HAUFE.

Ausgabe 03/2026

# ACCESS

## IM UNTERNEHMEN

## FORMULARE FILTERN

Filteren Sie die Daten in Formularen mit unserer Lösung (ab Seite 50).

## In diesem Heft:

### DIE DBENGINE-KLASSE

Lernen Sie die DBEngine-Klasse und ihre praktischen Eigenschaften und Funktionen kennen.

SEITE 31

### FELDER BERECHNEN – WO IST DER BESTE ORT?

Erfahren Sie, ob man berechnete Felder besser in Tabellen oder in Abfragen nutzt.

SEITE 2

### FORMULARE MIT ADODB-RECORDSETS

Füllen Sie Formulare mit den Daten aus einem ADODB-Recordset.

SEITE 46

# Access-Formulare besser filtern

**Die eingebauten Filter-Funktionen für Access-Formulare sind zwar funktional, aber nicht besonders effizient zu bedienen. Man sieht nie, welcher Filter aktuell aktiv ist und welches Feld nach welchem Wert gefiltert wird. In einer der Lösungen in dieser Ausgabe von Access im Unternehmen liefern wir eine Filterfunktion, die in einem Formular das Hinzufügen von Filterkriterien erlaubt, die beliebig mit »Und« oder »Oder« verknüpft werden können und in allen Formularen flexibel eingesetzt werden können.**

Unserem universell einsetzbaren Filterformular widmen wir gleich zwei Beiträge: In **Filterformular für alle Fälle: Einbau und Bedienung** zeigen wir ab Seite 50, wie die Lösung genutzt werden kann.

Für alle, die wissen wollen, wie das Filterformular im Detail funktioniert, stellen wir im Beitrag **Ein Filterformular für alle Fälle: Die Programmierung** die Technik vor, die in diesem Formular steckt (ab Seite 57).

Uns erreichen immer wieder Fragen zu berechneten Feldern. In Access ist es möglich, solche Felder sowohl in Tabellen als auch in Abfragen zu verwenden. Im Beitrag **Berechnete Felder in Tabellen oder besser in Abfragen?** beleuchten wir ab Seite 2, welche Vor- und Nachteile die beiden Varianten haben und welche wir empfehlen.

In einer früheren Ausgabe haben wir uns bereits mit dem Füllen von ListView-Steuerelementen mit den Daten aus Tabellen oder Abfragen beschäftigt. Im Beitrag **ListView aus Tabellen oder Abfragen füllen, Teil 2** zeigen wir nun ab Seite 6, wie wir die ListView-Einträge bearbeiten können und wie wir eine Sortierung nach numerischen Werten sowie nach Datumsangaben umsetzen können.

Eine weitere Fortsetzung einer Beitragsreihe finden wir unter dem Titel **Mit Ordnern und Dateien im TreeView arbeiten, Teil 2** (ab Seite 23). Hier schauen wir uns an, wie wir die Änderungen, die wir an den Ordnern und Dateien im TreeView durchführen, auf das Dateisystem übertragen, um die Informationen auf beiden Seiten synchron zu halten.

Formulare zeigen normalerweise Daten aus lokalen oder verknüpften Tabellen an. Es gibt aber noch Alternativen. Eine davon ist das Füllen mit ADODB-Recordsets. Wie das genau funktioniert, besprechen wir im Beitrag **Detailformular und Datenblatt mit ADODB-Recordset** ab Seite 15.

Darauf baut unser Beitrag **Filtern und Sortieren von Formularen mit Recordset auf** (ab Seite 39). Wenn wir ein Formular mit den Daten eines ADODB-Recordsets füllen, funktionieren die eingebauten Funktionen zum Filtern und Sortieren nicht mehr wie erwartet. Im Beitrag zeigen wir, ob und wie wir diese Funktionen dennoch zum Laufen bringen können.

Wer in Access per VBA auf Datenbanken zugreift, nutzt meist die DAO-Bibliothek – oft ohne zu wissen, dass ganz oben in der gesamten Objekthierarchie die »DBEngine« steht. Sie repräsentiert die Datenbank-Engine selbst und bietet Zugriff auf Arbeitsbereiche, Verbindungen und globale Einstellungen. Alles zu dieser Klasse lesen Sie in **Die DBEngine-Klasse der DAO-Bibliothek** ab Seite 31.

Schließlich liefern wir im Beitrag **Button-Wizard programmieren** noch eine Lösung, mit der wir schnell optisch ansprechende Schaltflächen zu einem Formular hinzufügen können (ab Seite 68).

Viel Spaß beim Lesen wünscht Ihnen

Ihr André Minhorst

# Berechnete Felder in Tabellen oder besser in Abfragen?

Seit Access 2010 können Tabellenfelder einen berechneten Ausdruck enthalten – etwa die Verknüpfung von Vor- und Nachname oder das Produkt aus Menge und Preis. Die Funktion klingt praktisch, hat aber erhebliche Einschränkungen: Viele Funktionen stehen nicht zur Verfügung, das Ergebnis ist schreibgeschützt, und berechnete Felder lassen sich weder indizieren noch in Verknüpfungen verwenden. Dieser Beitrag zeigt, was geht, was nicht geht – und warum berechnete Spalten in Abfragen fast immer die bessere Wahl sind.

## Was sind berechnete Felder in Tabellen?

Ein berechnetes Feld ist ein Tabellenfeld mit dem Datentyp **Berechnet**. Statt eines gespeicherten Werts trägt es in der Feldeigenschaft **Ausdruck** eine Formel, die Access bei jeder Änderung der Quelldaten neu berechnet. Das Ergebnis erscheint in der Datenblattansicht und steht auch in Abfragen, Formularen und Berichten zur Verfügung, ohne dass dort erneut gerechnet werden muss.

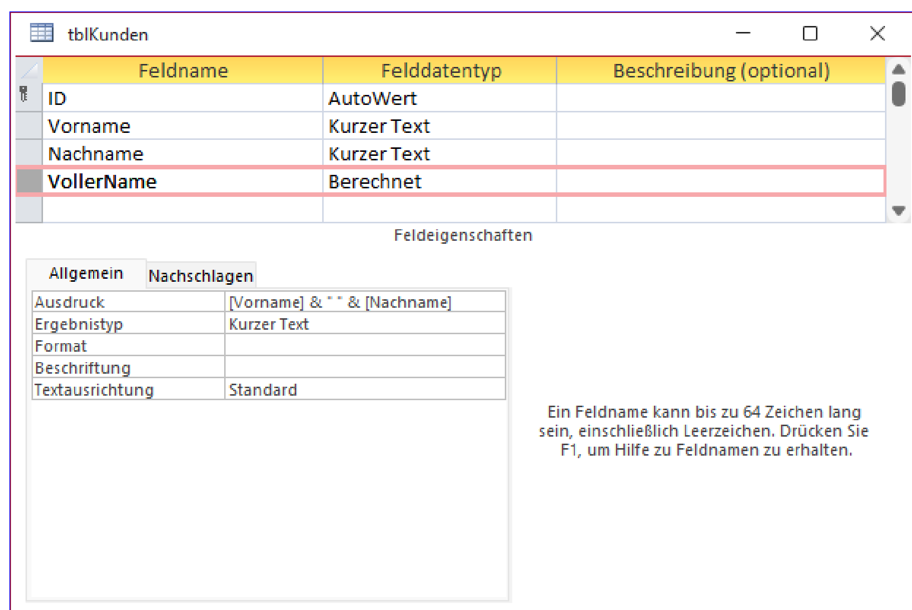


**Bild 1:** Ein berechnetes Feld im Tabellenentwurf – der Ausdruck wird in der Eigenschaft **Ausdruck** eingetragen, der Datentyp des Ergebnisses unter **Ergebnistyp**.

Das einfachste Beispiel ist die Zusammensetzung eines vollständigen Namens. Hat eine Tabelle die Felder **Vorname** und **Nachname**, lässt sich ein Feld **VollerName** so berechnen:

```
[Vorname] & " " & [Nachname]
```

tblKunden

| ID | Vorname | Nachname | VollerName |
|---|---|---|---|
| 1 | Anna | Müller | Anna Müller |
| 2 | Bernd | Schmidt | Bernd Schmidt |
| 3 | Clara | Weber | Clara Weber |
| (Neu) | | | |

Datensatz: 1 von 3 · Kein Filter · Suchen

**Bild 2:** Das berechnete Feld **VollerName** erscheint in der Datenblattansicht und steht automatisch in allen darauf basierenden Abfragen und Formularen bereit.

Die Eigenschaften tragen wir wie in Bild 1 ein.

Nach einem Wechsel in die Datenblattansicht erhalten wir das Ergebnis aus Bild 2.

## Welche Elemente sind in berechneten Tabellenfeldern erlaubt?

Der wichtigste Unterschied zu berechneten Spalten in Abfragen ist der stark eingeschränkte Funktionsumfang. Als Faustregel gilt: Nur die Funktionen, die der **Ausdrucks-**

**Generator** beim Erstellen eines berechneten Tabellenfeldes anbietet, funktionieren dort auch tatsächlich. Alle anderen werden beim Speichern mit einem Fehler abgewiesen.

- **Operatoren:** Arithmetische Operatoren (+, -, *, /), Vergleichsoperatoren (=, <, >, <=, >=, <>), Textverkettung (&), logische Operatoren (**And**, **Or**, **Not**).
- **Verfügbare Funktionen (Auswahl):** Textfunktionen: **Left**, **Right**, **Mid**, **Len**, **Trim**, **LTrim**, **RTrim**, **UCase**, **LCase**, **InStr**, **Replace**, **Str**, **Val**. Mathematische Funktionen: **Abs**, **Int**, **Round**, **Sqr**, **Mod**. Datumsfunktionen: **Year**, **Month**, **Day**, **Weekday**, **DateSerial**. Prüffunktionen: **IIf**, **IsNull**, **IsNumeric**, **Nz**.
- **Nicht verfügbar (Auswahl): Date()**, **Now()**, **DateDiff()**, **DateAdd()**, **Format()**, **CDate()**, **DLookup()** und alle anderen Domänenfunktionen (**DSum**, **DCount** usw.) sowie alle eigenen VBA-Funktionen. Ausdrücke, die auf Felder aus anderen Tabellen oder Abfragen zugreifen, sind ebenfalls nicht möglich.

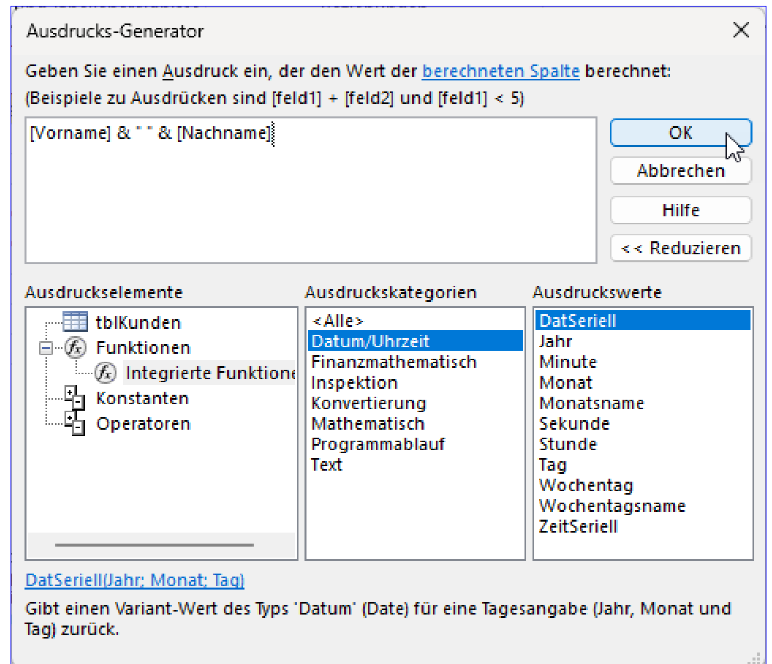


**Bild 3:** Der Ausdrucksgenerator mit einer Datumsberechnung

Die möglichen Elementen werden im Ausdrucks-Generator aufgelistet, den wir mit einem Klick auf die Schaltfläche mit den drei Punkten öffnen.

Der Ausdrucks-Generator zeigt im Kontext eines berechneten Tabellenfeldes nur die tatsächlich verfügbaren Funktionen – alles, was dort nicht aufgeführt ist, wird beim Speichern abgewiesen (siehe Bild 3).

## Typische Ausdrücke für berechnete Felder

Arithmetische Ausdrücke auf Basis von Feldern derselben Tabelle funktionieren zuverlässig. In einer Bestellposition mit den Feldern **Menge** und **Einzelpreis** liefert der folgende Ausdruck den Gesamtpreis:

```
[Menge] * [Einzelpreis]
```

Auch bedingte Ausdrücke mit **IIf** sind möglich, solange alle Argumente Feldwerte oder Konstanten sind. Der folgende Ausdruck berechnet einen Mengenrabatt:

```
IIf([Menge] >= 10, [Einzelpreis] * 0.9, [Einzelpreis])
```

Für Datumsberechnungen gilt: Funktionen, die das aktuelle Datum liefern, stehen nicht zur Verfügung. Einfache Differenzen zwischen zwei Datumsfeldern sind jedoch möglich, da Access Datumswerte intern als Zahlen speichert. Hat eine Tabelle die Felder **Bestelldatum** und **Lieferdatum**, lässt sich die Lieferdauer in Tagen direkt berechnen:

```
[Lieferdatum] - [Bestelldatum]
```

Das Ergebnis ist eine Zahl – der Ergebnistyp des berechneten Feldes sollte daher auf **Ganze Zahl** gesetzt werden. Auf demselben Prinzip basiert auch das Quartal eines Datums. Da **Format([Datum],"q")** in Tabellenfeldern nicht erlaubt ist, hilft ein mathematischer Umweg mit **Month**, Division und **Round**:

# ListView aus Tabellen oder Abfragen füllen, Teil 2

**Im ersten Teil dieser Beitragsreihe haben wir ein ListView-Steuerelement mit den Daten aus einer Mitarbeitertabelle gefüllt, Icons hinzugefügt und eine Sortierfunktion implementiert. Im vorliegenden Beitrag bauen wir dieses Beispiel weiter aus: Wir zeigen, wie man dem ListView-Steuerelement neue Einträge hinzufügen und bestehende Einträge bearbeiten oder löschen kann. Außerdem erklären wir, wie man die Elemente auch nach numerischen Daten oder nach Datumsfeldern korrekt sortiert. Schließlich fügen wir noch ein Kontextmenü hinzu, mit dem man den Status der Mitarbeiter anpassen kann.**

## Rückblick auf Teil 1

Im ersten Teil dieser Beitragsreihe (www.access-im-unternehmen.de/1574) haben wir zunächst eine Abfrage namens **qryMitarbeiter** erstellt, die alle benötigten Lookup-Tabellen – **tblAnreden**, **tblAbteilungen** und **tblStatus** – einbindet. Anschließend haben wir das **ListView**-Steuerelement in der Ereignisprozedur **Form_Load** eingerichtet und mit Spaltenüberschriften versehen.

Die Prozedur **ListViewFuellen** liest die Datensätze per Recordset aus und fügt sie als **ListItem**- und **ListSubItem**-Objekte ein. Außerdem haben wir per Windows-API-Aufruf die Spaltenbreiten automatisch an Inhalt und Überschrift angepasst, für jeden Mitarbeiter ein statusabhängiges Icon aus der Tabelle **MSysResources** eingeblendet und eine einfache Sortierfunktion per Klick auf den Spaltenkopf ergänzt. Bild 1 zeigt den Stand am Ende von Teil 1.

## Sortierung nach Zahlen- und Datumsfeldern

Im ersten Teil haben wir bereits eine Sortierfunktion eingebaut, die über einen Klick auf den jeweiligen Spaltenkopf ausgelöst wird. Dabei zeigte sich jedoch ein Problem: Zahlen und Datumsangaben werden lexikografisch sortiert, also wie Zeichenketten.

