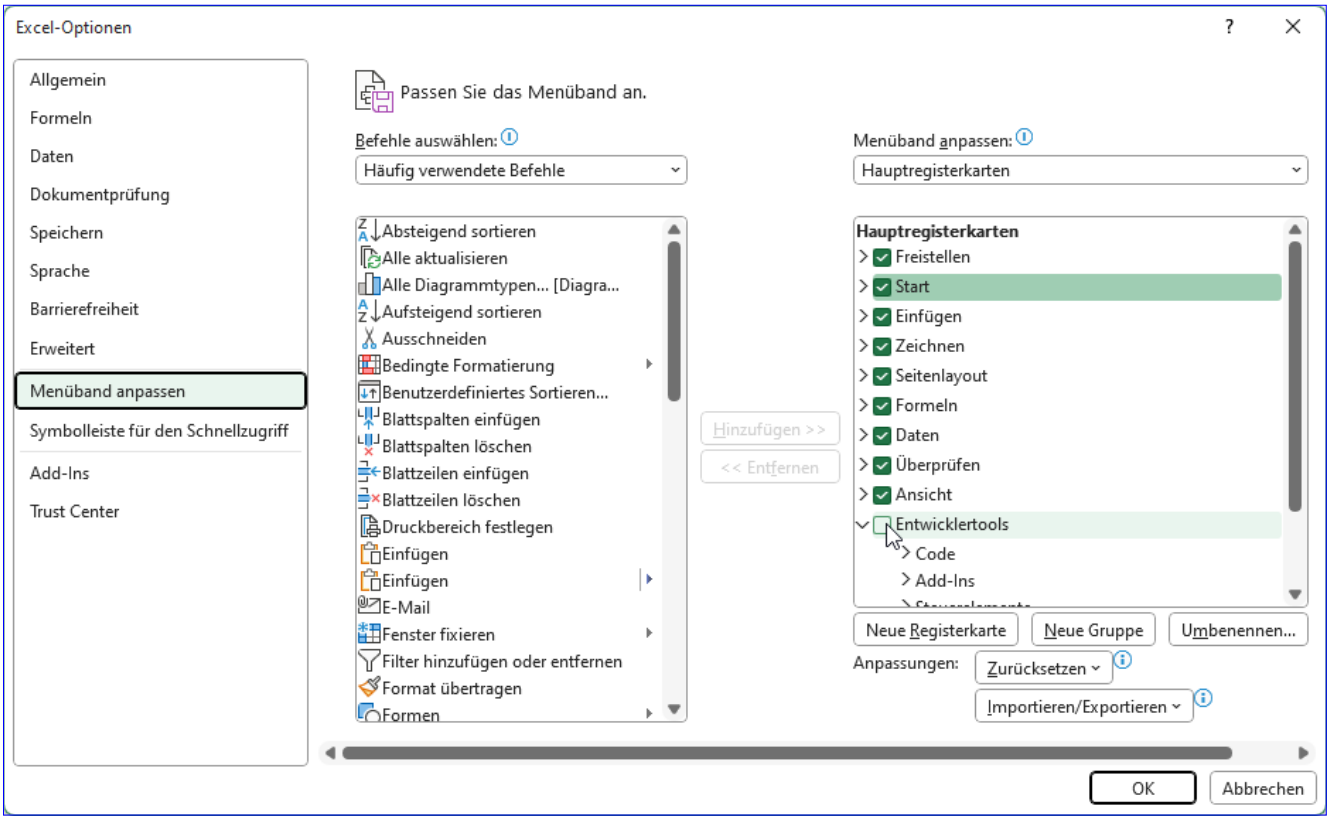


Bild 1: Einblenden der Entwicklertools und damit der Werkzeuge zum Aufzeichnen von Makros

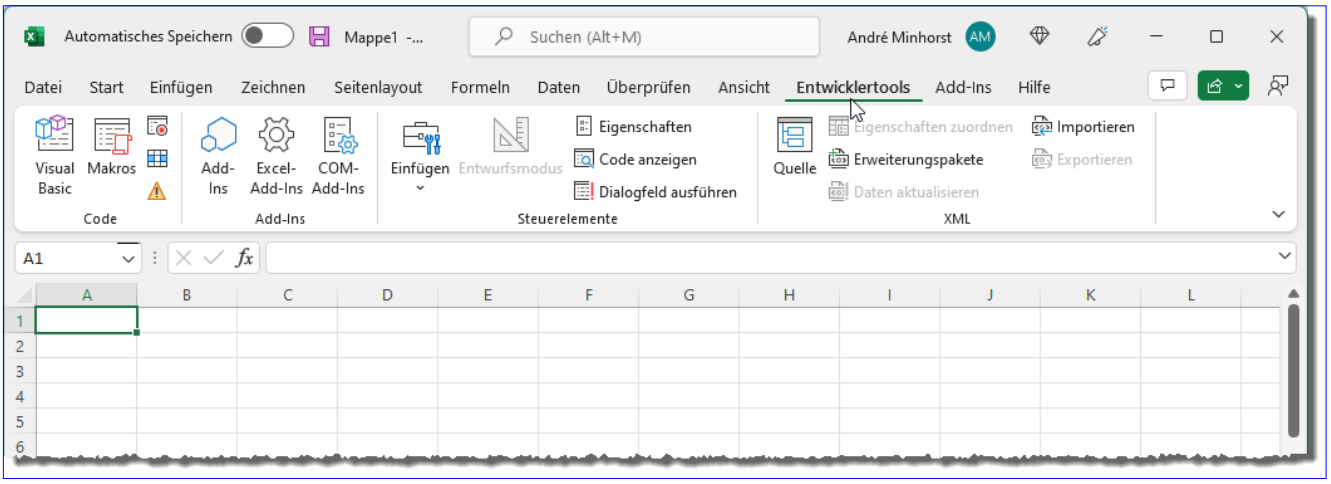


Bild 2: Die Entwickertools im Ribbon

Optionen. Im nun erscheinenden Dialog **Excel-Optionen** wechseln wir zum Bereich **Menüband anpassen**.

Hier finden wir in der rechten Liste mit der Überschrift **Hauptregisterkarten** den Eintrag **Entwicklertools**, der standardmäßig nicht aktiviert ist.

Also setzen wir einen Haken vor diese Eigenschaft und schließen den Dialog wieder (siehe Bild 1).

Gleich nach dem Schließen finden wir im Ribbon von Excel einen neuen Tab-Reiter namens **Entwicklertools** vor, der nach dem Anklicken wie in Bild 2 aussieht.

Interessant für uns sind vor allem die Befehle in der ersten Gruppe **Code** und hier speziell der Befehl **Makro aufzeichnen**, der beim Verbreitern des Excel-Fensters vollständig sichtbar wird (siehe Bild 3).

In den übrigen Office-Anwendungen wie Word oder PowerPoint musst Du dieses Ribbon-Tab übrigens separat einblenden. PowerPoint, Outlook und Access bieten gar keine Möglichkeit zur Aufzeichnung von Makros an.

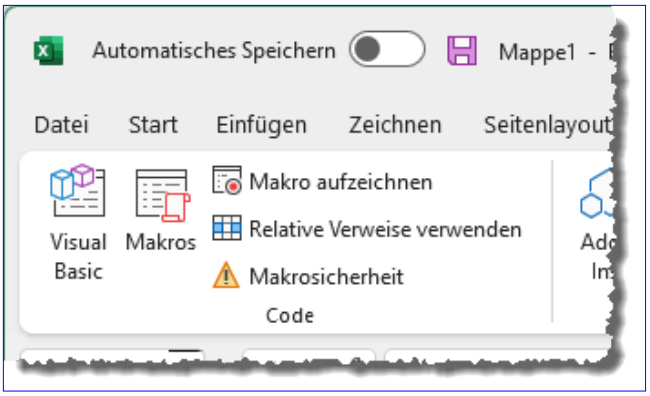


Bild 3: Der Befehl **Makro aufzeichnen** in Excel

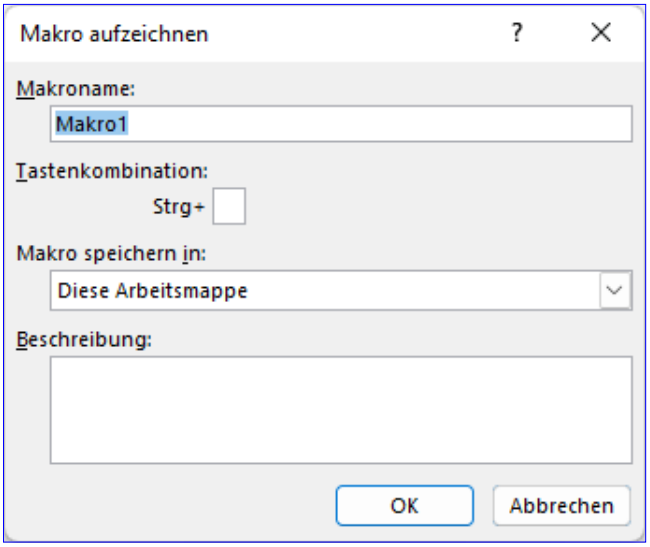


Bild 4: Der Dialog **Makro aufzeichnen** in Excel

Der Dialog »Makro aufzeichnen« von Excel

Klickst Du auf diesen Eintrag, erscheint der Dialog Makro aufzeichnen aus Bild 4. Dieser kommt mit dem voreingestellten Makronamen **Makro1**, den Du als erstes Ändern solltest. Als Namen kommen alle Bezeichnungen in Frage, die mit einem Buchstaben beginnen und aus Buchstaben, Zahlen und Unterstrichen bestehen. Kurz gesagt: Hier greifen die Regeln für die Benennung von VBA-Prozeduren, denn als solche wird das Makro aufgezeichnet.

Tastenkombination für das Makro festlegen

Der Dialog erlaubt es, eine Tastenkombination einzustellen, mit der sich das aufzuzeichnende Makro ausführen lässt. Hierbei sind wir allerdings recht eingeschränkt, da wir nur ein Zeichen eingeben können und diese mit der Schaltfläche **Strg** kombiniert wird. Und Du kannst zusätzlich noch die Umschalttaste gedrückt halten. Allerdings gibt es bereits eine Reihe von Funktionen, die durch Kombinationen von **Strg** mit Buchstaben oder Zahlen abgedeckt werden, also solltest Du eine nutzen, die noch nicht verwendet wird. Du kannst diese in Excel zwar in jedem Fall verwenden, aber die eigentliche Funktion der Tastenkombination steht dann nicht mehr zur Verfügung.

Wenn Du also beispielsweise die Tastenkombination **Strg + A** verwendest, dann kannst Du damit zwar von nun an das verknüpfte Makro starten, aber Du kannst nicht mehr den kompletten Inhalt des Tabellenblatts markieren.

Wo soll das Makro angelegt werden?

Die nächste Frage ist, wo das Makro gespeichert werden soll. Unter Excel gibt es drei Möglichkeiten:

- **Diese Arbeitsmappe**: Hier wird das Makro standardmäßig gespeichert. Es landet dann in einem neuen, leeren Standardmodul im VBA-Projekt der aktuellen Arbeitsmappe. Wichtig: Sollen die Makros mit der Arbeitsmappe gespeichert werden, muss diese als **.xlsm**-Datei gespeichert werden.

- **Persönliche Makroarbeitsmappe**: Sofern noch nicht erstellt, legt Excel hier eine versteckte Excel-Datei namens **Personal.xlsb** an und speichert das Makro in dieser. Damit ist das Makro nicht nur in der aktuell geöffneten Excel-Arbeitsmappe einsetzbar, sondern auch in anderen Arbeitsmappen. Achtung: Wenn wir dieses testweise ausprobieren und hier ein Makro anlegen, wird dieses Projekt nun immer im VBA-Editor von Excel angezeigt. Am Ende des Artikels beschreiben wir daher, wie Du die Datei **Personal.xlsb** und das entsprechende VBA-Projekt wieder loswirst.

- **Neue Arbeitsmappe**: Dies speichert das aufzuzeichnende Makro in einer neuen Arbeitsmappe.

Schließlich kannst Du noch eine Beschreibung des Makros im Dialog **Makro aufzeichnen** hinterlegen.

Der Dialog »Makro aufzeichnen« von Word

In Word sieht der Dialog **Makro aufzeichnen** etwas anders aus als in Excel (siehe Bild 5). Hier finden wir

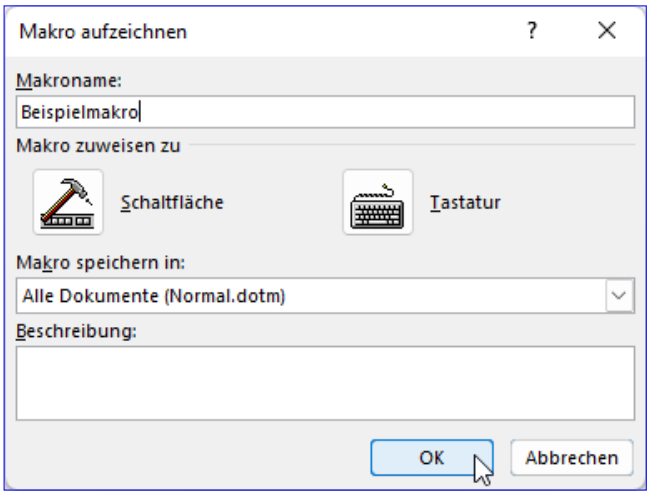


Bild 5: Der Dialog **Makro aufzeichnen** in Word

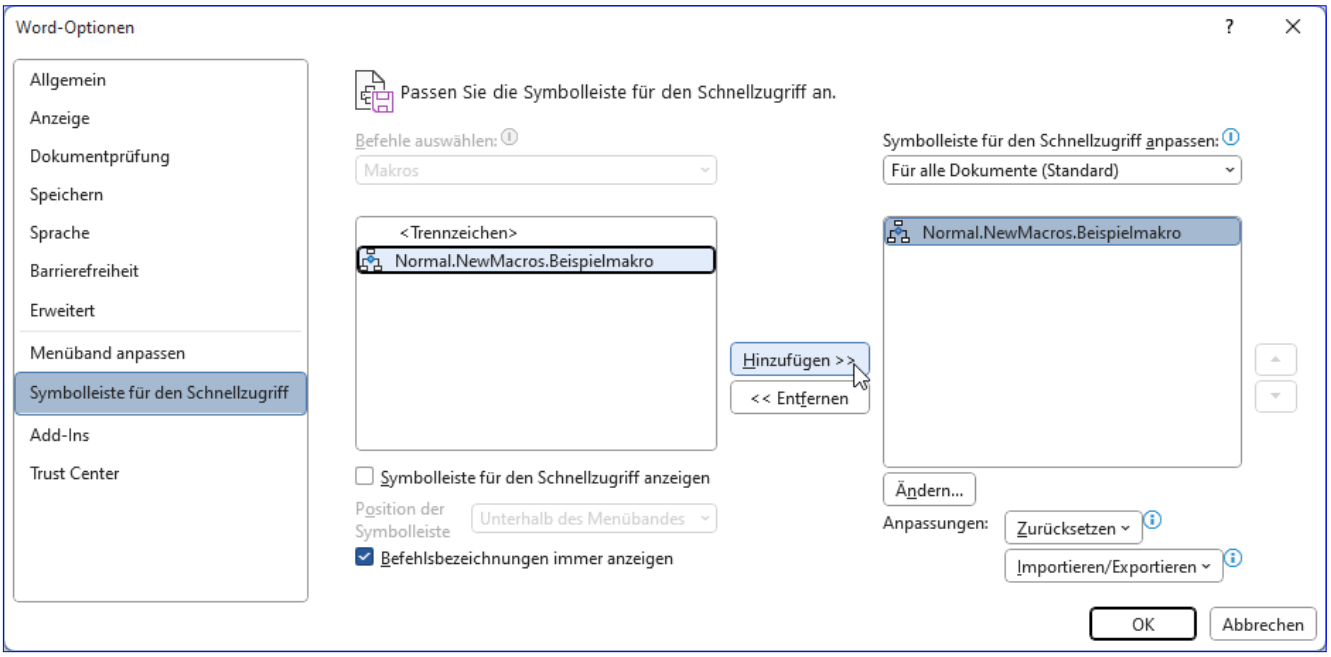


Bild 6: Anpassen der Schnellzugriffsleiste zum Hinzufügen der Makrobefehle in Word

nicht nur die Möglichkeit, eine Tastenkombination zu hinterlegen, sondern auch noch die Option **Schaltfläche**. Klicken wir diese Option an, erscheint der Dialog **Word-Optionen** und zeigt eine abgespeckte Version des Bereichs **Symbolleiste für den Schnellzugriff** an.

Wenn wir hier beispielsweise den Makronamen **Beispielmakro** angegeben haben, dann erscheint in der linken Liste der verfügbaren Befehle ein Eintrag namens **Normal.NewMacros.Beispielmakro**.

Diesen fügen wir durch Markieren und Anklicken der Schaltfläche **Hinzufügen** zur Symbolleiste für den Schnellzugriff hinzu (siehe Bild 6). Anschließend lässt sich das Makro, wenn es erst einmal aufgezeichnet ist, dann über die Schnellzugriffsleiste aufrufen.

Normal.NewMacros.Beispielmakro ist allerdings kein besonders hübscher Text für eine Schaltfläche der Schnellzugriffsleiste.

Um diesen zu ändern, klicken wir auf die Schaltfläche **Ändern...** unter der Liste mit den Befehlen der Schnellzugriffsleiste und öffnen damit den Dialog aus Bild 7, mit dem wir die Beschriftung ändern und auch noch ein Icon hinzufügen können.

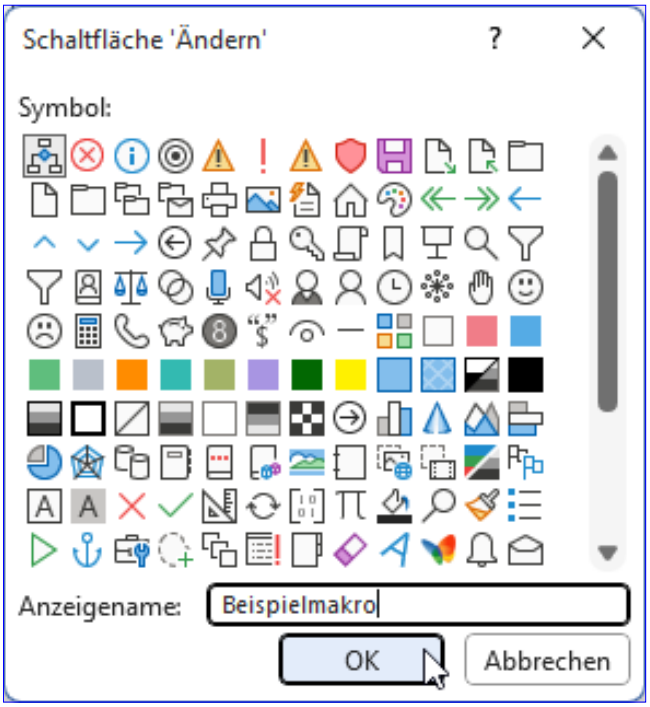


Bild 7: Ändern der Bezeichnung und Hinzufügen eines Icons

Danach lässt sich das Makro in Word über das Ribbon wie in Bild 8 aufrufen.

Makrobefehle in Excel zur Schnellzugriffsleiste hinzufügen

Auch wenn der oben vorgestellte Dialog **Makro** aufzeichnen von Excel nicht direkt die Möglichkeit enthält, die aufgezeichneten Makros über Befehle in der Schnellzugriffsleiste aufzurufen, so ist dies dennoch nachträglich möglich. Dazu öffnen wir auch unter Excel den Optionen-Dialog und wechseln zum Bereich **Symbolleiste für den Schnellzugriff**.

Hier sehen wir allerdings zuerst alle verfügbaren Befehle. Um nur die Makros anzuzeigen, wählen wir über der linken Liste im Auswahlfeld **Befehle auswählen** den Eintrag **Makros** aus. Daraufhin zeigt das Listenfeld alle aktuell verfügbaren Makros an (siehe Bild 9). Diese fügen wir wie unter Word zur Schnellzugriffsleiste hinzu. Auch hier können wir im Dialog **Schaltfläche 'Ändern'** die Beschriftung anpassen und ein Icon hinzufügen.

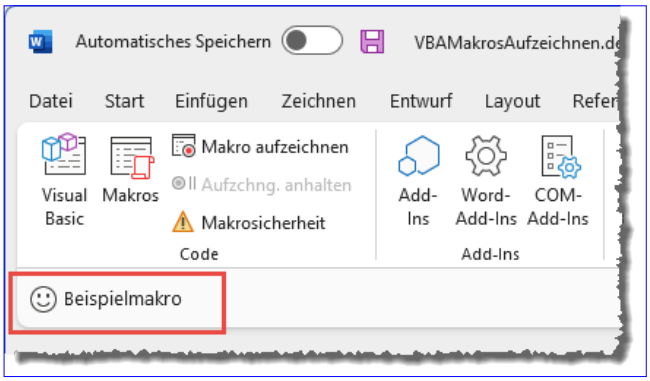
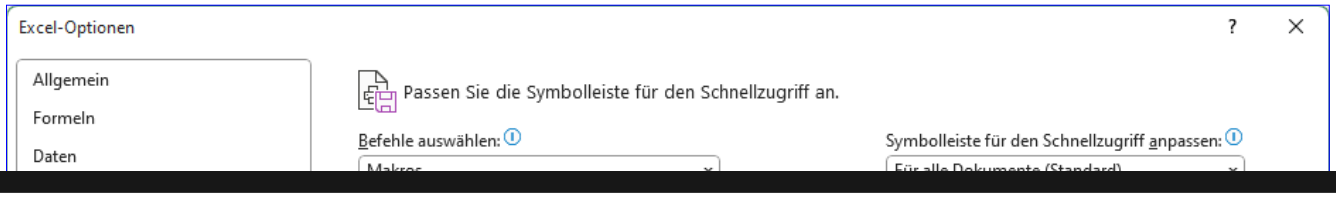


Bild 8: Eintrag für das Makro in der Schnellzugriffsleiste

Tastenkombination für Makro unter Word einstellen

Wenn wir im Dialog **Makro aufzeichnen** in Word auf die Schaltfläche Tastatur klicken, aktivieren wir einen weiteren Dialog namens **Tastatur anpassen** (siehe Bild 10). Dabei handelt es sich um den auch in anderem Kontext einsetzbaren Dialog zum Anpassen von Tastenkombinationen. In der Kategorie **Makros** finden wir das aktuelle Makro vor und können eine Tastenkombination für dieses festlegen.



VBA Basics: Module, Klassen und Co.

Unter VBA strukturieren wir den Code in verschiedene Elemente. Die übergeordneten Elemente sind die Module. Hier unterscheiden wir zwischen Klassenmodulen und Standardmodulen. Darunter können wir Variable, Konstanten und ähnliche deklarieren sowie auszuführende Anweisungen unterbringen. Diese Anweisungen müssen eine Voraussetzung erfüllen: Sie müssen in einer Sub- oder Function-Prozedur oder innerhalb einer Property-Methode eingetragen werden. In diesem Artikel schauen wir uns die Grundlagen von Modulen, Klassenmodulen und Objektmodulen im Detail an.

Module unter VBA

Module sind im Prinzip Textdokumente, die Code aufnehmen können. Allerdings sind diese Textdokumente nicht in Form einzelner Dateien verfügbar, wie es beispielsweise bei anderen Entwicklungsumgebungen beziehungsweise Programmiersprachen der Fall ist, sondern sie sind alle in einer einzigen Datei gespeichert. Unter Access befinden sich die Module gemeinsam mit den übrigen Elementen wie Tabellen, Abfragen, Formularen und Berichten direkt in der **.accdb**-Datei. Unter Excel, Word und PowerPoint landen sie im jeweiligen Dokument. Unter Outlook, wo es keine Dateien wie bei den anderen Office-Anwendungen gibt, liegt das VBA-Projekt sogar in Form einer eigenen Datei vor.

Module lassen sich allerdings auch von den übrigen Office-Anwendungen als Textdateien exportieren und so kannst Du sie auch in anderen Office-Dokumenten beziehungsweise Access-Datenbanken wieder importieren.

Modularten

Es gibt eigentlich nur zwei Modularten:

- **Klassenmodule**: Dies sind Module mit einigen besonderen Eigenschaften. In der Regel müssen diese erst initialisiert und mit einer Objektvariablen referenziert werden, damit man mit den enthaltenen Elementen arbeiten kann. In manchen Fällen kann man jedoch auch direkt auf diese zugreifen, ohne dass man sie explizit initialisieren muss. Beispiele für Klassenmodule sind die Module von Formularen und Berichten unter Access, die Module für die Arbeitsmappe und die einzelnen Tabellen unter Excel, das Modul **ThisDocument** zu einem Word-Dokument oder auch das Modul **ThisOutlookSession** unter Outlook.

- **Standardmodule**: Standardmodule kann man den VBA-Projekten aller Office-Anwendungen hinzufügen. Sie müssen im Gegensatz zu Klassenmodulen nicht initialisiert werden, daher stehen die darin enthaltenen Variablen, **Sub**- und **Function**-Prozeduren und andere Elemente jederzeit zur Ver-

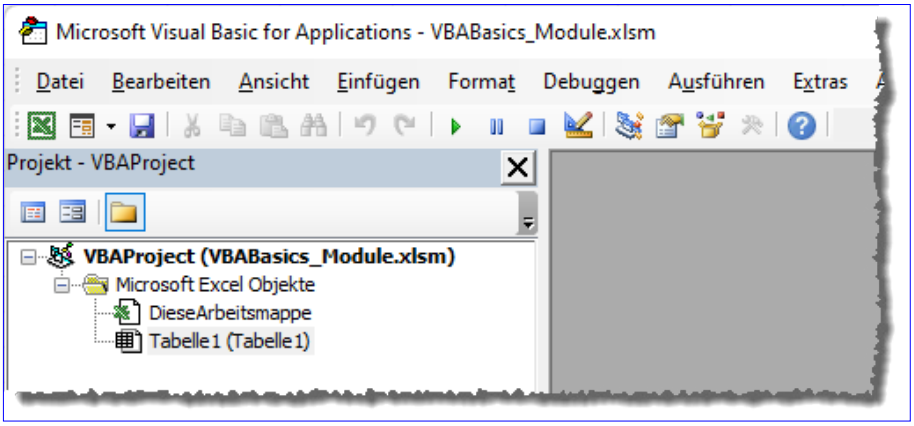


Bild 1: Verwalten von Modulen im Projekt-Explorer

fügung. Standardmodule kann man sehr gut nutzen, um Elemente mit gemeinsamen Eigenschaften zu speichern. So verwende ich beispielsweise immer ein Modul namens **mdlTools**, in dem ich allerlei Prozeduren und Funktionen angelegt habe, die ich an verschiedenen Stellen benötige.

Die Klassenmodule können wir allerdings noch einmal aufteilen in alleinstehende Klassenmodule und Klassenmodule von Office-Objekten wie Access-Formularen, Word-Dokumenten, Excel-Tabellenblättern et cetera, die beim Anlegen dieser Elemente oder spätestens beim Hinzufügen von Ereignisprozeduren zu diesen Elementen erstellt werden.

Alleinstehende Klassenmodule sind solche Klassenmodule, die wir selbst erstellen können und die nicht mit anderen Elementen wie Formularen, Word-Dokumenten et cetera verknüpft sind. In diesem kapseln wir zusammenhängende Variablen, Methoden, Funktionen und Ereignisse.

Auf Basis dieser benutzerdefinierten Klassenmodule lassen sich Objekte erstellen, die für die sogenannte objektorientierte Programmierung notwendig sind. Auch wenn die Möglichkeiten der objektorientierten Programmierung unter VBA nicht so umfangreich sind wie etwa unter C# oder VB.NET, wollen wir die-

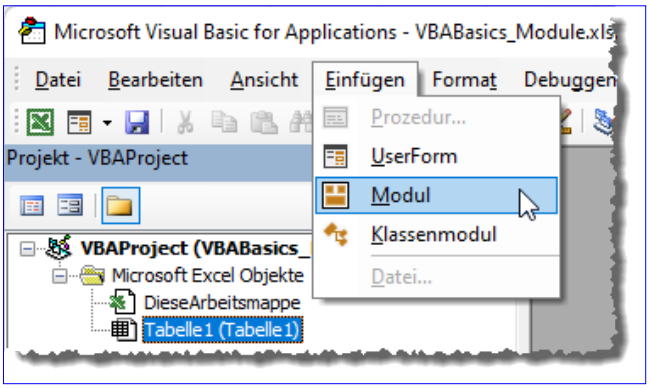


Bild 2: Hinzufügen eines Moduls per Menübefehl

Schließlich gibt es noch die Möglichkeit, **UserForm**-Elemente anzulegen, also Formulare für die jeweilige Anwendung. Diese zeigt der Projekt-Explorer unter **Formulare** an. Dem Thema **UserForm** widmen wir uns auch in einem eigenen Artikel.

