HAUFE.
Ausgabe 04/2019

ACCESS
IM UNTERNEHMEN

INDIVIDUELLE REIHENFOLGE

Sortieren Sie Daten nach individuellen Kriterien in Listenfeldern und Datenblättern per Mausklick (ab S. 6)

In diesem Heft:

AUTOCOMPLETE IN TEXTFELDERN

Statten Sie Textfelder mit einer praktischen Autocomplete-Funktion aus.

SEITE 45

BUTTON VOM FORMULAR INS RIBBON

Verschieben Sie Steuerelemente vom Formular ins Ribbon und gewinnen Sie Platz für Ihre Daten.

SEITE 31

DATEN EFFEKTIV SCHÜTZEN

Kombinieren Sie verschiedene Techniken, um den Schutz Ihrer Daten zu optimieren.

SEITE 59

Eins nach dem anderen

Wenn Sie Datensätze sortieren wollen, gibt es noch andere Kriterien wie die auf- oder absteigende Reihenfolge nach dem Alphabet. Es gibt auch individuelle Eigenschaften, nach denen Sie Daten sortieren wollen – beispielsweise die Priorität einer Aufgabe. Diese weisen die Besonderheit auf, dass Sie dafür ein eigenes Feld anlegen, dessen Werte Sie zum Herstellen der gewünschten Reihenfolge anpassen. Interessant wird es, wenn es um eine benutzerfreundliche Möglichkeit geht, dies zu erledigen.

Genau darum kümmern wir uns in dieser Ausgabe gleich in zwei Beiträgen. Unter dem Titel **Listenfeld: Reihenfolge mehrerer Einträge ändern** schauen wir uns ab Seite 2 an, wie Sie die Reihenfolge von Einträge in einem Listenfeld ändern. Dort können Sie die markierten Einträge per Schaltfläche verschieben. Solche Lösungen haben wir schon in früheren Ausgaben vorgestellt. Der Unterschied ist: Diesmal können Sie nicht nur einen, sondern gleich mehrere Einträge im Listenfeld markieren und verschieben.

Der zweite Beitrag zu diesem Thema heißt **Datenblatt: Reihenfolge mehrerer Einträge ändern** (ab Seite 17). Hier wollen wir die Lösung des zuvor erwähnten Beitrags an die Darstellung im Datenblatt anpassen. Dazu sind mehr Schritte nötig, als man denken mag – so greift man etwa auf die markierten Einträge im Datenblatt ganz anders zu als auf die in einem Listenfeld.

Da diese Lösung im Datenblatt nur funktioniert, wenn die Einträge auch nach dem Feld sortiert sind, in dem die Sortierreihenfolge gespeichert wird, wollen wir verhindern, dass der Benutzer die Reihenfolge im Datenblatt verändern kann. Wie das gelingt, erläutern wir im Beitrag **Datenblattfunktionen einschränken** ab S. 55. Hier beschreiben wir nicht nur, wie Sie das Ändern der Sortierreihenfolge verhindern, sondern auch, wie Sie Manipulationen allgemein vorbeugen können – zum Beispiel durch Deaktivieren der dafür vorgesehenen Menüs.

Das Thema der neuen Diagramme von Access greifen wir ab Seite 2 im Beitrag **Diagramme mit gefilterten Daten** erneut auf. Diesmal schauen wir uns an, wie Sie die Daten im Diagram abhängig von einem Feld im übergeordneten Formular filtern können. So können Sie etwa nur die Umsätze für eine bestimmte Kategorie im Diagramm anzeigen.

Eine tolle und praktische Lösung liefert der Beitrag **Autocomplete in Textfeldern** ab Seite 45. Hier zeigen wir, wie Sie die Autocomplete-Funktion, die Sie bereits von gebundenen Kombinationsfeldern kennen, auch in Textfeldern einsetzen können. Damit braucht der Benutzer dann nur noch einen oder mehrere Buchstaben einzugeben und die Autocomplete-Funktion schlägt dann den nächsten bereits vorhandenen Eintrag der zugrunde liegenden Tabelle zum automatischen Ergänzen vor.

Das in den vorherigen Ausgaben begonnene Thema der Zugriffsrechte für die Daten einer Datenbank setzen wir ab Seite 59 im Beitrag **Tabellen vor unerlaubtem Zugriff schützen** fort. Hier zeigen wir, wie Sie die Daten im Backend per Kennwort schützen und das Frontend so vorbereiten, dass der Benutzer nicht mehr direkt auf die enthaltenen Tabellen zugreifen kann.

Und schließlich lernen Sie im Beitrag **Button vom Formular ins Ribbon** ab Seite 31 noch, wie Sie Formular schlanker gestalten, indem Sie Schaltflächen in das Ribbon übertragen.

Und nun: Viel Erfolg beim Sichern Ihrer Daten!

Ihr André Minhorst

Diagramme mit gefilterten Daten

In Ausgabe 2/2019 haben wir in zwei Artikeln die modernen Diagramme von Access ab Version 2016 vorgestellt. Im vorliegenen Beitrag zeigen wir Ihnen, wie Sie diese abhängig von den in einem Formular angezeigten Daten nutzen können. Dazu verwenden wir ein Kombinationsfeld zur Auswahl etwa einer Kategorie von Artikeln, zu denen wir dann die Umsätze über einen bestimmten Zeitraum ausgeben. Der Benutzer kann die Kategorie dann jeweils anpassen und so das Diagramm mit den entsprechenden Daten aktualisieren.

Umsätze nach einzelnen Kategorien

Im ersten Beispiel soll das Diagramm die Umsätze nach einer bestimmten Kategorie anzeigen, die der Benutzer über ein Kombinationsfeld auswählen kann. Dazu erstellen wir ein neues, leeres Formular und fügen im oberen Bereich ein Kombinationsfeld und im unteren Bereich ein Diagramm hinzu – und zwar über den Ribbon-Eintrag **Entwurf|Steuerelemente|Diagramm einfügen|Linie|Linie**.

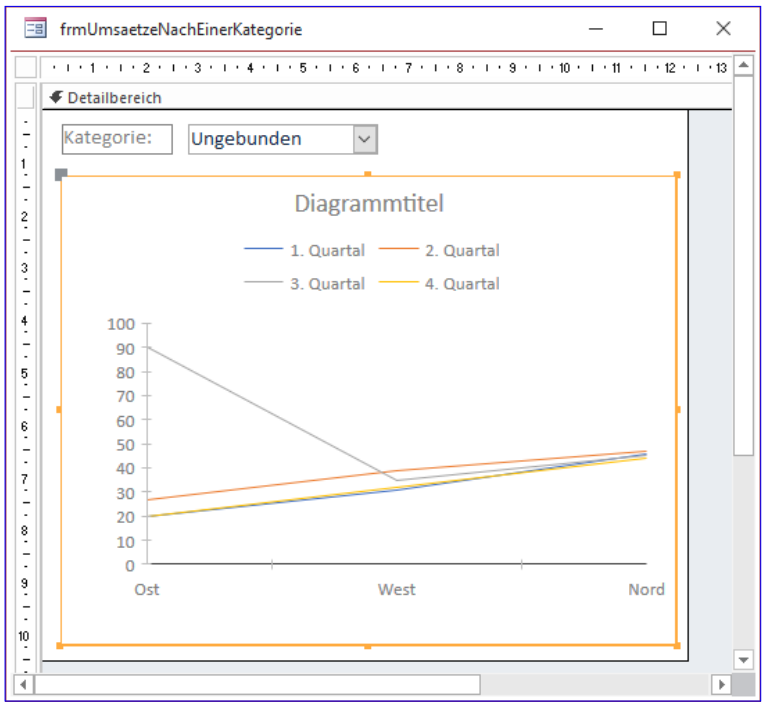


Bild 1: Entwurf des Formulars **frmUmsaetzeNachEinerKategorie**

Das Kombinationsfeld bezeichnen wir mit **cboKategorien**, das Diagramm-Steuerelement mit **dgrUmsaetze**. Der erste Entwurf sieht wie in Bild 1 aus. Das Kombinationsfeld bestücken wir über die Eigenschaft **Datensatzherkunft** mit der folgenden Abfrage:

```
SELECT KategorieID, Kategoriename FROM tblKategorien ORDER BY Kategoriename;
```

Damit das Kombinationsfeld nur den Wert der zweiten Spalte anzeigt, stellen wir die Eigenschaft **Spaltenanzahl** auf **2** und **Spaltenbreiten** auf **0cm** ein. Außerdem soll direkt beim Öffnen der erste Eintrag ausgewählt werden, was wir mit der folgenden Prozedur erreichen, die durch das Ereignis **Beim Laden** ausgelöst wird:

```
Private Sub Form_Load()
    Me!cboKategorien = Me!cboKategorien.ItemData(0)
End Sub
```

Datenquelle für das Diagramm definieren

Als Datenquelle für das Diagramm verwenden wir die Abfrage **qryUmsatzNachKategorieUndQuartal**. Diese sieht im Entwurf wie in Bild 2 aus und enthält drei Felder.

Das erste namens **Quartal** liefert die Bezeichnung des Quartals, die aus dem Bestelldatum gewonnen und als Gruppierung zusammengestellt wird. Das zweite Feld **Preis** steuert die Summe der Einzelpreise der Bestellpositionen bei. Das dritte Feld liefert die **KategorieID** des bestellten Artikels.

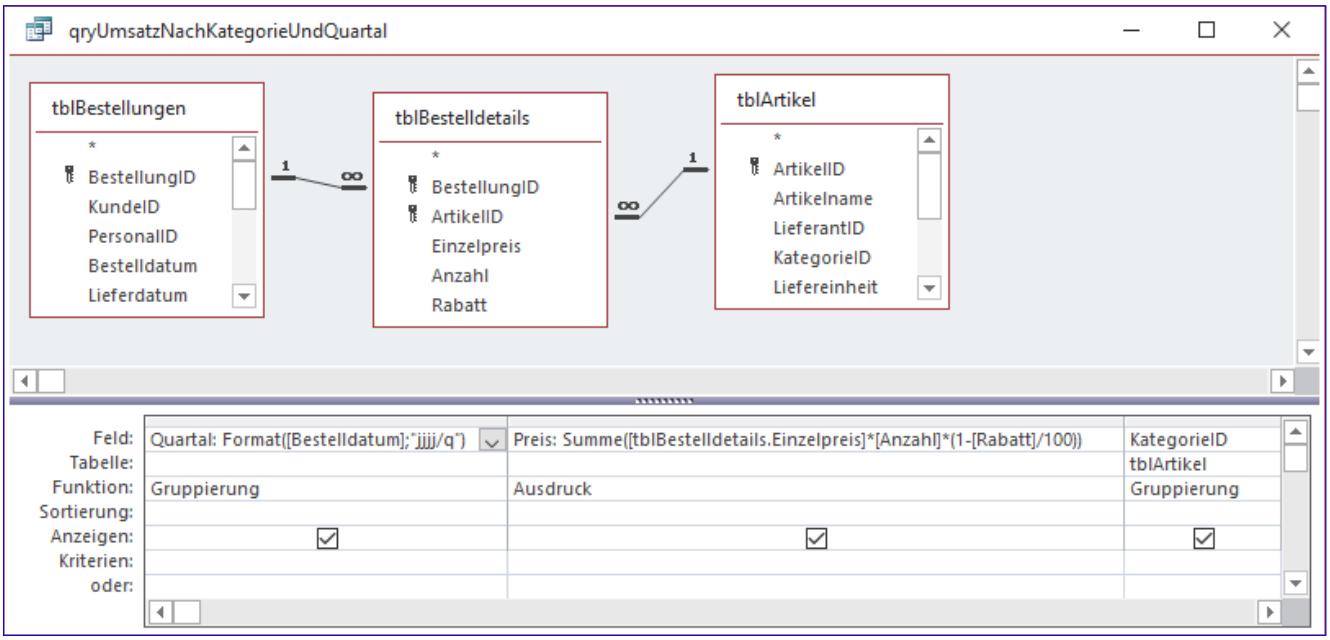


Bild 2: Entwurf der Abfrage **qryUmsatzNachKategorieUndQuartal**

Die Abfrage liefert so je einen Datensatz mit der Bestellsumme je Quartal und Kategorie. Wenn wir nun die Abfrage schließen und zum Entwurf des Formulars **frmUmsaetzeNachEinerKategorie** wechseln, erscheint auch der Bereich **Diagrammeinstellungen**. Hier wählen wir als Datenquelle die soeben erstellte Abfrage aus. Danach wenden wir uns dem Eigenschaftenblatt des Diagramm-Steuerelements zu. Hier stellen wir im Bereich Daten die Eigenschaft **Verknüpfen von** auf den Wert **KategorieID** und **Verknüpfen nach** auf den Wert **cboKategorien** ein (siehe Bild 3).

Diese Eigenschaften arbeiten genauso wie die eines Unterformular-Steuerelements: Sie sorgen dafür, dass im Diagramm immer die Daten angezeigt werden, die dem Wert aus dem unter **Verknüpfen nach** angegebenen Feld oder Steuerelement entsprechen.

Wenn Sie nun in die Formularansicht wechseln, können Sie durch Auswahl der verschiedenen Kategorien die Umsätze der einzelnen Quartale für diese Kategorie anzeigen (siehe Bild 4).

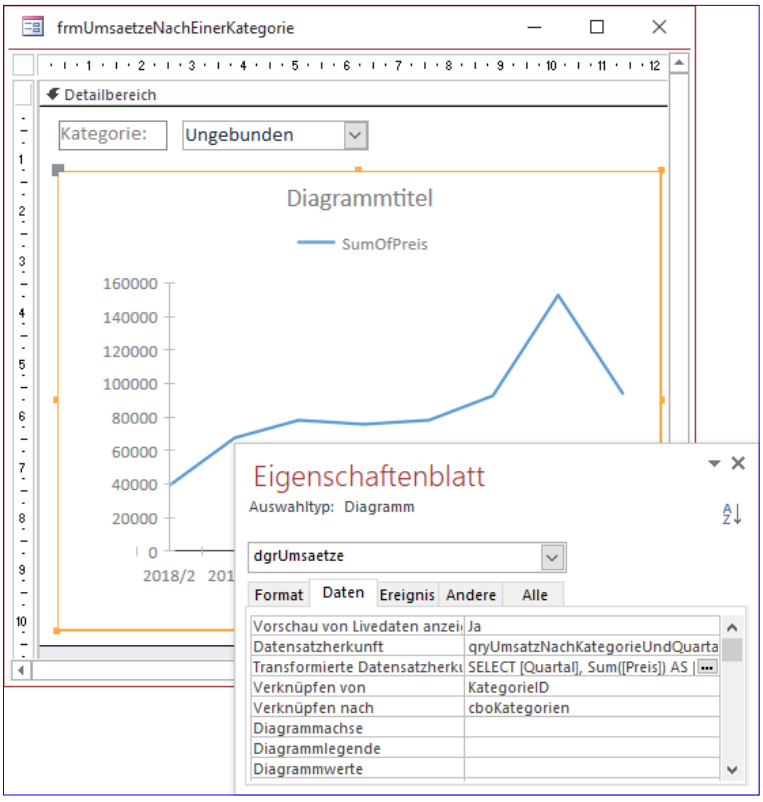


Bild 3: Einstellen der Eigenschaften zum Verknüpfen der Daten

Listenfeld: Reihenfolge mehrerer Einträge ändern

Wir haben bereits in mehreren Beiträgen beschrieben, wie Sie die individuelle Reihenfolge von Elementen einer Tabelle über den Inhalt eines Feldes etwa namens »ReihenfolgeID« einstellen können – zum Beispiel in Listenfeldern oder Unterformularen in der Datenblattansicht. Dort haben wir die Reihenfolge dann durch Markieren der Einträge und anschließendes Betätigen etwa von Schaltflächen mit Beschriftungen wie »Ganz nach oben«, »Nach oben«, »Nach unten« oder »Ganz nach unten« geändert. Im vorliegenden Beitrag schauen wir uns nun an, wie wir im Listenfeld die Reihenfolge für mehrere Einträge gleichzeitig ändern können.

Mehrfach-Reihenfolge im Listenfeld

Listenfelder haben gegenüber der Datenblattansicht den großen Vorteil, dass Sie damit nicht nur zusammenhängende Einträge markieren können, sondern sogar Einträge, die beliebig im Listenfeld verteilt sind. Die einzige Voraussetzung dafür ist, dass Sie für die Eigenschaft **Mehrfachauswahl** des Listenfeldes den Wert **Einzeln** oder **Erweitert** auswählen. Welchen Sie nutzen, hängt von den Anforderungen und den Gewohnheiten der Benutzer ab – für die hier vorgestellte Lösung funktionieren beide gleich gut.

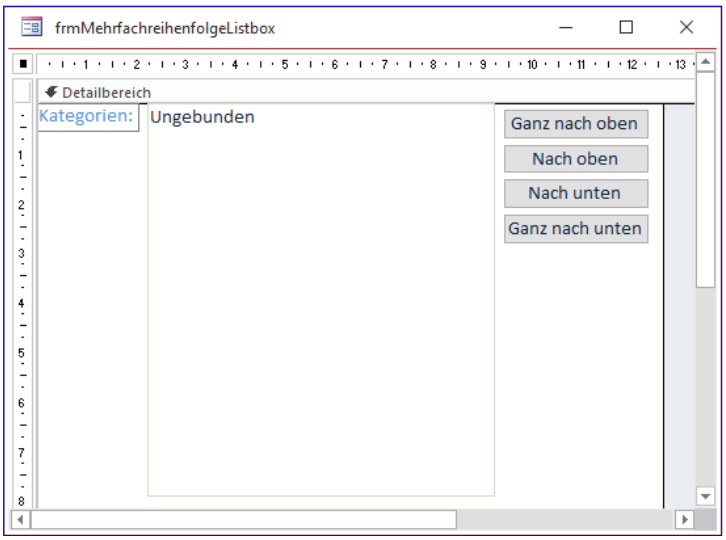


Bild 1: Entwurf des Formulars **frmMehrreihenfolgeListBox**

Entwurf des Beispielformulars

Das Beispielformular bauen wir wie in Bild 1 auf. Wir fügen ein Listenfeld namens **lstKategorien** ein sowie vier Schaltflächen namens **cmdTop**, **cmdUp**, **cmdDown** und **cmdBottom**.

