

ACCESS

IM UNTERNEHMEN

MICROSOFT ACCESS LEBT!

Mit der neuen Version liefert Microsoft eine Zoom-Funktion für die Benutzeroberfläche (ab Seite 2).



In diesem Heft:

VBA-OPTIONEN OPTIMAL NUTZEN

Lernen Sie die wichtigsten Optionen für den VBA-Editor kennen.

BESSER SUCHEN MIT REGULÄREN AUSDRÜCKEN

Erfahren Sie, wie Sie reguläre Ausdrücke für die Suche nach Texten nutzen können.

RECHNEN MIT DATUM

So stolpern Sie nie mehr über die typischen Fallen beim Rechnen mit Datumsangaben!

SEITE 8**SEITE 22, 32****SEITE 65**

Reguläre Ausdrücke in VBA – jetzt fest eingebaut

Wer in VBA prüfen will, ob eine Eingabe eine gültige E-Mail-Adresse ist, baut sich meist eine Schleife aus »InStr«, »Mid« und »Left« zusammen und kämpft dann mit den Sonderfällen. Mit regulären Ausdrücken genügt eine einzige Zeile. Das Werkzeug dafür gehört seit Jahren zu Windows – über eine Bibliothek, die Microsoft gerade abkündigt. Gleichzeitig hat das VBA-Team reagiert und die Funktionalität fest in VBA eingebaut. Höchste Zeit, sich das Thema gründlich anzusehen.



Dem widmen wir gleich zwei Beiträge. In **Reguläre Ausdrücke in VBA, Teil 1: Basics** binden wir das Objekt ein, lernen seine vier Klassen kennen und arbeiten die Mustersprache durch (ab Seite 22). Dort steht auch, warum Sie in ausgelieferten Anwendungen die späte Bindung verwenden sollten – nur sie übersteht den Wegfall von VBScript.

Teil 2: Praxis-Rezepte wird ab Seite 32 konkret: fertige Muster für E-Mail-Adressen, Telefonnummern und Postleitzahlen, dazu Funktionen zum Herauslösen von Daten aus Freitextfeldern, zum Bereinigen kopierter Texte und zum Füllen von Textvorlagen. Zum Schluss zeigen wir eine Falle, die Access zuverlässig zum Stillstand bringt.

Ein Muster prüft allerdings nur die Form einer Eingabe. Ob eine IBAN auch stimmt, verrät erst ihre Prüfziffer. Wie Sie diese berechnen – ebenso wie bei ISBN, GTIN, Kreditkartennummern und der Umsatzsteuer-Identifikationsnummer – zeigt **Prüfziffern per VBA berechnen und prüfen** ab Seite 45.

Warum eine Abfrage die Datensätze von heute nicht findet, warum die Kalenderwoche zum Jahreswechsel danebenliegt und warum die Summe der Arbeitszeiten plötzlich 07:59 statt 08:00 lautet, klärt **Rechnen mit Datum und Zeit ohne Überraschungen** ab Seite 65.

Gute Nachrichten für alle, die sich fragen, ob Access noch eine Zukunft hat: Das Entwicklerteam legt nach. **Zoomen in Access, Teil 2** ab Seite 2 zeigt, was seit unserem ersten Beitrag dazugekommen ist – Zoom in der Endlosansicht, in geteilten und in Popup-Formularen, und der Zoomfaktor lässt sich nun per VBA auch auslesen.

VBA-Optionen, die jeder Entwickler kennen muss stellt ab Seite 8 die wichtigsten Einstellungen des VBA-Editors vor. Die Option für die Variablendeklaration wirkt allerdings nur für neu angelegte Module – in gewachsenen Anwendungen fehlt **Option Explicit** deshalb oft an vielen Stellen. Wie Sie diese Lücken aufspüren und in einem Durchgang schließen, lesen Sie ab Seite 57 unter **Fehlendes »Option Explicit« finden und nachrüsten**.

Ähnlich flächendeckend arbeitet **Formulareigenschaften global ändern** ab Seite 13: Der Beitrag liefert ein Werkzeug, das beliebige Eigenschaften für alle Formulare und Berichte ausliest, sichert und setzt – praktisch etwa für die neuen Zoom-Eigenschaften.

Viel Spaß beim Lesen wünscht Ihnen

Ihr André Minhorst

Zoomen in Access, Teil 2

Im ersten Teil unserer Reihe mit dem Titel »Formulare zoomen in Access« (www.access-im-unternehmen.de/1605) haben wir uns den ersten Wurf von Microsoft zum Zoomen in Formularen angesehen. Hier war das Zoomen noch auf wenige Ansichten beschränkt, aber Microsoft hat nun nachgebessert. Auch haben wir bemängelt, dass es aktuell noch keine Möglichkeit gibt, den Zoomfaktor programmatisch abzufragen. Außerdem gab es nur einige wenige Befehle, mit denen wir bestimmte Zoomfaktoren einstellen konnten. Auch hier liefert die neue Version eine Neuerung, denn wir können den Zoomfaktor nun nicht nur per VBA genau setzen, sondern diesen auch abfragen. Diese und weitere Verbesserungen, die mit den Versionen 2608 und 2609 veröffentlicht wurden, stellen wir in diesem Beitrag vor.

Voraussetzung

Die hier vorgestellten Funktionen sind erst mit Version 2609 vollständig verfügbar.

Was bisher geschah ...

Im ersten Beitrag zum Thema **Zoomen in Access** haben wir bereits einige neue Funktionen rund um das Zoomen vorgestellt:

- Zoom von Tabellen, Abfragen und Formularen in der Datenblattansicht
- Formulare in der Formularansicht
- Zoomen über die Benutzeroberfläche (50 %, 75 %, 100 %, 125 %, 150 %, 175 %, 200 % und 500 %) und über die Statusleiste
- Zoomen per Tastenkombination (**Strg + Alt + Plus/Minus**)

- Zoomen per Maus (**Strg + Alt + Mausrad**)

- Einstellen eines anwendungsweiten Standard-Zoomfaktors über die Access-Optionen

Nun schauen wir uns an, was sich seit dem letzten Update mit Stand der Version 2609 geändert hat.

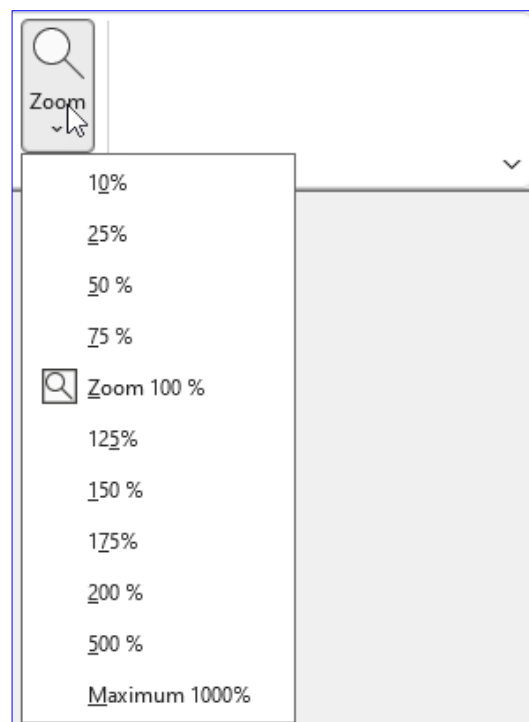


Bild 1: Neue Zoomfaktoren in der Benutzeroberfläche

Nachtrag: Standard-Zoom-Faktor per VBA

Im vorherigen Beitrag haben wir erwähnt, dass wir den Standard-Zoomfaktor mit der Access-Option **Standardzoom** einstellen können. Dies lässt sich zudem per VBA erledigen. Zum Auslesen:

```
Debug.Print Application.GetOption("Default Zoom Level")
```

Zum Einstellen (beispielsweise auf 125 %):

```
Application.SetOption("Default Zoom Level"), 125
```

Neue Zoomfaktoren über die Benutzeroberfläche verfügbar

Die erste Änderung finden wir im Ribbon vor, wenn wir bei geöffnetem Formular die Schaltfläche mit den Zoomfaktoren aufklappen (siehe Bild 1).

Hier entdecken wir neue Zoomfaktoren, nämlich **10 %**, **25 %** und **1000 %**.

Zoomen in der Endlosansicht und in geteilten Formularen

Neben den bisherigen Ansichten werden nun auch die Endlosansicht und geteilte Formulare unterstützt. Das Zoomen gelingt über die gleichen Wege wie bei den übrigen Formularansichten (siehe Bild 2).

Ändern des Zoomfaktors per Mausrad

Bisher haben wir die folgende Kombination zum Ändern des Zoom-Faktors über das Mausrad benötigt (eventuell ist diese Änderung nicht neu – aber wir hatten keine ältere Access-Version mehr zur Verfügung, um dies zu verifizieren):

Strg + Alt + Mausrad

Nun reicht jedenfalls das Betätigen der folgenden Kombination aus:

Strg + Mausrad

Zoomen von Popup-Formularen

Auch für Popup-Formulare steht die Zoom-Funktion nun zur Verfügung. Popup-Formulare sind solche Formulare, die wir zum Beispiel mit dem folgenden

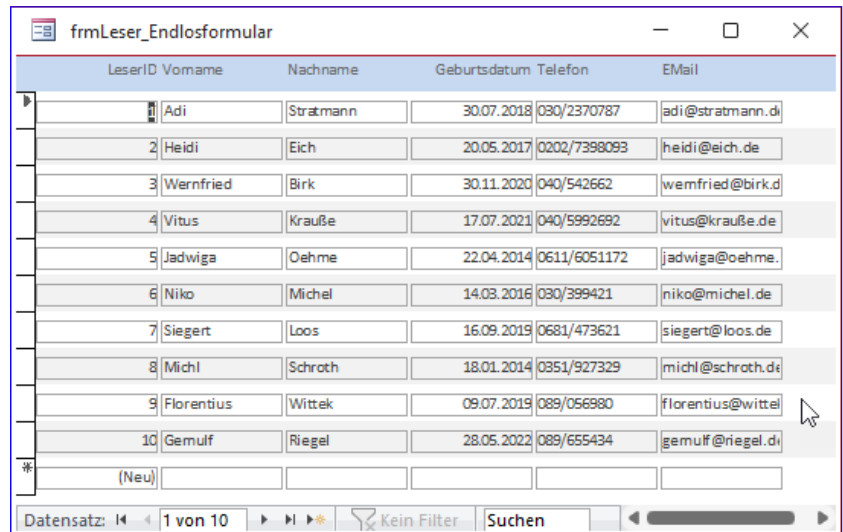


Bild 2: Ein Endlosformular mit einem Zoomfaktor von 50 %

Befehl öffnen – oder für die wir die Eigenschaft **Popup** auf **Ja** einstellen:

```
DoCmd.OpenForm "Formularname", _  
WindowMode:=acDialog
```

Zoomen anwendungsweit aktivieren und deaktivieren

In den Access-Optionen kennen wir aus dem vorherigen Beitrag bereits die Einstellung **Standardzoom**. Hier ist

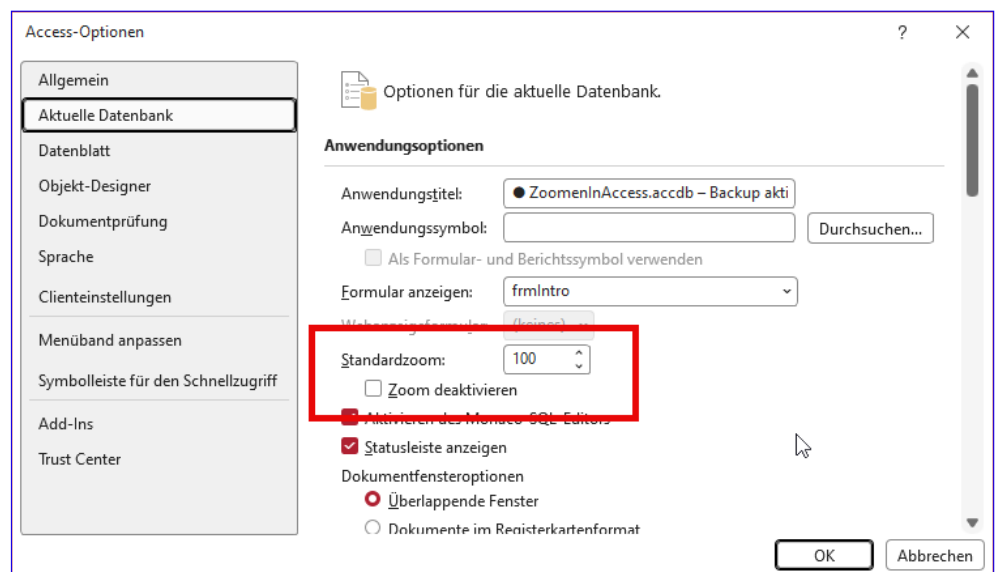


Bild 3: Einstellen des Standard-Zoomfaktors und Aktivieren oder Deaktivieren der Zoom-Funktion

eine neue Option namens **Zoom deaktivieren** hinzugekommen, mit der wir für die aktuelle Anwendung festlegen können, ob der Benutzer den Zoomfaktor einstellen kann oder nicht (siehe Bild 3).

