Fehler und Ausnahmen behandeln

Fehlerfreie Programme 6.1

Schreiben Sie einfach ein völlig fehlerfreies Programm! Gut, nachdem wir jetzt alle einmal herzlich gelacht haben, können wir uns der zweitbesten Lösung zuwenden: Schreiben Sie ein Programm, das sich wenigstens um seine Fehler kümmert.

Dabei geht es hier vor allem um Laufzeitfehler. Diese treten, wie der Name schon sagt, erst bei der Ausführung des Programms auf, bei dem ansonsten syntaktisch alles in Ordnung ist. Einen typischen Fall zeigt der folgende Code, der den Namen der aktuellen Datenbank anzeigen soll.

```
Sub DBNameZeigen()
   MsgBox CurrentDb.Name
End Sub
```

Sie glauben, mit so einer einfachen Zeile sei kaum ein Fehler zu machen? Doch! Noch viel lästiger ist: je intensiver Sie solchen Code testen, desto sicherer werden Sie die Fehlerquelle übersehen.

Sobald Sie nämlich im VBA-Editor arbeiten, muss die zugehörige Datenbank-Datei wirklich sichtbar geöffnet sein. Dann gibt es auch eine `CurrentDb`. Ist dieser Code jedoch in einem AddIn (welches es auch für Access-Datenbanken gibt) enthalten, könnte er ausgeführt werden, ohne dass eine Datenbank als `CurrentDb` offen ist.

Solcher Code ist trotz seiner Kürze schwierig zu testen, weil Sie dafür laufend ein AddIn erzeugen und einbinden müssten. Daher finden Sie nachfolgend ein Beispiel, das nur dazu dient, solche Laufzeitfehler zu provozieren und untersuchen zu können.

```
Sub FehlerProvozieren()
   Dim intA As Integer
   Dim intB As Integer
   intA = InputBox("Erste Zahl?")
   intB = InputBox("Zweite Zahl?")
   MsgBox "Ergebnis: " & intA / intB
End Sub
```

Lassen Sie das Makro mit F5 laufen und geben beispielsweise 12 und 7 ein, so werden Sie feststellen, dass es fehlerfrei läuft und das Ergebnis 1,714… sogar mit Nachkommastellen korrekt angezeigt wird.

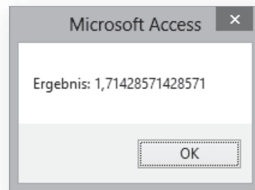


Abbildung 6.1: Das Ergebnis der Prozedur bei korrektem Programmablauf

Fragen Sie die richtige Datenbank

Hier sehen Sie eine Lösung für Fälle, in denen immer der Name der Datenbank gezeigt werden soll, in der auch das Modul enthalten ist. Das ist im Falle eines AddIns der Name der AddIn-Datenbank, ansonsten die geöffnete Datenbank.

```
Sub DBNameZeigenVonDieser()
   MsgBox CodeDb.Name
End Sub
```

Während `CurrentDb` die vom Benutzer geöffnete Datenbank meint, ist `CodeDb` diejenige, in der sich der Code befindet.

Fehler provozieren

Nun war ja Sinn dieses Makros, Laufzeitfehler erzwingen zu können. Ein solcher Fehler muss auftreten, wenn Sie eine mathematisch verbotene Division durch 0 vornehmen. Geben Sie für die zweite Zahl 0 ein, erhalten Sie folgende Fehlermeldung und der Code unterbricht vor der Ausführung der `MsgBox`-Zeile.

Abbildung 6.2: Die Fehlermeldung für Laufzeitfehler

Sie erkennen dies daran, dass die Zeile gelb hinterlegt ist. Wenn Sie die Programmausführung endgültig stoppen wollen, müssen Sie im Dialogfeld auf *Beenden* klicken.

Hier wäre es selbstverständlich ein Leichtes, mit `If intB <> 0 Then` die `MsgBox`-Zeile nicht immer auszuführen. Das geht aber erstens nicht für alle Laufzeitfehler und zweitens soll es ja gerade der Sinn dieser Prozedur sein, zu Testzwecken einen einfachen Laufzeitfehler erzeugen zu können.