Das führt dazu, dass beispielsweise der Wert 10 vor dem Wert 2 erscheint und Datumsangaben nicht chronologisch, sondern nach ihrem Textwert geordnet werden.

frmMitarbeiterImListView

| ID | Anrede | Vorname | Nachname | Telefon | E-Mail | Geburtstag | Abteilung | Status |
|---|---|---|---|---|---|---|---|---|
| 3 | Herr | Wernfried | Birk | 040/370787 | wernfried@birk.de | 27.02.2015 | Softwareentwicklung | Verfügbar |
| 14 | Frau | Katharina | Bloch | 0531/5196531 | katharina@bloch.de | 31.01.2004 | Vertrieb | Urlaub |
| 21 | Frau | Roselind | Bohn | 089/3506491 | roselind@bohn.de | 28.12.2001 | Geschäftsführung | Krank |
| 11 | Herr | Hasko | Heigl | 089/056980 | hasko@heigl.de | 10.09.2007 | Geschäftsführung | Verfügbar |
| 16 | Herr | Detrich | Kock | 0234/6580164 | detrich@kock.de | 26.06.2010 | Softwareentwicklung | Dienstreise |
| 4 | Herr | Ulfried | Köster | 040/7398093 | ulfried@köster.de | 20.12.2007 | Softwareentwicklung | Dienstreise |
| 7 | Herr | Philip | Langbein | 089/6051172 | philip@langbein.de | 25.02.2015 | Softwareentwicklung | Krank |
| 15 | Frau | Gerti | Markmann | 0234/5802013 | gerti@markmann.de | 30.11.2013 | Buchhaltung | Verfügbar |
| 6 | Herr | Lutz | Mast | 030/5992692 | lutz@mast.de | 31.10.2007 | Buchhaltung | Verfügbar |
| 17 | Herr | Waldemar | Menzel | 030/9507444 | waldemar@menzel.de | 21.09.2012 | Vertrieb | Urlaub |
| 20 | Frau | Wiltrud | Nehls | 0341/230428 | wiltrud@nehls.de | 05.01.2007 | Geschäftsführung | Verfügbar |
| 5 | Frau | Inka | Neugebauer | 0711/542662 | inka@neugebauer.de | 26.08.2011 | Vertrieb | Verfügbar |
| 10 | Herr | Eckart | Pfister | 0221/927329 | eckart@pfister.de | 13.05.2006 | Buchhaltung | Verfügbar |
| 13 | Frau | Sonnhild | Pollak | 0231/903341 | sonnhild@pollak.de | 21.09.2008 | Vertrieb | Dienstreise |
| 8 | Frau | Adele | Scheele | 0228/399421 | adele@scheele.de | 25.05.2013 | Vertrieb | Verfügbar |
| 19 | Frau | Anne-Rose | Schenke | 0231/620037 | anne-rose@schenke.de | 19.05.2011 | Vertrieb | Verfügbar |
| 12 | Herr | Erdmann | Schöll | 0201/655434 | erdmann@schöll.de | 17.10.2011 | Softwareentwicklung | Verfügbar |

**Bild 1:** Aktueller Stand des **ListView**-Steuerelements

```
Private Const LVM_FIRST As Long = &H1000
Private Const LVM_SORTITEMSEX As Long = LVM_FIRST + 81

#If VBA7 Then
    Private Declare PtrSafe Function SendMessage Lib "user32" Alias "SendMessageA" (ByVal hWnd As LongPtr, _
        ByVal wMsg As Long, ByVal wParam As LongPtr, ByVal lParam As LongPtr) As LongPtr
#Else
    Private Declare Function SendMessage Lib "user32" Alias "SendMessageA" (ByVal hWnd As Long, ByVal wMsg As Long, _
        ByVal wParam As Long, ByVal lParam As Long) As Long
#End If
```

**Listing 1:** Benötigte API-Deklarationen

Um das zu beheben, ersetzen wir die eingebaute Sortierfunktion durch eigene Prozeduren, die Zahlen- und Datumswerte korrekt vergleichen.

Als Basis dienen die API-Konstanten und die **SendMessage**-Deklaration aus Listing 1, die wir im allgemeinen Modul **mdlListViewTabellen2** ablegen.

Für die eigentliche Sortierung erweitern wir die Ereignisprozedur **ctlListView_ColumnClick** aus Teil 1.

Nach dem Setzen von **SortKey** und **SortOrder** rufen wir eine eigene Prozedur **ListViewSortieren** auf, die den Vergleichstyp je Spalte festlegt (siehe Listing 2).

Die Prozedur **ListViewSortieren** im Modul **mdlListViewTabellen2** verzweigt per **Select Case** je nach Spaltenindex in die passende Sortierprozedur: Index 1 (ID) landet in **ListViewSortNumerisch**, Index 7 (Geburtstag) in **ListViewSortDatum**, alle anderen Spalten nutzen weiterhin die eingebaute Textsortierung über **objLV.Sorted = True** (siehe Listing 3).

Beide Sortierprozeduren arbeiten nach dem gleichen Prinzip: Sie vergleichen im Bubblesort-Durchlauf die Werte zweier benachbarter Zeilen direkt und tauschen bei Bedarf sofort deren Inhalte. Da die **Index**-Eigenschaft eines **ListItem**-Objekts schreibgeschützt ist, können wir die Reihenfolge der Einträge nicht durch Umsetzen von Indizes

```
Private Sub ctlListView_ColumnClick(ByVal ColumnHeader As Object)
    Static intAktuelleSortierspalte As Integer
    Static bolAufsteigend As Boolean
    Dim objListView As MSComctlLib.ListView
    Set objListView = Me.ctlListView.Object
    objListView.SortKey = ColumnHeader.Index - 1
    If intAktuelleSortierspalte = ColumnHeader.Index - 1 Then
        bolAufsteigend = Not bolAufsteigend
        objListView.SortOrder = Abs(bolAufsteigend)
    Else
        objListView.SortOrder = lvwAscending
    End If
    intAktuelleSortierspalte = ColumnHeader.Index - 1
    Call ListViewSortieren(objListView, ColumnHeader.Index - 1, bolAufsteigend)
End Sub
```

**Listing 2:** Angepasste ColumnClick-Prozedur

```
Public Sub ListViewSortieren(objLV As MSComctlLib.ListView, intSpalte As Integer, bolAufsteigend As Boolean)
    Select Case intSpalte
        Case 1 'ID: numerisch
            Call ListViewSortNumerisch(objLV, intSpalte, bolAufsteigend)
        Case 7 'Geburtstag: Datum
            Call ListViewSortDatum(objLV, intSpalte, bolAufsteigend)
        Case Else 'alle anderen: Text
            objLV.Sorted = True
    End Select
End Sub
```

**Listing 3:** Verzweigung nach Spaltentyp

verändern. Stattdessen lassen wir die **ListItem**-Objekte an ihrer Position und schreiben nur die angezeigten Inhalte um – das Ergebnis ist dasselbe.

## Routine zum Vertauschen von ListView-Einträgen

Den Austausch übernimmt die private Hilfsprozedur **ListViewZeilenTauschen**. Sie erhält das **ListView**-Objekt sowie die Indizes der beiden zu tauschenden Zeilen. Zunächst speichert sie den **Text** der ersten Spalte von Zeile A in einer Hilfsvariablen, kopiert den Wert aus Zeile B nach A und schreibt den gespeicherten Wert nach B – der klassische Drei-Wege-Tausch. Anschließend wiederholt sie diesen Vorgang für jedes **ListSubItem** der Zeile. Da im **ListView**-Steuerelement auch das Icon je Zeile gespeichert ist, tauscht die Prozedur zuletzt noch den **SmallIcon**-Index, damit das Icon beim Umsortieren mitwandert und weiterhin zum jeweiligen Datensatz passt (siehe Listing 4).

## Sortieren mit Bubblesort

Die Prozedur **ListViewSortNumerisch** implementiert den Bubblesort mit zwei verschachtelten Schleifen. Die äußere Schleife über **i** läuft von 1 bis **lngAnzahl - 1** und steuert

```
Private Sub ListViewZeilenTauschen(objLV As MSComctlLib.ListView, lngA As Long, lngB As Long)
    Dim strTemp As String
    Dim i As Long
    strTemp = objLV.ListItems(lngA).Text
    objLV.ListItems(lngA).Text = objLV.ListItems(lngB).Text
    objLV.ListItems(lngB).Text = strTemp
    For i = 1 To objLV.ListItems(lngA).ListSubItems.Count
        strTemp = objLV.ListItems(lngA).ListSubItems(i).Text
        objLV.ListItems(lngA).ListSubItems(i).Text = objLV.ListItems(lngB).ListSubItems(i).Text
        objLV.ListItems(lngB).ListSubItems(i).Text = strTemp
    Next i
    'SmallIcon-Index ebenfalls tauschen
    Dim lngIconA As Long
    lngIconA = objLV.ListItems(lngA).SmallIcon
    objLV.ListItems(lngA).SmallIcon = objLV.ListItems(lngB).SmallIcon
    objLV.ListItems(lngB).SmallIcon = lngIconA
End Sub
```

**Listing 4:** Hilfsprozedur zum Tauschen zweier **ListView**-Zeilen

```
Public Sub ListViewSortNumerisch(objLV As MSComctlLib.ListView, intSpalte As Integer, bolAufsteigend As Boolean)
    Dim i As Long, j As Long
    Dim lngAnzahl As Long
    lngAnzahl = objLV.ListItems.Count
    'Bubblesort: Zeileninhalt direkt tauschen
    For i = 1 To lngAnzahl - 1
        For j = 1 To lngAnzahl - i
            Dim lngA As Long, lngB As Long
            If intSpalte = 0 Then
                lngA = CLng(objLV.ListItems(j).Text)
                lngB = CLng(objLV.ListItems(j + 1).Text)
            Else
                lngA = CLng(objLV.ListItems(j).ListSubItems(intSpalte).Text)
                lngB = CLng(objLV.ListItems(j + 1).ListSubItems(intSpalte).Text)
            End If
            If bolAufsteigend Then
                If lngA > lngB Then Call ListViewZeilenTauschen(objLV, j, j + 1)
            Else
                If lngA < lngB Then Call ListViewZeilenTauschen(objLV, j, j + 1)
            End If
        Next j
    Next i
End Sub
```

**Listing 5:** Numerische Sortierung

die Durchläufe. Die innere Schleife über **j** läuft von 1 bis **lngAnzahl - i** – nach jedem Durchlauf steht der größte noch unsortierte Wert am Ende, weshalb der Vergleichsbereich mit jedem Durchlauf um eine Position kürzer wird. Im Schleifenrumpf lesen wir die Werte der beiden betrachteten Zeilen aus: Bei Spaltenindex 0 steht der Wert in **ListItems(j).Text**, bei allen anderen Spalten in **ListItems(j).ListSubItems(intSpalte).Text**. Beide Werte wandeln wir per **CLng** in ganzzahlige Werte um und vergleichen sie.

Ist die Reihenfolge falsch – bei aufsteigender Sortierung also **lngA > lngB**, bei absteigender **lngA < lngB** – rufen wir **ListViewZeilenTauschen** auf (siehe Listing 5).

Die Prozedur **ListViewSortDatum** ist identisch aufgebaut – der einzige Unterschied ist die Typumwandlung per **CDbl(CDate(...))**, da Access Datumsangaben intern als Gleitkommazahl speichert (siehe Klassenmodul **Form_frmMitarbeiterImListView**).

Nach diesen Anpassungen liefert ein Klick auf den Spaltenkopf **ID** eine korrekte numerische Sortierung, und ein Klick auf **Geburtstag** eine chronologische.

In Bild 2 sehen wir, dass die Sortierung nach dem Geburtstag nun funktioniert.

## Bearbeitungs-Dialog per Doppelklick öffnen

Damit der Benutzer einen Mitarbeiterdatensatz direkt aus dem **ListView**-Steuerelement heraus bearbeiten kann, öffnen wir per Doppelklick auf einen Eintrag ein Bearbeitungsformular. Das Formular **frmMitarbeiterDetail** ist an die Tabelle **tblMitarbeiter** gebunden und enthält alle Felder des Datensatzes (siehe Bild 3).

Die Eigenschaft **Automatisch zentrieren** stellen wir aus **Ja**, **Bildlaufleisten**, **Navigationsschaltflächen** und **Datensatzmarkierer** auf **Nein** und **Zyklus** auf **Aktueller Datensatz** ein.

# Detailformular und Datenblatt mit ADODB-Recordset

**Access bietet von Haus aus eine komfortable Datenbindung für Formulare: Man trägt eine Tabelle oder Abfrage als Datensatzquelle ein, bindet die Steuerelemente über ihren Steuerelementinhalt an Felder – und Access erledigt den Rest. Wer jedoch sicherstellen möchte, dass seine Tabellen nicht von außen ausgelesen werden können, darf keine Tabellen oder Tabellenverknüpfungen mehr im Frontend haben. Wir müssen uns also nach einer Alternative umsehen. Dazu entfernen wir alle kritischen Elemente aus dem Frontend und bauen die Formulare um. Aber wie greifen wir auf die Daten zu? Dazu nutzen wir die Recordset-Eigenschaft von Formularen und Steuerelementen. Diese können ein per VBA zusammengestelltes Recordset nutzen. In diesem ersten Teil zeigen wir, wie das mit einfachen Detailformularen und in der Datenblattansicht funktioniert. Danach bauen wir unser Wissen aus und beschäftigen uns mit Haupt- und Unterformularen und Steuerelementen wie dem Kombinations- und dem Listenfeld.**

## ADODB statt DAO

Warum nutzen wir in diesem Artikel ADODB statt DAO? Weil es flexibler ist, gerade wenn wir die vorgestellten Techniken auch für den Zugriff auf SQL Server-Datenbanken nutzen wollen.

## Beispieldatenbank: Frontend mit kennwortgeschütztem Backend

Als Beispiel nutzen wir eine Frontend-Datenbank, die aktuell noch über Tabellenverknüpfungen auf die Tabellen eines Backends zugreift. Dieses ist wiederum per Kennwort geschützt.

Das ist ein wichtiger Baustein im Hinblick auf die Sicherheit der Daten, denn wenn der Benutzer Zugriff auf das Backend hat, kann er dieses theoretisch auch einfach kopieren. In diesem Fall wären die geplanten Sicherheitsmaßnahmen im Frontend nutzlos.

Unser Frontend enthält zu Beginn noch die Verknüpfungen zu den Tabellen im Backend, die wir mit dem Bordmitteln von Access hinzugefügt haben (siehe Bild 1).

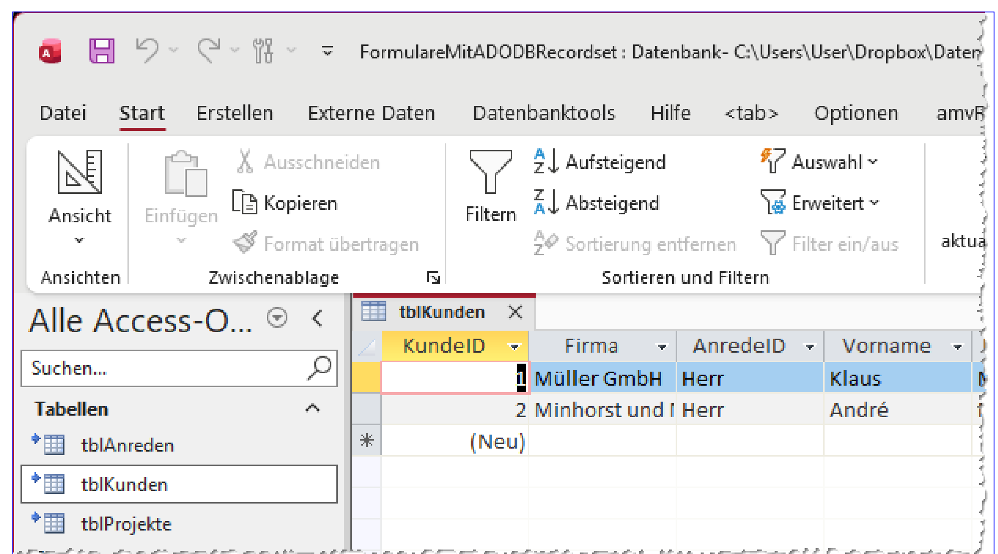


**Bild 1:** Datenbank mit Tabellenverknüpfungen

### ADODB-Bibliothek referenzieren

Wir können mit Early Binding oder Late Binding auf die Elemente der ADODB-Bibliothek zugreifen.

