Tutorial

3

EDV INNOVATION & CONSULTING

Stefan Kulpa

Gerhard-Domagk-Str. 6
D-41540 Dormagen
http://www.kulpa-online.com
Email: stefan@kulpa-online.com

VBA

EINFÜHRUNG

LEVEL ADVANCED

Inhaltsverzeichnis

EFFEKTIVES „ERROR CODING“ IMPLEMENTIEREN 4

Explizites Codieren 4

Benutze IMMER Option Explicit 4

Benutze IMMER explizite Variablentypen 5

Vermeide Deftype-Statements 5

Initialisieren Variablen 6

Nutze TypeName, VarType und TypeOf 7

Argumente 7

Nutze immer ByRef und ByVal 7

Nutze explizite Typ-Argumente 8

Setzte explizit Standardwerte für optionale Argumente 8

Validiere alle Argumente 8

Benutze benannte Argumente 8

Arrays (Datenfelder) 9

Unterstelle niemals untere Array Grenzen 9

Vermeide hart-codierte Array Grenzen 10

Vermeide die Option Base Anweisung 10

CODIER-EMPFEHLUNGEN 11

Implementiere immer einen Else-Fall 11

Vermeide es, Standardeigenschaften zu benutzen 12

Vermeide vermischte Datentypen in Ausdrücken 12

Benutze Konstanten 13

Vermeide Operatoren-Prioritäten 13

Prüfe String-Längen 13

Schließe alle offenen Objekte 14

Setze Objekte auf Nothing 14

Schalte Fehlerhandlings explizit (wieder) aus 14

Treffe niemals Annahmen über die „externe“ Welt 15

Codiere niemals mit „Kopieren und Einfügen“ 15

Benutze + und & korrekt 15

Setze Eigenschaften zur Laufzeit 15

„ERROR CODING“ MECHANISMEN 16

On Error Resume Next 16

On Error Goto 16

Programmablauf wiederaufnehmen 16

Resume 17

Resume Next 17

Resume Line 17

Mehrere Error Handles 18

Das Error-Objekt 18

Überprüfen der Fehlernummer 19

Bereinigen des Error-Objekts 20

Ausschalten des Error-Handlers 21

Gültigkeitsbereich von Error-Handlern 21

Fehlercodes im Aufrufstapel 23

DEBUGGING IN VBA 24
Debugging erfolgt im Code 24
Das Direktfenster 25
Das Lokal-Fenster 26
Das Überwachungsfenster 27
WISSENSWERTES 28
Gültigkeitsbereiche und Lebensdauer von Variablen 28
Gültigkeitsbereiche: Die nackten Fakten 29
Pass auf Deine Typen auf... 30
Datentypen (Zusammenfassung) 30

Kapitel 1

EFFEKTIVES „ERROR CODING“ IMPLEMENTIEREN

Einen VBA-Code schreiben ist eine Sache. Ihn zum Laufen zu bringen, eine andere. Denn hier beginnt die eigentliche Arbeit. Fehler aufzuspüren und zu beseitigen ist ein wichtiger Teil des Programmierens. Wie man das in VBA am besten macht, beschreibt dieses Tutorial. Wir gehen außerdem darauf ein, wie man Fehlern eine Falle stellt und dem Programm damit ermöglicht, unbeabsichtigte Ereignisse elegant zu parieren.

Wir möchten Ihnen vorab schon einen Tipp mit auf den Weg geben, der eventuell sehr wichtig für Ihre Gemütsverfassung sein kann: **Regen Sie sich niemals, also das bedeutet an keinen Tag, zu keiner Stunde, wegen eines Programmierfehlers auf!** Ein guter Programmierer ist auch ein guter Fehlersucher, der Fehlermeldungen nur als dankbaren Hinweis der Umgebung betrachtet, wie er eventuell den Fehler schneller finden kann.