Fehlerbehandlung ankündigen

Sobald Sie die eben gesehene VBA-interne Fehlermeldung durch eine eigene Fehlerbehandlung ersetzen wollen, müssen Sie dies vorher ankündigen. Während sich Visual Basic ansonsten vom ursprünglichen einfachen BASIC aus den 60er Jahren zu einer modernen, ziemlich objektorientierten Hochsprache gewandelt hat, ist der Umgang mit Laufzeitfehlern leider in der digitalen Steinzeit stehen geblieben.

Bevor der Fehler im Code auftritt müssen Sie nämlich mit `On Error Goto BIstNull` einstellen, dass das Programm im Fehlerfall nicht hängen bleibt. Stattdessen geht es dann zur Sprungmarke mit dem frei erfundenen Namen `BIstNull`. Jede Sprungmarke muss mit einem Doppelpunkt abgeschlossen werden und steht unweigerlich am Anfang einer Zeile. Damit sieht die Prozedur aus wie im folgenden Code.

```
Sub FehlerProvozieren()
   Dim intA As Integer
   Dim intB As Integer
   intA = InputBox("Erste Zahl?")
   On Error GoTo BIstNull
   intB = InputBox("Zweite Zahl?")
   MsgBox "Ergebnis: " & intA / intB
BIstNull:
   MsgBox "Hoppla, ein Fehler!"
End Sub
```

Der Fehler ist nun zwar noch nicht wirklich behoben, aber Sie können jetzt wenigstens mit eigenem Code darauf reagieren.

Abbildung 6.3: Der Fehler erzeugt eine eigene Fehlermeldung

Sprungmarken

Sprungmarken sind eigentlich noch die alten Zeilennummern im neuen Gewand. In den Anfangszeiten von BASIC lasen sich die Programme so:

```
Sub UraltBeispiel()
10:    intA$ = InputBox("Erste Zahl?")
20:    On Error GoTo 100
30:    intB$ = InputBox("Zweite Zahl?")
40:    MsgBox "Ergebnis: " + intA$ / intB$
100:   MsgBox "Hoppla, ein Fehler!"
End Sub
```

Vor jeder Zeile stand eine Zeilennummer mit folgendem Doppelpunkt, die oft in Zehner-Schritten gezählt wurde, damit für zwischengeschobene Zeilen noch Reserve war. Sonst musste eben neu durchnummeriert werden.

Die heutigen, fein differenzierten Datentypen gab es auch noch nicht. Mit dem Dollarzeichen wurden Zeichenketten-Variablen gekennzeichnet, die anderen waren Zahlenvariablen. Texte wurden ohne Bedenken mit dem +-Zeichen »addiert«, eine Technik, die heute noch in manchem Code zu sehen ist.

Fehlerbehandlung ausgrenzen

Während Sie diesen Fehler jetzt vielleicht ausführlich testen, sollten Sie nicht vergessen, auch den fehlerfreien Fall nochmals zu prüfen. Sie werden möglicherweise überrascht sein, auch hierbei Ihre »Fehler«-Meldung zu erhalten, denn eine solche Sprungmarke ist keineswegs abweisend.

Sie müssen vorher explizit die Anweisung `Exit Sub` (oder im Falle einer Funktion natürlich `Exit Function`) in den Code aufnehmen, damit dieser Teil im Normalfall nicht ausgeführt wird. Das ist im nächsten Code bereits berücksichtigt.

Verschiedene Fehler behandeln

Nun wäre es allzu schön, wenn immer nur ein einziger Fehler aufträte. In diesem Beispiel etwa ist auch die Eingabe einer zu großen Zahl verboten, denn `Integer`-Variablen erlauben maximal 32.767 als größten Wert.

Am übersichtlichsten ermitteln Sie mittels `Select Case`, welcher Fehler aufgetreten ist, denn diese sind sprachneutral durchnummeriert. Die `Number`-Eigenschaft des `Err`-Objekts kennt diesen eindeutigen Fehlercode, die `Description`-Eigenschaft liefert automatisch eine Beschreibung in der Landessprache Ihrer Access-Version.