Außerdem hinterlegen wir als **Datensatzherkunft** für das Listenfeld die Tabelle **tblKategorien**, deren Entwurf wie in Bild 2 aussieht. Genau genommen verwenden wir sogar eine Abfrage auf Basis dieser Tabelle, welche die Datensätze nach dem Feld **ReihenfolgeID** sortiert:

```
SELECT tblKategorien.KategorieID, tblKa-
tegorien.Kategoriename, tblKategorien.
```

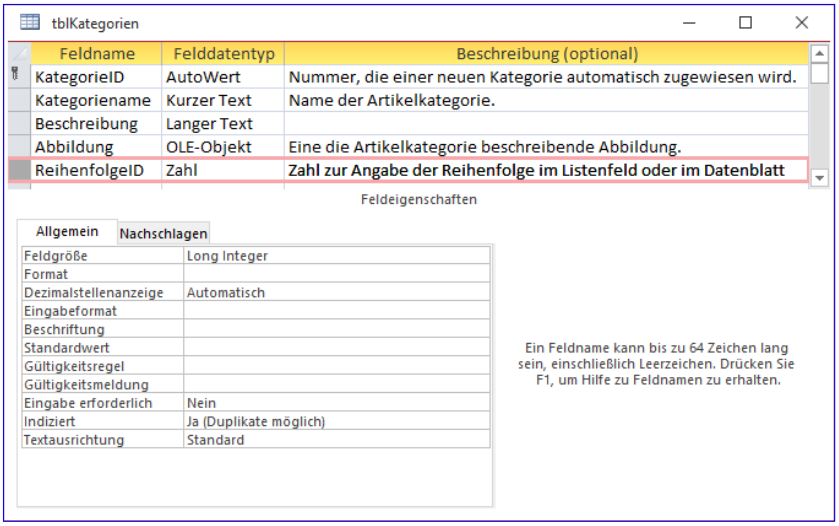


Bild 2: Entwurf der Tabelle **tblKategorien**

```
ReihenfolgeID FROM tblKategorien ORDER BY tblKategorien.
ReihenfolgeID:
```

Die Eigenschaft **Spaltenanzahl** des Listenfeldes stellen wir auf **3** ein, die Eigenschaft **Spaltenbreiten** auf **0cm;5cm**. Dadurch erscheinen nur das zweite und das dritte Feld der Datensatzherkunft im Listenfeld, also die Felder **Kategoriename** und **ReihenfolgeID**.

Definition des Änderns der Reihenfolge

Bevor wir uns an die Programmierung begeben, müssen wir noch herausfinden, wie die Reihenfolge überhaupt geändert werden soll. Bei einem einzelnen Element, dessen Reihenfolge verschoben werden soll, ist das kein Problem: Hier ermitteln wir die **ReihenfolgeID** des zu verschiebenden Elements und den des Zielelements und nehmen entsprechende Änderungen an den Inhalten dieses Felds für die betroffenen Datensätze vor.

Bei mehreren gleichzeitig zu verschiebenden Datensätzen, die darüberhinaus noch nicht einmal zusammenhängen müssen, ist allerdings zuvor zu definieren, welcher Datensatz wo landet. Wenn Sie beispielsweise den zweiten und den dritten Eintrag markieren und die Schaltfläche **Ganz nach oben** betätigen, ist die Aufgabe klar: Dann sollen der zweite und der dritte Datensatz die erste und die zweite Position einnehmen und der erste auf die dritte Position verschoben werden:

```
Vorher:     Nachher:
Eintrag 1   Eintrag 2
Eintrag 2   Eintrag 3
Eintrag 3   Eintrag 1
...         ...
```

Gleiches gilt, wenn diese beiden Datensätze etwa mit der Schaltfläche **Nach unten** um eine Position nach unten verschoben werden sollen: Dann landen der zweite und der dritte Eintrag auf der dritten und vierten Position und der vierte Eintrag wird nach oben auf die zweite Position verschoben:

```
Vorher:     Nachher:
Eintrag 1   Eintrag 1
Eintrag 2   Eintrag 4
Eintrag 3   Eintrag 2
Eintrag 4   Eintrag 3
...         ...
```

Was aber, wenn der Benutzer etwa den zweiten und den vierten Eintrag verschiebt? Dann gibt es zwei Möglichkeiten:

- Die Einträge werden in dem Raster, in dem sie sich gerade befinden, nach oben verschoben.
- Die zu verschiebenden Einträge werden zu einem Block zusammengefasst und an die Zielposition verschoben.

Beides lässt sich wieder besser am Beispiel erläutern. Angenommen, wir wollen den zweiten und den vierten Eintrag nach der ersten Methode um eine Position nach oben verschieben, sieht das so aus:

```
Vorher:     Nachher:
Eintrag 1   Eintrag 2
Eintrag 2   Eintrag 1
Eintrag 3   Eintrag 4
Eintrag 4   Eintrag 3
```

Die Einträge werden also nach oben verschoben, behalten aber die Abstände zueinander bei. Oder wir verschieben einfach alle markierten Elemente an die Zielposition und alle verdrängten Elemente nach unten:

```
Vorher:     Nachher:
Eintrag 1   Eintrag 2
Eintrag 2   Eintrag 4
Eintrag 3   Eintrag 1
Eintrag 4   Eintrag 3
```

Da die zweite Methode einfacher zu programmieren ist, werden wir diese verwenden. Auch, weil uns kein Anwen-

dungsfall einfällt, bei dem man nach der erstgenannten Methode verfahren sollte.

Code für Formular und Listenfeld

Im Klassenmodul des Formulars legen wir zunächst drei Konstanten an, die wichtige Daten aufnehmen, die an vielen Stellen verwendet werden – zum Beispiel den Namen der Quelltabelle, des Primärschlüsselfeldes dieser Tabelle und des Reihenfolge-Feldes.

Auf diese Weise brauchen Sie diese Bezeichnungen nur an einer Stelle zu ändern, wenn Sie die Reihenfolge-Funktionen einmal in ein anderes Formular übernehmen wollen:

```
Const m_Table As String = "tblKategorien"
Const m_PKField As String = "KategorieID"
Const m_Orderfield As String = "ReihenfolgeID"
```

Außerdem deklarieren wir eine Variable namens **m_ListBox**, in der wie einen Verweis auf das Listenfeld speichern, das die Datensätze in der Reihenfolge anzeigen soll:

```
Dim m_ListBox As ListBox
```

Diese Variable füllen wir gleich beim Laden des Formulars in der folgenden Prozedur mit einem Verweis auf das gewünschte Listenfeld:

```
Private Sub Form_Load()
    Set m_ListBox = Me!lstKategorien
    FillOrderID
End Sub
```

Hier rufen wir auch gleich die Prozedur **FillOrderID** auf. Diese sorgt dafür, dass die Werte des Feldes **ReihenfolgeID** aktualisiert werden, so dass diese durchnummeriert sind.

ReihenfolgeID aktualisieren

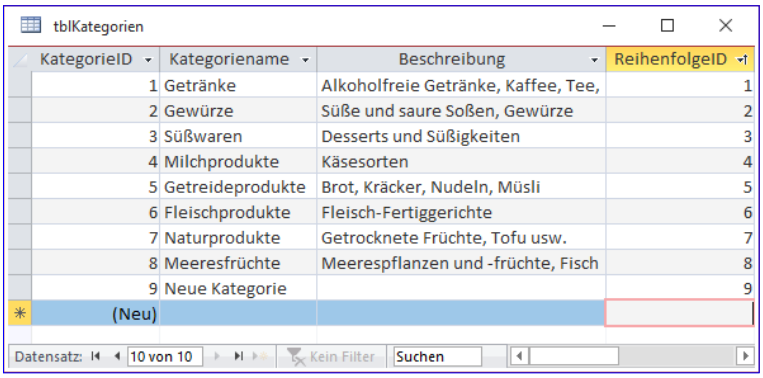
Die Prozedur **FillOrder** sieht wie in Listing 1 aus. Diese Prozedur greift die Inhalte der Konstanten **m_PKField**, **m_Orderfield** und **m_Table** auf und stellt damit ein Recordset zusammen, das die Datensätze der angegebenen Tabelle, hier **tblKategorien**, mit den interessanten Feldern **KategorieID** und **ReihenfolgeID** enthält. Dieses Recordset durchläuft die Prozedur in einer **Do While**-Schleife und aktualisiert dort den Wert des Feldes aus **m_Orderfield**, hier **ReihenfolgeID**, mit dem Wert der Eigenschaft

```
Public Sub FillOrderID()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Set db = CodeDb
    Set rst = db.OpenRecordset("SELECT " & m_PKField & ", " & m_Orderfield & " FROM " & m_Table & " ORDER BY " _
        & m_Orderfield, dbOpenDynaset)
    Do While Not rst.EOF
        rst.Edit
        rst(m_Orderfield) = rst.AbsolutePosition + 1
        rst.Update
        rst.MoveNext
    Loop
    rst.Close
    Set rst = Nothing
    Set db = Nothing
End Sub
```

Listing 1: Prozedur zum Wiederherstellen der Werte des Feldes **ReihenfolgeID**

AbsolutePosition des Recordsets und addiert noch den Wert **1** hinzu. Dadurch werden die Datensätze der Tabelle mit Werten für das Feld **ReihenfolgeID** durchnummeriert.

Das Ergebnis sieht dann wie in Bild 3 aus.



tblKategorien

KategorieID	Kategoriename	Beschreibung	ReihenfolgeID
1	Getränke	Alkoholfreie Getränke, Kaffee, Tee,	1
2	Gewürze	Süße und saure Soßen, Gewürze	2
3	Süßwaren	Desserts und Süßigkeiten	3
4	Milchprodukte	Käsesorten	4
5	Getreideprodukte	Brot, Kräcker, Nudeln, Müsli	5
6	Fleischprodukte	Fleisch-Fertiggerichte	6
7	Naturprodukte	Getrocknete Früchte, Tofu usw.	7
8	Meeresfrüchte	Meerespflanzen und -früchte, Fisch	8
9	Neue Kategorie		9
(Neu)			

Datensatz: 10 von 10 Kein Filter Suchen

Bild 3: Die Tabelle **tblKategorien** mit Werten im Feld **ReihenfolgeID**

Schaltflächen mit Code versehen

Die vier Schaltflächen versehen wir mit jeweils einer Prozedur, welche die Reihenfolge der aktuell markierten Einträge aktualisiert. Die Prozedur für die oberste Schaltfläche **cmdTop** lautet wie folgt:

```
Private Sub cmdTop_Click()
    Dim lngTarget As Long
    Dim lngMove() As Long
    lngMove = GetMove
    lngTarget = DMin(m_Orderfield, m_Table)
    InterchangeOrder lngTarget, lngMove
    RequeryControls lngMove
End Sub
```

Hier rufen wir als Erstes die Funktion **GetMove** auf. Diese ist dafür verantwortlich, ein Array zusammenzustellen, dass alle Primärschlüsselwerte der zu verschiebenden Einträge des Listenfeldes aufnimmt (siehe Listing 2). Die Funktion durchläuft dazu eine **For Each**-Schleife über alle ausgewählten Elemente des Listenfeldes (**ItemsSelected**), das wir mit **m_ListBox** referenziert haben. Der 0-basierte Index des jeweils durchlaufenen Elements wird in der Variablen **var** gespeichert. Darin wird zuerst das Array **lngMove** auf eine Anzahl von Einträgen entsprechend dem Wert der Variablen **i** erweitert. Dann weisen wir dem Element mit dem Index **i** den Wert der ersten Spalte des Listenfeldes zu (**ItemData(var)**). Schließlich erhöhen wir den Wert der Variablen **i** um eins, bevor wir den nächsten ausgewählten Eintrag untersuchen.

```
Private Function GetMove() As Long()
    Dim lngMove() As Long
    Dim i As Integer
    Dim var As Variant
    For Each var In m_ListBox.ItemsSelected
        ReDim Preserve lngMove(i)
        lngMove(i) = m_ListBox.ItemData(var)
        i = i + 1
    Next var
    GetMove = lngMove
End Function
```

Listing 2: Die Prozedur **GetMove**

Die nächste Anweisung ermittelt mit der **DMin**-Funktion den kleinsten Wert des Feldes aus **m_Orderfield** (hier **ReihenfolgeID**) der Tabelle aus **m_Table** (hier **tblKategorien**) und speichert diesen in der Variablen **lngTarget**. Das ist in diesem Fall das Ziel beim Verschieben des Elements. Danach folgt der Aufruf der Prozedur **InterchangeOrder** mit **lngTarget** und dem Array **lngMove** als Parameter. Diese schauen wir uns weiter unten an. Die Prozedur stellt die **ReihenfolgeID**-Werte für die betroffenen Elemente neu ein.

Schließlich ruft die Prozedur noch eine weitere Routine namens **RequeryControls** auf. Diese aktualisiert zuerst den Inhalt des Listenfeldes, wodurch die Elemente nach dem Ändern der Werte des Feldes **ReihenfolgeID** durch die vorgegebene Sortierung gegebenenfalls neu angeordnet werden. Dann ruft sie die Prozedur **SelectEntries** auf, die dafür sorgt, dass die vor dem Verschieben markierten Einträge erneut markiert werden. Durch das **Requery** gehen die Markierungen nämlich verloren. Schließlich sorgt die Prozedur **ActivateControls** dafür, dass die Schaltflä-

chen entsprechend der aktuell ausgewählten Elemente aktiviert oder deaktiviert werden. Wenn nämlich das erste Element markiert ist, sollen die Schaltflächen **Ganz nach oben** und **Nach oben** deaktiviert werden. Wenn das letzte Element markiert ist, sollen die Schaltflächen **Nach unten** und **Ganz nach unten** deaktiviert sein:

```
Private Sub RequeryControls(lngMove() As Long)
    m_ListBox.Requery
    SelectEntries lngMove()
    ActivateControls
End Sub
```

Schaltfläche »Nach oben«

Bevor wir uns die hier erwähnten Prozeduren ansehen, werfen wir noch einen Blick auf den Code der übrigen Schaltflächen. Zunächst die Schaltfläche, mit der die aktuell markierten Einträge um eine Position nach oben verschoben werden sollen:

```
Private Sub cmdUp_Click()
    Dim lngTarget As Long, lngTargetMove As Long
    Dim lngMove() As Long
    lngMove = GetMove
    lngTargetMove = DLookup(m_Orderfield, m_Table, _
        m_PKField & "=" & lngMove(0))
    lngTarget = DMax(m_Orderfield, m_Table, m_Orderfield _
        & "<" & lngTargetMove)
    InterchangeOrder lngTarget, lngMove
    RequeryControls lngMove
End Sub
```

Der Unterschied ist, dass wir den Wert für **lngTarget**, also für die Zielposition, auf etwas aufwendigere Weise ermitteln, und zwar mit zwei Domänenfunktionen. Die erste **DLookup**-Funktion ermittelt den Wert des Feldes **ReihenfolgeID** für den Datensatz der Tabelle **tblKategorien**, der dem ersten markierten Datensatz entspricht. Wenn also die Datensätze mit den Primärschlüsselwerten **2** und **3** markiert sind, die in diesem Fall auch die Positionen **2** und **3** innehaben, dann ist der mit dem Primärschlüsselwert **2** der aus **lngMove(0)**, also der erste Wert des Arrays. Für diesen ermittelt die **DLookup**-Funktion dann den Wert des Feldes **ReihenfolgeID**, hier auch **2**, und schreibt ihn in die Variable **lngTargetMove**. Die folgende **DMax**-Funktion ermittelt dann den größten Wert des Feldes **ReihenfolgeID**, der kleiner als der Wert aus **lngTargetMove** ist. Damit ist dann die Ziel-Reihenfolgenposition ermittelt und in der Variablen **lngTarget** gespeichert. Mit **lngTarget** und **lngMove()** wird dann wiederum die Prozedur **InterchangeOrder** aufgerufen und dann die Prozedur **RequeryControls**.

Schaltfläche »Ganz nach unten«

Die durch diese Schaltfläche ausgelöste Prozedur ist wieder genauso einfach wie die für die Schaltfläche **Ganz nach oben**. Hier verwenden wir lediglich für die Variable **lngTarget** den maximalen Wert des Feldes **ReihenfolgeID** der Tabelle **tblKategorien**:

```
Private Sub cmdBottom_Click()
    Dim lngTarget As Long
    Dim lngMove() As Long
    lngMove = GetMove
    lngTarget = DMax(m_Orderfield, m_Table)
    InterchangeOrder lngTarget, lngMove
    RequeryControls lngMove
End Sub
```

Schaltfläche »Nach unten«

Etwas komplizierter ist dagegen die Schaltfläche mit der Beschriftung **Nach unten** (siehe Listing 3). Hier ermitteln wir wieder das Array mit den Primärschlüsselwerten der markierten Elemente. Daraus berechnen wir dann über die Funktion **UBound** den größten Index-Wert und speichern ihn in **lngMax**. Diesen nutzen wir dann in einer **DLookup**-Funktion als Vergleichswert für das Kriterium, mit der wir den **ReihenfolgeID**-Wert für den untersten markierten Eintrag ermitteln. Die folgende **DMin**-Funktion liest dann den kleinsten **ReihenfolgeID**-Wert der Tabelle **tblKategorien** ein, der größer ist als der **ReihenfolgeID**-Wert aus **lngTargetMove**. Der Rest entspricht der Vorgehensweise der vorherigen Prozeduren.

Reihenfolge vertauschen

Diese Prozedur erwartet die **ReihenfolgeID** für das Element, vor oder hinter dem die zu verschiebenden Elemente landen sollen (**lngTargetOrderID**) und das Array mit den Primärschlüsselwerten der zu verschiebenden Elemente als Parameter. Ob die Elemente vor oder hinter das Zielelement verschoben werden, hängt von der Richtung ab: Wenn die Elemente nach oben verschoben werden, landen Sie vor dem Zielelement, wenn Sie nach unten verschoben werden, hinter dem Zielelement. Die Prozedur durchläuft im ersten Teil (siehe Listing 4) zuerst eine **For... Next**-Schleife, in der die einzelnen Elemente aus dem Array **lngMove** zu einer kommaseparierten Liste zusammengefügt werden. Anschließend sieht der Inhalt etwa so aus:

```
Private Sub cmdDown_Click()
    Dim lngTarget As Long
    Dim lngMove() As Long
    Dim lngTargetMove As Long
    Dim lngMax As Long
    lngMove = GetMove
    lngMax = UBound(lngMove)
    lngTargetMove = DLookup(m_Orderfield, m_Table, m_PKField & "=" & lngMove(lngMax))
    lngTarget = DMin(m_Orderfield, m_Table, m_Orderfield & ">" & lngTargetMove)
    InterchangeOrder lngTarget, lngMove
    RequeryControls lngMove
End Sub
```

Listing 3: Prozedur zum Verschieben der markierten Einträge nach unten

```
, 1, 2, 3
```

Von dieser Zeichenkette trennt die folgende **Mid**-Funktion die ersten beiden Zeichen ab, also das Komma und das Leerzeichen. Die folgende Prozedur fügt dann vorn die **IN**-Klausel an und fasst die Liste der Primärschlüsselwerte in runde Klammern ein. Das Ergebnis sieht dann etwa so aus:

```
IN (1, 2, 3)
```

Danach ermitteln wir per **DLookup** den **ReihenfolgeID**-Wert für das erste markierte Element aus dem Array **lngMove** und speichern es in der Variablen **lngTargetMove**. Ist der Parameter **lngTargetOrderID**, der die **ReihenfolgeID** mit dem Ziel des Verschiebens angibt, kleiner als **lngTargetMove**, dann soll das Verschieben offensichtlich nach oben erfolgen. In diesem Fall kommt der erste Teil der nachfolgenden **If...Then**-Bedingung zum Einsatz, anderenfalls der **Else**-Teil. Im **If**-Teil stellen wir eine SQL-Abfrage zum Ermitteln der zu verschiebenden Elemente zusammen. Wenn wir etwa die Elemente mit den Primärschlüsselwerten **4**, **6** und **8** um eine Position nach oben verschieben wollen, resultiert das in der folgenden SQL-Abfrage:

```
SELECT * FROM tblKategorien WHERE KategorieID IN (4, 6, 8)
ORDER BY ReihenfolgeID
```

Auf Basis dieser Abfrage erstellt die Prozedur dann ein Recordset namens **rstVerschieben**. Vor dem Ausführen der **Do While**-Schleife über alle Datensätze des Recordsets tragen wir den Wert aus **lngTargetOrderID** in die Zählervariable **j** ein. Danach beginnen wir in der Schleife mit dem Anpassen zunächst der **ReihenfolgeID**-Werte der zu verschiebenden Datensätze. Dazu versetzen wir den aktuellen Datensatz mit der **Edit**-Methode in den Bearbeiten-Zustand. Dann stellen wir den Wert des Feldes aus **m_Orderfield**, hier **ReihenfolgeID**, auf den Wert aus der Variablen **j** ein. Für den ersten Datensatz, der im Beispiel den Wert **4** für das Feld **ReihenfolgeID** aufweist, erhält dieses Feld nun den Wert **3**. Nach dem Speichern der Änderungen erhält der Datensatz mit dem Wert **6** im Feld **ReihenfolgeID** den Wert **4**, also **j + 1**. Nach dem Einstellen der **ReihenfolgeID**-Werte für die zu verschiebenden Einträge müssen natürlich noch die für die übrigen

Datenblatt: Reihenfolge mehrerer Einträge ändern

Wir haben bereits in mehreren Beiträgen beschrieben, wie Sie die individuelle Reihenfolge von Elementen einer Tabelle über den Inhalt eines Feldes etwa namens »ReihenfolgeID« einstellen können – zum Beispiel in Listenfeldern oder Unterformularen in der Datenblattansicht. Dort haben wir die Reihenfolge dann durch Markieren der Einträge und anschließendes Betätigen etwa von Schaltflächen mit Beschriftungen wie »Ganz nach oben«, »Nach oben«, »Nach unten« oder »Ganz nach unten« geändert. Im vorliegenden Beitrag schauen wir uns nun an, wie wir im Unterformular in der Datenblattansicht die Reihenfolge für mehrere Einträge gleichzeitig ändern können.