Diese Option können wir auch per VBA auslesen oder setzen. Zum Auslesen verwenden wir die folgende Anweisung:

```
Debug.Print Application.GetOption("Disable Zoom")
```

Um die Option einzustellen, können Sie folgenden Befehl nutzen. Die Übergabe des Werts **True** deaktiviert das Zoomen, **False** aktiviert es wieder:

```
Application.SetOption _  
    "Disable Zoom", True
```

Beachten Sie, dass diese Einstellung erst nach dem erneuten Laden der Access-Datenbank wirkt.

Zoomen pro Formular ein- und ausschalten

Wir können die Zoomfunktion aber auch je Formular ein- oder ausschalten. Dazu hat Microsoft eine neue Eigenschaft namens **Zoom zulassen** hinzugefügt, die wir in den Eigenschaften des Formulars unter **Format** finden (siehe Bild 4).

Auch diese Einstellung können wir per VBA vornehmen.

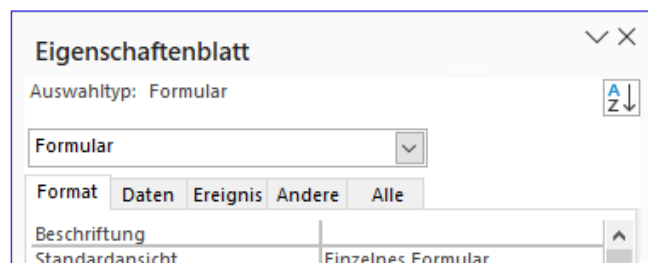


Bild 4: Zoom für ein Formular aktivieren oder deaktivieren

```
Me.Properties("AllowZooming") = True
```

Um diese Eigenschaft für ein Formular von außen einzustellen, ersetzen wir **Me** durch die entsprechende Referenz, zum Beispiel:

```
Forms!Formularname.Properties("AllowZooming") = True
```



DAS WAR DIE LESEPROBE

Hier lesen Sie weiter – und finden noch viel mehr.

Den kompletten Beitrag und alle anderen Artikel dieser Ausgabe erhalten Sie im Abonnement von **Access im Unternehmen**: praxisnahe VBA-Lösungen, komplett nachvollziehbar erklärt, dazu die Beispieldatenbanken zum Download.

Jetzt Abonnement sichern >

andreminhorst.de/access-im-unternehmen

VBA-Optionen, die jeder Entwickler kennen muss

Nicht nur Access, sondern auch der VBA-Editor bietet eine Reihe von Optionen an. Diese erlauben, das Aussehen und die Funktionalität des VBA-Editors anzupassen. Es gibt zwar nicht die Fülle von Optionen, die Sie vielleicht von modernen Entwicklungsumgebungen kennen, aber manchmal ist weniger auch mehr. In diesem Beitrag stellen wir die wichtigsten Optionen für den VBA-Editor vor – damit verhindern Sie beispielsweise dauerhaft das Verwenden von nicht deklarierten Variablen, nutzen verschiedene Einstellungen für die Fehlerbehandlung oder deaktivieren nervige Meldungen bei Syntaxfehlern.

Variablendeklaration per Option erzwingen

Die Optionen des Access-Hauptfensters sind hinlänglich bekannt. Weniger oft nutzt man hingegen die Optionen des VBA-Editors – zumindest, wenn man danach geht, wie oft wir bei unseren Kunden Anwendungen sehen, in denen die Anweisung **Option Explicit** fehlt.

Diese Anweisung finden wir üblicherweise oben im VBA-Modul, gleich hinter der Anweisung **Option Compare Database**:

Option Compare Database

Option Explicit

Allerdings finden wir sehr oft nur die erste Anweisung vor. Dabei ist **Option Explicit** sehr wichtig: Damit stellen wir sicher, dass bei der Verwendung nicht deklarerter Variablen im VBA-Code ein Kompilierfehler ausgelöst wird.

Nicht deklarierte Variablen können problematisch werden, weil so beispielsweise durch Tippfehler verursachte unterschiedliche Schreibweisen für die gleiche Variable ein unerwartetes Verhalten hervorrufen.

Warum aber finden wir die Anweisung **Option Explicit** in dem einen VBA-Projekt vor, in anderen fehlt es jedoch teilweise völlig?

Das hängt vermutlich mit einer Option des VBA-Editors zusammen, die standardmäßig deaktiviert ist, nämlich **Variablendeklaration erforderlich**. Diese finden wir gleich auf der ersten Seite des Optionen-Dialogs (siehe Bild 1).

Sobald wir diese Option aktivieren, wird **Option Explicit** bei jedem neu angelegten Modul automatisch hinzugefügt.

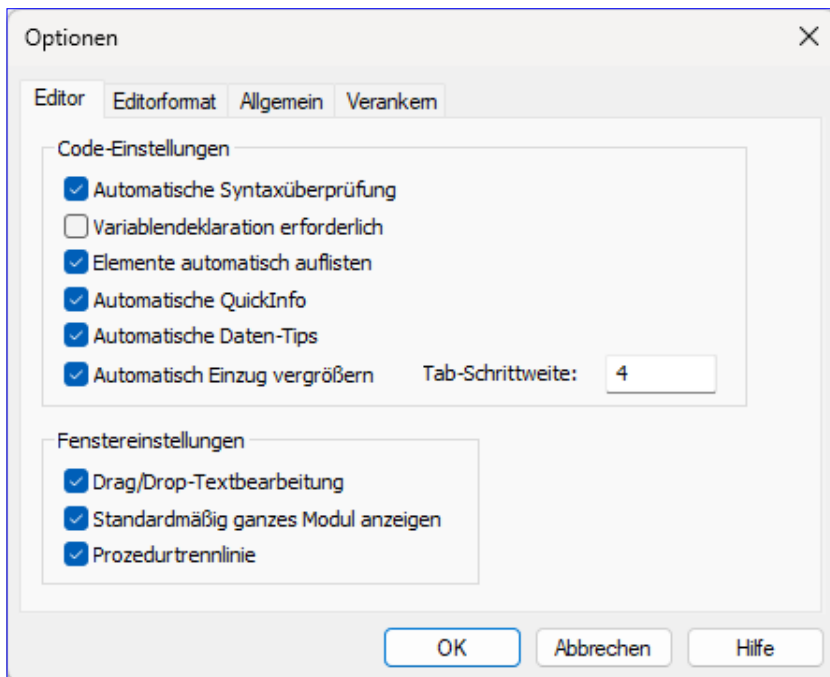


Bild 1: Erste Seite der Optionen im VBA-Editor

Leider sorgt dies nicht automatisch dafür, dass die Anweisung **Option Explicit** nachgerüstet wird, wo es noch fehlt. Eine Lösung, wie Sie schnell Module ohne **Option Explicit** finden können, zeigen wir im Beitrag **Fehlendes Option Explicit finden und nachrüsten** (www.access-im-unternehmen.de/1619).

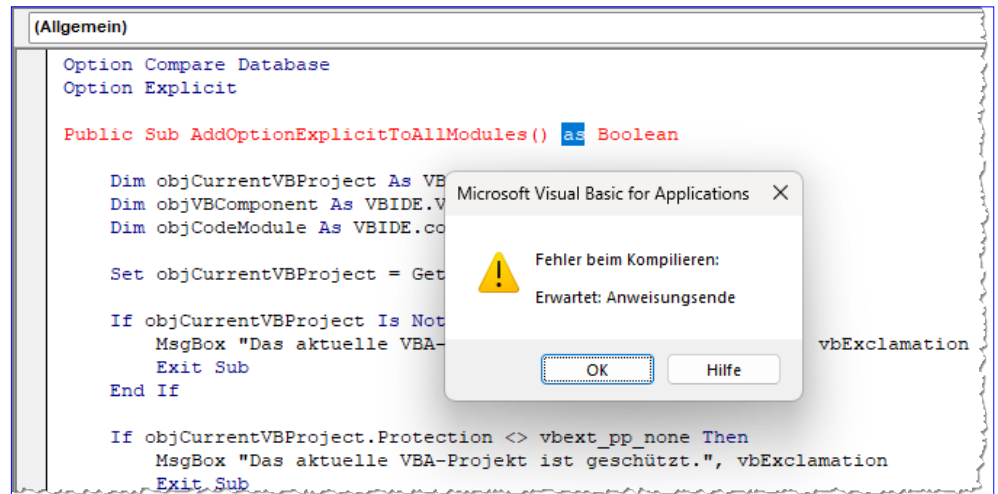


Bild 2: Anzeige einer Meldung für einen Syntaxfehler

Wer diese Option jedoch direkt aktiviert, braucht sich später nicht um fehlende **Option Explicit**-Anweisungen zu kümmern.

Automatische Syntaxüberprüfung deaktivieren

Der erste Eintrag auf der ersten Seite des Optionen-Dialogs heißt **Automatische Syntaxüberprüfung**.

Dieser ist standardmäßig aktiviert und sorgt so dafür, dass bei Eingabe von Codezeilen, die einen Syntaxfehler enthalten, direkt eine Fehlermeldung angezeigt wird.

Bild 2 zeigt ein Beispiel. Hier haben wir einen Rückgabeparameter definiert, obwohl wir keine **Function**-, sondern eine **Sub**-Prozedur definiert haben.

aus einer anderen Programmiersprache in das Modul einfügen, um dieses Schritt für Schritt in VBA zu übersetzen. Hier erhalten wir dann ständig neue Fehlermeldungen. Wenn wir diese nicht mehr erhalten wollen, können wir einfach die Option **Automatische Syntaxüberprüfung** ausschalten.

Benutzerdefinierte Fehlerbehandlung umgehen

Laufzeitfehler sollten an einer geeigneten Stelle behandelt werden, damit der Benutzer keine eingebauten Fehlermeldungen von VBA sieht. Bei größeren Prozeduren und an den Einstiegspunkten einer Anwendung empfiehlt sich in der Regel eine eigene Fehlerbehandlung. Kleinere Hilfsprozeduren können Fehler dagegen an die aufrufende Prozedur weiterreichen.

DAS WAR DIE LESEPROBE

Hier lesen Sie weiter – und finden noch viel mehr.

Den kompletten Beitrag und alle anderen Artikel dieser Ausgabe erhalten Sie im Abonnement von **Access im Unternehmen**: praxisnahe VBA-Lösungen, komplett nachvollziehbar erklärt, dazu die Beispieldatenbanken zum Download.

Jetzt Abonnement sichern >

andreminhorst.de/access-im-unternehmen

Formulareigenschaften global ändern

Mit »AllowZooming« und »ZoomResizesWindow« hat Microsoft zwei Formulareigenschaften eingeführt, die sich ausschließlich über die Properties-Auflistung ansprechen lassen – und das auch nur, wenn das Formular in der Entwurfsansicht geöffnet ist. Wer diese Eigenschaften für eine gewachsene Anwendung mit einigen Dutzend Formularen einstellen möchte, hat stumpfe Fleißarbeit vor sich. Ein guter Anlass, sich ein Werkzeug zu bauen, das beliebige Eigenschaften für alle Formulare und Berichte einer Anwendung ausliest, sichert und setzt. Dieser Beitrag zeigt, wie das gelingt – und wofür sich so ein Werkzeug jenseits der Zoom-Funktion noch alles einsetzen lässt.

Wofür braucht man so etwas?

Die neuen Zoom-Eigenschaften, auf die wir im Beitrag **Zoomen in Access, Teil 2** (www.access-im-unternehmen.de/1614) eingehen, sind nur der aktuelle Anlassfall. Wenn Sie einmal darüber nachdenken, welche Einstellungen in einer Anwendung eigentlich für **alle** Formulare gleich sein sollten, kommt schnell eine ansehnliche Liste zusammen:

- Neue Eigenschaften nachrüsten, die Microsoft ergänzt hat – neben den Zoom-Eigenschaften etwa die angekündigte Eigenschaft für abgerundete Ecken
- Ein einheitliches Erscheinungsbild herstellen: Datensatzmarkierer, Navigationsschaltflächen, Bildlaufleisten, Rahmenart und Trennlinien
- Von überlappenden Fenstern auf Registerkarten umstellen – dabei müssen **Popup**, **Modal**, **AutoCenter** und die Rahmenart bei jedem Formular angefasst werden
- Nach dem Umstieg auf ein SQL Server-Backend die Datensatzsperrung und den Recordset-Typ anpassen
- Eine Anwendung vor der Auslieferung härten: Kontextmenüs abschalten, Filter sperren, Entwurfsänderungen unterbinden

- Aus einer Vollversion eine Nur-Lese-Variante erzeugen, indem **AllowEdits**, **AllowAdditions** und **AllowDeletions** flächendeckend abgeschaltet werden
- Gespeicherte Filter und Sortierungen aufspüren: **FilterOnLoad** und **OrderByOnLoad** schleichen sich beim Entwickeln ein und sorgen beim Kunden für leere oder quälend langsame Formulare

Der letzte Punkt zeigt schon, dass wir nicht nur schreiben, sondern auch lesen wollen. Ein Werkzeug, das uns erst einmal auflistet, wie die Dinge stehen, ist oft nützlicher als eines, das blind überall den gleichen Wert einträgt.

Das Grundmuster

Der Ablauf ist bei allen Eigenschaften derselbe: Formular in der Entwurfsansicht öffnen, Eigenschaft setzen, Formular speichern und schließen. In der einfachsten Form sieht das aus wie in Listing 1.