Der folgende Code zeigt deren Einsatz vor allem für den Fall, dass vielleicht noch ein übersehener weiterer Fehler auftreten kann. Ohne die `Case Else`-Anweisung würde dieser regelrecht verschwiegen und Ihre Prozedur einfach ohne weitere Reaktion enden.

```
Sub FehlerProvozieren()
   Dim intA As Integer
   Dim intB As Integer
   intA = InputBox("Erste Zahl?")
   On Error GoTo BIstFalsch
   intB = InputBox("Zweite Zahl?")
   MsgBox "Ergebnis: " & intA / intB
   Exit Sub
BIstFalsch:
```

```
    Select Case Err.Number
    Case 6: MsgBox "Zweite Zahl größer als 32.767!"
    Case 11: MsgBox "Zweite Zahl darf nicht 0 sein!"
    Case Else: MsgBox "Fehler Nr. " & Err.Number & " aufgetreten: " & Err.Description
    End Select
End Sub
```

Fehler an mehreren Stellen

Die Fehlerquellen für `intB` mögen damit ausführlich behandelt worden sein, nicht jedoch diejenigen für `intA`. Zwar gelten auch hier die Werte-Grenzen für Integer, aber diesmal ist die 0 als Eingabe zulässig. Also muss eine eigene Fehlerbehandlung stattfinden, die rechtzeitig angekündigt und mit einer getrennten Sprungmarke abgewickelt wird, wie hier zu sehen.

```
Sub FehlerProvozieren()
    Dim intA As Integer
    Dim intB As Integer
    On Error GoTo AIstFalsch
    intA = InputBox("Erste Zahl?")
    On Error GoTo BIstFalsch
    intB = InputBox("Zweite Zahl?")
    MsgBox "Ergebnis: " & intA / intB
    Exit Sub
AIstFalsch:
    Select Case Err.Number
    Case 6: MsgBox "Erste Zahl größer als 32.767!"
    Case Else: MsgBox "Fehler Nr. " & Err.Number & " aufgetreten: " & Err.Description
    End Select
BIstFalsch:
    Select Case Err.Number
    Case 6: MsgBox "Zweite Zahl größer als 32.767!"
    Case 11: MsgBox "Zweite Zahl darf nicht 0 sein!"
    Case Else: MsgBox "Fehler Nr. " & Err.Number & " aufgetreten: " & Err.Description
    End Select
End Sub
```

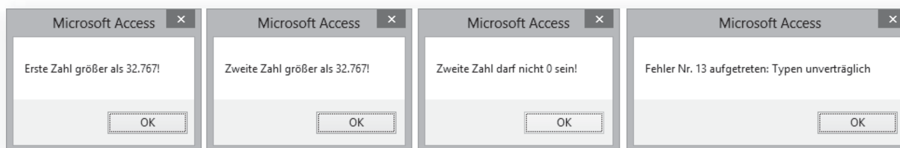


Abbildung 6.4: Verschiedene Fehler lösen die jeweils passende Fehlermeldung aus

Sie ahnen an dieser Stelle sicherlich schon, dass die Fehlerbehandlung möglicherweise den größten Teil Ihres Codes beanspruchen wird, wenn Sie das genau so handhaben wollen.

Fehler beheben

Für den Benutzer ist es frustrierend, lediglich einen Fehler gemeldet zu bekommen, diesen aber nicht beheben zu können. Dabei ist es mit minimalem Aufwand möglich, die Neueingabe der falschen Zahl zu erlauben. Da Sie wissen, welche Zahl jeweils falsch ist, kann die Ausführung des Codes von der Fehlerbehandlung wieder zur passenden `InputBox()`-Zeile führen. Auch dazu braucht es Sprungmarken, zu denen Ihr Code mit `Resume`-Anweisungen verzweigt wie hier:

```
Sub FehlerProvozieren()
   Dim intA As Integer
   Dim intB As Integer
AWiederholen:
   On Error GoTo AIstFalsch
   intA = InputBox("Erste Zahl?")
BWiederholen:
   On Error GoTo BIstFalsch
   intB = InputBox("Zweite Zahl?")
   MsgBox "Ergebnis: " & intA / intB
   Exit Sub
AIstFalsch:
   Select Case Err.Number
   Case 6
      MsgBox "Erste Zahl größer als 32.767!"
      Resume AWiederholen
   Case Else
      MsgBox "Fehler Nr. " & Err.Number & " aufgetreten: " & Err.Description
   End Select
BIstFalsch:
   Select Case Err.Number
   Case 6
      MsgBox "Zweite Zahl größer als 32.767!"
      Resume BWiederholen
   Case 11
      MsgBox "Zweite Zahl darf nicht 0 sein!"
      Resume BWiederholen
   Case Else
      MsgBox "Fehler Nr. " & Err.Number & " aufgetreten: " & Err.Description
   End Select
End Sub
```