Mehrfach-Reihenfolge im Datenblatt

Im Beitrag **Listenfeld: Reihenfolge mehrerer Einträge ändern** (**www.access-im-unternehmen.de/1197**) zeigen wir, wie das Ändern der Reihenfolge mehrerer markierter Einträge gleichzeitig im Listenfeld funktioniert. Aber oft sollen Daten auch in einem Unterformular in der Datenblattansicht dargestellt werden, was Vorteile gegenüber einem Listenfeld bietet – zum Beispiel das schnelle Sortieren, Filtern oder das Ändern der Spaltenanordnung oder -breite.

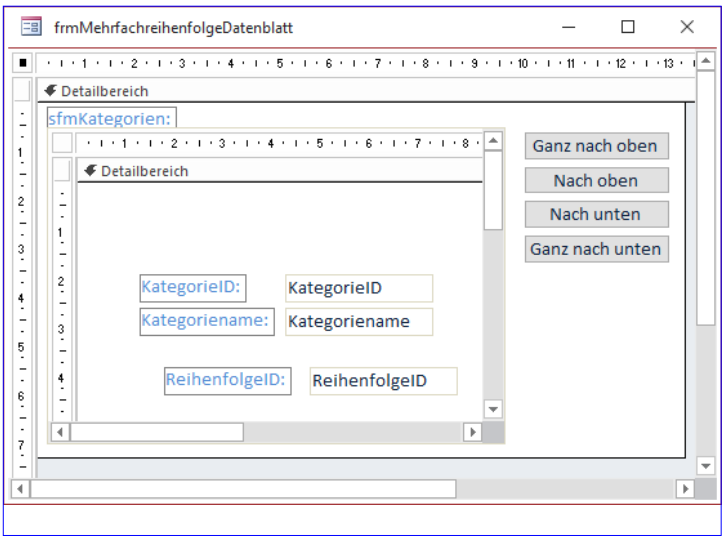


Bild 1: Entwurf des Formulars **frmMehrreihenfolgeDatenblatt**

Außerdem kann der Benutzer in einem Unterformular in der Datenblattansicht auch direkt Daten ändern, was im Listenfeld nicht möglich ist. Also zeigen wir auch für das Unterformular in der Datenblattansicht, wie Sie die Reihenfolge der markierten Datensätze ändern können.

Beispielformular

Die Beispielkonstellation sieht wie in Bild 1 aus. Hier haben wir ein Unterformular namens **sfmMehrfachreihenfolgeDatenblatt** erstellt, dass an die Tabelle **tblKategorien** gebunden ist – und zwar über die folgende SQL-Abfrage:

```
SELECT tblKategorien.KategorieID, tblKategorien.Kategoriename, tblKategorien.ReihenfolgeID FROM tblKategorien ORDER BY tblKategorien.ReihenfolgeID;
```

Dadurch werden die Datensätze aufsteigend nach der Sortierung des Feldes **ReihenfolgeID** angezeigt.

Das Unterformular soll die Daten der drei Felder **KategorieID**, **Kategoriename** und **ReihenfolgeID** anzeigen. Für die Eigenschaft **Standardansicht** haben wir den Wert **Datenblatt** eingestellt.

Danach haben wir das Unterformular aus dem Navigationsbereich von Access in den Entwurf des Hauptformulars **frmMehrfachreihenfolgeDatenblatt** gezogen. Das Unterformularsteuerelement, in dem das Unterformular angezeigt wird, haben wir aus Gründen der Übersicht in **sfm** umbenannt.

Rechts neben dem Unterformular befinden sich die vier Schaltflächen **cmdTop**, **cmdUp**, **cmdDown** und **cmdBottom**. Die Eigenschaften **Navigationsschaltflächen**, **Datensatzmarkierer**, **Trennlinien** und **Bildlaufleisten** des Hauptformulars haben wir auf **Nein** eingestellt, die Eigenschaft **Automatisch zentrieren** auf **Ja**.

Beschreibung der Aufgabe

Im Gegensatz zum Listenfeld können wir mit der Datenblattansicht nur zusammenhängende Datensätze markieren. Dafür gibt es beim Datenblatt ganz andere Möglichkeiten, um die jeweils markierten Datensätze zu identifizieren. Diese werden Sie in den folgenden Abschnitten kennenlernen.

Im ersten Schritt wollen wir sicherstellen, dass jeweils nur die aktuell notwendigen Schaltflächen aktiviert sind. Das heißt, dass die beiden Schaltflächen **cmdTop** und **cmdUp** deaktiviert werden sollen, wenn der Benutzer den obersten Datensatz im Datenblatt markiert hat, denn dann können die markierten Elemente nicht mehr weiter nach oben verschoben werden. Das gleiche gilt für die Schaltflächen **cmdDown** und **cmdBottom**, wenn der letzte Datensatz im Datenblatt markiert ist. Wenn der Benutzer alle Datensätze markiert, sollen dementsprechend alle vier Schaltflächen deaktiviert sein.

Eine besondere Herausforderung besteht darin, dass beim Wechseln der Markierung im Unterformular Steuerelemente im Hauptformular geändert sollen. Es ist zwar einfach möglich, dies über eine entsprechende Ereignisprozedur im Unterformular zu realisieren. Diese würde dann etwa über **Me.Parent.cmdUp** auf die Schaltfläche **cmdUp** zugreifen. Schöner und wartungsfreundlicher ist es allerdings, wenn der Code im Hauptformular versammelt ist. Das heißt, dass wir im Hauptformular eine Objektvariable zum Referenzieren des Unterformulars deklarieren und für diese Objektvariable die Ereignisse implementieren, die normalerweise im Klassenmodul des Unterformulars gelandet wären.

Aktivieren und Deaktivieren der Schaltflächen

Die Schaltflächen sollen in Abhängigkeit von den aktuell markierten Einträgen des Unterformulars aktiviert und deaktiviert werden. Das Problem dabei ist, dass wir erst noch ein Ereignis finden müssen, dass beim Setzen der Markierung im Unterformular ausgelöst wird. Das Ereignis **Bei Markierungsänderung** hört sich spannend an, aber es kann nur in der veralteten **PivotChart**- oder **PivotTable**-Ansicht eingesetzt werden.

Also nutzen wir die **Bei Maustaste auf**-Ereigniseigenschaft eines der Bereiche des Unterformulars. Wir finden schnell heraus, dass dieses Ereignis für das Formular selbst ausgelöst wird, wenn wir auf einen der Datensatzmarkierer klicken, also auf den in Bild 2 markierten Bereich. Im Unterformular selbst sieht die Ereignisprozedur wie folgt aus:

```
Private Sub Form_MouseUp(Button As Integer, _
    Shift As Integer, X As Single, Y As Single)

End Sub
```

Wir wollen diese allerdings im Klassenmodul des Hauptformulars implementieren. Dazu deklarieren wir die Variable **sfm** wie folgt:

```
Dim WithEvents m_SubForm As Form
```

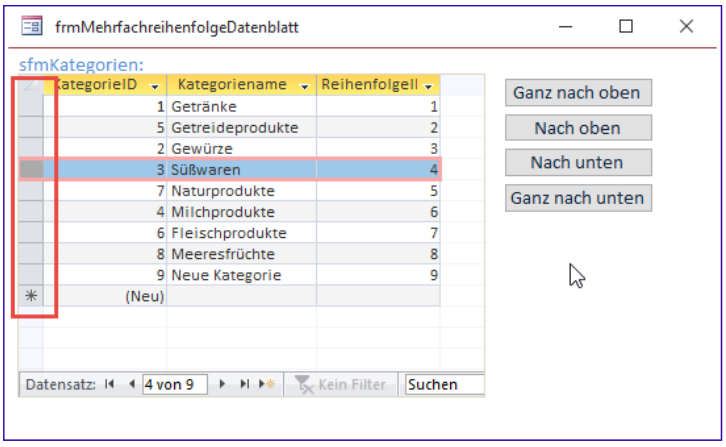


Bild 2: Bereich, der das Ereignis **OnMouseDown** auslöst

Bei dieser Gelegenheit deklarieren wir direkt drei Konstanten, die später oft verwendete Bezeichnungen enthalten – für die Tabelle, das Primärschlüsselfeld der Tabelle und das Feld, nach dem die Sortierung erfolgt, in diesem Fall **ReihenfolgeID**:

```
Const m_Table As String = "tblKategorien"
Const m_PKField As String = "KategorieID"
Const m_Orderfield As String = "Reihen-
folgeID"
```

Auf diese Weise können Sie, wenn Sie den Code dieser Lösung in anderen Formularen einsetzen wollen, schnell die Bezeichnungen der von Ihnen verwendeten Tabelle und Felder angeben.

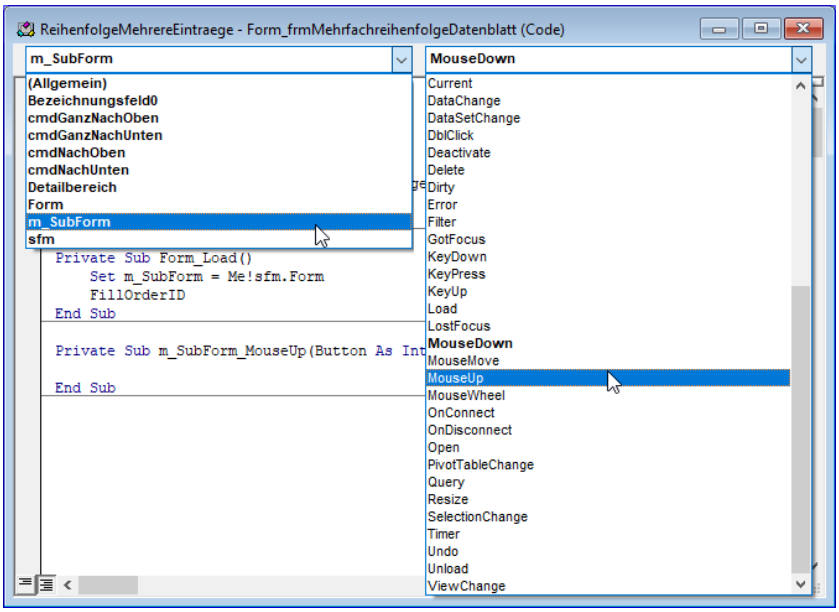


Bild 3: Anlegen des Ereignisses für das Unterformular

Danach füllen wir diese Variable in der Prozedur, die durch das Ereignis **Beim Laden** des Hauptformulars ausgelöst wird, wie folgt:

```
Private Sub Form_Load()
    Set m_SubForm = Me!sfm.Form
    m_SubForm.OnMouseUp = "[Event Procedure]"
    FillOrderID
End Sub
```

Die zweite Anweisung stellt per Code die Eigenschaft **Bei Maustaste auf** des Unterformulars auf den Wert **[Ereignisprozedur]** ein – diesen Schritt würden Sie normalerweise über das Eigenschaftenblatt erledigen.

Die Prozedur **FillOrderID** füllt das in **m_Orderfield** angegebene Feld der Tabelle aus **m_Table** mit durchlaufenden Werten von **1** bis zur Anzahl der enthaltenen Datensätze.

Diese Prozedur beschreiben wir ausführlich im Beitrag **Listenfeld: Reihenfolge mehrerer Einträge ändern** (**www.access-im-unternehmen.de/1197**).

Nun hinterlegen wir eine Ereignisprozedur für das Ereignis **Bei Maustaste ab** des Unterformulars.

Dazu wählen Sie im Codefenster des Moduls **Form_frmMehrfachreihenfolgeDatenblatt** aus dem linken Kombinationsfeld den Wert **m_SubForm** und aus dem rechten Kombinationsfeld den Wert **MouseUp** aus (siehe Bild 3).

Damit das Ereignis auch ausgelöst wird, müssen Sie für das Unterformular noch ein Klassenmodul anlegen. Dazu können Sie einfach die Eigenschaft **Enthält Modul** des als Unterformular verwendeten Formulars auf **Ja** einstellen.

Markierte Datensätze erkennen

Die so entstandene Prozedur, die wie folgt aussieht, müssen wir nun noch mit dem Code füllen, der die Markierung prüft und die Schaltflächen aktiviert oder deaktiviert. Diesen Code lagern wir in eine neue Prozedur namens **ActivateControls** aus:

```
Private Sub m_SubForm_MouseDown(Button As Integer, _
        Shift As Integer, X As Single, Y As Single)
    ActivateControls
End Sub
```

Hier geben wir testweise erst einmal die Werte der Eigenschaften **SelTop** und **SelHeight** im Direktbereich aus, um zu prüfen, ob diese die entsprechenden Werte zur durchgeführten Markierung liefern:

```
Private Sub ActivateControls()
    Debug.Print m_SubForm.SelTop, m_SubForm.Sel-
Height
End Sub
```

Das gelingt auch recht gut. Die Eigenschaft **SelTop** liefert immer den 1-basierten Index des ersten markierten Eintrags, die Eigenschaft **SelHeight** die Anzahl der markierten Einträge. Wenn Sie etwa erst auf den Datensatzmarkierer des zweiten Eintrags klicken und dann bei gedrückter **Umschalt**-Taste auf den fünften Eintrag, werden vier Einträge vom zweiten bis zum fünften Eintrag markiert (siehe Bild 4).

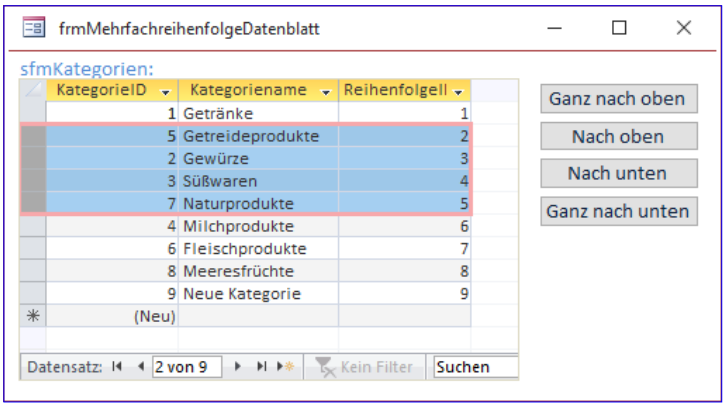


Bild 4: Markieren mehrerer Einträge

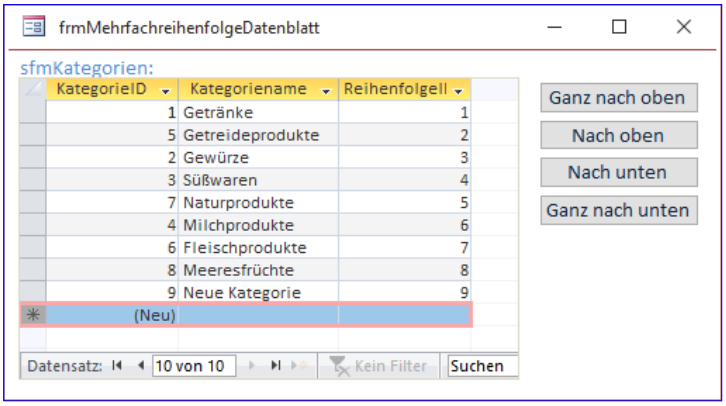


Bild 5: Markieren des letzten Eintrags

Die Eigenschaft **SelTop** liefert dann den Wert **2**, die Eigenschaft **SelHeight** den Wert **4**. Sie können auch mehrere Datensätze markieren, indem Sie auf den Datensatzmarkierer des ersten zu markierenden Datensatzes klicken und dann bei gedrückter linker Maustaste den Mauszeiger bis zum Datensatzmarkierer des letzten gewünschten Datensatzes herunterziehen.

Hier gibt es noch ein paar Besonderheiten:

- Wenn das Unterformular das Hinzufügen von Datensätzen erlaubt, können Sie auch die neue, leere Zeile markieren (siehe Bild 5). Dies liefert dann als Ergebnis für die Eigenschaft **SelTop** den Wert **9** und für die Eigenschaft **SelLength** den Wert **1**. Wie wir dies behandeln, schauen wir uns weiter unten an.

- Sie können auch mit der Maus auf den schmalen Bereich zwischen zwei Datensatzmarkierern klicken. Das ist dann der Fall, wenn sich der Mauszeiger zuvor in das Symbol aus Bild 6 verwandelt hat. Hier liefert **SelTop** den Index des darunter liegenden Datensatzes, im Beispiel den Wert **6**. **SelLength** liefert aber zum Glück den Wert **0**, sodass wir diesen Sonderfall gut behandeln können.

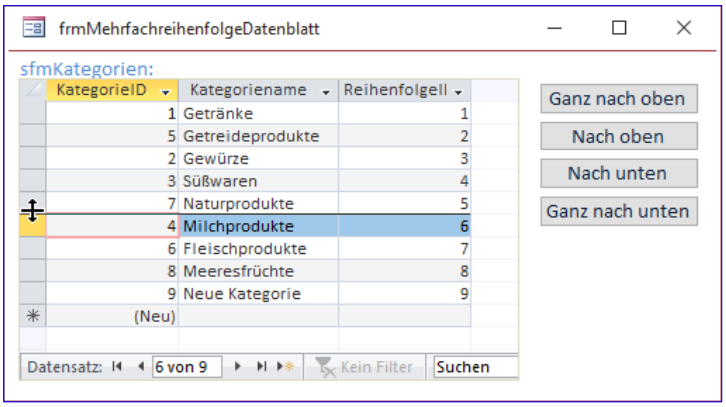


Bild 6: Anklicken des Bereichs zwischen zwei Datensätzen

Button vom Formular ins Ribbon

Formulare sind oft völlig überfrachtet mit Steuerelementen zur Anzeige und Auswahl von Daten und Schaltflächen. Das lässt sich zumindest teilweise ändern, indem Sie einige Elemente aus dem Formular in ein Ribbon auslagern, das dann beim Erscheinen des Formulars eingeblendet wird. Bei einfachen Schaltflächen ist das noch recht einfach, aber sobald die Steuerelemente etwa abhängig von den angezeigten Daten ein- oder ausgeblendet werden sollen, wird es interessant. In diesem Beitrag schauen wir uns an, wie das gelingt.

