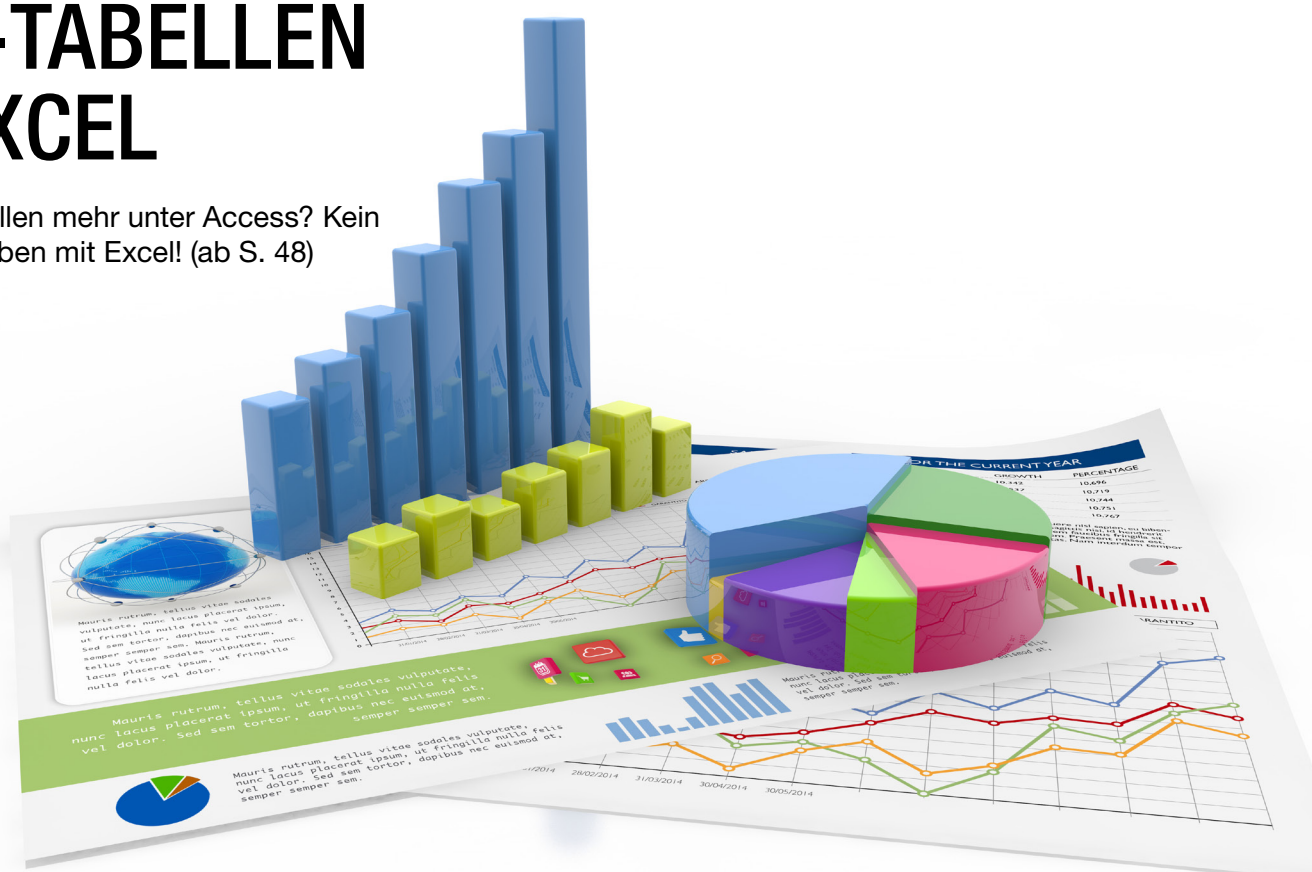
HAUFE.
Ausgabe 01/2021

ACCESS
IM UNTERNEHMEN

PIVOT-TABELLEN MIT EXCEL

Keine Pivot-Tabellen mehr unter Access? Kein Problem, dann eben mit Excel! (ab S. 48)



In diesem Heft:

SUCHKRITERIEN ZUSAMMENSTELLEN

Stellen Sie SQL-Ausdrucke einfach per VBA zusammen.

SEITE 29

RIBBON-ANZEIGE IM GRIFF

Zeigen Sie das Ribbon in der gewünschten Ansicht an.

SEITE 9

AUTOKORREKTUR FLEXIBEL NUTZEN

Lernen Sie, wie Sie die Autokorrektur per VBA programmieren

SEITE 2

Pivot-Tabellen und -Diagramme

Microsoft hat Access in den letzten Jahren um viele wichtige Funktionen »erleichtert«. Nach dem Sicherheitssystem und der Replikation finden wir seit Access 2013 auch die Pivot-Tabellen und -Diagramme nicht mehr als Ansichten in Access. Immerhin bietet Excel diese für die Aufbereitung und Analyse von Daten wichtigen Tools nach wie vor an. Und da Excel sich von Access aus gut steuern lässt, nutzen wir diese Funktionen in dieser Ausgabe für die Aufbereitung von Daten aus einer Access-Datenbank.

Excel bietet fast die gleichen Möglichkeiten an, die wir noch in Access 2010 zur Erstellung von Pivot-Tabellen und Pivot-Diagrammen gefunden haben. Wir schauen uns in dieser Ausgabe an, welche Möglichkeiten es gibt, um die Daten aus den Tabellen einer Access-Datenbank unter Excel in Pivot-Tabellen oder -Diagrammen darzustellen.

Der Beitrag **Pivot-Tabellen und -Diagramme in Excel** zeigt dabei ab Seite 48, wie Sie eine in einer Access-Datenbank bereitgestellte Abfrage von Access aus exportieren und als Datenquelle für eine Pivot-Tabelle nutzen können. Aufbauend auf der von Access aus erstellten Excel-Datei mit den Daten dieser Abfrage bauen wir in Excel eine Pivot-Tabelle auf. Basierend auf dieser Tabelle gehen wir noch einen Schritt weiter und legen ein Pivot-Diagramm für die Daten an.

Um dem Benutzer einige Schritte auf dem Weg von der Access-Abfrage bis zur Pivot-Tabelle in Excel zu ersparen, verwenden wir im Beitrag **Pivot-Tabellen und -Charts automatisch erstellen** ab Seite 60 die Programmiersprache VBA. Damit exportieren wir die Daten der Access-Abfrage per Mausklick und erzeugen auf die gleiche Weise eine Excel-Datei mit der gewünschten Pivot-Tabelle.

ParamArrays sind eine praktische Einrichtung, wenn Sie eine noch nicht bekannte Anzahl an Parametern an eine Funktion übergeben wollen. Alles rund um ihre Verwendung mit dem Schlüsselwort **ParamArray** lesen Sie im Beitrag **ParamArray-Auflistungen in VBA nutzen** ab Seite 41.

Die Autokorrektur-Funktion kennen viele nur als nervende Einrichtung, die an unerwünschten Stellen Texte verschlimmbessert. Der Beitrag **Autokorrektur-Einträge im Griff** zeigt ab Seite 2, wie die Autokorrektur genau funktioniert und wie Sie diese steuern können. Schließlich gehen wir noch einen Schritt weiter und zeigen, wie Sie eigene Einträge hinzufügen und damit sogar Abkürzungen wir **mfg** automatisch in Texte wie **Mit freundlichen Grüßen** umwandeln.

Das Ribbon macht manchmal, was es will – mal ist es maximiert, mal minimiert. Das ist unpraktisch, wenn Ihre Anwendung wichtige Funktionen im Ribbon anbietet und der Benutzer diese gar nicht standardmäßig zu sehen bekommt. Der Beitrag **Ribbon-Anzeige im Griff** zeigt ab Seite 9, wie Sie das Ribbon in Ihren Anwendungen so anzeigen, wie Sie es wünschen – und welche Fallstricke es noch zu beachten gibt.

Schließlich zeigen wir Ihnen im Beitrag **Add-In-Tools für den Formularentwurf** (ab Seite 34), wie Sie per Add-In schnell Steuerelemente mit durchnummerierten Bezeichnungen wie **txt01**, **txt02** und so weiter versehen.

Viel Spaß beim Lesen!

Ihr André Minhorst

Autokorrektur-Einträge im Griff

Die Autokorrektur ist ein Office-weit verwendetes Tool. Sie funktioniert auch bei der Eingabe von Texten in Access-Steuerelementen. Aber kann man die Einträge der Autokorrektur auch anpassen – und wo werden diese überhaupt gespeichert? Und kann man auch per VBA auf die enthaltenen Daten zugreifen und diese gegebenenfalls automatisiert erweitern? All diese Fragen klären wir im vorliegenden Beitrag.

Wem passiert es nicht einmal, dass er einen Tippfehler beim Eingeben von Daten in das Textfeld eines Access-Formulars macht? Oder auch in Excel, Word und Co.? Wenn das in Access passiert, weil Sie beispielsweise das Wort **wechler** statt **welcher** eingeben, korrigiert die Autokorrektur dies automatisch und zeigt auch an, dass sie aktiv war. Das erkennen Sie wie in Bild 1 an der aufklappbaren Schaltfläche, welche verschiedene Optionen anbietet:

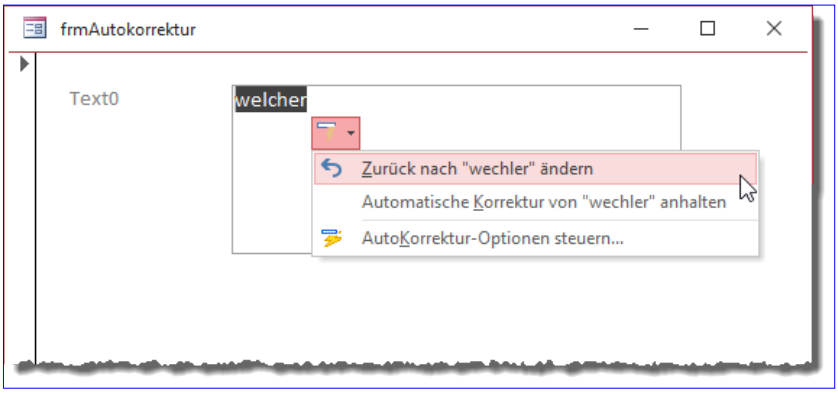


Bild 1: Die Autokorrektur in Aktion

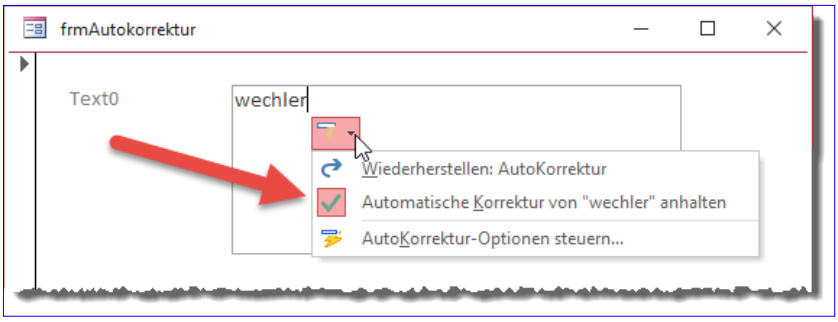


Bild 2: Die Autokorrektur wurde für dieses Wort deaktiviert.

- **Zurück nach "wechler" ändern**: Macht die durch die Autokorrektur vorgenommene Änderung rückgängig.
- **Automatische Korrektur von "wechler" anhalten**: Macht die Änderung ebenfalls wieder rückgängig und sorgt außerdem dafür, dass die Autokorrektur bei diesem Wort nicht mehr aktiv wird.
- **AutoKorrektur-Optionen steuern**: Öffnet einen Dialog, mit dem Sie die Optionen für die Autokorrektur ändern können.

Wenn Sie die Autokorrektur für ein Wort deaktivieren, erscheint vor diesem Eintrag ein Haken-Symbol (siehe Bild 2).

Autokorrektur-Optionen ändern

Wenn Sie die dritte Option wählen, zeigt Access den Dialog aus Bild 3 an. Hier finden Sie zunächst einige allgemeine Optionen, mit denen oft auftretende allgemeine Fehleingaben korrigiert werden können – zum Beispiel die Eingabe von zwei großen Zeichen am Wortanfang, weil man schneller tippt, als man die Umschalttaste wieder losgelassen hat.

Darunter finden Sie eine Liste der voreingestellten durch die Autokorrektur zu ersetzenden Begriffe. Links ist der zu

korrigierende Wert, rechts die Korrektur. Sie können mit diesem Dialog verschiedene Aktionen durchführen:

- Hinzufügen neuer Einträge: Dazu geben Sie die gewünschten Werte in die Felder **Ersetzen** und **Durch** ein und klicken auf die Schaltfläche **Hinzufügen**.

- Löschen vorhandener Einträge: Markieren Sie den zu löschenden Eintrag in der Liste und betätigen Sie die **Löschen**-Schaltfläche.

Aber Vorsicht: Diese Änderungen wirken sich auf die Autokorrektur für alle Office-Anwendungen für den aktuellen Benutzer aus. Wenn Sie also alle Einträge entfernen, wird auch Word keine Autokorrekturen mehr vornehmen.

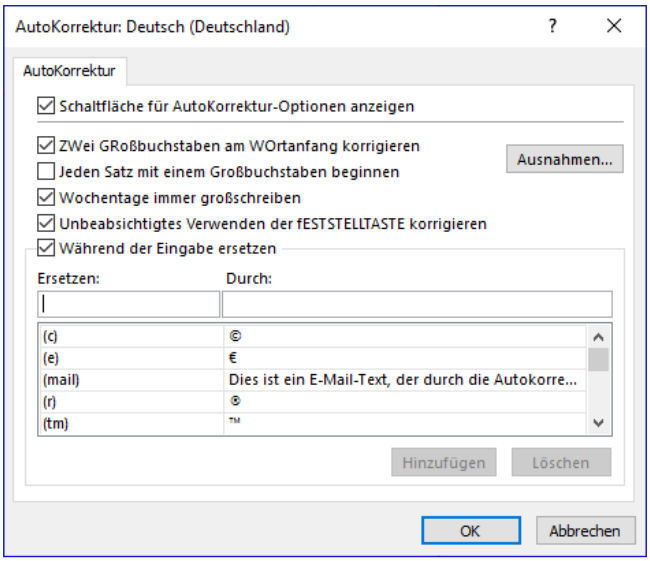


Bild 3: Optionen der Autokorrektur

Autokorrektur per VBA anpassen

Wir sind bekannt dafür, dass wir alles, was mit VBA ferngesteuert werden kann, auch unter die Lupe nehmen.

Dazu schauen wir uns zunächst die Möglichkeiten im Objektkatalog an. Geben Sie hier **Autocorrect** als Suchbegriff ein, erhalten Sie alle interessanten Elemente (siehe Bild 4).

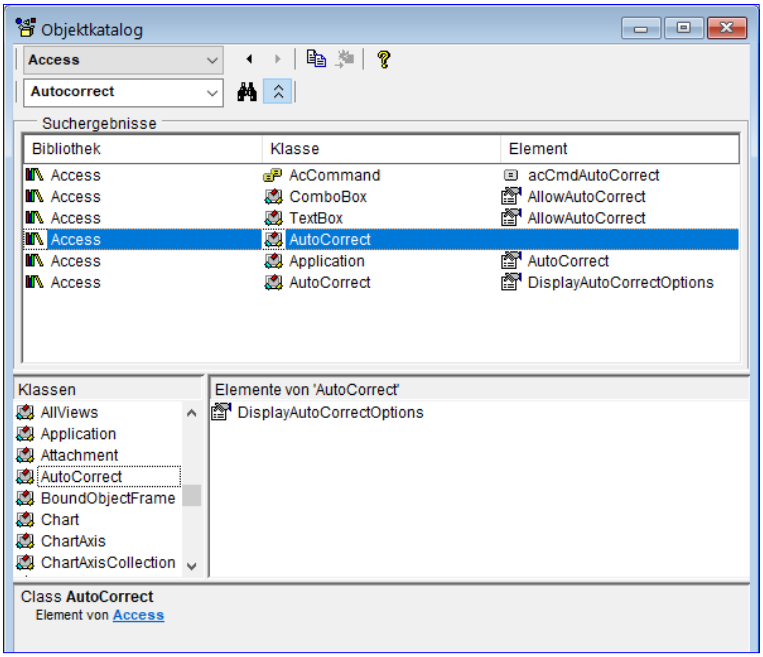


Bild 4: Das Autocorrect-Objekt im Objektkatalog

Das **AutoCorrect**-Objekt enthält selbst nur ein Element, nämlich **DisplayAutoCorrectOptions**. Auf den ersten Blick könnte man meinen, das wäre eine Methode, mit der sich der Optionen-Dialog für die Autokorrektur-Einstellungen öffnen lässt. Tatsächlich handelt es sich um eine Eigenschaft. Sie nimmt die Werte **True** oder **False** entgegen. Stellen wir die Eigenschaft testweise durch Eingabe der folgenden Anweisung im Direktbereich des VBA-Editors auf **False** ein:

```
AutoCorrect.DisplayAutoCorrectOptions = False
```

Dies führt dazu, dass die Autokorrektur zwar noch ausgeführt wird, allerdings erscheint das SmartTag mit den Optionen nicht mehr.

Neben dieser Eigenschaft gibt es noch die RunCommand-Konstante **acCmdAutoCorrect**. Diese öffnet wiederum

den Optionen-Dialog, wenn Sie einen Aufruf wie den folgenden nutzen:

```
RunCommand acCmdAutoCorrect
```

Schließlich gibt es noch die Eigenschaft **AllowAutoCorrect** für die beiden Steuerelementtypen **TextBox** und **ComboBox**. Damit können Sie festlegen, ob die Autokorrektur in einem bestimmten Textfeld aktiviert sein soll. Für das aktuelle Steuerelement deaktivieren wie die Autokorrektur beispielsweise wie folgt im Direktbereich des VBA-Editors:

```
Screen.ActiveControl.AllowAutoCorrect = False
```

Danach arbeitet die Autokorrektur einfach nicht mehr.

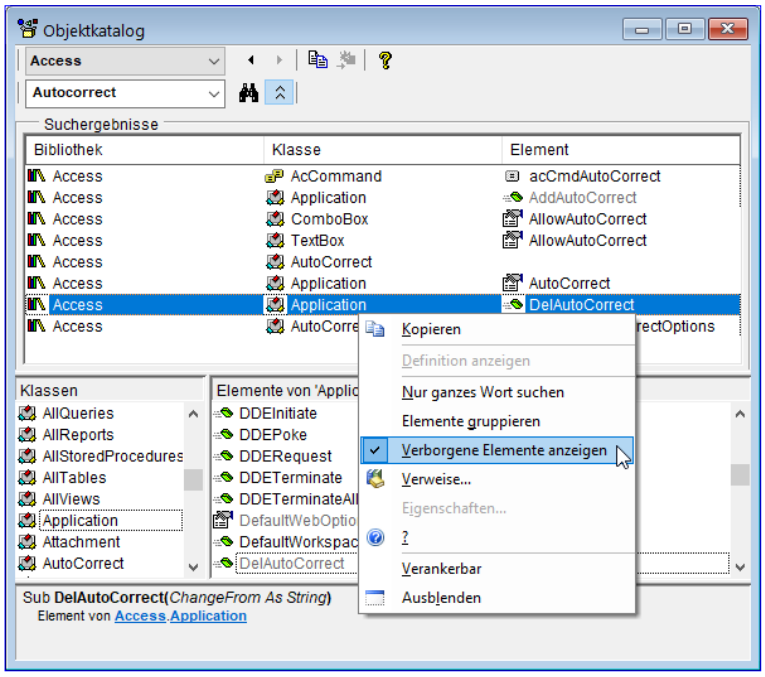


Bild 5: Einblenden verborgener Einträge

Autokorrekturen per VBA hinzufügen und löschen

Wir vermissen bisher noch eine Möglichkeit, per VBA Einträge zur Liste der Autokorrekturen hinzuzufügen. Auf herkömmlichem Wege können wir auch lange suchen, denn die beiden dazu benötigten Methoden sind als verborgen gekennzeichnet. Um diese sichtbar zu machen, klicken wir im Objektkatalog mit der rechten Maustaste in den Bereich **Suchergebnisse**. Im nun erscheinenden Kontextmenü wählen wir den Eintrag **Verborgene Elemente anzeigen** aus (siehe Bild 5).