Mit Modulen arbeiten

Die Verwaltung von Modulen erledigen wir im VBA-Editor im Bereich **Projekt-Explorer**. Dieser sieht beispielsweise für ein neues, leeres Excel-VBA-Projekt wie in Bild 1 aus. Hier sehen wir die beiden standardmäßig vorhandenen Module – eines, welches Elemente für die zum VBA-Projekt gehörende Arbeitsmappe aufnimmt und eines für die einzige bisher vorhandene Tabelle. Fügst Du neue Tabellen hinzu, landen entsprechend neue Klassenmodule im Projekt-Explorer.

VBA Basics: Makros, Prozeduren, Funktionen und Co.

VBA-Code in VBA-Projekten von Office-Anwendungen landet zuerst einmal in Modulen. Darunter gibt es einige weitere Strukturen, auf welche die Anweisungen aufgeteilt werden. Während Deklarationen von Variablen auch direkt in einem Modul angelegt werden können, müssen ausführbare Codezeilen zwingend in Konstrukten untergebracht werden, die je nach Anwendung Makros oder Prozeduren genannt werden. Außerdem gibt es noch Funktionen. Was es mit all diesen Begriffen auf sich hat und wieso diese nicht einheitlich definiert sind, erläutern wir in diesem Artikel.

Makros vs. Prozeduren

Wer erstmal mit VBA unter Excel, Word oder PowerPoint in Kontakt kommt, erledigt dies in der Regel über das Aufzeichnen von Makros – mehr dazu im Artikel **VBA: Makros aufzeichnen** (**www.access-im-unternehmen.de/324**). Bei der Aufzeichnung kommt ein sogenanntes Makro heraus. Schaut man sich das im VBA-Editor an, findet man jedoch eine Sub-Prozedur vor. Wir könnten nun einfach die Begriffe Makro und Sub-Prozedur synonym verwenden, wenn da nicht noch Access wäre: Hier gibt es nämlich neben VBA noch einen eigenen Objekttyp zum Erstellen von Automatisierungen, der wiederum Makro heißt.

Allerdings hat das Makro unter Access nichts mit VBA zu tun, sondern es handelt sich dabei eher um eine Möglichkeit zum »Zusammenklicken« von Automatisierungen. Microsoft hat also den Begriff Makro in verschiedenen Office-Anwendungen für unterschiedliche Techniken verwendet, was bei Benutzern und Programmierern für Verwirrung sorgt.

Allerdings ist das nicht nur dort problematisch, sondern auch für unser Magazin **Visual Basic Entwickler**. Wir wollen hier ja über den Einsatz von **Visual Basic for Applications** (**VBA**) und verwandten Sprachen wie **Visual Basic.NET** oder **twinBASIC** berichten, und zwar für alle Office-Anwendungen.

Um Verwechslungen auszuschließen, werden wir also den Begriff »Makro« nur noch dort verwenden, wo wir tatsächlich die Funktion zum Aufzeichnen von Makros nutzen. An allen anderen Stellen bezeichnen wir alle Code-Konstrukte, die in der ersten Zeile das Schlüsselwort **Sub** tragen, als Prozedur.

Prozeduren und Funktionen

Neben den Prozeduren gibt es noch zwei weitere Konstrukte, die zur Ausführung bestimmte Codezeilen aufnehmen können. Das erste ist die sogenannte Funktion, das zweite sind die **Property...**-Methoden. Diese treten allerdings erst in Zusammenhang mit der objektorientierten Programmierung von Klassenmodulen auf, sodass wir in diesem Artikel nicht darauf eingehen wollen.

Eigentlich sind die Bezeichnungen »Prozedur« und »Funktion« auch wieder nicht ganz konsequent, denn eigentlich sind beide Prozeduren – eine mit dem Schlüsselwort **Sub** und eine mit dem Schlüsselwort **Function**. Die korrekteren Bezeichnungen sind wohl **Sub**-Prozedur und **Function**-Prozedur. Allerdings denke ich, dass auch die Bezeichnungen **Prozedur** und **Funktion** verständlich sind, solange sie konsequent verwendet werden.

Prozeduren und Funktionen führen beide eine oder mehrere Anweisungen aus. Der wichtigste Unterschied zwischen beiden ist, dass die Funktionen ein Funktionsergebnis direkt zurückgeben können. Dazu zwei Beispiele. Eine Prozedur sieht im einfachsten Fall wie folgt aus:

```
Sub Meldung()
    MsgBox "Meldung per Prozedur"
End Sub
```

Eine sehr einfache Funktion sieht so aus:

```
Function Eingabe()
    Eingabe = InputBox("Funktionswert eingeben:")
End Function
```

Wir sehen schon, dass die Prozedur lediglich eine Meldung anzeigt. Die Funktion hingegen ruft eine InputBox auf und weist das Ergebnis einer Variablen namens **Eingabe** zu.

Aber was soll das denn nun? Wir haben doch im Artikel **VBA-Basics: Variablen** (**www.access-im-unternehmen.de/319**) geschrieben, dass Variablen ein Präfix wie zum Beispiel **str** für eine **String**-Variable enthalten sollen? Und überhaupt – es ist nirgends eine Variable namens **Eingabe** deklariert?

Und das ist der Punkt: Der Name einer Funktion ist gleichzeitig der Name der Variablen, der wir den Rückgabewert der Funktion zuweisen. Es ist also erstens kein Präfix erforderlich, außer Du möchtest diesen bei Funktionen verwenden. Zweitens ist aber auch kein Datentyp angegeben, obwohl wir das im oben genannten Artikel ebenfalls empfohlen haben, weil sonst der Datentyp **Variant** verwendet wird, der mehr Speicherplatz als ein explizit angegebener Datentyp benötigt.

Und wir haben geschrieben, dass die obigen Beispiele sehr einfache Beispiele sind. Dementsprechend haben wir auch nicht explizit einen Gültigkeitsbereich für die Prozedur und die Funktion angegeben.

Genau wie bei Variablen gilt hier: Wenn kein Schlüsselwort wie **Public** oder **Private** angegeben wurde, ist das Element öffentlich erreichbar. Explizit müssten wir die Prozedur also wie folgt definieren, wenn diese im kompletten VBA-Projekt erreichbar sein soll:

```
Public Sub Meldung()
    MsgBox "Meldung per Prozedur"
End Sub
```

Für die Funktion gilt das Gleiche – und hier geben wir nun auch noch den Datentyp für den Rückgabewert **Eingabe** an, der nicht nur Funktionsname, sondern auch noch Variable ist:

```
Public Function Eingabe() As String
    Eingabe = InputBox("Funktionswert eingeben:")
End Function
```

Routine als Synonym für Prozeduren und Funktionen

Wir werden im folgenden und auch in weiteren Artikeln Grundlagen und Techniken vorstellen, die für Prozeduren und Funktionen identisch sind. An diesen Stellen erwähnen wir nicht immer »Prozeduren und Funktionen«, sondern nutzen gelegentlich den Begriff Routine als Synonym für beide.

Aufruf per Direktbereich

Die Prozedur und die Funktion können wir beide über den Direktbereich des VBA-Editors aufrufen. Dazu aktivieren wir diesen, sofern noch nicht eingeblendet, mit **Strg + G**. Wenn er schon eingeblendet ist, kannst Du ihn mit dieser Tastenkombination aktivieren.

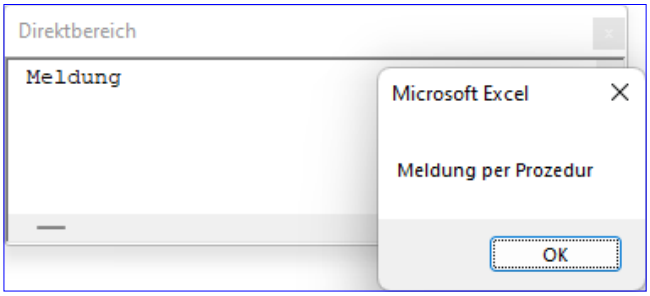


Bild 1: Aufruf einer Prozedur

Dort gibst Du einfach den Namen der Prozedur ein und schließt den Aufruf mit der Eingabetaste ab. Das Ergebnis sieht wie in Bild 1 aus.

Die Funktion können wir auf die gleiche Weise aufrufen. Das zeigt die **InputBox** an, aber die Eingabe wird nicht ausgewertet. Um den Inhalt der **InputBox** auszugeben, können wir beispielsweise die folgende Anweisung nutzen:

```
Debug.Print Eingabe
```

Dies ruft die Anzeige der **InputBox** wie in Bild 2 hervor. Und nach der Eingabe wird der eingegebene Text unter der Anweisung im Direktbereich ausgegeben.

Aufruf einer Funktion von einer anderen Prozedur oder Funktion aus

Normalerweise soll das Ergebnis einer Funktion jedoch nicht im Direktbereich ausgegeben werden, sondern wir speichern es in einer Variablen oder verwenden es anderweitig weiter.

Außerdem wollen wir in diesem Zusammenhang direkt einmal zeigen, wie wir die Funktion von einer anderen Funktion aus aufrufen können. Dazu deklarieren wir im folgenden Beispiel eine **String**-Variable namens **strEingabe** und weisen nun dieser das Ergebnis der Funktion **Eingabe** zu. Danach geben wir dieses wieder im Direktbereich aus, um zu prüfen, ob das Ergebnis korrekt verarbeitet wurde:

```
Public Sub EingabeSpeichern()
    Dim strEingabe As String
    strEingabe = Eingabe
    Debug.Print strEingabe
End Sub
```

Die Prozedur **EingabeSpeichern** rufen wir zunächst wieder vom Direktbereich aus auf, indem wir ihren Namen eingeben und die Eingabetaste betätigen.

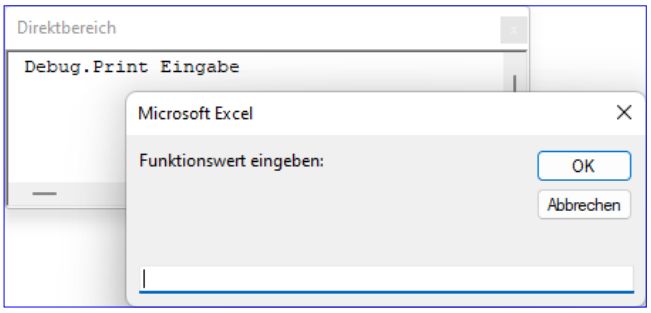


Bild 2: Aufruf einer Funktion

Aufruf einer anderen Routine unterbricht die aktuelle Routine

Wichtig für das Verständnis des Ablaufs ist, dass Routinen unter VBA immer synchron ablaufen, dass heißt, dass immer nur eine gleichzeitig ausgeführt wird. Wenn Routine A also mit einer Anweisung Routine B aufruft, dann wird Routine A solange angehalten, bis Routine B beendet ist. Anderenfalls würde man von asynchroner Ausführung reden, was aber in reinem VBA nicht möglich ist.

Aufruf einer Prozedur von einer anderen Prozedur aus

Wenn wir eine Prozedur von einer anderen Prozedur aus aufrufen wollen, können wir dies genau wie im Direktbereich erledigen.

Wir tragen also einfach den Namen der zu startenden Prozedur als eigene Zeile innerhalb einer anderen Prozedur ein – hier wieder mit der zuerst vorgestellten Prozedur namens **Meldung**:

```
Public Sub AndereProzedurAufrufen()
    Meldung
End Sub
```

Dies funktioniert, aber etwas sauberer wirkt die Variante, in der wir die **Call**-Anweisung für den Aufruf der Prozedur nutzen:

```
Call Meldung
```

Dies können wir auch im Direktbereich verwenden. Die **Call**-Anweisung hat keine Nachteile, aber sie kann in bestimmten Situationen Vorteile liefern. Diese sind aber rar. Wenn Du zum Beispiel zwei Prozeduraufrufe in einer Zeile unterbringen möchtest, was unüblich ist und bestenfalls ein oder zwei Codezeilen spart, kannst Du Call einsetzen:

```
Call Meldung: Call Eingabe
```

Wenn Du hier nur **Meldung : Eingabe** schreibst, wird Meldung als Sprungmarke interpretiert und der Code sieht anschließend so aus:

```
Meldung:
    Eingabe
```

Prozeduren und Funktionen direkt aufrufen

Es gibt noch weitere Methoden für den Aufruf von Prozeduren und Funktionen. Diese funktionieren aber nur, solange diese keine Parameter verwenden. Ist das der Fall, kannst Du die auszuführende Prozedur oder Funktion einfach im VBA-Editor markieren, indem Du die Einfügemarke an einer beliebigen Stelle des Codes der Prozedur oder Funktion platzierst und dann eine der folgenden Aktionen ausführst:

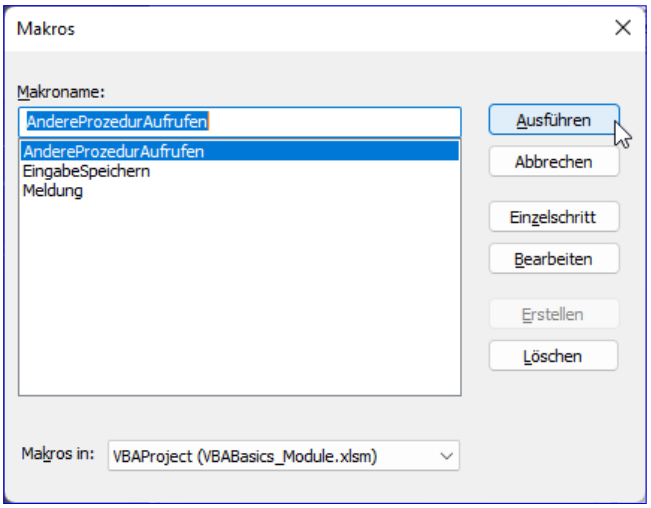


Bild 3: Dialog zum Aufrufen einer der ausführbaren Prozeduren

spielsweise würde dieser Dialog auch die Prozeduren anzeigen, die sich in den Klassenmodulen der aktuellen Arbeitsmappe und der Arbeitsblätter befinden. Dieser zeigt alle verfügbaren ausführbaren Prozeduren an. Die ausführbaren Funktionen erkennt der Dialog nicht.

Gültigkeitsbereich von Prozeduren und Funktionen im Standardmodulen

Wenn wir über die Definition von Routinen reden, dann müssen wir auch das Thema der Gültigkeit anreißen. Je nach verwendeten Schlüsselwörtern

VBA Basics: Bedingungen

Wenn man in VBA-Routinen bestimmte Anweisungen in Abhängigkeit von einem Wert einer Variablen, einer Eingabe oder anderen Bedingungen ausführen lassen möchte, verwendet man sogenannte Bedingungen. Unter VBA gibt es dazu die If...Then-Bedingung und die Select Case-Bedingung. Streng genommen gibt es noch einige VBA-Funktionen, die auch Bedingungen enthalten. Diese schauen wir uns aber in einem anderen Artikel an. Hier geht es zunächst um die beiden genannten Konstrukte.

Manche Prozeduren oder Funktionen unter VBA erfordern es, dass abhängig von bestimmten Voraussetzungen unterschiedliche Anweisungen ausgeführt werden sollen. Ein einfaches Beispiel: Wenn Du per **MsgBox**-Funktion vom Benutzer einen der Werte **vbYes** oder **vbNo** abfragst, dann wirst Du im Code auch für die beiden Eingaben jeweils eigene Anweisungen ausführen wollen. Wenn nur die beiden Möglichkeiten bereitstellen, bist Du mit einer einfachen **If...Then**-Bedingung gut vorbereitet. Es kann auch sein, dass es noch mehr verschiedene Auswahlmöglichkeiten gibt. Dann würde man eine **If...Then**-Bedingung mit mehr als nur zwei Zweigen verwenden – oder vielleicht sogar eine **Select Case**-Bedingung, die das Verwalten mehrerer Zweige oft einfacher und übersichtlicher macht.

Die If...Then-Bedingung

Eine **If...Then**-Bedingung kann nur aus einem einzigen Zweig bestehen, dessen Anweisungen nur bei der Erfüllung der angegebenen Bedingung ausgeführt werden. Die Bedingung ist im einfachsten Fall wie folgt aufgebaut:

```
If [Bedingung] Then
    [Anweisungen]
End If
```

[Bedingung] kann dabei ein beliebiger Ausdruck sein, der den Wert **True** oder **False** zurückliefert. Das Ergebnis des für **[Bedingung]** angegebenen Ausdrucks muss also den Datentyp **Boolean** aufweisen. Wenn **[Bedingung]** den Wert **True** enthält, werden die zwischen **If...Then** und **End If** enthaltenen Anweisungen ausgeführt. Das sieht im einfachsten Fall wie folgt aus:

```
Dim bolBedingung As Boolean
bolBedingung = True
If bolBedingung Then
    MsgBox "bolBedingung ist True"
End If
```

Viele Entwickler schreiben die **If...Then**-Zeile in diesem Fall wie folgt:

```
If bolBedingung = True Then
...
```

Das ist nicht grundsätzlich falsch, aber **bolBedingung** enthält bereits einen **Boolean**-Wert, der Vergleich mit **True** hat hier keinen zusätzlichen Nutzen und kostet lediglich Rechenzeit. Wenn Du allerdings im **If...Then**-Zweig prüfen willst, ob die Bedingung nicht erfüllt ist, dann benötigen wir mindestens einen weiteren Operator. Die erste Möglichkeit ist der Vergleich mit dem Wert **False**:

```
If bolBedingung = False Then
    MsgBox "bolBedingung ist False"
End If
```

Die zweite Möglichkeit ist der Einsatz des Operators **Not**:

```
If Not bolBedingung Then
    MsgBox "bolBedingung ist False"
End If
```

In den folgenden Beispielen verwenden wir statt der **Boolean**-Variable einen richtigen Vergleichsausdruck, dessen Ergebnis wir mithilfe einer **MsgBox**-Anweisung ermitteln. Klickt der Benutzer auf **Ja**, erscheint die nachfolgend definierte Meldung:

```
Dim lngResult As VbMsgBoxResult
lngResult = MsgBox("Ja oder Nein?", vbYesNo)
If lngResult = vbYes Then
    MsgBox "Benutzer hat 'Ja' angeklickt."
End If
```

Der Else-Zweig

In einer **If...Then**-Bedingung sind wir allerdings nicht darauf beschränkt, nur Anweisungen für die im **If... Then**-Teil angegebene Bedingung auszuführen. Wir können auch Anweisungen ausführen, wenn die Bedignung nicht erfüllt ist. Die einfachste und universellste Variante dafür ist der **Else**-Zweig.

Im folgenden Beispiel behandeln wir den Fall, dass der Benutzer auf **Nein** klickt, im **Else**-Zweig:

```
lngResult = MsgBox("Ja oder Nein?", vbYesNo)
If lngResult = vbYes Then
    MsgBox "Benutzer hat 'Ja' angeklickt."
Else
    MsgBox "Benutzer hat 'Nein' angeklickt."
End If
```

Da **lngResult** nur die beiden Wert **vbYes** oder **vbNo** enthalten kann, wird **vbNo** im **Else**-Zweig abgehandelt. Wenn wir neben **Ja** und **Nein** noch weitere Möglichkeiten anbieten, zum Beispiel **Abbrechen**, würde der **Else**-Zweig auch diese behandeln:

```
lngResult = MsgBox("Ja oder Nein?", vbYesNoCancel)
```

Klickt der Benutzer also auf **Abbrechen**, zeigt die vorherige **If...Then**-Bedingung auch an, dass der Benutzer auf **Nein** geklickt hat. Um dies zu behandeln, gibt es noch den **ElseIf**-Zweig.

Der ElseIf-Zweig

Wenn wir nicht nur zwei Möglichkeiten auswerten wollen, müssen wir entsprechend mehr Zweige anbieten. Im Falle von **Ja**, **Nein** und **Abbrechen** fügen wir im folgenden Beispiel eine weitere Unterscheidung hinzu, die gezielt auf den Wert **vbNo** untersucht. Alles, was nicht **vbYes** oder **vbNo** ist, wird im **Else**-Teil behandelt, in diesem Fall also **vbCancel**:

```
lngResult = MsgBox("Ja oder Nein?", vbYesNoCancel)
If lngResult = vbYes Then
    MsgBox "Benutzer hat 'Ja' angeklickt."
ElseIf lngResult = vbNo Then
    MsgBox "Benutzer hat 'Nein' angeklickt."
Else
    MsgBox "Benutzer hat 'Abbrechen' angeklickt."
End If
```

Wir könnten im **Else**-Teil auch explizit auf **vbCancel** prüfen, indem wir diesen wie folgt in einen **ElseIf**-Zweig umwandeln:

```
If lngResult = vbYes Then
    MsgBox "Benutzer hat 'Ja' angeklickt."
ElseIf lngResult = vbNo Then
    MsgBox "Benutzer hat 'Nein' angeklickt."
ElseIf lngResult = vbCancel Then
    MsgBox "Benutzer hat 'Abbrechen' angeklickt."
End If
```

Das passt in diesem Fall, da wir genau drei Fälle behandeln wollen.

Else-Zweig am Ende

Wenn Du einen **Else**-Zweig verwendest, musst Du diesen hinter allen verwendeten **ElseIf**-Zweigen einfü-

gen. Anderenfalls wird beim Kompilieren der Fehler aus Bild 1 ausgelöst.

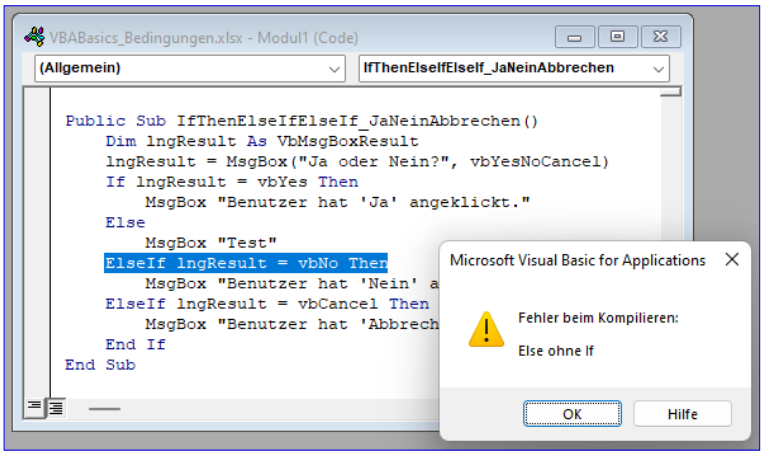


Bild 1: Fehler, wenn der **Else**-Zweig nicht am Ende steht

Else-Zweig als Backup

In den meisten Fällen weiß man vorher, welche Werte in einer **If...Then**-Bedingung verarbeitet werden sollen. Manchmal können jedoch auch Werte auftauchen, die man zuvor noch nicht kennt oder erwartet. In diesem Fall kann es sinnvoll sein, den **Else**-Zweig mit einer entsprechenden Meldung oder einer anderen Aktion auszustatten, mit der sowohl der Benutzer als auch der Entwickler informiert werden, wenn es unbehandelte Fälle gibt.

Performance optimieren durch richtige Reihenfolge

Wenn Du Schritt für Schritt durch eine solche **If...Then**-Bedingung läufst, stellst Du fest, dass die Bedingungen von oben nach unten geprüft werden und dass nach dem Erreichen einer gültigen Bedingung die im entsprechenden Zweig enthaltenen Anweisungen ausgeführt werden und die nachfolgenden Zweige nicht mehr untersucht werden.

Da auch das schlichte Untersuchen der Bedingungen in einem **If...Then...ElseIf...Else**-Konstrukt Rechenzeit kostet und sich das beispielsweise bei vielen Wie-

Diese sieht beispielsweise wie folgt aus:

```
Dim bol As Boolean
bol = True
If bol Then MsgBox "True"
```

Damit lassen sich **If...Then**-Bedingungen ohne **ElseIf** oder **Else**-Zweig abbilden.

Allerdings sind wir hier nicht eine einzige Anweisung beschränkt – wir können auch mehrere hintereinander angeben:

```
If bol Then MsgBox "True": Debug.Print "True"
```

VBA-Basics: MsgBox- und InputBox-Funktion

Die beiden Funktionen MsgBox und InputBox ermöglichen die schnelle Abfrage von Benutzerfeedback. Soll eine Datei überschrieben werden? Wie soll die neue Kategorie heißen? Das sind nur zwei von vielen Beispielen für den Einsatz dieser beiden Funktionen. Dieser Artikel beschreibt die beiden so kompakt wie möglich, damit Du sie schnell in Deine eigenen Anwendungen integrieren kannst.

Die MsgBox-Funktion

Die erste der beiden Funktionen hat zwei Haupteinsatzzwecke:

- Ausgabe von Informationen oder Warnungen an den Benutzer
- Ermitteln von einfachen Antworten des Benutzers wie OK oder Abbrechen durch entsprechende Schaltflächen

MsgBox zur Anzeige von Meldungen

Wenn Du einfach nur eine Meldung mit Informationen anzeigen möchtest, reicht der einfache Aufruf der **MsgBox**-Funktion unter Angabe der anzuzeigenden Informationen und gegebenenfalls eines Symbols.

Um dem Benutzer mehrere Auswahlmöglichkeiten für eine Reaktion zu geben, kannst Du verschiedene Schaltflächen anzeigen lassen und diese abfragen.

Der einfachste **MsgBox**-Aufruf gibt einfach nur einen Text aus und zeigt eine **OK**-Schaltfläche an:

```
MsgBox "Dies ist eine Beispielmeldung."
```

Diese können wir über den Direktbereich des VBA-Editors aufrufen oder auch in eine Prozedur oder Funktion einbinden. Das Ergebnis siehst Du in Bild 1.

MsgBox mit Titel

Die oben erzeugte Meldung zeigt den gewünschten Text an, und auch die Titelzeile ist gefüllt. Wenn Du den dortigen Text ersetzen willst, der standardmäßig den Namen der übergeordneten Anwendung anzeigt, brauchst Du nur den dritten Parameter der **MsgBox**-Funktion mit dem gewünschten Text zu füllen:

```
MsgBox "Beispielmeldung mit individuellem Titel", , _
    "Unser Titel"
```

Das Ergebnis siehst Du in Bild 2. Alternativ und um die unübersichtliche Anzeige der Kommata zwischen den Parametern zu entschärfen kannst Du auch benannte Parameter verwenden, wenn sich die Parameter nicht an der vorgesehenen Stelle befinden (für den ersten Parameter ist dies daher nicht erforderlich):

```
MsgBox "Beispielmeldung mit individuellem Titel", _
    Title:="Unser Titel"
```

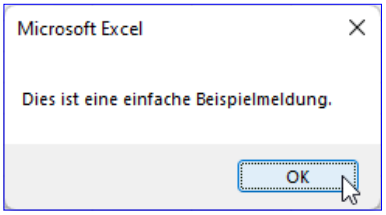


Bild 1: Einfache Meldung

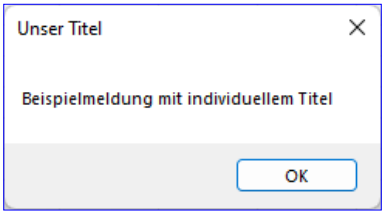


Bild 2: Meldung mit Titel

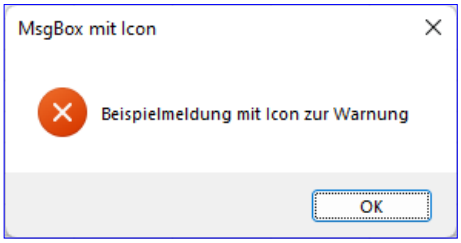


Bild 3: Meldung mit Icon

Icons für die MsgBox

Im nächsten Schritt fügen wir der Meldung wie in Bild 3 ein Icon hinzu. Das erledigen wir mit dem zweiten Parameter namens **Buttons**. Die Bezeichnung deutet eigentlich eher darauf hin, dass wir hiermit Informationen über Schaltflächen angeben, aber dazu gleich mehr. Es ist nämlich beides möglich. Der Aufruf für die gezeigte Meldung lautet:

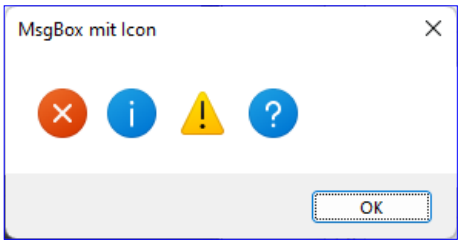


Bild 4: Meldung mit allen Icons

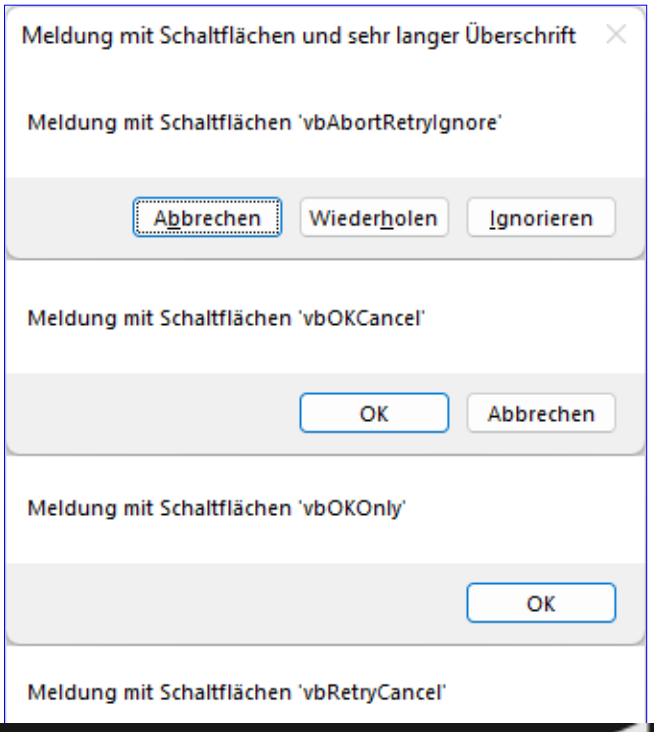
```
MsgBox "Beispielmeldung mit Icon zur Warnung", _
    vbCritical, "MsgBox mit Icon"
```

Wir übergeben also die Konstante **vbCritical**, um das Icon für einen Warnhinweis auszugeben.