Konkret nehmen wir uns das Beispiel aus dem Beitrag **Datenblatt: Reihenfolge mehrerer Einträge ändern** (**www.access-im-unternehmen.de/1198**) vor. Hier finden wir in einem Formular ein Unterformular in der Datenblattansicht vor (siehe Bild 1). Dieses enthält vier Schaltflächen, mit denen Sie die Reihenfolge der Einträge des Datenblatts über die Werte des Feldes **ReihenfolgeID** manipulieren können. Im Falle dieses Beispiels enthält das Datenblatt nicht so viele Felder, dass diese nicht auf das Formular passen würde.

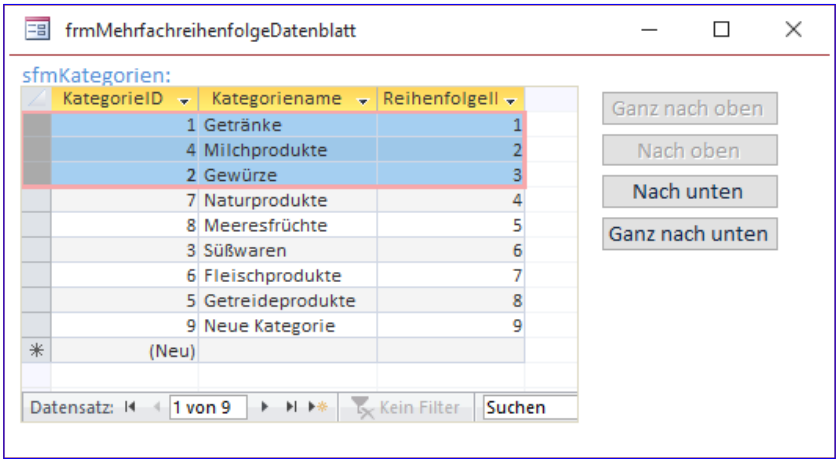


Bild 1: Ausgangsformular

Wenn dies allerdings der Fall wäre, würde man sich die Schaltflächen an einer anderen Position wünschen. Am Besten, damit diese gar nicht mehr im Weg sind, direkt im Ribbon. Und genau das ist Thema des vorliegenden Artikels: Wir wollen zeigen, wie Sie die Schaltflächen aus einem Formular in das Ribbon übertragen und dabei alle vorgegebenen Funktionen beibehalten.

In unserem Fall betrifft das vor allem das Aktivieren und Deaktivieren der Schaltflächen in Abhängigkeit der angezeigten Daten. Das heißt, dass die Schaltflächen zum Verschieben der Elemente nach oben nur aktiviert sein sollen, wenn die Elemente auch nach oben verschoben werden können. Das gleiche gilt für die Schaltflächen zum Verschieben nach unten – diese sollen nur aktiviert werden, wenn die Elemente auch nach unten verschoben werden können. Außerdem sollen die Schaltflächen natürlich die im Klassenmodul des Formulars gespeicherten Ereignisprozeduren aufrufen.

Eine weitere Anforderung ist, dass das Ribbon nur in Zusammenhang mit der Anzeige dieses Formulars angezeigt werden soll. Um dieses zu öffnen, fügen wir der Anwendung noch ein Start-Ribbon hinzu, das beim Start angezeigt wird und das Öffnen des Formulars **frmMehrfachreihenfolgeDatenblatt** erlaubt. Wir könnten theoretisch auch direkt das Unterformular in der Datenblattansicht öffnen, aber das würde bedeuten, dass wir den kompletten Code des Hauptformulars in das Klassenmodul des Unterformulars übertragen und anpassen müssten. Diesen Schritt wollen wir erst einmal nicht gehen. Daher maximieren wir einfach das Unterfor-

mular innerhalb des Hauptformulars und sorgen dafür, dass das Hauptformular mit dem Unterformular beim Anklicken des entsprechenden Ribbon-Eintrags angezeigt wird. Das Ribbon-Tab für das Formular **frmMehrfachreihenfolgeDatenblatt** soll dann rechts neben dem Startribbon erscheinen.

Start-Ribbon erstellen

Um das erste Ribbon zu erstellen, fügen wir zunächst eine neue Tabelle namens **USysRibbons** zur Datenbank hinzu. Dieser Tabelle fügen Sie die drei folgenden Felder hinzu (siehe Bild 2):

- **ID**: Autowertfeld mit Primärschlüsselindex
- **RibbonName**: Textfeld/Kurzer Text
- **RibbonXML**: Memofeld/Langer Text

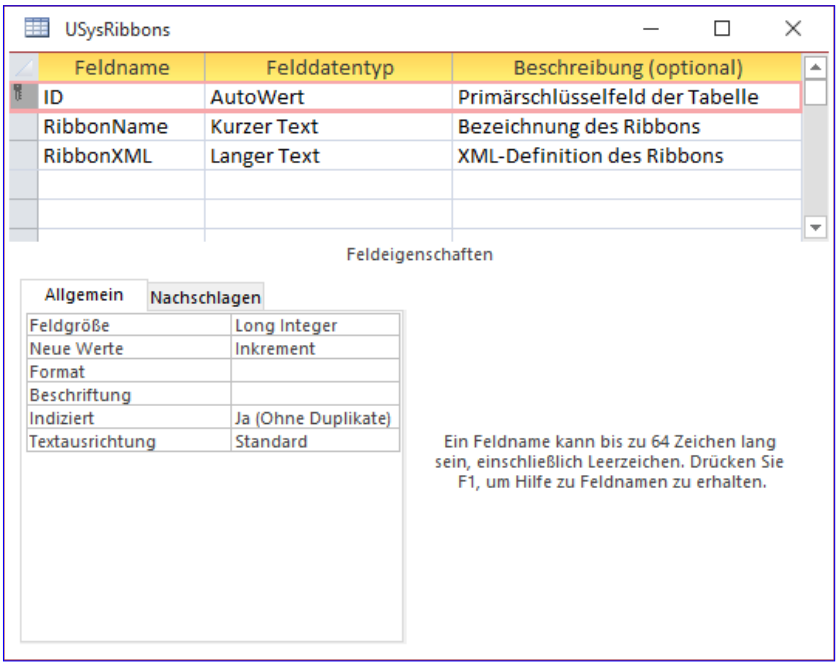


Bild 2: Entwurf der Tabelle **USysRibbons**

USysRibbons wieder einblenden

Wenn Sie nicht die Anzeige von Systemobjekten und ausblendeten Objekten deaktiviert haben, verschwindet die Tabelle allerdings direkt nach dem Speichern aus dem Navigationsbereich.

Um diese wieder sichtbar zu machen, klicken Sie mit der rechten Maustaste auf die Titelzeile des Navigationsbereichs und wählen aus dem Kontextmenü den Eintrag **Navigationsoptionen...** aus.

Im nun erscheinenden Dialog **Navigationsoptionen** aktivieren Sie die beiden Optionen **Ausgeblendete Objekte anzeigen** und **System-Objekte anzeigen** (siehe Bild 3). Danach erscheint die Tabelle

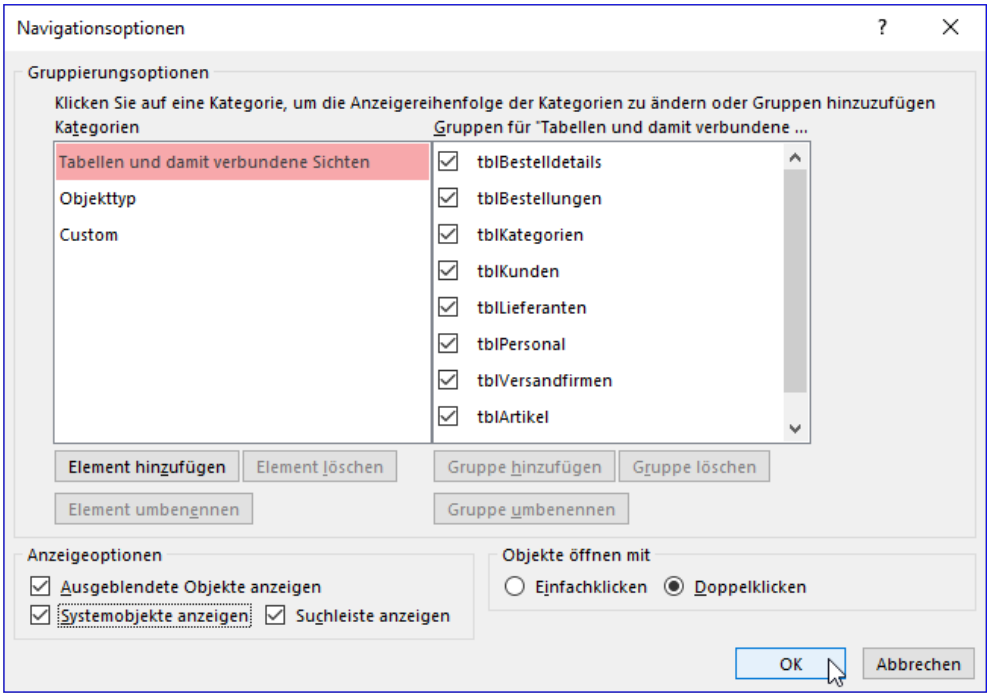


Bild 3: Die Navigationsoptionen von Access

USysRibbons wieder im Navigationsbereich.

Ribbon-Definition für das Start-Ribbon

Danach fügen wir dem Feld **RibbonName** der Tabelle den Wert **Main** und dem Feld **RibbonXML** den XML-Code aus Listing 1 hinzu.

```
<?xml version="1.0"?>
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui">
  <ribbon>
    <tabs>
      <tab id="tabBeispiele" label="Beispiele">
        <group id="grpFormulare" label="Formulare">
          <button imageMso="AccessFormDatasheet" label="Mehrfachreihenfolge
            Datenblatt" id="btnMehrfachreihenfolgeDatenblatt" onAction="onAction"
            size="large"/>
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

Listing 1: XML-Code für das Start-Ribbon

Hier definieren wir die ineinander verschachtelten Elemente **customUI**, **ribbon**, **tabs**, **tab**, **group** und schließlich **button**. Das **button**-Element erhält schließlich über das Attribut **imageMso** den Namen eines eingebauten Icons, eine Beschriftung, eine ID, den Wert **large** für das Attribut **size** und den Wert **onAction** für das Attribut **onAction**. Das bedeutet, dass wir an irgendeiner Stelle – in diesem Fall in einem Standardmodul – eine Prozedur namens **onAction** bereitstellen müssen, die bestimmten syntaktischen Anforderungen genügt. Die Tabelle sieht danach wie in Bild 4 aus.

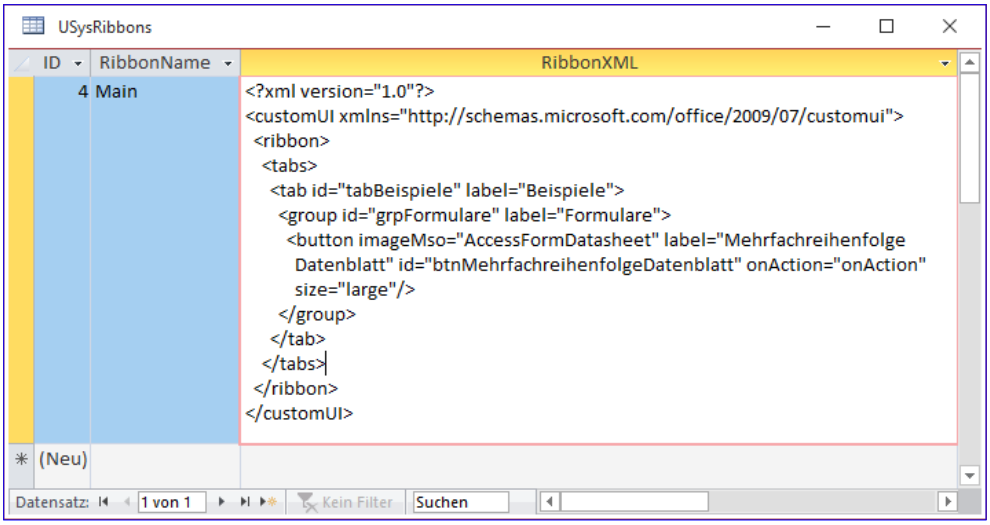


Bild 4: Ribbon-Definition für das Start-Ribbon

Code für die Schaltfläche zum Öffnen des Formulars

Die oben genannte Prozedur **onAction** erstellen wir schließlich so:

```
Sub onAction(control As IRibbonControl)
    DoCmd.OpenForm "frmMehrfachreihenfolgeDatenblatt"
End Sub
```

Die Prozedur öffnet also dann das gewünschte Formular.

Ribbonfehler anzeigen

Damit die Benutzeroberfläche Fehler beim Verwenden der benutzerdefinierten Ribbons anzeigt, müssen wir noch eine Access-Option aktivieren. Dazu öffnen Sie über den Eintrag **Datei|Optionen** zunächst den Optionen-Dialog von Access. Hier finden Sie im Bereich **Clienteinstellungen|Allgemein** die Option **Fehler von Benutzeroberflächen-Add-Ins anzeigen**. Diese aktivieren Sie und schließen den Dialog mit der Schaltfläche **OK** (siehe Bild 5).

Abschließende Schritte

Nun müssen wir nur noch drei Schritte erledigen:

- Die Anwendung schließen und wieder öffnen.
- Die Option **Name des Menübands** auf die durch das Schließen und Öffnen nun verfügbaren Eintrag **Main** einstellen (siehe Bild 6).
- Die Anwendung erneut schließen und wieder öffnen, um das Ribbon nun anzuzeigen.

Nun erscheint das Ribbon wie in Bild 7. Hier klicken Sie auf den Tab namens **Beispiele** und erhalten die Schaltfläche zum Öffnen des Formulars.

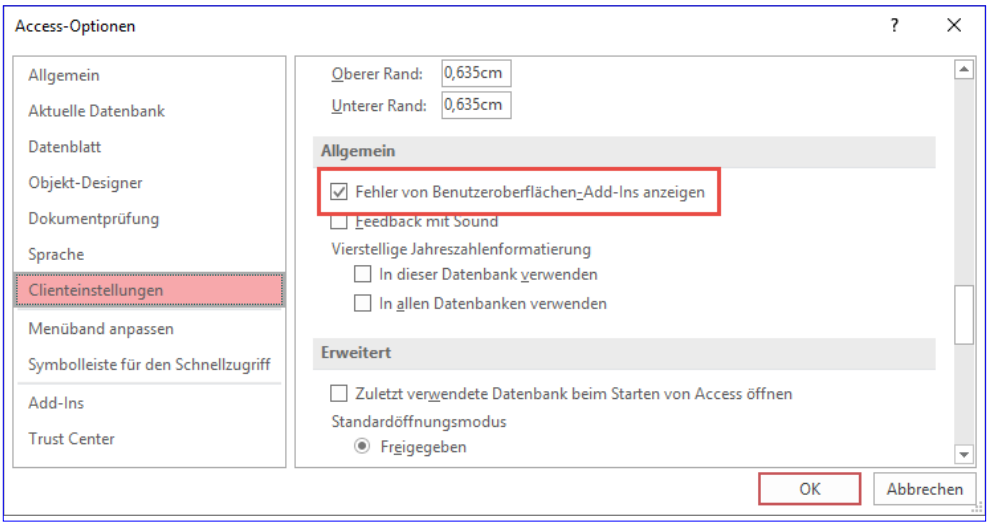


Bild 5: Ribbon-Option zum Anzeigen von Ribbon-Fehlern

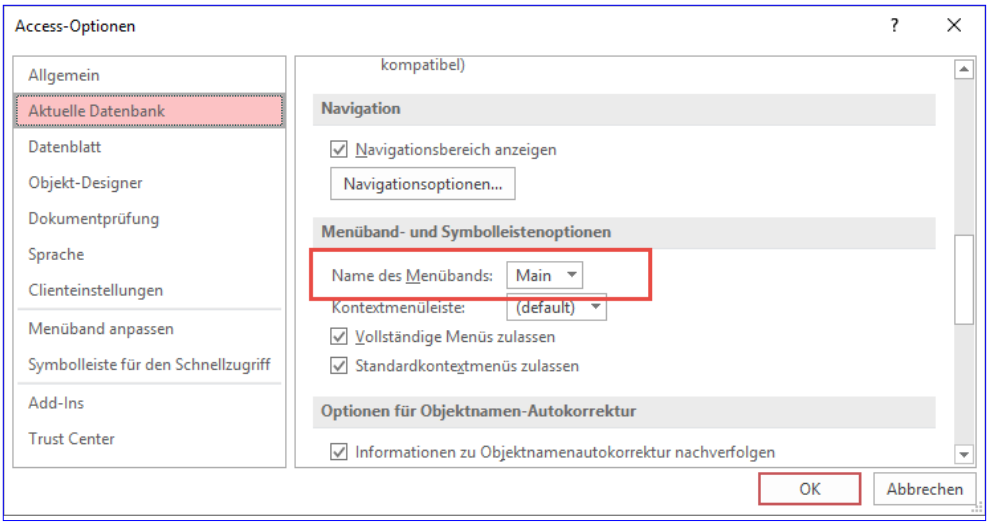


Bild 6: Start-Ribbon für die Anwendung einstellen

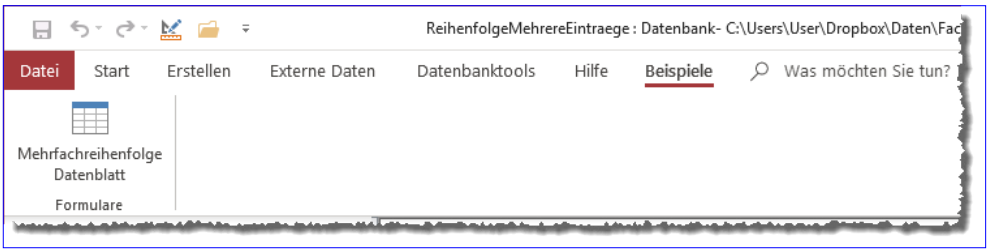


Bild 7: Das Start-Ribbon

Schaltflächen ins Ribbon

Um die Funktionen der vier Schaltflächen nun über das Ribbon ausführen zu können, legen wir zunächst ein neues Ribbon mit einem eigenen Tab für das Formular **frmMehrfachreihenfolgeDatenblatt** an.

Der XML-Code für dieses Ribbon sieht zunächst wie in Listing 2 aus und bildet das Grundgerüst, das wir gleich noch ausbauen. Erstmal möchten wir überhaupt ein Ribbon mit dem Formular einblenden. Diesen XML-Code fügen Sie in einem zweiten Datensatz in die Tabelle **USysRibbons** ein, und zwar mit dem Eintrag **Reihenfolge** im Feld **RibbonName**.