Zum Programmieren nutzen wir allerdings Early Binding, da wir so IntelliSense verwenden können. Dazu fügen wir dem VBA-Projekt einen entsprechenden Verweis hinzu (siehe Bild 2).

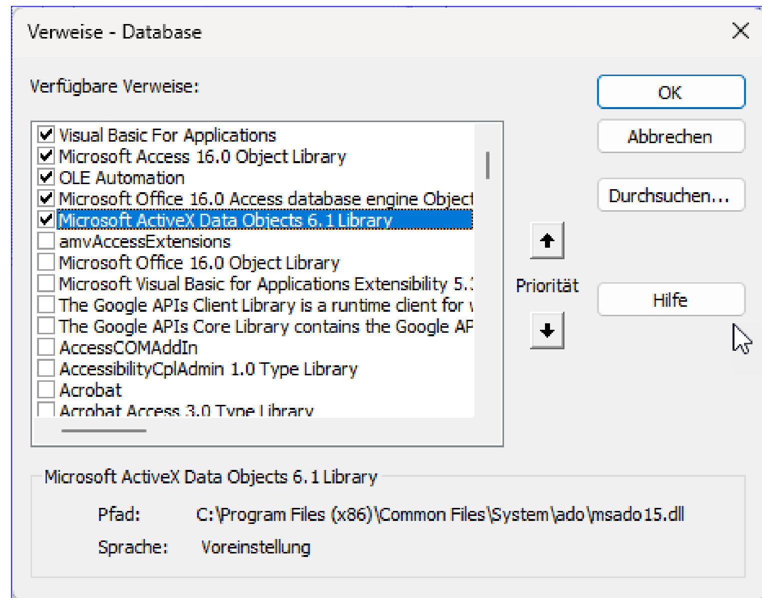


**Bild 2:** Verweis auf die ADODB-Bibliothek

### VBA-Funktionen für den Zugriff auf Connection und Recordset

Wir werden immer wieder den gleichen Code verwenden, um die Recordset-Eigenschaften von Formularen und Steuerelementen mit Daten zu füllen. Daher legen wir gleich ein paar Funktionen an, welche diese Aufgabe erledigen und rufen diese einfach vom Formular aus auf.

Das Modul **mdlADODB** enthält alle Hilfsfunktionen, die wir für den Zugriff auf die Datenbank benötigen. Wir legen es einmalig an und verwenden es in allen Formularen der Anwendung. Die erste Konstante steuert, welches Backend verwendet wird:

```
'Art der Verbindung:
'1 = CurrentProject.Connection
'2 = Access-Backend (.accdb)
'3 = SQL Server
Private Const cIntConnection As Integer = 2
```

Mit dem Wert **1** nutzen wir **CurrentProject.Connection** – also die Verbindung, die Access für das aktuelle Frontend ohnehin offen hält. Das ist sinnvoll, wenn Frontend und Backend dieselbe Datenbank sind oder wenn die Tabellen des Backends per Tabellenverknüpfung eingebunden sind. Mit dem Wert **2** öffnen wir eine eigene ADODB-Verbindung auf eine kennwortgeschützte Access-Backend-Datenbank. Mit dem Wert **3** verbinden wir uns mit einem SQL Server. Der Wechsel des Backends ist damit auf eine einzige Zeile reduziert.

Die Verbindungsstrings für die beiden echten Verbindungen stehen als Konstanten im Modul. Für das Access-Backend ergänzen wir den Provider-String später um den Dateipfad und das Kennwort:

```
'Verbindungsstring für Access-Backend (ACE-Provider)
Private Const strCnnAccess As String = _
    "Provider=Microsoft.ACE.OLEDB.12.0;Data Source="
'Verbindungsstring für SQL Server (Windows-Authentifiz.)
Private Const strCnnSQLServer As String = _
    "Provider=MSOLEDBSQL;Server=amvDesktop2023;" & _
    "Database=ADODBRecordsets;Trusted_Connection=Yes;"
```

Für das Access-Backend benötigen wir zusätzlich den Dateinamen der Backend-Datenbank sowie das Kennwort. Beides steht ebenfalls als Konstante im Modul. Den vollständigen Pfad ermitteln wir zur Laufzeit per **GetDBPfad**, damit die Anwendung auch nach einem Verschieben von Frontend und Backend in einen anderen Ordner weiterhin funktioniert:

```
'Dateiname des Access-Backends
Private Const strDBName As String = _
    "FormulareMitADODBRecordset_BE.accdb"
```

```
'Datenbankpasswort des Access-Backends
Private Const strDBPasswort As String = "kennwort"

Private Function GetDBPfad() As String
    GetDBPfad = CurrentProject.Path & "\" & strDBName
End Function
```

Nur der Dateiname steht fest im Code – der Ordner wird immer zur Laufzeit aus **CurrentProject.Path** abgeleitet. Wer Backend und Frontend in getrennten Ordnern betreibt, trägt stattdessen einfach den vollständigen Pfad als Konstante ein.

**CurrentProject.Path** kann auch durch einen festen Pfad ersetzt werden. Alternativ kann man die Funktion auch so umbauen, dass der Pfad beispielsweise aus der Registry eingelesen wird. Wenn dort kein Pfad gespeichert ist, könnte man diesen per Datei-Dialog abfragen.

## Die Funktion GetConnection

Die Funktion **GetConnection** ist der zentrale Baustein des Moduls. Sie gibt abhängig von **cIntConnection** eine geöffnete **ADODB.Connection** zurück.

Alle anderen Funktionen des Moduls – und alle Formulare der Anwendung – rufen ausschließlich diese eine Funktion auf, um eine Verbindung zu erhalten (siehe Listing 1).

Bei **Case 1** verlassen wir die Funktion sofort per **Exit Function**: **CurrentProject.Connection** ist bereits offen und gehört Access – wir dürfen sie weder über **cnn.Open** öffnen noch später schließen.

Bei den Werten **2** und **3** stellen wir zunächst den Verbindungsstring zusammen und öffnen am Ende gemeinsam eine neue Verbindung. Diese neue Verbindung müssen wir später auch wieder schließen – dazu gleich mehr.

```
Public Function GetConnection() As ADODB.Connection
    Dim cnn As ADODB.Connection
    Dim strConn As String
    Select Case cIntConnection
        Case 1
            'CurrentProject.Connection wiederverwenden
            Set GetConnection = CurrentProject.Connection
            Exit Function
        Case 2
            'Access-Backend: Verbindungsstring zusammenstellen
            strConn = strCnnAccess & GetDBPfad()
            If strDBPasswort <> "" Then
                strConn = strConn & ";Jet OLEDB:Database Password=" & strDBPasswort
            End If
        Case 3
            'SQL Server: Verbindungsstring direkt verwenden
            strConn = strCnnSQLServer
    End Select
    Set cnn = New ADODB.Connection
    cnn.Open strConn
    Set GetConnection = cnn
End Function
```

**Listing 1:** Die Funktion **GetConnection**

```
Public Function GetRecordset(ByVal strSQL As String, Optional ByVal lngCursorType As Long = adOpenStatic, _
        Optional ByVal lngLockType As Long = adLockOptimistic) _
        As ADODB.Recordset
    Dim cnn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Set cnn = GetConnection()
    Set rst = New ADODB.Recordset
    rst.CursorLocation = adUseClient
    rst.Open strSQL, cnn, lngCursorType, lngLockType
    Set GetRecordset = rst
End Function
```

**Listing 2:** Die Funktion **GetRecordset**

Beim Access-Backend ergänzen wir den Provider-String um den Pfad aus **GetDBPfad()** und, falls vorhanden, um das Kennwort.

Der Schlüssel für das Datenbankpasswort lautet **Jet OLEDB:Database Password** – dieser Name stammt noch aus der Zeit des Jet-Datenbankmoduls und wird vom neueren ACE-Provider unverändert weiterverwendet.

## Die Funktion GetRecordset

Die zweite Hilfsfunktion **GetRecordset** öffnet ein Recordset auf Basis eines SQL-Strings und gibt es zurück. Sie kapselt die immer gleichen Schritte – Verbindung holen, Recordset anlegen, Cursor konfigurieren, öffnen – so dass wir in den Formularen nur noch den SQL-String übergeben müssen (siehe Listing 2).

Cursor- und Sperrtyp sind als optionale Parameter mit sinnvollen Standardwerten ausgelegt. **adOpenStatic** liefert einen statischen Cursor, der alle Daten einmalig in den Arbeitsspeicher lädt. Das ist die Voraussetzung für **adUseClient**, denn clientseitige Cursor unterstützen nur statische oder Forward-Only-Cursor. **adLockOptimistic** sperrt den Datensatz erst beim **Update**-Aufruf kurz, was Konflikte im Mehrbenutzerbetrieb minimiert.

Warum **adUseClient** fest und nicht als Parameter? Weil wir das Recordset anschließend an die **Recordset**-Eigenschaft eines Formulars binden wollen – und Access erwartet dafür zwingend einen clientseitigen Cursor. Wer ein Recordset nur lesend durchlaufen möchte, ohne es ans Formular zu binden, kann zwar auch einen serverseitigen Cursor verwenden, aber für den Zweck dieser Beitragsreihe ist **adUseClient** die richtige Wahl.

Ein wichtiges Detail: **cnn** ist hier eine lokale Variable. Bei **adUseClient** ist das kein Problem – ADODB trennt das Recordset nach dem Öffnen intern von der Verbindung, sodass die Connection freigegeben werden kann, ohne das Recordset zu beeinträchtigen.

Das Recordset lebt danach eigenständig im Speicher. Bei einem serverseitigen Cursor wäre das anders: Dort muss die Verbindung so lange offen bleiben, wie das Recordset verwendet wird.

## Die Testprozedur

Bevor wir das Modul in Formularen einsetzen, prüfen wir mit einer einfachen Testprozedur, ob Verbindung und Recordset korrekt funktionieren. **Test_GetRecordset** öffnet ein Recordset mit den ersten zehn Datensätzen der Tabelle **tblKunden** und gibt **KundeID** und **Firma** im Direktbereich aus:

```
Public Sub Test_GetRecordset()
    Dim rst As ADODB.Recordset
    Set rst = GetRecordset( _
        "SELECT TOP 10 * FROM tblKunden")
```

# Mit Ordnern und Dateien im TreeView arbeiten, Teil 2

**Im ersten Teil dieses Beitrags haben wir die Grundlagen für die Arbeit mit Ordnern und Dateien im TreeView-Steuerelement gelegt: Ordner und Dateien im Explorer anzeigen und öffnen, das Ermitteln des Pfades über die Tabellen-IDs, Kontextmenüs für Ordner- und Datei-Elemente sowie die vollständige Implementierung der Ordner-Operationen Kopieren, Ausschneiden, Einfügen, Umbenennen und Löschen. Im vorliegenden zweiten Teil ergänzen wir die noch fehlenden Datei-Operationen und erörtern die Funktionen im Modul »mdlDateisystem«, welche die TreeView-Änderungen tatsächlich auf das Dateisystem übertragen.**

## Voraussetzung

Dieser Beitrag baut unmittelbar auf Teil 1 auf (**www.access-im-unternehmen.de/1585**). Alle dort beschriebenen Module, Funktionen und Formulare werden vorausgesetzt und hier nicht erneut erklärt.

Wir setzen auf dem dort beschriebenen Formular **frmDateienImTreeView** auf (siehe Bild 1).

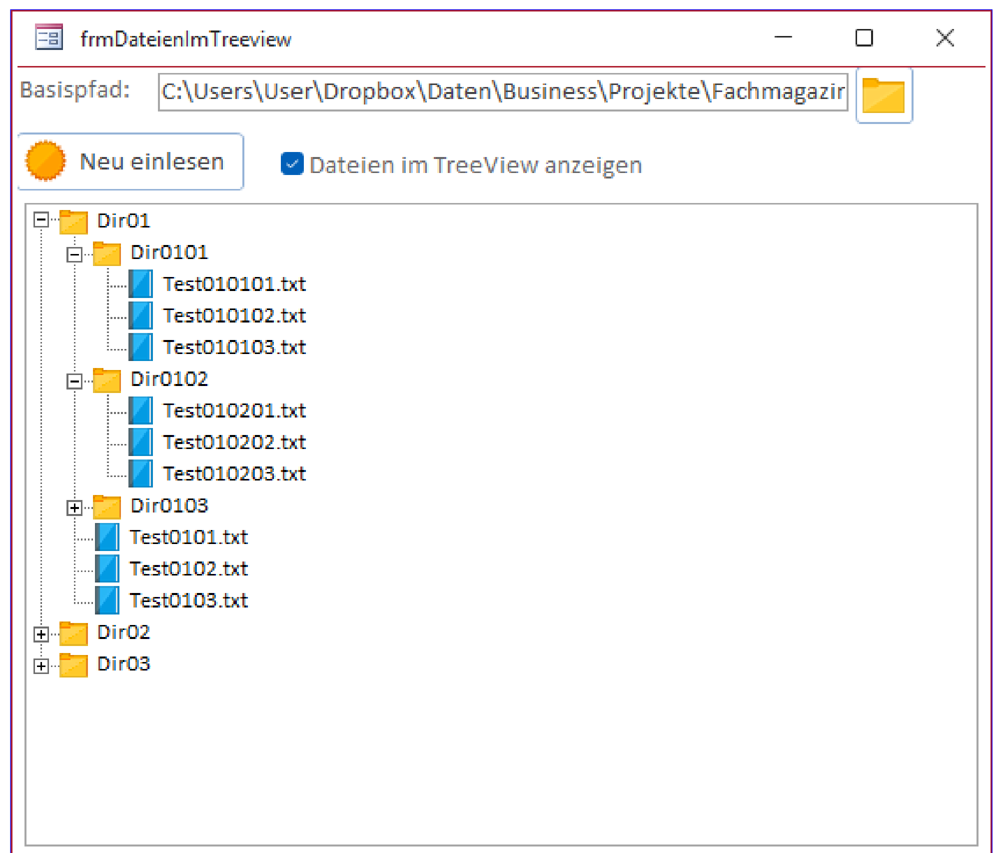


**Bild 1:** Das **TreeView**-Steuerelement mit den Ordnern und Dateien

## Ergänzung im Formular: SelectedItem setzen

Im Vergleich zum Listing aus Teil 1 wurde die Ereignisprozedur **ctlTreeView_MouseDown** um eine zusätzliche Zuweisung ergänzt. Bislang wurde **objCurrentNode** nur dann geleert und das markierte Element abgewählt, wenn kein Node getroffen wurde. Nun wird außerdem in allen anderen Fällen – also wenn tatsächlich ein Node-Element angeklickt wurde – die Eigenschaft **SelectedItem** des **TreeView**-Steuerelements explizit auf dieses Element gesetzt:

```
Set objCurrentNode = ctlTreeView.Object.HitTest(x, y)
If objCurrentNode Is Nothing Then
```

```
    Me.ctlTreeView.Object.SelectedItem = Nothing
Else
    Me.ctlTreeView.Object.SelectedItem = objCur-
rentNode
End If
```

Das stellt sicher, dass beim Aufruf von Kontextmenü-Funktionen, die über **Screen.ActiveControl** auf das aktive Steuerelement zugreifen, das angeklickte Element stets korrekt als markiert gilt.

## Datei einfügen – neuer Kontextmenü-Eintrag im Ordner-Menü

Das Kontextmenü für Ordner-Elemente wurde gegenüber dem in Teil 1 gezeigten Listing um einen weiteren Eintrag ergänzt.