Die möglichen Konstanten für die anzuzeigenden Icons lauten:

- **vbCritical**: Zeigt einen roten Warnhinweis an.
- **vbExclamation**: Zeigt ein Ausrufezeichen an.
- **vbInformation**: Zeigt ein Informations-Icon an.
- **vbQuestion**: Zeigt ein Fragezeichen-Icon an.

dern auch noch die anzuzeigenden Schaltflächen festlegen können. Wenn Du kein Icon anzeigen möchtest, sondern nur verschiedene Schaltflächen, gibst Du einfach nur die Konstante für die gewünschten Schaltflächen an.



Umfangreiche Texte in Code integrieren

Es gibt Aufgaben, um die schlägt man sich nicht. Eine davon ist es, längere Texte hart im VBA-Code zu verdrahten, sodass diese später weiterverarbeitet werden können. Ein Beispiel ist das Zusammenstellen eines XML-Dokuments, von dem man eine Vorlage hat, und das man mit eigenen Werten füllen möchte, um es dann beispielsweise als Anfrage an einen Webservice zu schicken. Oder man möchte den Inhalt einer Mail per VBA an Outlook schicken und versenden, nachdem man den Mailtext noch um individuelle Inhalte wie Anrede oder Name des Empfängers ergänzt hat. Sprich: Uns liegt ein mehrzeiliger Text vor, den wir irgendwie in eine Variable packen wollen – und zwar ausschließlich per VBA-Code. Wie das gelingt, zeigt der vorliegende Artikel.

Mein aktuelles Beispiel, das mich zum Entwickeln der in diesem Artikel vorgestellten Lösung veranlasst hat, ist das Zusammenstellen eines XML-Dokuments. Wer schon einmal ein XML-Dokument mit der Anfrage an einen Webservice geschickt hat, kennt das vielleicht – in der Dokumentation des Webservice-Betreibers finden wir ein umfassendes XML-Dokument, das mit Beispieldaten gefüllt ist und das nur noch an unsere eigenen Zwecke angepasst werden soll. Eine Zeile darin lautet dann beispielsweise:

```
<password>pass</password>
```

Wir wollen diese samt den restlichen Zeilen des XML-Dokuments mit unseren realen Daten anpassen und das resultierende Dokument in einer Variablen speichern, um es dann an den Webservice zu schicken. Nur für diese eine Zeile würden wir dann eine Prozedur wie die folgende programmieren:

```
Public Sub XMLZusammenstellen()
    Dim strText As String
    strText = "<password>pass</password>" & vbCrLf
    Debug.Print strText
End Sub
```

Im nächsten Schritt würden wir dann das hier fest im Code untergebrachte Kennwort **pass** durch einen Parameter ersetzen. Diese Version sieht wie folgt aus:

```
Public Sub XMLZusammenstellen(strPassword As String)
    Dim strText As String
    strText = "<password>" & strPassword _
        & "</password>" & vbCrLf
    Debug.Print strText
End Sub
```

Dann rufen wir die Prozedur mit einem Parameterwert wie folgt auf:

```
XMLZusammenstellen "MeinKennwort"
```

Das liefert dann:

```
<password>MeinKennwort</password>
```

Für eine einzige Zeile ist der Aufwand zum Parametrisieren überschaubar, für ein paar weitere Zeilen vielleicht auch noch. Aber das XML-Dokument, das zu füllen war, sah in den VBA-Code kopiert zunächst wie in Bild 1 aus. Das heißt, wir müssen jede Zeile mit Ausnahme der ersten wie folgt ergänzen, also vorn um die Zuweisung des bisher in der Variablen gespeicherten Textes plus der neuen Zeile an die Variable:

```
strXML = strXML & "   <soapenv:Header>" & vbCrLf
```

Wenn wir das für einige zig Zeilen erledigen müssen, wird es Zeit, über eine Automatisierung nach-

zudenken. Darüber hinaus müssen wir wie bei der in XML-Dokumenten üblichen ersten Zeile, die noch die Definition von Namespaces enthält, noch einfache Anführungszeichen durch doppelte Anführungszeichen ersetzen wie bei diesem Beispiel (die eigentlichen Namespace-URLs haben wir aus Gründen der Übersicht gekürzt):

```
<soapenv:Envelope xmlns:soapenv="..." xmlns:cis="..."
xmlns:ns="...">
```

Daraus wird dann die folgende Zeile – auch hier mit gekürzten Namespace-URLs (in einer Zeile):

```
strXML = strXML & "<soapenv:Envelope xmlns:soapenv=""..."" 
xmlns:cis=""..."" xmlns:ns=""..."">" & vbCrLf
```

Code per Code anpassen

Die Anpassungen sind, so nervig sie auch sein mögen, nicht besonders komplex. Also können wir dies vermutlich recht einfach automatisieren, wenn wir die richtigen Befehle herausfinden. Dazu benötigen wir als Erstes eine Bibliothek, welche uns den Zugriff auf das Objektmodell des VBA-Editors ermöglicht und so auch den Zugriff auf den Inhalt von Codefenstern. Diesen Verweis fügen wir über den **Verweise**-Dialog hinzu, den wir

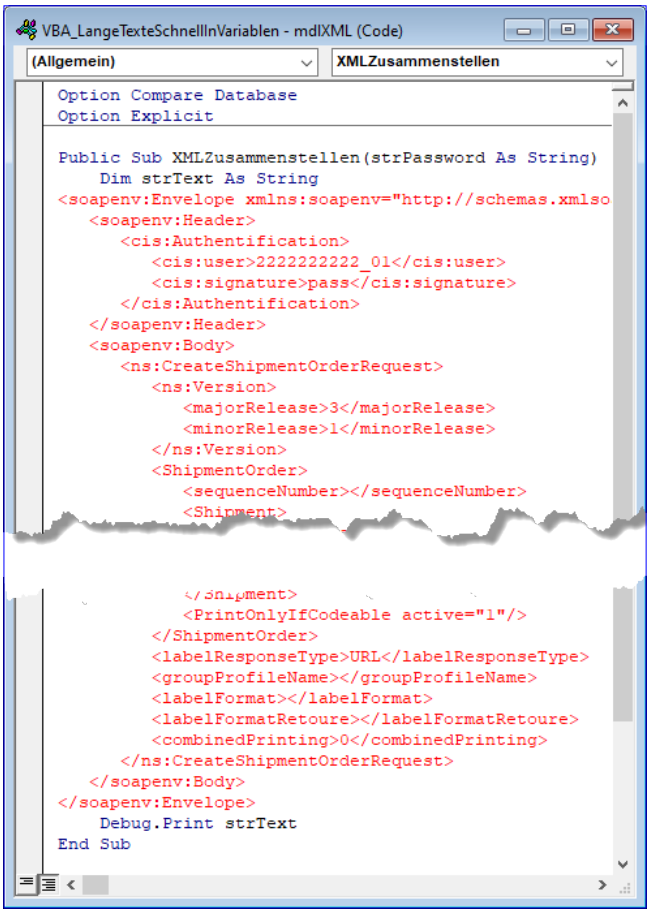


Bild 1: Ein langes XML-Dokument soll in den VBA-Code eingearbeitet werden.

vom VBA-Editor aus mit **Extras|Verweise** öffnen. Die gewünschte Bibliothek heißt **Microsoft Visual Basic for**

Ribbons in Office-Dokumenten

Word, Excel und PowerPoint sind die Office-Anwendungen, mit denen Du Dokumente anzeigen und bearbeiten, aber auch automatisieren kannst. Wenn Du beispielsweise einer Excel-Arbeitsmappe eigene, per VBA programmierte Funktionen hinzufügen möchtest, musst Du diese irgendwie aufrufen. Eine Schaltfläche in einem Excel-Arbeitsblatt ist eine Möglichkeit. Die Alternative ist, einen entsprechenden Befehl im Ribbon zu platzieren. Wie das gelingt, zeige ich im vorliegenden Artikel. Dabei erfährst Du auch einige Dinge rund um die Office Open XML-Formate und wie die Daten in diesen Dokumenten gespeichert sind – darunter auch die Definitionen von Ribbons und die darin anzuzeigenden Icons.

Grundlagen: Das Office Open XML-Format

Jede Datei, die in einem der **Office Open XML**-Formate gespeichert ist, besteht aus einem Dateicontainer, der auf einer einfachen, komponentenbasierten und komprimierten ZIP-Dateiformatspezifikation aufbaut. Das gilt zum Beispiel für Dateien mit der Dateiendung **.docx**, **.xlsx** oder **.pptx**, aber natürlich auch für verwandte Dokumentformate.

Kern der **Office Open XML**-Formate ist die Verwendung von XML-Referenzschemas und eines ZIP-Containers. Jede Datei setzt sich aus einer Auflistung einer beliebigen Anzahl von Komponenten zusammen. Diese Auflistung definiert das Dokument.

Dokumentkomponenten werden mithilfe des ZIP-Formats in der Containerdatei beziehungsweise dem Paket gespeichert. Bei den meisten Komponenten handelt es sich um XML-Dateien, die in der Containerdatei gespeicherte Anwendungsdaten, Metadaten und sogar Kundendaten beschreiben.

Im Containerpaket können andere, Nicht-XML-Komponenten einschließlich Komponenten wie Binärdateien, die im Dokument eingebettete Bilder oder OLE-Objekte darstellen, enthalten sein. Zusätzlich werden durch Beziehungskomponenten die Beziehungen zwischen Komponenten festgelegt.

Dieser Entwurf stellt die Struktur für Office-Dateien dar. Während sich der Inhalt der Datei aus Komponenten zusammensetzt, beschreiben die Beziehungen, wie die einzelnen Komponenten zusammenarbeiten.

Die Zusammensetzung einer Datei nach dem **Office Open XML**-Standard kannst Du Dir ansehen, indem

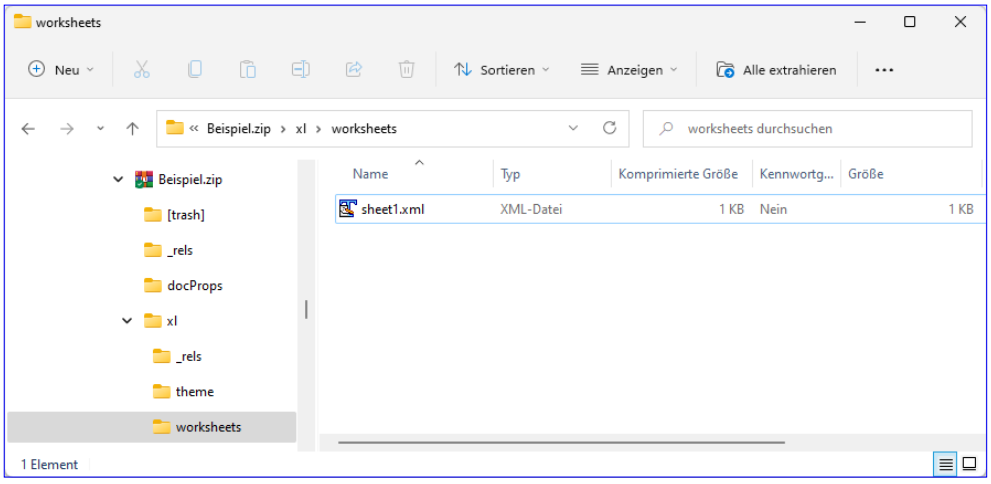


Bild 1: Komprimierter Zip-Container einer Excel-Arbeitsmappe

Du ein solches Dokument extrahierst. Dazu gehst Du wie folgt vor – hier am Beispiel einer neuen, leeren Excel-Datei mit dem Dateinamen **Beispiel.xlsx**:

- Benenne die Datei im Windows Explorer in **Beispiel.zip** um.
- Bestätige den nun erscheinenden Hinweis mit einem Klick auf die Schaltfläche **Ja**.
- Durch einen Doppelklick auf die Datei wird die Ordnerstruktur geöffnet und die Einzelkomponenten, aus der die Datei besteht, werden angezeigt. Je nach Inhalt der Datei werden nur die tatsächlich benötigten Ordner zur Speicherung einzelner Komponenten erstellt.
- Um die Datei wieder in der Office-Anwendung bearbeiten zu können, stellst Du einfach den alten Dateinamen wieder her.

Wir wollen uns aber zuerst kurz den Inhalt anschauen. Die wichtigsten Daten, also die in den Arbeitsblättern gespeicherten Informationen, finden wir in einem Unterordner namens **xl|worksheets**. In Bild 1 sehen wir beispielsweise das mit **sheet1.xml** benannte Arbeitsblatt.

Ribbons anpassen

Die standardmäßig in Office integrierten Ribbons mit ihren Tabs, Groups und Controls wurden von Microsoft so in die interne Anwendungsstruktur integriert, dass eine direkte Bearbeitung durch den Benutzer oder Entwickler über die Benutzeroberfläche oder auch über ein Objektmodell, wie bei den Menü- und Symbolleisten von Office 2003 und älter, nicht möglich ist.

Stattdessen formulieren wir die gewünschten Änderungen etwa zum Ausblenden oder Deaktivieren integrierter Elemente oder zum Hinzufügen eigener Tabs, Groups und Controls in das Ribbon nach einem bestimmten Schema in der Auszeichnungssprache XML. Wie ein XML-Dokument zur Anpassung des Ribbons aussieht, ist an dieser Stelle noch gar nicht wichtig – darum kümmern wir uns in weiteren Artikeln. Erstmal interessiert uns, wie man die in einem XML-Dokument formulierten Anpassungen für den Benutzer sichtbar macht.

Dazu gibt es mehrere Möglichkeiten, die auch davon abhängen, in welchem Zusammenhang die Anpassungen gültig sein sollen. Es kann sein, dass der Benutzer eine dauerhafte Änderung wünscht, um etwa Befehle aus dem **Datei**-Menü direkt im **Start**-Tab einer Office-Anwendung verfügbar zu machen, oder eine Änderung soll nur in Zusammenhang mit einer bestimmten Dokumentvorlage wirksam werden. Vielleicht ist es aber auch nur ein einziges Dokument, wie eine Excel-Datei, die ja prinzipiell eine eigene Anwendung sein kann, das Änderungen am Ribbon hervorrufen soll.

Wo man nun Hand anlegen muss, um eine Anpassung des Ribbons zu erreichen, hängt auch von der Anwendung ab: Bei Excel, Word und PowerPoint kann man dies durch Manipulation des jeweils geladenen Dokuments erreichen (wobei dies auch eine normale oder eine globale Vorlage sein kann), bei Access funktioniert dies ebenfalls über eine, wenn auch anders geartete, Anpassung der geladenen Datei und in Outlook sind schon richtige Programmierkenntnisse gefragt.

Da wir die Vorgehensweise unter Access und Outlook in separaten Artikeln beschreiben, geht es zunächst zu den **Office Open XML**-Dokumenten, die ohne großen Änderungsaufwand sowohl als Dokumente als auch als Vorlagen verwendet werden können.

Ribbon-Definition als XML-Datei

Dort speichert man die gewünschten Ribbon-Anpassungen in einem XML-Dokument, das zwingend den Dateinamen **customUI.xml** haben muss und in einem Ordner mit der Bezeichnung **customUI** gespeichert

wird, der sich im Containerpaket der **Office Open XML**-Datei befindet. Es reicht aber nicht aus, die XML-Datei einfach nur innerhalb des Paketes zu speichern, wir müssen der Anwendung, die dieses Dokument öffnet, auch noch mitteilen, dass dort eine solche Datei enthalten ist.

Um eine Beziehung zwischen der Arbeitsmappendatei und der Anpassungsdatei herzustellen, muss in der sogenannten **.rels**-Datei im Verzeichnis **_rels** ein Verweis auf die XML-Anpassungsdatei hinterlegt werden.

Da es einen beträchtlichen Aufwand bedeutet, die genannten Dateien und Ordner bei jeder Datei, in der das Ribbon geändert werden soll, zu erstellen und zu modifizieren, verwenden Sie am besten den **Office RibbonX Editor**, den Sie unter folgender Adresse kostenlos downloaden können:

```
https://github.com/fernandreu/office-ribbonx-editor/releases
```

Der **Custom UI Editor** ist ein praktisches Tool zum Einfügen von Ribbon-XML-Code und der dazugehörigen Bilder in Dateien im **Office Open XML**-Format.

Anpassung des Ribbons für eine Excel-Arbeitsmappe

sonst im **Office RibbonX Editor** nicht geöffnet werden kann).

- Starte den **Office RibbonX Editor** und öffne die zuvor erstellte Excel-Datei.
- Die Datei wird nun wie in Bild 2 angezeigt. Öffne das Kontextmenü der Datei **Beispiel.xlsx** und wähle den Eintrag **Insert Office 2010+ Custom UI Part** aus.
- Ein Doppelklick auf den so hinzugefügten neuen Eintrag **customUI14.xml** öffnet den XML-Editor für diese Datei.
- Hier geben wir den Code wie in Bild 3 ein.
- Prüfe den XML-Code mit der **Validate**-Schaltfläche des **Office RibbonX Editors**.
- Speichere und schließe die Excel-Arbeitsmappe im **Office RibbonX Editor**.
- Öffne die Excel-Arbeitsmappe wieder in Excel, um Dir das Ergebnis der Ribbon-Änderung anzusehen.

Hier finden wir nun bereits die neue Schaltfläche vor (siehe Bild 4). Nun wollen wir noch ein Icon und eine

Outlook: Application_Startup feuert nicht

Das Ereignis »Startup« des »Application«-Objekts von Outlook ist für viele benutzerdefinierte Erweiterungen von Outlook essenziell, da es die Möglichkeit bietet, direkt beim Starten von Outlook VBA-Code auszuführen. Damit lassen sich für verschiedene Anwendungen wichtige Automatismen anstoßen – zum Beispiel das Deklarieren und Initialisieren von Objektvariablen, für die Ereignisse implementiert werden sollen. Leider passiert es gelegentlich, dass die Ereignisprozedur Application_Startup beim Starten von Outlook nicht aufgerufen wird. Woran das liegt und wir dies ändern, zeigt der vorliegende Artikel.

Du hast das Ereignis Application_Startup im Modul ThisOutlookApplication implementiert und wenn Du Outlook startest, wird diese nicht ausgelöst? Das kann verschiedene Gründe haben:

- Die Makroeinstellungen für Outlook erlauben die Ausführung von Makros nicht. Der Klassiker, der gerade bei neu installierten Office-Anwendungen auftritt.
- Outlook ist bereits geöffnet, gegebenenfalls auch unsichtbar. In diesem Fall kann **Application_Startup** nicht ausgelöst werden, weil Outlook ja gar nicht neu startet. Wenn Outlook per Code initialisiert wird, feuert **Application_Startup** übrigens nicht.
- Ein spezieller Registry-Eintrag, den Outlook beim Starten abfragt, weist einen Wert auf, der das Ausführen von **Application_Startup** unterbindet.

In den folgenden Abschnitten schauen wir uns die Lösungen für diese Probleme an.

Makroeinstellungen kontrollieren und anpassen

Wenn **Application_Startup** beim Öffnen von Outlook nicht ausgeführt wird, solltest Du als Erstes die Makroeinstellungen prüfen. Dazu brauchst Du gar nicht erst den Optionen-Dialog zu öffnen – wechsle einfach zum VBA-Editor (**Alt + F11**), öffne das Modul **ThisOutlookApplication**, platziere die Einfügemarke in der Prozedur **Application_Startup** und betätige die Taste **F5**.

Wenn die Prozedur nun auch nicht ausgeführt wird, sondern die Meldung aus Bild 1 erscheint, hast Du schon die Lösung des Problems gefunden.

Um Makros zu aktivieren, öffnest Du mit dem Ribbonbefehl **Datei|Optionen** den Optionen-Dialog von Outlook.

Hier wechselst Du zum Bereich **Trust Center** und klickst dort auf **Einstellungen für das Trust Center...**, was den Dialog **Trust Center** öffnet.

Dort aktivierst Du im Bereich **Makroeinstellungen** eine der Optionen **Benachrichtigungen für alle Makros** oder **Alle Makros aktivieren**.

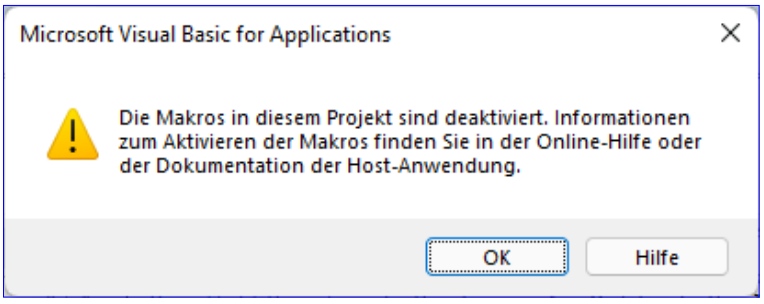


Bild 1: Meldung beim Versuch, ein Makro trotz Deaktivierung auszuführen

Wenn **Benachrichtigungen für alle Makros** aktiviert ist, erscheint beim Start von Outlook die Meldung aus Bild 2 – vorausgesetzt, es liegt keine der übrigen Ursachen für eine Fehlfunktion von **Application_Startup** vor.

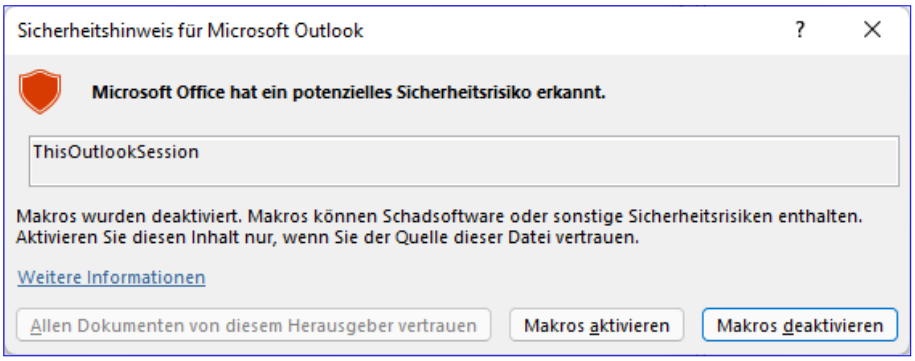
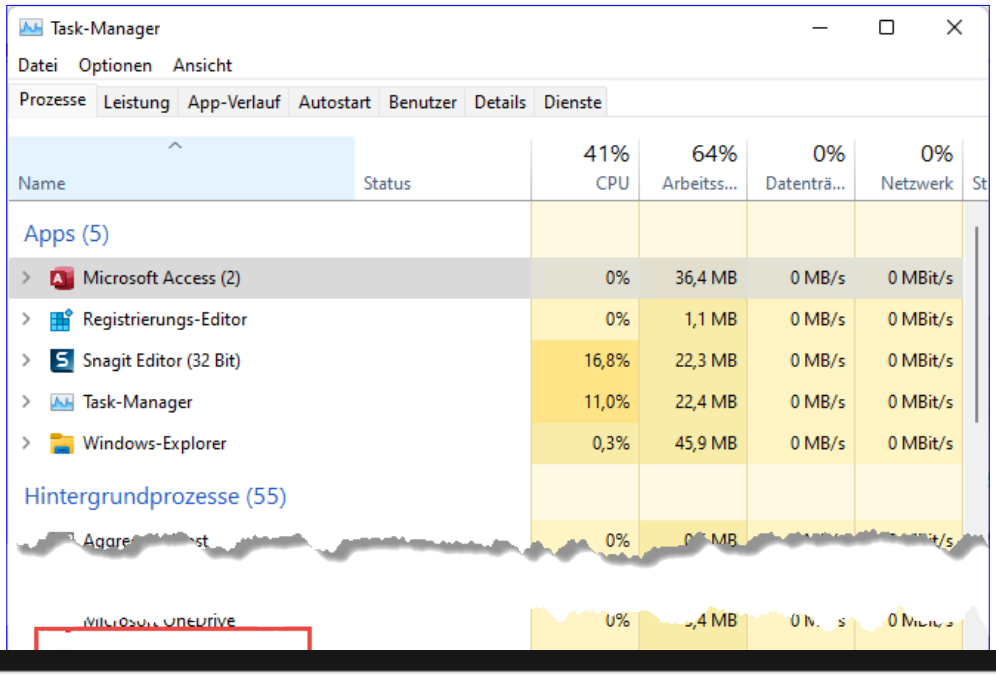


Bild 2: Meldung, wenn Benachrichtigungen für Makros aktiviert sind

Ist **Alle Makros aktivieren** selektiert, sollte **Application_Startup** beim Start ausgeführt werden. Wenn nicht, prüfe, ob Du es wie oben beschrieben von Hand aufrufen kannst, was nun funktionieren sollte. Dann ist einer der folgenden beiden Gründe für das Problem verantwortlich.



Outlook ist bereits geöffnet

Es kann auch sein, dass Outlook bereits durch eine andere Anwendung geöffnet wurde und diese

Outlook: E-Mails nach Eingang verarbeiten

Die verschiedenen Klassen von Outlook bieten eine ganze Reihe von Ereignissen an. Diese werden durch unterschiedliche Aktionen ausgelöst. Eine dieser Aktion ist das Eintreffen einer oder mehrerer neuer E-Mails. Dieses Ereignis mit einer geeigneten Ereignisprozedur abzufangen ist ein sinnvolles Beispiel für die Beschreibung der Programmierung von Ereignissen unter Outlook. Mit E-Mails kann man nach dem Eingang eine Menge anstellen – diese löschen, in einen anderen Ordner verschieben, die Message-Datei im Dateisystem sichern oder auch die Inhalte auslesen, um diese etwa in eine Datenbank zu schreiben. In diesem Artikel beschreiben wir erst einmal allgemein, wie wir überhaupt mit VBA-Code auf den Eingang einer E-Mail reagieren können.

Outlook bietet alles, was man braucht – E-Mails werden abgerufen und landen im Posteingangs-Ordner. Man kann verschiedene Regeln aufstellen, nach denen die eingehenden E-Mails automatisch nach den gewünschten Kriterien in verschiedene Ordner verschoben werden, und wenn es auch der Ordner **Gelöschte Objekte** oder der Spam-Ordner ist. Und natürlich können wir die E-Mails auch manuell in andere Ordner verschieben oder auch Aufgaben auf Basis einer E-Mail erstellen. Noch schöner wäre es allerdings, wenn wir die gewünschten Schritte individuell programmieren könnten.

Welches Objekt und welches Ereignis?

Dazu müssen wir erst einmal wissen, wie wir eine Ereignisprozedur programmieren, mit der wir auf den Eingang einer E-Mail reagieren können. Der erste Schritt ist dabei, herauszufinden, welches Ereignis welcher Klasse wir überhaupt nutzen können. In weiteren Artikel zur Programmierung von Outlook per VBA haben wir bereits gesehen, dass es unterschiedliche Klassen gibt, mit denen wir die verschiedenen Elemente der Benutzeroberfläche referenzieren können. Mit dem **Application**-Objekt greifen wir auf die Anwendung selbst zu, mit dem **Explorer**-Objekt auf die verschiedenen Ansichten von Outlook und mit dem **Inspector**-Objekt auf von Outlook aus geöffnete Fenster beispielsweise zur Anzeige einer E-Mail. Außerdem gibt es noch Klassen, mit denen wir die einzelnen Ordner oder die darin enthaltenen Elemente wie E-Mails, Kontakte, Termine oder Aufgaben referenzieren können.

Wenn wir neu im Thema sind und nicht genau wissen, welche Klasse welche Ereignisse bietet, hilft uns ein Blick in den Objektkatalog. In unserem Beispiel, wo wir auf den Eingang einer E-Mail reagieren wollen, gibt es unterschiedliche Ideen, welches Klasse ein passendes Ereignis bietet. Vielleicht gibt es ein Ereignis, das ausgelöst wird, wenn einem Ordner ein neues Element hinzufügt wird? Oder wird ein solches Ereignis von einer anderen Klasse zur Verfügung gestellt?