```
<?xml version="1.0"?>
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui">
  <ribbon>
    <tabs>
      <tab id="tabReihenfolge" label="Reihenfolge">
        <group id="grpReihenfolgeAendern" label="Reihenfolge ändern">
          <button imageMso="ShapeUpArrow" label="Ganz nach oben" id="btnTop" size="large"/>
          <button id="btnUp" imageMso="ShapeUpArrow" label="Nach oben" size="large"/>
          <button id="btnDown" imageMso="ShapeDownArrow" label="Nach unten" size="large"/>
          <button id="btnBottom" imageMso="ShapeDownArrow" label="Ganz nach unten" size="large"/>
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

Listing 2: XML-Code für das Ribbon zum Formular

Damit dies gelingt, stellen wir die Eigenschaft **Name des Menübands** des Hauptformulars auf den Namen des neu erstellten Ribbons ein – allerdings erst, nachdem wir nach dem Einfügen des neuen Eintrags zur Tabelle **USysRibbons** die Anwendung geschlossen und wieder geöffnet haben (siehe Bild 8). Nur dadurch werden neu hinzugefügte Ribbon-Namen auch in den Auswahlfeldern der jeweiligen Eigenschaften sichtbar.

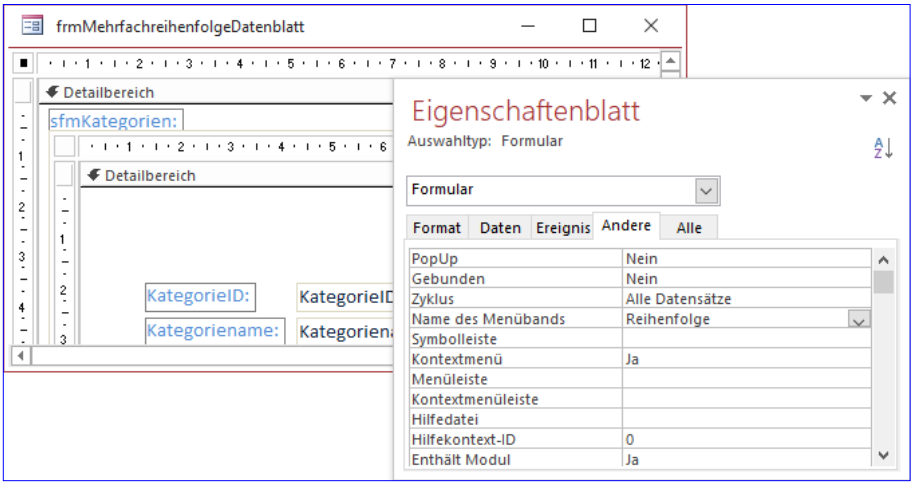


Bild 8: Menüband festlegen

Ribbon für das Unterformular

Wenn wir das Formular nun über den Ribbon-Befehl des Start-Ribbons öffnen, erscheint zwar das Formular, aber das **Tab**-Element aus dem zugeordneten Ribbon wird nicht angezeigt (siehe Bild 9). Warum das? Ganz einfach: Die Schaltflächen sind zu Beginn deaktiviert, damit bleibt das Unterformular das einzige Steuerelement, das den Fo-

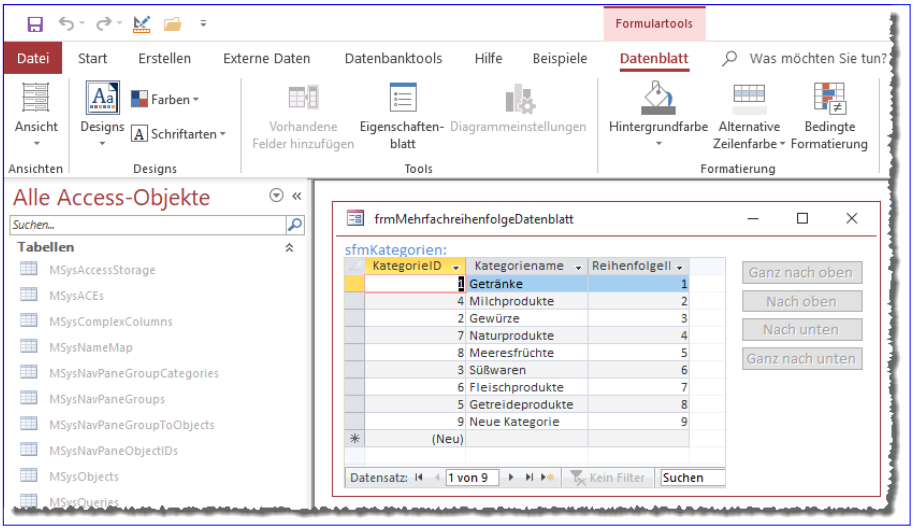


Bild 9: Kein neues Ribbon für das Formular?

kus erhalten kann. Und das Unterformular hat genau wie das Formular eine eigene Eigenschaft namens **Name des Menübands**. Wir haben nun zwei Möglichkeiten:

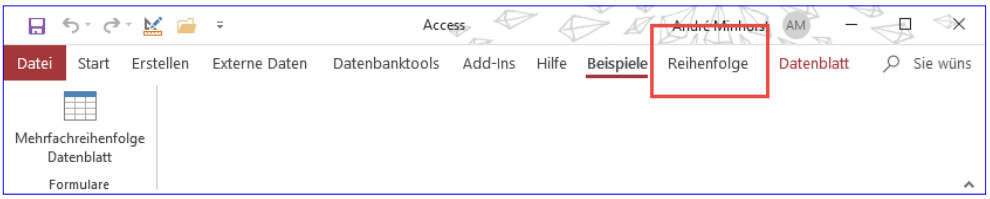


Bild 10: Das Reihenfolge-Ribbon wird nicht aktiviert.

- Wir stellen diese Eigenschaft einfach auf das gleiche Ribbon ein wie die für das Hauptformular.
- Wir fügen eine Ereignisprozedur für das Hauptformular hinzu, dass dem Unterformular das Ribbon des Hauptformulars zuweist.

Wir entscheiden uns für die zweite Variante, da wir dann gegebenenfalls nur das Ribbon für das Hauptformular ändern müssen:

```
Private Sub Form_Load()
    Me!sfm.Form.RibbonName = Me.RibbonName
End Sub
```

Damit erhalten wir nun beim Öffnen des Formulars **frmMehrfachreihenfolgeDatenblatt** zwar das neue Ribbon-Tab, allerdings wird dieses nicht aktiviert (siehe Bild 10).

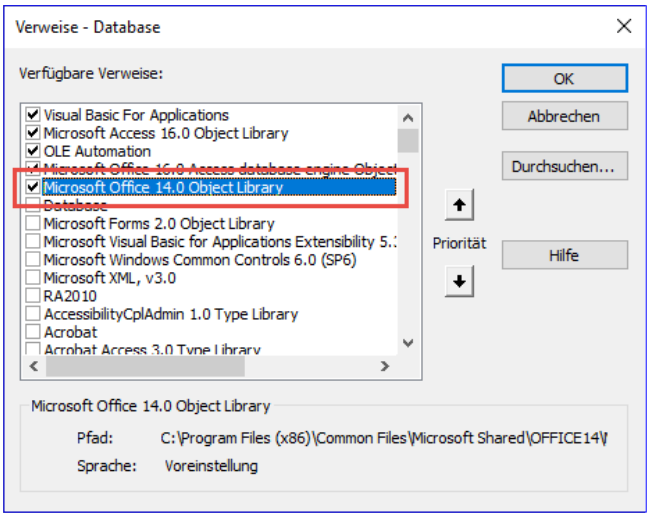


Bild 11: Verweis auf die Office-Bibliothek

Dazu gibt es wiederum drei Möglichkeiten:

- Wir aktivieren das Ribbon-Tab per VBA-Code.
- Wir definieren das Ribbon-Tab als kontextsensitives Ribbon-Tab.
- Wir stellen die Eigenschaft **startFromScratch** des **ribbon**-Elements auf **true** ein. Damit werden alle übrigen Elemente ausgeblendet.

Wir schauen uns alle drei Varianten an.

Ribbon-Tab per VBA-Code aktivieren

Für diesen Fall benötigen wir zunächst einen Verweis auf die Bibliothek **Microsoft Office 14.0 Object Library**, wobei wir diesen über den **Verweise**-Dialog hinzufügen.

Diesen öffnen Sie über den VBA-Editor mit dem Menübefehl **Extras|Verweise**. Dort fügen Sie dann den genannten Eintrag hinzu (siehe Bild 11).

Nun fügen wir dem Ribbon-Element **customUI** das Attribut **onLoad** hinzu:

```
<customUI
    xmlns="http://schemas.microsoft.com/office/2009/07/customui"
onLoad="onLoad_Reihenfolge">
```

Die hier angegebene Ereignisprozedur **onLoad_Reihenfolge** hinterlegen wir wieder im Modul **mdlRibbons**. Die Prozedur weist lediglich der Variablen **objRibbon_Reihenfolge** einen Verweis auf das per Parameter mitgelieferte Objekt **IRibbonUI** zu:

```
Sub onLoad_Reihenfolge(ribbon As IRibbonUI)
    Set objRibbon_Reihenfolge = ribbon
End Sub
```

Diese Variable müssen wir nun noch deklarieren, was wir im gleichen Modul erledigen:

```
Public objRibbon_Reihenfolge As IRibbonUI
```

Nun bietet dieses Objekt verschiedene Methoden an, darunter auch die **ActivateTab**-Methode, die den Namen des zu aktivierenden **Tab**-Elements entgegennimmt. Wann und wie aber wollen wir diese Methode aufrufen? Eigentlich sollte dies gleich beim Öffnen des Formulars geschehen, also etwa in der Ereignisprozedur für eine der Ereigniseigenschaften **Beim Öffnen** oder **Beim Laden**.

Wenn wir aber etwa für die Ereignisprozedur **Form_Load** die folgende Anweisung anlegen, gelingt das nicht:

```
Private Sub Form_Load()
    objRibbon_Reihenfolge.ActivateTab "tabReihenfolge"
End Sub
```

Wir erhalten dann nämlich die Fehlermeldung aus Bild 12, der dadurch ausgelöst wird, dass die Objektvariable noch nicht mit dem Verweis auf das **customUI**-Objekte gefüllt wurde. Das versuchen wir zu umgehen, indem wir das Aktivieren des **Tab**-Elements ein wenig verzögern. Dazu stellen wir die Eigenschaft **Zeitgeberintervall** des Hauptformulars auf den Wert **500** ein (für 500 Millisekunden) und für die Ereigniseigenschaft **Bei Zeitgeber** hinterlegen wir die folgende Ereignisprozedur:

```
Private Sub Form_Timer()
    objRibbon_Reihenfolge.ActivateTab "tabReihenfolge"
    Me.TimerInterval = 0
End Sub
```

Allerdings erhalten wir hier den gleichen Fehler. Woran liegt das? Wir könnten annehmen, dass die Prozedur

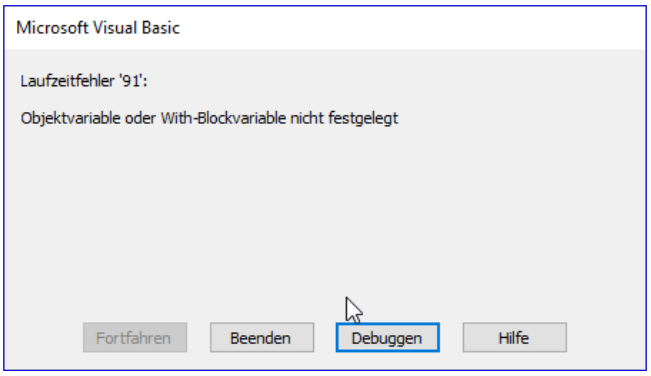


Bild 12: Fehler beim Versuch, die **ActiveTab**-Methode zu nutzen

onLoad_Reihenfolge erst ausgeführt wird, wenn das Ribbon-Tab aus dieser Ribbon-Definition angezeigt wird. Allerdings scheint auch dies nicht der Fall zu sein. Wir können ja zwischendurch immer prüfen, ob **objRibbon_Reihenfolge** einen Wert hat – und zwar mit der folgenden VBA-Anweisung, abgesetzt im Direktbereich des VBA-Editors:

```
? objRibbon_Reihenfolge Is Nothing
```

Dies liefert aber aktuell immer den Wert **True**, **objRibbon_Reihenfolge** ist also leer. Die Lösung für dieses Problem war: Wir haben das Formular mehrfach aufgerufen und bereits zuvor dafür gesorgt, dass **On_Load** aufgerufen wurde. Damit war **objRibbon_Reihenfolge** gefüllt. Allerdings haben wir anschließend vermutlich durch einen Laufzeitfehler dafür gesorgt, dass die Variable wieder geleert wurde. Da **On_Load** aber bereits ausgeführt war, wurde sie nicht erneut gefüllt ... Es gilt also zu beachten: Entweder Sie verhindern Laufzeitfehler durch entsprechend stabile Programmierung mit Fehlerbehandlungen, oder Sie müssen zwischendurch die Anwendung einmal neu laden, damit alle Ribbons et cetera wieder korrekt geladen werden können.

Nach einem Neustart der Anwendung erhalten wir jedoch das gewünschte Ribbon (siehe Bild 13).

Noch eine Optimierung ist es, die Eigenschaft **Zeitgeberintervall** gar nicht erst über das Eigenschaftenblatt

Autocomplete in Textfeldern

Autocomplete kennen Sie vermutlich von Kombinationsfeldern. Hier können Sie einen oder mehrere Buchstaben eingeben und das Kombinationsfeld zeigt gleich den nächsten Eintrag der Datensatzherkunft an, der mit diesen Anfangsbuchstaben beginnt. Dieses Verhalten wollen wir auch gern für Textfelder programmieren. Dazu starten wir mit einer etwas einfacheren Variante, die Sie vielleicht vom VBA-Editor kennen: Wenn Sie dort beginnen, einen Objekt- oder Variablennamen zu schreiben, können Sie mit der Tastenkombination Strg + Leertaste dafür sorgen, dass IntelliSense aktiviert wird und passende Einträge anzeigt. Wir wollen dafür sorgen, dass an dieser Stelle einfach der erste passende Eintrag erscheint – wenn wir eine Liste der verfügbaren Einträge wollten, könnten wir ja direkt ein Kombinationsfeld verwenden.

Autocomplete per Tastenkombination

Im VBA-Editor verwendet man die Tastenkombination **Strg + Leertaste**, um beispielsweise Klassen oder Variablen anzuzeigen, die mit den bisher eingetippten Buchstaben beginnen. Wenn Sie etwa im Direktfenster die Buchstaben **Curr** eingeben und dann diese Tastenkombination verwenden, erscheint die Liste aus Bild 1 zur Auswahl.

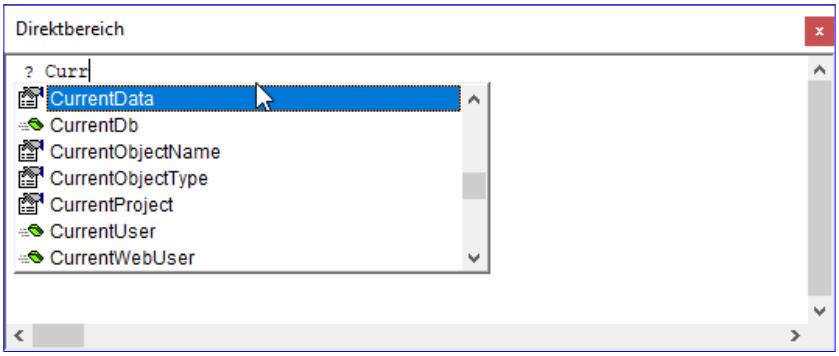


Bild 1: Autocomplete per Intellisense im VBA-Editor

Was für ein Verhalten wollen wir bei dieser ersten Technik erhalten und welche Voraussetzungen müssen dafür erfüllt sein? Die Voraussetzung ist, dass das Textfeld an ein Feld einer Tabelle gebunden ist, aus dem wir die möglichen zu ergänzenden Texte gewinnen können.

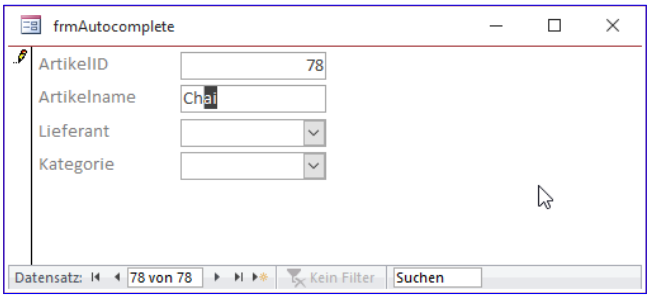


Bild 2: Autocomplete im Textfeld

Dann wollen wir dem Benutzer ermöglichen, einen oder mehrere Buchstaben einzugeben und dann mit der Tastenkombination **Strg + Leertaste** den nächsten passenden Eintrag zu ergänzen – so, dass die ergänzten Buchstaben markiert sind. Wenn wir also etwa in ein Textfeld, das an das Feld **Artikelname** der Tabelle **tblArtikel** gebunden ist, die Zeichenkette **Ch** eingeben und dann **Strg + Leertaste** betätigen, soll die Zeichenkette zu **Chai** ergänzt werden, wobei die ergänzten Buchstaben markiert sein sollen. Die Einfügemarke soll sich dabei hinter dem zuletzt eingegebenen Buchstaben befinden und vor dem ersten Buchstaben der Markierung (siehe Bild 2).

Das weitere Verhalten hängt davon ab, was der Benutzer als nächstes macht. Hier ist die Auswahl der möglichen Schritte:

- Verlassen des Textfeldes: aktueller Text inklusive markiertem, vorgeschlagenem Text wird behalten
- Betätigen der **Nach links**- oder **Nach rechts**-Taste beziehungsweise der Tasten **Pos1** oder **Ende**: aktueller Text wird behalten, Markierung wird aufgehoben
- Betätigen der **Zurück**-Taste: Markierter Text wird gelöscht.
- Eingabe eines anderen Zeichens als des ersten Zeichens der Markierung: Suche nach einem passenden Eintrag und Anzeige der ergänzenden Zeichen mit Markierung
- Eingabe des nächsten Zeichens der Markierung: Zeichen wird übernommen, die restlichen Zeichen bleiben markiert

Tastenkombination abfangen

Die erste Aufgabe ist, die Tastenkombination **Strg + Leertaste** abzufangen. Wir schauen in einem Kombinationsfeld, wann die Ergänzung stattfindet – beim Niederdrücken der Taste oder beim Loslassen der Taste. Das Ergebnis ist: Die Ergänzung erfolgt bereits beim Herunterdrücken der Taste. Also verwenden wir das Ereignis **Bei Taste ab** des Textfeldes, das wir in unserem Beispiel **txtArtikelname** genannt haben.

Dazu wählen wir im Eigenschaftenblatt für das Ereignis **Bei Taste ab** den Eintrag **[Ereignisprozedur]** aus und klicken dann auf die Schaltfläche mit den drei Punkten (**...**) neben dem Eigenschaftsfeld.

Dies legt im Klassenmodul des Formulars die folgende leere Prozedur an:

```
Private Sub txtArtikelname_KeyDown(KeyCode As Integer, _
                                   Shift As Integer)

End Sub
```

Um zu ermitteln, welche Werte die Parameter **KeyCode** und **Shift** beim Betätigen der Tastenkombination **Strg + Leertaste** liefern, geben wir diese innerhalb der Prozedur mit der folgenden Anweisung im Direktfenster aus:

```
Debug.Print KeyCode, Shift
```

Danach wechseln wir in die Formularansicht, setzen den Fokus in das Textfeld **txtArtikel** und geben die Tastenkombination **Strg + Leertaste** ein. Das Ergebnis ist, dass **KeyCode** den Wert **32** und **Shift** den Wert **2** liefert (**32** entspricht auch der Konstanten **vbKeySpace**, **2** der Konstanten **acCtrlMask**).