Daraufhin erscheinen in den Suchergebnissen zwei neue Einträge:

- **AddAutoCorrect**: Fügt einen Autokorrekt-Eintrag hinzu.
- **DelAutoCorrect**: Löscht einen Autokorrekt-Eintrag.

Die beiden Befehle tauchen nun auch unter IntelliSense auf, wenn Sie **Application** gefolgt von einem Punkt im Direktbereich oder im Codefenster eingeben (siehe Bild 6).

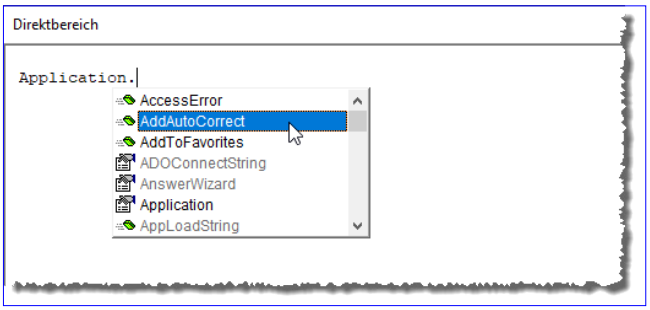


Bild 6: Befehl zum Hinzufügen eines Autokorrektur-Eintrags

Um einen Eintrag hinzuzufügen, geben Sie für den ersten Parameter der **AddAutoCorrect**-Methode den zu ersetzenden Ausdruck und für den zweiten den neuen Ausdruck ein:

```
Application.AddAutoCorrect "daß", "dass"
```

Den neuen Eintrag finden Sie dann nach dem Öffnen der Autokorrektur-Optionen in der Liste vor (siehe Bild 7).

Um diesen Eintrag wieder zu entfernen, verwenden Sie die Methode **DelAutoCorrect**. Diesmal brauchen Sie nur den zu ersetzenden Ausdruck einzugeben:

Ribbon-Anzeige im Griff

Das Ribbon von Microsoft Access macht meist, was es soll – es bietet die im aktuellen Kontext benötigten Befehle zur Auswahl an. Manchmal zickt es aber anscheinend rum: Es verschwindet oder wird nur beim Überfahren der Registerreiter sichtbar. Keine Frage: Meist sitzt das Problem vor dem Rechner und hat dieses Verhalten irgendwie angestoßen. Nur: Wie bekomme ich das Ribbon wieder in den Griff? Dieser Beitrag erläutert, welche verschiedenen Zustände das Ribbon haben kann und wie Sie diese über die Benutzeroberfläche und per VBA einstellen beziehungsweise wiederherstellen können.

Standardansicht des Ribbons

Normalerweise erscheint das Ribbon wie in Bild 1. Sie können die verschiedenen Register beziehungsweise Tabs per Klick anzeigen und diese werden dauerhaft angezeigt.

Es kann jedoch auch sein, dass nur die Leiste mit den Tab-Überschriften erscheint (siehe Bild 2). Diese Anzeige ist recht praktisch, wenn nicht viel Platz auf dem Bildschirm verfügbar ist. Sie klicken dann auf einen der Reiter, um die Befehle des jeweiligen Tabs einzublenden. Nachdem Sie den gewünschten Befehl aufgerufen haben, verschwindet die Leiste mit den Ribbon-Befehlen wieder.

Ribbon anheften

Wenn Sie das Ribbon dauerhaft sehen möchten, finden Sie nach dem Ausklappen auf der rechten Seite ein Pin-Symbol. Überfahren Sie dieses mit der Maus, erscheint die Beschreibung, die besagt, dass sich die Befehlsleiste mit dieser Schaltfläche oder mit der Tastenkombination **Strg + F1** anheften lässt (siehe Bild 3).

Mit der Tastenkombination **Strg + F1** können Sie die Leiste mit den Ribbon-Befehlen auch wieder in den nicht angehefteten Zustand bringen. Dieser Vorgang nennt sich reduzieren – über die Benutzeroberfläche stellen sie

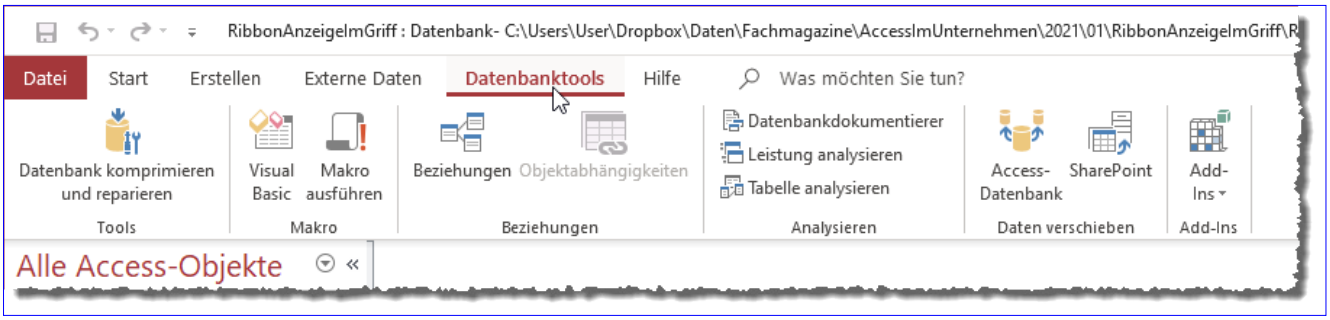


Bild 1: Standardansicht des Ribbons

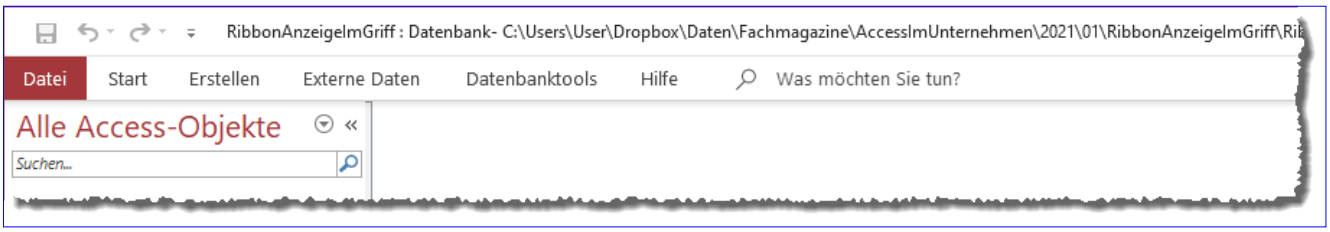


Bild 2: Ribbon minimiert

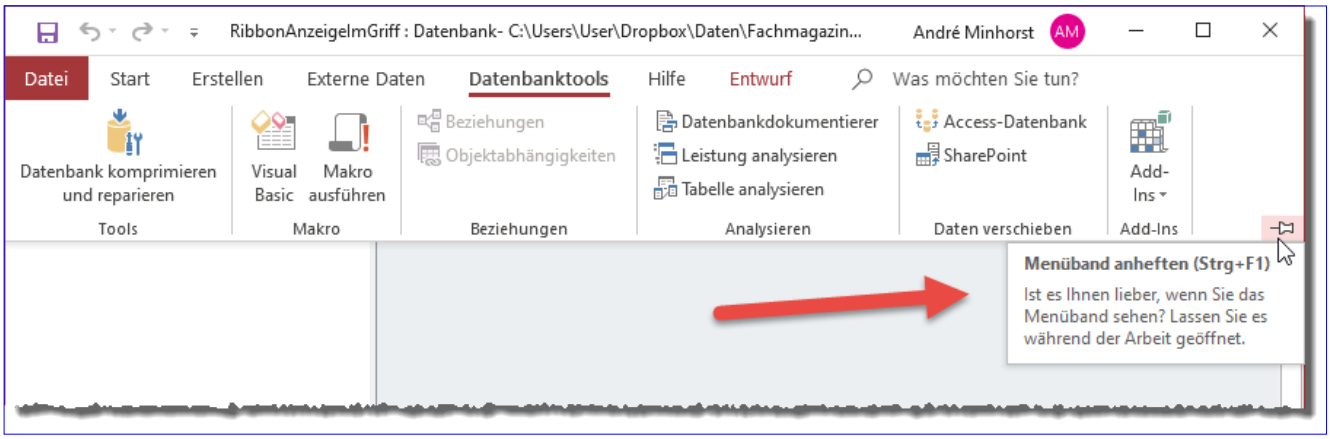


Bild 3: Ausgeklapptes, nicht angeheftetes Ribbon mit der Möglichkeit zum anheften

diesen Zustand her, indem Sie auf den kleinen Pfeil nach oben klicken, der sich im angehefteten Zustand rechts unten im Ribbon befindet.

Ein weiterer Weg, das Ribbon in den angehefteten Zustand zu überführen, ist ein Doppelklick auf einen der Tab-Reiter des Ribbons. Andersherum können Sie ein angeheftetes Ribbon per Doppelklick auf einen der Tab-Reiter auch wieder reduzieren.

Anheften und reduzieren per VBA

Es kann Gründe geben, warum Sie in Ihrer Anwendung den angehefteten oder den reduzierten Zustand herstellen möchten – beispielsweise damit der Benutzer sicher immer alle Ribbon-Befehle sieht und diese nicht verborgen sind. Die Einstellung ist Access-spezifisch, das heißt, dass diese nicht geändert wird, wenn Sie eine Anwendung schließen und die nächste mit Access wieder öffnen.

Wenn der Benutzer also normalerweise aus Platzmangel mit dem reduzierten Ribbon arbeitet und Sie dieses maximieren wollen, um die Funktionalität Ihrer Anwendung sicherzustellen, können Sie das per VBA erledigen. Sie sollten den vorherigen Zustand beim Schließen der Anwendung allerdings wieder herstellen.

Anheften und Reduzieren per Minimieren

Hier taucht eine kleine Ungereimtheit auf. Der folgende Befehl versetzt das Ribbon in den reduzierten Zustand, wenn es angeheftet war:

```
CommandBars.ExecuteMso "MinimizeRibbon"
```

Das ist noch logisch. Sie können den Befehl jedoch auch verwenden, um das Ribbon vom reduzierten in den angehefteten Zustand zu versetzen!

Wenn wir also zuverlässig den angehefteten oder den reduzierten Zustand herstellen wollen, müssen wir vorher abfragen, in welchem Zustand sich das Ribbon gerade befindet. Dafür gibt es wiederum keinen eingebauten Befehl, sondern wir nutzen die Tatsache, dass das Ribbon als **CommandBar**-Objekt referenzierbar ist, und zwar über **CommandBars("Ribbon")**. Machen Sie sich keine Hoffnungen: Das Ribbon ist nun nicht über das **CommandBar**-Objektmodell zu bearbeiten. Das **CommandBar** namens **Ribbon** enthält lediglich ein Control mit dem Wert **Ribbon** in der Eigenschaft **Caption**.

Nützlich ist jedoch, dass die **Height**-Eigenschaft dieses **Control**-Objekts je nach Zustand unterschiedliche Werte zurückliefert. Haben wir das Ribbon zuvor in den angehefteten Zustand versetzt, liefert **Height** unter Access 365 den folgenden Wert:

```
? Commandbars("Ribbon").Controls(1).Height
 161
```

Stellen wir den reduzierten Zustand her, erhalten wir dieses Ergebnis:

```
? Commandbars("Ribbon").Controls(1).Height
 65
```

Diese Werte ändern sich gelegentlich. Unter Access 2010 liefert der reduzierte Zustand beispielsweise den Wert **54** und der angeheftete Zustand den Wert **142**.

Man könnte also annehmen, dass der Wert im reduzierten Zustand unter 100 liegt und im angehefteten Zustand über 100. Allerdings haben wir die Rechnung gemacht, ohne die Möglichkeit zu betrachten, dass man die Schnellzugriffsleiste über und unter dem Ribbon platzieren kann.

Diesen Zustand ändern Sie, indem Sie in der Schnellzugriffsleiste auf das Symbol für das Aufklappmenü klicken und dort den Eintrag **Unter dem Menüband anzeigen** auswählen (siehe Bild 4).

Wenn die Schnellzugriffsleiste sich unter dem Ribbon befindet, liefert **CommandBars("Ribbon").Controls(1).Height** ganz andere Werte, nämlich **100** für den reduzierten Zustand und **196** für den angehefteten Zustand (Access 365)! Unter Access 2010 liegen die Werte bei **78** und **168**.

Wir können uns also bei der Ermittlung des aktuellen Zustandes nicht auf eine einfache Funktion wie die folgende verlassen, die von einem bestimmten Wert ausgeht:

```
Public Function MenuebandAngeheftet() As Boolean
    If CommandBars("Ribbon").Controls(1).Height > 100 Then
        MenuebandAngeheftet = True
    End If
End Function
```

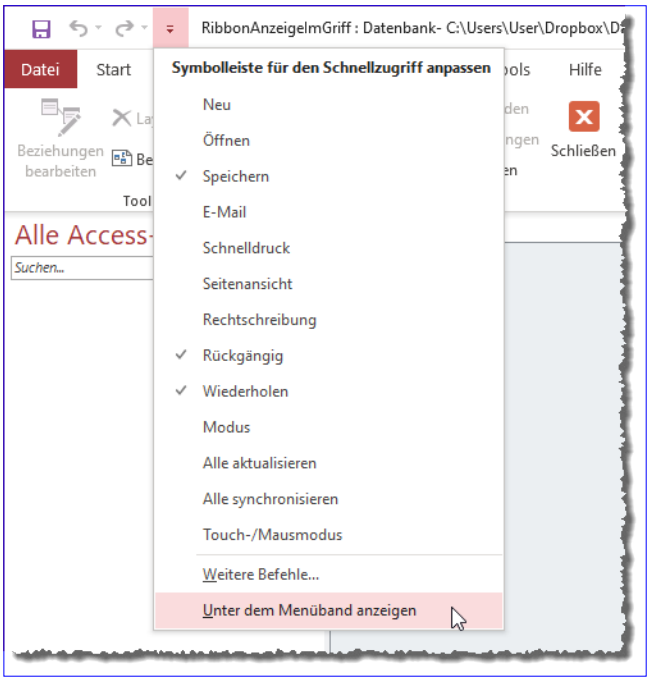


Bild 4: Einstellen, ob die Schnellzugriffsleiste über oder unter dem Ribbon angezeigt werden soll

Wo befindet sich die Schnellzugriffsleiste aktuell?

Stattdessen müssen wir noch ermitteln, ob die Schnellzugriffsleiste gerade über oder unter dem Ribbon angezeigt wird und das in die Funktion einbeziehen.

Aber Google lieferte keine Erkenntnisse darüber, wo wir eine Einstellung finden, mit der wir die Position der Schnellzugriffsleiste, auch Quick Access Toolbar genannt, einstellen können. Schließlich haben wir es auf gut Glück in der Registry gesucht und dort **QuickAccessToolbar** als Suchbegriff eingegeben. Das hat den Eintrag aus Bild 5 geliefert. Der passende Registry-Zweig lautet:

```
HKEY_CURRENT_USER\SOFTWARE\Microsoft\Office\16.0\Common\
Toolbars\Access
```

Die Abbildung zeigt die Einstellung für die Position über dem Ribbon. Wenn wir die Schnellzugriffsleiste wie oben beschrieben unter das Ribbon verschieben, finden wir in

Metadaten per Zusatztabelle verwalten

Im Laufe der Zeit können sich einen Menge Daten ansammeln, die Sie in Zusammenhang mit Kunden, Produkten oder ähnlichen Entitäten speichern wollen. Dabei möchten Sie vielleicht nicht für jede neue Eigenschaft ein neues Feld anlegen und damit den Tabellenentwurf ändern. Das gilt umso mehr, wenn eine Anwendung bereits von vielen Kunden genutzt wird. Es gibt jedoch eine Alternative: Sie können eine zusätzliche Tabelle hinzufügen, die in jedem Datensatz eine Eigenschaft mit dem jeweiligen Wert für die Entität speichert. Die Frage ist nur: Wie können wir diese Daten genauso nutzen, als wenn diese wie üblich in der gleichen Tabelle wie der Kunde oder das Produkt gespeichert werden?

Nehmen wir an, Sie möchten einer Kundentabelle zwei neue Felder hinzufügen, mit denen Sie festlegen, wann der Kunde sich für den Newsletter angemeldet hat und wann er sich wieder abgemeldet hat. Sie wollen aber nicht die Kundentabelle anpassen, weil Sie das in der Vergangenheit schon öfter gemacht haben und dies immer mit viel Aufwand verbunden ist.

Stattdessen wollen Sie eine Lösung schaffen, mit der Sie langfristig zwar nach Bedarf neue Felder hinzufügen können, aber nicht jedesmal den Entwurf der Kundentabelle ändern müssen.

Lösung per Extra-Tabelle

Die Lösung ist eine zusätzliche Tabelle, welche ausschließlich Metadaten zu den Elementen der Haupttabelle speichert. Wie muss eine solche Tabelle aussehen, wenn wir dort nach Wunsch verschiedene Attribute mit ihren Werten hinzufügen wollen?

Anforderungen an die Metadaten-Tabelle

Die erste Anforderung ist, dass die Daten der Metadaten-Tabelle den Einträgen der Haupttabelle zugeordnet werden können müssen. Die Metadaten-Tabelle, nennen wir Sie hier **tblKundenMeta**, erhält also neben dem Primärschlüsselfeld **KundeMetaID** ein Fremdschlüsselfeld namens **KundeID**. Außerdem wollen wir den Namen des Attributs und den Wert des Attributs in der Tabelle speichern. Dazu legen wir zwei weitere Felder namens **Attributname** und **Attributwert** an. Dem Feld **Attributname** weisen wir logischerweise den Felddatentyp **Kurzer Text** zu.

Aber welchen Datentyp soll das Feld **Attributwert** erhalten – immerhin sollen dort gegebenenfalls Daten mit verschiedenen Datentypen gespeichert werden? In diesem Fall können wir nur den Datentyp **Kurzer Text** wählen, da alle anderen Datentypen Einschränkungen haben – zum Beispiel können Sie Zahlenfelder nicht mit Texten füllen, aber umgekehrt schon.

Jedes Attribut nur einmal pro Datensatz?

Grundsätzlich sollten wir für die Kombination des Fremdschlüsselfeldes (hier **KundeID**) und **Attributname** einen eindeutigen Index festlegen, damit jedes Attribut für jeden Kunden nur einmal angelegt werden kann – so, wie es auch in einem einfachen Feld direkt in der Kundentabelle der Fall ist.

Andererseits stellt sich die Frage, ob es nicht auch Anwendungsfälle gibt, in denen mehrere Werte für ein Attribut benötigt werden, die unter dem gleichen Attributnamen abrufbar sein sollen. Dann wäre ein zusammengesetzter, eindeutiger Index nicht sinnvoll. Und was, wenn es zwar Felder gibt, die nur einmal vorkommen dürfen, andere aber mehrmals? Dies müssten wir in der

Prozedur prüfen und behandeln, mit der wir Daten zu der Meta-Tabelle hinzufügen.