Haben wir Outlook geöffnet, starten wir mit **Alt + F11** den VBA-Editor. Hier zeigen wir mit **F2** den Objektkatalog an. Oben im Fenster wählen wir statt dem Eintrag **Alle Bibliotheken** die Bibliothek **Outlook** aus, direkt darunter geben wir als Suchbegriff **Mail** ein.

In der Liste der Suchergebnisse suchen wir nun nach Elementen, die durch ein Blitz-Icon markiert werden und stoßen schnell auf die beiden Ereignisse **NewMail** und **NewMailEx** (siehe Bild 1). Klicken wir **NewMail** an, sehen wir die Beschreibung dieses Ereignisses im unteren Bereich. Außerdem sehen wir, dass es sich hierbei um ein Ereignis der Klasse **Outlook.Application** handelt.

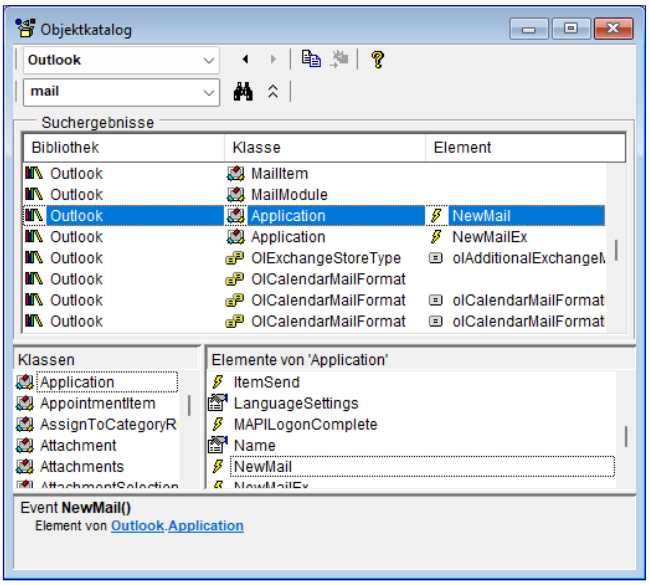


Bild 1: Ermitteln von Ereignissen für E-Mails

Das Gleiche gilt für das Ereignis **NewMailEx**. Klicken wir dieses im Objektkatalog an, finden wir schnell heraus, dass es im Gegensatz zum Ereignis **NewMail** noch einen Parameter namens **EntryIDCollection** mit dem Datentyp **String** liefert (siehe Bild 2).

Neben diesen beiden offensichtlichen Möglichkeiten gibt es noch ein weiteres Ereignis, das durch das Empfangen einer E-Mail ausgelöst wird. Dabei handelt es sich um das Ereignis **ItemAdd** der **Items**-Auflistung eines **Folder**-Objekts, das ausgelöst wird, wenn einem Ordner ein Element hinzugefügt wird. Um dieses zu nutzen, um auf den Eingang neuer E-Mails zu reagieren, müssen wir den richtigen Ordner damit ausstatten, in der Regel den Ordner **Posteingang**. Wie das gelingt, zeigen wir gegen Ende des Artikels.

Ereignisse implementieren

Um ein Ereignis in Form einer Ereignisprozedur zu implementieren, benötigen wir zunächst ein Klassenmodul. Nur in einem

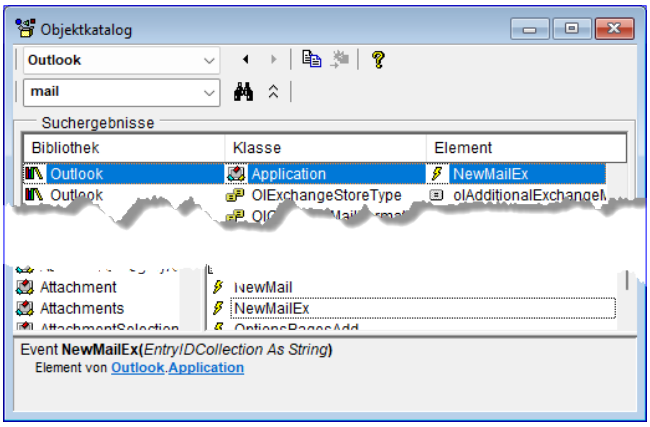


Bild 2: Das Ereignis **NewMailEx** im Objektkatalog

solchen können wir die Klasse, welche das Ereignis bereitstellt, mit dem Schlüsselwort **WithEvents** deklarieren, und ohne dieses Schlüsselwort können wir die Ereignisprozedur nicht implementieren.

Zum Glück haben wir mit dem Klassenmodul **ThisOutlookSession** immer ein Klassenmodul zur Hand, welches wir noch nicht einmal initialisieren müssen, das es standardmäßig bereits initialisiert ist. Noch besser: Die Klasse **Application** steht hier immer bereits sodass wir diese noch nicht einmal deklarieren müssen. Um ein Ereignis für **Outlook.Application** zu implementieren, müssen wir nur zwei Schritte erledigen:

- Wir wählen im linken Kombinationsfeld des Codefensters des Moduls **ThisOutlookSession** den Eintrag **Application** aus. Dies legt automatisch das

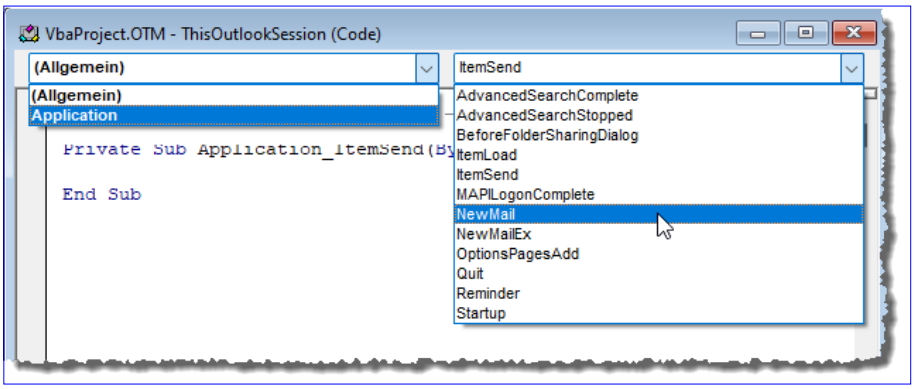


Bild 3: Das Ereignis **NewMailEx** implementieren

Standardereignis für diese Klasse an, in diesem Fall **Application_ItemSend**.

- Dann wählen wir im rechten Kombinationsfeld das Ereignis aus, das wir eigentlich implementieren wollen, nämlich zunächst einmal **NewMail** (siehe Bild 3)

Anschließend entfernen wir zunächst die Prozedur **Application_ItemSend** wieder. Außerdem wählen wir im rechten Kombinationsfeld noch den Eintrag **NewMailEx** aus, um auch die zweite für uns interessante Ereignisprozedur hinzuzufügen.

Prüfen, ob die Ereignisse ausgelöst werden

Jetzt wollen wir herausfinden, ob die Ereignisprozeduren automatisch aufgerufen werden, wenn wir E-Mails mit Outlook abrufen. Dazu fügen wir den Prozeduren einfach jeweils eine Anweisung hinzu, sodass diese anschließend wie folgt aussehen:

```
Private Sub Application_NewMail()
    Debug.Print "NewMail"
End Sub

Private Sub Application_NewMailEx(_
        ByVal EntryIDCollection As String)
    Debug.Print "NewMailEx", EntryIDCollection
End Sub
```

Rufen wir nun E-Mails ab, erhalten wir im Direktbereich beispielsweise beim Abrufen von zwei E-Mails eine Ausgabe wie in Bild 4.

Das Ereignis **NewMail** liefert keinen Parameter, somit können wir damit nur herausfinden, ob eine E-Mail abgerufen wurde und nicht, welche.

Das Ereignis **NewMailEx** liefert immerhin einen Parameter namens **EntryIDCollection**. Dieser enthielt für frühere Versionen der Bibliothek **Microsoft Outlook**

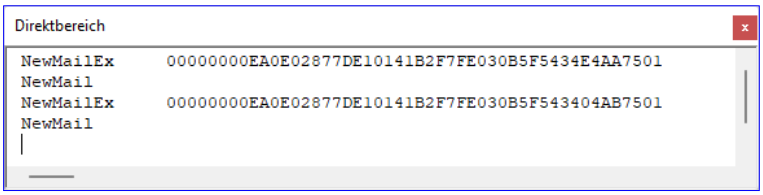


Bild 4: Ausgabe im Direktbereich bei Maileingang

x.0 Object Library eine Komma-separierte Liste der **EntryIDs** aller eingegangenen E-Mails. In der aktuellen Version wird das Ereignis für jede E-Mail einmal ausgelesen und liefert dementsprechend immer nur eine **EntryID**. Eine **EntryID** ist ein eindeutiger Bezeichner für die Outlook-Elemente, mit dem wir auf das jeweilige Element zugreifen können.

Zugriff auf die eingegangenen E-Mails

Nun schauen wir uns an, wie wir die mit der **NewMailEx**-Ereignisprozedur eingelesenen **EntryID**-Werte nutzen können, um auf die eingegangenen E-Mails zuzugreifen. Wir programmieren die Lösung so, dass sie auch mit früheren Versionen der Outlook-VBA-Bibliothek kompatibel ist. Wir gehen also davon aus, dass der Parameter **EntryIDCollection** auch mehrere, durch Kommata getrennte Einträge enthalten kann.

Dazu erweitern wir die Prozedur wie in Listing 1. Wir deklarieren ein **String**-Array, um die mit **EntryIDCollection** gelieferten EntryIDs aufzunehmen. Außerdem nutzen wir zum Referenzieren der Objekte zunächst die Variable **objItem**, die wir als **Object** deklarieren, und nur wenn es sich bei dem eingegangenen Element um eine E-Mail handelt, weisen wir es der Variablen **objMailItem** des Typs **MailItem** zu.

Den Inhalt von **EntryIDCollection**, der gegebenenfalls aus mehreren durch Kommata getrennten Einträgen besteht, teilen wir mit der **Split**-Funktion auf und schreiben die einzelnen Elemente in das Array **strEntryIDs**.

Dann durchlaufen wir die Elemente des Arrays in einer **For...Next**-Schleife und ermitteln mit der Funk-

```
Private Sub Application_NewMailEx(ByVal EntryIDCollection As String)
    Dim strEntryIDs() As String
    Dim objItem As Object
    Dim objMailItem As MailItem
    Dim i As Integer
    strEntryIDs = Split(EntryIDCollection, ",")
    For i = LBound(strEntryIDs) To UBound(strEntryIDs)
        Set objItem = Application.Session.GetItemFromID(strEntryIDs(i))
        Select Case objItem.Class
            Case olMail
                Set objMailItem = objItem
                With objMailItem
                    Debug.Print .Subject, .ReceivedTime
                End With
        End Select
    Next i
End Sub
```

Listing 1: Ausgabe von E-Mail-Details beim Maileingang

tion **GetItemFromID** der aktuellen Session einen Verweis auf die eingegangenen Elemente. Für das jeweils in **objItem** befindliche Element prüfen wir anhand der **Class**-Eigenschaft, ob es sich dabei um ein **MailItem**-objekt handelt. In diesem Fall weisen wir das Objekt der Variablen **objMailItem** zu, was den Zugriff auf die Elemente der **MailItem**-Klasse per IntelliSense erlaubt.

Damit geben wir für jedes Element den Inhalt der Ei-

```
    .Categories = "Archiviert"
    .Save
    .SaveAs "C:\...\Mails\" & strEntryIDs(i) & ".msg"
End With
```

Die erste Zeile weist der eingegangenen E-Mail die Kategorie **Gespeichert** zu (wenn diese noch nicht existiert, wird sie automatisch angelegt). Die zweite speichert die Änderung an der E-Mail in Outlook. Die

Outlook: E-Mail per Drag and Drop nach Access

Wenn Du eine Kundendatenbank mit Access programmiert hast und die Kommunikation per E-Mail über Outlook läuft, möchtest Du vielleicht einem Kundendatensatz die E-Mails dieses Kunden zuweisen. Dafür gibt es verschiedene Möglichkeiten. In diesem Artikel schauen wir uns eine an, bei der Du die E-Mail per Drag and Drop auf einen Bereich in einem Access-Formular ziehst. Dort verarbeiten wir die E-Mail und speichern bestimmte Daten in einer Tabelle, damit die E-Mail bei Bedarf vom Kunden-Formular aus wieder in Outlook angezeigt werden kann. Wer sich schon mit dem Thema beschäftigt hat, weiß, dass man E-Mails eigentlich nicht nach Access ziehen kann. Deshalb umschiffen wir dieses Problem mit einem kleinen Trick.

Beschreibung der Lösung

Die Lösung soll folgenden Vorgang abbilden: In einer Access-Datenbank sind die Daten von Kunden gespeichert, die in einem Formular angezeigt werden können. Über Outlook kommunizieren wir per E-Mail mit diesem Kunden. Die E-Mails wollen wir nicht nur in Outlook sehen, sondern wir wollen auch in der Access-Kundendatenbank die E-Mails des Kunden sehen und diese gegebenenfalls in Outlook anzeigen.

Dabei wollen wir aber keineswegs alle E-Mails von diesem Kunden und an diesen Kunden automatisch in der Access-Datenbank speichern, sondern der Benutzer soll entscheiden, welche E-Mails des Kunden in der Datenbank gespeichert werden sollen.

Dazu benötigen wir eine einfache Möglichkeit, um die E-Mail zur Kundentabelle hinzuzufügen. Eine der intuitivsten Möglichkeiten, um Elemente von einer Anwendung zur anderen zu bewegen, ist Drag and Drop – also das Ziehen des Elements, in diesem Fall der E-Mail, mit der Maus.

Die E-Mail soll nach dem Ziehen in einen bestimmten Bereich im Formular mit den Kundendaten in einem Unterformular angezeigt werden. Im Hintergrund nutzen wir dazu neben der Kundentabelle eine Tabelle, die alle relevanten Daten zu den E-Mails des Kunden speichert. Darunter befindet sich auch der eindeutige Schlüssel dieser E-Mail in Outlook, mit dem wir diese in Outlook von der Kundendatenbank aus öffnen können.

Aufbau des Datenmodells

Das Datenmodell zu unserer Lösung umfasst lediglich zwei Tabellen:

- **tblKunden**: Diese Tabelle enthält rudimentäre Kundendaten – eben so viele, dass wir diese in einem Kundenformular darstellen können.

- **tblMailItems**: Diese Tabelle enthält ein Fremdschlüsselfeld zur Tabelle **tblKunden**, womit wir die

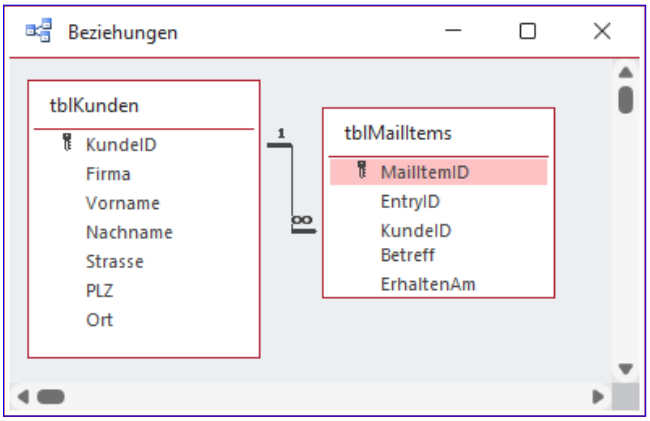


Bild 1: Die Tabellen der Datenbank in der Übersicht

E-Mail-Datensätze den Kundendatensätzen zuordnen können. Außerdem enthält sie ein Feld für den Betreff der E-Mail sowie für den eindeutigen Schlüssel der E-Mail.

Das Datenmodell sieht im Beziehungen-Fenster wie in Bild 1 aus.

Tabellen anlegen

Um die Tabellen anzulegen, fügst Du nach dem Erstellen einer neuen, leeren Access-Datenbank mit dem Ribbon-Befehl **Erstellen|Tabellen|Tabellenentwurf** jeweils eine neue Tabelle in der Entwurfsansicht an.

Für die Tabelle **tblKunden** fügst Du die Felder mit den Feldnamen und den Felddatentypen wie in Bild 2 hinzu.

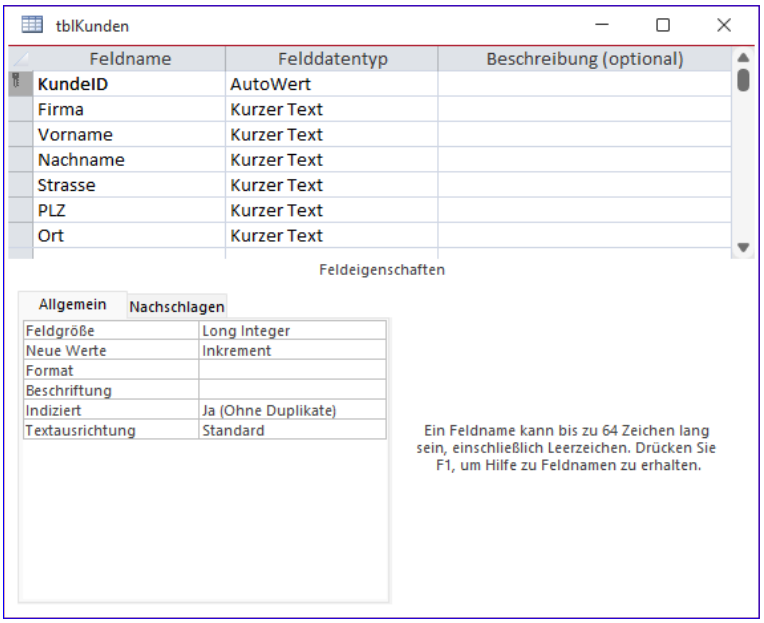


Bild 2: Die Tabelle **tblKunden** in der Entwurfsansicht

Die Tabelle **tblMailItems** soll die Felder wie in Bild 3 enthalten. Für das Feld **EntryID** haben wir einen eindeutigen Index definiert, damit jede E-Mail nur einmal zu dieser Tabelle hinzugefügt werden kann. Um das Feld **KundeID** mit der Tabelle **tblKunden** zu verknüpfen, verwenden wir das Beziehungsfenster. Hier fügen wir die beiden Tabellen wie in der obigen Abbildung hinzu und ziehen dann das Feld **KundeID** der einen Tabelle auf das gleichnamige Feld der anderen Tabelle.

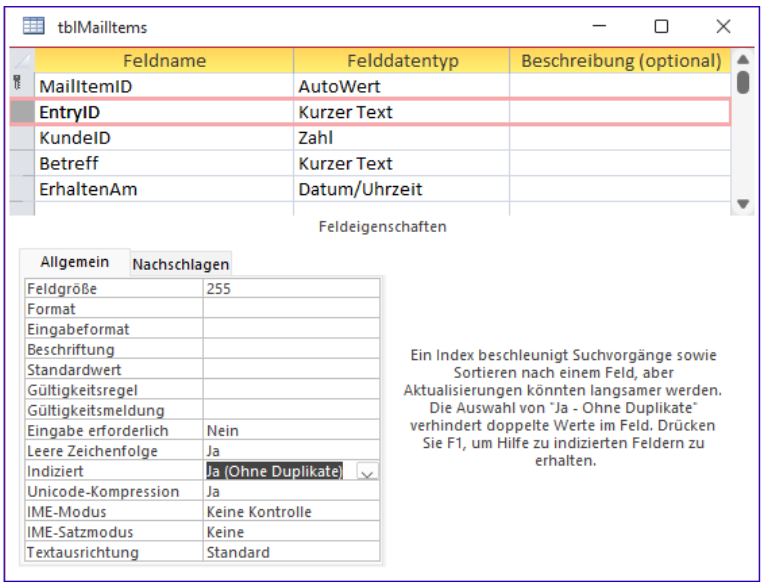


Bild 3: Die Tabelle **tblMailItems** in der Entwurfsansicht

Im nun erscheinenden Dialog **Beziehungen bearbeiten** kannst Du noch die referentielle Integrität festlegen (siehe Bild 4).

Formulare erstellen

Das einzige Formular der Anwendung soll jeweils einen Kundendatensatz anzeigen sowie dessen E-Mails, die ihm per Drag and Drop zugewiesen wurden. Dazu benötigen wir ein Haupt- und ein Unterformular.

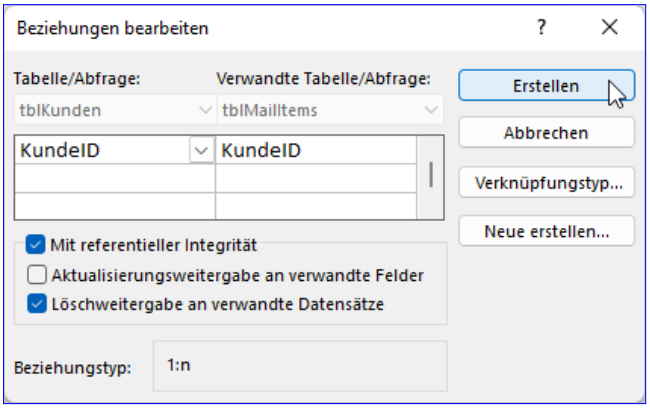


Bild 4: Einstellungen für die angelegte Beziehung

Unterformular zur Anzeige der E-Mails anlegen

Das Unterformular soll die Daten der Tabelle **tblMailItems** anzeigen. Dazu fügen wir mit dem Ribbonbefehl **Erstellen|Formulare|Formularentwurf** ein neues Formular in der Entwurfsansicht hinzu. Für seine Eigenschaft **Datensatzquelle** legen wir die Tabelle **tblMailItems** fest. Danach können wir über **Formularentwurf|Tools|Vorhandene Felder hinzufügen** die Felder **Betreff** und **ErhaltenAm** in den Formularentwurf ziehen. Damit die Daten in der Datenblattansicht angezeigt werden, stellen wir die Eigenschaft **Standardansicht** auf **Datenblatt** ein. Vor dem Schließen speichern wir das Formular unter dem Namen **sfmKundenEMails**.

Hauptformular für die Kundendaten anlegen

Anschließend legen wir ein weiteres Formular an, dem wir die Tabelle **tblKunden** als **Datensatzquelle** hinzufügen. Aus dieser ziehen wir alle Felder außer dem Primärschlüsselfeld **KundeID** in den Detailbereich des Formularentwurfs und ordnen diese nach Wunsch an.

Anschließend ziehen wir das bereits geschlossene Formular **sfmKundenEMails** aus dem Navigationsbereich in den Formularentwurf.

Dieses arrangieren wir unter den übrigen Steuerelementen. Außerdem fügen wir darunter noch eine Schaltfläche hinzu, mit der wir die aktuell im Unterformular markierte E-Mail öffnen wollen. Das Zwischenergebnis sieht wie in Bild 5 aus.

Dadurch, dass die in den beiden Formularen angezeigten Datensatzquellen über das Fremdschlüsselfeld **KundeID** der Tabelle **tblMailItems** miteinander verknüpft sind, erstellt Access automatisch auch eine Verknüpfung zwischen Haupt- und Unterformular.

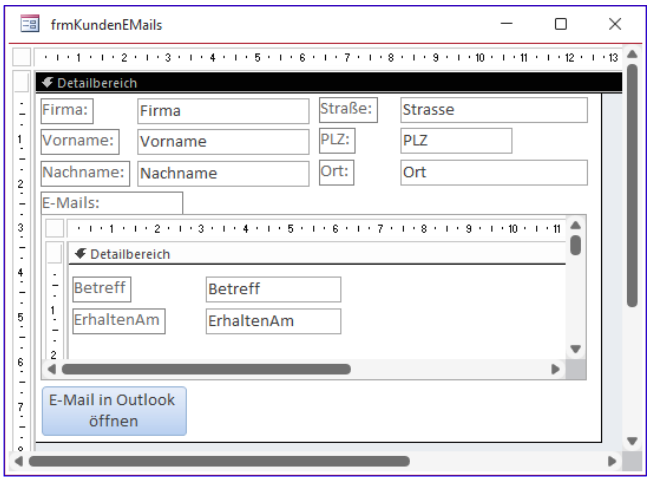


Bild 5: Haupt- und Unterformular in der Entwurfsansicht

Diese sorgt dafür, dass das Unterformular nur solche Datensätze der Tabelle **tblMailItems** anzeigt, die mit dem aktuell im Hauptformular angezeigten Datensatz der Tabelle **tblKunden** verknüpft sind. Achtung: Dieses rudimentäre Beispiel ist nicht mit Funktionen ausgestattet, die verhindern, dass der Benutzer im Unterformular Daten anlegt, während das Hauptformular noch keinen Datensatz anzeigt. Solche Datensätze werden später nicht mehr angezeigt, weil sie mit keinem Kunden verknüpft sind.

Ziel für Drag and Drop hinzufügen

Um E-Mails per Drag and Drop zum Formular hinzuzufügen, müssen wir noch einen Zielbereich definie-

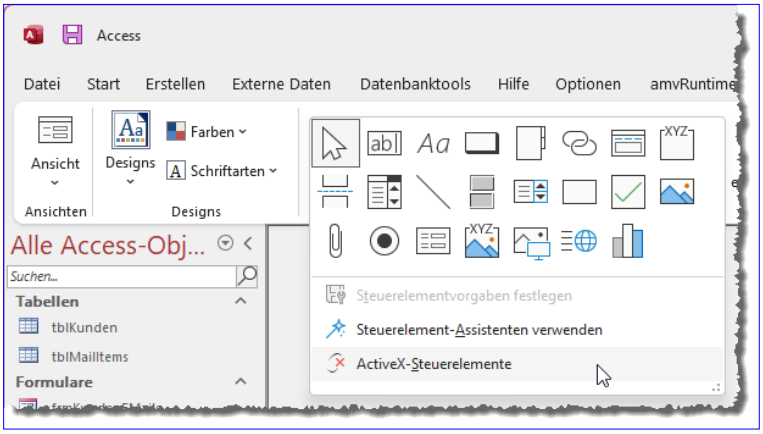


Bild 6: Hinzufügen eines ActiveX-Steuerelements

ren. Die eingebauten Formular- und Steuerelementklassen bieten leider keine Drag and Drop-Funktionalität, daher müssen wir zu einer Alternative greifen und etwas improvisieren. Also fügen wir dem Entwurf des Formulars ein **ListView**-Steuerelement hinzu.

Prinzipiell bieten zwar auch andere Steuerelemente die Möglichkeit, auf Drag and Drop-Vorgänge zu reagieren, aber mit dem **ListView**-Steuerelement können wir noch eine Bilddatei als Icon integrieren, die wir zur Kenntlichmachung der Funktion dieses Steuerelements nutzen können. Dazu sind jedoch einige weitere Schritte nötig.

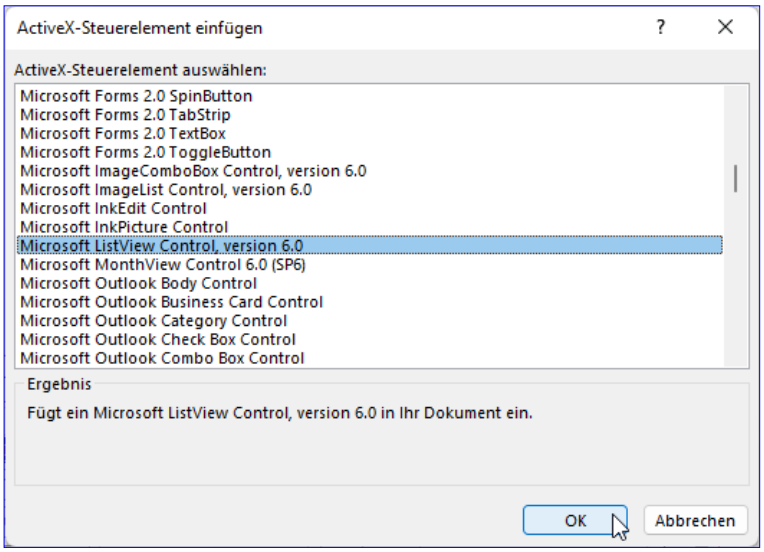


Bild 7: Auswahl des **ListView**-Steuerelements

ListView als Drag and Drop-Ziel hinzufügen

Der erste Schritt ist das Hinzufügen des **ListView**-Steuerelements. Dazu wechseln wir in die Entwurfsansicht des Formulars und betätigen im Ribbon unter **Formularentwurf|Steuerelemente** die Schaltfläche zum Anzeigen weiterer Steuerelemente, wo wir den Eintrag **ActiveX-Steuerelemente** finden (siehe Bild 6).

Dies öffnet den Dialog **ActiveX-Steuerelement einfügen**, wo wir den Eintrag **Microsoft ListView Control, version 6.0** selektieren und auf **OK** klicken (siehe Bild 7).

Dies fügt das Steuerelement direkt zum Formular hinzu, sodass wir es nur noch platzieren und die Größe anpassen müssen. Außerdem stellen wir für die Eigenschaft **Name** den Wert **ctlMaildrop** ein. Danach erhalten wir den Stand aus Bild 8.