Nach dem Eintrag **Ordner umbenennen** erscheint nun auch **Datei einfügen**, der die Funktion **DateiEinfuegen** ohne Parameter aufruft:

```
With cbr.Controls.Add(msoControlButton)
    .Caption = "Datei einfügen"
    .OnAction = "=DateiEinfuegen()"
    .FaceId = 2512
End With
```

Dieser Eintrag erlaubt es, eine zuvor kopierte oder ausgeschnittene Datei in den Zielordner einzufügen, ohne dass das Datei-Kontextmenü benötigt wird.

```
Private Sub CreateCommandBarFile()
    Dim strKey As String
    Dim cbr As Office.CommandBar
    strKey = objCurrentNode.Key
    On Error Resume Next
    CommandBars("Ordner").Delete
    On Error GoTo 0
    Set cbr = CommandBars.Add("Ordner", msoBarPopup, False, True)
    With cbr.Controls.Add(msoControlButton)
        .Caption = "Datei kopieren"
        .OnAction = "=DateiKopieren('" & strKey & "')"
        .FaceId = 1667
    End With
    With cbr.Controls.Add(msoControlButton)
        .Caption = "Datei ausschneiden"
        .OnAction = "=DateiAusschneiden('" & strKey & "')"
        .FaceId = 1667
    End With
    With cbr.Controls.Add(msoControlButton)
        .Caption = "Datei umbenennen"
        .OnAction = "=DateiUmbenennen()"
        .FaceId = 1667
    End With
    With cbr.Controls.Add(msoControlButton)
        .Caption = "Datei löschen"
        .OnAction = "=DateiLoeschen('" & strKey & "')"
        .FaceId = 1667
    End With
    With cbr.Controls.Add(msoControlButton)
        .Caption = "Datei mit Zielanwendung öffnen"
        .OnAction = "=DateiOeffnen('" & strKey & "')"
        .FaceId = 1667
    End With
    cbr.ShowPopup
End Sub
```

**Listing 1:** Prozedur zum Anzeigen des Kontextmenüs für Dateien

Da **DateiEinfuegen** keinen Key-Parameter erwartet, ermittelt die Funktion das Ziel-Element selbst über **Screen.ActiveControl.Object.SelectedItem**.

## Das Kontextmenü für Dateien

Das Kontextmenü für Datei-Elemente wird in der Prozedur **CreateCommandBarFile** zusammengesetzt. Es enthält die fünf Einträge **Datei kopieren**, **Datei ausschneiden**, **Datei umbenennen**, **Datei löschen** und **Datei mit Zielanwendung öffnen** (siehe Bild 2).

Der Aufbau der Prozedur aus Listing 1 ist identisch mit dem Ordner-Kontextmenü aus Teil 1.

Der Key des angeklickten **Nodes** wird aus **objCurrentNode.Key** gelesen und an die jeweilige Funktion übergeben.

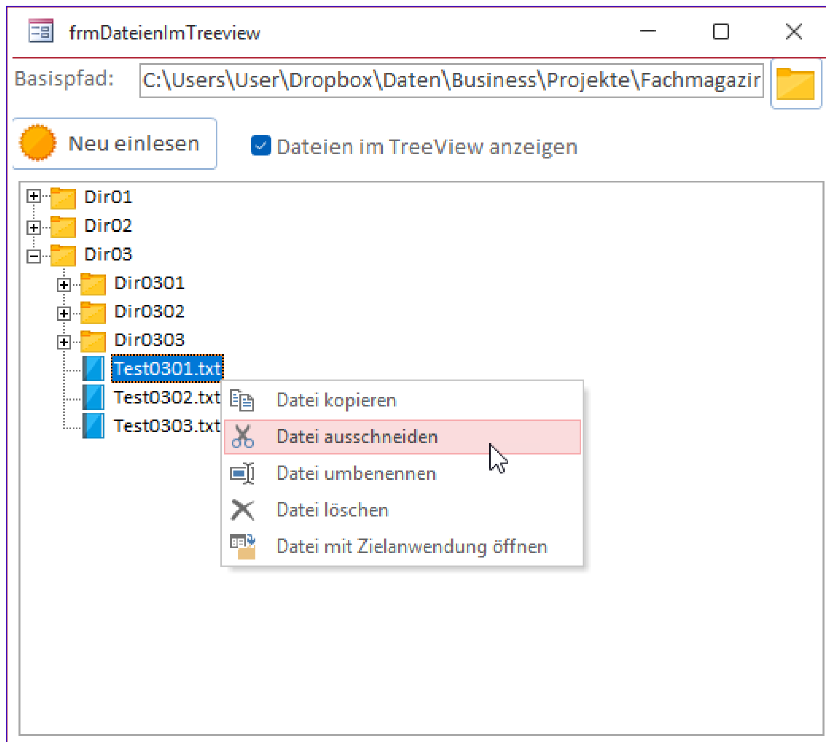


**Bild 2:** Kontextmenü für Dateien

Alle fünf Funktionen liegen im Modul **mdlKontextmenues**.

## Dateien kopieren und ausschneiden

Analog zu den bereits beschriebenen Ordner-Variablen **strKeyOrdnerKopiert** und **strKeyOrdnerAusgeschnitten** gibt es im Modulkopf von **mdlKontextmenues** zwei weitere öffentliche Variablen für Dateien:

```
Public strKeyDateiKopiert As String
Public strKeyDateiAusgeschnitten As String
```

Die zugehörigen Funktionen schreiben den Key in die jeweilige Variable und leeren die andere:

```
Public Function DateiKopieren(strKey As String)
    strKeyDateiKopiert = strKey
    strKeyDateiAusgeschnitten = ""
End Function

Public Function DateiAusschneiden(strKey As String)
    strKeyDateiAusgeschnitten = strKey
    strKeyDateiKopiert = ""
End Function
```

Das Vorgehen ist das gleiche, das wir in Teil 1 bereits für Ordner beschrieben haben.

## Eine Datei umbenennen

Beim Umbenennen einer Datei gibt es einen kleinen Unterschied zu Ordnern. Die Funktion **OrdnerUmbenennen** greift direkt über die globale Variable **objTreeView** auf das Steuerelement zu.

Für Dateien wurde eine andere Lösung gewählt, weil der Umbenennen-Befehl über das Datei-Kontextmenü ohne **Key**-Parameter aufgerufen wird. Die Funktion holt sich das aktive Steuerelement über **Screen.ActiveControl**:

```
Public Function DateiUmbenennen()
    Dim ctlTreeView As Object
    Set ctlTreeView = Screen.ActiveControl
    ctlTreeView.Object.StartLabelEdit
End Function
```

Das bereits aus Teil 1 bekannte Ereignis **ctlTreeView_AfterLabelEdit** im Formularmodul übernimmt dann den neuen Namen sowohl in die Tabelle **tblFiles** als auch im Dateisystem über den Aufruf von **FS_DateiUmbenennen**.

## Eine Datei löschen

Die Funktion **DateiLoeschen** löscht eine Datei aus dem Dateisystem, aus der Tabelle **tblFiles** und aus dem **TreeView**-Steuerelement (Listing 2).

Sie übernimmt den Key des zu löschenden Elements, ermittelt daraus die ID und holt sich das aktive Steuerelement über **Screen.ActiveControl**.

```
Public Function DateiLoeschen(strKey As String)
    Dim db As DAO.Database
    Dim lngID As Long
    Dim ctlTreeView As Object

    Set ctlTreeView = Screen.ActiveControl
    lngID = Mid(strKey, 2)
    Set db = CurrentDb

    If FS_DateiLoeschen(lngID) = True Then
        db.Execute "DELETE FROM tblFiles WHERE FileID = " _
            & lngID, dbFailOnError
        ctlTreeView.Object.Nodes.Remove "i" & lngID
    End If
End Function
```

Nur wenn **FS_DateiLoeschen** den Wert **True** zurückgibt – also die Datei tatsächlich im Dateisystem gelöscht werden konnte – werden auch der Datenbank-Datensatz und der TreeView-Eintrag entfernt. Das vermeidet inkonsistente Zustände.

## Eine Datei öffnen

Um eine Datei in der zugehörigen Standardanwendung zu öffnen, nutzen wir die Funktion **DateiOeffnen**. Sie ermittelt aus dem Key die ID, ruft **GetFilePath** auf und übergibt den vollständigen Pfad an **FollowHyperlink**:

```
Public Function DateiOeffnen(strKey As String) As Long
    Dim strPath As String
    Dim lngID As Long

    lngID = Mid(strKey, 2)
    strPath = GetFilePath(lngID)
    FollowHyperlink strPath
End Function
```

Das Ergebnis ist identisch mit dem Doppelklick-Verhalten, das wir bereits in Teil 1 beschrieben haben. Zusätzlich ist der Befehl nun auch über das Kontextmenü erreichbar.

## Eine Datei einfügen

Die Funktion **DateiEinfuegen** ist die komplexeste der Datei-Kontextmenü-Funktionen, weil sie sowohl den Fall des Einfügens einer kopierten als auch einer ausgeschnittenen Datei behandelt (siehe Listing 2).

Die Funktion prüft zunächst, ob eine Datei zum Kopieren oder zum Ausschneiden markiert ist. Das Ziel-Element – also der Ordner, in den eingefügt werden soll – wird über das aktuell im **TreeView**-Steuerelement selektierte Element bestimmt.

Die Quelle wird über die ID aus **strKeyDateiKopiert** beziehungsweise **strKeyDateiAusgeschnitten** ermittelt.

Beim Kopieren rufen wir **FS_DateiKopieren** auf. Ist das erfolgreich, legen wir in der Tabelle **tblFiles** einen neuen Datensatz an – mit einem **INSERT INTO ... SELECT**-Ausdruck, der Dateiname, Größe und Datum übernimmt, aber die neue **ParentID** verwendet.

Die neue **FileID** ermitteln wir danach mit **SELECT @@IDENTITY** und fügen das Element unter dem Zielordner in das **TreeView**-Steuerelement ein.

Beim Ausschneiden rufen wir **FS_DateiVerschieben** auf. Nach erfolgreichem Verschieben aktualisieren wir die **ParentID** des vorhandenen Datensatzes in **tblFiles** per **UPDATE**.

Wir prüfen über **db.RecordsAffected**, ob genau ein Datensatz geändert wurde.

Ist das der Fall, entfernen wir den Node an seiner alten Position und fügen ihn beim Zielordner neu ein.

## Ordner löschen: FS-Aufruf ergänzt

Im Vergleich zum Listing aus Teil 1 wurde die Funktion **OrdnerLoeschen** um einen Aufruf von **FS_OrdnerLoeschen** ergänzt, der vor dem Löschen der Datenbankdatensätze ausgeführt wird:

# Die DBEngine-Klasse der DAO-Bibliothek

**Wer in Access per VBA auf Datenbanken zugreift, nutzt meist die DAO-Bibliothek – oft ohne zu wissen, dass am Anfang der gesamten Objekthierarchie die »DBEngine« steht. Sie repräsentiert die Datenbank-Engine selbst und bietet Zugriff auf Arbeitsbereiche, Verbindungen und globale Einstellungen. Dieser Beitrag stellt die wichtigsten Eigenschaften und Methoden der DBEngine-Klasse vor und zeigt anhand praktischer Beispiele, wie Sie diese gewinnbringend einsetzen.**

## Die DAO-Objekthierarchie

Die DAO-Bibliothek (**Data Access Objects**) organisiert ihre Objekte in einer festen Hierarchie. An der Wurzel steht die **DBEngine**-Klasse. Sie ist kein Objekt, das Sie explizit instanziieren müssen – VBA stellt es Ihnen über die DAO-Bibliothek automatisch als globale Instanz bereit. Die Hierarchie lautet vereinfacht: **DBEngine** – **Workspace** – **Database** – **TableDef**/**QueryDef**/**Recordset**.

Sie können **DBEngine** direkt ansprechen, ohne vorher ein Objekt zu deklarieren:

```
Debug.Print DBEngine.Version
```

Alternativ können Sie auch eine explizit typisierte Variable verwenden. Das verbessert die IntelliSense-Unterstützung:

```
Dim dbe As DAO.DBEngine
Set dbe = DBEngine
Debug.Print dbe.Version
```

## Wichtige Eigenschaften der DBEngine

Die **DBEngine**-Klasse stellt eine Reihe von Eigenschaften zur Verfügung, über die Sie globale Einstellungen der Jet- bzw. ACE-Engine steuern können.

### Version

Die Eigenschaft **Version** gibt die Versionsnummer der aktuell verwendeten DAO-Bibliothek als Zeichenkette zurück. Das ist nützlich, um zur Laufzeit zu ermitteln, mit welcher Engine die Anwendung arbeitet:

```
Debug.Print DBEngine.Version 'z.B. "16.0"
```

Der zurückgegebene Wert bezieht sich auf die DAO-Version, nicht auf die Access-Version. Für die Bibliothek **Microsoft DAO 3.6 Object Library** liefert sie beispielsweise den Wert **3.6**, für die heute verwendete **Microsoft Office 16.0 Access database engine Object Library** den Wert **16.0**. Damit lässt sich also einfach ermitteln, ob die alte oder die neue DAO-Bibliothek verwendet wird.

### DefaultUser und DefaultPassword

Mit **DefaultUser** und **DefaultPassword** legen Sie fest, welche Anmeldedaten beim Öffnen von Datenbanken ohne explizite Angabe von Benutzername und Kennwort verwendet werden.

Diese Eigenschaften stammen noch aus der älteren Workgroup-Sicherheit von Access und sind in modernen Umgebungen kaum noch relevant, können aber beim Zugriff auf ältere gesicherte Datenbanken nützlich sein:

```
DBEngine.DefaultUser = "Admin"
DBEngine.DefaultPassword = ""
```

Dies funktioniert nur mit der alten Workgroup-Security, die in **.mdb**-Dateien bis Access 2003 zum Einsatz kam, in **.accdb**-Dateien ist dies wirkungslos.

### LoginTimeout

Die Eigenschaft **LoginTimeout** legt fest, wie viele Sekunden die Engine wartet, bevor ein Anmeldeversuch an

einem ODBC-Server als fehlgeschlagen gilt. Der Standardwert beträgt 20 Sekunden.

In der Praxis ist die Wirkung dieser Eigenschaft jedoch begrenzt: Moderne ODBC-Treiber wie der ODBC Driver 17 for SQL Server haben eigene interne Verbindungs- und Retry-Timeouts, die Vorrang vor dem hier gesetzten Wert haben. So kann es vorkommen, dass trotz eines gesetzten Werts von 2 Sekunden erst nach 10 oder mehr Sekunden eine Fehlermeldung erscheint, weil der Treiber seinen eigenen Zyklus durchläuft.

Die Einstellung funktioniert auch nur für ODBC und andere externe Treiber, nicht für den direkten Zugriff über die ACE-Engine.

Es gibt zwar in ADODB ebenfalls eine Eigenschaft, deren Name verheißen lässt, dass man damit den Zeitpunkt eines Verbindungsabbruchs steuern könnte, jedoch funktioniert auch die dortige **ConnectionTimeout**-Eigenschaft nur bei einer sehr beschränkten Menge von Konstellationen zuverlässig. Das ist beispielsweise bei bestimmten falschen Angaben des Servernamens der Fall. In anderen Situationen schalten sich andere Automatismen dazwischen, die sich nicht durch die Einstellung der Zeit bis zum Verbindungsabbruch steuern lassen.

## IniPath

Mit **IniPath** können Sie einen Registry-Pfad angeben, unter dem die Jet/ACE-Engine ihre Konfiguration liest. Normalerweise liest die Engine ihre Einstellungen aus dem Standardpfad in der Windows-Registry.

Über diese Eigenschaft können Sie einen alternativen Schlüssel angeben, um Engine-Parameter wie Cache-Größen oder Sperrmodi individuell zu konfigurieren. Dies ist ein fortgeschrittenes Thema, das hauptsächlich für Mehrbenutzerumgebungen mit speziellen Anforderungen interessant ist.

## SystemDB

Die Eigenschaft **SystemDB** gibt den Pfad zur aktuell verwendeten Arbeitsgruppeninformationsdatei (**.mdw**) an. Diese Eigenschaft ist ein Relikt der älteren Access-Sicherheitsarchitektur. In modernen Anwendungen ohne Workgroup-Sicherheit hat sie keine praktische Bedeutung mehr. Sie können sie lesen, um zu prüfen, welche **.mdw**-Datei aktuell verwendet wird:

```
Debug.Print DBEngine.SystemDB
```

## Workspaces-Auflistung

Die **Workspaces**-Auflistung enthält alle aktuell geöffneten **Workspace**-Objekte.