Datenmodell von Haupt- und Metatabelle

Für unser Beispiel sieht das Datenmodell wie in Bild 1 aus. Die Beziehung zwischen den Tabellen wird über das Fremdschlüsselfeld **KundeID** der Tabelle **tblKundenMeta** hergestellt. Für diese Beziehung legen wir referentielle Integrität mit Löschweitergabe fest.

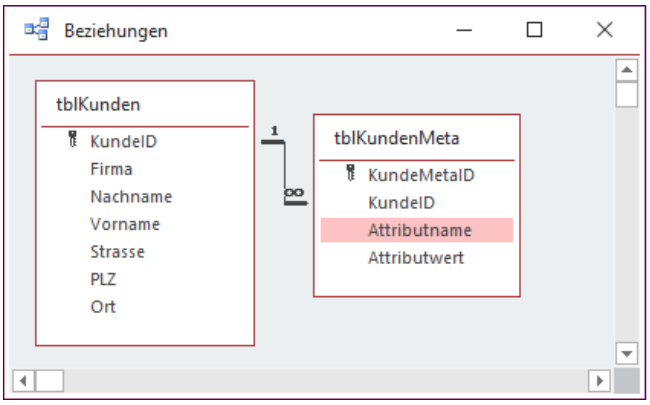


Bild 1: Datenmodell mit Metatabelle

Wenn der Benutzer dann einen Datensatz der Tabelle **tblKunden** löscht, werden die damit verknüpften Datensätze der Tabelle **tblKundenMeta** automatisch mitgelöscht.

Bearbeiten der Metadaten über die Benutzeroberfläche

Die flexibelste Art, die Metadaten in einem Formular anzuzeigen, ist ein Unterformular, dass alle Datensätze der Tabelle **tblKundenMeta** anzeigt, die zu dem im Hauptformular gehörenden Datensatz der Tabelle **tblKunden** gehören. Zum Erstellen des Unterformulars gehen Sie wie folgt vor:

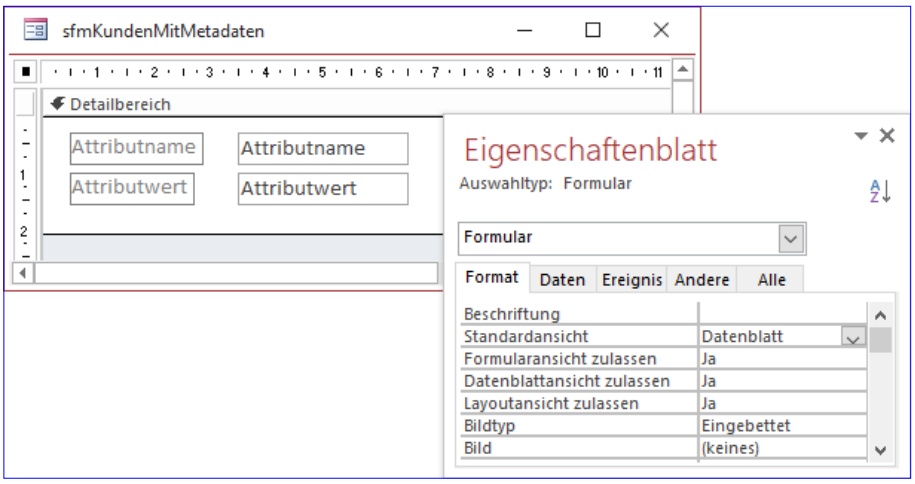


Bild 2: Entwurf des Unterformulars **sfmKundenMitMetadaten**

- Legen Sie ein neues Formular namens **sfmKundenMitMetadaten** an.
- Stellen Sie die Eigenschaft **Datensatzquelle** auf die Tabelle **tblKundenMeta** ein.
- Ziehen Sie die Felder **Attributname** und **Attributwert** in den Detailbereich des Entwurfs.
- Stellen Sie die Eigenschaft **Standardansicht** auf **Datenblatt** ein.
- Schließen und speichern Sie das Unterformular.

Den Entwurf des Unterformulars mit den Eigenschaften finden Sie in Bild 2.

Das Hauptformular legen Sie wie folgt an:

- Legen Sie ein neues Formular namens **frmKundenMitMetadaten** an.
- Stellen Sie die Eigenschaft **Datensatzquelle** auf **tblKunden** ein.
- Ziehen Sie alle Felder der Tabelle in den Entwurf.
- Ziehen Sie das Formular **sfmKundenMitMetadaten** aus dem Navigationsbereich in den Formularentwurf.

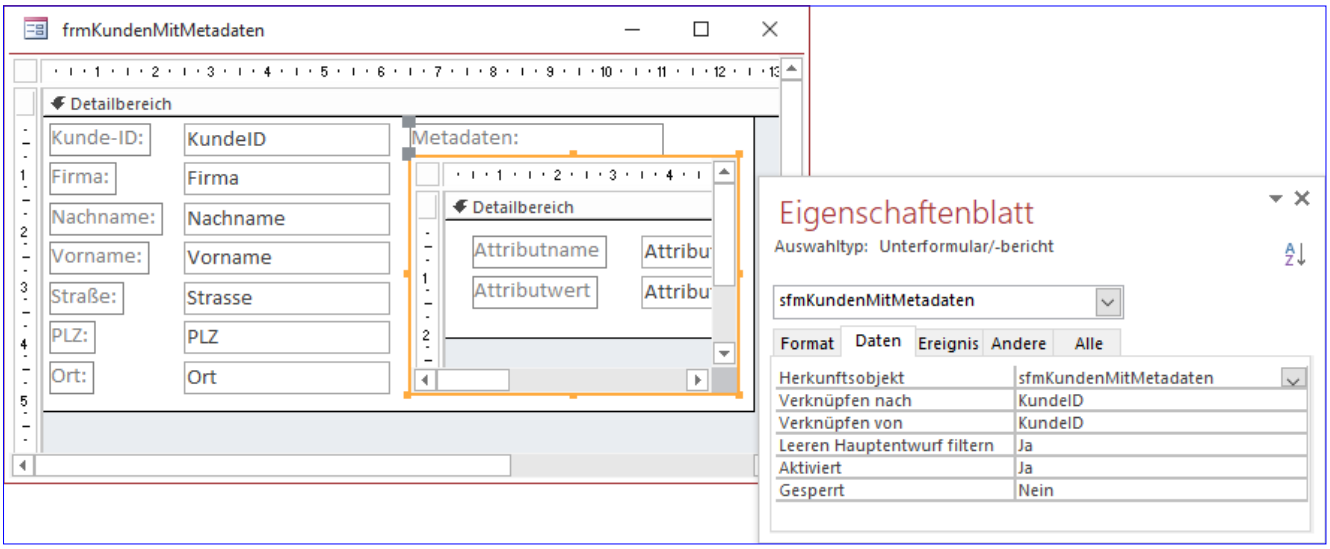


Bild 3: Entwurf des Hauptformulars **frmKundenMitMetadaten**

- Speichern und schließen Sie das Formular.

Wenn die Tabellen korrekt verknüpft sind und Sie nun das Unterformular-Steuerelement markieren, zeigen die Eigenschaften **Verknüpfen von** und **Verknüpfen nach** beide den Wert **KundeID** an (siehe Bild 3).

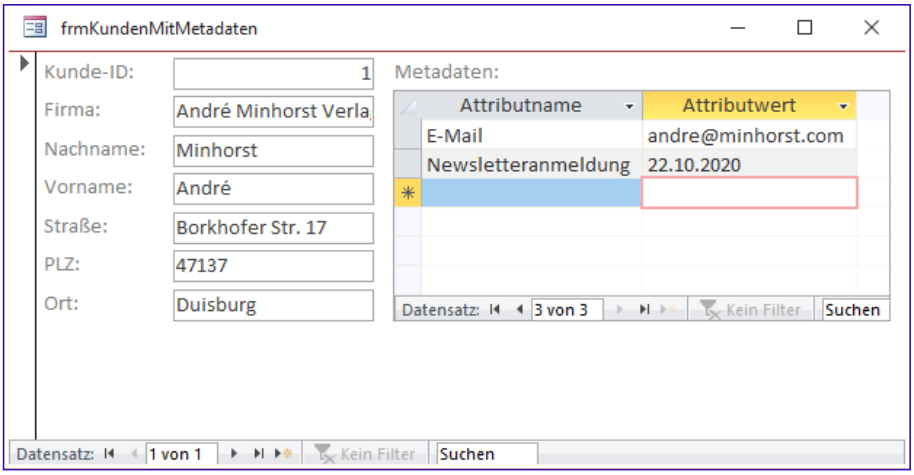


Bild 4: Eingabe von Attributen und Attributwerten

Attribute eingeben

Wechseln Sie in die Formularansicht des Hauptformulars, können Sie erste Daten eintragen. Beginnen Sie mit denen auf der linken Seite. Danach fügen wir rechts zuerst einen Attributnamen und dann den Wert ein. Auf diese Weise legen wir die Eigenschaften **E-Mail** und **Newsletteranmeldung** mit den gewünschten Weren (siehe Bild 4).

Alternative Eingabemöglichkeit

Wenn Sie nun entscheiden, dass Sie für neue Metadaten zwar keine Änderungen am Datenmodell vornehmen möchten, aber sehr wohl an der Benutzeroberfläche, wird es viel aufwendiger.

Aber warum sollte man Änderungen an der Benutzeroberfläche tolerieren, wenn man das Datenmodell nicht antasten möchte? Es kann beispielsweise sein, dass das Backend nicht so einfach angepasst werden kann, weil es beispielsweise ein SQL Server-Backend ist. Beim Frontend hingegen ist das kein Problem – wenn es vernünftig programmiert ist, können Sie dieses einfach ersetzen, ohne dass der Betrieb eingeschränkt wird.

Daher wollen wir nun die beiden neuen Attribute E-Mail und Newsletteranmeldung wie die Felder der Tabelle

tblKunden im Formular anzeigen. Dazu legen wir ein neues Formular namens **tblKundenMitMetafeldern** an.

Wenn wir eine Datensatzquelle für alle Felder erhalten wollen, benötigen wir eine entsprechend formulierte Abfrage. Diese sollte also nicht nur die Felder der Tabelle **tblKunden** enthalten, sondern auch noch zwei Felder namens **EMail** und **Newsletteranmeldung**.

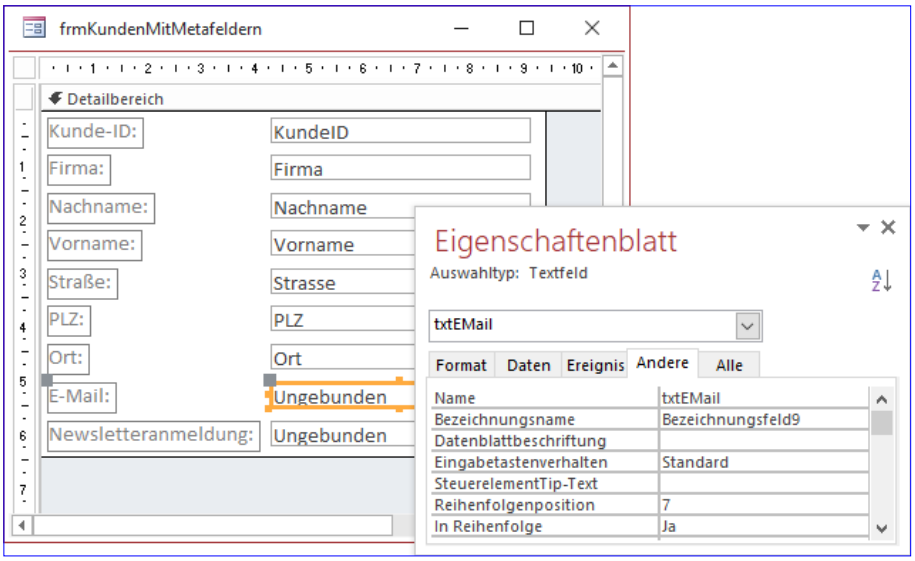


Bild 5: Ungebundene Felder für die Metadaten

Um es kurz zu machen: Eine solche Abfrage, die gleichzeitig auch noch aktualisierbar ist, lässt sich nicht erstellen. Aktualisierbar muss diese aber sein, weil wir auch Daten in das Formular eingeben wollen.

Also legen wir für die beiden Attribute **E-Mail** und **Newsletteranmeldung** ungebundene Textfelder an, die wir beim Anzeigen des Datensatzes mit den Werten der Tabelle **tblKundenMitMetadaten** füllen.

Wenn der Benutzer die Inhalte ändert, tragen wir die Änderungen per Code in diese Tabelle ein.

Die beiden Textfelder nennen wir **txtEMail** und **txtNewsletteranmeldung**. Wir fügen diese wie in Bild 5 in das Formular ein.

Meta-Attributnamen in die Marke-Eigenschaft schreiben

Für später tragen wir für die beiden Textfelder **txtEMail** und **txtNewsletteranmeldung** die jeweiligen Attributnamen in die Eigenschaft **Marke** ein, die Sie auf der Registerseite **Andere** des Eigenschaftenblatts finden.

Diese benötigen wir später für die VBA-Programmierung.

Anzeigen der Daten aus der Metadaten-Tabelle

Damit die Felder beim Anzeigen eines neuen Datensatzes gefüllt werden, hinterlegen wir eine Ereignisprozedur für das Ereignis **Beim Anzeigen**. Diese enthält zwei Anweisungen. Die erste verwendet die **DLookup**-Funktion, um den Wert des Feldes **Attributwert** des Datensatzes der Tabelle **tblKundenMeta** zu ermitteln, für den das Feld

```
Private Sub Form_Current()
    Me!txtEMail = Nz(DLookup("Attributwert", "tblKundenMeta", "KundeID = " & Nz(Me!KundeID, 0) _
        & " AND Attributname = 'E-Mail'"), "")
    Me!txtNewsletteranmeldung = Nz(DLookup("Attributwert", "tblKundenMeta", "KundeID = " & Nz(Me!KundeID, 0) _
        & " AND Attributname = 'Newsletteranmeldung'"), "")
End Sub
```

Listing 1: Einlesen der Metadaten und Schreiben in die Textfelder

KundeID dem aktuellen Kunden und das Feld **Attributname** dem gesuchten Attribut entspricht (siehe Listing 1).

Es kann vorkommen, dass gar nicht alle gesuchten Attribute in der Tabelle **tblKundenMeta** vorhanden sind. Daher fassen wir das Ergebnis der **DLookup**-Funktion noch in die **Nz**-Funktion ein, die eine leere Zeichenkette zurückliefert, wenn kein passender Datensatz gefunden werden konnte.

Schließlich wird dieser Prozedur auch aufgerufen, wenn das Formular einen neuen, leeren Datensatz anzeigt. In diesem Fall ist das Feld **KundeID** noch leer. Also fassen wir auch dieses mit der **Nz**-Funktion ein und liefern den Wert **0** zurück, wenn **KundeID** den Wert **Null** hat. Die zurückgelieferten Zeichenketten landen in jedem Fall in den beiden Feldern **txtEMail** und **txtNewsletterAngemeldet** (siehe Bild 6).

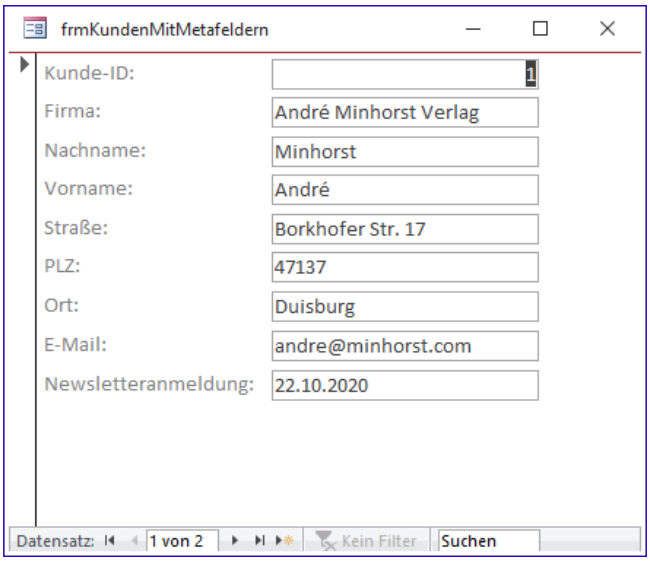


Bild 6: Die unteren beiden Textfelder enthalten Metadaten aus der verknüpften Tabelle **tblKundenMeta**.

```
Private Sub Form_AfterUpdate()
    Dim db As DAO.Database
    Dim lngKundeMetaID As Long
    Set db = CurrentDb
    lngKundeMetaID = Nz(DLookup("KundeMetaID", "tblKundenMeta", "KundeID = " & Me!KundeID _
        & " AND Attributname = 'E-Mail'"), 0)
    If lngKundeMetaID = 0 Then
        db.Execute "INSERT INTO tblKundenMeta(KundeID, Attributname, Attributwert) VALUES(" & Me!KundeID _
            & ", 'E-Mail', '" & Me!txtEMail & "')", dbFailOnError
    Else
        db.Execute "UPDATE tblKundenMeta SET Attributwert = '" & Me!txtEMail & "' WHERE KundeID = " & Me!KundeID _
            & " AND Attributname = 'E-Mail'", dbFailOnError
    End If
    lngKundeMetaID = Nz(DLookup("KundeMetaID", "tblKundenMeta", "KundeID = " & Me!KundeID _
        & " AND Attributname = 'Newsletteranmeldung'"), 0)
    If lngKundeMetaID = 0 Then
        db.Execute "INSERT INTO tblKundenMeta(KundeID, Attributname, Attributwert) VALUES(" & Me!KundeID _
            & ", 'Newsletteranmeldung', '" & Me!txtNewsletteranmeldung & "')", dbFailOnError
    Else
        db.Execute "UPDATE tblKundenMeta SET Attributwert = '" & Me!txtNewsletteranmeldung & "' WHERE KundeID = " _
            & Me!KundeID & " AND Attributname = 'Newsletteranmeldung'", dbFailOnError
    End If
End Sub
```

Listing 2: Schreiben der Metadaten

Ändern der Metadaten

Nun wollen wir sicherstellen, dass Änderungen an den Daten der beiden Textfelder **txtEMail** und **txtNewsletteranmeldung** auch in der Tabelle **tblKundenMeta** landen.

Das soll erst dann geschehen, wenn auch die Daten aus den übrigen Textfeldern gespeichert werden, also beim Speichern des Datensatzes. Dazu eignen sich die beiden Ereignisse **Vor Aktualisierung** und **Nach Aktualisierung** des Formulars.

Wir verwenden **Nach Aktualisierung**, weil der Datensatz der Tabelle **tblKunden** zum Zeitpunkt des Auslösens des Ereignisses **Vor Aktualisierung** noch nicht gespeichert ist (siehe Listing 2). Dann haben wir auch noch keinen Primärschlüsselwert eines neuen Datensatzes der Tabelle **tblKunden** und können folglich auch keine damit verknüpften Datensätze in der Tabelle **tblKundenMeta** anlegen.

In der Prozedur prüfen wir zunächst, ob es bereits einen Datensatz in der Tabelle **tblKundenMeta** gibt, der zum aktuellen Kunden gehört und das betroffene Attribut enthält.

Falls ja, landet der Primärschlüsselwert dieses Datensatzes der Tabelle **tblKundenMeta** in der Variablen **lngKundeMetaID**, anderenfalls der Wert **0**.

Hat **lngKundeMetaID** danach den Wert **0**, legen wir mit einer **INSERT INTO**-Abfrage einen neuen Datensatz in der Tabelle **tblKundenMeta** an, der das Attribut mit dem Wert aus dem Textfeld enthält. Falls nicht, aktualisieren wir den vorhandenen Datensatz für den Kunden und das Attribut mit einer **UPDATE**-Abfrage.