Damit können wir schon eine Bedingung in das Ereignis einbauen, unter der wir aktiv werden wollen:

```
Private Sub txtArtikelname_KeyDown(KeyCode As Integer, _
                                   Shift As Integer)
    If KeyCode = vbKeySpace And Shift = acCtrlMask Then
        MsgBox "Action!"
    End If
End Sub
```

Danach gehen wir an die Umsetzung der ersten Aufgabe. Das Ergebnis sieht wie in Listing 1 aus. Die Prozedur verwendet drei Variablen. **intPosStart** speichert die Position der Eingabemarke zwischen und **strTreffer** eventuell gefundene Treffer. **strText** speichert den beim Betätigen der Taste im Textfeld vorhandenen Text.

Nach dem Prüfen der Tastenkombination speichert die Prozedur die aktuelle Position der Einfügemarke in der Variablen **intSelStart** und den aktuellen Text in **strText**. Dann sucht sie mit der **DLookup**-Funktion in der zugrunde liegenden Tabelle nach einem Wert, der mit den Zeichen aus **strText** beginnt.

Findet sie einen solchen Wert, wird dieser in **strTreffer** geschrieben, sonst landet der zweite Parameter der **Nz**-

```
Private Sub txtArtikelname_KeyDown(KeyCode As Integer, Shift As Integer)
    Dim intSelStart As Integer
    Dim strTreffer As String
    Dim strText As String
    If KeyCode = vbKeySpace And Shift = acCtrlMask Then
        intSelStart = Me!txtArtikelname.SelStart
        strText = Me!txtArtikelname.Text
        strTreffer = Nz(DLookup("Artikelname", "tblArtikel", "Artikelname LIKE '" & strText & "*'"), strText)
        Me!txtArtikelname.Text = strTreffer
        Me!txtArtikelname.SelStart = intSelStart
        Me!txtArtikelname.SelLength = 255
        KeyCode = 0
    End If
End Sub
```

Listing 1: Automatisches Ergänzen bei Strg + Leertaste

Funktion als Ergebnis in **strTreffer** – in diesem Fall der vorherige Inhalt des Textfeldes aus **strText**.

Danach folgt die Aktualisierung des Textfeldes. Dabei weisen wir der Eigenschaft **Text** den Inhalt von **strTreffer** zu und setzen die Eingabemarke über die Eigenschaft **SelStart** auf die vorherige Position, also **intSelStart**. Außerdem stellen wir die Eigenschaft **SelLength** auf den Wert **255** ein, was der größten Zeichenkette entspricht, die wir in dieses Textfeld eingeben können. Damit markieren wir schlicht den Text von der Position der Einfügemarke aus bis zum Schluss.

Danach folgt noch eine wichtige Anweisung (**KeyCode = 0**). Diese sorgt dafür, dass die eingegebene Taste nicht an das Textfeld geschickt wird. Ohne diese Anweisung würde die Prozedur zwar wie gewünscht funktionieren, aber durch das anschließende Senden des Leerzeichens an das Textfeld würde der markierte Bereich wieder gelöscht und durch das Leerzeichen überschrieben werden.

Damit haben wir aber nur den Beginn gemacht. Nun kommen wir an den Punkt, wo der Benutzer das nächste Zeichen eingibt, eines löscht, die Einfügemarke mit den Tasten **Nach links**, **Nach rechts**, **Pos1** oder **Ende** bewegt oder das Feld verlässt.

Einschränkung: Einfügemarke nicht hinter dem letzten Zeichen

An dieser Stelle wollen wir uns bereits um eine kleine Einschränkung kümmern: Der zuvor beschriebene Ablauf soll nicht durchgeführt werden, wenn der Benutzer die Tastenkombination **Strg + Leerzeichen** betätigt, während die Einfügemarke sich nicht am Ende des bisher eingegebenen Textes befindet. Wir müssen also noch eine Prüfung der Position der Einfügemarke bezogen auf den vorhandenen Text hinzufügen. Diese fügen wir hinter den beiden Anweisungen zur Ermittlung der Startposition der Markierung und des Textes ein und prüfen, ob **intSelStart** genau der Länge des eingegebenen Textes entspricht – was bedeutet, dass die Einfügemarke sich hinter dem letzten Zeichen befindet:

```
---
strText = Me!txtArtikelname.Text
If intSelStart = Len(strText) Then
    ...
End If
KeyCode = 0
...
```

Verlassen des Feldes mit Autocomplete

Wenn der Benutzer mit **Strg + Leertaste** Autocomplete aktiviert und gegebenenfalls eine Ergänzung hinzugefügt

hat und dann das Feld beispielsweise mit der Tabulator-Taste oder durch einen Mausklick auf ein anderes Steuerelement verlässt, soll der enthaltene Text inklusive des markierten Teils beibehalten werden.

Für dieses Verhalten brauchen wir gar nichts zu tun – es ist bereits implementiert.

Bewegen der Einfügemarke

Die Einfügemarke kann der Benutzer bei markiertem Text auf verschiedene Arten bewegen – zum Beispiel auf die folgenden:

- **Nach links**-Taste: Bewegt die Einfügemarke um eine Position nach links.
- **Nach rechts**-Taste: Bewegt die Einfügemarke um eine Position nach rechts.
- **Pos1**-Taste: Bewegt die Einfügemarke an die erste Position.
- **Ende**-Taste: Bewegt die Einfügemarke an die letzte Position.

Wichtig ist an dieser Stelle, dass diese Bewegungen keine Änderungen am Text vornehmen.

Wir wollen das Verhalten wie bei der Autocomplete-Funktion von Kombinationsfeldern abbilden. Dort wird bei Betätigung dieser Tasten die übliche Funktion ausgeführt und die Markierung im Text wird entfernt.

Und auch hier bekommen wir das gewünschte Verhalten wieder frei Haus geliefert – es ist bereits implementiert.

Betätigen der Zurück- und der Entf-Taste

Sie ahnen es bereits: Auch hier brauchen Sie nichts zu tun. Wenn Sie die **Zurück**- oder die **Entf**-Taste betätigen, während der hintere Teil des Textes markiert ist, wird der markierte Text einfach gelöscht.

Eingabe weiterer Zeichen

Nun wird es allerdings wieder interessant: Wenn der Benutzer ein weiteres Zeichen eingibt, gibt es zwei Möglichkeiten:

- Das Zeichen entspricht dem ersten Zeichen der Markierung. Dann soll dieses Zeichen aus der Markierung herausgenommen werden und die übrigen in der Markierung enthaltenen Zeichen sollen weiterhin markiert bleiben.
- Das Zeichen entspricht nicht dem ersten Zeichen der Markierung. Dann soll die Prozedur eine neue Suche starten, wobei der erste, nicht markierte Teil des Textes im Textfeld plus dem soeben eingegebenen Zeichen als Suchbegriff verwendet wird.

Wir schauen uns erst den Fall an, dass der Benutzer das erste Zeichen der Markierung eingegeben hat. In diesem Fall wollen wir wissen, welches Zeichen der Benutzer eingegeben hat und dieses dann mit dem ersten Zeichen der Markierung vergleichen.

KeyCode, Shift und Ascii

Das ist mit den Parametern **KeyCode** und **Shift**, die wir von der Ereignisprozedur geliefert bekommen, nicht so einfach – zumindest dann nicht, wenn es sich bei den Zeichen nicht um Buchstaben oder Zahlen handelt. Wenn Sie beispielsweise ein kleines **a** eingeben, liefert **KeyCode** den Wert **65**. Wenn Sie ein großes **A** eingeben, also **Umschalt + a**, liefert **KeyCode** ebenfalls den Wert **65**. Zusätzlich liefert **Shift** den Wert **1**, also **acShiftMask**. Der Ascii-Wert für das große **A** ist jedoch **65**, für das kleine **a** lautet er **97**. Die Ascii-Werte für Zeichen ermitteln Sie mit der **Asc**-Funktion:

```
Debug.Print Asc("a")
97
```

Bei den Buchstaben ist das kein Problem: Wir könnten dann in einer kleinen Funktion prüfen, ob **Shift** den Wert

acShiftMask liefert. Ist das der Fall, entspricht der Wert von **KeyCode** dem Ascii-Wert **65**. Ist **Shift** nicht gleich **acShiftMask**, müssen wir zum Wert von **KeyCode** die Zahl **32** hinzu addieren.

Das funktioniert so für alle Buchstaben. Dann schauen wir uns weitere gängige Zeichen an, die in Textfeldern verwendet werden – zum Beispiel das Minus-Zeichen, das gern als Bindestrich verwendet wird. Die **Asc**-Funktion liefert dafür:

```
Debug.Print Asc("-")
 45
```

KeyCode liefert allerdings den Wert **189**. Damit würde die Funktion zum Übersetzen von **KeyCode** in Ascii schon umfangreicher werden. Genau genommen müssten wir damit alle Zeichen übersetzen, da wir ja nie wissen, welche Zeichen der Benutzer benötigt.

Allerdings gibt es noch eine Alternative: die Ereignisprozedur **Bei Taste**. Die sieht im leeren Zustand für unser Textfeld **txtArtikelname** wie folgt aus:

```
Private Sub txtArtikelname_KeyPress(KeyAscii As Integer)
    Debug.Print KeyAscii
End Sub
```

Der hier verwendete Parameter **KeyAscii** liefert genau die gleichen Werte, die auch die **Asc**-Funktion zurückgibt. Damit könnten wir also für die weiterführende Eingabe von Zeichen arbeiten.

Zeichen mit dem Bei Taste-Ereignis untersuchen

Aber wenn wir nun eine weitere Ereignisprozedur hinzunehmen, um die Eingabe regulärer Zeichen zu untersuchen – macht es dann Sinn, die Prüfung auf **Strg + Leerzeichen** in der Ereignisprozedur **Bei Taste ab** zu belassen? Und kommen sich die beiden Prozeduren dann nicht ins Gehege, weil die **Bei Taste**-Ereignisprozedur auch nochmal auf **Strg + Leertaste** reagiert?

Ja, es macht Sinn, denn die beiden Ereignisprozeduren kommen sich bei Eingabe der Tastenkombination **Strg + Leertaste** nicht ins Gehege. Genau genommen wird das Ereignis **Bei Taste** durch diese Tastenkombination überhaupt nicht ausgelöst. Also arbeiten wir zunächst mit dem Ereignis **Bei Taste** weiter.

Die entsprechende Ereignisprozedur füllen wir wie folgt:

```
Private Sub txtArtikelname_KeyPress(KeyAscii As Integer)
    Dim intSelStart As Integer
    Dim intSelLength As Integer
    Dim strText As String
    Dim strNaechster As String
    intSelStart = Me!txtArtikelname.SelStart
    intSelLength = Me!txtArtikelname.SelLength
    strText = Me!txtArtikelname.Text
    If intSelLength > 0 Then
        strNaechster = Mid(strText, intSelStart + 1, 1)
        If KeyAscii = Asc(strNaechster) Then
            With Me!txtArtikelname
                .Text = strText
                .SelStart = intSelStart + 1
                .SelLength = 255
            End With
        End If
    End If
End Sub
```

Der Plan ist prinzipiell gut: Wir erfassen wieder die Position der Markierung und die Länge der Markierung sowie den gesamten Text. Dann prüfen wir, ob es eine Markierung mit mehr als **0** Zeichen gibt. Ist das der Fall, ermitteln wir den ersten Buchstaben der Markierung und tragen diesen in die Variable **strNaechster** ein.

Wenn **KeyAscii** dem Ascii-Wert von **strNaechster** entspricht, hat der Benutzer genau das nächste Zeichen eingegeben. Dann schreiben wir den Text aus strText zurück in das Textfeld, lassen die Markierung gegenüber der vorherigen Markierung um ein Zeichen weiter hinten

Datenblattfunktionen einschränken

Die Datenblattansicht von Access bietet eine ganze Reihe von Funktionen wie das Sortieren nach den verschiedenen Feldern, das Filtern nach beliebigen Kriterien, das Anordnen von Spalten oder auch das Ein- und Ausblenden einzelner Felder. Manchmal möchten Sie aber vielleicht gar nicht, dass der Benutzer etwas am Design des Datenblatts ändert. Das ist etwa der Fall bei der Lösung aus dem Beitrag Datenblatt: Reihenfolge mehrerer Einträge ändern, wo die Datensätze immer nach einem Feld zur Festlegung der Reihenfolge angezeigt werden sollen. Der vorliegende Beitrag zeigt, wie Sie Möglichkeiten für den Benutzer einschränken.

Optionen per Kontext- und Popupmenü

Wenn Sie ein einfaches Formular in der Datenblattansicht öffnen, haben Sie gleich in den Spaltenköpfen eine Reihe von Möglichkeiten, um die Darstellung des Datenblatts zu manipulieren. Die erste ist offensichtlich: Sie erreichen diese über das Pfeil nach unten-Symbol im Spaltenkopf einer jeden Spalte. Diese bietet einige Sortier- und Filteroptionen (siehe Bild 1).

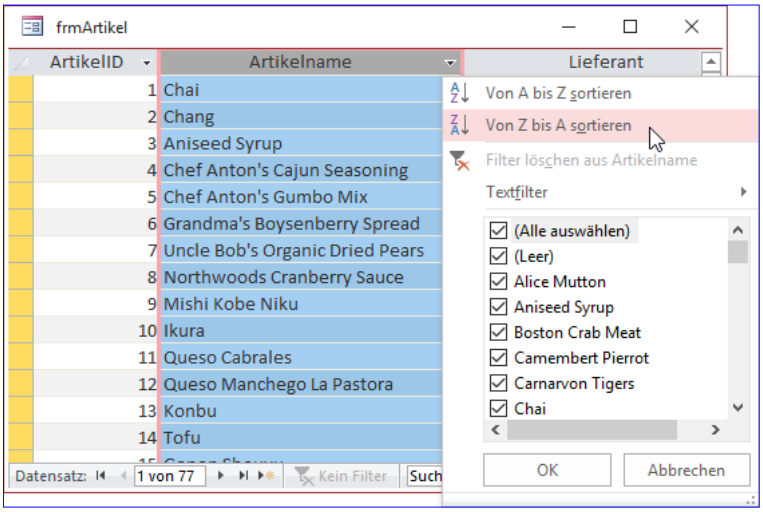


Bild 1: Aufklappmenü einer Spalte

Wenn Sie mit der rechten Maustaste auf den Spaltenkopf einer der Spalten klicken, erhalten Sie weitere Möglichkeiten zum Anpassen des Aussehens des Datenblatts. Hier kommen zu den bereits genannten noch Möglichkeiten zum Ein- und Ausblenden von Spalten und zum Einstellen der Spaltenbreiten hinzu (siehe Bild 2).

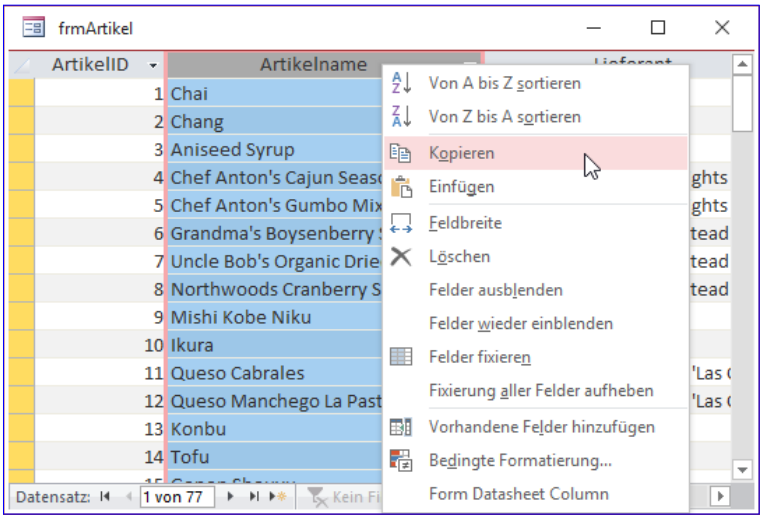


Bild 2: Kontextmenü einer Spalte

Diese Funktionen können wir dem Benutzer sehr schnell und effizient entziehen: Dazu brauchen Sie nur in den Entwurf des Formulars zu wechseln und dort das Eigenschaftenblatt zu öffnen – am schnellsten mit der Taste **F4**, wenn es nicht ohnehin schon eingeblendet ist.

Hier finden Sie im Bereich **Andere** die Eigenschaft **Kontextmenü** vor, die Sie auf den Wert **Nein** einstellen (siehe Bild 3).

Wechseln Sie danach in die Datenblattansicht des Formulars, finden Sie dieses wie in Bild 4 vor. Hier werden die Pfeile nach unten in den Spaltenköpfen gar nicht mehr angezeigt. Und auch Kontextmenüs werden beim Klicken mit der rechten Maustaste nicht mehr angezeigt. Damit wäre zumindest die Möglichkeit der Anpassung über diese beiden Menütypen schon einmal unterbunden.

Änderungen über das Ribbon

Wenn Sie jedoch im Ribbon auf den Tab-Reiter **Start** klicken, finden Sie dort einige der Befehle zum Filtern und Sortieren wieder (siehe Bild 5). Wie werden wir diese Befehle los beziehungsweise wie können wir diese deaktivieren? Die erste Lösung ist, dass Sie ein Ribbon definieren, dass all die eingebauten Befehle ausblendet und das mit dem Formular angezeigt wird, dessen Datensätze nicht sortiert oder gefiltert werden sollen. Die Definition dieses Ribbons, dass Sie in der Tabelle **USysRibbons** unterbringen, sieht so aus:

```
<?xml version="1.0"?>
<customUI xmlns="http://schemas.microsoft.com/of-
fice/2009/07/customui">
  <ribbon startFromScratch="true"/>
</customUI>
```

Es stellt also einfach nur den Wert des Attributs **startFromScratch** des **ribbon**-Elements auf **true** ein. Damit erhalten wir dann bei geöffnetem Formular das Ergebnis aus Bild 6.