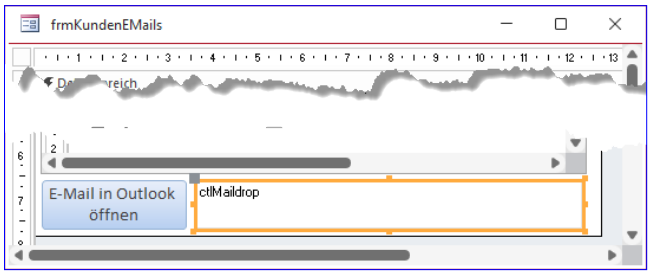


Bild 8: Platzieren und Benennen des **ListView**-Steuerelements

Wechseln wir in die Formularansicht, sehen wir allerdings nur ein leeres, weißes Steuerelement. Selbst für interne Zwecke wäre mir das zu spartanisch, und wenn andere Benutzer damit arbeiten sollen, müssen diese schon wissen, was mit dem Steuerelement anzufangen ist

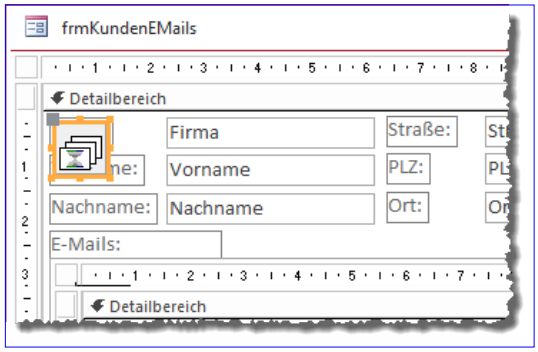


Bild 9: Das **ImageList**-Steuerelement

Wir könnten nun eine Beschriftung über dem Steuerelement anlegen, die auf die Funktion hinweist, aber wir wollen es gleich richtig machen: Das Steuerelement selbst soll per Icon plus Text auf seine Funktion hinweisen und wenn der Benutzer mit einem gezogenen Element das Steuerelement überfährt, sollen sich Icon und Text entsprechend verändern.

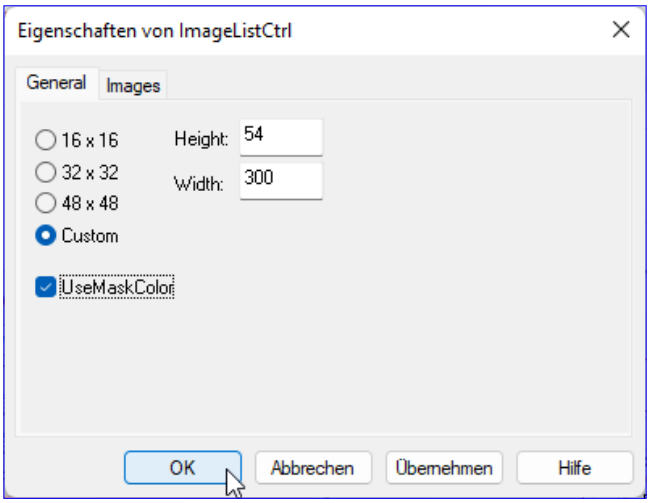


Bild 10: Einstellen der Abmessungen für die anzuzeigenden Icons

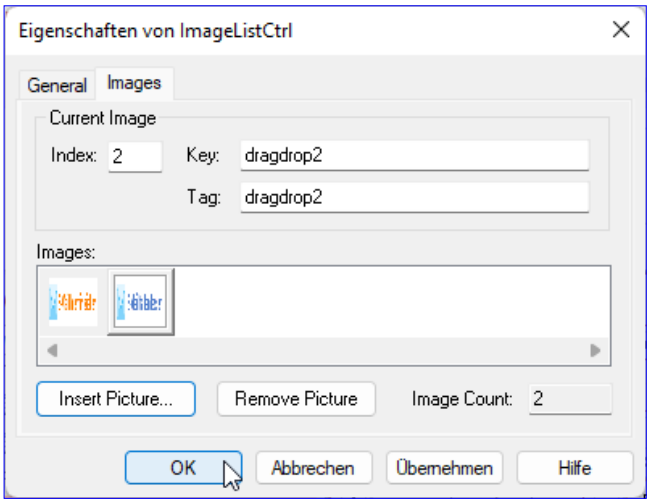


Bild 11: Hinzufügen und Benennen der Bilder

Icons für das Steuerelement bereitstellen

Wer bereits mit dem **TreeView**- oder dem **ListView**-Steuerelement gearbeitet und dort Icons verwendet hat, der weiß, dass ohne ein weiteres Steuerelement nichts geht: Wir benötigen nämlich noch ein **ImageList**-Steuerelement, um die Icons für das **ListView**-Steuerelement verfügbar zu machen.

Dieses fügen wir auf dem gleichen Wege wie das **ListView**-Steuerelement zum Formular hinzu und nennen es **ctlImageList**. In der Liste der ActiveX-Steuerelemente heißt dieses Steuerelement **Microsoft ImageList Control, version 6.0**. Nach dem Einfügen müssen

Diese Icons fügen wir nun dem **ImageList**-Steuerelement hinzu. Dazu klicken wir dieses zunächst doppelt an, um seinen Eigenschaften-Dialog zu öffnen. Hier tragen wir die Größe der soeben erstellten Bilddateien ein, in diesem Fall mit einer Höhe von **50** und einer Breite von **300** (siehe Bild 10). Achtung: Die Höhe kann nur angepasst werden, solange das Steuerelement noch keine Images enthält.

Auf der zweiten Registerseite des Eigenschaften-Dialogs klicken wir auf die Schaltfläche **Insert Picture...**, wählen das erste Bild aus und stellen die Eigenschaften

Einfache Excel-Formulare erstellen

Excel kann nicht nur einfach Daten in Tabellenform anzeigen. Es bietet auch verschiedene Möglichkeiten, um Daten auszuwerten, beispielsweise in Diagrammen, oder auch Möglichkeiten zur komfortableren Eingabe von Daten. Wer zum Beispiel sonst mit Access arbeitet, weiß, dass man dem Benutzer dort üblicherweise Formulare zur Verfügung stellt, um die Daten einzugeben, zu betrachten und zu verwalten. Warum also sollte man den Anwender in Excel mit riesigen »Tapeten« von Daten quälen, statt ihm jeweils den Inhalt einer Zeile der Tabelle zum Betrachten und zum Bearbeiten anzubieten? Dieser Artikel zeigt eine sehr einfache Möglichkeit, um dem Benutzer ein ergonomischeres Erlebnis zu bieten.

Es gibt noch wesentlich ausgefeiltere und flexiblere Möglichkeiten, in Excel Formulare für verschiedene Zwecke anzuzeigen.

Bevor wir uns diese in weiteren Artikeln ansehen, wollen wir uns jedoch eine eher versteckte Variante anschauen.

Tabelle aus Bereich erstellen

Dazu benötigen wir zunächst eine Tabelle. Dabei zunächst zur Begriffsklärung der Hinweis, dass ein Arbeitsblatt (englisch **Worksheet**) nicht gleichzusetzen ist mit einer Tabelle – auch wenn der Registerreiter eines Arbeitsblattes unter dem Arbeitsblatt standardmäßig den Namen **Tabelle** enthält.

Wir gehen von einem einfachen Worksheet wie in Bild 1 aus. Hier markieren wir die Spaltenüberschriften und die Daten, die in dem noch zu erstellenden Formular angezeigt werden sollen. Dann betätigen wir den Ribbon-Befehl **Einfügen|Tabellen|Tabelle**.

Dies zeigt den Dialog **Tabelle erstellen** an. Hier können wir den Bereich für die Tabelle nochmal korrigieren und angeben, ob die Tabelle bereits Spaltenüberschriften enthält (siehe Bild 2).

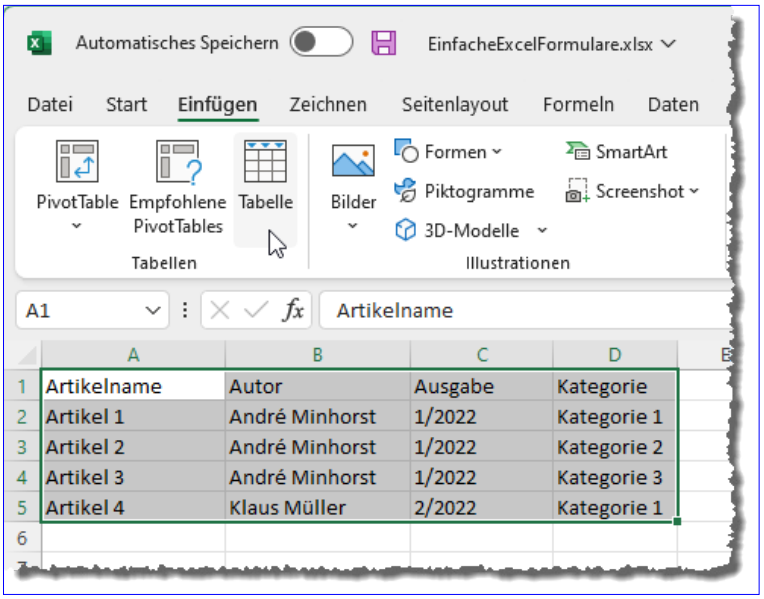


Bild 1: Markieren des Bereichs für die zu erstellende Tabelle

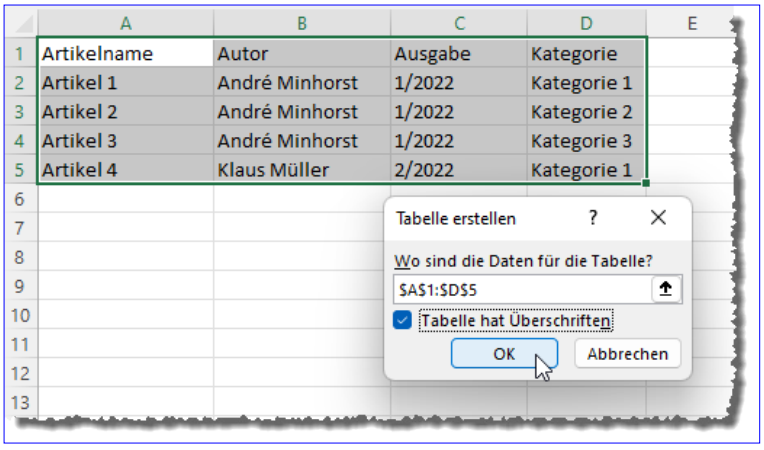
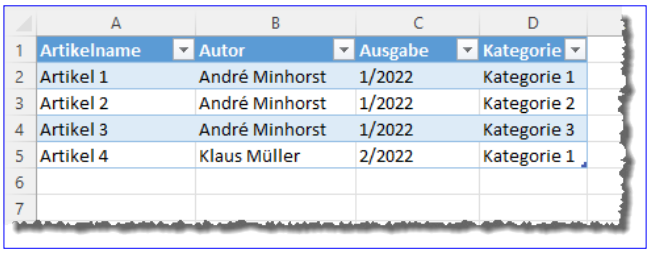


Bild 2: Erstellen einer Tabelle aus einem einfachen Worksheet



	A	B	C	D
1	Artikelname	Autor	Ausgabe	Kategorie
2	Artikel 1	André Minhorst	1/2022	Kategorie 1
3	Artikel 2	André Minhorst	1/2022	Kategorie 2
4	Artikel 3	André Minhorst	1/2022	Kategorie 3
5	Artikel 4	Klaus Müller	2/2022	Kategorie 1
6				
7				

Bild 3: Tabelle mit zusätzlichen Funktionen

Anschließend erscheint der markierte und in eine Tabelle umgewandelte Bereich mit neuen Formatierungen. Außerdem zeigen die Zellen mit den Spaltenüberschriften jeweils eine Schaltfläche mit einem Pfeil nach unten an (siehe Bild 3).

Klicken wir eine dieser Schaltflächen an, erscheinen Sortier- und Filteroptionen, wie man sie sonst von der Datenblattansicht von Access kennt (siehe Bild 4). Damit können wir die Anzeige der Daten der Tabelle auf verschiedene Arten beeinflussen. Das allein ist schon praktisch, aber letztlich nur die Grundlage für das, was wir eigentlich in diesem Artikel erreichen wollen – nämlich die Anzeige dieser Daten in einem Formular.

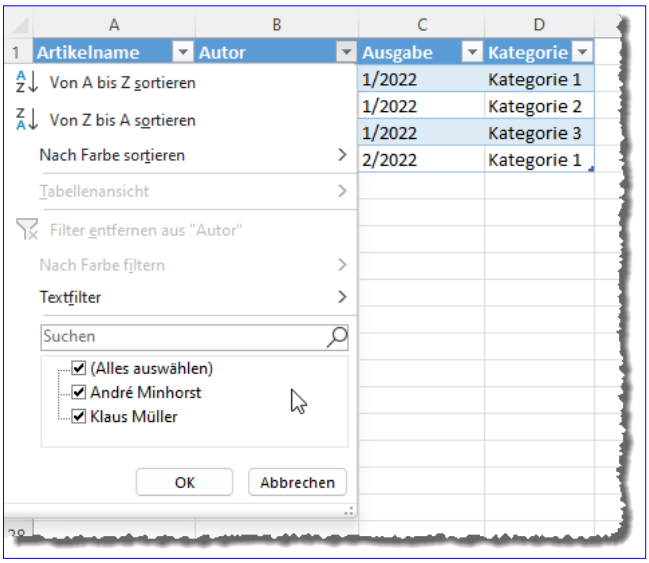


Bild 4: Sortieren und Filtern nach verschiedenen Kriterien

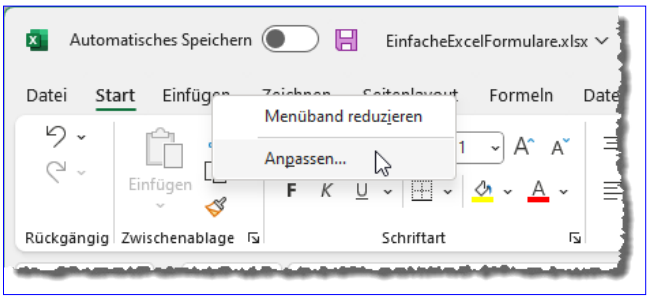


Bild 5: Anzeigen des Anpassen-Bereichs der Excel-Optionen

Befehl zum Anzeigen eines Formulars aktivieren

Die nachfolgend vorgestellte Funktion ist nicht bekannst Du schon einmal zum Eintrag **Maske...** na-

Buttons in Excel

In anderen Artikeln zeigen wir, wie Du Makros in Excel aufzeichnest oder wie Du per VBA-Prozedur auf die verschiedenen Elemente einer Excel-Arbeitsmappe zugreifen kannst. Diese Makros sind durchaus praktisch, aber der Nutzen reduziert sich doch, wenn man zum Ausführen beziehungsweise aufrufen immer den VBA-Editor anzeigen muss. Also zeigen wir im vorliegenden Artikel, wie Du einem Arbeitsblatt in Excel eine Schaltfläche hinzufügst, mit der Du die für dieses Arbeitsblatt vorgesehenen VBA-Prozeduren beziehungsweise Makros viel schneller und komfortabler aufrufen kannst.

Wie wir im Artikel **VBA: Makros aufzeichnen** (**www.vbentwickler.de/324**) gezeigt haben, kannst Du einfache Abläufe in Excel mit der Aufzeichnen-Funktion als Makro aufzeichnen.

Im Bereich **Entwicklertools** des Ribbons, in dem sich auf die Aufzeichnen-Funktion befindet, können wir über den Befehl **Makros** den gleichnamigen Dialog öffnen, der dann alle aktuell erreichbaren Makros anzeigt (siehe Bild 1). Hier kannst Du die Schaltfläche **Ausführen** anklicken, um das aktuell in der Liste markierte Makro zu starten.

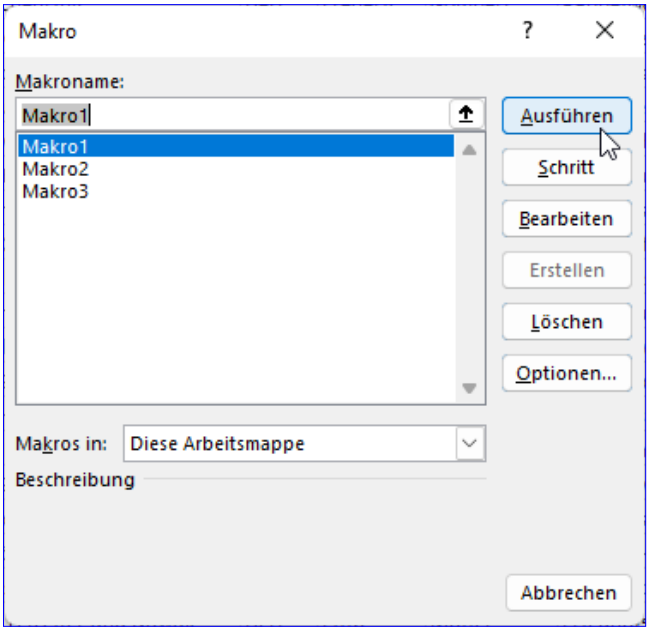


Bild 1: Aufruf eines Makros

Das ist allerdings ohne weitere Anpassungen schon die einfachste Methode, um ein Makro zu starten. Wenn man bedenkt, dass ein Makro eigentlich das Ausführen verschiedener Aktionen vereinfachen oder beschleunigen soll, ist das recht kompliziert.

Besser wäre es, wenn man direkt an der Stelle, an der die Aktion gefragt ist, eine Möglichkeit zu ihrem Aufruf platzieren könnte. Dazu eignet sich beispielsweise eine Schaltfläche. Wie wir diese anlegen und damit das angegebene Makro aufrufen, zeigen wir in den folgenden Abschnitten.

Schaltfläche hinzufügen

Die erste Voraussetzung zum Hinzufügen einer Schaltfläche ist bereits gegeben, wenn Du über das Ribbon Makros aufzeichnen kannst. Dann hast Du bereits die Ribbon-Registerkarte **Entwicklertools** eingeblendet. Wenn Du nun die Arbeitsmappe geöffnet hast, der Du die Schaltfläche hinzufügen möchtest, kannst Du den Ribbon-Eintrag **Entwicklertools|Steuerelemente|Einfügen** aufrufen und findest die Liste der verfügbaren Steuerelemente vor. Diese liefert zwei Bereiche, wobei wir als Erstes die Schaltfläche aus dem Bereich **Formularsteuerelemente** anklicken (siehe Bild 2).

Danach können wir mit der Maus im Arbeitsblatt einen Rahmen aufziehen, der die Größe der zu erstellenden Schaltfläche angibt (siehe Bild 3).

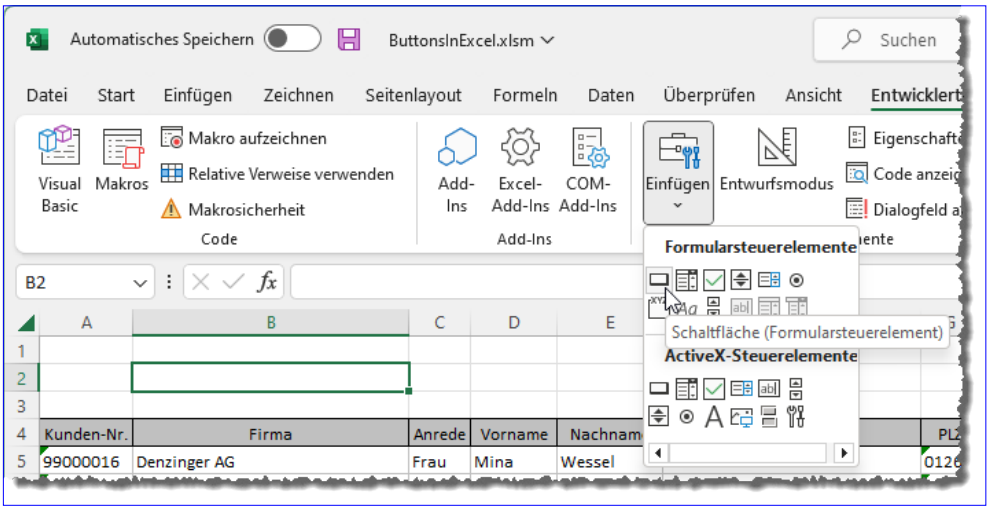


Bild 2: Hinzufügen einer Schaltlfäche

Makro zur Schaltfläche zuweisen

Nun erscheint ein weiterer Dialog namens **Makro zuweisen**, der wie in Bild 4 aussieht. Der Dialog schlägt zunächst den Namen für ein neu anzulegendes oder aufzuzeichnendes Makro vor, hier **Schaltfläche1_Klicken**. Möchtest Du dieses verwenden, kannst Du noch den Namen anpassen und dann eine der Schaltflächen **Neu** oder **Aufzeichnen...** anklicken, um das neue Makro direkt im VBA-Editor anzulegen und zu bearbeiten oder ein neues Makro mit dem angegebenen Namen aufzuzeichnen. Klickst Du einfach auf **OK**, wird zwar ein Makro namens **[Arbeitsmappenname].[Makroname]** zugewiesen, aber nicht erstellt, was in der Folge zu einem Fehler führt.

Alternativ kannst Du eines der vorhandenen Makros auswählen, dann ändert sich die Beschriftung der oberen Schaltfläche in **Bearbeiten**. Achtung: Klickst Du hier auf Bearbeiten, öffnet sich zwar der VBA-Editor mit der jeweiligen Prozedur, aber diese wird nicht als Makro zu der Schaltfläche zugewiesen!

Wenn Du der Schaltfläche also ein vorhandenes Makro zuweisen möchtest, wählst Du es einfach nur aus und klickst auf **OK**.

Zu Beispielzwecken passen wir nun einfach den Namen auf **btnMeldung_Click** und klicken auf die Schaltfläche **Neu**. Dann ergänzen wir die nun im VBA-Editor angelegte Prozedur wie folgt um den Aufruf einer Meldung:

```
Sub btnMeldung_Click()
    MsgBox "btnMeldung wurde angeklickt"
End Sub
```

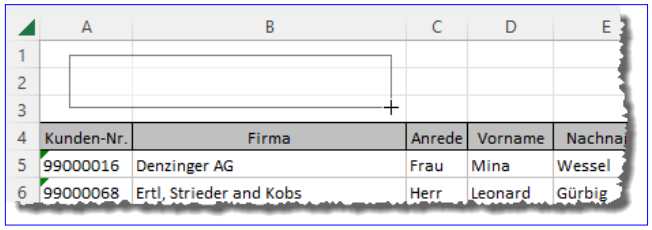


Bild 3: Aufziehen der Schaltfläche im Arbeitsblatt

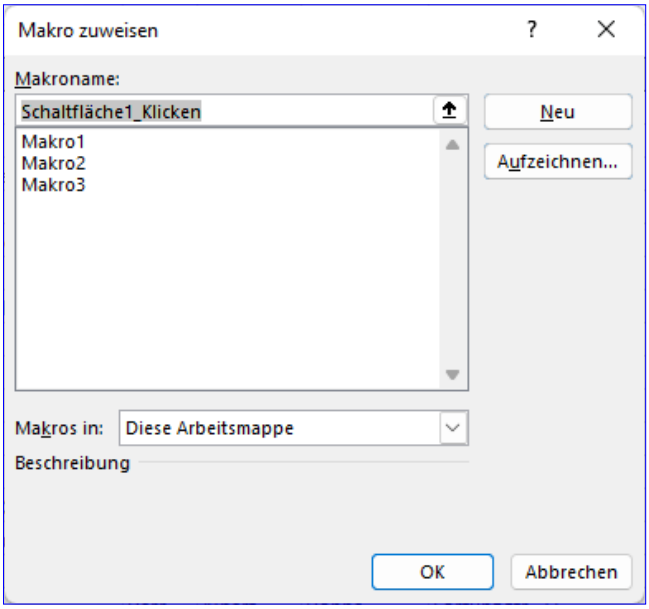


Bild 4: Zuweisen eines Makro zur neuen Schaltfläche

Der Dialog erlaubt im Kombinationsfeld **Makros in:** noch die Auswahl der Arbeitsmappe, in der sich das Makro befindet. Wir empfehlen, nur Makros aus der aktuellen Arbeitsmappe zu nutzen – zumindest, wenn die Arbeitsmappe einmal weitergegeben oder aus einem anderen Rechner genutzt werden soll.

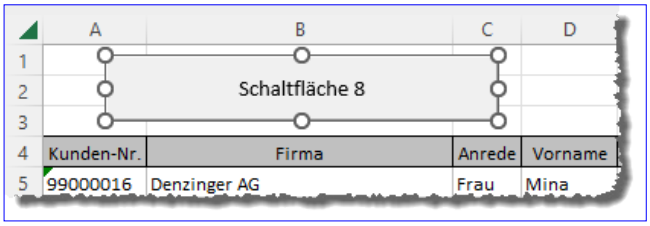


Bild 5: Die neue Schaltfläche

Schaltfläche ausprobieren

Nach dem Schließen des VBA-Editors erscheint die Schaltfläche vermutlich noch mit den Punkten zum Anpassen der Größe der Schaltfläche (siehe Bild 5).

Nach einem Klick irgendwo ins Arbeitsblatt verschwinden diese Markierungen und Du kannst die Schaltlfäche ausprobieren. Danach erscheint auch die in der Prozedur definierte Meldung.

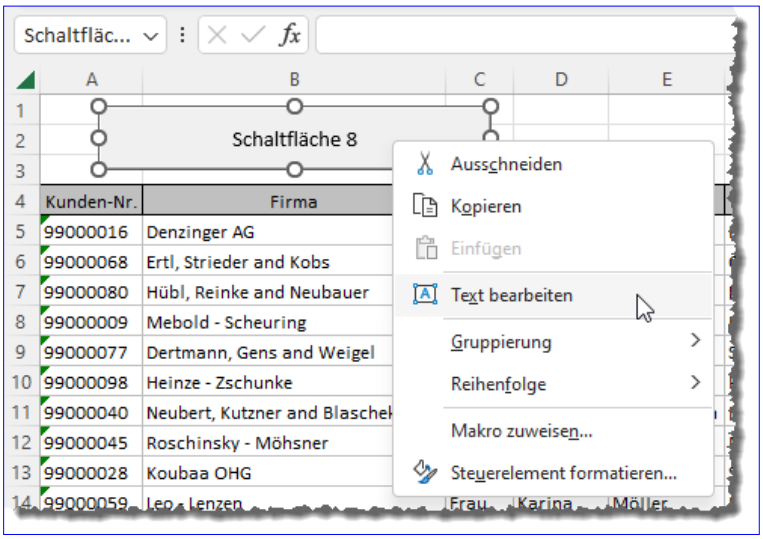


Bild 6: Kontextmenü der Schaltfläche

Schaltfläche anpassen

Als Nächstes passen wir die Beschriftung der Schaltfläche an. Dazu gibt es verschiedene Möglichkeiten. Die offensichtlichste ist ein Rechtsklick auf die Schaltfläche und das Auswählen des Eintrags **Text bearbeiten** (siehe Bild 6.

Weitere Möglichkeiten zur Formatierung

Im Kontextmenü finden wir noch weitere Möglich-

Position abhängig von Eigenschaften des Arbeitsblatts einstellen

Wenn wir keine weitere Anpassung vornehmen, kann die Position und Größe der Schaltfläche sich beim Ändern der Zeilenhöhen oder Spaltenbreiten der Zellen,

Excel: Workbooks und Worksheets per VBA

Das Objektmodell von Excel bietet eine ganze Reihe hierarchisch angeordneter Elemente, die wir uns in diesem Artikel ansehen. Dabei schauen wir uns auch gleich an, wie wir diese per VBA referenzieren können. Wir beginnen ganz oben in der Hierarchie mit der Excel-Anwendung und arbeiten uns dann über die verschiedenen Elemente bis hin zur einzelnen Zelle mit ihren Eigenschaften. Nach der Lektüre dieses Artikels kannst Du alle wichtigen Elemente eines Excel-Workbooks referenzieren, wirst diese in Schleifen durchlaufen und je nach Objekttyp neue Elemente hinzufügen, bearbeiten oder entfernen können. Außerdem lernst Du noch einige Grundlagen des VBA-Editors kennen.

Hinweis: Speichern als .xlsm-Datei nötig

Wir beschreiben in diesem Artikel einige Elemente, die zum VBA-Projekt einer Excel-Datei hinzugefügt werden können. Damit die Änderungen an diesem VBA-Projekt mit der Excel-Datei gespeichert werden, musst Du diese unter einem anderen Dateityp speichern, nämlich als **Excel-Arbeitsmappe mit Makros (*.xlsm)**.

Dazu erscheint allerdings auch noch eine Meldung, bevor Du die Datei ohne diese Änderungen speicherst. Lies diese genau, um nicht versehentlich den hinzugefügten VBA-Code zu verlieren.

Ganz oben: Das Application-Objekt

Wenn wir ganz oben in der Hierarchie der Excel-Objekte beginnen wollen, dann sprechen wir über das **Application**-Objekt, also die eigentliche Anwendung.