Das erledigen wir zuerst für das Feld **txtEMail** und dann für das Feld **Newsletteranmeldung**. Der Code ist recht redundant und kann ausgelagert und parametrisiert werden – dazu später mehr.

Zunächst prüfen wir, ob die Prozedur wie gewünscht funktioniert. Dazu legen wir einen neuen Datensatz an und speichern diesen dann beispielsweise durch den Wechseln zum nächsten neuen Datensatz, zum vorherigen Datensatz oder einfach durch Betätigen der Tastenkombination **Strg + S**.

Das funktioniert wie gewünscht – die Daten aus den beiden Textfeldern **txtEMail** und **txtNewsletteranmeldung** landen in zwei neuen Datensätzen der Tabelle **tblKundenMeta**.

Kein Speichern ohne »dreckigen« Datensatz

Nun probierern wir uns an einem vorhandenen Datensatz und ändern ein oder mehrere der gebundenen Felder sowie die beiden Textfelder **txtEMail** und **txtNewsletteranmeldung**. Auch das gelingt: Die geänderten Daten der beiden ungebundenen Textfelder landen in den entsprechenden Datensätzen der Tabelle **tblKundenMeta**.

Wenn wir allerdings nur die Daten von **txtEMail** und/oder **txtNewsletteranmeldung** ändern, werden die Änderungen nicht übernommen. Der Grund ist, dass wir den Datensatz nur durch das Ändern der ungebundenen Textfelder nicht in den bearbeiteten Zustand versetzen und somit auch das Ereignis **Nach Aktualisierung** beim Verlassen des Datensatzes nicht ausgelöst wird.

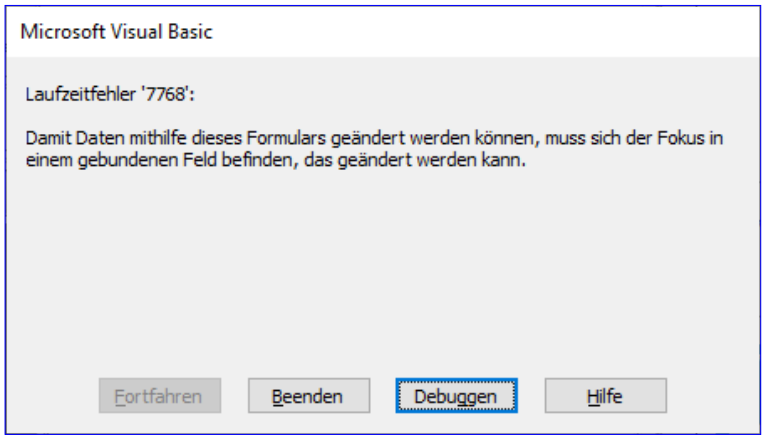


Bild 7: Die **Dirty**-Eigenschaft lässt sich nur setzen, wenn ein gebundenes Feld den Fokus hat.

Suchkriterien einfach zusammenstellen

Wenn Sie per VBA SQL-Suchkriterien zusammenstellen, also den Teil einer SQL-Abfrage, der mit WHERE beginnt, ist das in der Regel eine Menge Schreibarbeit. Eine Menge Schreibarbeit geht oft einher mit Fehleranfälligkeit. Sie benötigen Code, um zu schauen, ob überhaupt ein Vergleichswert eingegeben wurde, müssen dann die verschiedenen Vergleichsausdrücke zusammenstellen, diese mit OR oder AND verknüpfen, dabei unterschiedliche Datentypen beachten und davon abhängig Hochkommata setzen oder auch Datumsangaben konvertieren. All dies können Sie sich sparen, wenn Sie ein paar einfache Hilfsfunktionen einsetzen.

Wenn Sie ein Suchformular mit Steuerelementen zur Eingabe mehrerer Kriterien ausstatten (wie in Bild 1), müssen Sie diese auch auswerten. Das geschieht normalerweise per VBA.

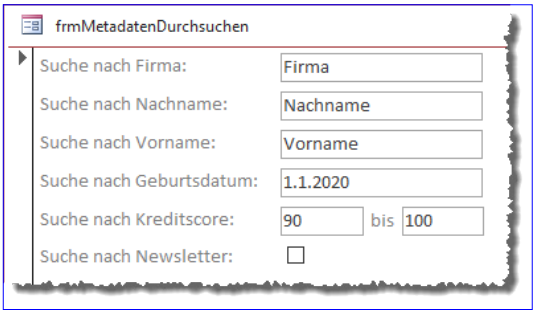


Bild 1: Beispiel-Suchformular

In einer Prozedur, die beispielsweise durch eine Suchen-Schaltfläche ausgelöst wird, stehen dann Anweisungen wie die aus Listing 1.

Hier deklarieren wir zu Beginn eine Variable namens **strWhere**, welche den Inhalt der **WHERE**-Klausel einer **SELECT**-Abfrage aufnehmen soll. Diese werden dann später zusammengesetzt und beispielsweise einem Unterformular oder einem Listenfeld als Datenquelle zugewiesen (über die Eigenschaft **Recordsource** oder **RowSource**).

In unserem Beispiel haben wir einige Felder, die einfache Texte abfragen wie **txtSucheNachFirma**. Andere fragen nach einem Datum wie **txtSucheNachGeburtsdatum** oder nach Zahlenwerten (**txtSucheNachKreditscoreBis**).

Für alle Suchsteuerelemente prüfen wir zunächst, ob dieses überhaupt einen Wert enthält (**Len(Nz(<Suchfeldname>, ""))**). Die **Nz**-Funktion wandelt **Null**-Werte in leere Zeichenfolgen um, die **Len**-Funktion ermittelt die Zeichenkettenlänge. Auf diese Weise prüfen wir in der **If... Then**-Bedingung jeweils, ob sich ein Suchbegriff im Steuerelement befindet. Nur dann führen wir den Teil innerhalb der Bedingung aus und fügen einen Teil zur Variablen **strWhere** hinzu:

```
If Not
Len(Nz(Me!txtSucheNachFirma, ""))
= 0 Then
    strWhere = strWhere & " AND
Firma LIKE '" 7
    & Me!txtSucheNachFirma & "'"
End If
```

Im ersten Fall fügen wir direkt einen Ausdruck hinzu, der mit **AND** beginnt, dann den Namen des zu untersuchenden Feldes (**Firma**), den Vergleichsoperator (**LIKE**) und den Vergleichswert (der Inhalt von **txtSucheNachFirma** in Hochkommata). Frage: Warum beginnen wir gleich den ersten Ausdruck mit **AND**, obwohl wir diesen Teil später, wenn wir diesen an **WHERE** anhängen, sowieso entfernen müssen? Der Grund ist einfach: Wenn wir allen Elementen der **Where**-Klausel ein **AND** voranstellen, können wir auch davon ausgehen, dass bei Vorhandensein mindestens eines Ausdrucks auch ein **AND** ganz vorn steht – und dieses dann einfach abschneiden.

Vergleichsausdrücke mit Datum sehen etwas anders aus. Der Grundaufbau ist zwar gleich, aber wir verwenden hier eine Funktion, um den Inhalt des Datumsfeldes, zum

```
Dim strWhere As String
If Not Len(Nz(Me!txtSucheNachFirma, "")) = 0 Then
    strWhere = strWhere & " AND Firma LIKE '" & Me!txtSucheNachFirma & "'"
End If
If Not Len(Nz(Me!txtSucheNachVorname, "")) = 0 Then
    strWhere = strWhere & " AND Vorname LIKE '" & Me!txtSucheNachVorname & "'"
End If
If Not Len(Nz(Me!txtSucheNachNachname, "")) = 0 Then
    strWhere = strWhere & " AND Nachname LIKE '" & Me!txtSucheNachNachname & "'"
End If
If Not Len(Nz(Me!txtSucheNachGeburtsdatum, "")) = 0 Then
    strWhere = strWhere & " AND Geburtsdatum = " & ISODatum(Me!txtSucheNachGeburtsdatum)
End If
If Not Len(Nz(Me!txtSucheNachKreditscoreVon, "")) = 0 Then
    strWhere = strWhere & " AND Kreditscore >= " & Replace(Me!txtSucheNachKreditscoreVon, ",", ".")
End If
If Not Len(Nz(Me!txtSucheNachKreditscoreBis, "")) = 0 Then
    strWhere = strWhere & " AND Kreditscore <= " & Replace(Me!txtSucheNachKreditscoreBis, ",", ".")
End If
If Not IsNull(Me!chkSucheNachNewsletter) Then
    strWhere = strWhere & " AND Newsletter = " & CInt(Me!chkSucheNachNewsletter)
End If
If Len(strWhere) > 0 Then
    strWhere = Mid(strWhere, 6)
    strWhere = " WHERE " & strWhere
End If
Debug.Print strWhere
```

Listing 1: Zusammensetzen einer Where-Bedingung

Beispiel **31.12.2020**, in einen universell einsetzbaren SQL-Datumsausdruck umzuwandeln, in diesem Fall **#2020/12/31 00:00:00#**. Dazu verwenden wir die Hilfsfunktion **ISODatum**, die Sie im Modul **mdlTools** finden:

```
If Not Len(Nz(Me!txtSucheNachGeburtsdatum, "")) = 0 Then
    strWhere = strWhere & " AND Geburtsdatum = " _
                & ISODatum(Me!txtSucheNachGeburtsdatum)
End If
```

Beim Datum geben wir im Gegensatz zum Auch hier stellen wir dem Teilausdruck ein **AND** voran und fügen diesen dann zum bereits vorhandenen Ausdruck **strWhere** hinzu. Wollen wir mit Zahlenfeldern vergleichen, benötigen wir keine Hochkommata, aber ähnlich wie bem Datum müssen wir berücksichtigen, dass die Zahlenwerte ein Komma enthalten können. Unter SQL wird aber immer der Punkt als Dezimaltrennzeichen verwendet. Das heißt, dass wir die **Replace**-Prozedur nutzen, um ein eventuell vorhandenes Komma durch einen Punkt als Dezimaltrennzeichen ersetzen:

```
If Not Len(Nz(Me!txtSucheNachKreditscoreVon, "")) = 0 Then
    strWhere = strWhere & " AND Kreditscore >= " _
        & Replace(Me!txtSucheNachKreditscoreVon, ",", ".")
End If
```

Schließlich müssen wir noch Kontrollkästchen betrachten. Je nach Anwendungsfall kann ein Kontrollkästchen in der deutschen Version Werte wie Ja/Nein oder Wahr/Falsch

Add-In-Tools für den Formularentwurf

Als ich neulich mal wieder einige Steuerelemente zu einem Formular hinzugefügt habe, die durchnummerierte Namen erhalten sollten wie txt01, txt02 und so weiter, kam die Eingebung: Warum diese langweilige Arbeit immer wieder von Hand erledigen, statt einfach ein Tool dafür zu entwickeln? Gesagt, getan: Es sollte ein kleines Add-In her, mit dem sich diese erste kleine Aufgabe vereinfachen ließ. Wie Sie vermutlich auch, programmiere ich nämlich lieber als dass ich immer wiederkehrende Aufgaben durchführe. Das Durchnummerieren von markierten Steuerelementen nach bestimmten Vorgaben soll die erste Funktion dieses Add-Ins sein. Ihnen und mir fallen sicher noch noch weitere Ideen für den Einsatz dieses Add-Ins ein!

Der Auslöser für die Programmierung des in diesem Beitrag beschriebenen Add-Ins ist das unscheinbare Formulare aus Bild 1. Die drei Kombinationsfelder sollen die Namen **cboSucheNach1**, **cboSucheNach2** und **cboSucheNach3** erhalten und die Textfelder rechts daneben die Namen **txtSucheNach1**, **txtSucheNach2** und **txtSucheNach3**.

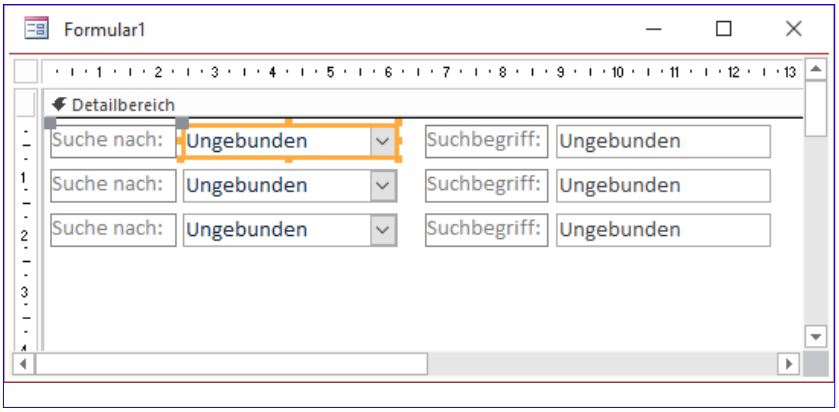


Bild 1: Zu benennende Steuerelemente

Darauf, das von Hand zu erledigen, hatte ich keine Lust. Also habe ich zuerst eine kleine Prozedur geschrieben, die wie in Listing 1 aussieht. Sie erwartet die folgenden Parameter, um die Benennung der Steuerelemente so flexibel wie möglich zu machen:

- **strBezeichnung**: Bezeichnung des Steuerelements inklusive eines Platzhalters, an dessen Stelle die Nummern eingetragen werden.
- **strPlatzhalter**: Platzhalter, der durch die Nummern ersetzt werden soll.
- **intStellen**: Anzahl der Stellen für die Durchnummerierung. Der Wert **2** würde beispielsweise bei den ersten Namen um eine führende **0** ergänzt werden, also beispielsweise **02**.
- **intStart**: Gibt die Nummer für das erste Steuerelement an. Standardwert ist **1**. Geben Sie einen anderen Wert an, wird die Nummerierung mit diesem begonnen.

Ein Aufruf dieser Prozedur sieht beispielsweise wie folgt aus:

```
SteuerelementeBenennen "cboSucheNach[]", "[]", 1, 1
```

Hier wird immer der Basisname **cboSucheNach[]** verwendet, bei dem die Zeichenfolge **[]** durch die aktuelle Zahl ersetzt wird – in diesem Fall mit einer Stelle, also ohne führende **0** bei den ersten Einträgen.

```
Public Sub SteuerelementeBenennen(strBezeichnung As String, strPlatzhalter As String, intStellen As Integer, _
        Optional intStart As Integer = 1)
    Dim frm As Form
    Dim ctl As Control
    Dim intNummer As Integer
    Dim strStellen As String
    intNummer = intStart
    Set frm = Screen.ActiveForm
    strStellen = String(intStellen, "0")
    For Each ctl In frm.Controls
        If ctl.InSelection Then
            ctl.Name = Replace(strBezeichnung, strPlatzhalter, Format(intNummer, strStellen))
            intNummer = intNummer + 1
        End If
    Next ctl
End Sub
```

Listing 1: Prozedur zum Benennen von Steuerelementen

Die Prozedur trägt zuerst den Wert der Startzahl aus dem Parameter **intStart** in die Variable **intNummer** ein. Dann referenziert sie das aktuell geöffnete Formular mit **Screen.ActiveForm**. Diese erste Fassung enthält noch keine Prüfung, ob beispielsweise überhaupt ein Formular geöffnet ist oder ob der Benutzer Steuerelemente zum Umbenennen markiert hat.

Dann verwendet die Prozedur den Wert aus dem Parameter **intStellen**, um in **strStellen** eine Zeichenkette mit so vielen Nullen zu füllen, wie es in **intStellen** angegeben ist. Damit starten wir in eine **For Each**-Schleife über alle Steuerelemente des mit **frm** referenzierten Formulars. In der Schleife prüfen wir zuerst in einer **If...Then**-Bedingung, ob das aktuell durchlaufene Steuerelement markiert ist. Dazu nutzen wir die Eigenschaft **InSelection**. In der **If...Then**-Bedingung stellen wir die Eigenschaft **Name** des mit **ctl** referenzierten aktuellen Steuerelements auf die gewünschte Bezeichnung ein. Diese ermitteln wir, indem wir den Platzhalter **strPlatzhalter** in **strBezeichnung** mit **Replace** ersetzen, und zwar durch den Wert aus der Variablen **intNummer** mit dem Format aus **strStellen**.

Prüfungen

Damit haben wir den größten Teil bereits erledigt. Bevor wir ein Add-In auf Basis dieser Prozedur anlegen, wollen wir jedoch noch zwei Prüfungen einbauen. Die erste soll untersuchen, ob überhaupt ein Formular in der Entwurfsansicht geöffnet ist. Dazu deklarieren wir eine Variable namens **bolEntwurfsansicht**:

```
Dim bolEntwurfsansicht As Boolean
```

Die Abfrage von Screen.ActiveForm fassen wir in **On Error Resume Next/On Error Goto 0** ein, um die Fehlerbehandlung für diese eine Anweisung zu deaktivieren. Der Grund ist, dass der Aufruf ohne geöffnetes Formular einen Fehler auslöst, den wir vermeiden wollen. Dann prüft die Prozedur, ob **frm** leer ist, was der Fall ist, wenn kein Formular geöffnet ist. Ist **frm** nicht leer, prüfen wir mit der Eigenschaft **CurrentView**, ob die aktuelle Ansicht die Entwurfsansicht ist. Falls ja, stellen wir die Variable **bolEntwurfsansicht** auf **True** ein:

```
...
On Error Resume Next
Set frm = Screen.ActiveForm
On Error Goto 0
```

```
If Not frm Is Nothing Then
    If frm.CurrentView = acCurViewDesign Then
        bolEntwurfsansicht = True
    End If
End If
```

Hat **bolEntwurfsansicht** danach noch den Wert **False** hat, ist das aktive Element entweder kein Formular oder dieses ist nicht in der Entwurfsansicht geöffnet. In diesem Fall gibt die Prozedur eine entsprechende Meldung aus und wird abgebrochen:

```
If bolEntwurfsansicht = False Then
    MsgBox "Es ist kein Formular in der Entwurfsansicht ↵
            geöffnet.", vbOKOnly + vbExclamation, "Kein ↵
                                        Formular geöffnet"
    Exit Sub
End If
```

Außerdem kann es sein, dass wir die gleichen Steuerelemente zu Testzwecken mehrere Male umbenennen oder das diese aus anderen Gründen zuvor bereits solche Namen erhalten haben, wie Sie diese mit der Prozedur zuweisen wollen. Dabei kann es vorkommen, dass Sie einem Steuerelement einen Namen geben wollen, der bereits vergeben ist. Dies würde einen Fehler auslösen. Um dies zu verhindern durchlaufen wir eine ähnliche **For Each**-Schleife wie diejenige, welche die gewünschten Namen zuweist, bereits vorher und vergeben Namen wie **xyz1**, **xyz2** und so weiter für die Steuerelemente, bevor diese ihre endgültigen Namen erhalten. Danach müssen wir **intNummer** erneut mit **intStart** initialisieren:

```
For Each ctl In frm.Controls
    If ctl.InSelection Then
        ctl.Name = "xyz" & intNummer
        intNummer = intNummer + 1
    End If
Next ctl
intNummer = intStart
```

Ungünstige Aktivierreihenfolge

Es kann sein, dass die Aktivierreihenfolge der zu benennenden Steuerelemente nicht der Reihenfolge entspricht, in der die Steuerelemente neben- oder untereinander angeordnet sind. Die **For Each**-Schleife über alle **Control**-Elemente des Formulars durchläuft diese jedoch in der Reihenfolge, die der Aktivierreihenfolge entspricht.

Die Aktivierreihenfolge können Sie jedoch zuvor über den Dialog **Reihenfolge** einstellen, wenn die Aktivierreihenfolge nicht stimmt (siehe Bild 2). Diesen Dialog öffnen Sie über den Ribbon-Befehl **Entwurf|Tools|Eigenschaften**.