Spaghetti-Code

So wie im vorigen Beispiel sieht er aus: der berüchtigte »Spaghetti-Code«. Je nachdem, welcher Fehler auftritt und wie er behandelt wird, läuft die Programmausführung nicht mehr linear die Zeilen entlang, sondern springt im Code wild von oben nach unten. Das ist genau die Art schlechte Programmierung, die schon in alten BASIC-Zeiten mit ihren vielen `GoTo`-Anweisungen verpönt war. Das Programm läuft zwar, aber keiner versteht mehr warum.

Nebenbei ist die Prozedur von zuvor 5 Zeilen auf jetzt 34 Zeilen aufgebläht worden. Solcher Code lässt sich nicht mehr ernsthaft weiterentwickeln, weil er völlig unübersichtlich wird.

Fehler ignorieren

Sie können das jedoch deutlich vereinfachen, wenn Sie einen Fehler gar nicht behandeln, sondern nur die Meldung unterdrücken wollen. Das ist beispielsweise der Fall, wenn eine Datei beim Programmende gelöscht sein soll. Ist sie beim Löschen nicht vorhanden, tritt ein Laufzeitfehler auf, der mit `On Error Resume Next` ignoriert werden kann:

```
Sub DateiLoeschen()
   On Error Resume Next
   Kill "c:\abcdef.xyz"
End Sub
```

Im Normalfall dürfte es Ihnen ja egal sein, ob die Datei fehlerfrei gelöscht wird oder beim Löschversuch der Fehler auftritt, dass die Datei gar nicht vorhanden war. In beiden Fällen ist sie anschließend weg.

Fehler-Hierarchie

Leider gibt es noch genug Überraschungen, wenn die Fehlerbehandlung selbst ganz einfach bleibt. VBA leitet die Fehlerbehandlung nämlich notfalls an übergeordnete Prozeduren weiter, obwohl in der aktuellen Prozedur gar keine Fehlerbehandlung stattfinden soll, wie der folgende Code zeigt:

```
Sub GanzOben()
   On Error GoTo Mist
   InDerMitteA
   InDerMitteB
   Exit Sub
Mist:
   MsgBox "Ein Fehler!"
End Sub

Sub InDerMitteA()
   Unten1
```

```
End Sub

Sub InDerMitteB()
   Unten2
End Sub

Sub Unten1()
   Dim intTest As Integer
   intTest = 99
End Sub

Sub Unten2()
   Dim intTest As Integer
   intTest = 123456            'hier tritt ein Laufzeitfehler auf!
End Sub
```

Wenn Sie nun `GanzOben` mit F5 ausführen, gibt es in `Unten2` einen Laufzeitfehler wegen des Überlaufs einer `Integer`-Variablen. Da Sie in `GanzOben` aber die eigene Fehlerbehandlung eingeschaltet hatten, ist dies in `Unten2` noch aktiv. Dort gibt es jedoch keine `On Error`-Anweisungen, also springt der Code zurück in die Prozedur `InDerMitteB`. Auch dort fehlt die Fehlerbehandlung, daher springt VBA erneut zurück in dessen aufrufende Prozedur `GanzOben`.

Sie können sich sicher ungefähr vorstellen, wie schwierig es wird, einen solchen Fehler über mehrere Prozeduren hinweg wirklich zu behandeln. Vor allem können Sie nicht aus der Fehlerbehandlung von `GanzOben` zur problematischen Zeile in `Unten2` zurückspringen!