Ein solches Objekt gibt es für alle Office-Anwendungen, die VBA-Unterstützung anbieten, also zum Beispiel Access, Excel, Outlook, PowerPoint und Word. Das **Application**-Objekt enthält alle weiteren Objekte, die wir für die Automatisierung der jeweiligen Anwendung benötigen – zum Beispiel Eigenschaften, über die wir die aktive Arbeitsmappe (**Workbook**) oder das aktuelle Tabellenblatt (**Worksheet**) referenzieren und diese steuern können.

Application-Objekt: Von innen oder außen referenzieren?

Bei jeder Office-Anwendung gibt es verschiedene Möglichkeiten, das **Application**-Objekt der jeweiligen Anwendung zu referenzieren.

Dabei kommt es darauf an, von wo aus man darauf zugreifen möchte. Wir haben zum Beispiel die folgenden Optionen:

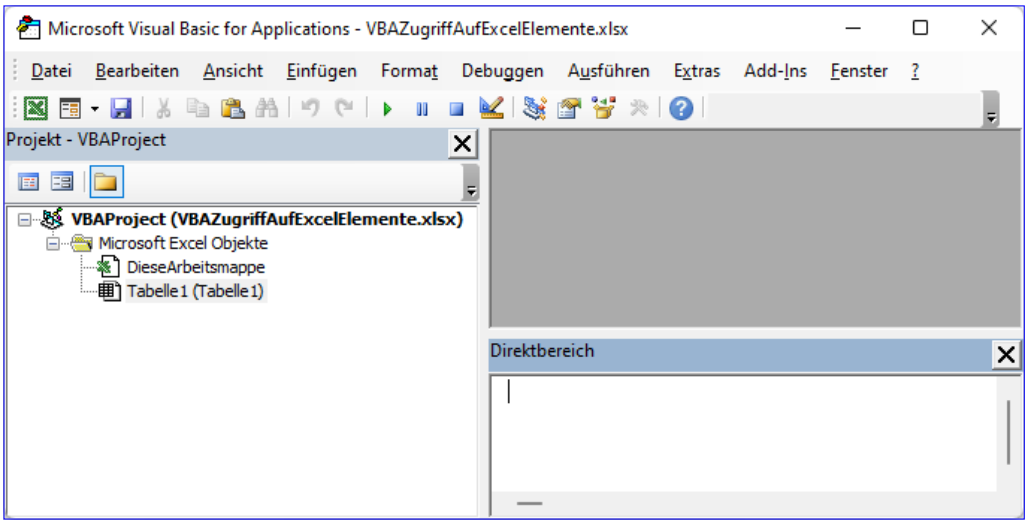


Bild 1: Der VBA-Editor für eine neue, leere Excel-Datei

- Zugriff vom VBA-Projekt einer Excel-Datei. Hier können wir einfach über die **Application**-Klasse auf die aktuelle Instanz von Excel zugreifen oder seine Eigenschaften, Methoden und Ereignisse nutzen.

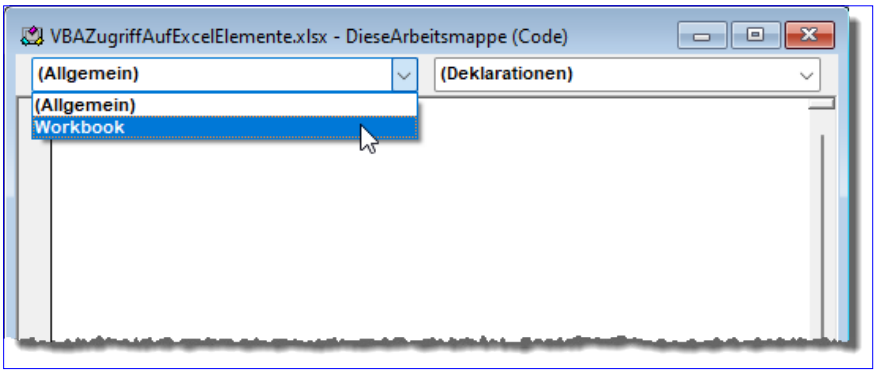


Bild 2: Auswahl der Klasse, auf welches sich das Klassenmodul bezieht

- Von einem VBA-Projekt oder einem ähnlichen Projekt außerhalb der Excel-Anwendung (zum Beispiel einer Anwendung oder einer DLL auf Basis von VB.NET oder VB6/twinBASIC). Dann gibt es verschiedene Techniken, um eine neue Excel-Instanz zu erstellen oder eine vorhandene zu nutzen oder auch um ein Workbook zu öffnen oder ein vorhandenes zu referenzieren.
- Von einem COM-Add-In aus, das von Excel selbst gestartet wird. Hier übergibt Excel beim Starten des COM-Add-Ins eine Referenz auf das Application-Objekt der aktuellen Instanz an das COM-Add-In, wo diese Referenz in einer Variablen gespeichert und anschließend verwendet werden kann.

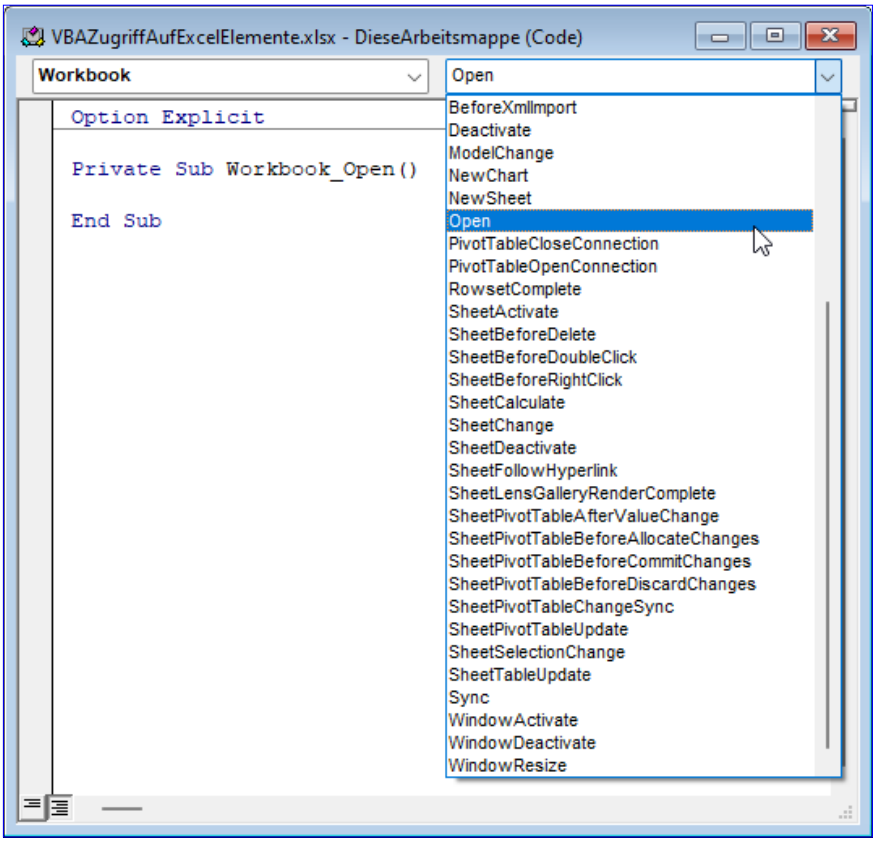


Bild 3: Auswahl der Ereignisse der Klasse

Wir wollen in diesem Artikel von der ersten Variante ausgehen. Die übrigen beiden Möglichkeiten, über das **Application**-Objekt von Excel auf das angezeigte **Workbook** mit den Worksheets und den enthaltenen Daten zuzugreifen, stellen wir in weiteren Artikeln vor.

VBA-Editor anzeigen

Wenn Du unter Excel bisher noch nicht oder nur in Zusammenhang mit dem Aufzeichnen von Makros mit dem VBA-Editor Kontakt hattest, ist dies der einfachste Weg, diesen zu aktivieren: Ist Excel geöffnet, betätigst Du einfach die Tastenkombination **Alt + F11** und der VBA-Editor erscheint.

Dieser zeigt zu Beginn üblicherweise die Ansicht aus Bild 1 an. Es kann auch sein, dass der linke Be-

reich (der Projekt-Explorer) oder der untere Bereich nicht angezeigt werden. Diese kannst Du mit **Strg + R** (Projekt-Explorer) und **Strg + G** (Direktbereich) einblenden.

Der Projekt-Explorer zeigt aktuell zwei Objekte an, nämlich ein Klassenmodul namens **DieseArbeitsmappe**, dem Du Code hinzufügen kannst, der sich auf die komplette Arbeitsmappe bezieht und eines namens **Tabelle1**, das den Code für das Worksheet beziehungsweise das Tabellenblatt namens **Tabelle1** aufnimmt. Fügen wir weitere Tabellenblätter hinzu, landen automatisch entsprechende Klassenmodule im VBA-Projekt der Excel-Datei.

Klassenmodule für Workbook und Worksheet

Dass es sich bei den beiden Modulen um Klassenmodule und nicht um Standardmodule handelt, erkennt man nach dem Öffnen der Module, wenn man auf das linke obere Kombinationsfeld im Codefenster klickt. Hier erscheint beispielsweise für das Modul **DieseArbeitsmappe** ein Eintrag namens **Workbook** (siehe Bild 2), bei den Klassenmodulen zu den Tabellenblättern erscheint der Eintrag **Worksheet**.

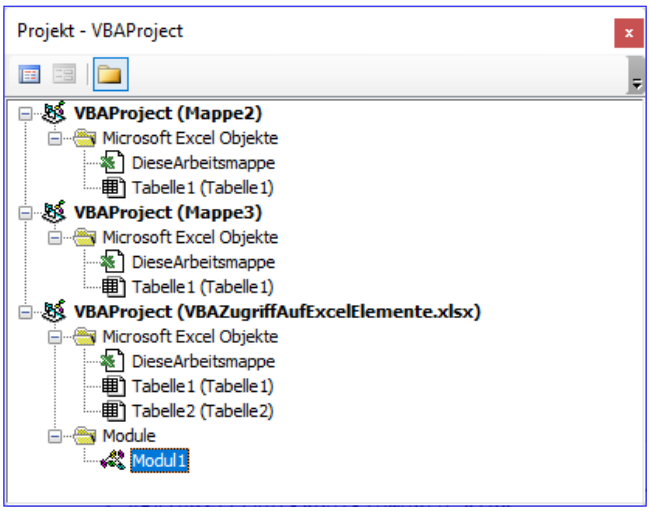


Bild 4: Projekt-Explorer mit mehr als einem Excel-VBA-Projekt

Wir können die Klasse, auf das sich die Klassenmodule beziehen, also direkt auswählen. Was geschieht danach? Der VBA-Editor legt direkt nach der Auswahl eine Prozedur für das Standardereignis dieser Klasse an, im Fall des Moduls **DieseArbeitsmappe** das Ereignis **Workbook_Open** (siehe Bild 3).

Außerdem können wir nach der Auswahl des Eintrags **Workbook** im rechten Kombinationsfeld die übrigen Ereignisse der Klasse auswählen und somit weitere Ereignisprozeduren anlegen.

Workbook: Kein Buch, sondern eine Datei

Aus Sicht von VBA hat jede Excel-Datei ein eigenes VBA-Projekt. Wenn Du mehrere Excel-Dateien auf dem gleichen Rechner öffnest und den VBA-Editor aktivierst, findest Du im Projekt-Explorer für jede Excel-Datei ein eigenes VBA-Projekt.

Bei drei geöffneten Excel-Dateien erscheinen diese im Projekt-Explorer wie in Bild 4.

Jedes dieser VBA-Projekte enthält genau ein Klassenmodul des Typs **Workbook** und ein oder mehrere des Typs **Worksheet**. Auch diese sind in der Abbildung gut zu erkennen.

Auf aktives Workbook zugreifen

Um auf ein Workbook zuzugreifen, gibt es verschiedene Möglichkeiten. Das aktive Workbook, also das Workbook, das aktuell in Windows den Fokus hat, können wir mit dem folgenden Ausdruck referenzieren:

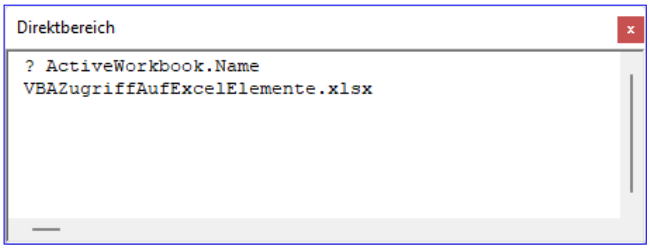


Bild 5: Zugriff auf das aktuelle **Workbook**-Element

```
? ActiveWorkbook.Name
VBAZugriffAufExcelElemente.xlsx
```

Wenn mehrere Workbooks und somit mehrere Excel-Fenster geöffnet sind, gilt dies ebenfalls.

Diesen Befehl setzen wir beispielsweise im Direktbereich des VBA-Editors ab, der sich zum Ausprobieren einzelner Anweisungen gut eignet (siehe Bild 5).

Workbook des aktuellen VBA-Projekts referenzieren

Wenn Du hingegen das Workbook referenzieren willst, das zum aktuellen VBA-Projekt gehört, verwendest Du die Funktion **ThisWorkbook**. Diese Funktion liefert einen Verweis auf das Workbook, in dessen VBA-Projekt die Funktion aufgerufen wird.

Wenn Du diesen Aufruf wie folgt in einer Prozedur unterbringst, ist der Fall klar – es liefert den Namen des Workbooks, zu dem das VBA-Projekt gehört und niemals das eines anderen VBA-Projekts:

```
Private Sub Workbook()
    Debug.Print "ThisWorkbook: " & ThisWorkbook.Name
End Sub
```

Wenn Du den Namen hingegen über den Direktbereich abfragst, und der VBA-Editor zeigt gerade mehr als ein VBA-Projekt an, weil mehr als eine Excel-Datei geöffnet ist, dann liefert **ThisWorkbook.Name** den Namen des Workbooks, das gerade im Projekt-Explorer markiert ist (beziehungsweise eines der enthaltenen Elemente).

Direkter Zugriff auf die Elemente der Application-Klasse

Weiter oben haben wir gesagt, dass die **Application**-Klasse das oberste Element im VBA-Objektmodell von Excel ist. Wie können wir jetzt ohne Nutzung von **Application** auf ein Element wie **ActiveWorkbook** zugreifen? Der Grund ist, dass wir die Elemente der **Application**-Klasse innerhalb des eigenen VBA-Projekts auch ohne Angabe von Application nutzen können.

Wir können also alle Eigenschaften und Methoden, die sich unterhalb des **Application**-Objekts befinden, direkt angeben und nutzen. Welche das sind, sehen wir im Objektkatalog, den wir mit der Taste **F2** öffnen können. Wählen wir hier ganz oben die Bibliothek **Excel** aus, finden wir unter **Klassen** schnell den Eintrag **Application**.

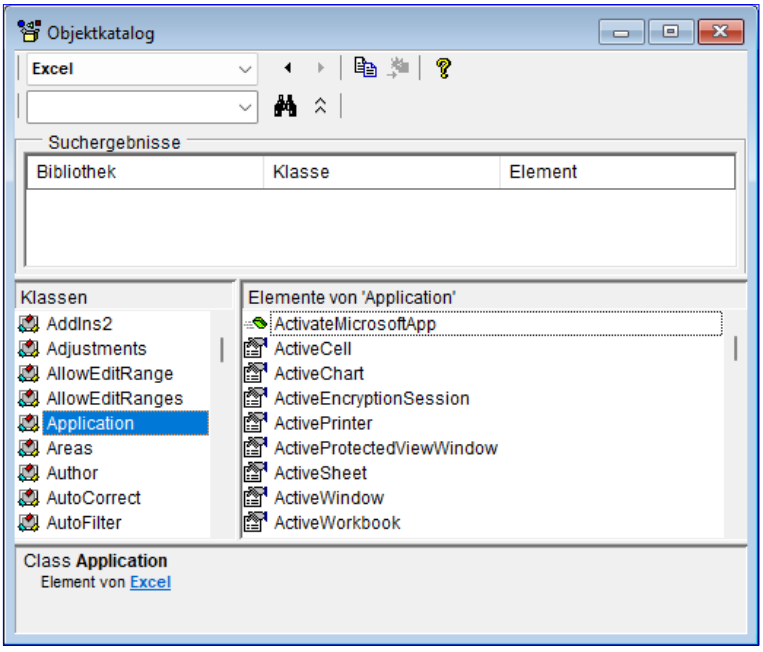


Bild 6: Elemente der **Application**-Klasse im Objektkatalog

Klicken wir diesen an, zeigt der Objektkatalog auf der rechten Seite alle enthaltenen Elemente an (siehe Bild 6).

Auf die Liste der Workbooks zugreifen

Somit können wir nicht nur über die Eigenschaft **ActiveWorkbook** auf das aktuelle Workbook zugreifen, sondern auch auf eine Auflistung wie Workbooks. Diese liefert die

Anzahl der aktuell geöffneten Workbooks beziehungsweise Excel-Dateien – wieder im Direktbereich eingesetzt:

```
? Workbooks.Count
3
```

Dieses Ergebnis erscheint, wenn wie in der obigen Abbildung drei Workbooks geöffnet sind.

Alle geöffneten Workbooks durchlaufen

Wenn wir alle aktuell geöffneten **Workbook**-Elemente ermitteln wollen, reicht der Direktbereich allerdings nicht mehr aus.

Dazu legen wir eine Prozedur an, was wir wahlweise in einem der beiden vorhandenen Klassenmodule erledigen können oder auch in einem neuen Standardmodul.

In dieser Prozedur deklarieren wir eine Variable namens **objWorkbook**, mit der wir jeweils ein Workbook referenzieren wollen.

Danach durchlaufen wir in einer **For Each**-Schleife alle Elemente der **Workbooks**-Auflistung und referenzieren dabei das aktuelle Elemente jeweils mit der Variablen **objWorkbook**:

```
Private Sub AlleWorkbooks()
    Dim objWorkbook As Workbook
    For Each objWorkbook In Workbooks
        Debug.Print objWorkbook.Name
    Next objWorkbook
End Sub
```

Das Ergebnis sehen wir in Bild 7.

Workbook per Name referenzieren

Wir können die Workbooks nicht nur durchlaufen, sondern diese auch gezielt referenzieren. Dabei verwenden wir ebenfalls die **Workbooks**-Auflistung. Außerdem geben wir in Klammern den Namen des Workbooks an:

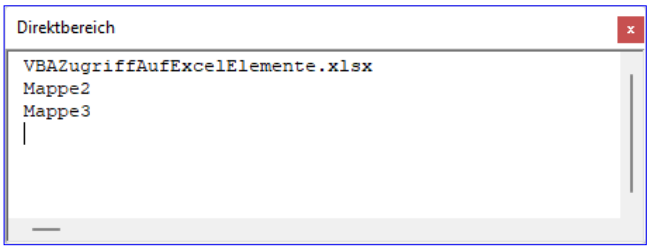


Bild 7: Ausgabe der Workbooks

```
Dim objWorkbook As Workbook
Set objWorkbook = Workbooks("Mappe7")
Debug.Print objWorkbook.Name
```

Workbook aktivieren

Wenn mehrere Workbooks in Excel geöffnet sind und Du eines der Workbooks aktivieren möchtest, sodass es im Vordergrund von Windows angezeigt wird, kannst Du die **Activate**-Methode des **Workbook**-Objekts verwenden. Für ein Workbook namens **Mappe7** gelingt dies beispielsweise wie folgt:

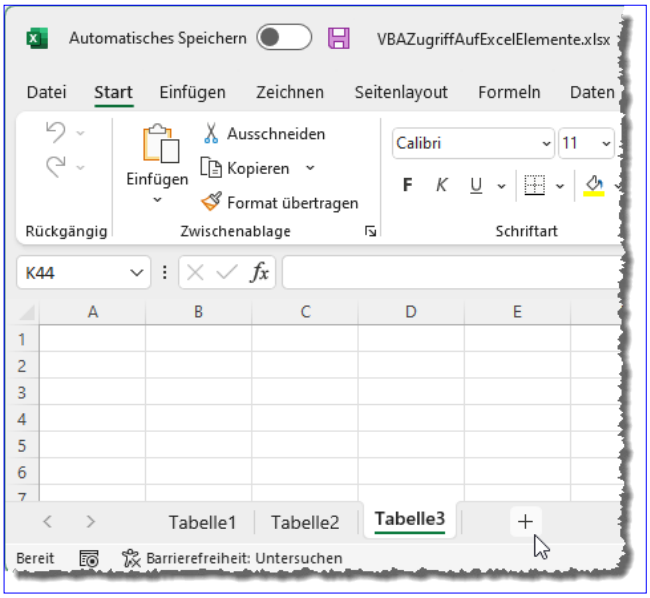


Bild 8: Hinzufügen neuer Arbeitsblätter

```
Dim objWorkbook As Workbook
Set objWorkbook = Workbooks("Mappe7")
objWorkbook.Activate
```

Nach diesem Schritt ist dieses Workbook übrigens auch über die Eigenschaft **ActiveWorkbook** erreichbar.

Neues Workbook erstellen

Wir können per VBA auch neue Workbooks erstellen – dies hat den gleichen Effekt, als wenn wir einfach eine neue Excel-Instanz öffnen und dort eine neue Arbeitsmappe erstellen. Dazu benötigen wir lediglich den Befehl **Add** der **Workbooks**-Auflistung:

```
Workbooks.Add
```

Setzen wir diesen im Direktbereich ab, erscheint direkt eine neue Excel-Instanz mit der neuen Arbeitsmappe. Entsprechend finden wir auch im Projekt-Explorer direkt einen neuen Eintrag für das VBA-Projekt dieser Arbeitsmappe mit **Workbook**- und **Worksheet**-Klassen vor.

Außerdem liefert die **ActiveWorkbook**-Funktion nun einen Verweis auf das neu erstellte Workbook.

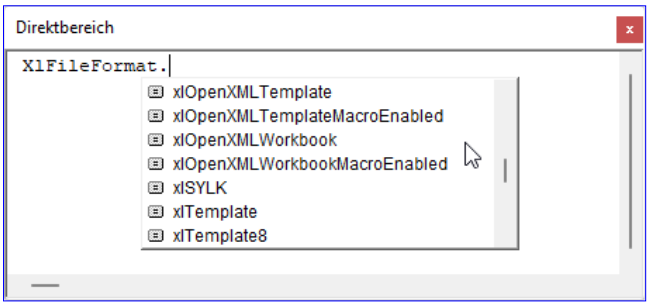


Bild 9: Excel-Formate per IntelliSense

Workbook per VBA schließen

Um ein Workbook zu schließen, gibt es zwei Möglichkeiten. Die erste ist, die **Close**-Methode der **Workbooks**-Auflistung zu verwenden. Diese schließt nicht etwa eine als Parameter übergebene Excel-Datei, sondern alle aktuell geöffneten Workbooks:

```
Workbooks.Close
```

Die zweite Möglichkeit ist das explizite Schließen eines Workbooks. Dazu referenzieren wir dieses – beispielsweise über **ActiveWorkbook** oder **ThisWorkbook**:

```
ThisWorkbook.Close
```

Damit schließen wir genau das mit **ThisWorkbook** re-

Excel: Zellen und Bereiche per VBA

Im Artikel »Excel: Workbooks und Worksheets per VBA« haben wir uns angesehen, wie wir Arbeitsmappen und Arbeitsblätter mit VBA nutzen können. Im vorliegenden Artikel gehen wir einen Schritt weiter und nehmen uns die offensichtlichen Elemente eines Arbeitsblatts vor – die Zellen. Nicht weniger spannend sind allerdings die Bereiche, unter VBA »Range« genannt. Wie wir diese referenzieren, auslesen und bearbeiten können, zeigen wir auf den folgenden Seiten.

Worksheet referenzieren

Im oben genannten Artikel **Excel: Workbooks und Worksheets per VBA** (**www.access-im-unternehmen.de/326**) zeigen wir, wie Du **Workbook**- und **Worksheet**-Elemente referenzieren kannst. Im aktuellen Artikel verwenden wir oft das aktuelle **Worksheet**-Element als Basis für den Zugriff auf verschiedene Eigenschaften wie **Range** oder **Cells**.

Dieses können wir mit **ActiveSheet** referenzieren, allerdings bietet **ActiveSheet** kein IntelliSense an. Das liegt daran, dass **ActiveSheet** nicht nur ein **Worksheet**-Objekt, sondern auch ein **Chart**-Objekt zurückliefern könnte.

m diese Einschränkung zu umgehen, referenzieren wir das **Worksheet**-Objekt jeweils mit folgenden Anweisungen:

```
Dim wks As Worksheet
Set wks = ActiveSheet
```

wks ist nun explizit als **Worksheet**-Objekt deklariert und liefert folglich dessen Elemente per IntelliSense. Wir werden die Deklaration und Zuweisung des aktuellen Worksheets nicht in jedem Beispiel explizit ausführen, sondern nach dem ersten Beispiel einfach mit **wks** arbeiten.

Gleichwohl sei erwähnt, dass Du innerhalb des VBA-Projekts einer Excel-Arbeitsmappe auch direkt die **Range**- oder die **Cells**-Eigenschaft nutzen kannst. Im

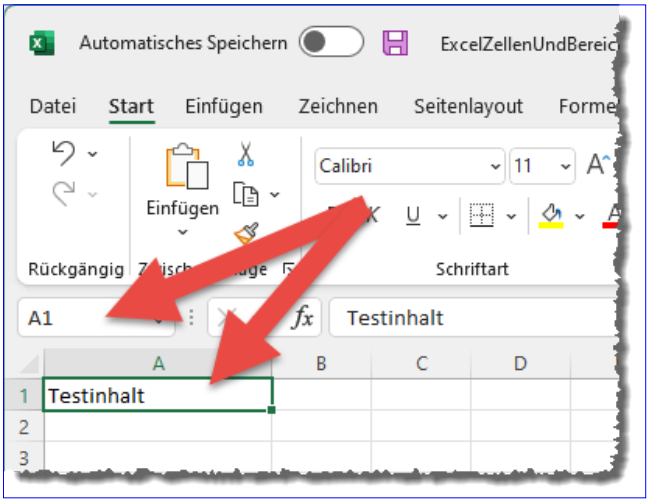


Bild 1: Die Bezeichnung der aktuellen Zelle, hier **A1**

Direktfenster erhältst Du also beispielsweise mit folgendem Ausdruck den Inhalt der Zelle mit der Adresse **A1**:

```
? Range("A1")
```

Da ich allerdings davon ausgehe, dass Du auch von anderen Anwendungen wie Access, Word oder Outlook oder auch von COM-Add-Ins auf die Inhalt von Excel-Arbeitsmappen zugreifen willst, verwenden wir immer die explizite Angabe der Objekte.

Einzelne Zelle über die A1-Notation per Range referenzieren

Im Excel-Arbeitsblatt erhalten wir schon Hinweise darauf, wie ein Weg zum Referenzieren einer Zelle aussehen könnte (siehe Bild 1).

Damit sind die Spalten- und Zeilenüberschriften gemeint, die aus Buchstaben und Zahlen bestehen. Die Spaltenköpfe enthalten zunächst die Buchstaben von **A** bis **Z**, dann geht es mit **AA** bis **ZZ** weiter und dann mit **AAA** bis **XFD**, was der Spalte 16.384 entspricht. Die Zeilenköpfe enthalten die Zahlen von 1 bis 1.048.576.

Eine Zelle hat eine Benennung, die aus dem Buchstaben der Spalte und der Zahl der Zeile besteht. Die Zeile oben links heißt also **A1**, während die Zeile ganz unten rechts **XFD1048576** heißt (ich hoffe, Elon Musk liest das nicht, sonst benennt er sein nächstes Kind nach dieser Zelle).

Diese Bezeichnung ist auch gleichzeitig der erste Weg, mit dem wir auf eine Zelle zugreifen können. Um beispielsweise über den Ausdruck **A1** auf eine Zelle zugreifen zu können, verwenden wir die **Range**-Eigenschaft des **Worksheet**-Objekts:

```
Public Sub ZelleMitRange()
    Dim wks As Worksheet
    Set wks = ActiveSheet
    Debug.Print wks.Range("A1")
End Sub
```

Dies gibt den Inhalt der Zelle im Direktbereich aus.

Einzelne Zelle über die R1C1-Notation per Cells referenzieren

Wenn Du den numerischen Index der Zeile und Spalte der Zelle kennst, die Du referenzieren möchtest, kannst Du auch die **Cells**-Eigenschaft nutzen.

Diese erwartet als Parameter die Angabe des Zeilenindexes und des Spaltenindexes in der **R1C1**-Notation.