Parameter per Formular eingeben

Damit der Benutzer die Parameter bequem eingeben kann, wenn er das Add-In aufgerufen hat, stellen wir ihm ein passendes Formular zur Verfügung. Dieses soll die zuletzt verwendeten Werte außerdem speichern. Dazu erstellen wir zuerst eine Tabelle namens **tblOptionen**. Diese enthält die Felder aus Bild 3.

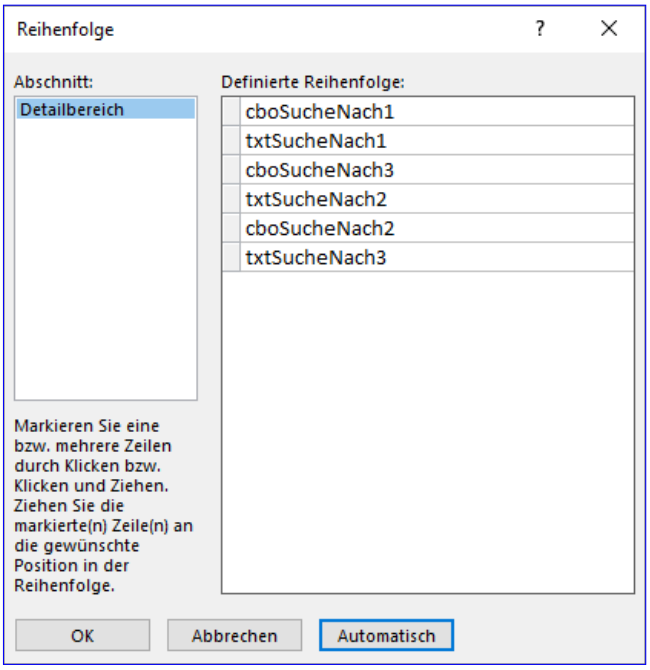


Bild 2: Zu benennende Steuerelemente

ParamArray-Auflistungen in VBA nutzen

Wenn Sie unter VBA eine Prozedur oder eine Funktion definieren, enthält diese immer eine feste Anzahl von Parametern. Diese können Sie auch als optional deklarieren, sodass tatsächlich weniger Werte übergeben werden als Parameter vorhanden sind. Was aber, wenn Sie den Spieß einmal umdrehen und mehr Werte als vorhandene Parameter übergeben wollen – und das auch noch flexibel? Dann kommt die ParamArray-Auflistung ins Spiel. In diesem Beitrag schauen wir uns an, was Sie damit machen können.

Parameter und optionale Parameter

Normalerweise legen Sie für eine Prozedur oder Funktion, nachfolgend als Routine bezeichnet, die Anzahl der Parameter genau fest:

```
Public Sub BeispielParameter(strText As String, _
                             lngZahl As Long)
    Debug.Print strText
    Debug.Print lngZahl
End Sub
```

Der Aufruf erfolgt dann mit der exakten Zahl an Parametern:

```
BeispielParameter "Beispieltext", 123
```

Sie können auch optionale Parameter, die Sie mit dem Schlüsselwort **Optional** versehen und die sich immer am Ende der Parameterliste befinden müssen:

```
Public Sub BeispielOptionaleParameter(_
            strText As String, Optional lngZahl As Long)
    Debug.Print strText
    Debug.Print lngZahl
End Sub
```

Diese Prozedur können Sie so aufrufen:

```
BeispielOptionaleParameter "Beispieltext", 123
```

Sie können den zweiten Parameter aber auch weglassen:

```
BeispielOptionaleParameter "Beispieltext"
```

Die Routine verwendet dann einen Standardwert, den Sie angeben können oder auch nicht. Wenn Sie ihn nicht angeben, wird bei **String**-Variablen eine leere Zeichenkette, bei Zahlen der Wert **0** und bei **Boolean**-Werten **False** angenommen.

Flexible Anzahl an Parametern

Wenn Sie eine flexible Anzahl an Parametern übergeben wollen, wozu brauchen wir dann mehr als optionale Parameter? Davon können wir schließlich so viele definieren, wie wir benötigen:

```
Public Sub VieleParameter(Optional lng1 As Long, _
          Optional lng2 As Long, Optional lng3 As Long, _
          Optional lng4 As Long, Optional lng5 As Long, _
                                 Optional lng6 As Long)
    Debug.Print lng1, lng2, lng3, lng4, lng5, lng6
End Sub
```

Allerdings stimmt das nicht so ganz, denn nach unseren Tests können Sie maximal 60 Parameter für eine Routine definieren – anderenfalls erscheint eine Meldung wie in Bild 1. Und damit kommen wir zum **ParamArray**.

Das Schlüsselwort ParamArray

Sie können in einer Routine genau einen Parameter mit dem Schlüsselwort **ParamArray** ausstatten. Dabei muss es sich zwingend um den letzten Parameter der Routine handeln.

Auflistungszeichen aus Textdateien ersetzen

Wenn Sie die Daten aus Textdateien oder ähnlichen Datenquellen einlesen möchten, benötigen Sie ein Trennzeichen, welches die einzelnen Spalten einer Zeile trennt. Das ist meist durch ein Tabulator-Zeichen, das Semikolon oder das Komma gegeben. Beim Tabulator-Zeichen treten meist keine Probleme auf, aber beim Semikolon oder beim Komma kann es zu folgendem Problem kommen: Die Zeile könnte Spalten enthalten, die ihrerseits das als Trennzeichen verwendete Zeichen enthalten. In diesem Beitrag schauen wir uns an, wie Sie mit solchen Datenquellen unter VBA umgehen können.

Ein ganz einfaches Beispiel für eine solche Zeile ist diese:

```
1; 'Beispielartikel'; 'Dies ist ein Beispielartikel.'
```

Hier können wir einfach nach dem Semikolon trennen und erhalten den Inhalt der drei Spalten. Aber was ist mit der folgenden Zeile?

```
2; 'Noch ein Beispielartikel'; 'Semikola kommen selten
vor; manchmal aber schon.'
```

Hier können wir nun nicht mehr einfach nach dem Semikolon trennen. Stattdessen müssten wir untersuchen, ob sich das Semikolon innerhalb eines Paars von einfachen Anführungszeichen befindet – oder auch innerhalb eines Paares von normalen Anführungszeichen.

Wir starten einmal mit einer Prozedur, die nach Schema F vorgeht und so tut, als würde es innerhalb von Spalten keine Delimiter-Zeichen geben. Diese stellt in einem **String**-Array namens **strTestdaten** die beiden oben genannten Zeilen zusammen und durchläuft dieses dann in einer **For...Next**-Schleife über alle Elemente des **String**-Arrays. Darin ruft sie die Prozedur **SpaltenErmitteln** auf, der Sie die jeweilige Zeile übergibt sowie das zu verwendende Delimiter-Zeichen (siehe Listing 1).

Diese Funktion soll ein Array der Spalten der Zeile zurückliefern, die wir dann innerhalb einer weiteren **For...**

```
Public Function Testdaten()
    Dim strTestdaten(2) As String
    Dim i As Integer
    Dim j As Integer
    Dim strSpalten() As String
    strTestdaten(0) = "1; 'Beispielartikel'; 'Dies ist ein Beispielartikel.'"
    strTestdaten(1) = "2; 'Noch ein Beispielartikel'; 'Semikola kommen selten vor; manchmal aber schon.'"
    For i = LBound(strTestdaten) To UBound(strTestdaten)
        strSpalten = SpaltenErmitteln(strTestdaten(i), ";")
        For j = LBound(strSpalten) To UBound(strSpalten)
            Debug.Print j, strSpalten(j)
        Next j
    Next i
End Function
```

Listing 1: Prozedur, die Zeilen an eine Funktion zum Trennen einer Zeile nach dem Trennzeichen übergibt

```
Public Function SpaltenErmitteln(strZeile As String, strDelimiter As String) As Variant
    Dim strSpalten() As String
    strSpalten = Split(strZeile, strDelimiter)
    SpaltenErmitteln = strSpalten
End Function
```

Listing 2: Funktion zum Auftrennen von Zeilen am Delimiter

Next-Schleife, diesmal mit der Zählervariablen **j** versehen, im Direktbereich des VBA-Editors ausgeben.

Die Funktion **SpaltenErmitteln** erwartet die zu untersuchende Zeile sowie das Delimiter-Zeichen. Sie trennt dann die Zeile mit der **Split**-Methode auf, welche die Zeile und den Delimiter erwartet, und ein Array der Elemente zurückliefert (siehe Listing 2).

Das Ergebnis für die obigen beiden Zeilen sieht dann wie folgt aus, wenn wir zusätzlich noch die Spaltennummern ausgeben:

```
0    1
1    'Beispielartikel'
2    'Dies ist ein Beispielartikel.'
0    2
1    'Noch ein Beispielartikel'
2    'Semikola kommen selten vor
3    manchmal aber schon.'
```

Sie sehen hier bereits, dass einmal drei und einmal vier Spalten ermittelt wurden. Das Semikolon innerhalb der Anführungszeichen wurde nämlich als Delimiter erkannt. Das ist keine Basis, um die Daten etwa in eine Tabelle einzutragen.

Also müssen wir einen Weg finden, um nur die Delimiter zu berücksichtigen, die sich nicht innerhalb von Anführungszeichen befinden.

Spalten nur außerhalb von Literalen erkennen

Dazu erweitern wir die Funktion **SpaltenErmitteln** erheblich und gehen innerhalb der Funktion prinzipiell wie folgt vor:

- Wir teilen die Zeile zunächst wieder mit der **Split**-Funktion an allen Stellen, die das Semikolon (oder ein anderes Trennzeichen) enthalten, auf.

- Dann durchlaufen wir alle Elemente des Arrays und schauen, ob die jeweils aktuelle Zeile eine ungerade Anzahl von Anführungszeichen enthält. In diesem Fall setzen wir einen Marker, der beim nächsten Durchlauf dafür sorgt, dass die folgende Zeile an die zuvor untersuchte angehängt wird. Das wiederholen wir solange, bis wir nochmal in einer Zeile eine ungerade Anzahl von Anführungszeichen vorfinden. Dann schließen wir die Zeile ab und ändern den Wert dieses Markers.

Im Detail sieht das wie in Listing 3 aus. Die Funktion **SpaltenErmitteln** nimmt nun noch einen weiteren Parameter entgegen, mit dem Sie festlegen, welches Zeichen als Anführungszeichen verwendet wird – zum Beispiel das Hochkomma oder das normale Anführungszeichen.

Dann deklarieren wir zwei Arrays. Das erste nimmt das Ergebnis der ersten **Split**-Anweisung entgegen, das einfach nur eine Unterteilung der Zeile an allen Vorkommen des Delimiters vornimmt. Die zweite soll dann die tatsächlichen Zeilen aufnehmen:

```
Dim strSpalten() As String
Dim strSpaltenBereinigt() As String
```

i ist die Laufvariable zum Durchlaufen der Elemente des Arrays **strSpalten**, **intAnzahlSpalten** ist die Anzahl der Spalten zum Redimsionieren des im zweiten Schritt zusammengestellten Arrays, **intAnzahlAnfuehrungszeichen** nimmt die Anzahl der Anführungszeichen innerhalb eines Element des ersten Arrays auf und **bolAnfueh-**

Pivot-Tabellen und -Diagramme in Excel

Microsoft hat die Pivot-Tabellen und -Diagramme mit der Version 2010 aus Access herausgenommen. Damit ist Excel das einzige Produkt in der Office-Familie, dass diese praktische Darstellungsweise von Daten beherrscht. Schauen wir uns also an, wie wir diese nutzen können. Im vorliegenden Beitrag widmen wir uns den Grundlagen von Pivot-Tabellen und -Diagrammen in Excel. In weiteren Beiträgen schauen wir uns dann an, wie wir diese von Access aus füllen und nutzen können, sodass der Benutzer möglichst wenige zusätzliche Handgriffe ausführen muss.

Wozu Pivot-Tabellen oder -Diagramme?

Manche Datenmengen sind so groß, dass es unmöglich ist, durch diese Ergebnisse Rückschlüsse zu ziehen. In diesem Fall muss man die Daten irgendwie anders darstellen oder zusammenfassen, um diese sinnvoll auswerten zu können. Unter Access kennen wir dazu Abfragen. Allerdings bedingen Abfragen, dass der Benutzer sich mit dem Entwurf von Abfragen auskennt. Das ist trotz der Entwurfsansicht von Abfragen von Access, welche die dahinterliegende Abfragesprache SQL maskiert, nicht trivial. Einfacher geht es mit sogenannten Pivot-Tabellen. Diese erlauben es, die Daten des gewünschten Bereichs einer Tabelle in verschiedenen Konstellationen darzustellen, diese zu gruppieren und Berechnungen anzustellen.

Beispieldaten

Als Beispieldaten nutzen wir einige Daten der Südsturm-Datenbank, die wir ein einer Abfrage zusammengefasst haben. Diese sieht wie in aus und enthält Daten aus den Tabellen **tblKunden**, **tblBestellungen**, **tblBestelldetails**, **tblArtikel**, **tblKategorien** und **tblLieferanten** (siehe Bild 1). So haben wir ausreichend Spielmaterial für die ersten Gehversuche mit Pivot-Tabellen und -Diagrammen unter Excel.

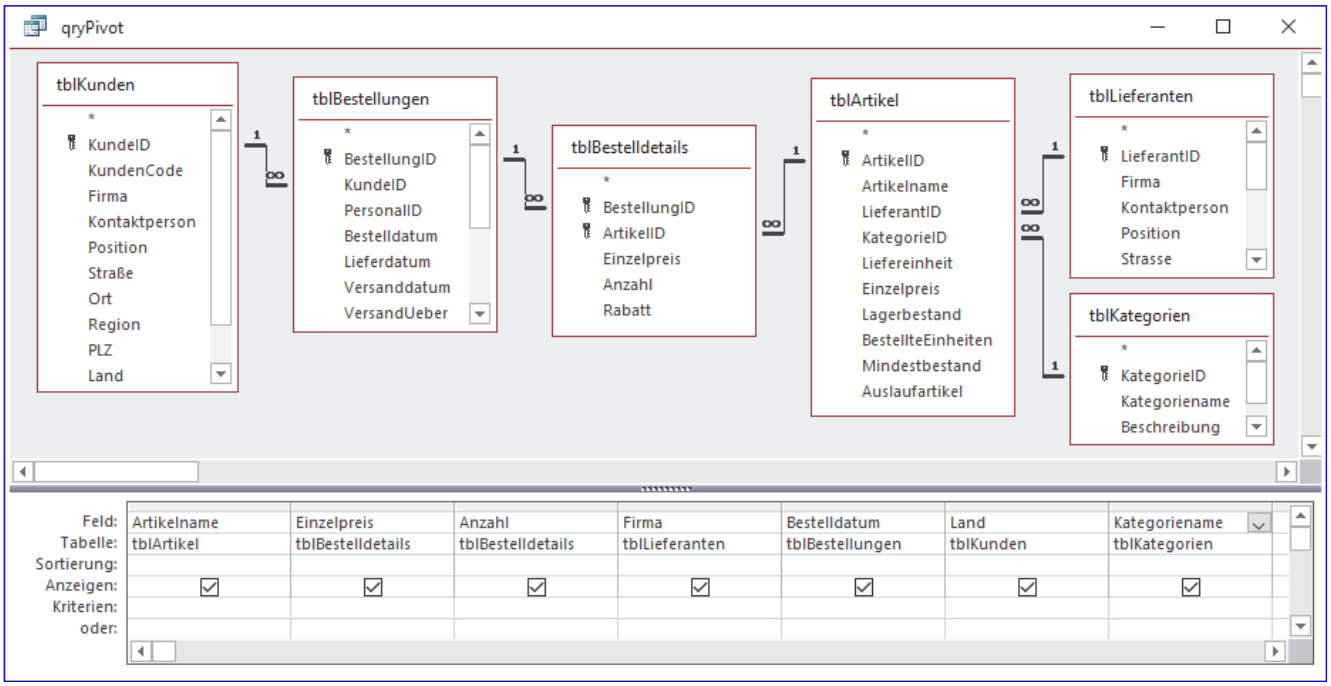


Bild 1: Datenquelle für die Pivot-Auswertungen

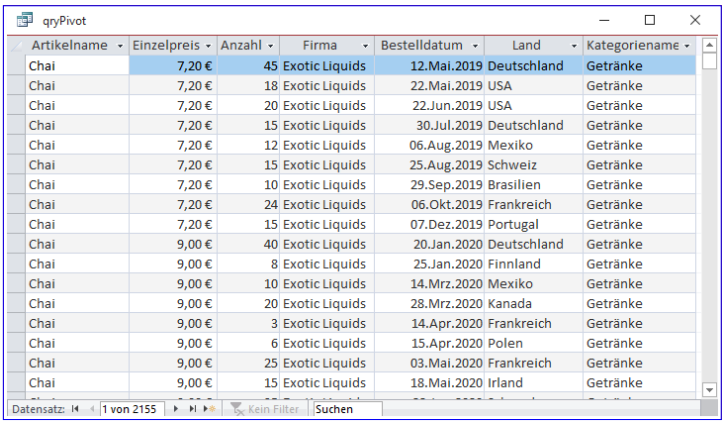


Bild 2: Daten für die Pivot-Auswertungen

Die Daten können wir nach Artikel, Lieferant, Bestelldatum und Zielland gruppieren, und mit über 2.000 Datensätzen sollte auch die Datenmenge ausreichend sein (siehe Bild 2).

Beispieldaten exportieren

Die Daten exportieren wir nun im Excel-Format. Dazu markieren Sie die unter dem Namen **qryPivot** gespeicherte Abfrage im Navigationsbereich und wählen den Kontextmenü-Eintrag **Exportieren|Excel** aus (siehe Bild 3).

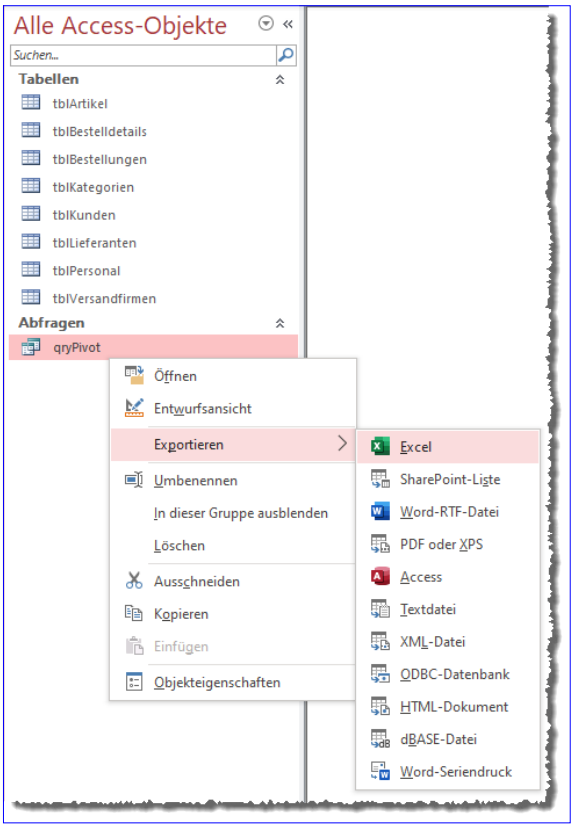


Bild 3: Exportieren der Daten in eine Excel-Datei

Im ersten Schritt des Export-Assistenten geben Sie den Namen der Zieldatei an. Danach klicken Sie einfach auf **OK**, wir benötigen keinen Export mit Layout oder dergleichen. Die neu erstellte Excel-Datei enthält dann ein Tabellenblatt mit den Daten (siehe Bild 4). Damit können wir die Access-Datenbank vorerst schließen.