Standardmäßig enthält sie genau einen Eintrag: den Standardarbeitsbereich **Workspaces(0)**. Über diesen gelangen Sie zur aktuellen Datenbank:

```
Sub DatenAusExternerDB()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim strPfad As String
    strPfad = "C:\Daten\Archiv.accdb"
    Set db = DBEngine.OpenDatabase(strPfad, False, True)
    Set rst = db.OpenRecordset( _
        "SELECT * FROM tblBestellungen", dbOpenSnapshot)
    Do While Not rst.EOF
        Debug.Print rst!BestellNr
        rst.MoveNext
    Loop
    rst.Close
    db.Close
    Set rst = Nothing
    Set db = Nothing
End Sub
```

**Listing 1:** Die Prozedur DatenAusExternerDB

```
Dim ws As DAO.Workspace
Set ws = DBEngine.Workspaces(0)
Debug.Print ws.Name 'Ausgabe: #Default Workspace#
```

## Wichtige Methoden der DBEngine

Neben den Eigenschaften bietet die **DBEngine**-Klasse mehrere Methoden, die direkt auf der Ebene der Engine operieren. Diese stellen wir in den folgenden Abschnitten vor.

## OpenDatabase

Mit **OpenDatabase** öffnen Sie eine Datenbankdatei und erhalten ein **Database**-Objekt zurück. Die Methode hat folgende Signatur:

```
DBEngine.OpenDatabase(Name, [Exclusive], [ReadOnly], 
[Connect])
```

Der Parameter **Name** gibt den Pfad zur Datenbankdatei an. **Exclusive** legt fest, ob die Datenbank exklusiv geöffnet werden soll, **ReadOnly** schränkt den Zugriff auf Lesen ein.

Der optionale Parameter **Connect** ermöglicht die Übergabe einer Verbindungszeichenfolge für ODBC-Quellen.

Ein typisches Beispiel: Sie möchten aus VBA heraus eine externe Access-Datenbank öffnen, um dort Daten zu lesen, ohne diese als verknüpfte Tabelle einbinden zu müssen (siehe Listing 1).

Beachten Sie: Die so geöffnete Datenbank ist von der aktuellen Access-Sitzung unabhängig und muss am Ende explizit mit **Close** geschlossen und das Objekt auf **Nothing** gesetzt werden.

## CreateWorkspace

Mit **CreateWorkspace** können Sie einen neuen Arbeitsbereich anlegen. In der Praxis spielt diese Methode heute vor allem noch für Jet-/ACE-Arbeitsbereiche eine Rolle. Die Methode hat folgende Signatur:

```
DBEngine.CreateWorkspace(Name, UserName, Password, [Use-
Type])
```

Der Parameter **UseType** legt den Typ des Arbeitsbereichs fest. Der Wert **dbUseJet** erstellt einen Jet-/ACE-Arbeitsbereich.

Der frühere Wert **dbUseODBC** war für **ODBCDirect** vorgesehen, das in modernen DAO-Versionen nicht mehr unterstützt wird – die Verwendung führt sogar zu einem Fehler (siehe Bild 1).

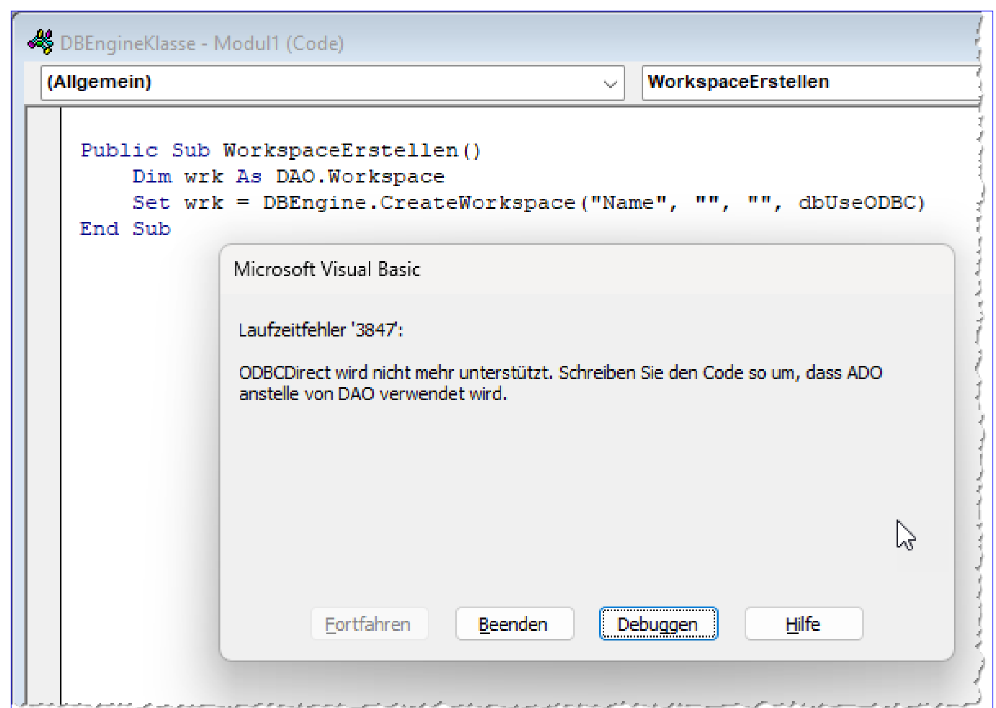


**Bild 1:** Die Verwendung von **dbUseODBC** führt zu einem Fehler

```
Sub TransaktionMitWorkspace()
    Dim ws As DAO.Workspace
    Dim db As DAO.Database
    Set ws = DBEngine.CreateWorkspace("MeinWS", "Admin", "", dbUseJet)
    Set db = ws.OpenDatabase(CurrentDb().Name)
    ws.BeginTrans
    On Error GoTo Fehler
    db.Execute "UPDATE tblArtikel SET Preis = Preis * 1.05 WHERE Kategorie = 'A'", dbFailOnError
    db.Execute "INSERT INTO tblPreishistorie (Datum, Faktor) VALUES (Date(), 1.05)", dbFailOnError
    ws.CommitTrans
    GoTo Ende
Fehler:
    ws.Rollback
    MsgBox "Fehler: Transaktion zurückgerollt."
Ende:
    db.Close
    ws.Close
    Set db = Nothing
    Set ws = Nothing
End Sub
```

**Listing 2:** Die Prozedur **TransaktionMitWorkspace**

Unabhängig davon steht über **DBEngine.Workspaces(0)** immer der Standardarbeitsbereich zur Verfügung, über den sich auch Transaktionen steuern lassen.

## Transaktionen über Workspace steuern

Über **DBEngine.Workspaces(0)** können Sie auf den aktuellen Workspace zugreifen und darüber Transaktionen steuern. Transaktionen über dieses **Workspace**-Objekt sind besonders dann nützlich, wenn Sie sicherstellen möchten, dass mehrere Schreiboperationen entweder vollständig oder gar nicht ausgeführt werden. Im Fehlerfall sorgt **Rollback** dafür, dass keine halbfertigen Änderungen in der Datenbank verbleiben. Ein Beispiel sehen wir in Listing 2.

## CompactDatabase

Die Methode **CompactDatabase** komprimiert eine Datenbankdatei und erstellt dabei eine neue, optimierte Kopie. Da die Methode auf einer geschlossenen Datenbank operiert, kann sie nicht auf die aktuell geöffnete Datenbankdatei angewendet werden. Die Signatur lautet:

```
DBEngine.CompactDatabase(SrcName, DstName, [DstLocale],
[Options], [SrcLocale])
```

Ein praktisches Einsatzszenario ist das Komprimieren einer externen Archivdatenbank, die nicht in der laufenden Access-Sitzung geöffnet ist:

```
Sub ArchivKomprimieren()
    Dim strQuelle As String
    Dim strZiel As String
    strQuelle = "C:\Daten\Archiv.accdb"
    strZiel = "C:\Daten\Archiv_kompakt.accdb"
    On Error GoTo Fehler
    DBEngine.CompactDatabase strQuelle, strZiel
    Kill strQuelle
    Name strZiel As strQuelle
    MsgBox "Archivdatenbank erfolgreich komprimiert."
    Exit Sub
Fehler:
    MsgBox "Fehler beim Komprimieren: " & Err.Description
End Sub
```

# Filtern und Sortieren von Formularen mit Recordset

**Im ersten Teil dieser Beitragsreihe (»Detailformular und Datenblatt mit ADODB-Recordset«, www.access-im-unternehmen.de/1601) haben wir gezeigt, wie man ein Formular per VBA-Code an ein ADODB-Recordset bindet. Dabei sind wir auf ein grundlegendes Problem gestoßen: Die eingebauten Sortier- und Filterfunktionen von Access – sowohl im Ribbon als auch im Dropdown-Menü des Datenblatt-Spaltenkopfes – funktionieren bei ADODB-gebundenen Formularen nicht. In diesem Beitrag zeigen wir, wie wir diese Funktionen mit eigenen Mitteln vollständig nachbauen.**

## Das Problem: Warum funktioniert die eingebaute Filterung nicht?

Wenn Access ein Formular filtert oder sortiert, greift es intern auf die **Datensatzquelle** des Formulars zu – also auf die Tabelle oder Abfrage, die in der gleichnamigen Eigenschaft eingetragen ist.

Bei einem ADODB-gebundenen Formular ist diese Eigenschaft leer, denn wir haben die Datensatzquelle bewusst nicht eingetragen und das Recordset stattdessen per VBA gesetzt.

Access findet also keine SQL-Datensatzquelle und quittiert den Versuch zu filtern mit der Fehlermeldung **Geben Sie einen gültigen Wert ein** oder **Der Datenprovider konnte nicht initialisiert werden**.

Die Lösung ist konsequent: Wir übernehmen die Kontrolle über Filterung und Sortierung vollständig selbst. Anstatt Access die SQL-Abfrage manipulieren zu lassen, bauen wir den SQL-String in unserem VBA-Code zusammen und laden das Recordset neu.

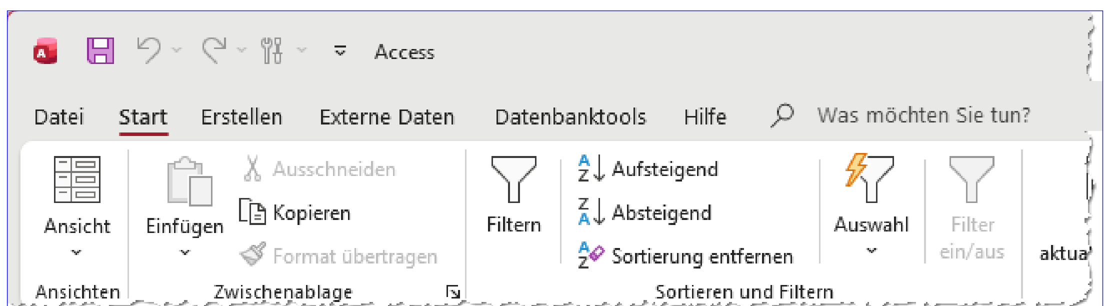


**Bild 1:** Die Anreden werden nicht mehr angezeigt.

Das hat den zusätzlichen Vorteil, dass die Lösung ohne Änderung auch beim Wechsel auf einen SQL Server funktioniert.

## Zwei Wege für Sortierung und Filterung

Im Datenblatt gibt es zwei Möglichkeiten, wie der Benutzer sortieren und filtern kann.

Erstens über das **Ribbon** – die Gruppe **Sortieren und Filtern** im Tab **Start** (siehe Bild 1).

Zweitens über das **Dropdown-Menü im Spaltenkopf** des Datenblatts, das erscheint wenn man mit der Maus über den Spaltenkopf fährt und auf den kleinen Pfeil klickt (siehe Bild 2).

Für das Ribbon verwenden wir eine angepasste Ribbon-Definition in der Tabelle **USysRibbons**, mit der wir die eingebaute Gruppe ausblenden und durch eine eigene

ersetzen. Dort haben wir volle Kontrolle über jeden Button und jeden Callback.

Das Dropdown-Menü im Spaltenkopf hingegen ist ein eingebauter Access-Mechanismus, den wir nicht direkt ersetzen können. Wir können ihn aber über das Ereignis **Bei Filter anwenden** (**Form_ApplyFilter**) abfangen. Access ruft dieses Ereignis auf, bevor es versucht, den Filter anzuwenden – und gibt uns damit die Möglichkeit, die Aktion selbst zu übernehmen.

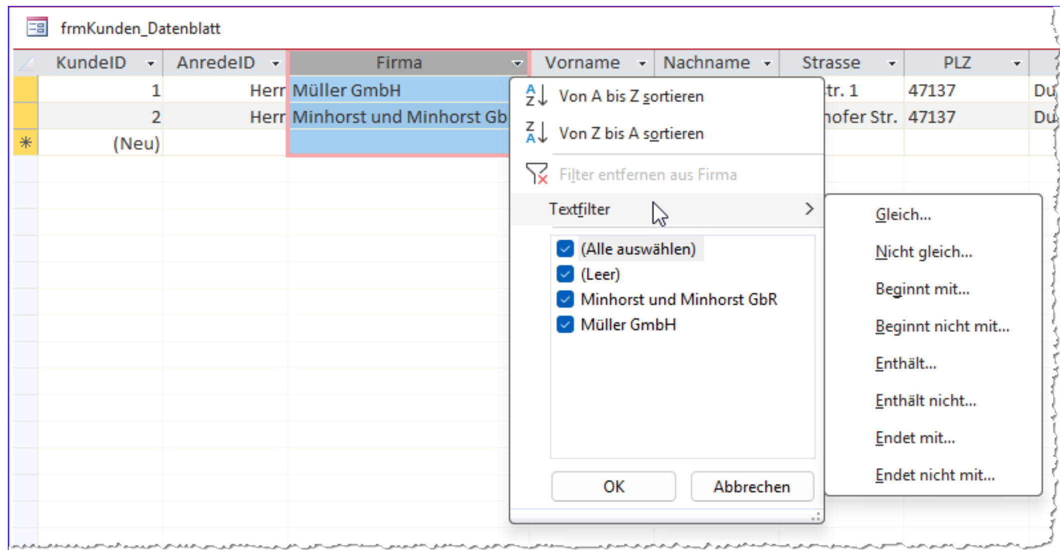


**Bild 2:** Weitere Filtermöglichkeiten in der Datenblattansicht

Eine Information vorneweg: Die Filter unter **Textfilter**, also zum Beispiel **Gleich...**, konnten wir nicht technisch sauber abfangen, daher sind diese in Zusammenhang mit Recordsets in Formularen nicht wie gewohnt nutzbar.

## Den SQL-Zustand im Modul halten

Die Grundidee unserer Lösung ist einfach: Wir halten den aktuellen SQL-String des Formulars in öffentlichen

```
Public Function GetWhereTeil(strSQL As String) As String
    Dim strBisOrderBy As String
    If InStr(strSQL, " WHERE ") > 0 Then
        strBisOrderBy = strSQL
        If InStr(strBisOrderBy, " ORDER BY ") > 0 Then
            strBisOrderBy = Left(strBisOrderBy, _
                InStr(strBisOrderBy, " ORDER BY ") - 1)
        End If
        GetWhereTeil = Mid(strBisOrderBy, _
            InStr(strBisOrderBy, " WHERE ") + 7)
    End If
End Function

Public Function GetOrderByTeil(strSQL As String) As String
    If InStr(strSQL, " ORDER BY ") > 0 Then
        GetOrderByTeil = Mid(strSQL, _
            InStr(strSQL, " ORDER BY ") + 10)
    End If
End Function
```

**Listing 1:** Hilfsfunktionen zum Extrahieren von **WHERE**- und **ORDER BY**-Teilen

```
<group idMso="GroupSortAndFilter" visible="false"/>
<group id="grpSortierenFiltern" label="Sortieren und Filtern"
       insertAfterMso="GroupClipboard">
  <toggleButton idMso="FiltersMenu" size="large"/>
  <separator id="sep0"/>
  <toggleButton idMso="SortUp" size="normal"/>
  <toggleButton idMso="SortDown" size="normal"/>
  <button id="btnSortierungAufheben" label="Sortierung entfernen"
          imageMso="SortRemoveAllSorts" size="normal"
          onAction="OnAction_SortRemove"/>
  <separator id="sep1"/>
  <menu idMso="SortSelectionMenu" size="large"/>
  <separator id="sep2"/>
  <button id="btnFilterAufheben" label="Filter ein/aus"
          imageMso="FilterToggleFilter" size="large"
          onAction="OnAction_FilterRemove"
          getEnabled="getEnabled"/>
</group>
```

**Listing 2:** Ribbon-XML für die eigene Gruppe **Sortieren und Filtern**

Variablen im Modul **mdlADODB** vor. So haben sowohl das Formularmodul als auch die Ribbon-Callbacks jederzeit Zugriff auf den aktuellen Stand:

```
'SQL-Zustand des aktiven Formulars
Public strBaseSQL As String
Public strLastSQL As String
Public strPendingSQL As String
```

**strBaseSQL** enthält den Basis-SQL-String ohne WHERE und **ORDER BY** – also zum Beispiel **"SELECT * FROM tblKunden"**. Dieser wird einmalig in **Form_Open** gesetzt und ändert sich nie. **strLastSQL** enthält den zuletzt ausgeführten SQL-String inklusive aller aktiven Filter und Sortierungen. **strPendingSQL** ist ein Zwischenspeicher, den wir im Timer-Trick benötigen – dazu gleich mehr.