Eigene Fehlerbehandlung wieder ausschalten

Mit `On Error Goto 0` können Sie die Fehlerbehandlung jederzeit wieder an VBA zurückgeben. Danach meldet sich bei Problemen wieder der VBA-eigene Dialog für Laufzeitfehler. Da sieht man mit der `0` noch ganz deutlich die ursprünglichen Zeilennummern des BASIC hindurchschimmern …

Alternativen

Gibt es dazu Alternativen? Ja und nein. Wenn Sie nur für einen kleinen Nutzerkreis programmieren, sollten Sie durchaus überlegen, gar keine Fehlerbehandlung einzubauen.

Da sich das nicht überall vermeiden lässt, sollten Sie wenigstens auf den Spaghetti-Code verzichten. Das geht, indem Sie bei Fehlern einfach im Code weiterlaufen, dort aber die Fehlernummer abfragen, wie in dem nachfolgenden stark vereinfachten Code beispielhaft zu sehen ist.

```
Sub DateiLoeschen()
   '...vorheriger Code
```

```
    On Error Resume Next
    Kill "c:\abcdef.xyz"              'löscht die Datei, falls vorhanden
    Select Case Err.Number
    Case 53
       'Datei war nicht da, kann also ignoriert werden!
    Case Is <> 0
       MsgBox "Fehler Nr. " & Err.Number & ": " & Err.Description, vbCritical
    End Select
    On Error GoTo 0
    '...weiterer Code
End Sub
```

Dadurch lässt sich vor allem das Hin- und Herspringen der Codeausführung vermeiden und eine Fehlerbehandlung in den übersichtlicheren Block-Strukturen erledigen.

Zentrale Fehlerbehandlung

Wenn es sich gar nicht vermeiden lässt, können Sie versuchen, die Fehlerbehandlung wenigstens einigermaßen zentral zu verarbeiten. Leider sind Ihnen da enge Grenzen gesetzt, weil eine `Resume`-Anweisung immer in der Prozedur stehen muss, in der sich auch das zugehörige Label befindet.

1. Erstellen Sie zur Optimierung eine allgemeine Funktion `WieWeiterMitFehler()`, die den Benutzer fragt, was nun weiter geschehen soll:

```
Function WieWeiterMitFehler(strProzName As String, strModulName As String) As
VbMsgBoxResult
   WieWeiterMitFehler = MsgBox("Fehler Nr. " & Err.Number & ": " & _
      Err.Description & vbCrLf & vbCrLf & _
      "Wollen Sie ganz abbrechen, den Code wiederholen oder einfach ignorieren?", _
        vbAbortRetryIgnore + vbDefaultButton2, _
        "Fehler in '" & strProzName & "' im Modul '" & strModulName &"'")
End Function
```

2. Diese Funktion zeigt einen Standardtext mit der Fehlerbeschreibung sowie einer Entscheidungsmöglichkeit die aufgerufenen Prozeduren ganz zu beenden, die fehlerhafte Stelle erneut auszuführen oder einfach weiterzumachen. Die Wiederholung der Zeile mit dem Fehler ist natürlich nur dann sinnvoll, wenn der Benutzer zwischenzeitlich etwas verbessern konnte, etwa eine Datei umbenennen oder in das richtige Verzeichnis kopieren.

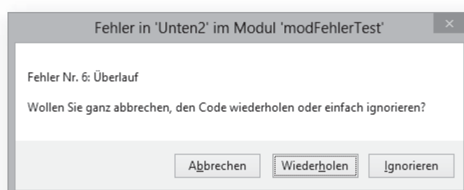


Abbildung 6.5: Die eigene, zentrale Fehlermeldung

3. Damit die Funktion `WieWeiterMitFehler()` den Namen der betroffenen Prozedur nennen kann, muss dieser als Parameter übergeben werden, eine automatische Anzeige ist in VBA nicht möglich. Dazu müssen Sie in jeder(!) Prozedur eine Konstante mit deren Namen deklarieren, damit der Aufruf einheitlich ist.