Die Schleife läuft über die Auflistung **AllForms** des **CurrentProject**-Objekts. Diese enthält alle Formulare der Anwendung – im Gegensatz zur Auflistung **Forms**, die nur die aktuell geöffneten Formulare liefert.

Der Parameter **acHidden** sorgt dafür, dass das Formular unsichtbar geöffnet wird, was den Vorgang deutlich beschleunigt.

```
Public Sub ZoomResizesWindowSetzen(ByVal bInWert As Boolean)

    Dim strFormular As String
    Dim i As Long

    For i = 0 To CurrentProject.AllForms.Count - 1
        strFormular = CurrentProject.AllForms(i).Name
        DoCmd.OpenForm strFormular, acDesign, , , acHidden
        Forms(strFormular).Properties("ZoomResizesWindow") = bInWert
        DoCmd.Close acForm, strFormular, acSaveYes
    Next i

End Sub
```

Listing 1: Das Grundmuster in seiner einfachsten Form

Rufen wir diese Prozedur wie folgt auf, finden wir im Eigenschaftenfenster aller Formulare nun den neuen Wert für die Eigenschaft **Zoom** ändert die Fenstergröße vor (siehe Bild 1).

Warum über die Properties-Auflistung?

Die meisten Formulareigenschaften lassen sich direkt ansprechen, etwa mit **Me.AllowEdits = False**. Bei neu

hinzugekommenen Eigenschaften funktioniert das jedoch meist nicht: Sie sind noch nicht im Objektmodell hinterlegt, sodass der Compiler den Ausdruck bereits beim Übersetzen bemängelt.

Das ist auch bei einigen älteren Eigenschaften der Fall. Der Umweg über die **Properties**-Auflistung löst das Problem, weil der Eigenschaftsname dort als Zeichenkette übergeben wird und erst zur Laufzeit ausgewertet wird.

Genau daraus ergibt sich aber auch die erste Fehlerquelle. Läuft die Anwendung unter einer älteren Access-Version, in der es die Eigenschaft noch gar nicht gibt, quittiert Access den Zugriff mit dem Laufzeitfehler 3270 (**Eigenschaft nicht gefunden**) beziehungsweise mit Fehler 2455.

Da unsere Schleife dann mittendrin abbricht, bleibt womöglich ein Formular in der Entwurfsansicht offen zurück.

Weitere Schwachstellen des einfachen Musters:

- Ist ein Formular bereits geöffnet, schlägt das Öffnen in der Entwurfsansicht fehl.
- Der Bildschirm flackert während des gesamten Durchlaufs.

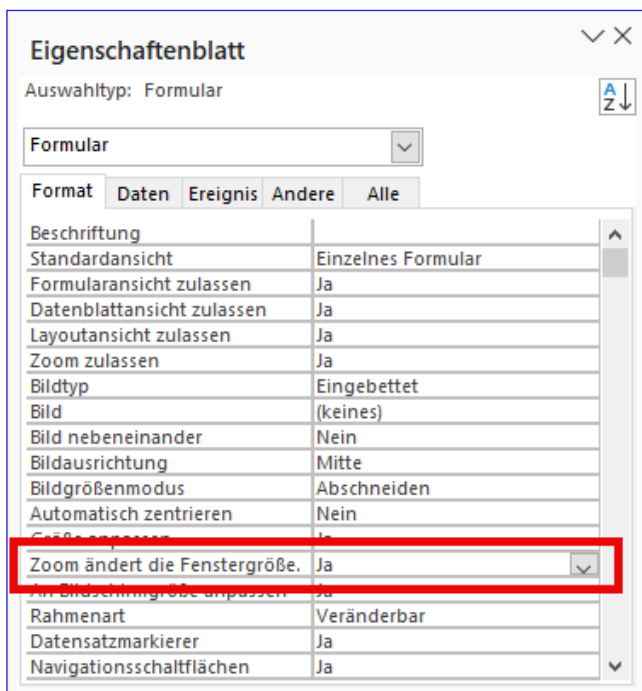


Bild 1: Per VBA-Code eingestellte Eigenschaft

```
Private Function EigenschaftSetzen(ByVal strFormular As String, ByVal strEigenschaft As String, _  
    ByVal varWert As Variant) As Long  
  
    On Error GoTo Fehler  
  
    DoCmd.OpenForm strFormular, acDesign, . . . , acHidden  
    Forms(strFormular).Properties(strEigenschaft) = varWert  
    DoCmd.Close acForm, strFormular, acSaveYes  
  
    Exit Function  
  
Fehler:  
    EigenschaftSetzen = Err.Number  
    On Error Resume Next  
    DoCmd.Close acForm, strFormular, acSaveNo  
  
End Function
```

Listing 2: Setzen einer Eigenschaft mit Fehlerbehandlung

- Es gibt keine Möglichkeit, einzelne Formulare auszunehmen.
- Die alten Werte sind unwiederbringlich verloren.

Diese vier Punkte arbeiten wir nun der Reihe nach ab.

Eine Eigenschaft für ein Formular setzen

Zunächst kapseln wir den Vorgang für ein einzelnes Formular in einer eigenen Funktion. Sie liefert **0** zurück, wenn alles geklappt hat, und andernfalls die Fehlernummer.

Wichtig ist der Fehlerbehandler: Er schließt das Formular ohne zu speichern, damit kein halb bearbeiteter Entwurf zurückbleibt (siehe Listing 2).

Der Durchlauf über alle Formulare

Darauf setzt die eigentliche Hauptprozedur auf. Sie erwartet den Eigenschaftsnamen, den gewünschten Wert und optional eine durch Semikolon getrennte Liste von Formularen, die übersprungen werden sollen.

Zurückgeliefert wird die Anzahl der geänderten Formulare – Listing 3 zeigt den Code.

Application.Echo False unterdrückt die Bildschirmaktualisierung – ohne diese Zeile flackert der Bildschirm während des gesamten Durchlaufs. Entscheidend ist, dass **Echo** in jedem Fall wieder eingeschaltet wird, auch im Fehlerfall; genau dafür sorgt die Sprungmarke **Ende** in Verbindung mit **Resume Ende**.

Die drei Zeilen zur Statusleiste innerhalb der Schleife sehen umständlich aus, sind aber nötig. Der naheliegende Aufruf **SysCmd acSysCmdSetStatus** bleibt bei abgeschalteter Bildschirmaktualisierung wirkungslos, weil Access die Statusleiste dann ebenfalls nicht neu zeichnet.

Der zweite Parameter von **Application.Echo** setzt dagegen den Text der Statusleiste, und das kurze Einschalten erzwingt genau eine Aktualisierung. **DoEvents** lässt sie durch, danach geht die Anzeige wieder aus. Weil die Formulare ohnehin unsichtbar geöffnet werden, ist von diesem Umschalten auf dem Bildschirm nichts zu sehen.

Aus demselben Grund braucht es am Ende kein **SysCmd acSysCmdClearStatus**: Das abschließende **Application**.

```
Public Function EigenschaftFuerAlleFormulare(ByVal strEigenschaft As String, ByVal varWert As Variant, _
    Optional ByVal strAusnahmen As String = "") As Long

    Dim strFormular As String
    Dim lngFehler As Long
    Dim lngGeaendert As Long
    Dim i As Long

    On Error GoTo Fehler

    Application.Echo False

    For i = 0 To CurrentProject.AllForms.Count - 1
        strFormular = CurrentProject.AllForms(i).Name
        ' Statusleiste aktualisieren

        Application.Echo True, "Bearbeite " & strFormular & " ..."
        DoEvents
        Application.Echo False

        If Not IstInListe(strFormular, strAusnahmen) Then

            If CurrentProject.AllForms(i).IsLoaded Then
                DoCmd.Close acForm, strFormular, acSaveYes
            End If

            lngFehler = EigenschaftSetzen(strFormular, strEigenschaft, varWert)
            If lngFehler = 0 Then
                lngGeaendert = lngGeaendert + 1
            Else
                Debug.Print "Fehler " & lngFehler & " bei " & strFormular
            End If
        End If
    End If

    Next i
```

DAS WAR DIE LESEPROBE

Hier lesen Sie weiter – und finden noch viel mehr.

Den kompletten Beitrag und alle anderen Artikel dieser Ausgabe erhalten Sie im Abonnement von **Access im Unternehmen**: praxisnahe VBA-Lösungen, komplett nachvollziehbar erklärt, dazu die Beispieldatenbanken zum Download.

Jetzt Abonnement sichern >

andreminhorst.de/access-im-unternehmen

Reguläre Ausdrücke in VBA, Teil 1: Basics

Wer in VBA prüfen will, ob eine Eingabe eine gültige E-Mail-Adresse ist, baut sich meist eine Schleife mit »InStr«, »Mid« und »Left« zusammen – und kämpft dann mit Sonderfällen. Mit regulären Ausdrücken erledigt eine einzige Zeile dieselbe Aufgabe. Das nötige Werkzeug gehört seit vielen Jahren zu Windows, allerdings über eine Bibliothek, die Microsoft gerade abkündigt. Gleichzeitig hat das VBA-Team reagiert und die Funktionalität fest eingebaut. Dieser erste Teil unserer Reihe zeigt, wie Sie das Objekt einbinden, welche Klassen es mitbringt und wie die Mustersprache funktioniert. Der zweite Teil liefert dann fertige Rezepte für den Alltag.

Wozu das Ganze?

Ein regulärer Ausdruck ist ein Suchmuster, das mit einer eigenen kleinen Sprache beschreibt, wie ein gesuchter Text aufgebaut ist.

Statt Zeichen für Zeichen zu prüfen, formulieren wir also die Regel – und die Suchmaschine kümmert sich um den Rest. Ein paar Aufgaben, die sich damit auf eine Zeile eindampfen lassen:

- Prüfen, ob eine Eingabe dem Format einer E-Mail-Adresse entspricht
- Alle Zahlen aus einem Freitextfeld herauslösen
- HTML-Tags aus einem kopierten Text entfernen
- Platzhalter in einer Textvorlage durch Werte ersetzen
- Mehrfache Leerzeichen und Zeilenumbrüche zu einem einzigen zusammenfassen
- Prüfen, ob eine Telefonnummer nur erlaubte Zeichen enthält

Der Einstieg lohnt sich also. Und weil das Ergebnis jeweils eine kleine Funktion ist, die Sie auch direkt in Abfragen einsetzen können, wächst mit jedem Muster Ihre eigene Werkzeugsammlung.

Das RegExp-Objekt einbinden

Bis vor Kurzem gab es genau einen Weg: das Objekt **VBScript.RegExp** aus der Bibliothek **Microsoft VBScript Regular Expressions 5.5**, die hinter der Datei **vbscript.dll** steckt. Sie binden es entweder über einen Verweis ein (frühe Bindung) oder erzeugen es zur Laufzeit über **CreateObject** (späte Bindung). Den Verweis finden Sie im VBA-Editor unter **Extras/Verweise** – Bild 1 zeigt den entsprechenden Eintrag.

Genau diese Bibliothek verschwindet aber. Microsoft hat die Abkündigung von VBScript im Oktober 2023 angekündigt und im Mai 2024 einen Zeitplan in drei Phasen nachgeschoben. In der ersten Phase, die bis 2026 oder 2027 läuft, bleibt VBScript als optionales Feature vorinstalliert und aktiviert. In der zweiten Phase wird es standardmäßig deaktiviert, und in einer dritten Phase ohne festen Termin verschwindet es vollständig aus Windows. Spätestens dann können Anwendungen nicht mehr darauf vertrauen, dass die bisherige VBScript-RegExp-Bibliothek unter Windows verfügbar ist.

Die gute Nachricht: Das VBA-Team hat vorgesorgt. **Ab Office-Version 2508 (Build 19127.20154) sind die RegExp-Klassen fest in VBA eingebaut** und stehen ohne jeden Verweis zur Verfügung. Sie können also einfach schreiben:

```
Dim objRegEx As RegExp
```

```
Set objRegExp = New RegExp
```

Ob Ihre Installation schon so weit ist, prüfen Sie am schnellsten mit einem Test im Direktbereich. Kompiliert die folgende Zeile ohne Fehler, sind die eingebauten Klassen vorhanden:

```
Debug.Print TypeName(New RegExp)  
RegExp
```

Alternativ werfen Sie einen Blick auf die Datei **vbe7.dll** im Verzeichnis der gemeinsam genutzten Office-Komponenten. Ab der Dateiversion 7.1.11.49 sind die Klassen enthalten. Sollte die Office-Version stimmen, die Klassen aber trotzdem fehlen, hilft erfahrungsgemäß eine Schnellreparatur der Office-Installation.

Für Anwendungen, die Sie ausliefern, ergibt sich daraus eine klare Empfehlung: Verwenden Sie die späte Bindung. Der Aufruf mit **CreateObject** funktioniert sowohl auf älteren Systemen, auf denen VBScript noch aktiv ist, als auch auf neuen Office-Versionen ohne VBScript – er greift dort automatisch auf die eingebauten Klassen zu. Die frühe Bindung an die neu in VBA integrierte Klasse funktioniert dagegen erst ab Office-Version 2508. Auf älteren Versionen setzt sie weiterhin einen Verweis auf die VBScript-Bibliothek voraus.

```
Dim objRegExp As Object  
Set objRegExp = CreateObject("VBScript.RegExp")
```

Eine Entwarnung noch, weil die Frage regelmäßig aufkommt: Die Objekte **Scripting.Dictionary** und **Scripting.FileSystemObject** sind von der Abkündigung **nicht** betroffen. Sie stammen aus der Bibliothek **scrrun.dll** und bleiben erhalten – das hat der zuständige Produktmanager auf Nachfrage ausdrücklich bestätigt.