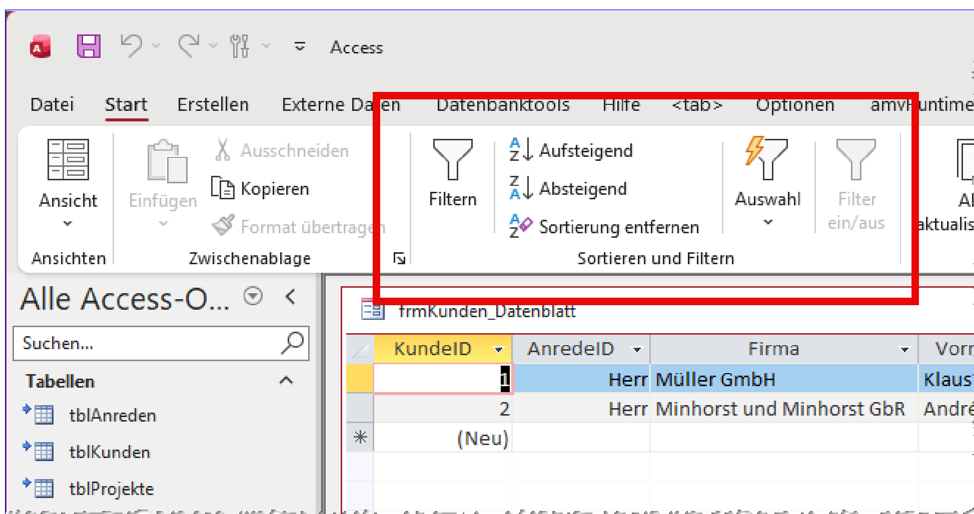


**Bild 3:** Neuer **Sortieren und Filtern**-Bereich im Ribbon

## Hilfsfunktionen für SQL-String-Manipulation

Um **WHERE**- und **ORDER BY**-Teile aus einem SQL-String zu extrahieren, haben wir zwei Hilfsfunktionen in **mdlADODB** angelegt.

Diese werden sowohl von den Ribbon-Callbacks als auch vom Formularmodul verwendet (siehe Listing 1).

```
Sub OnAction_Toggle(control As IRibbonControl, ByRef pressed As Boolean, ByRef CancelDefault)
    CancelDefault = True
    Dim strRichtung As String
    Select Case control.Id
        Case "SortUp"
            strRichtung = "ASC"
        Case "SortDown"
            strRichtung = "DESC"
    End Select
    SortierungAnwenden Screen.ActiveForm, Screen.ActiveControl.Name, strRichtung
End Sub
```

**Listing 3:** Callback **OnAction_Toggle** für Aufsteigend und Absteigend

**GetWhereTeil** gibt den Inhalt der WHERE-Klausel zurück – also den Teil nach dem Schlüsselwort **WHERE** und vor einem eventuellen **ORDER BY**. **GetOrderByTeil** gibt entsprechend den Teil nach **ORDER BY** zurück. Beide Funktionen geben einen leeren String zurück, wenn der jeweilige Teil nicht vorhanden ist.

## Sortierung über das Ribbon

Für die Sortierung über das Ribbon überschreiben wir die eingebauten Ribbon-Commands **SortUp** und **SortDown** mit eigenen Callbacks. In der Ribbon-XML-Definition der Tabelle **USysRibbons** blenden wir zunächst die eingebaute Gruppe **GroupSortAndFilter** aus und fügen eine eigene Gruppe ein (siehe Listing 2).

Das Ergebnis sehen wir in Bild 3.

Die Callbacks **SortUp** und **SortDown** überschreiben wir mit dem **commands**-Element der Ribbon-XML, das wir vor der **ribbon**-Sektion einfügen:

```
<commands>
  <command idMso="SortUp" onAction="OnAction_Toggle"/>
  <command idMso="SortDown" onAction="OnAction_Toggle"/>
  ... weitere Filter-Commands ...
</commands>
```

Der Callback **OnAction_Toggle** im Modul **mdlRibbons** wird aufgerufen wenn der Benutzer auf **Aufsteigend** oder **Absteigend** klickt. Er setzt **CancelDefault = True** um die eingebaute Aktion zu unterdrücken und ruft dann **SortierungAnwenden** auf (siehe Listing 3).

**Screen.ActiveControl.Name** liefert den Namen des Feldes, in dessen Spalte der Benutzer geklickt hat. Dieser wird zusammen mit der Sortierrichtung an **SortierungAnwenden** übergeben.

## Die Funktion SortierungAnwenden

**SortierungAnwenden** in **mdlADODB** ist die zentrale Funktion für alle Sortieroperationen. Sie baut aus dem be-

```
Public Sub SortierungAnwenden(frm As Form, strFeld As String, strRichtung As String)
   On Error Resume Next
    strFeld = frm.Controls(strFeld).ControlSource
    On Error GoTo 0
    strLastSQL = SQLMitSortierung(IIf(Len(strLastSQL) > 0, strLastSQL, strBaseSQL), strFeld, strRichtung)
    Set frm.Recordset = GetRecordset(strLastSQL)
    If Not objRibbon_Main Is Nothing Then
        objRibbon_Main.Invalidate
    End If
End Sub
```

**Listing 4:** Die Prozedur **SortierungAnwenden**

```
Public Function SQLMitSortierung(strSQLQuelle As String, strFeld As String, strRichtung As String) As String
    Dim strWhere As String
    strWhere = GetWhereTeil(strSQLQuelle)
    SQLMitSortierung = strBaseSQL
    If Len(strWhere) > 0 Then
        SQLMitSortierung = SQLMitSortierung & " WHERE " & strWhere
    End If
    SQLMitSortierung = SQLMitSortierung & " ORDER BY " & strFeld & " " & strRichtung
End Function
```

**Listing 5:** Die Funktion **SQLMitSortierung**

stehenden SQL-String einen neuen mit der gewünschten Sortierung zusammen und lädt das Recordset neu. Dabei bleibt ein eventuell aktiver Filter erhalten (siehe Listing 4).

Achtung: Bei Nachschlagefeldern wird hier nach dem Wert der gebundenen Spalte sortiert, nicht nach dem angezeigten Wert. Sie ruft **SQLMitSortierung** auf, die den neuen SQL-String zusammenstellt. Diese Hilfsfunktion extrahiert den **WHERE**-Teil aus dem bisherigen SQL und hängt die neue Sortierung an (siehe Listing 5).

Wichtig: Wir verwenden immer nur die zuletzt gewählte Sortierung – eine neue Sortierung ersetzt die vorherige komplett.

Eine Mehrfachsortierung über mehrere Felder ist über das Ribbon nicht sinnvoll abbildbar, da der Benutzer keine Rückmeldung erhält, welche Felder gerade sortiert sind. Nach dem Setzen des neuen Recordsets rufen wir **objRibbon_Main.Invalidate** auf. Das zwingt Access, alle Ribbon-Callbacks neu auszuwerten – insbesondere **getEnabled** für die Schaltfläche **Filter ein/aus**, die nur aktiv sein soll wenn ein Filter vorhanden ist.

## Sortierung aufheben

Die Schaltfläche **Sortierung entfernen** ruft den Callback **OnAction_SortRemove** auf. Dieser prüft ob überhaupt eine Sortierung aktiv ist und baut dann einen neuen SQL-String ohne ORDER BY zusammen – einen eventuell aktiven Filter behält er dabei (siehe Listing 6).

## Filterung über das Auswahl-Menü im Ribbon

Für die Filterung nutzen wir das eingebaute Ribbon-Menü **SortSelectionMenu** (Auswahl). Dieses zeigt automatisch nur die für den aktuellen Feldtyp passenden Filteroptionen an – bei Textfeldern etwa **Enthält** und **Beginnt mit**, bei Datumsfeldern **Vor** und **Nach**«. Die einzelnen Commands dieses Menüs überschreiben wir ebenfalls über das **commands**-Element mit unserem Callback **OnAction**.

```
Sub OnAction_SortRemove(control As IRibbonControl)
    If Len(GetOrderByTeil(strLastSQL)) = 0 Then Exit Sub
    strLastSQL = strBaseSQL & _
        IIf(Len(GetWhereTeil(strLastSQL)) > 0, " WHERE " & GetWhereTeil(strLastSQL), "")
    Set Screen.ActiveForm.Recordset = GetRecordset(strLastSQL)
    If Not objRibbon_Main Is Nothing Then
        objRibbon_Main.Invalidate
    End If
End Sub
```

**Listing 6:** Callback **OnAction_SortRemove**

# Filterformular für alle Fälle: Einbau und Bedienung

**Access bietet von Haus aus komfortable Möglichkeiten zum Filtern von Datensätzen – über das Ribbon, über das Dropdown-Menü im Spaltenkopf des Datenblatts oder über eigene Filtereigenschaften des Formulars. Diese Lösungen stoßen jedoch schnell an ihre Grenzen, sobald man komplexere Filterbedingungen kombinieren möchte oder dem Benutzer eine moderne, übersichtliche Filteroberfläche bieten will. Wir stellen in dieser zweiteiligen Beitragsreihe ein universell einsetzbares Filterformular vor, das sich mit wenigen Handgriffen in jedes bestehende Formular einbauen lässt. Im ersten Teil zeigen wir, wie das Filterformular eingebaut und bedient wird. Der zweite Teil beleuchtet die Programmierung hinter den Kulissen.**

## Der Ausgangspunkt: Ein Formular mit vielen Datensätzen

Bild 1 zeigt das Ausgangsformular **frmDatenMitFilter** mit einer Kundentabelle – 100 Datensätze, dargestellt als Datenblatt in einem Unterformular. In der Kopfzeile des Formulars befindet sich die Schaltfläche **Filtern**, die das Filterformular öffnet. Der Benutzer sieht zunächst alle Datensätze und kann über diese Schaltfläche jederzeit einen Filter setzen oder anpassen.

**Bild 1:** Das Ausgangsformular mit 100 Kundendatensätzen und der Schaltfläche zum Öffnen des Filterformulars

Mit einem Klick auf **Filtern** öffnet sich das Filterformular **amvFilter** als schwebendes Popup-Fenster. Das Ausgangsformular bleibt dabei vollständig bedienbar im Hintergrund – der Benutzer kann das Filterformular verschieben, um die Datensätze im Hintergrund zu sehen.

## Das Filterformular beim Öffnen

Bild 2 zeigt das frisch geöffnete Filterformular. Es startet stets leer und kompakt: Eine einzige Filterzeile ist sichtbar, bestehend aus einem Kombinationsfeld für die Feldauswahl und einem noch deaktivierten Kombinationsfeld für die Bedingungsauswahl. Die Schaltflächen **Zurücksetzen** und **Filter anwenden** sind zunächst deaktiviert – sie werden erst aktiv wenn mindestens eine Filterbedingung ausgefüllt ist.

**Bild 2:** Das Filterformular startet leer – Schaltflächen sind noch deaktiviert

Das Filterformular passt seine Höhe stets automatisch an die Anzahl der aktiven Zeilen an. Es startet also bewusst klein und nimmt nur den Platz ein, den es gerade benötigt.

Mit jeder neuen Eingabe wächst es um eine Zeile, beim Löschen einer Bedingung schrumpft es wieder.

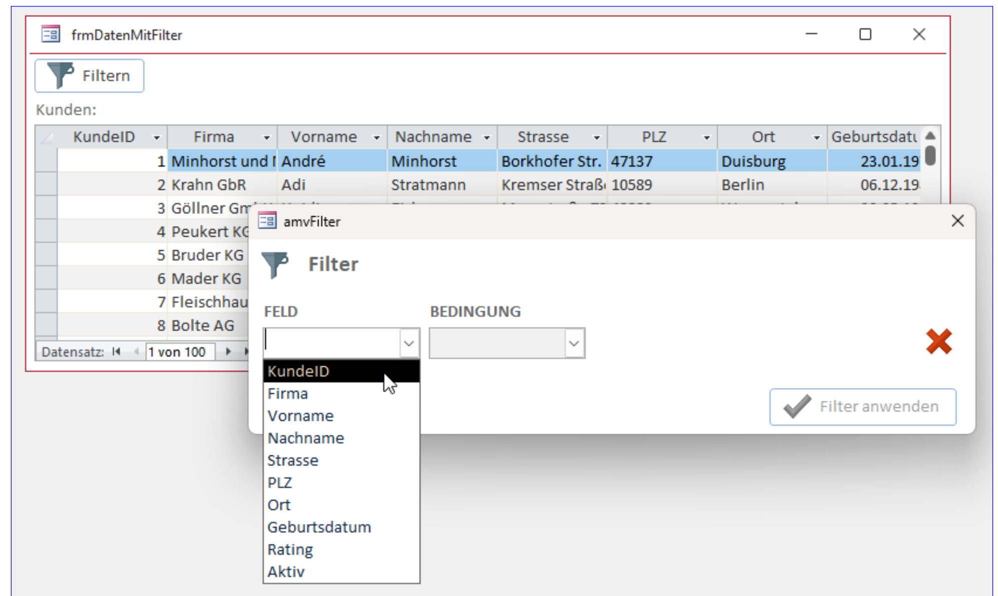

**Bild 3:** Die Feldauswahlliste zeigt automatisch alle Felder des Recordsets an

### Schritt 1: Ein Feld auswählen

Der erste Schritt beim Einrichten eines Filters ist die Auswahl des Feldes, das gefiltert werden soll.

Bild 3 zeigt die aufgeklappte Feldauswahlliste: Das Filterformular liest beim Öffnen automatisch alle Felder des Recordsets aus dem aufrufenden Formular ein, und zeigt sie hier an – in diesem Fall die Felder der Kundentabelle: **KundeID**, **Firma**, **Vorname**, **Nachname**, **Strasse**, **PLZ**, **Ort**, **Geburtsdatum**, **Rating** und **Aktiv**. Es ist keine manuelle Konfiguration der Felder notwendig.

Sobald der Benutzer ein Feld auswählt, passieren zwei Dinge gleichzeitig: Das Bedingungsfeld wird aktiviert und mit den zu diesem Feldtyp passenden Vergleichsoperatoren befüllt. Darunter erscheint automatisch eine zweite leere Filterzeile.

Das Filterformular erkennt dabei selbstständig ob es sich um ein Text-, Zahlen-, Datums- oder Ja/Nein-Feld handelt und bietet nur die jeweils sinnvollen Operatoren an.

### Schritt 2: Die Bedingung auswählen

Bild 4 zeigt die Bedingungsauswahl nachdem das Feld **KundeID** gewählt wurde. Da es sich um ein Zahlenfeld handelt, stehen die Operatoren **Gleich**, **Nicht gleich**, **Größer als**, **Kleiner als**, **Ist zwischen**, **Ist leer** und **Ist nicht leer** zur Verfügung.