4. Auch der Modulname lässt sich leider nicht automatisch ermitteln, dafür schreiben Sie einfach am Anfang aller Module eine Modul-öffentliche Konstante, auf die Sie als Parameter zugreifen können:

```
Const m_cstrModulName = "modFehlerTest"    'hier den Namen des Moduls angeben
```

5. Dann müssen Sie alle(!) Prozeduren gleichermaßen mit einer Fehlerbehandlung `On Error Goto Mist` und der lokalen Konstanten `cstrProzName` ausstatten. Da die eigentlichen Parameter der `WieWeiterMitFehler()`-Funktion Konstanten sind, können Sie alles ab `Exit Sub` direkt in jede Prozedur identisch hineinkopieren. Die Prozedur `Unten2` ändert sich damit beispielsweise wie folgt:

```
Sub Unten2()
   Dim intTest As Integer
   Const cstrProzName = "Unten2"
   On Error GoTo Mist
   intTest = 123456      'Laufzeitfehler!
   Exit Sub
Mist:
   Select Case WieWeiterMitFehler(cstrProzName, m_cstrModulName)
   Case vbAbort: End
   Case vbRetry: Resume
   Case vbIgnore: Resume Next
   End Select
End Sub
```

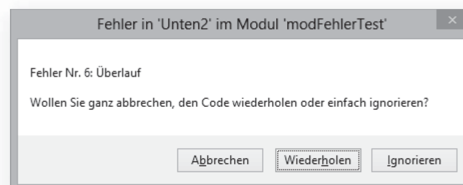


Abbildung 6.6: Die zentrale Fehlerbehandlung mit dem geänderten Code

Wie Sie schon sehen, vergrößert diese Fehlerbehandlung trotz des Versuchs einer zentralen Prozedur den benötigten Code deutlich. In jeder Prozedur müssen Sie knapp zehn Zeilen hinzufügen, aber dafür wenigstens nicht auch noch die Meldung einzeln programmieren. Bei üblicherweise längeren Prozeduren als den hier zu Demozwecken benutzten, relativiert sich das Problem auch noch ein wenig.

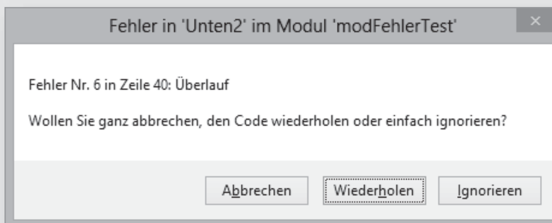
Zeilennummer des Fehlers ermitteln

Trotz des ganzen Aufwands wissen Sie im Fehlerfall nur, in welcher Prozedur der Fehler auftrat, aber nicht in welcher Zeile. Das geht immer noch mit einer undokumentierten Funktion `Erl()` *(Error line)*, die aber voraussetzt, dass die Zeilen tatsächlich wie zu frühen BASIC-Zeiten nummeriert sind. Die Prozedur `Unten2` sieht dann so aus:

```
Sub Unten2()
10:   Dim intTest As Integer
20:   Const cstrProzName = "Unten2"
30:   On Error GoTo Mist
40:   intTest = 123456
50:   Exit Sub
Mist:
   Select Case WieWeiterMitFehler(cstrProzName, m_cstrModulName)
   Case vbAbort: End
   Case vbRetry: Resume
   Case vbIgnore: Resume Next
   End Select
End Sub
```

Dementsprechend verändert sich die Fehleranzeige, sodass nun dank der automatisch gefüllten Funktion `Erl()` die Zeilennummer, genauer gesagt die Nummer vor der Zeile, in der Titelzeile der Fehlermeldung genannt wird:

```
Function WieWeiterMitFehler(strProzName As String, strModulName As String) As
VbMsgBoxResult
   WieWeiterMitFehler = MsgBox("Fehler Nr. " & Err.Number & " in Zeile " & Erl() & _
      ": " & Err.Description & vbCrLf & vbCrLf & _
      "Wollen Sie ganz abbrechen, den Code wiederholen oder einfach ignorieren?", _
      vbAbortRetryIgnore + vbDefaultButton2, _
      "Fehler in '" & strProzName & "' im Modul '" & strModulName & "'")
End Function
```



Damit Sie die Zeilen nicht manuell durchnummerieren müssen, finden Sie im Internet viele AddIns oder Module, die das für Sie übernehmen können. Immerhin muss das nach jeder Codeänderung erneut durchgeführt werden, damit es keine doppelten Nummern gibt und neue Zeilen auch Nummern erhalten!