Vier Klassen, mehr braucht es nicht
Das Objektmodell ist erfreulich übersichtlich.

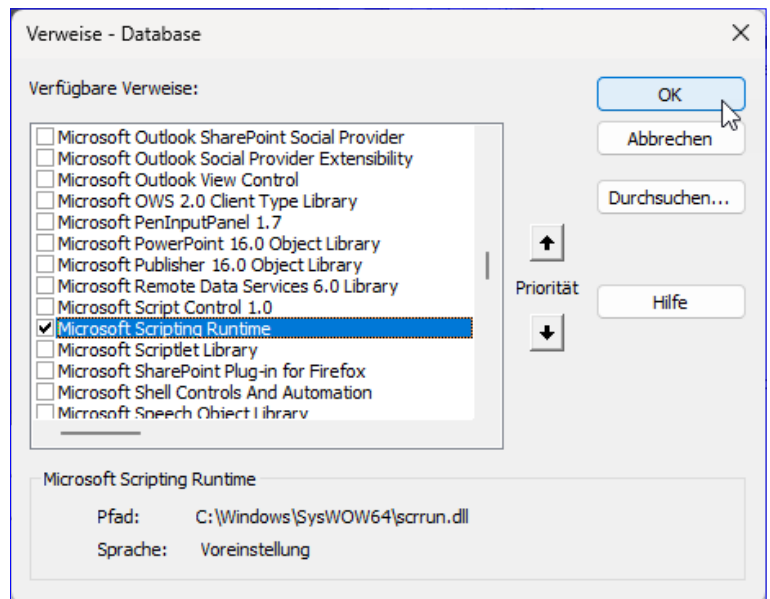


Bild 1: Der Verweis auf die VBScript-Bibliothek im **Verweise**-Dialog

Es besteht aus genau vier Klassen:

- **RegExp** ist das Suchobjekt. Hier legen Sie das Muster und die Optionen fest und starten die Suche.
- **MatchCollection** nimmt alle gefundenen Treffer auf. Die Auflistung ist nullbasiert und kennt die Eigenschaft **Count**.
- **Match** steht für einen einzelnen Treffer mit den Eigenschaften **Value**, **FirstIndex** und **Length**.
- **SubMatches** enthält die Inhalte der Klammergruppen innerhalb eines Treffers – ebenfalls nullbasiert.

Bei der späten Bindung deklarieren Sie alle diese Variablen schlicht als **Object**. Sie verlieren dadurch zwar IntelliSense, gewinnen aber die Lauffähigkeit auf allen Systemen.

Die vier Eigenschaften des RegExp-Objekts

Bevor wir suchen, stellen wir das Objekt ein. Es gibt genau vier Eigenschaften, und drei davon sind Schalter:

- **Pattern** nimmt den regulären Ausdruck als Zeichenkette auf.
- **Global** bestimmt, ob nach dem ersten Treffer weitergesucht wird. **Der Standardwert ist False** – das ist die häufigste Stolperfalle überhaupt, denn ein **Replace** ersetzt dann nur das erste Vorkommen.
- **IgnoreCase** schaltet die Unterscheidung von Groß- und Kleinschreibung ab. Auch hier lautet der Standardwert **False**, Suchen erfolgen also zunächst groß-/kleinschreibungsempfindlich.
- **Multiline** sorgt dafür, dass die Anker für Zeilenanfang und Zeilenende nicht nur am Anfang und Ende des gesamten Textes greifen, sondern an jedem Zeilenumbruch.

Damit haben wir alles beisammen, um die erste Suche zu starten.

Die Methode Test: Passt das Muster?

Am einfachsten ist **Test**. Die Methode liefert schlicht **True** oder **False** zurück, je nachdem, ob das Muster irgendwo im übergebenen Text vorkommt.

Das folgende Listing prüft, ob eine Zeichenkette ausschließlich eine fünfstellige Postleitzahl enthält.

```
Public Sub TestPostleitzahl()
    Dim objRegex As Object

    Set objRegex = CreateObject("VBScript.RegExp")
    objRegex.Pattern = "^d{5}$"
    Debug.Print objRegex.Test("47051")
    Debug.Print objRegex.Test("4705")
    Debug.Print objRegex.Test("47051 Duisburg")
End Sub
```

Die Ausgabe im Direktbereich sieht so aus:

```
Wahr
Falsch
Falsch
```

Der dritte Aufruf zeigt, wozu die beiden Anker gut sind. Das Dach am Anfang verlangt, dass der Treffer am Textanfang beginnt, das Dollarzeichen am Ende, dass er am Textende aufhört.

Ohne diese beiden Zeichen würde auch die dritte Zeile ein **Wahr** liefern, weil die fünf Ziffern ja irgendwo im Text vorkommen. Das ist ein beliebter Fehler bei Prüffunktionen: Wer verifizieren will, dass ein Feld **ausschließlich** aus dem gewünschten Muster besteht, muss beide Anker setzen.

Die Methode Execute: Was wurde gefunden?

Interessanter wird es, wenn wir wissen wollen, **was** gefunden wurde. Dazu dient **Execute**. Die Methode liefert eine **MatchCollection** zurück, durch die wir schlicht mit **For Each** laufen. Listing 1 sucht alle Zahlen in einem Freitext.

Der Direktbereich meldet:

```
Treffer: 7
12      4      4711
22      2      12
25      2      03
28      4      2026
39      1      3
50      2      19
53      2      90
```

Zwei Beobachtungen. Erstens war hier **Global = True** entscheidend – ohne diese Zeile hätten wir genau einen Treffer erhalten. Zweitens rechnen wir bei **FirstIndex** eine Eins hinzu.

Die Eigenschaft ist nämlich nullbasiert, während **Mid** und **Left** in VBA bei Eins zu zählen beginnen. Wer diesen Versatz übersieht, schneidet konsequent ein Zeichen zu früh aus. Dass das Datum in drei Teile zerfällt, liegt

natürlich am Muster: Wir haben **Ziffernfolge** gesagt, nicht **Datum**. Wie man das sauber formuliert, sehen wir gleich.

Klammergruppen und SubMatches

Klammergruppen haben in einem regulären Ausdruck zwei Aufgaben. Sie fassen einen Teilausdruck zusammen, und sie merken sich, was an dieser Stelle gefunden wurde. Genau diese gemerkten Teile landen in der Auflistung **SubMatches**. Damit zerlegen wir ein Datum in seine Bestandteile (siehe Listing 2).

Beachten Sie die maskierten Punkte im Muster. Ein unmaskierter Punkt bedeutet nämlich **ein beliebiges Zeichen** – wir wollen hier aber wirklich einen Punkt sehen. Der umgekehrte

```
Public Sub ZahlenFinden()  
    Dim objRegExp As Object  
    Dim objTreffer As Object  
    Dim objMatch As Object  
    Dim strText As String  
  
    strText = "Bestellung 4711 vom 12.03.2026 über 3 Stueck zu je 19,90 Euro"  
  
    Set objRegExp = CreateObject("VBScript.RegExp")  
    objRegExp.Pattern = "\d+"  
    objRegExp.Global = True  
  
    Set objTreffer = objRegExp.Execute(strText)  
  
    Debug.Print "Treffer: " & objTreffer.Count  
  
    For Each objMatch In objTreffer  
        Debug.Print objMatch.FirstIndex + 1, objMatch.Length, objMatch.Value  
    Next objMatch  
  
End Sub
```

Listing 1: Alle Zahlen eines Textes ermitteln

```
Public Sub DatumZerlegen()  
  
    Dim objRegExp As Object  
    Dim objTreffer As Object  
    Dim objMatch As Object  
  
    Set objRegExp = CreateObject("VBScript.RegExp")  
    objRegExp.Pattern = "(\\d{1,2})\\.(\\d{1,2})\\.(\\d{4})"  
    objRegExp.Global = True
```

DAS WAR DIE LESEPROBE

Hier lesen Sie weiter – und finden noch viel mehr.

Den kompletten Beitrag und alle anderen Artikel dieser Ausgabe erhalten Sie im Abonnement von **Access im Unternehmen**: praxisnahe VBA-Lösungen, komplett nachvollziehbar erklärt, dazu die Beispieldatenbanken zum Download.

Jetzt Abonnement sichern >

andreminhorst.de/access-im-unternehmen

Reguläre Ausdrücke in VBA, Teil 2: Praxis-Rezepte

Im ersten Teil dieser Reihe haben wir das RegEx-Objekt eingebunden, seine vier Klassen kennengelernt und die Mustersprache durchgearbeitet. Nun wird es konkret. Dieser Beitrag liefert fertige Muster für die Prüfung von E-Mail-Adressen, Telefonnummern, Postleitzahlen und Kfz-Kennzeichen, dazu Funktionen zum Herauslösen von Daten aus Freitextfeldern, zum Bereinigen kopierter Texte und zum Füllen von Textvorlagen. Alle Beispiele setzen die drei Hilfsfunktionen aus dem ersten Teil voraus – und am Ende zeigen wir eine Falle, die eine Access-Anwendung zuverlässig zum Stillstand bringt.

Muster gehören in Konstanten

Bevor wir loslegen, eine organisatorische Empfehlung: Legen Sie ein Standardmodul an, beispielsweise namens **mdlRegExPatterns**, in dem alle Muster als Konstanten stehen (siehe Listing 1). Der Grund ist weniger die Schreibarbeit als die Wartung – wenn sich herausstellt, dass Ihr Muster für Telefonnummern eine Schreibweise nicht erkennt, wollen Sie das an genau einer Stelle korrigieren und nicht an mehreren anderen Stellen.

Beachten Sie den maskierten Bindestrich in den Zeichenklassen. Innerhalb eckiger Klammern hat der Bindestrich die Bedeutung eines Bereichs, weshalb wir ihm mit einem vorangestellten Schrägstrich diese Rolle nehmen. Alternativ dürfen Sie ihn auch unmaskiert an die letzte Position der Klasse setzen.

Die Muster prüfen jeweils nur den formalen Aufbau. Bei einer IBAN werden weder die länderspezifische Länge noch die Prüfziffer kontrolliert. Auch das Kennzeichenmuster deckt bewusst nur die üblichen deutschen Standard-, Elektro- und Historienkennzeichen ab. Saison-, Wechsel- und Sonderkennzeichen benötigen eigene Erweiterungen.

Rezept 1: E-Mail-Adressen prüfen

Vorweg eine Einordnung, die Ihnen später Ärger erspart: Ein regulärer Ausdruck, der wirklich

jede nach den Internetstandards zulässige E-Mail-Adresse korrekt beurteilt, ist mehrere tausend Zeichen lang und in der Praxis kaum sinnvoll einzusetzen.

Ein pragmatisches Muster ist deshalb meist die bessere Wahl: Es fängt grobe Eingabefehler wie ein fehlendes At-Zeichen oder einen vergessenen Punkt in der Domain ab, bleibt aber verständlich und wartbar.

```
Option Compare Database
Option Explicit

Public Const cstrRegExEmail As String = _
    "^([A-Za-z0-9_%+\-]+(?:\.[A-Za-z0-9_%+\-]+)*)" & _
    "@([A-Za-z0-9](?:[A-Za-z0-9\-\]*[A-Za-z0-9])?)" & _
    "(?:\.[A-Za-z0-9](?:[A-Za-z0-9\-\]*[A-Za-z0-9])?)?)+$"

Public Const cstrRegExPlzDe As String = "^d{5}$"

Public Const cstrRegExIban As String = _
    "^([A-Z]{2}\d{2}[A-Z0-9]{11,30})$"

Public Const cstrRegExKennzeichen As String = _
    "^([A-ZÄÖÜ]{1,3}-[A-ZÄÖÜ]{1,2} ?\d{1,4}[EH]?)$"

Public Const cstrRegExUrl As String = _
    "^https?://([A-Za-z0-9]" & _
    "(?:[A-Za-z0-9\-\]*[A-Za-z0-9])?" & _
    "(?:\.[A-Za-z0-9]" & _
    "(?:[A-Za-z0-9\-\]*[A-Za-z0-9])?)+)" & _
    "(?:\d{1,5})?(?:[/#?][^\s]*)?$"
```

Listing 1: Die Muster gesammelt an einer Stelle

Was ein Muster grundsätzlich nicht leisten kann, ist die Prüfung, ob das Postfach tatsächlich existiert. Verlässlich klären lässt sich das nur über eine Bestätigungsnachricht.

Mit dieser Einschränkung im Hinterkopf eignet sich das Muster aus **cstrRegExEMail** gut als Plausibilitätsprüfung:

```
Debug.Print RegExTest("andre@access-im-unternehmen.de", _
    cstrRegExEMail)
Debug.Print RegExTest("andre@example.de", cstrRegExEMail)
Debug.Print RegExTest("andre.example.de", cstrRegExEMail)
```

Die Ausgabe im Direktbereich:

```
Wahr
Falsch
Falsch
```

In einem Formular setzen Sie die Prüfung sinnvollerweise im Ereignis **Vor Aktualisierung** des Textfelds ein. Wichtig ist die vorherige Prüfung auf einen leeren Wert: Ist die Eingabe optional, soll ein leergelassenes Feld nicht als ungültige E-Mail-Adresse beanstandet werden (siehe Listing 2).