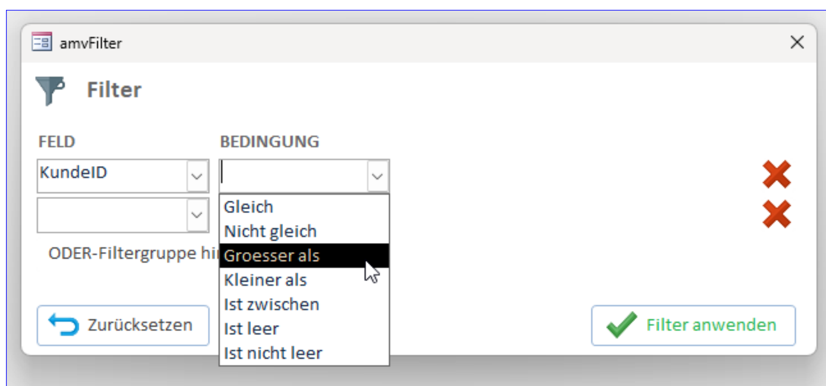

**Bild 4:** Die Bedingungsauswahl zeigt nur die für den Feldtyp passenden Operatoren

Für Textfelder wären es andere Operatoren wie **Enthält** oder **Beginnt mit** – der Benutzer sieht immer nur die Optionen die für den gewählten Feldtyp tatsächlich sinnvoll sind.

Wird eine Bedingung ausgewählt, die einen Vergleichswert erfordert, erscheint rechts neben der Bedingungsauswahl

automatisch ein Eingabefeld für den Wert. Bei Bedingungen wie **Ist leer** oder **Ist nicht leer** erscheint dagegen kein Eingabefeld – der Operator allein definiert bereits den vollständigen Filter.

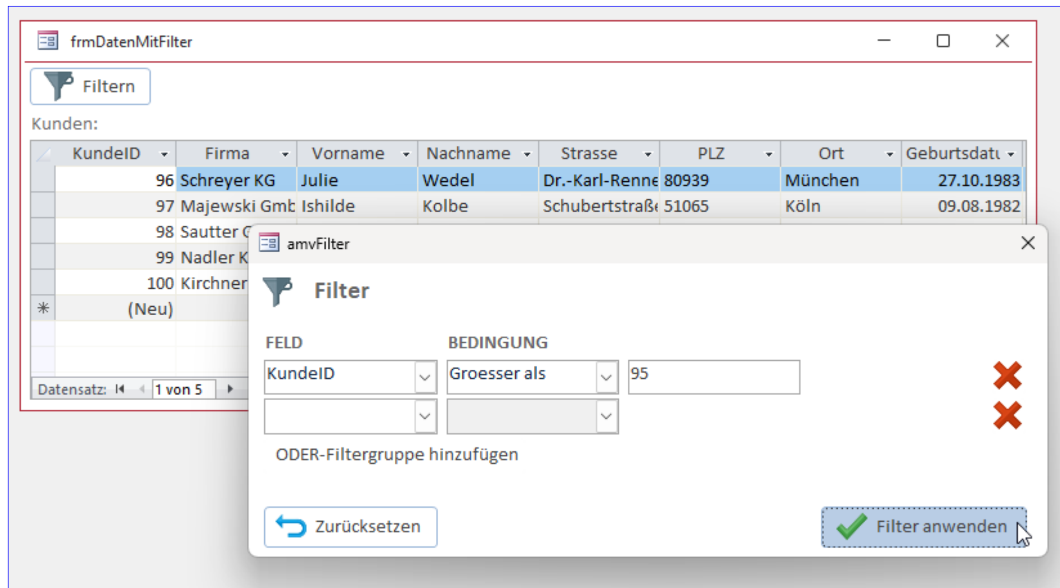


**Bild 5:** Der erste Filter ist gesetzt – von 100 Datensätzen bleiben 5 übrig

Das Formular validiert Tastatureingaben in Echtzeit: In Eingabefelder für Zahlenwerte können nur Ziffern, Komma, Punkt und Minuszeichen eingegeben werden – alle anderen Zeichen werden stillschweigend abgewiesen.

## Ersten Filter anwenden

Einen ersten Filter sehen wir in Bild 5: Hier haben wir das Feld **KundeID** ausgewählt, als Bedingung **Größer als** gesetzt und den Wert **95** eingestellt.

Darunter ist bereits eine zweite leere Zeile sichtbar, bereit für eine weitere Bedingung.

Die Schaltfläche **ODER-Filtergruppe hinzufügen** ist erschienen, und **Filter anwenden** ist nun aktiv. Ein Klick darauf überträgt den Filter sofort an das aufrufende Formular – im Hintergrund sieht man bereits die gefilterten Datensätze: von 100 Kunden bleiben 5 übrig, nämlich alle mit **KundeID** größer als 95.

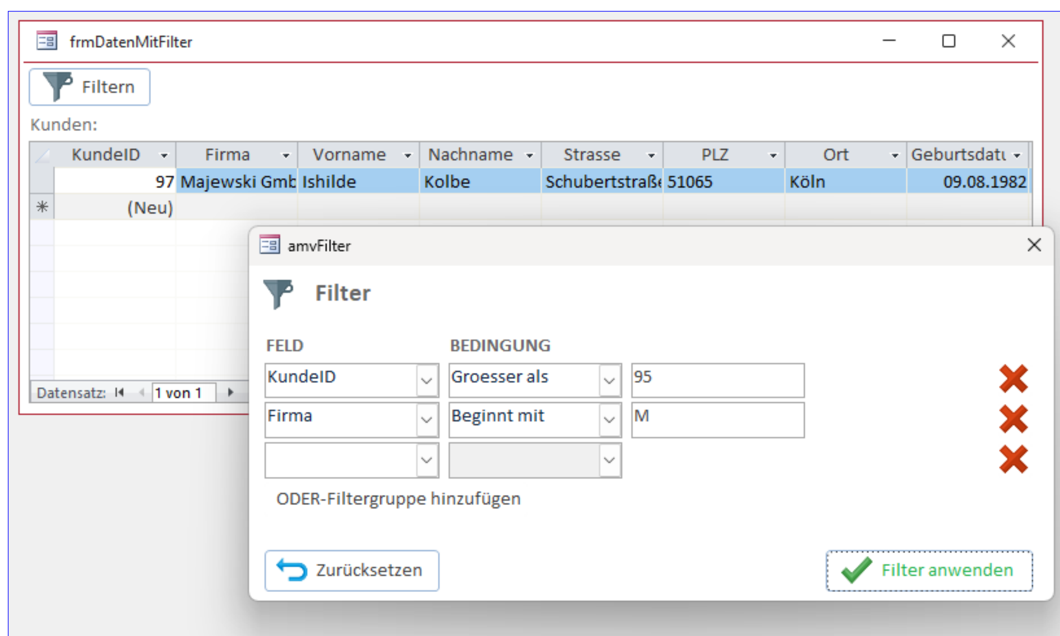


**Bild 6:** Zwei UND-verknüpfte Bedingungen – nur noch ein Datensatz erfüllt beide Kriterien

Das Filterformular bleibt nach dem Anwenden geöffnet. Der Benutzer kann es jederzeit anpassen: weitere Bedin-

# Ein Filterformular für alle Fälle: Die Programmierung

**Im ersten Teil dieser Beitragsreihe haben wir gezeigt, wie das Filterformular in eine eigene Datenbank eingebaut und bedient wird. In diesem zweiten Teil werfen wir einen Blick hinter die Kulissen: Wie ist das Filterformular aufgebaut, wie erkennt es den Feldtyp aus dem Recordset, wie baut es den SQL-WHERE-String zusammen, wie funktioniert die dynamische Höhenanpassung – und welche technischen Besonderheiten mussten gelöst werden. Einige dieser Lösungen sind nicht auf den ersten Blick offensichtlich, liefern aber wertvolle Denkanstöße für eigene Projekte.**

## Architektur: Zwei Bausteine, eine Schnittstelle

Das Filterformular besteht aus zwei Bausteinen: dem Modul **mdlFilter** mit allen Hilfsfunktionen und dem Formularmodul **Form_frmFilter** mit der gesamten Steuerlogik. Die Trennung ist bewusst gewählt: **mdlFilter** enthält ausschließlich Funktionen ohne Bindung an ein konkretes Formular – Feldtyperkennung, SQL-Aufbau, Datumsparsen, Wertevalidierung.

Das Formularmodul nutzt diese Funktionen und kümmert sich um alles was mit der Benutzeroberfläche zu tun hat: Steuerelemente ein- und ausblenden, Positionen setzen, Ereignisse verarbeiten.

Diese klare Aufgabentrennung hat einen praktischen Vorteil: **mdlFilter** kann unverändert wiederverwendet werden wenn das Formular künftig angepasst oder durch eine andere Darstellung ersetzt wird.

Die Kommunikation zwischen dem Filterformular und dem aufrufenden Formular läuft über zwei öffentliche Eigenschaften und eine öffentliche Prozedur. Das aufrufende Formular übergibt beim Öffnen über **CallingForm** eine Referenz auf sich selbst und über **RecordsetForm** das zu filternde Recordset.

Das Filterformular ruft später **FilterAnwenden** im aufrufenden Formular auf und übergibt den fertigen **WHERE**-String als Parameter. Diese drei Elemente bilden die vollständige Schnittstelle – mehr ist für den Einbau nicht notwendig. Das folgende Listing zeigt die beiden Property-Prozeduren:

```
Public Property Set CallingForm(frm As Form)
    Set m_frmAufrufend = frm
End Property

Public Property Set RecordsetForm(rst As Object)
    Set m_rst = rst
    m_bolDAO = (TypeName(rst) = "Recordset") _
        Or (TypeName(rst) = "Recordset2")
    LadeFelder
End Property
```

Der Parameter **RecordsetForm** ist als **Object** typisiert statt als konkreter Recordset-Typ.

Das hat zwei Vorteile: Erstens funktioniert die Übergabe sowohl mit einem DAO-Recordset als auch mit einem ADODB-Recordset, ohne dass der Aufrufer sich um den Typ kümmern muss.

Zweitens ist das Filterformular damit für künftige Erweiterungen vorbereitet. Das Flag **m_bolDAO** wird über **TypeName** gesetzt: Ein DAO-Recordset trägt den Typnamen **Recordset** oder **Recordset2** – je nach Access-Version und Datenbanktyp – ein ADODB-Recordset dagegen nicht.

Dieses Flag ist entscheidend für die Feldtyperkennung, wie wir im nächsten Abschnitt sehen werden.

Sobald **RecordsetForm** gesetzt wird, ruft die Property-Prozedur automatisch **LadeFelder** auf. Diese Prozedur liest alle Felder des Recordsets aus, bestimmt für jedes den Feldtyp und befüllt die **RowSource** aller 25 **cboFeld**-Kombinationsfelder mit den Feldnamen.

Das bedeutet: Der Benutzer sieht im Feldauswahl-Dropdown sofort nach dem Öffnen alle verfügbaren Felder – ohne dass eine manuelle Konfiguration der Felder notwendig wäre. Das Filterformular passt sich damit automatisch an jedes Formular an, dem es zugewiesen wird.

### Das Steuerelemente-Raster: 175 Elemente auf einem Haufen

Das Filterformular unterstützt bis zu fünf **ODER**-Gruppen mit je fünf Filterzeilen – also 25 Zeilen insgesamt.

Jede Zeile besteht aus sieben Steuerelementen die je nach Feldtyp und gewählter Bedingung wechselweise ein- und ausgeblendet werden.

Die Steuerelemente pro Zeile im Überblick: **cboFeld_g_z** für die Feldauswahl, **cboBedingung_g_z** für den Vergleichsoperator, **cboDatumstyp_g_z** für den Datumstyp bei Datumsfeldern, **txtWert_g_z** für den Vergleichswert bei Text- und Zahlenfeldern, **txtDatum_g_z** für konkrete Datumseingaben, **txtDatumA_g_z** für den Bis-Wert bei Datum-Zwischen-Bedingungen, **txtWert2_g_z** für den Bis-Wert bei Zahlen-Zwischen-Bedingungen und **cmdLoeschen_g_z** zum Entfernen der Zeile.

Das Namensschema ist einheitlich: **g** steht für die Gruppennummer (1 bis 5), **z** für die Zeilennummer (1 bis 5).

Alle Ereignisprozeduren im Formularmodul folgen demselben Schema und delegieren an generische Hilfsprozeduren: **cboFeld_1_1_AfterUpdate** ruft **FeldGewaehlt 1, 1** auf, **cmdLoeschen_2_3_Click** ruft **ZeileLeeren 2, 3** auf.

Das macht den Code überschaubar trotz der großen Zahl von Ereignisprozeduren. Für alle 25 Zeilen entstehen so je Ereignistyp 25 identisch aufgebaute Einzeiler. Die eigentliche Logik steckt in den generischen Hilfsprozeduren die **g** und **z** als Parameter erhalten und alle Steuerelementnamen daraus zusammensetzen – also etwa **Me("cboFeld_" & g & "_" & z)**.

Alle 175 Einzelsteuerelemente liegen im Formular an der Y-Position 0, also übereinander gestapelt. Nur die jeweils aktiven Zeilen werden sichtbar gemacht und an ihre richtige vertikale Position verschoben.

Unsichtbare Zeilen bleiben bei Y=0 und nehmen keinen Platz ein. Diese Technik ist der zentrale Trick der es überhaupt ermöglicht, die Höhe des Formulars dynamisch zu steuern: Access richtet die Höhe des Detailbereichs nicht nach den unsichtbaren Steuerelementen aus, sondern nach dem Wert der Eigenschaft **Section(acDetail).Height** – und den setzen wir per Code.

Würden die unsichtbaren Steuerelemente an ihren ursprünglichen Positionen verbleiben, würde Access den Detailbereich nicht kleiner darstellen als notwendig um alle Elemente – auch die unsichtbaren – zu umschließen. Das Formular wäre dann stets so hoch wie alle 25 Zeilen zusammen.

### Feldtypen erkennen: Die DAO/ADODB-Kollision

Sobald das Recordset übergeben wurde, liest **LadeFelder** alle Felder aus und bestimmt für jedes den Feldtyp. Das Filterformular unterscheidet vier eigene Feldtypen, die als öffentliche Konstanten in **mdlFilter** definiert sind: **cFeldtypText** (1), **cFeldtypZahl** (2), **cFeldtypDatum** (3) und **cFeldtypJaNein** (4).

Diese vier Typen steuern, welche Vergleichsoperatoren angeboten werden, welches Eingabefeld erscheint und wie der Wert später im SQL-Ausdruck gequotet wird. Die Zuordnung vom nativen Typwert des Recordset-Feldes auf einen dieser vier Typen übernimmt die Funktion **Get-**

**Feldtyp** in **mdlFilter**. Sie erhält neben dem numerischen Typwert auch das Flag **bolDAO** (siehe Listing 1).

Warum ist die strikte Trennung zwischen DAO und ADODB notwendig? Der Grund ist eine handfeste Kollision der Typnummern: DAO verwendet für den Typ **dbDouble** (Gleitkommazahl) den Wert 7, ADODB verwendet denselben Wert 7 für **adDate** (Datum).

Eine einheitliche Behandlung beider Bibliotheken mit derselben Fallunterscheidung ist daher nicht möglich. Würde man in einem DAO-Recordset den Wert 7 als Datum interpretieren, würden alle Double-Felder fälschlicherweise als Datumsfelder behandelt – mit entsprechend falschen Vergleichsoperatoren und Datumsformatierung im SQL-Ausdruck als Folge.

Die Lösung ist das Flag **m_bolDAO** das beim Setzen von **RecordsetForm** automatisch über **TypeName** bestimmt wird. Im DAO-Zweig von **GetFeldtyp** wird der Wert 7 korrekt als Zahl (**dbDouble**) behandelt, während DAO-Datumsfelder den eindeutigen Wert 8 (**dbDate**) tragen. Im ADODB-Zweig wird 7 dagegen korrekt als Datum (**adDate**) behandelt, während ADODB-Boolean-Felder den Wert 11 (**adBoolean**) haben.

Das **bolDAO**-Flag ist damit nicht nur eine technische Notwendigkeit für den aktuellen Stand des Filterformulars – es ist gleichzeitig die Vorbereitung für die geplante Erweiterung auf ADODB-gebundene Formulare.