6.2 Programmablauf verfolgen

Zur Fehlerbehandlung gehört auch, dem Code zur Laufzeit »zuzusehen« und zu prüfen, ob er wie geplant läuft und alle Variablen die erwarteten Inhalte besitzen. Dafür gibt es verschiedene Techniken:

- **Zeilenweise Ausführung:** dient dazu, von Anfang an für jede Zeile einzeln zu prüfen, ob sie überhaupt und richtig ausgeführt wird
- **Stop-Techniken:** ermöglichen es, per Code oder Breakpoint, den Ablauf des Programms an vorher festgelegten Punkten gezielt zu unterbrechen
- **Überwachungs-Ausdrücke:** zeigen die Inhalte der Variablen zu bestimmten Zeitpunkten an oder stoppen die Ausführung, wenn sich Variablen verändern

Oft werden Sie die hier gezeigten Möglichkeiten auch miteinander kombinieren.

Code zeilenweise ausführen

Bisher haben Sie den Code typischerweise mit der F5-Taste gestartet. Das entspricht dem Menübefehl *Ausführen/Sub/Userform ausführen* und lässt die Anweisungen ohne Unterbrechung laufen. Sie finden im *Debuggen*-Menü jedoch viele Alternativen dazu:

- **Einzelschritt:** Sie können mit F8 den Code im Einzelschritt testen. Dabei wird jeweils die nächste Zeile gelb gefärbt, die aber noch *vor* ihrer Ausführung steht.
- **Prozedurschritt:** Wenn in Ihren Prozeduren andere (Unter-)Prozeduren aufgerufen werden, können Sie sich deren Details ersparen, indem Sie diese Zeilen mit Umschalt+F8 in einem Schritt ausführen. Mit F8 würden Sie auch von solchen Unterprozeduren alle Zeilen einzeln starten.
- **Prozedur abschließen:** Haben Sie eine Unterprozedur schon teilweise ausgeführt, wollen nun aber nicht mehr jedes Detail sehen, können Sie mit Strg+Umschalt+F8 zur übergeordneten Prozedur zurückkehren
- **Ausführen bis Cursor-Position:** Anstatt jedes Mal einen Breakpoint (siehe unten) zu setzen, können Sie auch zwischendurch den Cursor in die gewünschte Zeile stellen und mit Strg+F8 alles bis dahin ausführen lassen

Code anhalten

Es gibt zwei Varianten, den Code automatisch anzuhalten, wenn er ausgeführt wird:

- **Breakpoints:** sind temporäre Markierungen, an denen der Code stoppt. Sie werden durch einen Klick auf den hellgrauen Bereich links vor der Zeile gesetzt und hinterlegen die Zeile dunkelrot.

- **Stop:** ist eine VBA-Anweisung, die ebenfalls dafür sorgt, dass die Codeausführung an dieser Stelle unterbrochen wird

Während die `Stop`-Anweisung wie jedes andere VBA-Schlüsselwort dauerhaft im Code enthalten bleibt, werden Breakpoints nicht mitgespeichert. Sie können mit *Debuggen/Alle Haltepunkte löschen* auch komplett entfernt werden, ohne die Datenbank schließen zu müssen.

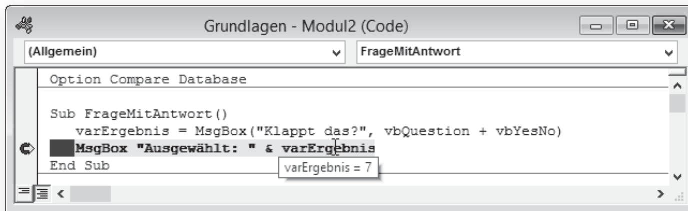


Abbildung 6.7: Durch den Breakpoint stoppt der Code und die Variable kann überprüft werden

In der obigen Abbildung sehen Sie einen (dunkelrot hinterlegten) Breakpoint, sodass die Codeausführung *vor* der so markierten Zeile stoppt. Bevor Sie ausgeführt wird, können Sie durch Überfahren mit der Maus im QuickInfo schon den Inhalt der Variablen `varErgebnis` sehen.