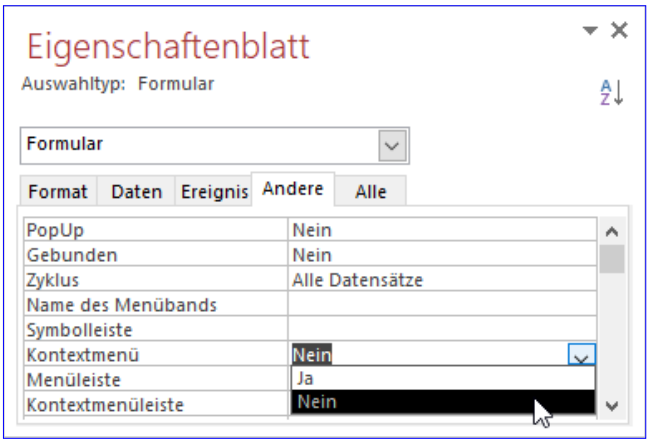


Bild 3: Kontextmenü deaktivieren

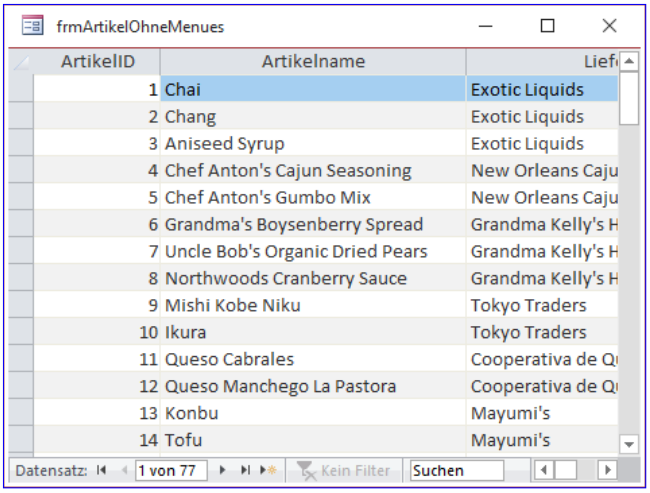


Bild 4: Datenblatt ohne Popup-Pfeile

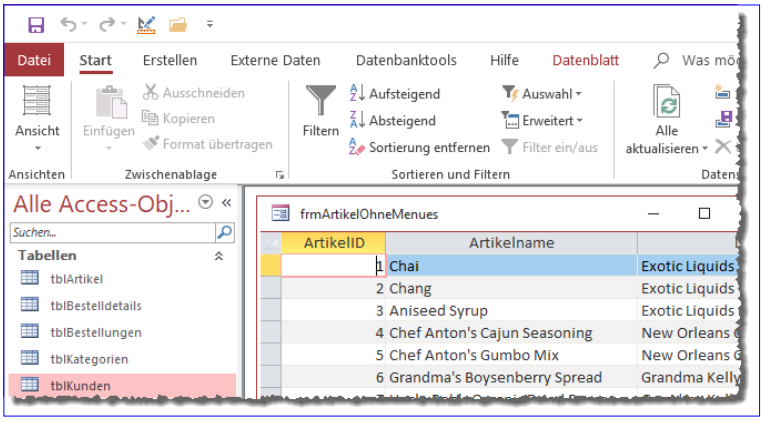


Bild 5: Befehle zum Sortieren und Filtern im Ribbon

Hier wurden alle Ribbon-Tabs mit Ausnahme von **Datenblatt** ausgeblendet, und das verbleibende Tab enthält keine Elemente mehr, die zum Ändern der Spalten oder der Reihenfolge der Datensätze benutzt werden können. Das wäre in Kombination mit der Deaktivierung der Kontextmenüs bereits eine wirksame Kombination.

Deaktivieren einzelner Befehle

Aber können wir das auch etwas feiner erledigen – also ohne direkt alle Befehle auszublenden? Also etwa so, dass der Benutzer noch

Tabellen vor unerlaubtem Zugriff schützen

In Ausgabe 3/2019 haben wir einige Elemente vorgestellt, die den Zugriff auf die Daten einer Datenbank einschränken sollen – zum Beispiel eine einfache Benutzerverwaltung und die Möglichkeit, den Zugriff auf Tabellen per Datenmakro einzuschränken. Im vorliegenden Beitrag wollen wir zeigen, wie Sie diese Techniken abrunden und den direkten Zugriff des Benutzers auf die Tabellen der Datenbank endgültig verhindern. Dazu exportieren wir diese in eine Backend-Datenbank, auf die wir diesmal nicht wie gewohnt per Verknüpfung zugreifen – sondern ausschließlich per VBA. Das hat den Vorteil, dass wir das Backend mit einem Kennwort versehen können, das nur im Code des Frontends zum Einsatz kommt – und diesen können wir durch Umwandlung in eine .mde- beziehungsweise .accde-Datenbank vor den Augen des Benutzers verbergen.

Voraussetzungen

Die Beispieldatenbank dieses Beitrags haben wir in einigen anderen Beiträgen vorbereitet. Der Beitrag **Benutzerverwaltung mit verschlüsselten Kennwörtern** (**www.access-im-unternehmen.de/1190**) zeigt, wie die Benutzerverwaltung programmiert wird.

Unter dem Titel **Zugriffsrechte mit Datenmakros** (**www.access-im-unternehmen.de/1193**) zeigen wir, wie Sie Tabellen so schützen, dass Sie nur noch nach der Anmeldung unter einem bestimmten Benutzerkonto auf die enthaltenen Daten zugreifen können.

Die Beispieldatenbank enthält also einen Anmeldedialog namens **frmAnmeldung**, mit dem der Benutzer die Zugangsdaten eingeben kann. Hat er keine Zugangsdaten eingegeben, ist durch die Definition entsprechender Datenmakros kein schreibender Zugriff auf die Tabellen der Datenbank möglich. Den lesenden Zugriff schränken wir noch durch die Prüfung der Berechtigungen beim Öffnen der jeweiligen Formulare ein.

Geplante Änderungen

Die erste Änderung, die wir durchführen, ist das Exportieren der Tabellen der Anwendung in eine Backend-Datenbank. Danach schützen wir diese Datenbank durch ein Datenbankkennwort. Schließlich müssen wir überall, wo wir in der Frontend-Datenbank auf die Daten der Tabellen oder Abfrage zugegriffen haben, entsprechenden VBA-Code für den Zugriff auf die Daten unter Berücksichtigung des Datenbankkennworts des Backends hinterlegen.

Natürlich ist das je nach der Komplexität der Anwendung eine mehr oder weniger große Aufgabe, aber wenn Sie eine Access-Anwendung ohne Einsatz des SQL Servers oder eines ähnlichen Systems unter maximaler Datensicherheit erstellen wollen, müssen Sie dies in Kauf nehmen. Schließlich wandeln Sie die Frontend-Datenbank in eine **.mde**- beziehungsweise **.accde**-Datenbank um, deren Code der Benutzer nicht mehr einsehen kann. Dieser Schritt ist wichtig, da wir das Kennwort für den Zugriff auf die Tabellen des Backends im Code hinterlegen.

Exportieren der Tabellen in eine Backend-Datenbank

Das Aufteilen der Datenbank erledigen Sie am einfachsten durch den Aufruf des Ribbon-Befehls **Datenbanktools|Daten verschieben|Access-Datenbank** (siehe Bild 1). Nach dem Aufruf wählen Sie im Dialog **Assistent zur Datenbankaufteilung** den Befehl **Datenbank aufteilen**. Anschließend brauchen Sie nur noch den Pfad der zu erstellenden Backend-Datenbank anzugeben. Wir nennen das Backend **Daten.accdb** und speichern es im gleichen Verzeichnis wie die Frontend-Datenbank.

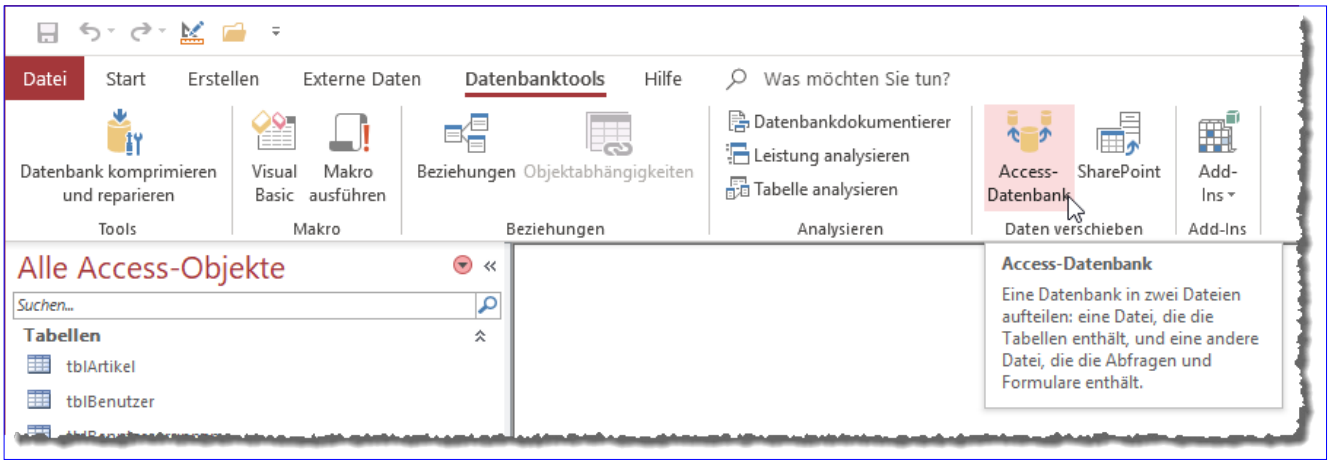


Bild 1: Aufruf des Befehls zum Aufteilen einer Datenbank

Danach zeigt die Frontend-Datenbank nicht mehr die Tabellen selbst im Bereich **Tabellen** des Navigationsbereich an, sondern nur noch die Verknüpfungen auf die Tabellen (siehe Bild 2). Diese befinden sich nun in der neu erstellten Backend-Datenbank.

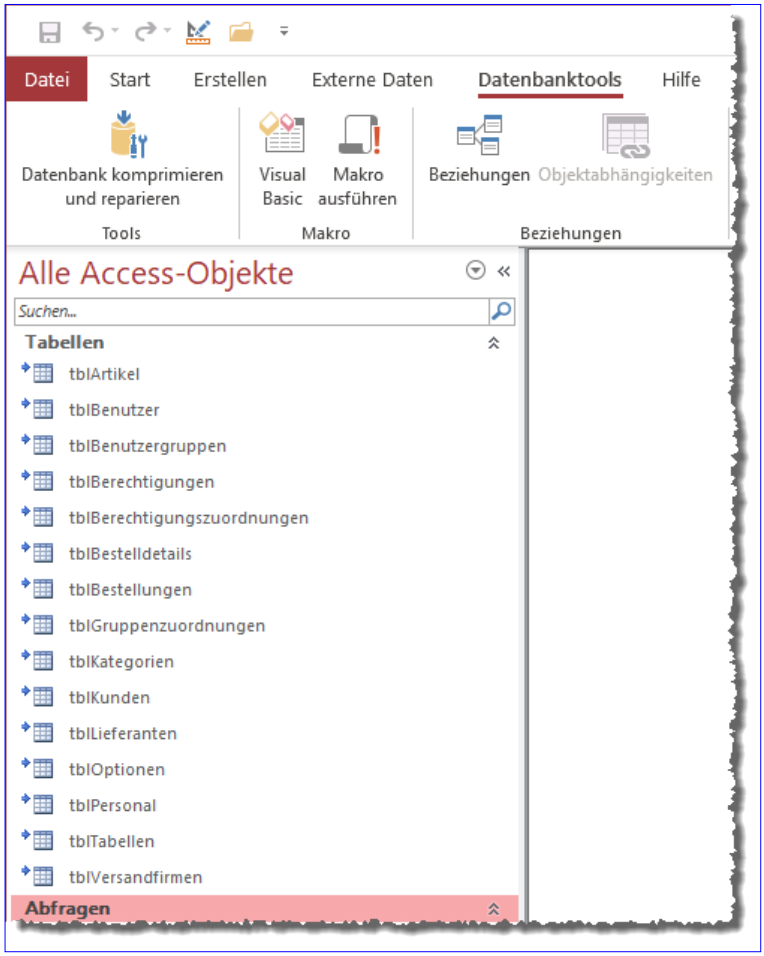


Bild 2: Verknüpfungen zu den Tabellen des Backends

Solange sich diese Verknüpfungen in der Frontend-Datenbank befinden, kann der Benutzer der Datenbank zumindest lesend auf die Daten der Tabellen zugreifen, da er diese über einen Doppelklick auf die Verknüpfungen öffnen kann. Um das zu verhindern, greifen wir zu einer harten Maßnahme: Wir löschen die Verknüpfungen schlicht und einfach. Damit funktionieren die Abfragen und Formulare, die an die Tabellen gebunden sind, nun nicht mehr, da die Datenquelle nicht mehr verfügbar ist. Und auch die Formulare wie etwa **frmBerechtigungenVerwalten**, die per VBA auf die Tabellen zugreifen, um die Daten in einer HTML-Tabelle anzuzeigen, versagen ihren Dienst.

Schützen der Backend-Datenbank durch ein Kennwort

Bevor wir die notwendigen Änderungen am Frontend vornehmen, um wieder auf die Daten der Tabellen zuzugreifen, legen wir ein Datenbankkennwort für die Backend-Datenbank fest. Dazu öffnen Sie diese direkt in einer

weiteren Access-Instanz. Dazu genügt ein Doppelklick auf die Datei **Daten.accdb**.

Anschließend öffnen Sie mit einem Klick auf den Ribbon-Reiter **Datei** den Backstage-Bereich. Unter Informationen finden Sie hier die Schaltfläche **Mit Kennwort verschlüsseln** (siehe Bild 3). Wenn Sie die Datenbank per Doppelklick geöffnet haben, ist diese noch nicht im Exklusiv-Modus geöffnet. Dies können Sie erledigen, indem Sie die Datenbank mit dem Ribbon-Befehl **DateiSchließen** schließen. Danach betätigen Sie den Ribbon-Befehl **DateiÖffnen**. Hier wählen Sie den Befehl **Durchsuchen** aus. Im nun erscheinenden **Öffnen**-Dialog wählen Sie die gewünschte Datei aus, in diesem Fall **Daten.accdb**, und wählen den Untereintrag **Exklusiv öffnen** der Schaltfläche **Öffnen** aus.

Bild 3: Zuweisen eines Datenbankkennworts zur Backend-Datenbank

Danach können Sie erneut den Ribbon-Reiter **Datei** anklicken und im Bereich **Information** die Schaltfläche **Mit Kennwort verschlüsseln** betätigen. Nun erscheint der Dialog **Datenbankkennwort festlegen**, in dem Sie das Kennwort zwei Mal eingeben und dieses dann durch einen Klick auf die Schaltfläche **OK** bestätigen (siehe Bild 4). Zu Beispielzwecken haben wir hier das Kennwort **kennwort** eingetragen.

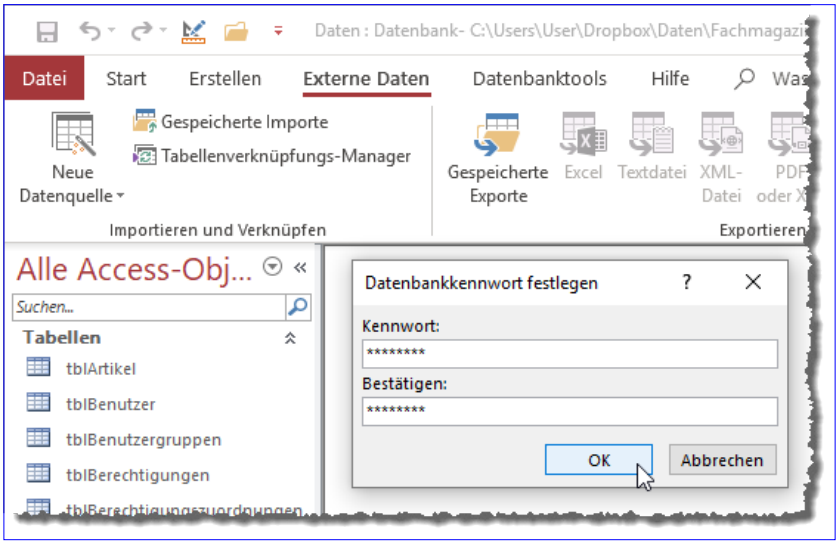


Bild 4: Festlegen des Datenbankkennworts

Wenn Sie die Backend-Datenbank nun schließen und erneut öffnen, erscheint der Dialog **Kennwort erforderlich** und verlangt nach der Eingabe des Kennworts.

Ersetzen der Bindung durch Zuweisen von Recordsets

Damit haben wir den Zugriff auf die Daten vom Frontend aus noch längst nicht wiederhergestellt – im Gegenteil, durch Vergabe des Kennworts haben wir diesen sogar noch erschwert. Doch nun werden wir Schritt für Schritt den Zugriff auf die Tabellen des Backends wieder hinzufügen.

Wenn wir die Frontend-Datenbank öffnen, sollte nach dem Bestätigen des Intro-Formulars der Dialog **frmAnmelden** erscheinen. Stattdessen erhalten wir die Fehlermeldung aus Bild 5. Als fehlerhafte Zeile wird die folgende Zeile markiert:

```
Set rst = db.OpenRecordset("SELECT Benutzername
FROM tblBenutzer", dbOpenDynaset)
```

Hier benötigen wir nur eine kleine Anpassung, die wie folgt aussieht:

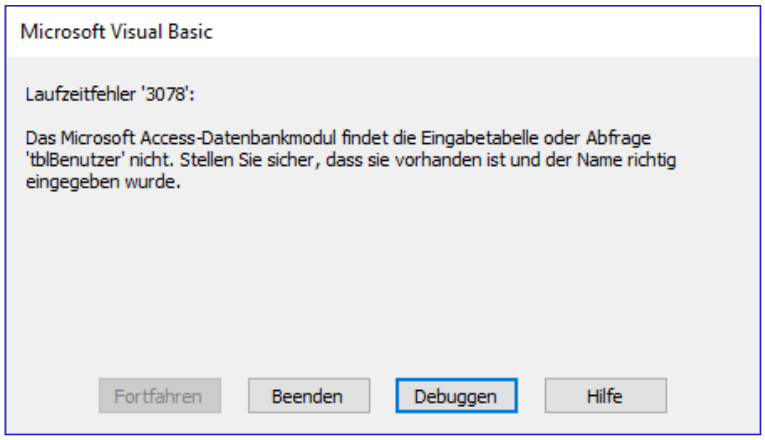


Bild 5: Fehlermeldung beim Versuch, auf die Tabelle **tblBenutzer** zuzugreifen

```
Set rst = db.OpenRecordset("SELECT Benutzername FROM
[;PWD=kennwort;DATABASE=" & CurrentProject.Path & "\Daten.
accdb].tblBenutzer", dbOpenDynaset)
```

Wir fügen also lediglich vor dem Namen der Tabelle einen Ausdruck in eckigen Klammern hinzu, der das Kennwort sowie den Pfad zur Backend-Datenbank enthält. Wenn wir das Formular **frmAnmeldung** nun öffnen, erscheint dieses ohne Fehler, denn es ist auch nicht an eine der Tabellen gebunden.

Während der Aktualisierung des Benutzernamens bei der Eingabe der Anmeldedaten wird jedoch in der Prozedur **txtBenutzername_Change** ein weiteres Recordset geöffnet, das die Tabelle **tblBenutzer** als Datenquelle verwendet. Hier passen wir die Anweisung wie folgt an:

```
Set rstBenutzer = db.OpenRecordset("SELECT * FROM
[;PWD=kennwort;DATABASE=" & CurrentProject.Path & "\Daten.
accdb].tblBenutzer WHERE Benutzername = '" & strBenutzer-
name & "'", dbOpenDynaset)
```

An dieser Stelle wird klar: Wir benötigen den Ausdruck **[;PWD=kennwort;DATABASE=" & CurrentProject.Path & "\Daten.accdb]** an mehr als einer Stelle. Wir können diesen Ausdruck an jeder Stelle einfügen, an der es erforderlich ist, allerdings wartet dann eine Menge Arbeit auf uns, wenn sich der Pfad der Backend-Datenbank ändert – oder auch das Kennwort dieser Datenbank. Also erstellen wir dazu zunächst eine einfache Funktion, welche uns die gewünschte Zeichenkette zusammenstellt:

```
Public Function BackendPathPassword() As String
    BackendPathPassword = "[;PWD=kennwort;DATABASE=" _
        & CurrentProject.Path & "\Daten.accdb]."
End Function
```

Der Zugriff auf diese Funktion sieht in der Anweisung aus dem ersten Beispiel wie folgt aus:

```
Set rst = db.OpenRecordset("SELECT Benutzername FROM " &
BackendPathPassword & "tblBenutzer", dbOpenDynaset)
```

Und für das zweite Beispiel sieht der Aufruf so aus:

```
Set rstBenutzer = db.OpenRecordset("SELECT * FROM " _
            & BackendPathPassword & "tblBenutzer WHERE ↵
            Benutzername = '" & strBenutzername & "'", ↵
                                        dbOpenDynaset)
```

Der nächste Fehler lauert in der Funktion **Anmeldung-Pruefen** im Modul **mdlBenutzerverwaltung** auf uns. Hier tritt der Fehler wieder in einem Aufruf der **OpenRecordset**-Methode auf. Wir machen kurzen Prozess und passen alle Aufrufe der **OpenRecordset**-Methode wie oben beschrieben an.