Wird das Filterformular künftig mit einem ADODB-Recordset aufgerufen, wird **m_bolDAO** automatisch auf

```
Public Function GetFeldtyp(lngTyp As Long, bolDAO As Boolean) As Long
    If bolDAO Then
        Select Case lngTyp
            Case 2, 3, 4, 5, 6, 7, 15, 16, 17, 18, 19, 20
                GetFeldtyp = cFeldtypZahl
            Case 8
                GetFeldtyp = cFeldtypDatum  'dbDate = 8
            Case 1
                GetFeldtyp = cFeldtypJaNein 'dbBoolean = 1
            Case Else
                GetFeldtyp = cFeldtypText
        End Select
    Else
        Select Case lngTyp
            Case 2, 3, 4, 5, 6, 14, 15, 16, 17, 18, 19, 20, 21, 131
                GetFeldtyp = cFeldtypZahl
            Case 7, 133, 134, 135
                GetFeldtyp = cFeldtypDatum  'adDate = 7
            Case 11
                GetFeldtyp = cFeldtypJaNein 'adBoolean = 11
            Case Else
                GetFeldtyp = cFeldtypText
        End Select
    End If
End Function
```

**Listing 1: GetFeldtyp** unterscheidet DAO- und ADODB-Typnummern

**False** gesetzt und der ADODB-Zweig kommt zum Einsatz, ohne dass an **GetFeldtyp** etwas geändert werden muss.

Der Aufruf in **LadeFelder** ist entsprechend einfach gehalten:

```
For i = 0 To m_rst.Fields.Count - 1
    m_strFeldnamen(i) = m_rst.Fields(i).Name
    m_lngFeldtypen(i) = GetFeldtyp( _
        m_rst.Fields(i).Type, m_bolDAO)
Next i
```

Die ermittelten Feldnamen und Feldtypen werden in den Modulvariablen **m_strFeldnamen** und **m_lngFeldtypen** gespeichert.

Auf diese Arrays greift das Formularmodul später zu, wenn der Benutzer ein Feld auswählt. Der Feldtyp steckt dann bereits im Array und muss nicht erneut aus dem Recordset ausgelesen werden.

Das ist effizienter und vermeidet Probleme, wenn das Recordset zwischenzeitlich geschlossen oder verändert wurde.

## Bedingungen und Wertfelder dynamisch einblenden

Sobald der Benutzer ein Feld auswählt, reagiert **FeldGewaehlt** auf das **AfterUpdate**-Ereignis des jeweiligen **cboFeld**-Kombinationsfelds. Die Prozedur ermittelt den Feldtyp des gewählten Felds aus dem gespeicherten Array und befüllt **cboBedingung** mit den passenden Operatoren über **GetBedingungRowsource** (siehe Listing 2).

Gleichzeitig mit dem Aktivieren des Bedingungsfelds blendet **FeldGewaehlt** alle Wertfelder der Zeile aus und setzt den Wert von **cboBedingung** auf **Null**. Ein Feldwechsel macht die bisherige Bedingung ungültig – ein Textoperator wie **Enthält** ergibt für ein Zahlenfeld keinen Sinn. Außerdem setzt **FeldGewaehlt** die **Left**-Positionen aller Wertfelder auf ihre Ursprungswerte zurück.

Das ist notwendig weil bei Datumsfeldern mit der Bedingung **Ist zwischen** die Positionen der Eingabefelder per Code verändert werden – dazu gleich mehr.

Schließlich blendet **FeldGewaehlt** automatisch eine neue leere Zeile unterhalb ein, sofern die aktuelle Zeile die letzte aktive in der Gruppe ist und noch nicht fünf Zeilen belegt sind.

```
Public Function GetBedingungRowsource(lngFeldtyp As Long) As String
    Select Case lngFeldtyp
        Case cFeldtypText
            GetBedingungRowsource = _
                "Gleich;Nicht gleich;Enthaelt;Enthaelt nicht;Beginnt mit;Endet mit;Ist leer;Ist nicht leer"
        Case cFeldtypZahl
            GetBedingungRowsource = _
                "Gleich;Nicht gleich;Groesser als;Kleiner als;Ist zwischen;Ist leer;Ist nicht leer"
        Case cFeldtypDatum
            GetBedingungRowsource = _
                "Gleich;Ist vor;Ist nach;Ist zwischen;Ist leer;Ist nicht leer"
        Case cFeldtypJaNein
            GetBedingungRowsource = "Wahr;Falsch"
    End Select
End Function
```

**Listing 2: GetBedingungRowsource** liefert die passenden Operatoren je Feldtyp

Sobald der Benutzer eine Bedingung auswählt, entscheidet **BedingungGewaehlt** welche Eingabefelder eingeblendet werden. Bei Bedingungen wie **Ist leer**, **Ist nicht leer**, **Wahr** oder **Falsch** erscheint kein Eingabefeld – der Operator definiert bereits vollständig den Filterausdruck. Für Textfelder erscheint **txtWert**.

Für Zahlenfelder erscheint ebenfalls **txtWert**, bei **Ist zwischen** zusätzlich **txtWert2** für den Bis-Wert. Bei Datumsfeldern mit **Ist zwischen** erscheinen **txtDatum** (Von) und **txtDatumA** (Bis) nebeneinander.

Bei allen anderen Datumsbedingungen erscheint das Kombinationsfeld **cboDatumstyp** mit den vordefinierten Zeiträumen – und je nach gewähltem Datumstyp zusätzlich **txtDatum** für ein konkretes Datum oder **txtWert** für die Anzahl Tage bei **Letzte X Tage**.

Da alle Steuerelemente einer Zeile an derselben X-Position liegen, werden beim Wechsel der Bedingung zuerst alle ausgeblendet und dann nur das jeweils benötigte eingeblendet. Bei Datumsfeldern mit **Ist zwischen** müssen **txtDatum** und **txtDatumA** nebeneinander sichtbar sein.

Da beide im Formular übereinander bei derselben X-Position liegen, setzt das Formular beim Einblenden die **Left**-Eigenschaft von **txtDatum** auf die Position des Wertfelds und platziert **txtDatumA** direkt dahinter.

Beim Wechsel zu einer anderen Bedingung oder beim Wechsel des Felds wird **Left** auf den gespeicherten Ursprungswert zurückgesetzt. Dieser Ursprungswert ist als Konstante im Erstellungsmodul hinterlegt und gilt für alle 25 Zeilen gleichermaßen.

## Eingaben validieren

Das Filterformular validiert Tastatureingaben in den Wertfeldern in Echtzeit. Jedes der 25 **txtWert**-Steuerelemente sowie die **txtDatum**-Felder implementieren ein **KeyPress**-Ereignis das an die zentrale Funktion **ValidiereEingabe**

```
Public Function ValidiereEingabe(KeyAscii As Integer, lngFeldtyp As Long) As Integer
    Select Case lngFeldtyp
        Case cFeldtypZahl
            If Not (Chr(KeyAscii) Like "[0-9]" _
                    Or KeyAscii = 44 _
                    Or KeyAscii = 46 _
                    Or KeyAscii = 45 _
                    Or KeyAscii < 32) Then
                KeyAscii = 0
            End If
        Case cFeldtypDatum
            If Not (Chr(KeyAscii) Like "[0-9]" _
                    Or KeyAscii = 46 _
                    Or KeyAscii = 45 _
                    Or KeyAscii = 47 _
                    Or KeyAscii < 32) Then
                KeyAscii = 0
            End If
    End Select
    ValidiereEingabe = KeyAscii
End Function
```

**Listing 3: ValidiereEingabe** filtert ungültige Zeichen je nach Feldtyp heraus

# Button-Wizard programmieren

**Im Beitrag »Schnelle Schaltflächen mit Stil« (www.access-im-unternehmen.de/1563) haben wir ein Formular vorgestellt, mit dem wir komfortabel Schaltflächen anlegen können – mit Icon, Beschriftung, passendem Namen und Ereignisprozedur. Dieses Formular lässt sich in dieser Lösung allerdings nur nutzen, wenn es in der gleichen Anwendung enthalten ist, deren Formulare wir damit anpassen wollen. Um dies praktischer zu gestalten, wollen wir die Lösung in einen Assistenten umwandeln, der beim Hinzufügen eines Buttons zu einem Formular automatisch angezeigt wird. Die dazu notwendigen Schritte beschreiben wir in diesem Beitrag.**

## Steuerelement-Wizards

Die Grundlagen zum Erstellen von Steuerelement-Assistenten haben wir bereits im Beitrag **Steuerelement-Wizards programmieren** (**www.access-im-unternehmen.de/1593**) beschrieben.

## Lösung in Button-Wizards umprogrammieren

Um aus der Access-Datenbank mit dem Formular zum Erstellen von Schaltflächen (siehe Bild 1) einen Steuerelement-Wizard zu machen, benennen wir diese in eine **.accda**-Datei um.

Außerdem fügen wir eine Tabelle namens **USysRegInfo** hinzu, in der wir die Informationen speichern, die beim Installieren des Wizards über den Add-In-Manager von Access in der Registry landen sollen.

Diese werden beim Start von Access ausgelesen, damit der Wizard an der entsprechenden Stelle aufgerufen wird.

Wie diese Tabelle aufgebaut ist und was die einzelnen Einträge bedeuten, erläutern wir ausführlich im oben genannten Beitrag. In diesem Fall enthält sie die Einträge aus Bild 2.

Wichtig ist, dass der Wert in der Zeile mit dem Wert **Library** im Feld **ValName** mit dem Namen der **.accda**-Datei übereinstimmt und das wir die zum Starten verwendete VBA-Funktion genauso nennen wie für **Function** angegeben.

## Startfunktion für den Wizard anlegen

In einem neuen Modul namens **mdlAddIn** hinterlegen wir die Funktion, die beim Starten des Wizards aufgeru-

**Bild 1:** Formular zum Erstellen von Schaltflächen

USysRegInfo

| Subkey | Type | ValName | Value |
|---|---|---|---|
| HKEY_CURRENT_ACCESS_PROFILE\Wizards\Control Wizards\CommandButton\amvButtonWizard | 0 | | |
| HKEY_CURRENT_ACCESS_PROFILE\Wizards\Control Wizards\CommandButton\amvButtonWizard | 1 | Function | Autostart_amvButtonWizard |
| HKEY_CURRENT_ACCESS_PROFILE\Wizards\Control Wizards\CommandButton\amvButtonWizard | 1 | Library | \|ACCDIR\amvButtonWizard.accda |
| HKEY_CURRENT_ACCESS_PROFILE\Wizards\Control Wizards\CommandButton\amvButtonWizard | 1 | Description | amvButtonWizard |
| HKEY_CURRENT_ACCESS_PROFILE\Wizards\Control Wizards\CommandButton\amvButtonWizard | 4 | Can Edit | 1 |

Datensatz: 6 von 6 Kein Filter Suchen

**Bild 2:** Die Tabelle **USysRegInfo** mit den Informationen des Wizards für die Registry

fen werden soll. Diese gestalten wir wie in Listing 1. Die Startzeile der Funktion muss genauso aufgebaut sein und muss zwingend die angegebenen Parameter enthalten.

In der Funktion rufen wir das Formular **frmButtonWizard** als modalen Dialog auf und übergeben diesem mit **OpenArgs** den Namen der neu erstellten Schaltfläche.

Diese Funktion hat eine Fehlerbehandlung, die speziell den Fehler **2501** nicht behandelt. Dieser tritt auf, wenn wir später im Ereignis **Form_Open** des Formulars das Öffnen mit **Cancel = True** abbrechen. Die entsprechende eingebaute Fehlermeldung wollen wir unterbinden, da wir bereits durch eine Meldung in **Form_Open** darauf hinweisen, dass das Formular nicht geöffnet werden konnte.

## Wizard installieren

Damit haben wir bereits das Grundgerüst erstellt und können den Wizard installieren. Dazu schließen wir die Wizard-Datenbank und öffnen eine beliebige andere Access-

```
Public Function Autostart_amvButtonWizard(Optional strControlName As String, Optional strLabelName As String) _
        As Variant
    On Error GoTo Fehler
    DoCmd.OpenForm "frmButtonWizard", WindowMode:=acDialog, OpenArgs:=strControlName
Ende:
    Exit Function
Fehler:
    Select Case Err.Number
        Case 0
        Case 2501
        Case Else
        MsgBox "Fehler " & Err.Number & vbCrLf & Err.Description, vbOKOnly + vbCritical, _
            "Fehler beim Start von amvButtonWizard"
    End Select
End Function
```

**Listing 1:** Funktion, die beim Start des Wizards aufgerufen wird

Datenbank im Administrator-Modus. Hier starten wir über den Ribbon-Befehl **Datenbanktools|Add-Ins|Add-Ins|Add-In-Manager** den Add-In-Manager, klicken auf **Neues hinzufügen...** und wählen unsere **.accda**-Datei aus.

Danach können wir prüfen, ob der Assistent gestartet wird, wenn wir einem Formular in der Entwurfsansicht einen neuen Button hinzufügen.

Dieser sollte nun im Dialog **Generator auswählen** neben dem eingebauten Eintrag **Befehlsschaltflächen-Assistent** erscheinen.

Wenn wir dort den Eintrag **amvButtonWizard** anklicken und das Formular **frmButtonWizard** erscheint, sind wir auf dem richtigen Weg.

Das Formular des Button-Wizards ist allerdings aktuell noch so programmiert, dass einfach eine neue Schaltfläche angelegt wird, und nicht die durch das Hinzufügen über die Toolbox erzeugte Schaltfläche angepasst wird. Dies werden wir in den folgenden Abschnitten ändern.

## Anpassen des Formulars frmButtonWizard

Um die Anpassung durchzuführen und gleichzeitig testen zu können, schließen wir die Datenbank, über die wir den Wizard installiert haben.

Dann öffnen wir die **.accda**-Datenbank selbst – allerdings die Version, die durch den Add-In-Manager in das Add-Ins-Verzeichnis kopiert wurde. Dieses Verzeichnis können wir mit der folgenden Prozedur ermitteln, die eine Funktion der nicht dokumentierten Klasse **Wizhook** nutzt:

```
Public Sub AddInDir()
    WizHook.Key = 51488399
    Debug.Print WizHook.OfficeAddInDir
End Sub
```

Diese liefert auf unserem System das folgende Verzeichnis:

```
C:\Users\[Benutzername]\AppData\Roaming\Microsoft\AddIns\
```

Die in diesem Verzeichnis enthaltene Datei **amvButtonWizard.accda** öffnen wir schließlich.

Hier passen wir das Ereignis **Form_Open** an (siehe Listing 2). Diesem fügen wir eine Prüfung hinzu, ob mit dem Öffnungsargument, das wir mit **Me.OpenArgs** auslesen, der Name des neu erstellten und anzupassenden Steuerelements übergeben wurde. Falls nicht, wird das Öffnen des Assistenten an dieser Stelle abgebrochen.

Anderenfalls speichern wir den Namen dieses Buttons in der **TempVar**-Variablen **Steuerelementname**. Damit sind die Arbeiten an der **Form_Open**-Prozedur abgeschlossen.

Außerdem müssen wir in den beiden Prozeduren **txtSteuerelementname_Change** und **txtBeschriftung_Change** die Zeile auskommentieren, welche die **TempVars**-Variable **Steuerelementname** ändert.

Jetzt müssen wir den Schritt anpassen, der das im Formular konfigurierte Steuerelement anlegt.

Wir wollen hier kein neues Steuerelement anlegen, sondern die durch den Benutzer angelegte Schaltfläche soll nach den Angaben im Formular angepasst werden.

## Schaltfläche anpassen statt anlegen

Wie erwähnt, hat die bisherige Lösung einen neuen Button angelegt – wir wollen aber den über die Toolbox angelegten Button anpassen. So hat der Benutzer auch die volle Kontrolle darüber, wo das Steuerelement angelegt wird.

Diese Aktion findet in der Prozedur **SchaltflaecheAnlegen** statt.

Hier können wir zunächst den Aufruf der Prozedur **PositionLetzterButton** entfernen, die dafür gedacht war, die neue Schaltfläche unter der zuletzt angelegten Schaltfläche einzufügen – da wir nun selbst bestimmen, wo die