Eine Einschränkung noch: In der Gültigkeitsregel eines Tabellenfelds lässt sich diese Prüfung nicht unterbringen, weil dort keine selbst geschriebenen Funktionen erlaubt sind. Sie gehört daher ins Formular

```
Private Sub txtEMail_BeforeUpdate(Cancel As Integer)

    If Len(Nz(Me.txtEMail, "")) = 0 Then Exit Sub

    If Not RegExTest(Me.txtEMail, cstrRegExEMail) Then
        MsgBox "Bitte geben Sie eine gültige E-Mail-Adresse ein.", vbExclamation
        Cancel = True
    End If

End Sub
```

Listing 2: Prüfung der E-Mail vor Aktualisierung des Steuerelements

– oder in eine Auswahlabfrage, mit der Sie den Bestand nachträglich auf fehlerhafte Werte untersuchen.

Rezept 2: Telefonnummern prüfen und vereinheitlichen

Telefonnummern sind der Klassiker unter den unsauberen Daten. In einer gewachsenen Tabelle finden sich Schreibweisen wie **0203/1234567**, **+49 203 1234567**, **(0203) 12 34 56 7** und **0203-1234567** nebeneinander. Ein Muster, das alle diese Varianten erkennt und gleichzeitig Unsinn zurückweist, wird schnell unübersichtlich.

Der praxistauglichere Weg besteht aus zwei Schritten: Zuerst prüfen wir grob auf erlaubte Zeichen, dann zählen wir die Ziffern. Das ist robuster als ein einzelnes Monstermuster und deutlich leichter zu warten (siehe Listing 3).

```
Public Function IstTelefonnummer(ByVal strNummer As String) As Boolean

    Dim strZiffern As String

    ' Nur erlaubte Zeichen?
    If Not RegExTest(strNummer, "^[0-9 \-/\(\)\+]+$") Then Exit Function

    ' Wie viele Ziffern bleiben übrig?
    strZiffern = RegExErsetzen(strNummer, "\D", "")

    IstTelefonnummer = (Len(strZiffern) >= 6) And (Len(strZiffern) <= 15)

End Function
```

Listing 3: Zweistufige Prüfung einer Telefonnummer

```
Public Function TelefonNormalisieren(ByVal strNummer As String, Optional ByVal strLand As String = "49") As String

    Dim strZiffern As String

    strZiffern = RegExErsetzen(strNummer, "\D", "")

    If Left(strZiffern, 2) = "00" Then
        ' 0049203... -> 49203...
        strZiffern = Mid(strZiffern, 3)
    ElseIf Left(strZiffern, 1) = "0" Then
        ' 0203... -> 49203...
        strZiffern = strLand & Mid(strZiffern, 2)
    End If

    TelefonNormalisieren = "+" & strZiffern

End Function
```

Listing 4: Umwandlung in das internationale Format

Die zweite Zeile der Funktion ist gleich noch für etwas anderes gut. Der Ausdruck `\D` trifft alles, was keine Ziffer ist – und weil wir ihn durch einen Leerstring ersetzen, bleiben genau die Ziffern übrig.

Auf dieser Idee baut die Vereinheitlichung auf, mit der wir alle Nummern in das internationale Format überführen (siehe Listing 4).

Ein Testlauf zeigt, dass die vier eingangs genannten Schreibweisen alle beim gleichen Ergebnis landen. Hier die Aufrufe:

```
Debug.Print TelefonNormalisieren("0203/1234567")
Debug.Print TelefonNormalisieren("+49 203 1234567")
Debug.Print TelefonNormalisieren("(0203) 12 34 56 7")
Debug.Print TelefonNormalisieren("0203-1234567")
```

Das Ergebnis lautet einheitlich:

```
+492031234567
+492031234567
+492031234567
+492031234567
```

Genau das brauchen Sie, wenn Sie Dubletten finden oder Nummern an eine Telefonanlage übergeben wollen. Die Funktion setzt allerdings voraus, dass nationale Nummern mit einer führenden Null und internationale Nummern mit **+** oder **00** gespeichert sind. Eine Nummer wie **2031234567** lässt sich ohne weitere Information keinem Land sicher zuordnen. Als Aktualisierungsabfrage über einen entsprechend geprüften Bestand räumt die Funktion in einem Durchgang auf.

Rezept 3: Daten aus Freitext herauslösen

Häufiger als das Prüfen ist das Extrahieren. In fast jeder gewachsenen Datenbank gibt es ein Bemerkungsfeld, in dem über Jahre Informationen gelandet sind, die eigentlich in eigene Felder gehören. Mit **RegexGruppe** aus dem ersten Teil holen wir sie heraus.

Der Klassiker ist die Adresszeile, in der Postleitzahl und Ort zusammenstehen. Das Muster trennt beides und liefert je nach Parameter den einen oder den anderen Teil:

```
Public Function PlzAusAdresse( _
    ByVal strZeile As String) As String
    PlzAusAdresse = RegexGruppe(strZeile, _
```

```
Public Function BetragAusText(ByVal strText As String) As Variant

    Dim strTreffer As String

    strTreffer = RegExGruppe(strText, "\d{1,3}(?:\.\d{3})+\,\d{2}|\d+\,\d{2}", 0)

    If Len(strTreffer) = 0 Then
        BetragAusText = Null
    Else
        BetragAusText = CCur(strTreffer)
    End If

End Function
```

Listing 5: Herauslösen eines Geldbetrags

```
        "\b(\d{5})\s+(.)$", 1)
End Function

Public Function OrtAusAdresse( _
    ByVal strZeile As String) As String
    OrtAusAdresse = RegExGruppe(strZeile, _
        "\b(\d{5})\s+(.)$", 2)
End Function
```

Ebenso oft gesucht: der Betrag in einem Text. Deutsche Beträge sind mit ihrem Punkt als Tausendertrennzeichen und dem Komma als Dezimaltrenner etwas eigenwillig, aber gut zu fassen. Der erste Teil des Musters beschreibt die Variante mit Tausenderpunkten, der zweite die ohne (siehe Listing 5).

Der Aufruf mit dem Text **Rechnungsbetrag 1.234,50 Euro netto** liefert den Währungswert 1234,5 zurück. Beachten Sie die Reihenfolge der beiden Alternativen: Die Suchmaschine nimmt die erste, die passt – stünde die einfache Variante vorn, bekämen wir bei **1.234,50** nur die **234,50** zu sehen. Diese Reihenfolge ist bei Alternativen fast immer entscheidend, und der Fehler ist tückisch, weil das Ergebnis ja plausibel aussieht.

Bis hierhin haben wir immer nur den ersten Treffer geholt. Oft brauchen wir aber alle: sämtliche E-Mail-Adressen in einem Verteiler, alle Rechnungsnummern in einer Bemerkung,

jeden Platzhalter einer Textvorlage. Dafür lohnt sich eine vierte Hilfsfunktion namens **RegExAlleTreffer**, die wir zu den dreien aus dem ersten Teil stellen (siehe Listing 6). Sie folgt demselben Bauprinzip und erwartet diese Parameter:

- **strText**: der zu durchsuchende Text.
- **strMuster**: der reguläre Ausdruck.
- **strTrenner**: optional, Vorgabewert ist ein Semikolon mit Leerzeichen. Damit werden die gefundenen Stellen zu einer Zeichenkette verkettet.

Die Eigenschaft **Global** steht hier zwingend auf **True** – ohne sie liefert die Suche genau einen Treffer, und die ganze Funktion wäre sinnlos. Weil das Ergebnis eine einfache Zeichenkette ist, lässt sich die Funktion unmittelbar in der Feldliste einer Abfrage einsetzen.

Ein durchgehendes Beispiel: die Verteilerliste

Nehmen wir eine Aufgabe, die in der Praxis regelmäßig anfällt. Ein Mitarbeiter kopiert den Verteiler einer E-Mail in das Bemerkungsfeld eines Kundendatensatzes. Was dort ankommt, ist alles andere als einheitlich – mal mit Namen und spitzen Klammern, mal mit runden Klammern, getrennt durch Semikola, Kommata oder Zeilenumbrüche:

```
Public Function RegExAlleTreffer(ByVal strText As String, ByVal strMuster As String, _
    Optional ByVal strTrenner As String = "; ") As String

    Static objRegEx As Object
    Dim objTreffer As Object
    Dim objMatch As Object
    Dim strErgebnis As String

    If objRegEx Is Nothing Then
        Set objRegEx = CreateObject("VBScript.RegExp")
    End If

    objRegEx.Pattern = strMuster
    objRegEx.Global = True
    objRegEx.IgnoreCase = True

    Set objTreffer = objRegEx.Execute(strText)

    For Each objMatch In objTreffer
        If Len(strErgebnis) > 0 Then
            strErgebnis = strErgebnis & strTrenner
        End If
        strErgebnis = strErgebnis & objMatch.Value
    Next objMatch

    RegExAlleTreffer = strErgebnis

End Function
```

Listing 6: Alle Fundstellen als Zeichenkette

```
Dim strVerteiler As String
strVerteiler = _
    "Anna Meier <a.meier@example.de>; " & _
    "Bernd Schulz (b.schulz@example.de), " & vbCrLf & _
    "c.mueller@example.de;" & vbCrLf & _
    "Dieter Weiß <d.weiss@beispiel.at>"
```

Für die Suche brauchen wir eine zweite Variante unseres E-Mail-Musters. Das Muster von weiter oben trägt die beiden Anker und beschreibt deshalb einen Text, der ausschließlich aus einer Adresse besteht.

Zum Herauslösen aus einem längeren Text müssen die Anker weg:

```
Public Const cstrRegExAdresse As String = _
    "[A-Za-z0-9._%+\-]+@[A-Za-z0-9.\-]+\.[A-Za-z]{2,}"
```

Das ist ein Punkt, an dem viele hängen bleiben, deshalb sei er ausdrücklich festgehalten: Prüfen und Extrahieren brauchen unterschiedliche Muster. Beim Prüfen wollen wir wissen, ob der gesamte Text eine Adresse ist – beim Extrahieren suchen wir Adressen irgendwo darin. Verwenden Sie versehentlich das verankerte Muster zum Suchen, finden Sie nie etwas; verwenden Sie das ankerlose Muster zum Prüfen, geht jeder Unsinn durch, der irgendwo ein Klammeraffe enthält.

Der Aufruf liefert dann eine saubere Liste:

```
Debug.Print RegExAlleTreffer(strVerteiler, _  
    cstrRegExAdresse)  
a.meier@example.de; b.schulz@example.de;  
c.mueller@example.de; d.weiss@beispiel.at
```

Bemerkenswert ist, was die Funktion alles ignoriert hat: die Namen davor, die spitzen und runden Klammern, die unterschiedlichen Trennzeichen und die Zeilenumbrüche. Genau das ist die Stärke regulärer Ausdrücke gegenüber

einer selbstgeschriebenen Zerlegung – wir mussten kein einziges dieser Sonderzeichen vorhersehen.

Von der Zeichenkette zum Array

Für die Anzeige ist eine verkettete Zeichenkette ideal, für die Weiterverarbeitung nicht. Wenn wir jede Adresse einzeln in eine Tabelle schreiben oder als Empfänger an eine E-Mail hängen wollen, brauchen wir ein Array. Die Variante aus Listing 7 liefert genau das.

```
Public Function RegExTrefferArray(ByVal strText As String, ByVal strMuster As String) As Variant  
  
    Static objRegEx As Object  
    Dim objTreffer As Object  
    Dim strErgebnis() As String  
    Dim i As Long  
  
    If objRegEx Is Nothing Then  
        Set objRegEx = CreateObject("VBScript.RegExp")  
    End If  
  
    objRegEx.Pattern = strMuster  
    objRegEx.Global = True  
    objRegEx.IgnoreCase = True  
  
    Set objTreffer = objRegEx.Execute(strText)  
  
    ' Kein Treffer: leeres Array zurueckgeben  
    If objTreffer.Count = 0 Then  
        RegExTrefferArray = Array()  
    End If  
End Function
```

DAS WAR DIE LESEPROBE

Hier lesen Sie weiter – und finden noch viel mehr.

Den kompletten Beitrag und alle anderen Artikel dieser Ausgabe erhalten Sie im Abonnement von **Access im Unternehmen**: praxisnahe VBA-Lösungen, komplett nachvollziehbar erklärt, dazu die Beispieldatenbanken zum Download.

Jetzt Abonnement sichern >

andreminhorst.de/access-im-unternehmen

Prüfziffern per VBA berechnen und prüfen

Im Beitrag »Reguläre Ausdrücke in VBA, Teil 2: Praxis-Rezepte« (www.access-im-unternehmen.de/1617) haben wir mit einem regulären Ausdruck geprüft, ob eine IBAN formal richtig aufgebaut ist. Das schützt aber nur vor groben Schnitzern – ein Zahlendreher in der Kontonummer geht anstandslos durch. Genau dafür gibt es Prüfziffern: eine oder zwei zusätzliche Stellen, aus den übrigen berechnet, die beim Vertippen fast immer nicht mehr passen. Dieser Beitrag zeigt die Verfahren hinter IBAN, ISBN, GTIN, Kreditkartennummer und Umsatzsteuer-Identifikationsnummer und liefert sie als fertige Funktionen, die Sie in Formularen und Abfragen einsetzen können.