Überwachungen hinzufügen

Da nicht alle Variablen ihren Inhalt anzeigen können, wenn Sie bei Laufzeitunterbrechung die Maus über ihrem Namen ruhen lassen, gibt es extra noch die Möglichkeit, Überwachungswerte hinzuzufügen. Diese Ergebnisse werden in einem eigenen Fenster aufgelistet und dürfen nicht nur einzelne Variablen, sondern ebenso Funktionen, Berechnungen und vor allem Objekte enthalten.

1. Um einen Wert zu überwachen, markieren Sie die Variable (hier aus der Prozedur `TeilMeinesFormulars` aus Kapitel 3) und lassen per Rechtsklick das PopUp-Menü anzeigen. Klicken Sie darin den Befehl *Überwachung hinzufügen…* an.

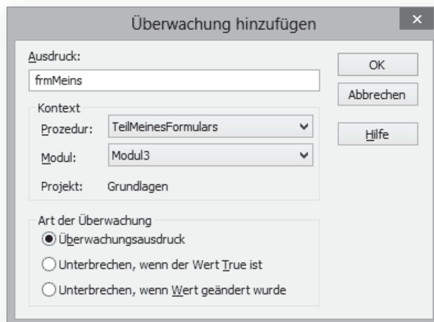


Abbildung 6.8: Bestätigen Sie den ÜberwachungHinzufügen-Dialog

2. Jetzt ist diese Variable im Überwachungsausdrücke-Fenster aufgenommen, aber dort zur Entwurfszeit noch ohne Wert.

3. Starten Sie die Prozedur (vorher muss das Formular *frmTest* geöffnet worden sein!), nachdem Sie in ihr entweder einen Breakpoint gesetzt oder eine `Stop`-Anweisung einge-

fügt haben. Jetzt sehen Sie im Überwachungsausdrücke-Fenster nicht nur deren Wert, sondern können für Objektvariablen sogar an den Plus-Zeichen die Unterobjekte und wiederum deren Unterobjekte sehen:

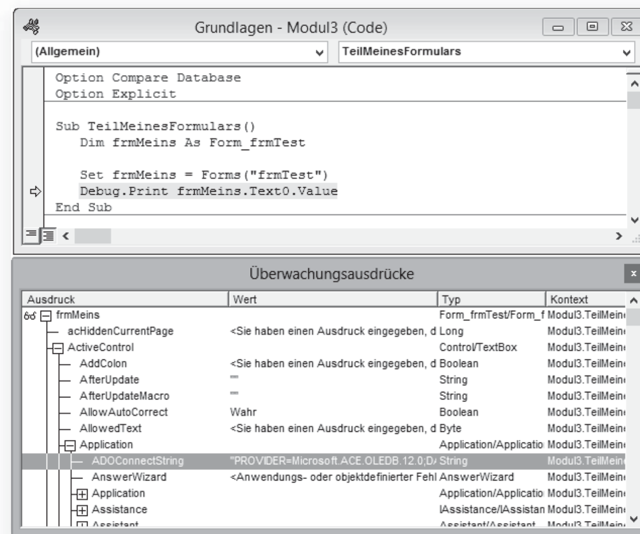


Abbildung 6.9: ***Das Überwachungsausdrücke-Fenster erlaubt detaillierte Einblicke in alle Objekte***

6.3 Übungen zu diesem Kapitel

In diesem Abschnitt finden Sie einige Übungen zu diesem Kapitel. Die richtigen Antworten finden Sie am Ende des Buchs.

Übung 6.1

Welche Fehler stecken im folgenden Code?

```
Sub FehlerImCode()
   Dim intWert1, intWert2 As Integer, intWert3 As Integer
   intWert1 = 150
   intWert2 = 932.5
   intWert3 = intWert1 * intWert2
End Sub
```

Übung 6.2

Ist `If CurrentDb.QueryDef = „qryTest" Then` ein Syntaxfehler oder ein Laufzeitfehler?

Übung 6.3

Mit welcher Anweisung können Sie die Fehlerbehandlung wieder an VBA zurückgeben?