Erste Pivot-Schritte

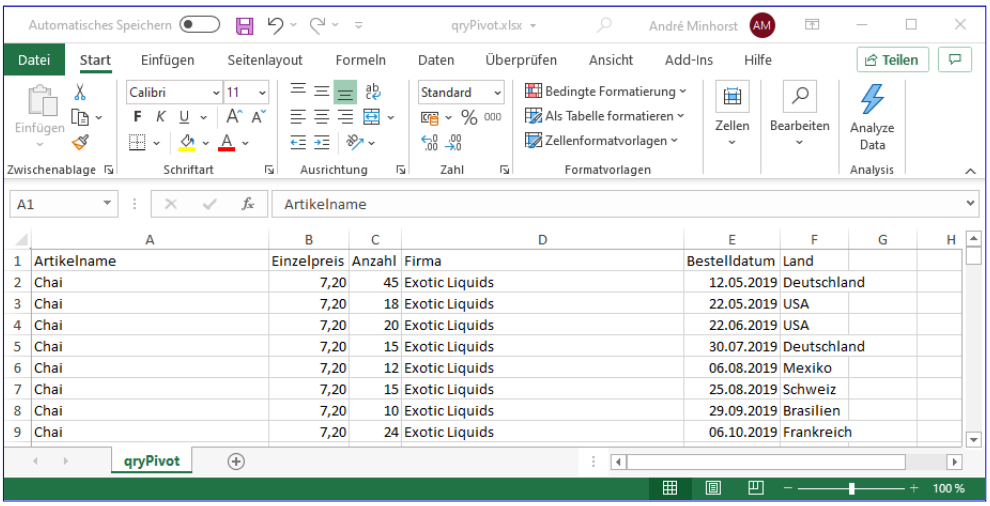


Bild 4: Unser Beispielmaterial für das Erstellen von Pivot-Tabellen

Nach dem Öffnen der Excel-Datei wollen wir gleich eine erste Pivot-Tabelle erstellen. Unter Excel wechseln Sie dazu im Ribbon zum Bereich **Einfügen**. Hier wählen wir den Eintrag **Tabellen|PivotTable** (siehe Bild 5).

Dies öffnet den Dialog aus Bild 6. Hier wählen Sie im oberen Bereich fest, aus welchem Bereich die Daten stammen, die mit der Pivot-

Tabelle ausgewertet werden sollen. In unserem Fall ist die Auswahl bereits automatisch erfolgt und umfasst alle Zeilen und Spalten inklusive Spaltenüberschriften. Wir können hier auch eine alternative Datenquelle verwenden, also beispielsweise direkt auf eine Access-Abfrage zugreifen. Wie das gelingt, schauen wir uns später an.

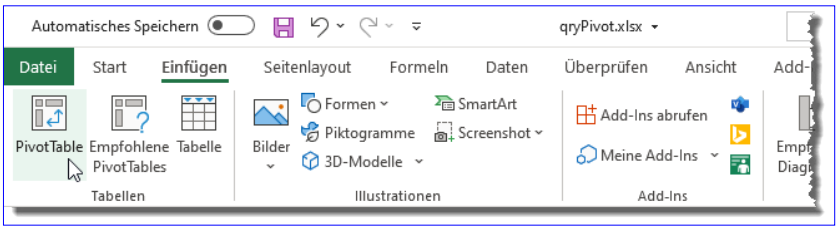


Bild 5: Erstellen einer Pivot-Tabelle per Ribbon-Befehl

Außerdem geben wir in diesem Dialog an, wo die neue Pivot-Tabelle platziert werden soll. Dazu können Sie ein neues Arbeitsblatt nutzen, aber Sie können diese auch direkt auf dem Arbeitsplatt mit den Daten platzieren.

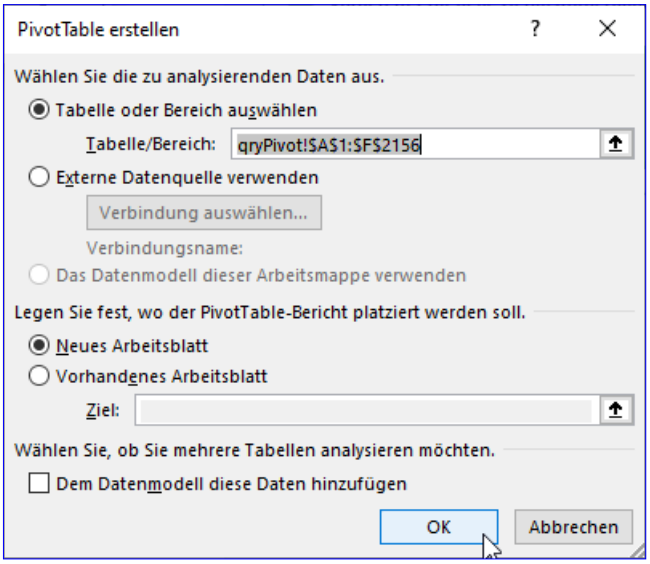


Bild 6: Dialog zum Erstellen einer Pivot-Tabelle

Nach einem Klick auf **OK** erscheint das neue Arbeitsblatt, das zwei wesentliche Bereiche anzeigt – auf der linken Seite den Platzhalter für die zu erzeugende Pivot-Tabelle und auf der rechten die Feldliste, mit der Sie die Felder der Datenquelle den verschiedenen Aufgaben zuweisen können (siehe Bild 7). Die vier Bereiche lauten **Filter**, **Spalten**, **Zeilen** und **Werte**.

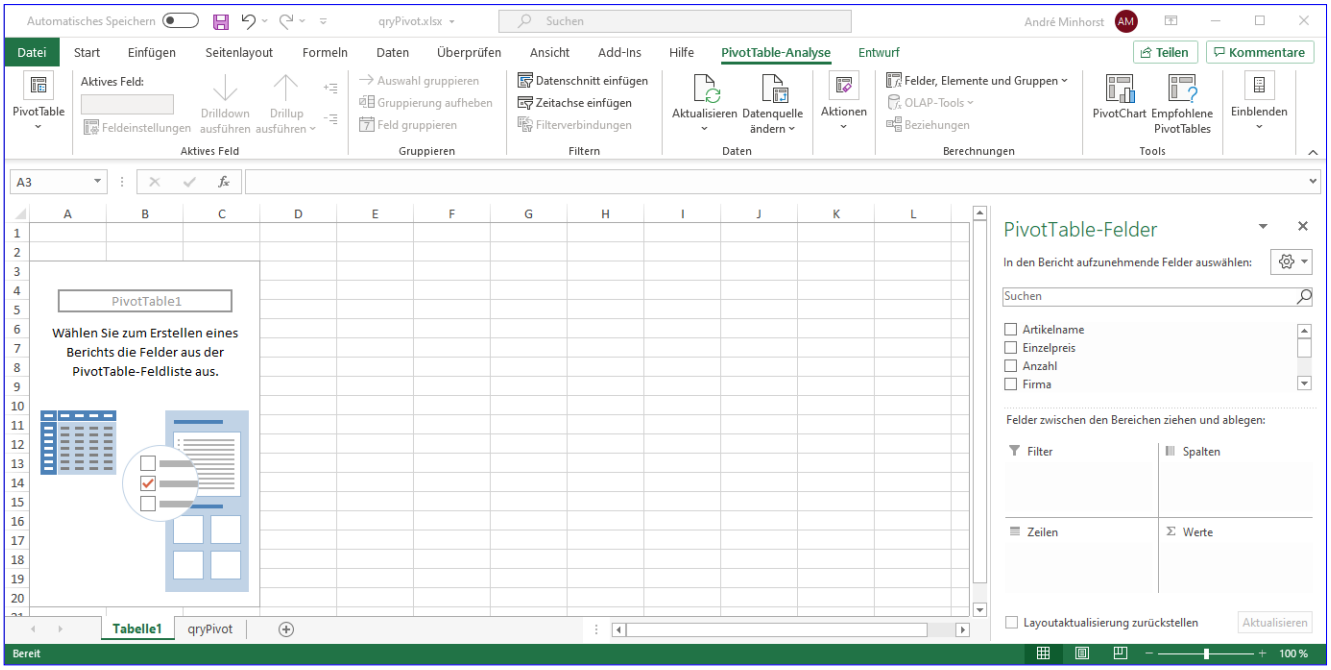


Bild 7: Ausgangspunkt zum Einrichten der Pivot-Tabelle

Zuweisen der Felder zur Pivot-Tabelle

Nun folgt der interessante Teil: Das Zuordnen der Felder aus der Feldliste zu den vier Bereichen. Dies können Sie auf zwei Arten erledigen:

- Durch Setzen von Haken an die hinzuzufügenden Felder aus der Feldliste. Dies fügt die Felder automatisch den Bereichen zu. Damit geben Sie die Entscheidung, welche Felder in welchem Bereich landen, in die Hand von Excel.

- Per Drag and Drop der Felder in jeweils einen der vier Bereiche.

Welches Feld Sie in welchen Bereich ziehen, hängt von Ihren Anforderungen ab. Wir wollen im ersten Schritt einmal ermitteln, welche Artikel in welche Länder verkauft wurden. Dazu ziehen wir das Feld **Land** in den Bereich **Spalten**, das Feld **Artikelname** in den Bereich **Zeilen** und das Feld **Anzahl** in den Bereich **Werte** (siehe Bild 8).

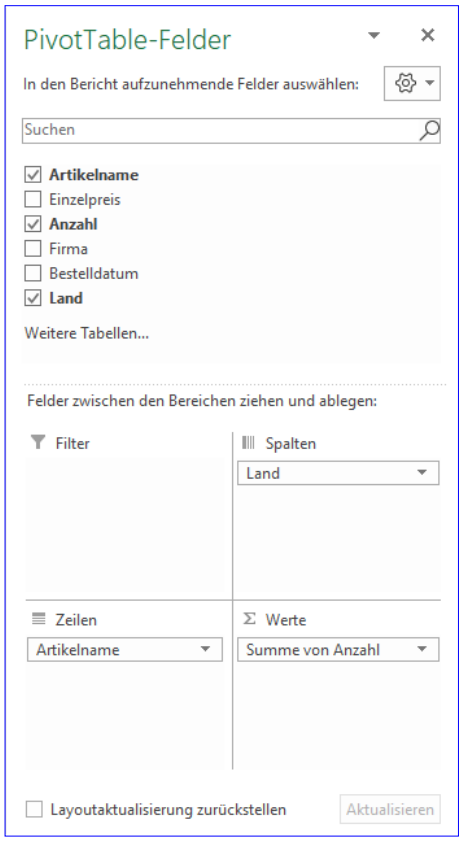
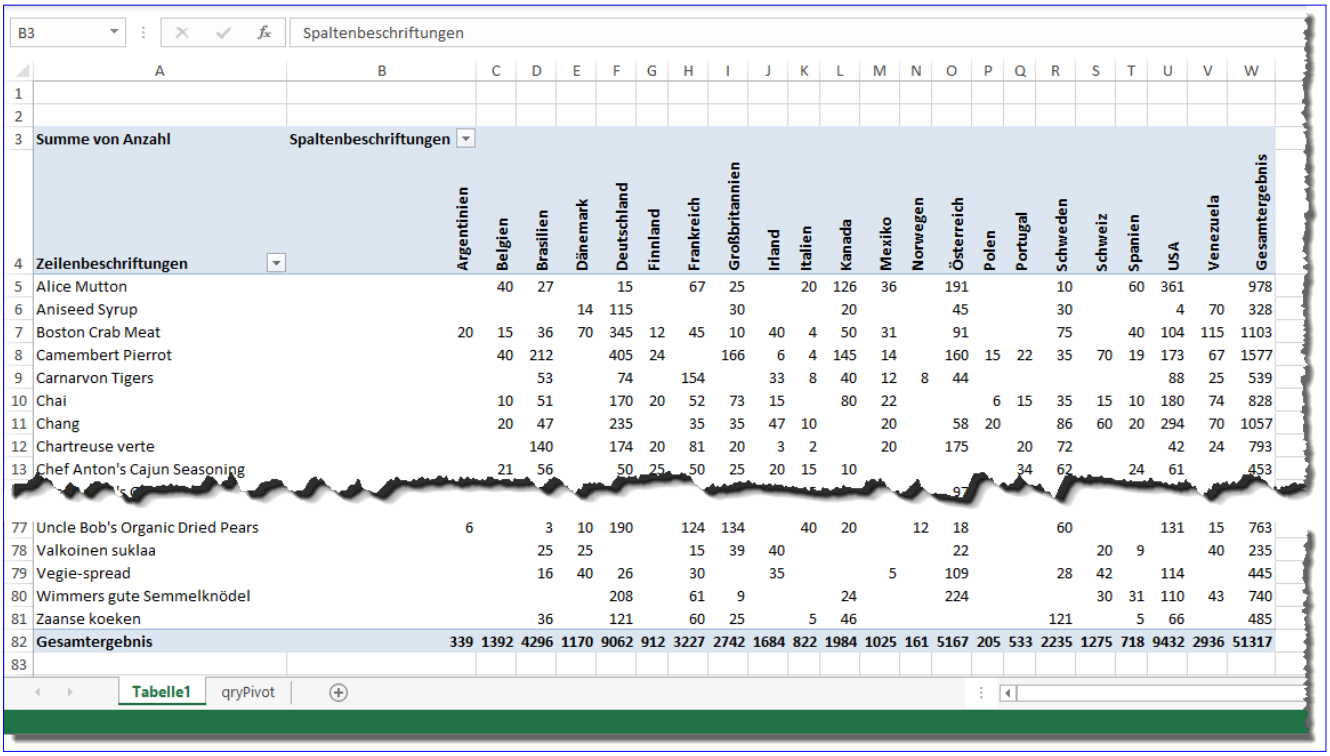


Bild 8: Zuordnen der Felder zu den Bereichen

Damit erhalten wir das Ergebnis aus Bild 9.

Wir haben hier noch eine kleine Änderung vorgenommen: Wir haben die Spaltenüberschriften vertikal angeordnet. Dazu klicken Sie mit der rechten Maustaste auf alle anzupassenden Felder, hier die mit den Spaltenüberschriften, und wählen den Kontextmenü-Eintrag **Zellen formatieren** aus. Im nun erscheinenden Dialog **Zellen formatieren** wechseln Sie zur Registerseite



Summe von Anzahl	Spaltenbeschriftungen																						
Zeilenbeschriftungen	Argentinien	Belgien	Brasilien	Dänemark	Deutschland	Finnland	Frankreich	Großbritannien	Irland	Italien	Kanada	Mexiko	Norwegen	Österreich	Polen	Portugal	Schweden	Schweiz	Spanien	USA	Venezuela	Gesamtergebnis
Alice Mutton		40	27		15		67	25		20	126	36		191			10		60	361		978
Aniseed Syrup				14	115			30			20			45			30			4	70	328
Boston Crab Meat	20	15	36	70	345	12	45	10	40	4	50	31		91			75		40	104	115	1103
Camembert Pierrot		40	212		405	24		166	6	4	145	14		160	15	22	35	70	19	173	67	1577
Carnarvon Tigers			53		74		154		33	8	40	12	8	44						88	25	539
Chai		10	51		170	20	52	73	15		80	22			6	15	35	15	10	180	74	828
Chang		20	47		235		35	35	47	10		20		58	20		86	60	20	294	70	1057
Chartreuse verte			140		174	20	81	20	3	2		20		175		20	72			42	24	793
Chef Anton's Cajun Seasoning		21	56		50	25	50	25	20	15	10					34	62		24	61		453
Uncle Bob's Organic Dried Pears	6		3	10	190		124	134		40	20		12	18			60			131	15	763
Valkoinen suklaa			25	25			15	39	40					22				20	9		40	235
Vegie-spread			16	40	26		30		35			5		109			28	42		114		445
Wimmers gute Semmelknödel					208		61	9			24			224				30	31	110	43	740
Zaanse koeken			36		121		60	25		5	46						121		5	66		485
Gesamtergebnis	339	1392	4296	1170	9062	912	3227	2742	1684	822	1984	1025	161	5167	205	533	2235	1275	718	9432	2936	51317

Bild 9: Ergebnis der ersten Pivot-Tabelle

Ausrichtung und stellen diese auf **90°** ein (siehe Bild 10).

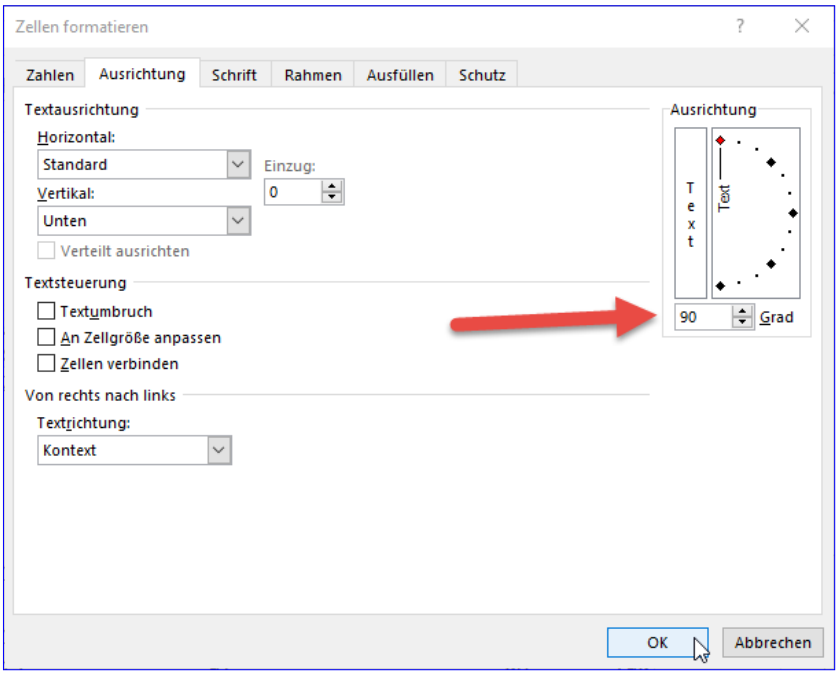


Bild 10: Vertikale Ausrichtung der Spaltenüberschriften

Das Ergebnis ist schon recht interessant: Wir sehen die Menge der verkauften Artikel je Land, die verkaufte Gesamtmenge eines Artikels sowie die Gesamtmenge aller Artikel je Land.

Filtern nach Zeilen oder Spalten

Die nun in den Zeilen und Spalten befindlichen Überschriften können Sie filtern, das heißt, Sie können zum Beispiel nur die Bestellungen aus europäischen Ländern anzeigen lassen. Dazu klicken Sie neben dem Text **Spaltenbeschriftungen** auf die Schaltfläche mit dem nach unten zeigenden Dreieck und selektieren die anzuzeigenden Einträge beziehungsweise wählen die nicht mehr gewünschten Einträge ab (siehe Bild 11). Nach einem Klick auf die Schaltfläche **OK** zeigt die Pivot-Tabelle direkt die resultierenden Werte an. Außerdem ändert sich das Symbol vom nach unten zeigenden Dreieck der Schaltfläche in ein **Filter**-Symbol. Entsprechend der Änderungen wird auch das Gesamtergebnis in der Spalte ganz rechts neu berechnet.