Das **R** und das **C** in dieser Bezeichnung stehen für die englischen Begriffe für Zeile und Spalte, also **Row** und **Cell**. Der folgende Ausdruck liefert also den Inhalt der Zelle in der ersten Zeile und der ersten Spalte:

	A	B	C	D	E
1	A1 - 1,1	B1 - 1,2	C1 - 1,3	D1 - 1,4	
2	A2 - 2,1	B2 - 2,2	C2 - 2,3	D2 - 2,4	
3	A3 - 3,1	B3 - 3,2	C3 - 3,3	D3 - 3,4	
4	A4 - 4,1	B4 - 4,2	C4 - 4,3	D4 - 4,4	
5					
6					

Bild 2: Beispielzellen

```
Debug.Print wks.Cells(1, 1)
```

Für eine andere Zeile, zum Beispiel die zweite, änderst Du den Wert des ersten Parameters. So erhältst Du den Inhalt der Zelle **A2**:

```
Debug.Print wks.Cells(2, 1)
```

Aktuelle Zelle referenzieren

Um die aktuelle Zelle zu referenzieren, benötigen wir gar kein **Worksheet**-Objekt – dieses wird explizit ermittelt. Um den Inhalt der aktuellen Zelle auszugeben, nutzen wir die folgende Anweisung:

```
Debug.Print ActiveCell
```

Mehrere Zellen mit Range referenzieren

Während beiden zuvor beschriebenen Beispiele sich jeweils auf das Referenzieren einer einzelnen Zelle beschränkten, können wir mit der **Range**-Eigenschaft auch mehr als eine Zelle gleichzeitig referenzieren.

Hier können wir die A1-Notation verwenden, über einen kleinen Umweg aber auch die R1C1-Notation. Angenommen, wir wollen einen Bereich mit allen Zellen des Worksheets aus Bild 2 referenzieren.

Mehrere Zellen mit der A1-Notation

Wenn wir die **Range**-Eigenschaft mit der A1-Notation nutzen, lautet die Referenz wie folgt:

```
Set rng = wks.Range("A1:D4")
```

Dass wir wirklich den gewünschten Bereich referenzieren, können wir zum Beispiel durch die Ausgabe der Anzahl der Zeilen und Spalten erreichen:

```
Debug.Print "Zeilen: " & rng.Rows.Count
Debug.Print "Spalten: " & rng.Columns.Count
```

Die Ausgabe lautet:

```
Zeilen: 4
Spalten: 4
```

Wir geben die linke obere und die rechte untere Zelle des Bereichs also in einer Zeichenkette durch einen Doppelpunkt getrennt an.

Mehrere Zellen mit der R1C1-Notation

Wir können die **Range**-Eigenschaft mit der **Cells**-Eigenschaft in der Form kombinieren, dass wir die linke, obere und die rechte, untere zu referenzierende Zelle jeweils mit der **Cells**-Eigenschaft referenzieren und diese per Komma getrennt der **Range**-Eigenschaft übergeben.

Für den gleichen Bereich wie im vorherigen Beispiel lautet der Ausdruck dann:

```
Set rng = wks.Range(Cells(1, 1), Cells(4, 4))
```

Komplette Spalte mit Range referenzieren

Wenn Du eine komplette Spalte, zum Beispiel die Spalte **A**, mit einem **Range**-Elemente referenzieren möchtest, gibst Du den Bereich ohne Angabe von Zeilen an:

```
Set rng = wks.Range("A:A")
```

Für mehrere zusammenhängende Spalten, zum Beispiel Spalte A bis Spalte C:

```
Set rng = wks.Range("A:C")
```

Komplette Zeile mit Range referenzieren

Für das Referenzieren einer kompletten Zeile, beispielsweise der ersten Zeile, mit dem **Range**-Element nutzt Du den folgenden Ausdruck:

```
Set rng = Range("1:1")
```

Für mehrere zusammenhängende Zeilen verwendest Du analog zu den Spalten den folgenden Ausdruck für die ersten drei Zeilen:

```
Set rng = Range("1:3")
```

Nicht zusammenhängende Zellen referenzieren

Wenn Du einzelne Zellen oder Bereiche von Zellen referenzieren möchtest, die nicht unbedingt zusammenhängen, dann gibst Du diese für die **Range**-Methode durch Kommata voneinander getrennt an.

Für einzelne Zellen sieht das so aus:

```
Dim rng As Range
Set rng = wks.Range("A1, B2, C3, D4")
Debug.Print rng.Cells.Count 'gibt 4 aus
```

Auf die gleiche Weise kannst Du auch nicht unbedingt zusammenhängende Bereiche referenzieren, hier für die Bereiche **A1** bis **A4** und **C1** bis **C4**:

```
Dim rng As Range
Set rng = wks.Range("A1:A4, C1:C4")
Debug.Print rng.Cells.Count 'gibt 8 aus
```

Wert aus einer Zelle lesen

Wie das Lesen aus Zellen funktioniert, haben wir in den vorherigen Beispielen zum größten Teil schon beschrieben.

Dort haben wir die ermittelten Inhalte der Zellen jeweils im Direktbereich ausgegeben.

Den Wert einer einzelnen Zelle kannst Du auch einer Variablen zuweisen und anschließend im Direktbereich ausgeben:

```
Dim strZelle As String
strZelle = wks.Cells(1, 1)
Debug.Print strZelle
```

Werte aus einem Bereich von Zellen lesen

Spannender wird es, wenn wir gleich mehrere Werte aus einer Zelle lesen wollen. Dazu gibt es verschiedene Möglichkeiten.

Eine davon ist, den Bereich über die R1C1-Notation mit der **Cells**-Methode in Schleifen zu durchlaufen und innerhalb der Schleife die Werte zu ermitteln oder auszugeben:

```
Dim r As Long
Dim c As Long
For c = 1 To 4
    For r = 1 To 4
        Debug.Print r, c, wks.Cells(r, c)
    Next r
Next c
```

Werte in ein Variant-Array einlesen

Vielleicht möchtest Du die Daten aber auch in ein Ar-

```
    Next r
Next c
```

Adress: Adresse eines referenzierten Bereichs ausgeben

Wenn Du wissen möchtest, welche Adresse ein Bereich hat, den Du zuvor referenziert hast, kannst Du die **Address**-Eigenschaft des **Range**-Objekts verwenden. Für eine einzelne Zelle gelingt dies wie folgt:

```
? Range("A1").Address
$A$1
```

Das Ergebnis ist im Prinzip die Adresse im A1-Format mit vorangestellten Dollar-Zeichen (**$**).

Bei einem Bereich, der mehr als eine Zelle umfasst, wie beispielsweise von **A1** bis **B3**, wird die Adresse wie folgt ausgegeben:

```
? Range("A1:B3").Address
$A$1:$B$3
```

Eine Zelle ist ein Range

Wenn wir eine Zelle referenzieren, beispielsweise über den Ausdruck **ActiveCell** für die aktive Zelle, dann würde man annehmen, dass man mit einem Objekt des Typs **Cell** arbeitet

Excel: Sheet-Navigation per Button

Wenn man ein Excel-Workbook mit vielen verschiedenen Worksheets verwendet, möchte man beim Öffnen des Workbooks vielleicht eine Übersichtsseite präsentiert bekommen, von der aus man per Mausklick auf entsprechende Schaltflächen zu den übrigen Worksheets gelangt – und am besten von dort aus mit einer weiteren Schaltfläche wieder zurück zur Übersicht. In diesem Artikel zeigen wir, wie das gelingt, und warum wir noch nicht mal eine einzige Zeile VBA-Code für dieses Vorhaben benötigen.

In unserem Beispiel gehen wir davon aus, dass wir vier Worksheets nutzen. Das erste namens **Start** soll Schaltfläche enthalten beziehungsweise entsprechende Formen mit Beschriftungen. Die drei weiteren Worksheets namens **Tabelle 1**, **Tabelle 2** und **Tabelle 3** enthalten jeweils eine Schaltfläche beziehungsweise eine Form, um zur Startseite zurückzukehren.

Grundlegende Informationen zu Schaltflächen und Alternativen unter Excel findest Du im Artikel **Buttons in Excel** (**www.vbentwickler.de/328**).

Startseite gestalten

Dem als Startseite verwendeten Worksheet fügen wir als Erstes die drei Formen hinzu, über die wir die drei übrigen Worksheets ansteuern wollen. Dazu wechseln wir im Ribbon zum Bereich **Einfügen|Illustrationen** und wählen dort den Befehl **Formen|Rechteck: Abgerundete Ecken** aus (siehe Bild 1).

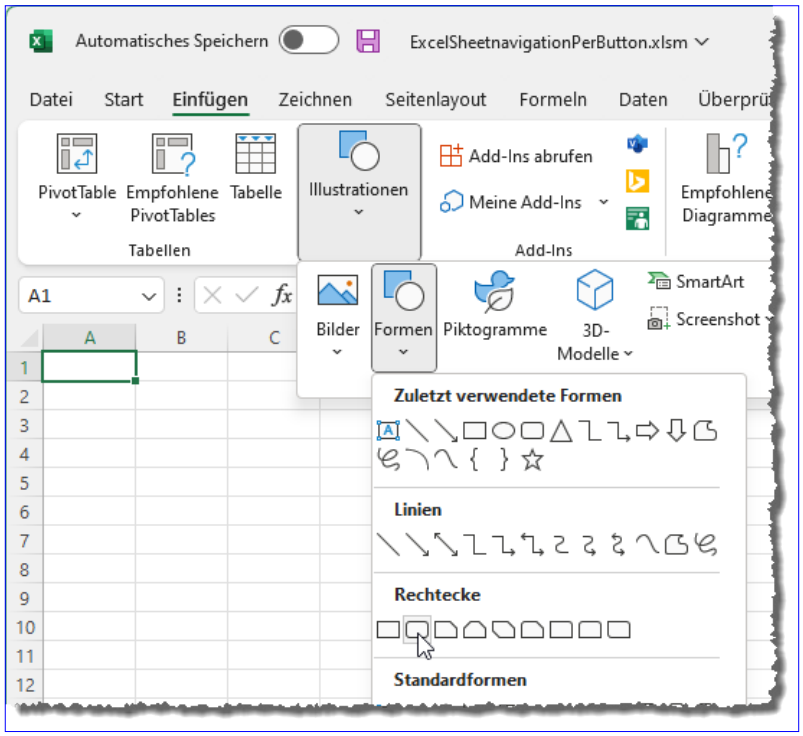


Bild 1: Hinzufügen der als Schaltfläche verwendeten Formen

Auf diese Weise fügen wir drei Rechtecke hinzu, schreiben den Text **Tabelle 1**, **Tabelle 2** und **Tabelle 3** hinein und passen die Größe und Ausrichtung der Texte an (siehe Bild 2).

Ein Tipp dazu: Stelle zuerst eine Schaltfläche fertig und kopiere diese dann, damit Du die Texte nicht immer wieder neu anpassen musst.

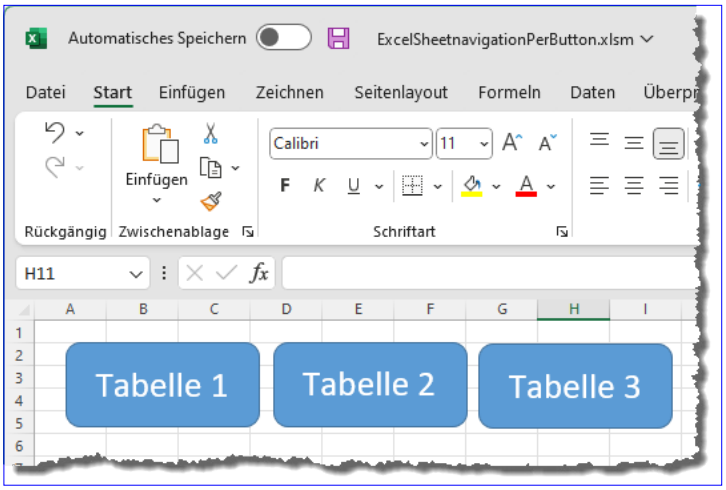


Bild 2: Die drei Schaltflächen zum Öffnen der weiteren Tabellen

Funktion zum Aufrufen einer Tabelle hinzufügen

Nun müssen wir noch die Funktionen hinterlegen, mit denen wir die weiteren Worksheets per Mausklick aufrufen können. Dazu klicken wir als Erstes mit der rechten Maustaste auf die erste Schaltfläche und wählen aus dem Kontextmenü den Eintrag **Link|Link einfügen...** aus (siehe Bild 3).

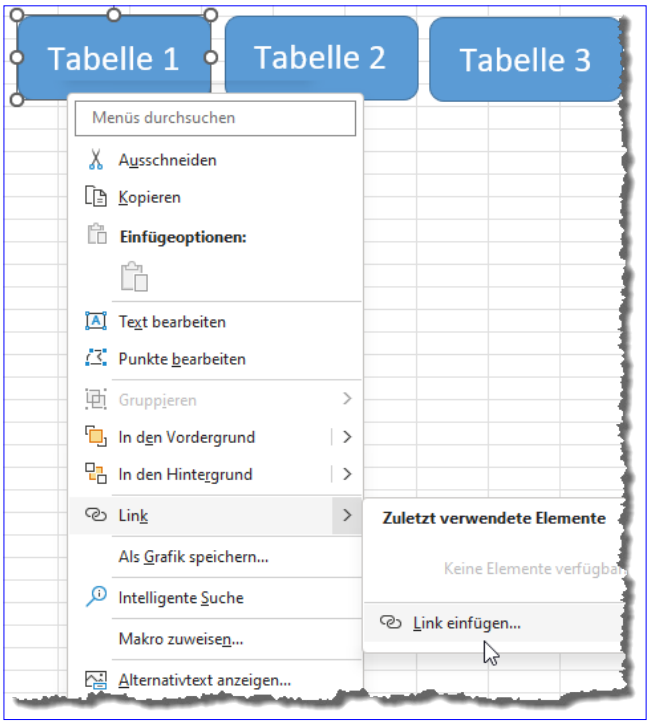


Bild 3: Hinzufügen eines Hyperlinks

Dies öffnet den Dialog **Link einfügen**, in dem wir im linken Bereich zum Eintrag **Aktuelles Dokument** wechseln. Danach finden wir in der Liste in der Mitte die Bezeichnungen unserer Worksheets vor. Hier wählen wir den Eintrag **'Tabelle 1'** aus (siehe Bild 4). Ein Klick auf die Schaltfläche **OK** übernimmt die Änderung und schließt den Dialog.

Ein Klick auf die Schaltfläche mit der Beschriftung **Tabelle 1** wechselt nun direkt zum Worksheet **Tabelle 1**.

Den gleichen Vorgang wiederholen wir für die beiden Schaltflächen zum Öffnen von **Tabelle 2** und **Tabelle 3**. Dabei machen wir von einer Abkürzung Gebraucht, denn wir können im Kontextmenü der Schaltfläche auch direkt den Eintrag **Link** klicken – ohne den Umweg über den Untermenüpunkt **Link einfügen...** zu nehmen.

Dazu kopieren wir eine der drei Schaltflächen der Startseite, indem wir diese mit der rechten Maustaste anklicken und aus dem Kontextmenü den Eintrag **Kopieren** auswählen (damit markierst Du die Schaltfläche und kopierst sie gleichzeitig – achte nur darauf, dass Du nicht versehentlich den Text der Schaltfläche

Excel: Arbeitsblätter per Ribbon steuern

Wenn Du eine eigene Excel-Lösung mit einigen Arbeitsblättern erstellt hast, stört es Dich vielleicht, dass Du nicht schnell über die Registerreiter unten im Excel-Fenster auf alle Arbeitsblätter zugreifen kannst. Je nach Fenstergröße und Anzahl der Arbeitsblätter zeigt Excel dort nämlich nicht alle Arbeitsblätter an. Damit haben wir allerdings ein schönes Beispiel für den Einsatz des Ribbons in einer Excel-Arbeitsmappe. Diesem fügen wir ein Tab mit einem Button für jedes Arbeitsblatt, das schnell erreichbar sein soll, hinzu. Das Ribbon hat noch einen Vorteil: Wenn wir dort für die wichtigsten Arbeitsblätter je eine Schaltfläche hinzufügen, können wir diese auch noch mit einem Icon ausstatten, um das gesuchte Arbeitsblatt noch schneller zu finden.

Grundlagen zu Ribbons in Office-Dokumenten

Wie wir einer Excel-Arbeitsmappe ein eigenes Ribbon hinzufügen, haben wir grundlegend im Artikel **Ribbons in Office-Dokumenten** (**www.vbentwickler.de/329**) erläutert.

Hier haben wir das Tool **Office RibbonX Editor** vorgestellt, mit dem wir das Ribbon eines Office-Dokuments bearbeiten können.

Excel-Arbeitsmappe mit VBA-Code erstellen

Da wir für die geplanten Ribbon-Buttons VBA-Code hinterlegen wollen, können wir direkt eine Excel-Arbeitsmappe mit der Dateiendung **.xlsm** anlegen. Diese nennen wir **Excel_ArbeitsblaetterPerRibbon.xlsm**.

Der Arbeitsmappe fügen wir neben dem vorhandenen Arbeitsblatt noch weitere Arbeitsblätter hinzu. Anschließend soll das Register am unteren Rand wie in Bild 1 aussehen. Hier ist anhand der drei Punkte (...) zu erkennen, dass in dieser Ansicht nicht alle Registerreiter angezeigt werden können.

Ribbondefinition zusammenstellen

Die Ribbondefinition ist nicht besonders komplex, da sie lediglich ein **tab**-, ein **group**- und einige **button**-Elemente enthält, wobei wir für das **tab**-Element noch per Attribut festlegen müssen, an welcher Position es angezeigt werden soll. Vielleicht möchtest Du auch alle eingebauten Ribbon-Tabs ausblenden und nur dieses **tab**-Element anzeigen – das ist noch einfacher zu realisieren.

Bevor wir mit dem Zusammenstellen beginnen, öffnen wir die Excel-Arbeitsmappe mit dem Tool **Office RibbonX Editor**. Hier klicken wir mit der rechten Maus-

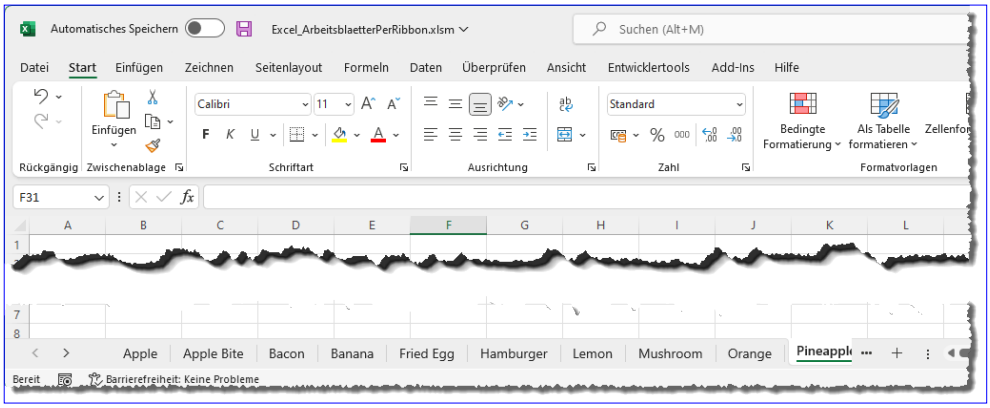


Bild 1: Arbeitsmappe mit einigen Arbeitsblättern

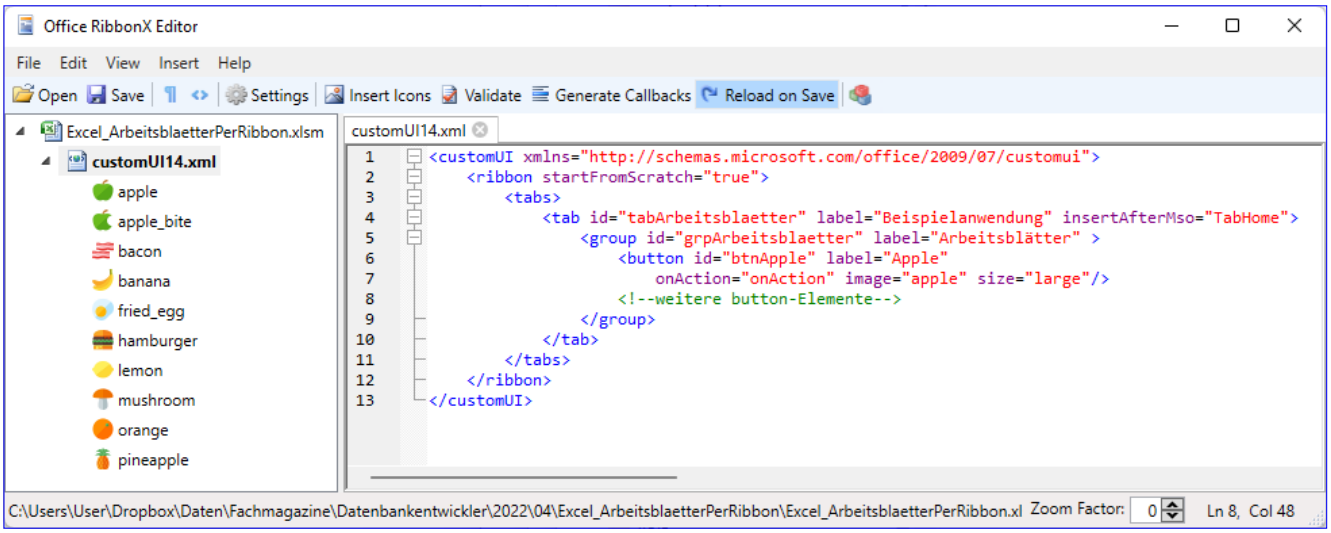


Bild 2: Anlegen der Ribbondefinition plus Icons

taste auf den Eintrag für die Arbeitsmappe und wählen den Befehl **Insert Office 2010+ Custom UI Part** aus. Danach fügen wir die Icons hinzu, die mit den **button**-Elementen angezeigt werden sollen. Dazu betätigen wir die Schaltfläche **Insert Icons** und wählen im nun erscheinenden Dialog die einzufügenden Bilddateien aus. Diese werden anschließend in der Liste links angezeigt.

Klicken wir doppelt auf den Eintrag **customUI14.xml**, erscheint ein leerer Bereich auf der rechten Seite, wo wir die Ribbondefinition eingeben können. Wenn wir alle anderen Elemente ausblenden wollen, können wir dies wie in Bild 2 definieren.

Hier finden wir allerdings nur ein **button**-Element vor:

```
<button id="btnApple" label="Apple" onAction="onAction"
image="apple" size="large"/>
```

XML-Code für Arbeitsblatt-Buttons per VBA erstellen

Um die **button**-Elemente für die übrigen Arbeitsblätter zu definieren, können wir diese von Hand eintippen oder kopieren und anpassen.

Oder wir verfolgen das Credo dieses Magazins und schreiben uns schnell eine Prozedur, die uns den Code für alle Registerreiter zusammenstellt.

```
Public Sub RibbonButtonsDefinieren()
    Dim strXML As String
    Dim strName As String
    Dim wks As Worksheet
    For Each wks In ThisWorkbook.Worksheets
        strName = wks.Name
        strXML = strXML & "<button id=""btn" & Replace(strName, " ", "_") & """ label=""" & strName _
            & """ onAction=""onAction"" image=""" & LCase(Replace(strName, " ", "_")) & """ size=""large""/>" & vbCrLf
    Next wks
    Debug.Print strXML
End Sub
```

Listing 1: Prozedur zum Zusammenstellen von **button**-Definitionen

Diese Prozedur sieht wie in Listing 1 aus. Sie deklariert eine Variable zum Referenzieren von **Worksheet**-Elementen und durchläuft diese in einer **For Each**-Schleife über alle Elemente der Auflistung **ThisWorkbook.Worksheets**.

Innerhalb der Schleife liest sie den Namen des aktuellen **Worksheet**-Elements aus, wie er unten in der Registerleiste dargestellt wird, und speichert diesen in der Variablen **strName**. In der zweiten Anweisung innerhalb der Schleife stellen wir den Code für das jeweilige **button**-Element zusammen.

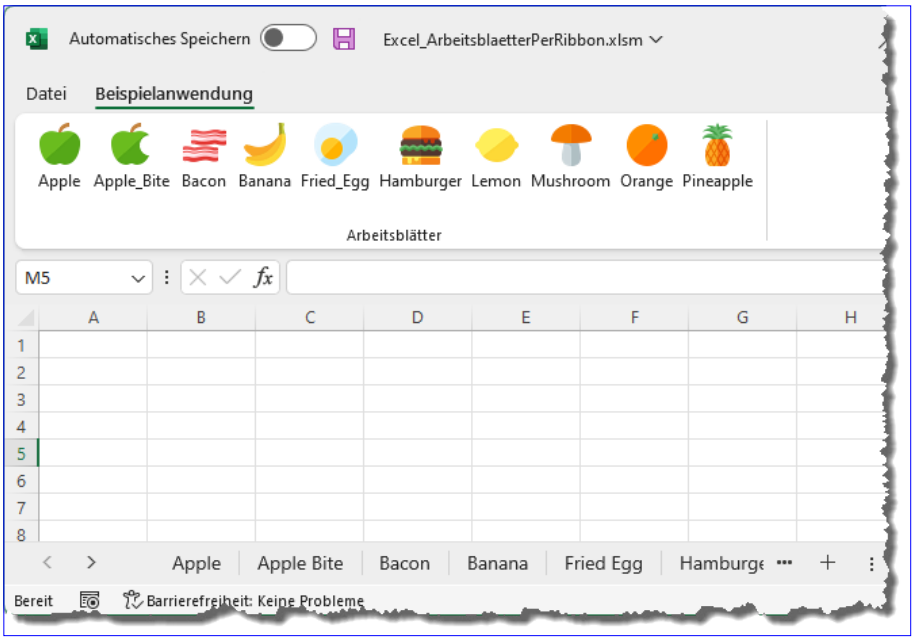


Bild 3: Jedes Arbeitsblatt hat nun einen eigenen Ribbon-Button.

Dabei setzen wir beispielsweise den Wert für das Attribut **id** aus dem Präfix **btn** und dem Namen aus **strName** zusammen, wobei wir hier noch Leerzeichen durch Unterstriche ersetzen. Aus **Apple Bite** wird so **Apple_Bite**. Der Wert für das Attribut **label** entspricht der Beschriftung des Registerreiters für dieses Arbeitsblatt und der Wert für das Attribut **image** entspricht ebenfalls dem Namen, allerdings wieder mit ersetzen

Callbacks zu den Ribbon-Buttons hinzufügen

Das Ribbon sieht nun schon genau so aus, wie wir es uns vorgestellt haben (siehe Bild 3). Allein das Anklicken der einzelnen Schaltflächen führt noch zu einer Fehlermeldung. Kein Wunder: Wir haben auch noch keine Callback-Prozeduren für das Attribut **onAction** hinterlegt. Die Basis, die wir im **Office RibbonX Editor** mit einem Klick auf die Schaltfläche **Generate Callbacks** erstellen lassen können, sieht wie folgt aus:

Texte übersetzen mit DeepL

Zum automatischen Übersetzen von Texten gibt es viele Anlässe. Vielleicht möchtest Du die Texte in einer Anwendung automatisch übersetzen lassen, damit Du selbst die Übersetzung nur noch prüfen und gegebenenfalls anpassen musst. Oder Du hast Texte in einer anderen Sprache, die Du gern in die deutsche Sprache übersetzen möchtest, um diese leichter lesen zu können. Wie auch immer: Es gibt zwar Dienste wie Google Translate, mit denen man das im Browser erledigen kann, aber wenn man viele oder umfangreiche Texte übersetzen lassen möchte, ist diese Lösung unbefriedigend. In diesem Fall bietet sich eine Automation des Vorgangs an. Und wie das geht, zeigen wir anhand eines der aktuell besten Übersetzungstools, nämlich DeepL. DeepL bietet ein API an, die wir per VBA oder mit anderen Programmiersprachen ansteuern können. Dieser Artikel stellt die Grundlagen dazu vor.

Übersetzen mit DeepL

Die erste Anlaufstelle für die Arbeit mit der API des Übersetzungstools **DeepL** ist die Webseite **https://www.deepl.com**. Hier finden wir gleich zwei Textfelder, deren linkes wir für die Eingabe des zu übersetzenden Textes nutzen können. Geben wir hier einen Text ein, erkennt DeepL automatisch die verwendete Sprache und übersetzt den Text in die Sprache, die für das rechte Textfeld ausgewählt ist. Das geht auch recht schnell, sodass das Ergebnis wie in Bild 1 aussieht.

Die DeepL-API

Mit einem Klick auf das Menü rechts oben finden wir schnell den Eintrag **API**. Klicken wir diesen an, landen wir auf einer Seite, die eine kostenlose Registrierung anbietet.

Wählen wir diese Option, landen wir auf der Seite mit den verschiedenen Angeboten (siehe Bild 2). Die gute Nachricht ist: die kostenlose Variante reicht zum Ausprobieren der API wie in diesem Artikel beschrieben völlig aus.