Was eine Prüfziffer leistet – und was nicht

Eine Prüfziffer enthält redundante Information. Sie enthält keine neue Information, sondern bildet in verdichteter Form ab, was in den übrigen Stellen bereits enthalten ist. Weicht sie ab, war ein Fehler im Spiel.

Die gängigen Verfahren erkennen einzelne falsch eingegebene Ziffern und viele typische Zahlendreher. Welche Fehler garantiert erkannt werden, hängt jedoch vom jeweiligen Verfahren ab: Das für die GTIN verwendete Modulo-10-Verfahren lässt beispielsweise bestimmte Vertauschungen benachbarter Ziffern unentdeckt, während die Modulo-11- und Modulo-97-Verfahren hier strenger sind.

Was eine Prüfziffer dagegen **nicht** leistet: Sie sagt nichts darüber aus, ob es das Konto, das Buch oder die Firma tatsächlich gibt. Eine IBAN mit korrekter Prüfziffer kann problemlos frei erfunden sein.

Und sie ersetzt auch nicht die Formatprüfung – beide Schritte gehören zusammen.

Das gemeinsame Muster

So unterschiedlich die Verfahren im Detail sind, der Ablauf ist immer derselbe. Erst wird der Wert normalisiert, dann die Struktur geprüft und zuletzt gerechnet. Beim Normalisieren dürfen allerdings **nur die für den jeweiligen Nummerntyp zulässigen Trennzeichen entfernt**

werden. Alle anderen Zeichen müssen zur Ablehnung der Eingabe führen.

Das klingt nach Erbsenzählerei, ist aber sicherheitsrelevant. Eine Funktion, die unterschiedslos jedes nicht alphanumerische Zeichen wegwirft, würde die Eingabe **DE89!3704 0044 0532 0130 00** klaglos als gültige IBAN durchwinken – das Ausrufezeichen verschwindet ja vor der Prüfung. Für eine Validierung ist das zu großzügig.

Wir trennen deshalb in zwei Schritte. Zuerst eine Funktion, die tatsächlich nur Ziffern und Großbuchstaben übrig lässt – siehe Listing 1.

Ein Aufruf im Direktbereich zeigt, was die Funktion tut:

```
Debug.Print NurAlphanumerisch("DE89 3704 0044")  
Debug.Print NurAlphanumerisch("0-8044-2957-x")
```

Ergebnis:

```
DE8937040044  
080442957X
```

Die Prüfung über die ASCII-Codes der Zeichen statt über einen Vergleich der Buchstaben ist beabsichtigt. Unter **Option Compare Database** würde ein Ausdruck wie **strZeichen >= "A"** auch Umlaute einschließen, was hier unerwünscht wäre.

```
Public Function NurAlphanumerisch(ByVal strWert As String) As String

    Dim lngPos As Long
    Dim strZeichen As String
    Dim strErgebnis As String

    For lngPos = 1 To Len(strWert)

        strZeichen = UCase(Mid(strWert, lngPos, 1))

        Select Case Asc(strZeichen)
            Case 48 To 57, 65 To 90
                strErgebnis = strErgebnis & strZeichen
        End Select

    Next lngPos

    NurAlphanumerisch = strErgebnis

End Function
```

Listing 1: Entfernen aller Trenn- und Sonderzeichen

Und darüber liegt eine zweite Funktion, die zuerst kontrolliert, ob im Rohwert überhaupt nur erlaubte Zeichen vorkommen. Ist das nicht der Fall, liefert sie einen Leerstring zurück – und jede nachfolgende Strukturprüfung schlägt fehl:

```
Private Function Bereinigt( _
    ByVal strWert As String, _
    ByVal strErlaubt As String) As String

    If Not RegExTest(strWert, strErlaubt) Then Exit Function

    Bereinigt = NurAlphanumerisch(strWert)

End Function
```

Der Unterschied wird sofort sichtbar. Die eckigen Klammern im Aufruf dienen nur dazu, den Leerstring im Direktbereich erkennbar zu machen:

```
Debug.Print "[" & Bereinigt("DE89 3704", _
    "[A-Z0-9 ]+$") & "]"
```

```
Debug.Print "[" & Bereinigt("DE89!3704", _
    "[A-Z0-9 ]+$") & "]"
```

Ergebnis:

```
[DE893704]
[]
```

Welche Trennzeichen erlaubt sind, entscheidet jedes Verfahren für sich: bei der IBAN nur Leerzeichen, bei ISBN und GTIN zusätzlich Bindestriche. Die Funktion **RegExTest** stammt aus dem vorigen Beitrag dieser Reihe; wer sie nicht zur Hand hat, kann die Vorprüfung auch mit einer Schleife über die Zeichen erledigen.

Rechnen ohne Überlauf: das Modulo-97-Verfahren

Die IBAN verwendet ein Verfahren, das auf den ersten Blick in VBA gar nicht umsetzbar scheint. Man ersetzt jeden Buchstaben durch eine zweistellige Zahl, hängt die ersten vier Zeichen hinten an und bildet vom Ergebnis den Rest bei der Division durch 97. Das Problem: Bei einer

deutschen IBAN entsteht dabei eine Zahl mit über 20 Stellen.

Ein **Long** reicht nur bis 2.147.483.647, und ein **Double** kann sehr lange Ganzzahlen nicht mehr zifferngau darstellen.

Der Ausweg ist so einfach wie elegant. Statt die ganze Zahl zu bilden, arbeiten wir uns Ziffer für Ziffer vor und nehmen nach jedem Schritt den Rest. Weil dieser höchstens 96 sein kann, bleibt der

Zwischenwert immer unter 970 und passt damit mühelos in ein **Long** (siehe Listing 2).

```
Public Function Modulo97(ByVal strZiffern As String) As Long

    Dim lngPos As Long
    Dim lngRest As Long

    For lngPos = 1 To Len(strZiffern)
        lngRest = (lngRest * 10 + CLng(Mid(strZiffern, lngPos, 1))) Mod 97
    Next lngPos

    Modulo97 = lngRest

End Function
```

Listing 2: Rest der Division durch 97 für beliebig lange Zahlen

```
Public Function IbanInZiffern(ByVal strIban As String) As String

    Dim strUmgestellt As String
    Dim lngPos As Long
    Dim lngCode As Long
    Dim strErgebnis As String

    ' Die ersten vier Zeichen wandern ans Ende
    strUmgestellt = Mid(strIban, 5) & Left(strIban, 4)

    For lngPos = 1 To Len(strUmgestellt)

        lngCode = Asc(Mid(strUmgestellt, lngPos, 1))

        If lngCode >= 48 And lngCode <= 57 Then
            strErgebnis = strErgebnis & Chr(lngCode)
        Else
            strErgebnis = strErgebnis & CStr(lngCode - 55)
        End If

    Next lngPos

    IbanInZiffern = strErgebnis

End Function
```

Listing 3: Umsetzen der IBAN in eine reine Ziffernfolge

Dass die Funktion auch mit Zahlen umgeht, die kein VBA-Datentyp mehr fassen kann, lässt sich leicht nachprüfen:

```
Debug.Print Modulo97("100")
Debug.Print Modulo97("12345678901234567890")
```

Ergebnis:

3
3

Dieses schrittweise Vorgehen lässt sich auf jeden Modulus übertragen – die Funktion selbst ist allerdings fest auf 97 programmiert.

Merken Sie sich das Muster unabhängig vom Thema Prüfziffern: Es ist immer dann Gold wert, wenn Sie mit Zahlen jenseits der VBA-Grenzen hantieren müssen.

Rezept 1: Die IBAN

Nun die Umsetzung der Zeichen in Ziffern. Buchstaben werden dabei durch ihre Position im Alphabet plus 9 ersetzt, also A durch 10 bis Z

durch 35. Da der Zeichencode von A bei 65 liegt, genügt eine Subtraktion von 55 (siehe Listing 3).

Damit lässt sich das Verfahren Schritt für Schritt nachvollziehen. Das D wird zu 13, das E zu 14, und der Rest muss am Ende 1 ergeben:

```
Debug.Print IbanInZiffern("DE89370400440532013000")
Debug.Print Modulo97( _
    IbanInZiffern("DE89370400440532013000"))
```

```
370400440532013000131489
```

```
1
```

Damit ist die eigentliche Prüffunktion kurz. Der Rest muss am Ende genau 1 betragen – das ist die Kernaussage des Verfahrens nach ISO 7064, das wir in der Funktion **IstIban** abgebildet haben (siehe Listing 4).

```
Public Function IstIban(ByVal strIban As String) As Boolean

    Dim strWert As String
    Dim lngLaenge As Long

    ' Erlaubt sind nur Buchstaben, Ziffern, Leerzeichen
    strWert = Bereinigt(strIban, "[A-Z0-9 ]+$")

    ' Grundstruktur pruefen
    If Not RegExTest(strWert, "[A-Z]{2}\d{2}[A-Z0-9]{11,30}$") Then
        Exit Function
    End If

    ' Land muss bekannt und die Laenge korrekt sein
    lngLaenge = IbanLaenge(Left(strWert, 2))

    If lngLaenge = 0 Then Exit Function
    If Len(strWert) <> lngLaenge Then Exit Function

    IstIban = (Modulo97(IbanInZiffern(strWert)) = 1)

End Function
```

Listing 4: Prüfen einer IBAN

Vier Aufrufe zeigen, welche Fehler die Funktion abfängt:

```
Debug.Print IstIban("DE89 3704 0044 0532 0130 00")
Debug.Print IstIban("DE89 3704 0044 0532 0130 01")
Debug.Print IstIban("DE89!3704 0044 0532 0130 00")
Debug.Print IstIban("XX89 3704 0044 0532 0130 00")
```

```
Wahr
Falsch
Falsch
Falsch
```

Der zweite Aufruf scheitert an der Prüfziffer, der dritte am eingeschmuggelten Ausrufezeichen und der vierte am unbekannten Ländercode.

Die Längenprüfung ist wichtiger, als sie aussieht. Ohne sie würde eine deutsche IBAN, bei der eine Ziffer fehlt, unter Umständen trotzdem als gültig durchgehen, weil das

Modulo-Verfahren die Länge nicht kennt. Die Zuordnung von Land zu Länge erledigt eine schlichte Fallunterscheidung, die wir in der Funktion **IbanLaenge** erledigen (siehe Listing 5).

Die Funktion liefert zum Beispiel folgende Ergebnisse:

```
Debug.Print IbanLaenge("DE")
Debug.Print IbanLaenge("CH")
Debug.Print IbanLaenge("US")
```

Ergebnisse:

```
22
21
0
```

Der Rückgabewert **0** bedeutet, dass das Land von der Funktion

nicht unterstützt wird. Die IBAN wird in diesem Fall abgelehnt, weil ihre vorgeschriebene Länge nicht geprüft werden kann – sonst könnte eine formal passende Nummer mit einem gar nicht vergebenen Ländercode als gültig durchgehen.

Die Liste deckt die für den deutschsprachigen Raum wichtigsten Länder ab. Insgesamt gibt es rund 80 IBAN-Länder; die vollständige und verbindliche Aufstellung pflegt die Organisation SWIFT im sogenannten IBAN-Register. Wenn Sie international arbeiten, ergänzen Sie die Liste entsprechend – oder lagern Sie in eine Tabelle aus.

Der umgekehrte Weg ist ebenso nützlich: aus Ländercode und der bereits zusammengesetzten nationalen Kontokennung, der BBAN, die vollständige IBAN erzeugen. Dazu setzt man wie in der Funktion **IbanErzeugen** aus Listing 6 zunächst zwei Nullen als Platzhal-

```
Public Function IbanLaenge(ByVal strLand As String) As Long

    Select Case UCase(strLand)
        Case "DE", "GB", "BG": IbanLaenge = 22
        Case "AT", "LT", "LU", "EE": IbanLaenge = 20
        Case "CH", "LI", "HR": IbanLaenge = 21
        Case "FR", "IT", "GR": IbanLaenge = 27
        Case "ES", "SE", "CZ", "SK", "RO": IbanLaenge = 24
        Case "NL", "DK", "FI": IbanLaenge = 18
        Case "BE": IbanLaenge = 16
        Case "NO": IbanLaenge = 15
        Case "SI": IbanLaenge = 19
        Case "PT": IbanLaenge = 25
        Case "TR": IbanLaenge = 26
        Case "PL", "HU": IbanLaenge = 28
        Case Else: IbanLaenge = 0
    End Select

End Function
```

Listing 5: Ermitteln der Länge einer IBAN

```
Public Function IbanErzeugen(ByVal strLand As String, ByVal strBban As String) As String

    Dim strRoh As String
    Dim lngLaenge As Long
    Dim lngRest As Long
    Dim strPruefziffern As String

    strLand = UCase(Bereinigt(strLand, "[A-Z]{2}$"))
    If Len(strLand) <> 2 Then Exit Function

    lngLaenge = IbanLaenge(strLand)
    If lngLaenge = 0 Then Exit Function
```

DAS WAR DIE LESEPROBE

Hier lesen Sie weiter – und finden noch viel mehr.