Kennwortgeschützter Zugriff per DLookup

Wenn wir die Schaltfläche **cmdAnmelden** im Formular **frmAnmeldung** betätigen, finden wir den nächsten Fehler in der folgenden **DLookup**-Anweisung:

```
lngBenutzerID = DLookup("BenutzerID", "tblBenutzer", _
                "Benutzername = '" & strBenutzername & "'")
```

Welche Möglichkeit haben wir hier, um den Zugriff auf die kennwortgeschützte Backend-Datenbank zu ermöglichen? Wir probieren es mit der folgenden Variante aus:

```
lngBenutzerID = DLookup("BenutzerID", BackendPathPassword _
   & "tblBenutzer", "Benutzername = '" & strBenutzername _
                                       & "'")
```

Wir fügen also einfach vorn an den zweiten Parameter den Ausdruck mit dem Kennwort und dem Pfad in eckigen Klammern an. Das Ergebnis ist allerdings ernüchternd und es sieht wie in Bild 6 aus.

PLookup statt DLookup

Statt der gewohnten Funktion **DLookup** verwenden wir eine benutzerdefinierte Version dieser Funktion, die wir **PLookup** nennen (für Password-Lookup). Zunächst schauen wir uns die kleine Änderung an, die den Aufruf betrifft:

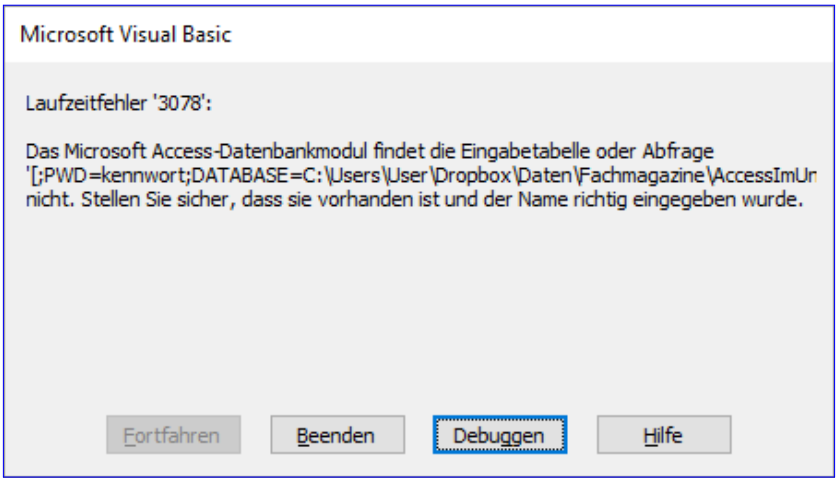


Bild 6: Fehler beim Versuch, das Kennwort in die **DLookup**-Abfrage zu integrieren

```
lngBenutzerID = PLookup("BenutzerID", "tblBenutzer", _
                "Benutzername = '" & strBenutzername & "'")
```

Die Funktion **PLookup** definieren wir im Modul **mdlBackend** wie folgt:

```
Public Function PLookup(strField As String, _
     strTable As String, strCriteria As String) As Variant
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT " & strField _
            & " FROM " & BackendPathPassword & strTable _
                & " WHERE " & strCriteria, dbOpenDynaset)
    If Not rst.EOF Then
        PLookup = rst.Fields(0)
    End If
End Function
```

Die Funktion **PLookup** erwartet die gleichen Parameter wie die eingebaute **DLookup**-Funktion. Sie erstellt ein neues Recordset, bei dem der Abfrageausdruck aus den zu einer **SELECT**-Anweisung zusammengesetzten Parametern der Funktion resultiert.

Dabei verwendet die Funktion beim Zusammenstellen der **SELECT**-Anweisung wieder die Funktion **BackendPathPassword**, welche den Ausdruck mit dem Kennwort und dem Backend-Pfad in eckigen Klammern und einen abschließenden Punkt liefert.

Sie können nun, ähnlich wie bei der Anpassung der Recordsets, auch alle Aufrufe der **DLookup**-Funktion anpassen, indem Sie **DLookup** durch **PLookup** ersetzen.

Abfragen anpassen

Die Funktion **BenutzerBerechtigungen** aus dem Modul **mdlBenutzerverwaltung**

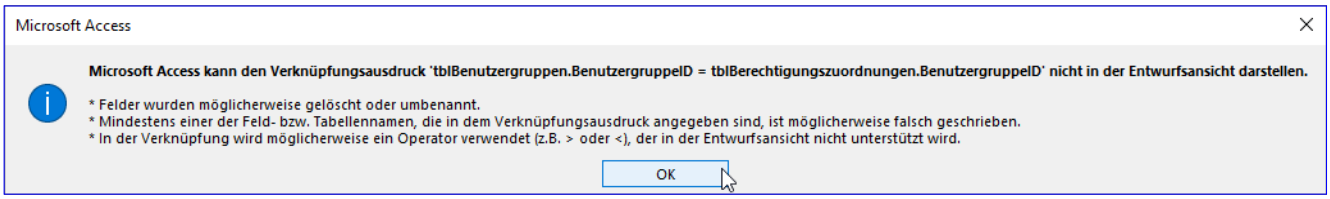


Bild 7: Fehler beim Anzeigen der Entwurfsansicht einer Abfrage

verwendet eine **SELECT**-Anweisung mit einer Abfrage als Datenquelle als Parameter einer **OpenRecordset**-Methode:

```
Set rst = db.OpenRecordset("SELECT * FROM ⤵
        qryBenutzerBerechtigungen WHERE BenutzerID = " ⤵
    & lngBenutzerID & " AND Tabelle = '" & strTabelle ⤵
                                & "'", dbOpenDynaset)
```

Die Abfrage **qryBenutzerBerechtigungen** referenziert logischerweise die Tabellen im Frontend beziehungsweise würde auch noch funktionieren, wenn die Verknüpfungen auf die Tabellen im Backend noch vorhanden wären.

Es gibt aber weder Tabellen noch Verknüpfungen im Frontend, weshalb die Abfrage nicht mehr funktionieren kann. Genau genommen können wir noch nicht einmal mehr den Entwurf ansehen – dies resultiert in der Fehlermeldung aus Bild 7.

Nach dem Schließen der Fehlermeldung erscheint immerhin noch die SQL-Ansicht der Abfrage (siehe Bild 8). Wir haben aber keine Möglichkeit, etwa über eine Eigenschaft festzulegen, dass die Abfrage auf die in der Backend-Datenbank befindlichen Tabellen zugreifen soll.

Wir wollen nun für jede Abfrage eine Funktion im Modul **mdlBackend** hinterlegen, welche den SQL-Code der Abfrage enthält und die Klausel mit dem Pfad und dem Kennwort zur Angabe der Quelltabellen hinzufügt. Dazu vereinfachen wir die Abfrage zunächst, indem wir Alias-Bezeichnungen für die Felder im **FROM**-Teil angeben und diese in der Feldliste und in den übrigen Bereichen wie den Kriterien und den Sortierungen nutzen. Vorher sieht die Abfrage noch wie folgt aus:

```
SELECT tblBenutzer.BenutzerID, tblBenutzer.Benutzerna-
me, tblTabellen.Tabelle, tblBerechtigungen.Berechtigung,
tblBerechtigungszuordnungen.BerechtigungID FROM tblTabel-
len INNER JOIN (tblBenutzer INNER JOIN ((tblBerechtigungen
INNER JOIN (tblBenutzergruppen INNER JOIN tblBerechti-
gungszuordnungen ON tblBenutzergruppen.Benutzergrupp-
eID = tblBerechtigungszuordnungen.BenutzergruppeID) ON
tblBerechtigungen.BerechtigungID = tblBerechtigungszuord-
nungen.BerechtigungID) INNER JOIN tblGruppenzuordnungen
ON tblBenutzergruppen.BenutzergruppeID = tblGruppenzu-
ordnungen.BenutzergruppeID) ON tblBenutzer.BenutzerID =
tblGruppenzuordnungen.BenutzerID) ON tblTabellen.TabelleID
= tblBerechtigungszuordnungen.TabelleID;
```

Nach den Änderungen erscheint diese schon etwas aufgeräumter:

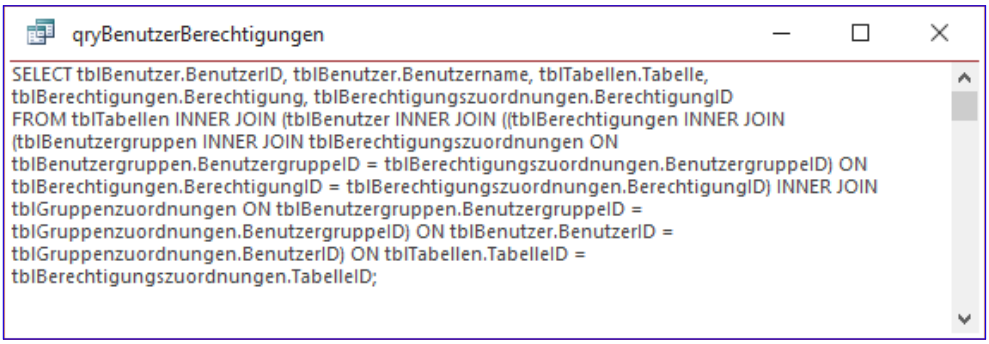


Bild 8: SQL-Ansicht der Abfrage

```
Public Function qryBenutzerBerechtigungen() As String
    Dim strSQL As String
    strSQL = "SELECT t2.BenutzerID, t2.Benutzername, t1.Tabelle, t3.Berechtigung, t5.BerechtigungID "
    strSQL = strSQL & "FROM " & BackendPathPassword & "tblTabellen AS t1 "
    strSQL = strSQL & "INNER JOIN (" & BackendPathPassword & "tblBerechtigungen AS t3 "
    strSQL = strSQL & "INNER JOIN (" & BackendPathPassword & "tblBenutzer AS t2 "
    strSQL = strSQL & "INNER JOIN ((" & BackendPathPassword & "tblBenutzergruppen AS t4 "
    strSQL = strSQL & "INNER JOIN " & BackendPathPassword & "tblGruppenzuordnungen AS t6 "
    strSQL = strSQL & "ON t4.BenutzergruppeID = t6.BenutzergruppeID) "
    strSQL = strSQL & "INNER JOIN " & BackendPathPassword & "tblBerechtigungszuordnungen AS t5 "
    strSQL = strSQL & "ON t4.BenutzergruppeID = t5.BenutzergruppeID) "
    strSQL = strSQL & "ON t2.BenutzerID = t6.BenutzerID) "
    strSQL = strSQL & "ON t3.BerechtigungID = t5.BerechtigungID) "
    strSQL = strSQL & "ON t1.TabelleID = t5.TabelleID"
    qryBenutzerBerechtigungen = strSQL
End Function
```

Listing 1: Funktion zum Zusammenstellen der Abfrage **qryBenutzerBerechtigungen**

```
SELECT t2.BenutzerID, t2.Benutzername, t1.Tabelle,
t3.Berechtigung, t5.BerechtigungID
FROM tblTabellen AS t1
INNER JOIN (tblBenutzer AS t2 INNER JOIN ((tblBerechtigun-
gen AS t3 INNER JOIN (tblBenutzergruppen AS t4 INNER JOIN
tblBerechtigungszuordnungen AS t5 ON t4.BenutzergruppeID
= t5.BenutzergruppeID) ON t2.BerechtigungID =
t5.BerechtigungID) INNER JOIN tblGruppenzuordnungen
AS t6 ON t4.BenutzergruppeID = t6.BenutzergruppeID)
ON t2.BenutzerID = t6.BenutzerID) ON t1.TabelleID =
t5.TabelleID;
```

Hier fügen wir nun innerhalb der wie folgt zu definierenden Funktion noch dynamisch den Inhalt der Funktion **BackendPathPassword** ein (siehe Listing 1). Und das ist auch der Grund, warum wir mit dem Schlüsselwort **AS** angegebene Alias-Bezeichnungen für die Tabellen arbeiten. Wenn wir immer die vollen Tabellennamen verwenden würden, müssten wir vor jedem Tabellennamen den Inhalt der Funktion **BackendPathPassword** einfügen. Wichtig ist noch, dass der SQL-String nicht mit einem Semikolon abschließt – gegebenenfalls wollen wir ja noch eine **WHERE**- oder **ORDER BY**-Klausel anhängen.

```
SELECT t2.BenutzerID, t2.Benutzername, t1.Tabelle, t3.Berechtigung, t5.BerechtigungID
FROM [;PWD=kennwort;DATABASE=C:\...\Daten.accdb].tblTabellen AS t1
INNER JOIN ([;PWD=kennwort;DATABASE=C:\...\Daten.accdb].tblBerechtigungen AS t3
INNER JOIN ([;PWD=kennwort;DATABASE=C:\...\Daten.accdb].tblBenutzer AS t2
INNER JOIN (([;PWD=kennwort;DATABASE=C:\...\Daten.accdb].tblBenutzergruppen AS t4
INNER JOIN [;PWD=kennwort;DATABASE=C:\...\Daten.accdb].tblGruppenzuordnungen AS t6
ON t4.BenutzergruppeID = t6.BenutzergruppeID)
INNER JOIN [;PWD=kennwort;DATABASE=C:\...\Daten.accdb].tblBerechtigungszuordnungen AS t5
ON t4.BenutzergruppeID = t5.BenutzergruppeID)
ON t2.BenutzerID = t6.BenutzerID)
ON t3.BerechtigungID = t5.BerechtigungID)
ON t1.TabelleID = t5.TabelleID
```

Listing 2: Die fertige Abfrage **qryBenutzerBerechtigungen**

Das Ergebnis der Funktion sieht wie in Listing 2 aus.

In der Funktion **BenutzerBerechtigungen** fügen wir die Funktion **qryBenutzerBerechtigungen** nun wie folgt in die **OpenRecordset**-Methode ein:

```
Set rst = db.OpenRecordset(qryBenutzerBerechtigungen _
    & " WHERE t2.BenutzerID = " & lngBenutzerID & " AND _
      t1.Tabelle = '" & strTabelle & "'", dbOpenDynaset)
```

Hier müssen wir auch noch den Tabellen-Alias für die beiden Parameter eintragen, also **t2.BenutzerID** statt **BenutzerID** und **t1.Tabelle** statt **Tabelle**. Sie sehen: So richtig komfortabel und einfach ist es nicht, es sind schon einige Anpassungen nötig.

Gebundene Formulare

Bei den gebundenen Formularen, also solchen Formularen, deren Eigenschaft **Datensatzquelle** die Tabelle oder Abfrage angibt, deren Daten das Formular anzeigen soll, verlieren wir den Komfort, den die Datenbindung bietet. So müssen Sie die Eigenschaft **Datensatzquelle** leeren und die Bindung an die Tabelle oder Abfrage über die Zuweisung eines **Recordset**-Objekts an die VBA-Eigenschaft **Recordset** erledigen.

Dadurch entfällt auch der Komfort, die Felder der Datensatzquelle einfach aus der Feldliste in das Formular zu ziehen.

Andererseits können Sie die Formulare auch zuerst bei aktivierter Datenbindung entwerfen und erst nach der Fertigstellung die Datenbindung entfernen.

Dazu müssten Sie natürlich temporär auch wieder die Tabellen in das Frontend holen – entweder als lokale Tabelle oder als Verknüpfung zur Tabelle im Backend.

Wir gehen an dieser Stelle davon aus, dass etwa das Formular **frmKunden** aktuell an die Tabelle **tblKunden** gebunden ist.

In diesem Fall führt das Öffnen des Formulars nach dem Entfernen der Tabellen und Verknüpfungen aus dem Frontend zu der Fehlermeldung **Die auf diesem Formular oder in diesem Bericht angegebene Datensatzquelle 'tblKunden' ist nicht vorhanden**.

Danach wird das Formular in der Entwurfsansicht angezeigt, wo auch gleich alle gebundenen Felder markiert sind. Ein Klick auf die Schaltfläche mit dem Ausrufezeichen aus Bild 9 zeigt den Grund für die Markierung – die für die Eigenschaft **Steuerelementinhalt** angegebenen Felder sind nicht in der Feldliste vorhanden.

Datensatzquelle ersetzen

Damit das Formular die Daten der Tabelle **tblKunden** aus dem Backend anzeigt, sind zwei Schritte nötig: das Entfer-

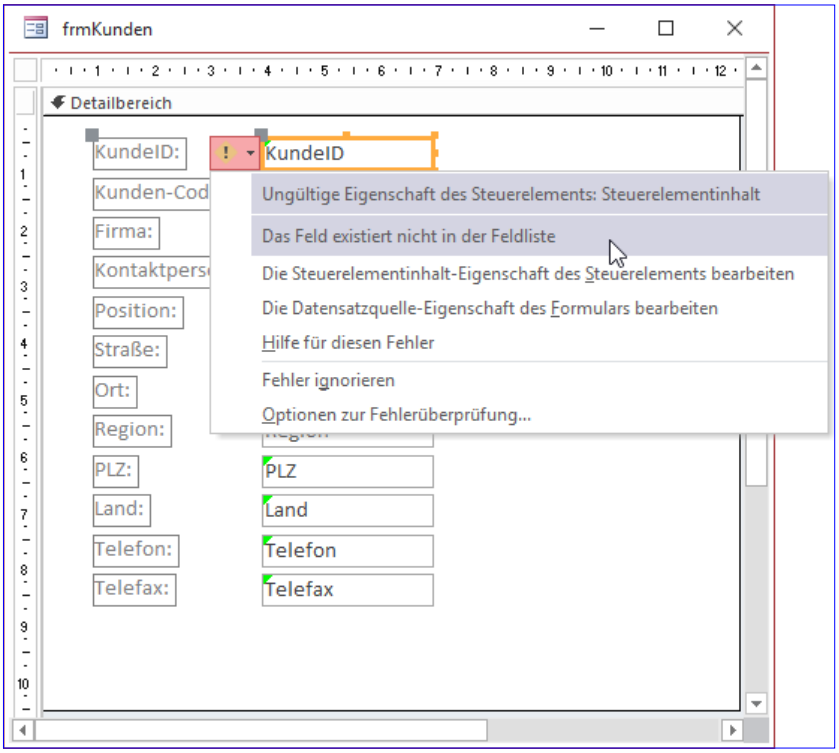


Bild 9: Meldung bei Steuerelementen mit fehlerhaftem Steuerelementinhalt