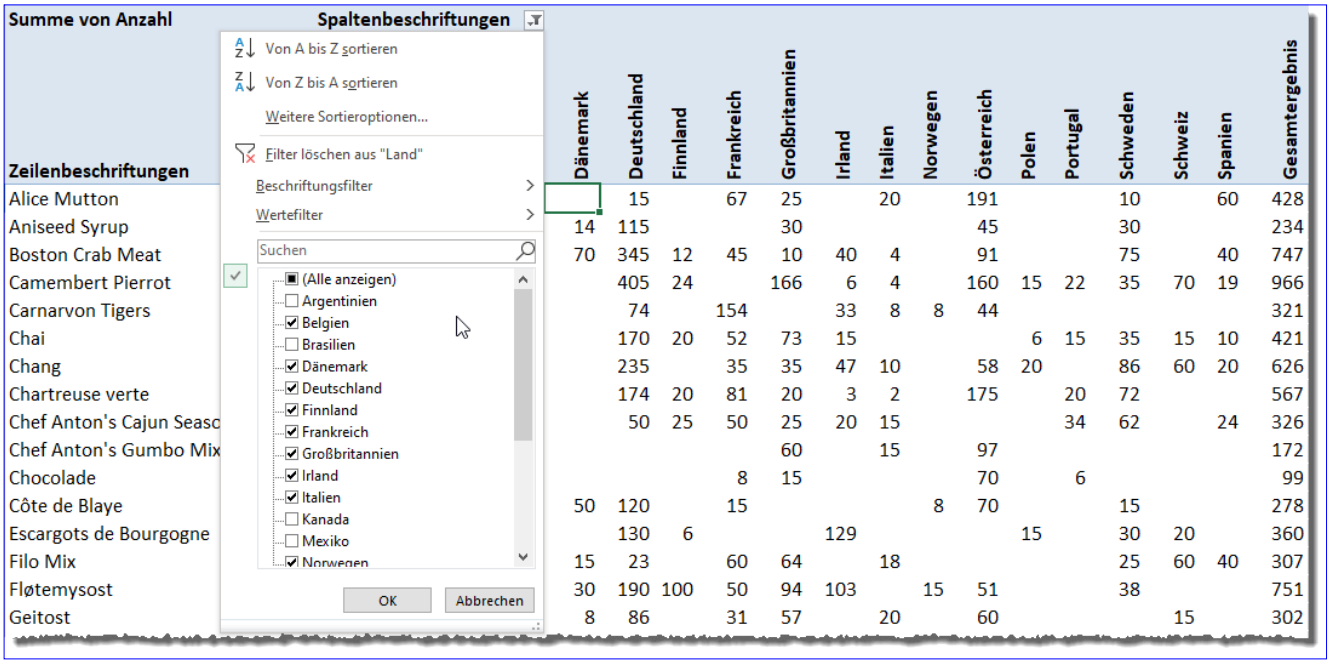


Bild 11: Vertikale Ausrichtung der Spaltenüberschriften

Pivot-Tabellen und -Charts automatisch erstellen

Da wir in Access seit Version 2013 keine eingebaute Anzeige von Pivot-Tabellen oder -Charts haben, müssen wir uns mit Excel behelfen. Die Grundlagen dazu finden Sie im Beitrag »Pivot-Tabellen und -Diagramme in Excel«. Nun schauen wir uns an, wie wir von Access aus per Schaltfläche ein mit den aktuellen Daten einer dafür vorgesehenen Abfrage nach Excel exportieren und dieser Excel-Datei eine Pivot-Tabelle hinzufügen können.

Ziel dieses Beitrags

Im Beitrag **Pivot-Tabellen und -Diagramme in Excel** (**www.access-im-unternehmen.de/****) haben wir uns angesehen, wie Sie die Daten einer Access-Abfrage manuell in eine Excel-Datei exportieren und dieser dann in einem neuen Arbeitsblatt eine Pivot-Tabelle hinzufügen. Diesen Vorgang wollen wir nun automatisieren. Das heißt, dass der Benutzer nur eine Schaltfläche anklicken soll und dann alle anderen Schritte automatisch ablaufen – bis zur Anzeige der Seite der Excel-Datei mit der Pivot-Tabelle. Das ist eine Möglichkeit, Kunden die gewünschte Auswertung anzeigen zu lassen. Weitergehende Anpassungen kann der Kunde dann in der Excel-Datei selbst vornehmen.

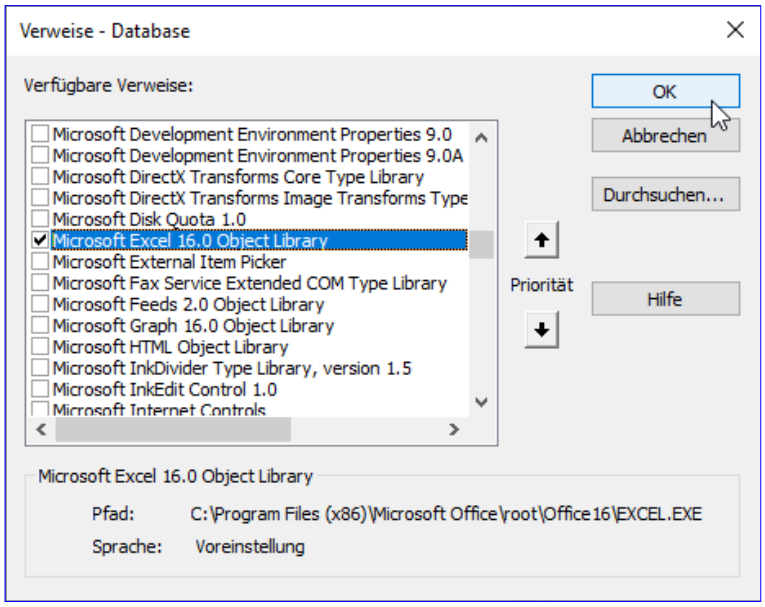


Bild 1: Verweis auf die **Excel**-Objektbibliothek

Excel automatisieren

Die Automatisierung von Excel ist wesentlich einfacher, wenn wir einen Verweis zur Objektbibliothek von Excel herstellen. Wir können dann zum Beispiel im VBA-Editor IntelliSense nutzen. Den Verweis fügen Sie hinzu, indem Sie im VBA-Editor (**Alt + F11**) den Menüpunkt **Extras|Verweise** betätigen und dort den Eintrag **Microsoft Excel 16.0 Object Library** hinzufügen (siehe Bild 1).

Daten exportieren

Das Exportieren der Daten für die gewünschte Tabelle oder Abfrage erledigen wir mit der Funktion **DatenExportieren** (siehe Listing 1). Diese erwartet zwei Parameter:

- **strQuelle**: Name der zu exportierenden Tabelle oder Abfrage
- **strDatei**: Pfad zu der zu erstellenden Datei

Die Funktion ruft bei deaktivierter Fehlerbehandlung die Anweisung **Kill** auf. Diese löscht eine eventuell bereits vorhandene Datei mit dem Namen aus **strDatei**. Die Fehlerbehandlung wird deaktiviert, weil **Kill** einen Fehler auslöst, wenn die angegebene Datei nicht vorhanden ist. Danach verwendet die Funktion die Methode **DoCmd.TransferSpreadsheet**, um die Daten aus der mit **strQuelle** angegebenen Tabelle oder Abfrage in die Datei aus **strDatei** zu exportieren.

```
Private Sub DatenExportieren(strQuelle As String, strDatei As String)
    On Error Resume Next
    Kill strDatei
    On Error GoTo 0
    DoCmd.TransferSpreadsheet acExport, acSpreadsheetTypeExcel12Xml, strQuelle, strDatei, True
End Sub
```

Listing 1: Exportieren der Datenquelle in eine Excel-Datei

Als Format verwenden wir **acSpreadsheetTypeExcel12Xml**. Die Dateiendung muss in diesem Fall **.xlsx** lauten.

Excel starten

Um die Excel-Instanz zu starten, in der wir die Pivot-Tabelle erzeugen wollen, nutzen wir eine weitere kleine Prozedur. Diese heißt **ExcelErzeugen** und sieht wie folgt aus:

```
Private Sub ExcelErzeugen()
    Set objExcel = New Excel.Application
    objExcel.Visible = True
End Sub
```

Für die hier erzeugte Excel-Instanz fügen wir im Klassenmodul des Formulars, aus dem wir die Funktionen aufrufen wollen, eine passende Objektvariable hinzu:

```
Dim objExcel As Excel.Application
```

```
Private Sub cmdPivotTabelleInExcelErstellen_Click()
    Dim objWorkbook As Excel.Workbook
    Dim strDatei As String
    Dim strQuelle As String
    strQuelle = "qryPivot"
    strDatei = CurrentProject.Path & "\qryPivot.xlsx"
    DatenExportieren strQuelle, strDatei
    ExcelErzeugen
    Set objWorkbook = WorkbookOeffnen(strDatei)
    '... weitere Anweisungen
End Sub
```

Listing 2: Exportieren und Öffnen der Daten

Um das soeben erstellte Workbook zu öffnen, verwenden wir diese Funktion:

```
Private Function WorkbookOeffnen(strDatei As String) _
                                        As Excel.Workbook
    Set WorkbookOeffnen = objExcel.Workbooks.Open(strDatei)
End Function
```

Dann kommen wir endlich zur Prozedur, die wir durch das Ereignis **Beim Klicken** einer Schaltfläche namens **cmdPivotTabelleInExcelErstellen** auslösen.

Diese deklariert eine Variable für das **WorkBook**-Objekt sowie für den Namen der Quelle und der zu erstellenden Datei und füllt diese noch mit Werten.

Diese Werte können Sie auch über entsprechende Steuerelemente im Formular durch den Benutzer füllen lassen.

Solange wir eine Pivot-Tabelle für eine spezielle Datenquelle erstellen, macht es aber keinen Sinn, eine andere als die dafür vorgesehene Datenquelle anzugeben.

Danach ruft die Prozedur nacheinander die bereits vorgestellten Routinen **DatenExportieren**, **ExcelErzeugen** und **objWorkbook** auf. Das Ergebnis ist eine geöffnete Excel-Instanz, welche die exportierten Daten in einem Worksheet anzeigt (siehe Bild 2).

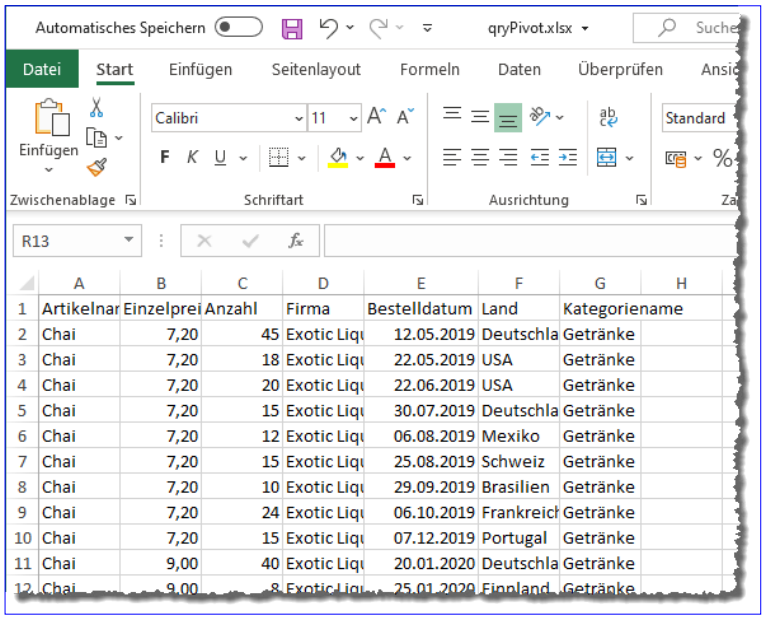
Nun benötigen wir noch ein zweites Worksheet, in dem wir die Pivot-Tabelle anlegen.

Pivot-Tabelle erzeugen

Das Erzeugen der eigentlichen Pivot-Tabelle lagern wir in die Prozedur **PivotTabelleErzeugen** aus, die zwei Parameter entgegennimmt:

- **objWorkbook**: Verweis auf die Excel-Datei
- **strQuelle**: Name der Quelltabelle oder -abfrage

Die Prozedur aus Listing 3 deklariert einige Variablen, zum Beispiel für das Worksheet mit den Daten und mit der zu erzeugenden Pivot-Tabelle oder für die verschiedenen **PivotField**-Elemente. Sie referenziert zuerst das beim Exportieren automatisch erstellte Worksheet mit dem Namen aus **strQuelle**, in unserem Beispiel **qryPivot**.



	A	B	C	D	E	F	G	H
1	Artikelnar	Einzelprei	Anzahl	Firma	Bestelldatum	Land	Kategoriename	
2	Chai	7,20	45	Exotic Liqu	12.05.2019	Deutschla	Getränke	
3	Chai	7,20	18	Exotic Liqu	22.05.2019	USA	Getränke	
4	Chai	7,20	20	Exotic Liqu	22.06.2019	USA	Getränke	
5	Chai	7,20	15	Exotic Liqu	30.07.2019	Deutschla	Getränke	
6	Chai	7,20	12	Exotic Liqu	06.08.2019	Mexiko	Getränke	
7	Chai	7,20	15	Exotic Liqu	25.08.2019	Schweiz	Getränke	
8	Chai	7,20	10	Exotic Liqu	29.09.2019	Brasilien	Getränke	
9	Chai	7,20	24	Exotic Liqu	06.10.2019	Frankreich	Getränke	
10	Chai	7,20	15	Exotic Liqu	07.12.2019	Portugal	Getränke	
11	Chai	9,00	40	Exotic Liqu	20.01.2020	Deutschla	Getränke	
12	Chai	9,00	8	Exotic Liqu	25.01.2020	Finnland	Getränke	

Bild 2: Die exportierten Excel-Daten

Dann fügt sie ein neues Worksheet hinzu, und zwar vor dem bereits bestehenden Worksheet mit den Rohdaten. Das erreichen wir, indem wir der **Add**-Methode als ersten Parameter das Worksheet angeben, vor dem das neue Worksheet erscheinen soll. Danach stellt die Prozedur die Namen der beiden Worksheets auf **Pivot-Tabelle** und **Pivot-Daten** ein.

Mit der Hilfsfunktion **BereichErmitteln** liest die Prozedur den Bereich mit den Daten aus dem mit **objDataSheet** referenzierten Worksheet ein:

```
Private Function BereichErmitteln(_
                objDataSheet As Worksheet) As Excel.Range
    Dim lngLetzteZeile As Long
    Dim lngLetzteSpalte As Long
    lngLetzteZeile = objDataSheet.Cells(_
                objDataSheet.Rows.Count, 1).End(xlUp).Row
    lngLetzteSpalte = objDataSheet.Cells(1, _
          objDataSheet.Columns.Count).End(xlToLeft).Column
    Set BereichErmitteln = objDataSheet.Cells(1, _
                1).Resize(lngLetzteZeile, lngLetzteSpalte)
End Function
```

Danach wird ein beim manuellen Erstellen nicht in Erscheinung tretender Cache auf Basis der zu berücksichtigenden Daten erzeugt.

Dieser dient als Grundlage für den Aufruf der Methode **CreatePivotTable**, der wir die als linke, obere Zelle zu verwendende Zelle des Ziel-Worksheets sowie eine Bezeichnung als Parameter übergeben.

Danach legen wir fest, welche Daten als Zeilenköpfe, Spaltenköpfe und als Werte genutzt werden sollen. Dabei weisen wir den Variablen **objPivotZeile1** und **objPivotZeile2** die Felder **Kategoriename** und **Artikelname** zu und legen für diese jeweils den Wert **xlRowField** als Wert der Eigenschaft **Orientation** fest.

Für die Variable **objPivotSpalte** nutzen wir das Feld **Land** sowie den Wert **xlColumnField** für die Eigenschaft **Orientation**.

Schließlich fehlen noch die Werte, die wir mit **objPivotWerte** referenzieren und die aus dem Feld **Anzahl** stammen. Hier lautet der Wert für die Eigenschaft **Orientation** auf **xlDataField**.

Statische Workflows im Griff

Es gibt immer wieder Standardabläufe im Büroalltag. Bei mir ist es das Schreiben von Artikeln. Für jeden Artikel gibt es einige Aufgaben, die immer in der gleichen Reihenfolge ablaufen. Wer denkt, das geht nach 20 Jahren automatisch, irrt sich – immer wieder bleibt mal eine Teilaufgabe liegen oder wird nicht in der richtigen Reihenfolge erledigt. Um das zu ändern, erstellen wir in diesem Beitrag eine Lösung, um wiederkehrende, nach dem gleichen Schema ablaufende Aufgaben zu verfolgen. Dazu speichern wir die einzelnen Schritte als Felder in einer Tabelle. Über die Benutzeroberfläche stellen wir die Aufgaben und die Teilschritte optisch so dar, dass die nun zu erledigen Teilaufgaben immer im Blickfeld sind!

Beispiel für einen Workflow

Wie sieht ein beispielhafter Workflow aus? Ich wähle gleich den aus meiner täglichen Arbeit – das Erstellen eines Beitrags etwa für das vorliegende Magazin. Hier tauchen die folgenden Aufgaben auf:

- Beitrag in Beitragsdatenbank aufnehmen
- Beitrag schreiben
- Beitrag setzen
- Beitrag zum Lektor schicken
- Beitrag wird durch Lektor geprüft
- Korrekturvorschläge des Lektors einarbeiten
- Beitrag in Beitragsdatenbank einlesen
- Beispieldateien in Beitragsdatenbank aufnehmen
- Beitrag und Beispieldateien online stellen

Zusätzlich gibt es je Magazin noch eine Reihe weiterer Aufgaben:

- PDF erstellen
- PDF zum Verlag schicken
- Verlag macht letzte Korrekturvorschläge
- Korrekturvorschläge des Verlags einarbeiten
- Magazin online stellen
- Downloadseite um neue Ausgabe erweitern
- Neuen Benutzer für die neuen Zugangsdaten anlegen
- Newsletter an Abonnenten schicken

Das sind bereits einige Aufgaben, die im Alltag noch etwas detaillierter ausfallen. Sie sollen jedoch reichen, um Beispielmaterial für die zu erstellende Lösung zu liefern.

In der Praxis gibt es für jeden Beitrag einen neuen Workflow und auch für jede neue Ausgabe. Wir könnten noch einen Schritt weitergehen und die Workflows einander unterordnen, also beispielsweise die Beitrags-Workflows den Magazin-Workflows zuweisen. Dies wollen wir aus Gründen der Übersicht in diesem Beitrag nicht erledigen.

Einfachste Methode

Wenn Sie immer wiederkehrende Unteraufgaben haben, können Sie einfach für jede Aufgabe ein Kontrollkästchen

zur Aufgabe hinzufügen und diese nach Erledigung der Aufgaben abhaken. Damit wissen Sie jederzeit, welchen Stand die Aufgabe hat. Wenn die Unteraufgaben sich nicht verändern, ist das auch die beste und einfachste Methode.

Wenn wir bei meinem Beispiel bleiben, wo sich die Unteraufgaben beim Schreiben eines Beitrags nie ändern, können Sie einfach für jede Unteraufgabe ein **Ja/Nein**-Feld zu der Tabelle hinzufügen, in der Sie auch weitere Daten zu den Beiträgen speichern – wie den Titel, den Inhalt et cetera.

Der Entwurf der Tabelle **tblBeitraege** sieht danach etwa wie in Bild 1 aus.

Danach können Sie die Felder wie in Bild 2 in einem Formular zur Bearbeitung der Beiträge unterbringen. Damit erhalten Sie einen detaillierten Überblick über alle Daten zu einem Beitrag.

Noch praktischer für einen Überblick über alle Aufgaben ist eine Übersicht der Aufgaben mehrerer Beiträge in einem Endlosformular oder in einem Datenblatt. Gegebenenfalls können Sie auch ein Listenfeld nutzen.