Den kompletten Beitrag und alle anderen Artikel dieser Ausgabe erhalten Sie im Abonnement von **Access im Unternehmen**: praxisnahe VBA-Lösungen, komplett nachvollziehbar erklärt, dazu die Beispieldatenbanken zum Download.

Jetzt Abonnement sichern >

andreminhorst.de/access-im-unternehmen

Fehlendes »Option Explicit« finden und nachrüsten

Zwei Wörter am Anfang eines Moduls ersparen Ihnen Stunden der Fehlersuche: **Option Explicit**. Wer sie vergisst, bekommt von VBA jeden Tippfehler in einem Variablennamen klaglos als neue Variable durchgewinkt – und einen Fehler, der sich erst Wochen später als falsches Ergebnis zeigt. In gewachsenen Anwendungen fehlt die Anweisung erfahrungsgemäß in einem Teil der Module. Dieser Beitrag zeigt, wie Sie die Lücken per VBA aufspüren und schließen, ohne jedes Modul einzeln zu öffnen.

Warum die Anweisung unverzichtbar ist

Ohne **Option Explicit** darf jede Variable ohne Deklaration verwendet werden. VBA legt sie beim ersten Auftreten stillschweigend als **Variant** an. Was das bedeutet, zeigt folgendes Listing – ein einziger fehlender Buchstabe genügt:

```
Public Sub OhneOptionExplicit()  
    Dim lngSumme As Long  
  
    lngSumme = 100  
    lngSume = lngSumme + 50    ' Tippfehler!  
  
    Debug.Print lngSumme  
End Sub
```

Die Ausgabe im Direktbereich lautet:

100

Erwartet hätte man 150. Stattdessen hat VBA die Variable **lngSume** neu angelegt, ihr das Ergebnis zugewiesen und sie danach nie wieder angesehen. Kein Fehler, keine Warnung – nur ein falscher Wert. Mit **Option Explicit** dagegen scheitert schon das Kompilieren mit der Meldung, dass die Variable nicht definiert ist.

Der Nebeneffekt ist ebenfalls bemerkenswert: Alle stillschweigend erzeugten Variablen sind vom Typ **Variant** und damit langsamer und

speicherhungriger als nötig. Es gibt schlicht keinen guten Grund, ohne diese Anweisung zu arbeiten.

Die Einstellung im VBA-Editor

Damit die Anweisung in neuen Modulen automatisch erscheint, gibt es im VBA-Editor eine Option. Sie finden sie unter **Extras/Optionen** auf der Registerseite **Editor** unter dem Namen **Variablendeklaration erforderlich** (siehe Bild 1). Ist sie aktiviert, setzt Access die Anweisung in jedes neu angelegte Modul.

Und genau hier liegt das Problem, das diesen Beitrag nötig macht: **Die Einstellung wirkt ausschließlich für Mo-**

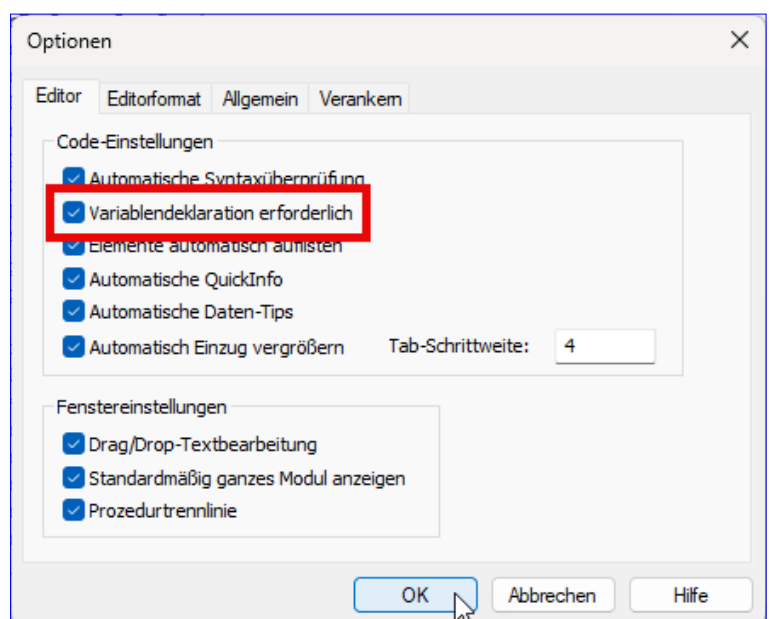


Bild 1: Die Option **Variablendeklaration erforderlich** im VBA-Editor

dule, die danach neu entstehen. Bestehende Module bleiben unangetastet. Wer die Option erst nach Jahren entdeckt – oder eine Datenbank übernimmt, in der sie nie aktiv war – hat einen bunt gemischten Bestand vor sich. Die Einstellung gilt außerdem pro Installation und nicht pro Datenbank; auf dem Rechner eines Kollegen kann sie also anders stehen.

Vorbereitung: Verweis setzen

Um Module per Code zu untersuchen, greifen wir auf das Objektmodell des VBA-Editors zu. Dazu benötigen wir einen Verweis auf die Bibliothek **Microsoft Visual Basic for Applications Extensibility 5.3 Object Library**, den wir wie in Bild 2 setzen.

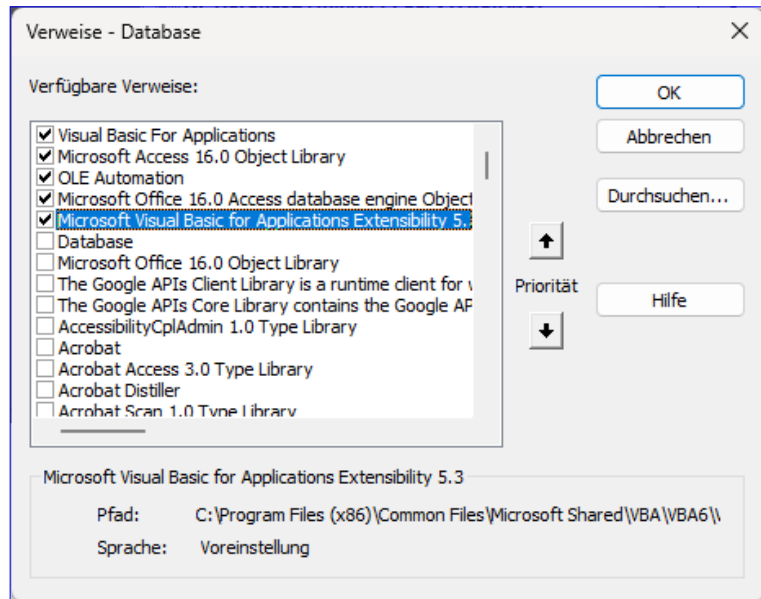


Bild 2: Setzen eines Verweises auf die Bibliothek **Microsoft Visual Basic for Applications Extensibility 5.3 Object Library**

Das eigene Projekt finden

Der VBA-Editor kann mehrere Projekte gleichzeitig geöffnet haben – etwa weitere Datenbanken oder Add-Ins. Wir brauchen deshalb eine Funktion, die gezielt das Projekt der aktuellen Datenbank herausucht. Der Vergleich läuft über den Dateinamen (siehe Listing 1).

Ein Test im Direktbereich zeigt, ob die Vorbereitungen geglückt sind:

```
Debug.Print GetCurrentVBProject.Name
```

```
Debug.Print GetCurrentVBProject.VBComponents.Count
```

```
Beispieldatenbank
```

```
14
```

Die Zahl umfasst dabei alle Standard- und Klassenmodule sowie die Klassenmodule von Formularen und Berichten. Letztere erscheinen unter Namen wie **Form_frmArtikel** – allerdings nur, wenn das Formular überhaupt ein Modul besitzt, seine Eigenschaft **Enthält Modul** also auf **Ja** steht.

```
Public Function GetCurrentVBProject() As VBIDE.VBProject
    Dim objVBProject As VBIDE.VBProject

    For Each objVBProject In Application.VBE.VBProjects

        If StrComp(objVBProject.FileName, CurrentProject.FullName, vbTextCompare) = 0 Then
            Set GetCurrentVBProject = objVBProject
            Exit Function
        End If

    Next objVBProject
End Function
```

Listing 1: Ermitteln des VBA-Projekts der aktuellen Datenbank

```
Public Function HasOptionExplicit(ByVal objCodeModule As VBIDE.CodeModule) As Boolean

    Dim lngLineNumber As Long
    Dim lngCommentPosition As Long
    Dim strLine As String

    For lngLineNumber = 1 To objCodeModule.CountOfDeclarationLines

        strLine = Trim$(objCodeModule.Lines(lngLineNumber, 1))

        ' Kommentar am Ende der Zeile entfernen
        lngCommentPosition = InStr(strLine, "'")
        If lngCommentPosition > 0 Then
            strLine = Trim$(Left$(strLine, lngCommentPosition - 1))
        End If

        If StrComp(strLine, "Option Explicit", vbTextCompare) = 0 Then
            HasOptionExplicit = True
            Exit Function
        End If

    Next lngLineNumber

End Function
```

Listing 2: Prüfen eines einzelnen Moduls

Ein Hinweis zur Fehlerbehandlung: Die Eigenschaft **FileName** löst einen Fehler aus, wenn ein Projekt noch nie gespeichert wurde. Kommen bei Ihnen mehrere Projekte ins Spiel, sollten Sie die Schleife mit **On Error Resume Next** absichern.

Prüfen, ob ein Modul »Option Explicit« enthält

Nun zur eigentlichen Prüfung, ob ein Modul die Anweisung **Option Explicit** enthält. Diese führen wir mit der Funktion **HasOptionExplicit** durch (siehe Listing 2). **Option Explicit** steht immer im Deklarationsbereich, also oberhalb der ersten Prozedur. Wie viele Zeilen dieser umfasst, verrät die Eigenschaft **CountOfDeclarationLines** – wir müssen also nicht das ganze Modul durchsuchen.

Zwei Details sind interessant. Die Methode **Lines** erwartet Startzeile und Anzahl, liefert hier also genau eine Zeile. Und der Vergleich mit **StrComp** und **vbTextCompare**

ignoriert die Groß- und Kleinschreibung – ein **option explicit** in Kleinbuchstaben wird also ebenfalls erkannt. Das Abschneiden am Apostroph sorgt dafür, dass auch eine Zeile mit angehängtem Kommentar richtig bewertet wird.

So sieht der Aufruf für ein bestimmtes Modul aus:

```
Dim objVBProject As VBIDE.VBProject
Set objVBProject = GetCurrentVBProject()
Debug.Print HasOptionExplicit(objVBProject. _
    VBComponents("mdlGarKeinOption").CodeModule)
```

Die Anweisung »Option Explicit« einfügen

Fehlt die Anweisung in einem Modul, tragen wir sie nach. Diese Aufgabe übernimmt die Prozedur **InsertOptionExplicit** aus Listing 3. Sie erwartet als einzigen Parameter das **CodeModule**-Objekt des Moduls, in dem **Option Explicit** ergänzt werden soll.

Die Prozedur arbeitet in zwei Schritten. Im ersten durchläuft die **For**-Schleife die Zeilen des Deklarationsbereichs, deren Anzahl die Eigenschaft **CountOfDeclarationLines** liefert.

Die Methode **Lines** holt die jeweilige Zeile in die Variable **strLine**, wobei **Trim\$** führende Leerzeichen entfernt. Beginnt die Zeile mit **Option Compare**, merkt sich die Prozedur die Zeilennummer in **lngOptionCompareLine** und verlässt die Schleife mit **Exit For**.

Im zweiten Schritt entscheidet der Wert dieser Variablen, wo die neue Zeile landet. Steht dort noch die **0**, gibt es im Modul keine **Option Compare**-Anweisung, und **InsertLines** setzt **Option Explicit** in Zeile **1**. Andernfalls übergeben wir **InsertLines** die gemerkte Zeilennummer.

Dabei ist zu beachten, dass **InsertLines** die neue Zeile **vor** der angegebenen Zeilennummer einfügt. **Option Explicit**

steht anschließend also über **Option Compare Database**. Für die Funktion ist das gleichgültig, denn die Reihenfolge der Option-Anweisungen spielt keine Rolle. Wer die gewohnte Schreibweise bevorzugt, übergibt stattdessen **lngOptionCompareLine + 1**.

Für einen Test rufen wir die Prozedur mit dem **CodeModule** eines einzelnen Moduls auf und lassen uns danach die ersten Zeilen des Deklarationsbereichs ausgeben:

```
Dim objCodeModule As VBIDE.CodeModule
Set objCodeModule = GetCurrentVBProject(). _
    VBComponents("mdlImport").CodeModule
InsertOptionExplicit objCodeModule
Debug.Print objCodeModule.Lines(1, 2)
```

Alle Module in einem Durchgang nachrüsten

Damit sind alle Bausteine beisammen. Die Prozedur **AddOptionExplicitToAllModules** aus Listing 4 setzt sie

```
Public Sub InsertOptionExplicit(ByVal objCodeModule As VBIDE.CodeModule)
    Dim lngLineNumber As Long
    Dim lngOptionCompareLine As Long
    Dim strLine As String

    For lngLineNumber = 1 To objCodeModule.CountOfDeclarationLines

        strLine = Trim$(objCodeModule.Lines(lngLineNumber, 1))

        If StrComp(Left$(strLine, 14), "Option Compare", vbTextCompare) = 0 Then
```

DAS WAR DIE LESEPROBE

Hier lesen Sie weiter – und finden noch viel mehr.