Wenn Du nicht auf mehr als 500.000 Zeichen im Monat kommst, wählst Du einfach diese Option – und wenn es mehr werden, wäre die zweite Option sinnvoll. Hier fallen allerdings nach aktuellem Stand 20

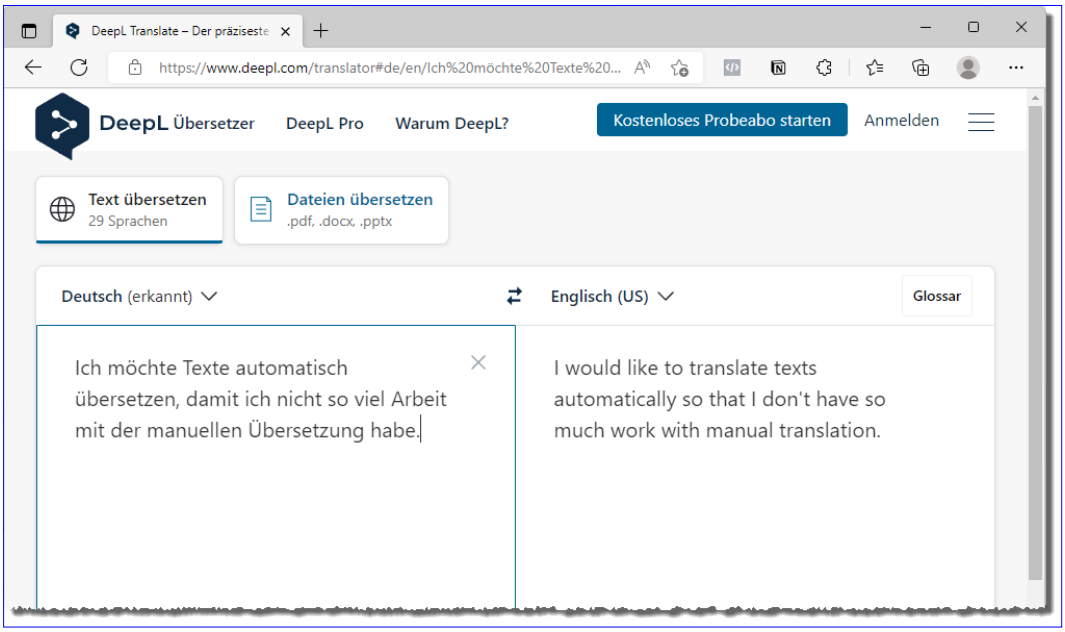


Bild 1: DeepL in Aktion

EUR pro 1.000.000 übersetzter Zeichen an. Wer einmal Texte von Hand übersetzt hat und einmal die Qualität der Übersetzung von DeepL geprüft hat, wird dies vermutlich als sehr faires Angebot ansehen. Aber wie gesagt: Für uns reicht die kostenlose Variante.

Zur Anmeldung benötigte DeepL Deine E-Mail-Adresse und Kennwort sowie einige wietere Informationen. Die Kreditkarteninformationen müssen angegeben werden, weil DeepL so Mehrfachanmeldungen und damit Missbrauch des kostenlosen Angebots umgehen möchte – es ist also eher eine vereinfachste Identitätsprüfung.

Nach der Prüfung des Kreditkartenkontos bestätigt man noch die Bedingungen und danach kann es schon losgehen.

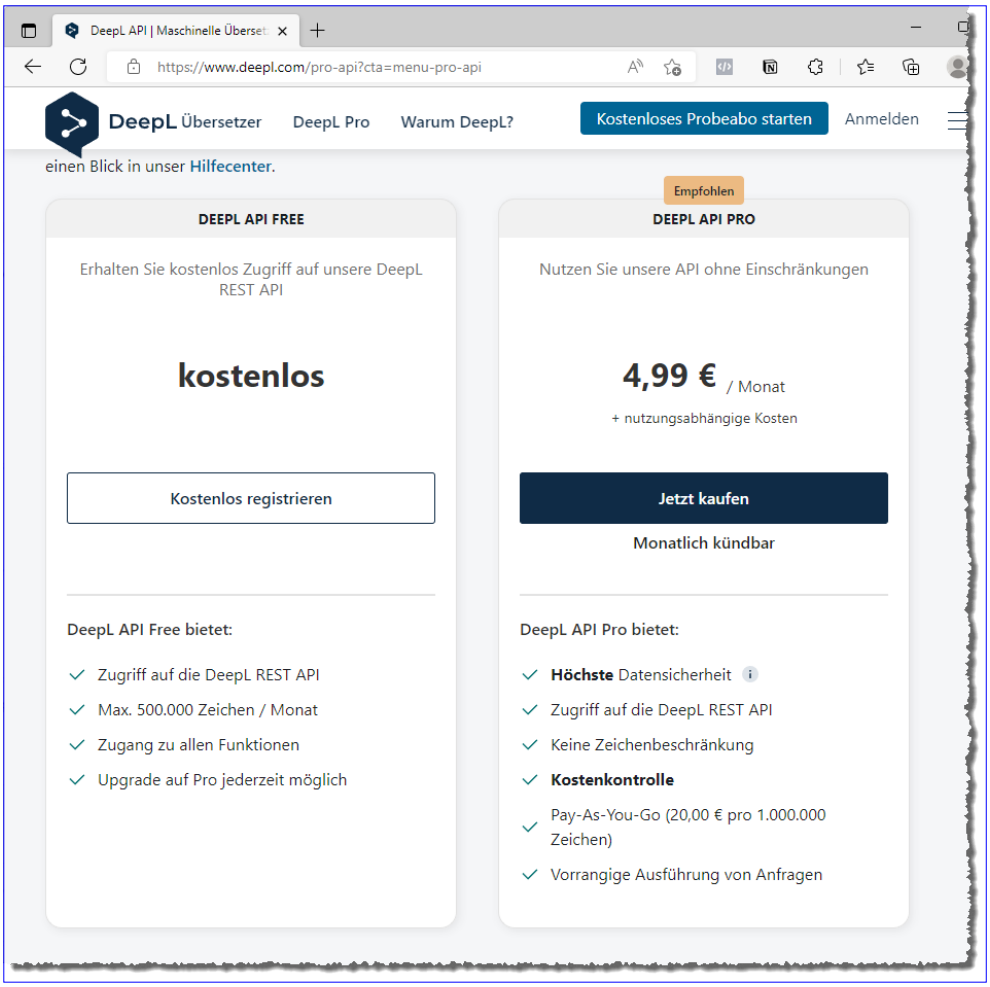


Bild 2: Angebot der DeepL-Api

DeepL-Konto verwalten

Im folgenden Schritt sehen wir eine Seite, von der aus wir uns unser Konto verwalten, aber auch einen Authentifizierungsschlüssel abrufen können.

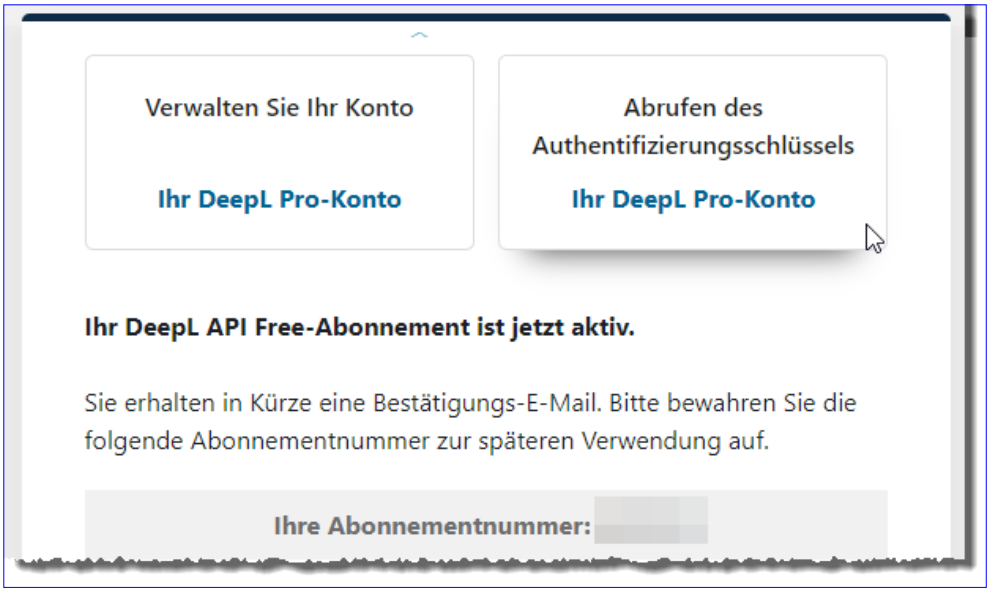


Bild 3: Link zum Authentifizierungsschlüssel

Letzteres ist genau das, was wir brauchen, um später per VBA auf die API von DeepL zuzugreifen (siehe Bild 3).

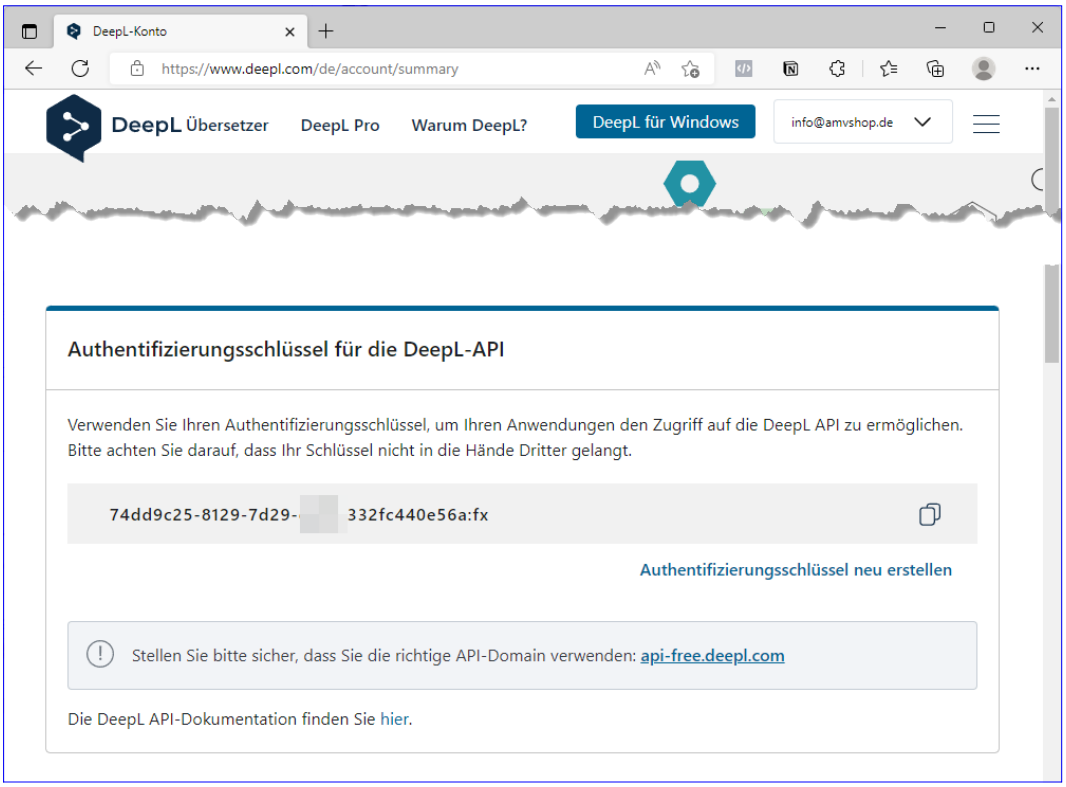


Bild 4: Holen des Authentifizierungsschlüsseld von DeepL

Den Authentifizierungsschlüssel finden wir dort schließlich im Bereich **Konto**. Diesen Schlüssel kannst Du schon einmal in die Zwischenablage kopieren und dann in einem VBA-Modul eine Konstante mit diesem füllen (siehe Bild 4).

Authentifizierungsschlüssel in Konstante speichern

Die Konstante zum Speichern des Authentifizierungsschlüssels sieht wie folgt aus:

```
Private Const cStrAuthKey As String = "74dd9c25-8129-7d29-
xxxx-332fc440e56a:fx"
```

In den nachfolgend vorgestellten Prozeduren holen wir den Wert dieser Konstanten mit der folgenden Funktion:

```
Public Function GetAuthKey()
    GetAuthKey = cStrAuthKey
End Function
```

Warum greifen wir nicht einfach auf die Konstante zu? Weil wir je nach Anwendung, also Access, Word, Excel et cetera, eine andere Möglichkeit wählen können, um den Authentifizierungsschlüssel zu speichern. In Access würden wir diesen beispielsweise in einer Optionentabelle speichern, unter Excel vielleicht in einem separaten Worksheet, in Word vielleicht wie hier vorgeschlagen in einer Konstanten.

Endpunkt in Konstante speichern

In einer weiteren Konstanten speichern wir die URL des zu verwendenden Endpunkts, also der Adresse, unter welcher die API zu erreichen ist.

Für die freie Version lautet dieser:

```
Private Const cStrAPIEndpoint As String = _
    "https://api-free.deepl.com"
```

Für die kostenpflichtige Version verwenden wir den folgenden Endpunkt:

```
Private Const cStrAPIEndpoint As String = _
    "https://api.deepl.com"
```

Funktion zum Übersetzen von Texten

Die Funktion, die wir zum Übersetzen von Texten aufrufen, heißt **TranslateDeepl** und sieht wie in Listing 1 aus. Sie erwartet die folgenden Parameter:

- **strToTranslate**: Zu übersetzender Text
- **intSourceLang**: Sprache des zu übersetzenden Textes
- **intTargetLang**: Sprache, in die der Text übersetzt werden soll

Der Parameter **intSourceLang** hat den Datentyp **SourceLanguage**. Dabei handelt es sich um eine Enumeration, die wir wie folgt im gleichen Modul definieren:

```
Public Enum SourceLanguage
    sGerman = 1
    sEnglish = 2
    sFrench = 6
    sItalian = 7
    sJapanese = 8
    sSpanish = 9
    sDutch = 10
    sPolish = 11
    sPortuguese = 12
```

```
Public Function TranslateDeepl(strToTranslate As String, intSourceLang As SourceLanguage, _
intTargetLang As TargetLanguage) As String
    Dim strRequest As String
    Dim strResponse As String
    Dim strAuth_Key As String
    Dim strText As String
    Dim strSource_Lang As String
    Dim strTarget_Lang As String
    Dim strStatus As String
    Dim col As Collection

    strAuth_Key = "?auth_key=" & GetAuthKey
    strText = "&text=" & URLEncode(strToTranslate)
```

Excel: Übersetzungen mit DeepL

Im Artikel »Texte übersetzen mit DeepL« (www.vbentwickler.de/322) haben wir gezeigt, wie man mithilfe einer in der Basisversion kostenlosen Web-API Übersetzungen von Texten durchführen kann. Das Ergebnis dieses Artikels war jedoch nur eine VBA-Funktion, mit der sich ein als Parameter angegebener Text übersetzen lässt. Das ist für Otto Normalverbraucher natürlich etwas sperrig, sodass wir diese Funktion nun einmal mit in einer Excel-Tabelle gespeicherten Texten ausprobieren wollen. Gleichzeitig lernen wir etwas über das Auslesen und Schreiben von Inhalten aus und in Excel-Tabellen.

DeepL-Code zum Workbook hinzufügen

Als Erstes fügen wir den benötigten Code zum VBA-Projekt des Workbooks hinzu, in welchem wir die Funktion zum Übersetzen einsetzen wollen. Dazu öffnen wir einfach den VBA-Editor, am schnellsten mit der Tastenkombination **Alt + F11**, und ziehen dann die Datei **mdlDeepL.bas** aus dem Download zu diesem Artikel in den Projekt-Explorer des VBA-Editors. Das Ergebnis sieht anschließend wie in Bild 1 aus – das Modul landet im Ordner **Module**.

Inhalt einer Zelle übersetzen

Wenn wir nun den Inhalt einer Zelle übersetzen wollen, brauchen wir nur die Funktion **TranslateDeepl**, die wir im Modul **mdlDeepL** finden, auf den Inhalt dieser Zelle anzuwenden und das Ergebnis in eine weitere Zelle zu schreiben.

Dabei starten wir mit der Konstellation aus Bild 2. Der Text aus der Zelle **A2** soll übersetzt und in die Zelle **B2** geschrieben werden.

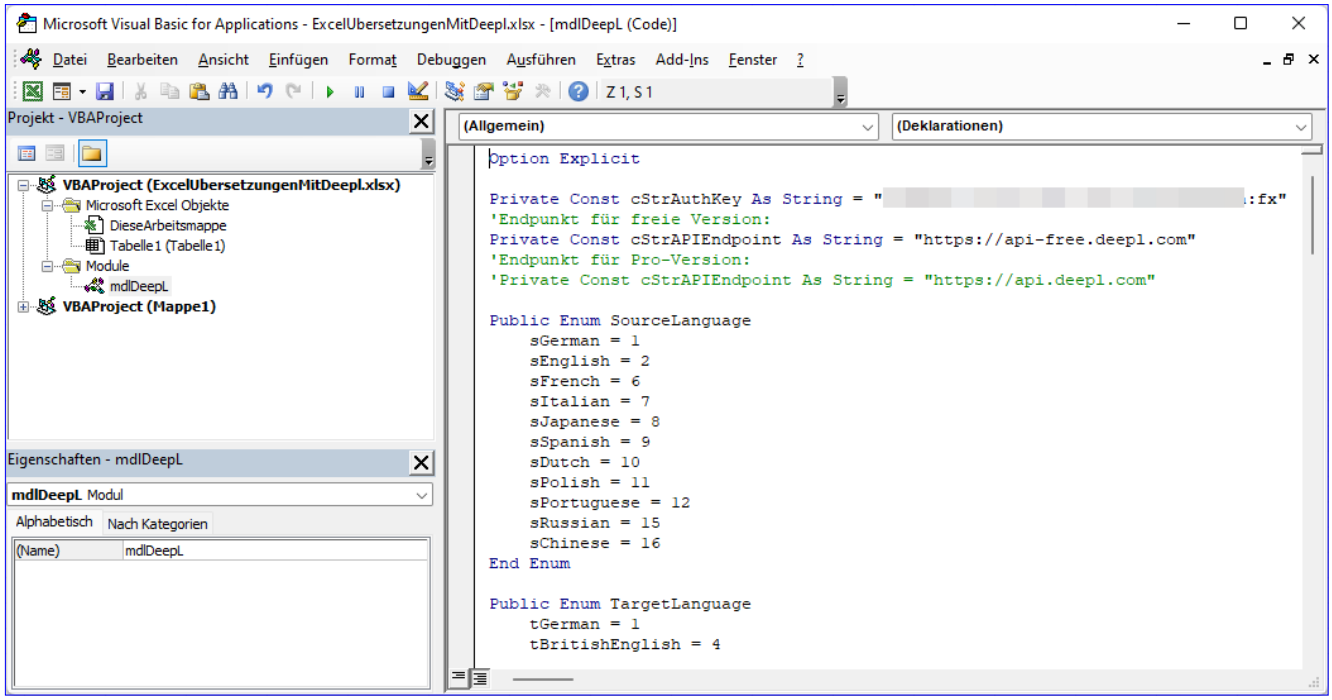


Bild 1: Eingefügtes Modul mit den Übersetzungs-Funktionen

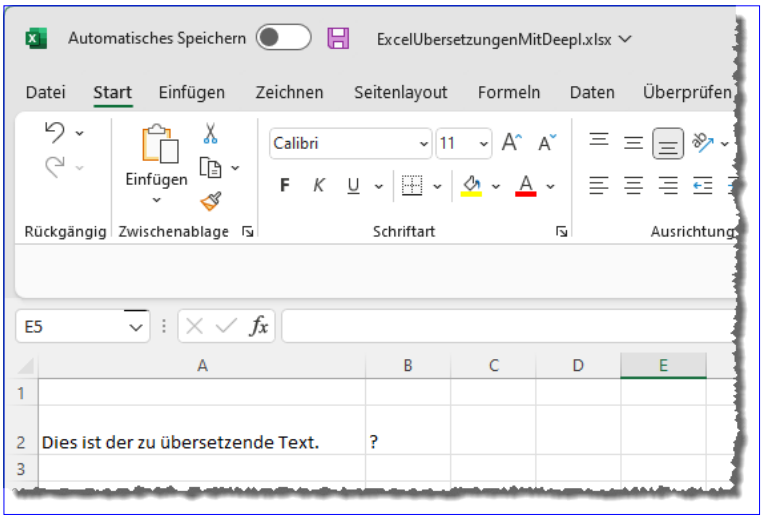


Bild 2: Dieser Text soll übersetzt werden.

Wir steuern dies zunächst über den VBA-Editor. Hier legen wir die folgende Prozedur an:

```
Public Sub Uebersetzen()
    Dim strOriginal As String
    Dim strUebersetzung As String
    strOriginal = ActiveSheet.Cells(2, 1)
    strUebersetzung = TranslateDeepl(strOriginal, _
        sGerman, tBritishEnglish)
    ActiveSheet.Cells(2, 2) = strUebersetzung
End Sub
```

Die Prozedur liest zuerst den Inhalt der Zelle aus der zweiten Zeile der ersten Spalte ein und schreibt diesen in die Variable **strOriginal**.

Dann ermittelt sie mit dem Aufruf der Funktion **TranslateDeepL** die Übersetzung des Textes und schreibt diese in die Variable **strUebersetzung**.

Dabei übergibt sie den Originaltext aus **strOriginal** als ersten Parameter sowie Konstanten für die Original- und die Zielsprache als zweiten und dritten Parameter. Das Ergebnis aus der Variablen **strUebersetzung** schreibt die Prozedur dann in die zweite Zeile der zweiten Spalte. Mehr brauchen wir nicht!

Und die Prozedur lässt sich auch komplett ohne Variablen schreiben:

```
Public Sub Uebersetzen()
    ActiveSheet.Cells(2, 2) = TranslateDeepl( _
        ActiveSheet.Cells(2, 1), _
        sGerman, tBritishEnglish)
End Sub
```

Längere Texte übersetzen

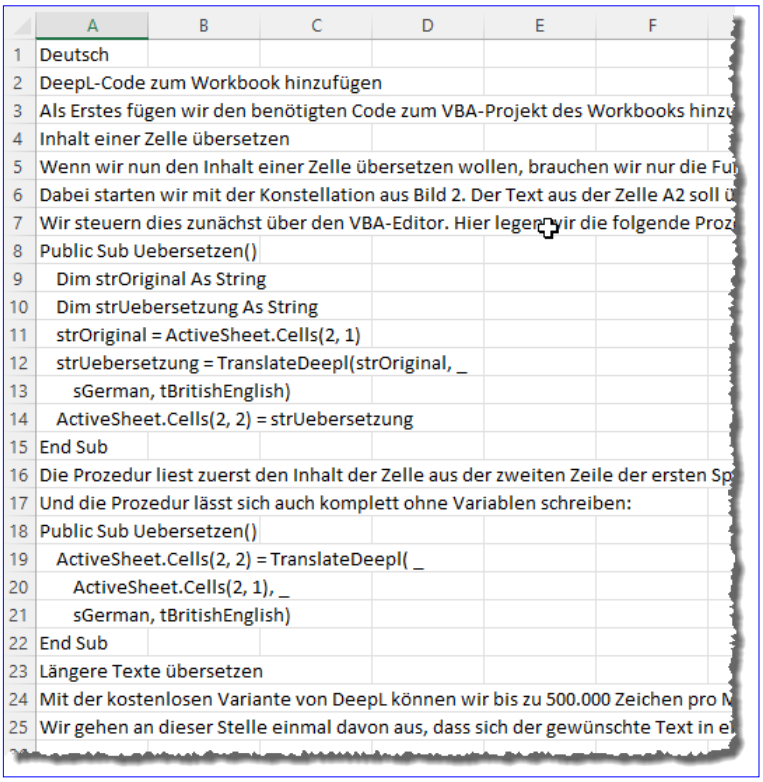
Mit der kostenlosen Variante von DeepL können wir bis zu 500.000 Zeichen pro Monat übersetzen. Das entspricht mehr als 300 Seiten dieses Magazins. Wir können also durchaus auch mal einen längeren Text übersetzen lassen. Wenn Du diesen wie im obigen Beispiel einfach in die Zelle A2 einfügst und die Prozedur startest, wird die Übersetzung des Textes in die Zelle rechts daneben eingetragen.

Wir gehen an dieser Stelle aber einmal davon aus, dass sich der gewünschte Text in einem anderen Dokument befindet und wir diesen erst einmal in geeigneter Form in die Zeilen einer Spalte der Excel-Tabelle bringen müssen.

Am besten so, das jeweils kleinere Häppchen in jeder Zelle landen – so können wir kleinere Unterschiede bei der Länge der verschiedenen Übersetzungen leicht ausgleichen.

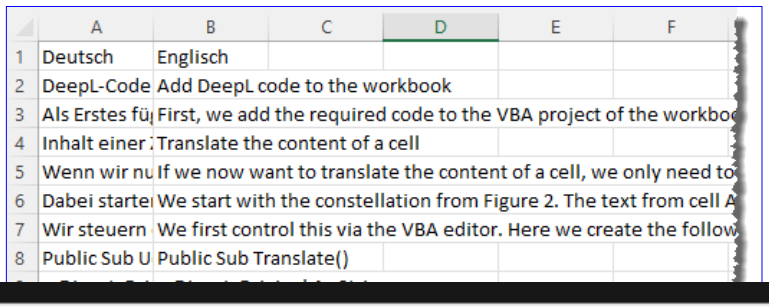
Wir haben den bisherigen Inhalt dieses Artikels einmal in die Zwischenablage kopiert und in die Zelle **A2** eingefügt und da unser Text Zeilenumbrüche enthält, wird er direkt absatzweise auf diese und die darunter liegenden Zellen aufgeteilt – im Beispiel bis zur Zelle **A25**. Das Ergebnis sieht danach etwa wie in Bild 3 aus.

Um die Absätze in den einzelnen Zellen Stück für Stück zu übersetzen, benötigen wir eine **For...Next**-Schleife.



	A	B	C	D	E	F
1	Deutsch					
2	DeepL-Code zum Workbook hinzufügen					
3	Als Erstes fügen wir den benötigten Code zum VBA-Projekt des Workbooks hinzu					
4	Inhalt einer Zelle übersetzen					
5	Wenn wir nun den Inhalt einer Zelle übersetzen wollen, brauchen wir nur die Fu					
6	Dabei starten wir mit der Konstellation aus Bild 2. Der Text aus der Zelle A2 soll ü					
7	Wir steuern dies zunächst über den VBA-Editor. Hier legen wir die folgende Proz					
8	Public Sub Uebersetzen()					
9	Dim strOriginal As String					
10	Dim strUebersetzung As String					
11	strOriginal = ActiveSheet.Cells(2, 1)					
12	strUebersetzung = TranslateDeepl(strOriginal, _					
13	sGerman, tBritishEnglish)					
14	ActiveSheet.Cells(2, 2) = strUebersetzung					
15	End Sub					
16	Die Prozedur liest zuerst den Inhalt der Zelle aus der zweiten Zeile der ersten Sp					
17	Und die Prozedur lässt sich auch komplett ohne Variablen schreiben:					
18	Public Sub Uebersetzen()					
19	ActiveSheet.Cells(2, 2) = TranslateDeepl(_					
20	ActiveSheet.Cells(2, 1), _					
21	sGerman, tBritishEnglish)					
22	End Sub					
23	Längere Texte übersetzen					
24	Mit der kostenlosen Variante von DeepL können wir bis zu 500.000 Zeichen pro M					
25	Wir gehen an dieser Stelle einmal davon aus, dass sich der gewünschte Text in ei					

Bild 3: Worksheet direkt nach dem Einfügen



	A	B	C	D	E	F
1	Deutsch	Englisch				
2	DeepL-Code	Add DeepL code to the workbook				
3	Als Erstes fü	First, we add the required code to the VBA project of the workbo				
4	Inhalt einer	Translate the content of a cell				
5	Wenn wir nu	If we now want to translate the content of a cell, we only need to				
6	Dabei starte	We start with the constellation from Figure 2. The text from cell A				
7	Wir steuern	We first control this via the VBA editor. Here we create the follow				
8	Public Sub U	Public Sub Translate()				

Diese läuft in der ersten Version von **l** gleich **2** bis **25**, weil die zu übersetzenden Texte sich in den so nummerierten Zeilen befinden.

Wir können die Anweisung aus dem vorherigen Beispiel weiterverwenden, wenn wir den Bezug auf die zu referenzierende Zelle mit der Laufvariablen **l** versehen. Das Ergebnis sieht wie folgt aus:

```
Public Sub MehrereZellenUebersetzen()
    Dim i As Long
    For i = 2 To 25
        ActiveSheet.Cells(i, 2) = _
            TranslateDeepl( _
            ActiveSheet.Cells(i, 1), _
            sGerman, tBritishEnglish)
    Next i
End Sub
```

Rufen wir diese Prozedur auf, durchläuft diese alle Zeilen von **2** bis **25** und schreibt die mit **TranslateDeepL** ermittelte Übersetzung zu dem Text aus der ersten Spalte jeweils in die rechts daneben befindliche Spalte ein. Das Ergebnis siehst Du in Bild 4.

Eine erste denkbare Optimierung ist nun die