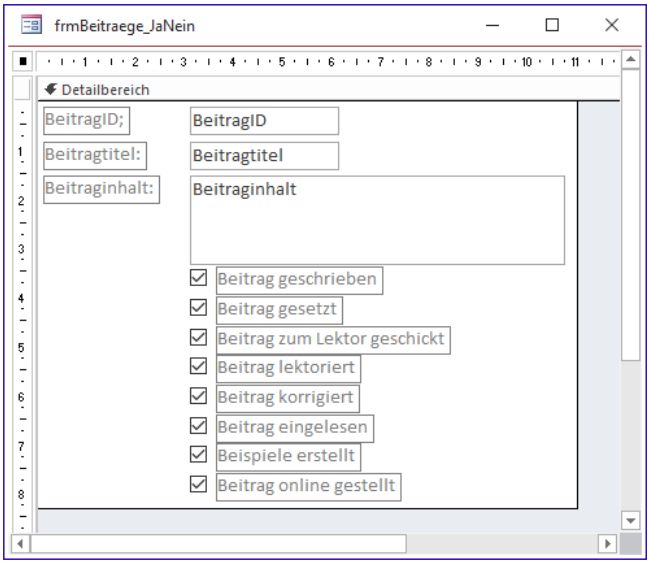


Bild 2: Formularentwurf mit Ja/Nein-Feldern

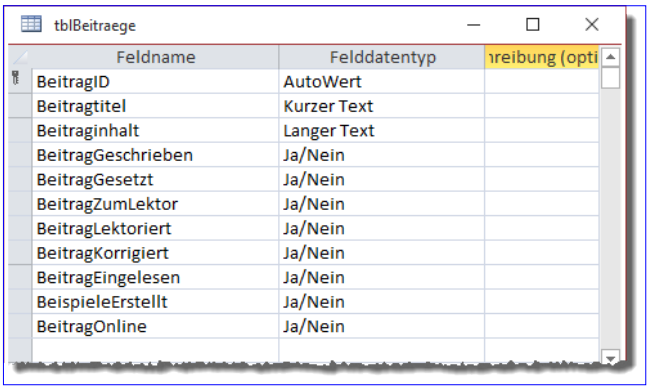


Bild 1: Einfache Variante zum Einbau eines Workflows

Bessere Reproduzierbarkeit mit Datum

Damit sehen wir nun, wie weit jeder Beitrag ist und welche Aufgaben noch zu erledigen sind. Aber was, wenn der Lektor fragt, wo der Beitrag bleibt, sie diesen aber schon als verschickt gekennzeichnet haben? Dann müssen Sie gegebenenfalls den kompletten Ordner der gesendeten Elemente in Outlook durchsuchen.

Etwas einfacher geht es, wenn Sie zumindest wissen, wann Sie den Beitrag an den Lektor verschickt haben. Zu diesem Zweck ändern wir den Datentyp der **Ja/Nein**-Felder der Tabelle **tblBeitraege** in **Datum/Zeit** um (siehe Bild 3).

Allerdings wollen wir diese Information nur abrufen, wenn unbedingt nötig. In einer Übersicht hingegen sollen nur Kontrollkästchen angezeigt werden, die angeben, ob die Unteraufgabe erledigt wurde oder nicht. In der Detailan-

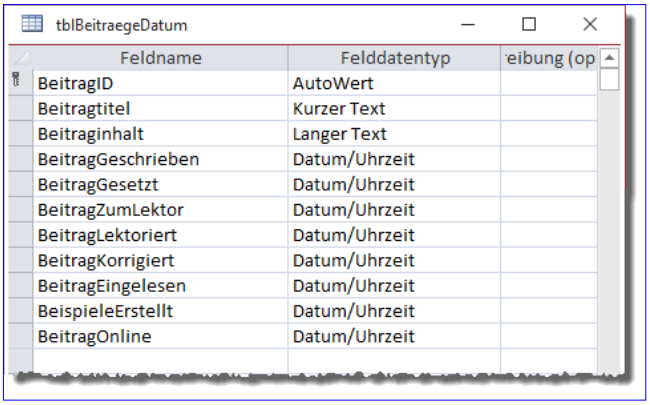


Bild 3: Tabelle mit Datum/Uhrzeit-Feldern

sicht wollen wir auch nur Kontrollkästchen anzeigen, die der Benutzer nach Erledigung einer Teilaufgabe abhaken kann. Nach dem Abhaken soll dann das Datum der Erledigung in das betroffene Feld eingetragen werden.

Zusammenfassend: Die Tabelle soll ein Datumsfeld für das Erledigungsdatum einer jeden Unteraufgabe enthalten. Die Formulare hingegen sollen ein Kontrollkästchen anzeigen. Das Kontrollkästchen soll angehakt sein, wenn das Datumsfeld nicht leer ist, wenn es leer ist, soll das Kontrollkästchen keinen Haken enthalten. Außerdem soll, wenn der Benutzer einen Haken in ein Kontrollkästchen setzt, das Erledigungsdatum in das zugrunde liegende **Datum/Zeit**-Feld eingetragen werden.

Wir haben also ein **Datum/Zeit**-Feld, dessen Status **Null**/**Nicht Null** in Form des Wertes eines Kontrollkästchens erscheinen soll. Dazu benötigen wir eine technische Lösung, die zum Beispiel in einer entsprechend formulierten Abfrage liegen könnte. Diese würde für jedes Unteraufgaben-Feld zwei Felder enthalten – einmal das **Datum/Zeit**-Feld und ein weiteres Feld, das prüft, ob das **Datum/Zeit**-Feld bereits einen Wert enthält oder nicht und abhängig davon den Wert **True** oder **False** ausgibt.

Diese Abfrage formulieren wir wie in Bild 4. Sie enthält das Feld **BeitragGeschrieben** und ein berechnetes Feld namens **BeitragGeschriebenJaNein**, das den Wert **-1** liefert, wenn das Feld **BeitragGeschrieben** gefüllt ist:

```
BeitragGeschriebenJaNein: Nicht
IstNull([BeitragGeschrieben])
```

Bild 5 zeigt, wie das Ergebnis der Abfrage für zwei Datensätze aussieht, von denen einer einen Datumswert im Feld **BeitragGeschrieben** aufweist und der andere den Wert **Null**.

Wir würden nun gern noch das berechnete Feld **BeitragGeschriebenJaNein** als Kontrollkästchen in der Datenblattansicht anzeigen. Dazu verwenden wir normalerweise die Eigenschaft **Steuerelement anzeigen** im Bereich

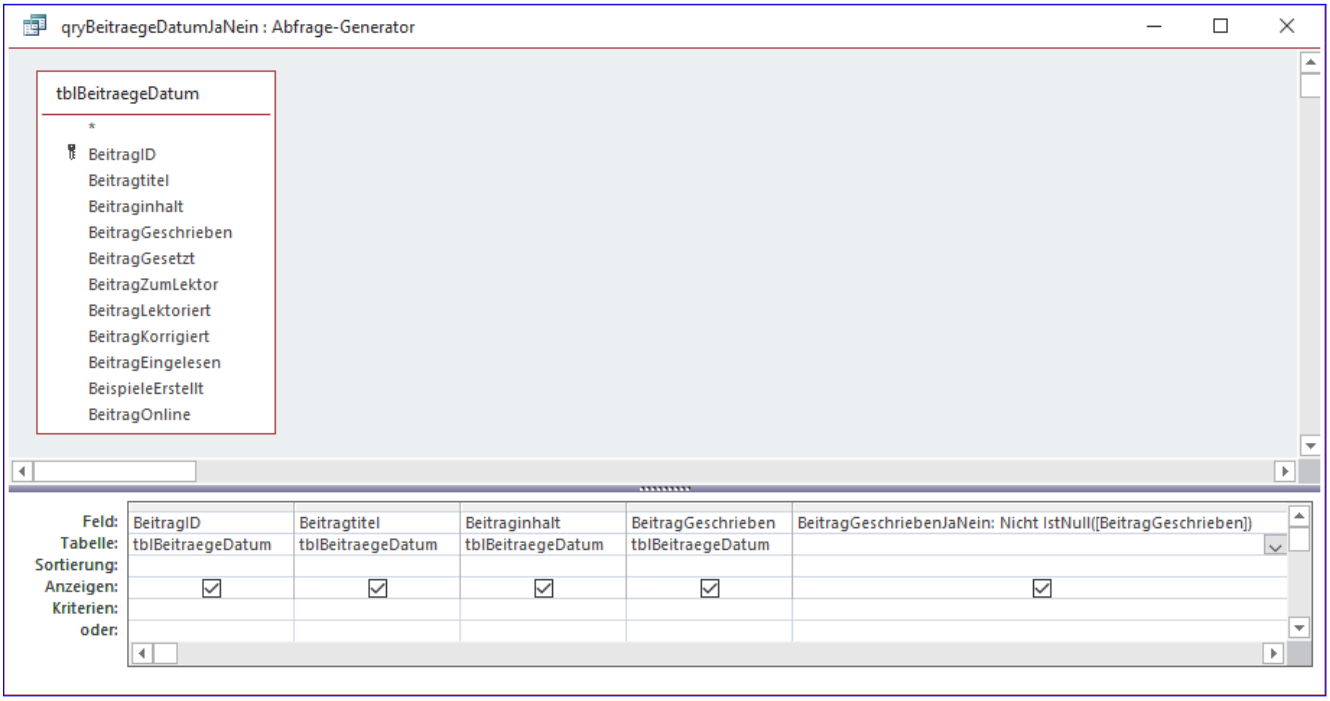


Bild 4: Abfrage, die sowohl das **Datum/Zeit**-Feld als auch ein davon abhängiges **Ja/Nein**-Feld liefert.

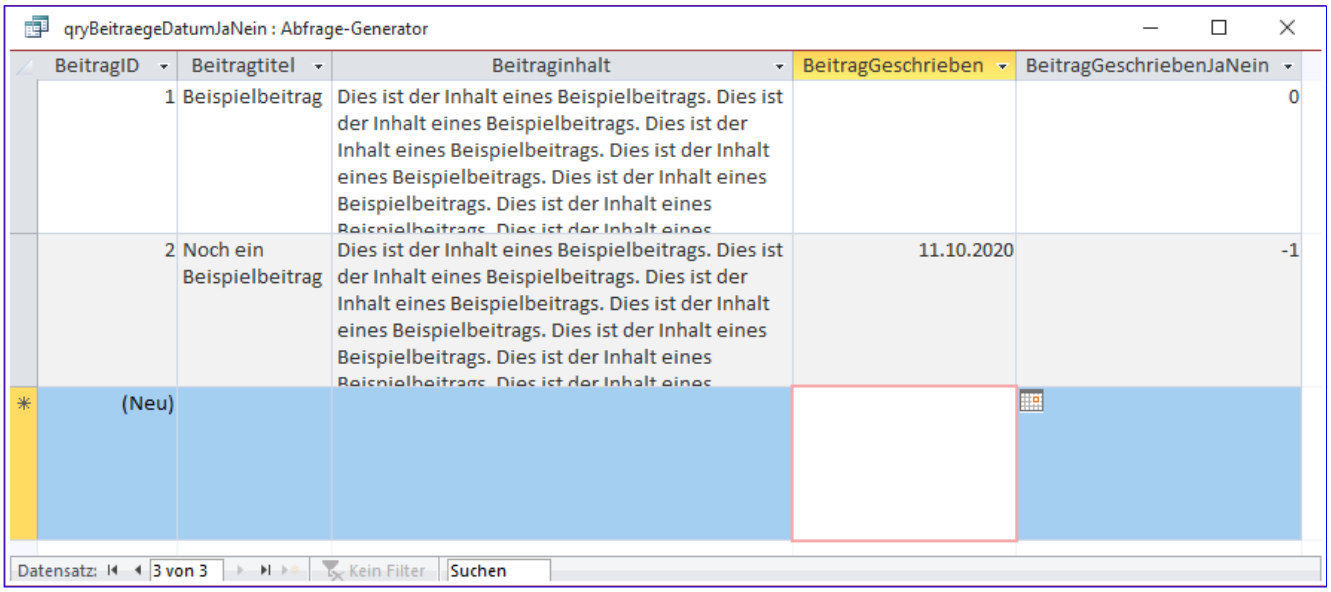


Bild 5: Ergebnis der Abfrage mit je einem gefüllten und einem nicht gefüllten **BeitragGeschrieben**-Feld

Nachschlagen der Eigenschaften einer Spalte der Abfrage. Hier finden wir allerdings nur die drei Möglichkeiten Textfeld, Listenfeld und Kombinationsfeld vor (siehe Bild 6). Anders ist es, wenn wir ein Abfragefeld auf Basis eines **Ja/Nein**-Feldes erstellen – dann stehen Kontrollkästchen, Textfeld und Kombinationsfeld zur Verfügung. Da dies hier nicht der Fall ist, müssen wir in den sauren Apfel beißen und vorerst mit Textfeldern arbeiten.

Bild 6: Die Anzeige als Kontrollkästchen ist nicht möglich.

Der Nachteil davon zeigt sich, wenn wir das nun in einen Formular abbilden wollen – wir hätten dort gern für die berechneten **Ja/Nein**-Felder gern direkt Kontrollkästchen erhalten, aber beim Hinzufügen im Formularentwurf übernimmt Access den in der Tabelle beziehungsweise Abfrage voreingestellten Steuerelementtyp.

Wenn wir die Felder aus der Abfrage, welche die **Ja/Nein**-Ausdrücke für die Datumsfelder berechnen, in das Formular ziehen und versuchen, diese in Kontrollkästchen umzuwandeln, finden wir auch hier diese Möglichkeit nicht vor (siehe Bild 7).

Also müssen wir wohl oder übel jeweils ein Kontrollkästchen für jedes Feld in den Formularentwurf ziehen und dann der Eigenschaft **Steuerelementinhalt** die berechneten **Ja/Nein**-Felder der als Datensatzquelle verwendeten Abfrage zuweisen. Außerdem benennen wir die Kontrollkästchen auch direkt mit dem passenden Präfix, also beispielsweise **chkBeitragGeschrieben**.

Für das berechnete Feld **BeitragGeschriebenJaNein** sieht das Kontrollkästchen wie in Bild 8 aus: Es hat den Namen **chkBeitragGeschrieben** und den Steuerelementinhalt **BeitragGeschriebenJaNein**.

Rechts daneben wollen wir nun noch das Datum abbilden für den Fall, dass die Unteraufgabe bereits erledigt wurde. Dazu fügen wir noch die entsprechenden Textfelder hinzu. Diese markieren wir in der Feldliste und ziehen sie gleich neben die bereits angelegten Kontrollkästchen. Die mit diesen Textfeldern ebenfalls zum Formularentwurf hinzugefügten Bezeichnungsfelder markieren wir und löschen diese direkt wieder. Die Textfelder platzieren wir so, dass sie sich direkt neben den Kontrollkästchen befinden.

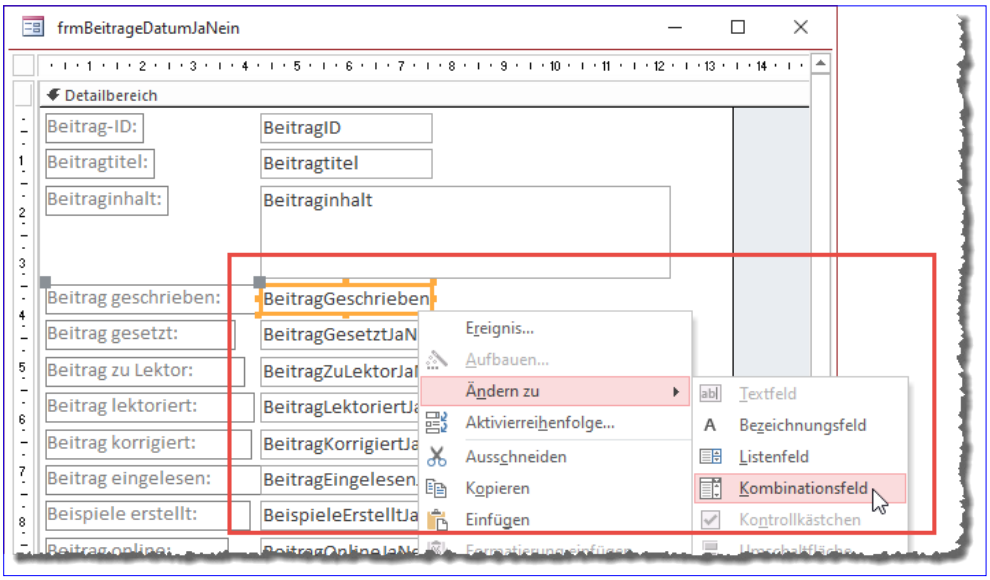


Bild 7: Textfelder können nicht in Kontrollkästchen umgewandelt werden

Die Namen der Textfelder ergänzen wir ebenfalls noch um das entsprechende Präfix, in diesem Fall txt. Das Textfeld für die erste Unteraufgabe heißt so beispielsweise **txtBeitragGeschrieben**.

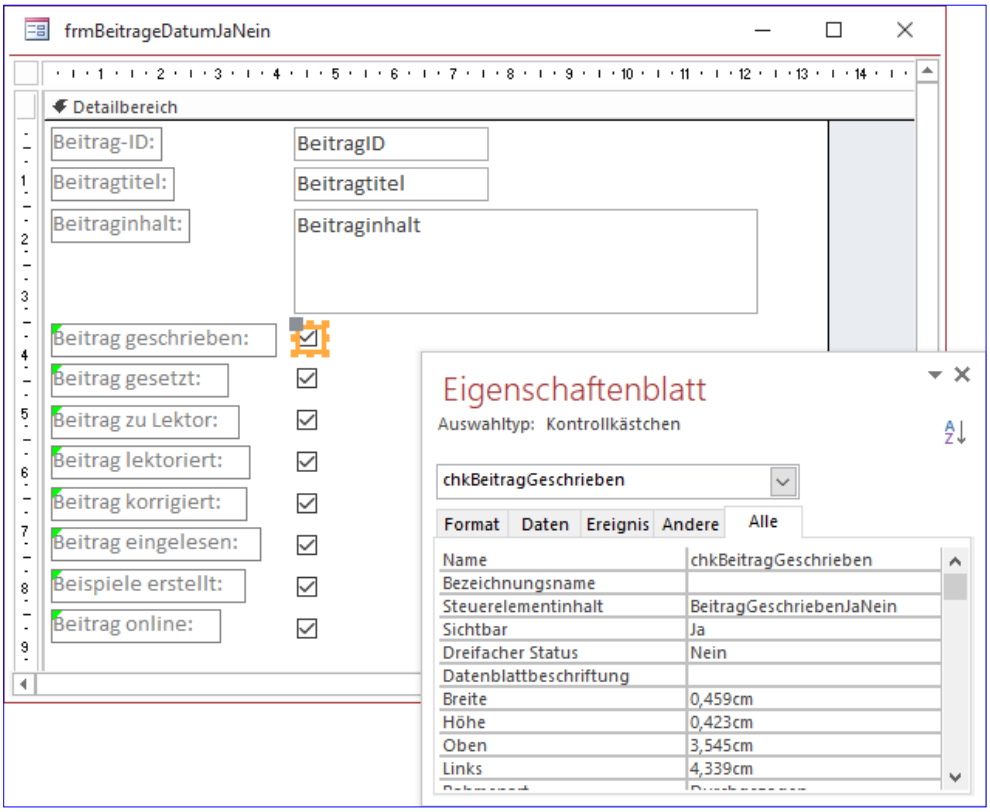


Bild 8: Kontrollkästchen für die Unteraufgaben

Datum für erledigte Aufgaben anzeigen

Nun fehlt noch Code, der dafür sorgt, dass die Textfelder nur erscheinen, wenn die jeweilige Unteraufgabe bereits erledigt wurde. Diesen platzieren wir in der Prozedur, die durch das Ereignis **Beim Anzeigen** des Formulars ausgelöst wird. Diese Prozedur finden Sie in Listing 1.

Hier berechnen wir für jedes Datumsfeld der Datensatzquelle, ob es den Wert Null hat und stellen die Eigenschaft Visible des dazugehörenden Textfeldes zur Anzeige des