Den kompletten Beitrag und alle anderen Artikel dieser Ausgabe erhalten Sie im Abonnement von **Access im Unternehmen**: praxisnahe VBA-Lösungen, komplett nachvollziehbar erklärt, dazu die Beispieldatenbanken zum Download.

Jetzt Abonnement sichern >

andreminhorst.de/access-im-unternehmen

Rechnen mit Datum und Zeit ohne Überraschungen

Datumsberechnungen wirken harmlos, bis dann die Abfrage die Datensätze von heute nicht findet, die Kalenderwoche zum Jahreswechsel danebenliegt oder die Summe der Arbeitszeiten plötzlich 07:59 statt 08:00 lautet. Alle drei Fälle haben dieselbe Ursache: Access speichert Datum und Uhrzeit anders, als die angezeigte Darstellung vermuten lässt. Dieser Beitrag zeigt, was dabei schiefgehen kann, und liefert die Funktionen, die Ihnen die üblichen Fallen abnehmen.

Wie Access Datum und Zeit speichert

Ein Wert des Datentyps **Date** wird intern als 64-Bit-Fließkommazahl gespeichert. Für die in heutigen Anwendungen üblichen Datumswerte ab dem 30. Dezember 1899 zählt der Ganzzahlanteil die Tage seit diesem Ausgangsdatum, der Nachkommaanteil gibt die Tageszeit als Bruchteil eines Tages an. Die folgende Prozedur macht das sichtbar:

```
Public Sub DatumAlsZahl()
```

```
    Debug.Print CDb1(#4/5/2026#)
    Debug.Print CDb1(#4/5/2026 12:00:00 PM#)
    Debug.Print CDb1(#4/5/2026 6:00:00 AM#)
    Debug.Print CDb1(#12/30/1899#)
```

```
End Sub
```

Die Ausgabe im Direktbereich:

```
46117
46117,5
46117,25
0
```

Der 5. April 2026 ist also der Tag Nummer 46117, 12 Uhr mittags entspricht einem halben Tag, 6 Uhr morgens einem Viertel. Aus dieser Darstellung folgen unmittelbar zwei nützliche Rechenregeln: Ein Tag mehr heißt schlicht **+ 1**, und die Differenz zweier Datumswerte, multipliziert mit 24, ergibt die Stundenzahl dazwischen. Beachten Sie die Datumsliterale im Code. Zwischen den Rautezeichen

erwartet VBA das amerikanische Format mit Monat vor Tag. Schreiben Sie **#5.4.2026#**, wandelt der Editor das stillschweigend in **#5/4/2026#** um – und das ist der 4. Mai, nicht der 5. April.

Der versteckte Zeitanteil

Genau hier liegt die häufigste Fehlerquelle überhaupt. Ein Feld vom Typ **Datum/Uhrzeit**, das mit **Now** gefüllt wurde, enthält eben auch die Uhrzeit. Die folgende Prozedur führt vor, was das für einen Vergleich bedeutet:

```
Public Sub ZeitanteilFalle()
```

```
    Dim datGespeichert As Date
```

```
    datGespeichert = #4/5/2026 9:37:12 AM#
```

```
    Debug.Print datGespeichert = #4/5/2026#
    Debug.Print Int(datGespeichert) = #4/5/2026#
    Debug.Print DateValue(datGespeichert) = #4/5/2026#
```

```
End Sub
```

Hier ist die Ausgabe dieser Prozedur:

```
Falsch
Wahr
Wahr
```

Der erste Vergleich scheitert, weil 46117,4008 nun einmal nicht 46117 ist. Für aktuelle, positive Datumswerte

schneiden sowohl **Int** als auch **DateValue** den Zeitanteil ab; **DateValue** drückt die fachliche Absicht deutlicher aus. In einer Abfrage sollten Sie allerdings keine der beiden Funktionen auf das Feld anwenden, weil Access dann für jeden Datensatz rechnen muss und den Index nicht mehr nutzen kann. Formulieren Sie stattdessen einen Bereich, wie ihn die folgende Abfrage zeigt:

```
SELECT *
FROM tblAuftraege
WHERE Erfasst >= #2026-04-05#
      AND Erfasst < #2026-04-06#
```

Die obere Grenze steht bewusst auf **kleiner als der Folgetag** und nicht auf **Between**. Ein **Between #2026-04-05# And #2026-04-06#** würde die Datensätze vom 6. April um Punkt Mitternacht mit einschließen, weil **Between** an beiden Enden einschließend arbeitet.

Datumsangaben in SQL-Ausdrücke einbauen

Setzen Sie eine solche Bedingung per VBA zusammen, brauchen Sie das Datum als Zeichenkette. Ein schlichtes Verketteten mit dem Datumswert ist problematisch, weil VBA dabei gewöhnlich ein von den Regionaleinstellungen

abhängiges Datumsformat erzeugt und Access den 5. April dann als 4. Mai liest. Die Funktion **SqlDatum** aus Listing 1 nimmt Ihnen die Umwandlung ab.

Die Funktion erzeugt das ISO-Format mit Jahr, Monat und Tag, das Access unabhängig von den Ländereinstellungen richtig auswertet. Die umgekehrten Schrägstriche im Formatausdruck sorgen dafür, dass Raute, Bindestrich und Doppelpunkt als Zeichen ausgegeben und nicht als Formatanweisung gedeutet werden. Beim Doppelpunkt ist das zwingend, denn ohne Maskierung setzt VBA dort das landestypische Zeitzeichen ein.

Die Prüfung mit **IsDate** weist ungeeignete Werte bereits in der Hilfsfunktion mit einer verständlichen Fehlermeldung zurück. Ohne sie liefe eine versehentlich übergebene Zeichenkette entweder in einen Fehler bei **CDate** oder in einen SQL-Ausdruck, dessen Ursache sich später nur mühsam zurückverfolgen lässt.

Ein **Null**-Wert wird zum SQL-Schlüsselwort **Null**. Das ist bei Wertzuweisungen verwendbar, etwa in einer Aktualisierungsabfrage. In Vergleichsbedingungen muss ein Nullwert dagegen mit **Is Null** beziehungsweise **Is Not Null**

```
Public Function SqlDatum(ByVal varDatum As Variant, Optional ByVal blnMitZeit As Boolean = False) As String

    If IsNull(varDatum) Then
        SqlDatum = "Null"

    ElseIf Not IsDate(varDatum) Then
        Err.Raise 13, "SqlDatum", "Der übergebene Wert ist kein gültiges Datum."

    ElseIf blnMitZeit Then
        SqlDatum = Format$(CDate(varDatum), "\#yyyy\~mm\~dd hh\:nn\:ss\#")

    Else
        SqlDatum = Format$(CDate(varDatum), "\#yyyy\~mm\~dd\#")
    End If

End Function
```

Listing 1: **SqlDatum** liefert ein Datum in der von Access erwarteten Schreibweise

geprüft werden – ein **WHERE Lieferdatum = Null** liefert grundsätzlich keine Treffer.

```
Debug.Print SqlDatum(#4/5/2026#)
Debug.Print SqlDatum(#4/5/2026 9:37:12 AM#, True)
Debug.Print SqlDatum(Null)
```

```
#2026-04-05#
#2026-04-05 09:37:12#
Null
```

Damit lässt sich der Tagesbereich aus der obigen Abfrage korrekt zusammensetzen:

```
Public Sub Tagesbereich()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim strSQL As String
    Dim datTag As Date

    datTag = #4/5/2026#
    strSQL = "SELECT * FROM tblAuftraege WHERE Erfasst >= " & _
        & SqlDatum(datTag) & " AND Erfasst < " & _
        & SqlDatum(datTag + 1)
    Set db = CurrentDb
    Set rst = db.OpenRecordset(strSQL, dbOpenDynaset)
    Do While Not rst.EOF
        Debug.Print rst.Fields(0), rst.Fields(1)
        rst.MoveNext
    Loop
End Sub
```

Noch robuster sind parametrisierte Abfragen. Dabei übergibt DAO den Datumswert typisiert, sodass weder das regionale Format noch die Schreibweise des Datums literals eine Rolle spielt. Folgendes Listing zeigt, wie das aussieht:

```
Public Sub TagesbereichParametrisiert()
    Dim db As DAO.Database
    Dim qdf As DAO.QueryDef
    Dim rst As DAO.Recordset
```

```
Dim datTag As Date

datTag = #4/5/2026#
Set db = CurrentDb
Set qdf = db.CreateQueryDef("", _
    "PARAMETERS pVon DateTime, pBis DateTime; " & _
    "SELECT * FROM tblAuftraege " & _
    "WHERE Erfasst >= pVon AND Erfasst < pBis")

qdf.Parameters("pVon") = datTag
qdf.Parameters("pBis") = DateAdd("d", 1, datTag)

Set rst = qdf.OpenRecordset
Do While Not rst.EOF
    Debug.Print rst.Fields(0), rst.Fields(1)
    rst.MoveNext
Loop

End Sub
```

Der leere Name im Aufruf von **CreateQueryDef** erzeugt eine temporäre Abfrage, die nicht im Navigationsbereich erscheint. Die beiden Parameter werden mit **DateTime** deklariert und anschließend als echte Datumswerte zugewiesen – eine Zeichenkette entsteht dabei an keiner Stelle.

Die Kalenderwoche nach ISO 8601

Die eingebaute Funktion **DatePart** kennt das Intervall **ww** und liefert damit eine Kalenderwoche. Mit den zusätzlichen Argumenten **vbMonday** und **vbFirstFourDays** legt man den Montag als ersten Wochentag und die erste Woche mit mindestens vier Tagen als erste Kalenderwoche fest. Beide Angaben sind für die Berechnung nach ISO 8601 erforderlich. Ohne sie beginnt die Woche am Sonntag und die erste Woche des Jahres mit dem 1. Januar.

Allerdings besitzt **DatePart** einen bekannten Grenzfall: Für den letzten Montag bestimmter Jahre kann die Funktion bereits die Woche 53 statt der ISO-Woche 1 des Folgejahres liefern. Die nachfolgend vorgestellte Berechnung über

den Donnerstag der jeweiligen Woche vermeidet auch dieses Problem.

Was **DatePart** dagegen nicht liefert, ist das Wochenjahr. Und genau daran können Auswertungen scheitern, wie die folgende Prozedur zeigt:

```
Public Sub KalenderwocheFalle()  
  
    Debug.Print Year(#12/31/2007#) & "-KW" _  
        & DatePart("ww", #12/31/2007#, _  
        vbMonday, vbFirstFourDays)  
    Debug.Print Year(#1/1/2021#) & "-KW" _  
        & DatePart("ww", #1/1/2021#, _  
        vbMonday, vbFirstFourDays)  
  
End Sub
```

Hier ist das Ergebnis der Prozedur:

```
2007-KW1  
2021-KW53
```

Beide Angaben sind falsch. Der 31. Dezember 2007 gehört zur Woche 1 des Jahres **2008**, der 1. Januar 2021 zur Woche 53 des Jahres **2020**. Die Wochennummern stimmen also – nur das Jahr passt nicht dazu, weil **Year** das Kalenderjahr liefert und nicht das Wochenjahr. Ein Jahr 2021 mit einer Woche 53 gibt es überhaupt nicht.

In der Praxis fällt das erst auf, wenn eine Auswertung über den Jahreswechsel läuft. Eine Gruppierung nach Jahr und Woche erzeugt dann Gruppen, die es nicht gibt, und verteilt die Datensätze einer einzigen Woche auf zwei Zeilen.

Die Norm löst das über eine einfache Regel: Eine Woche gehört zu dem Jahr, in dem ihr Donnerstag liegt. Damit lassen sich Wochennummer und Wochenjahr aus derselben Rechnung ableiten. Die Funktion **IsoKalenderwoche** aus Listing 2 übernimmt den ersten Teil.

Die Funktion ermittelt zuerst den Donnerstag der Woche, in der das übergebene Datum liegt. **Weekday** mit dem Parameter **vbMonday** liefert für Montag die 1, für Sonntag die 7; die Rechnung **Datum minus Wochentag plus 4** landet damit immer auf dem Donnerstag.

Der Aufruf von **DateValue** entfernt einen etwaigen Zeitanteil, der für die Kalenderwoche ohne Bedeutung ist. Anschließend ermittelt sie die Position dieses Donnerstags innerhalb des Jahres, teilt durch 7 und addiert 1.

Das Wochenjahr fällt dabei nebenbei ab: Es ist schlicht das Jahr des Donnerstags. Die Funktion **IsoKalenderjahr** aus Listing 3 gibt es zurück.

Für die beiden Problemfälle von oben und ein weiteres Beispiel sieht das nun so aus:

DAS WAR DIE LESEPROBE

Hier lesen Sie weiter – und finden noch viel mehr.

Den kompletten Beitrag und alle anderen Artikel dieser Ausgabe erhalten Sie im Abonnement von **Access im Unternehmen**: praxisnahe VBA-Lösungen, komplett nachvollziehbar erklärt, dazu die Beispieldatenbanken zum Download.

Jetzt Abonnement sichern >

andreminhorst.de/access-im-unternehmen