Jeder, der eigene Programme schreibt, wird mit Fehlermeldungen konfrontiert. Sie gehören genauso zur täglichen Arbeit, wie das ständige Klappern der Tastatur. Je mehr Sie sich allerdings über die Hinweise aufregen, desto genervter gehen Sie an die Lösung der Probleme heran und verschlimmern nur noch alles. Wie ging das Lind noch von Bobby McFerrin? **Don’t worry, be happy!**

Explizites Codieren

Benutze IMMER Option Explicit

Das folgende Beispiel

```
Sub DisplayName()
    sUserName = "J. Brown"
    MsgBox "My name is " & sUsrName
End Sub
```

würde ohne gesetztes Option Explicit problemlos kompiliert und ausgeführt werden.

Sobald jedoch **Option Explicit** gesetzt wird, meldet der Compiler die unbekannte Variable **sUsrName**. Eine bessere Variante wäre demnach:

```
Option Explicit
Sub DisplayName()
    Dim sUserName As String
    sUserName = "J. Brown"
    MsgBox "My name is " & sUserName
End Sub
```

In Visual Basic und in Visual Basic For Application lässt sich das automatische Einfügen von Option Explicit in jedes neue Modul einstellen (Extras → Optionen → Variablendeklaration erforderlich).

Benutze IMMER explizite Variablentypen

Visual Basic geht grundsätzlich davon aus, dass eine Variable vom Typ Variant ist, solange diese nicht explizit mit einem speziellen Datentyp deklariert wird.

```
    Dim i, j, k As Long
```

In diesem Beispiel werden mit **i** und **j** implizit zwei Variant Variablen und mit **k** explizit eine Longinteger Variable deklariert – obwohl dies sicher nicht so beabsichtigt war.

Besser wäre folgende Variante:

```
    Dim i As Long, j As Long, k As Long
```

oder

```
    Dim i As Long
    Dim j As Long
    Dim k As Long
```

Diese zweite Variante erlaubt es, hinter den Variablen deren Bestimmung zu beschreiben. Dies erfolgt leider nur allzu selten, was die Lesbarkeit des Codes nicht gerade fördert.

Vermeide Deftype-Statements

Visual Basic kennt folgende DefType-Befehle

```
DefBool, DefByte, DefInt, DefLng
DefCur, DefSng, DefDbl, DefDec
DefDate, DefStr, DefObj, DefVar
```

Wenn man diese Befehle noch nicht kennt, sollte man es dabei belassen. Wer diese Befehle jedoch kennt und vielleicht auch einsetzt, sollte dies zukünftig vermeiden.

Deftype-Befehle werden modulglobal gesetzt und werden auf alle Variablen in einem bestimmten Bereich angewendet, die nicht explizit mit einem Variablentypen deklariert werden.

Beispiel:

```
Option Explicit
DefInt A-K
Sub DefTypeCheck()
    Dim a, l, m As Long
    Debug.Print VarType(a) '-> 2 = Integer
    Debug.Print VarType(l) '-> 0 = nicht initialisiert
    Debug.Print VarType(m) '-> 3 = Long
End Sub
```

- ➔ Die Variable a fällt in den Bereich **DefInt A-K** und wird daher als Integer deklariert.
- ➔ Die Variable l fällt weder in den DefInt-Bereich noch wird sie explizit deklariert; es handelt sich also um einen Variant.
- ➔ Die Variable m wurde explizit als Longinteger deklariert.

Bereits in diesem simplen Beispiel sieht man die potentiellen Gefahren im Umgang mit den DefType-Deklarationen. Daher sollte man diese Art der impliziten Variablendeklaration ganz aus dem Code verbannen.

Grundsätzlich gilt: Nutze Variant- und Objekttyp-Variablen grundsätzlich nur in speziellen Situationen, wo es sich nicht vermeiden lässt.

Initialisieren Variablen

Man sollte nie die implizite Initialisierung von Variablen voraussetzen. Numerische Variablen werden durch die Deklaration normalerweise mit 0 und String-Variablen mit einem Leerstring initialisiert. Aber bleibt dies so auch in späteren Versionen bestehen? Wie wird eine Variant- oder Objekt-Variable initialisiert?

Was passiert, wenn VB Code nach VB-Script oder eine andere Sprache portiert wird? Erfolgt dort auch diese implizite Initialisierung? Es gibt noch eine Reihe anderer Fragen, so dass die einzig korrekte Antwort das explizite Initialisieren von Variablen ist. Dies ist nicht nur direkt nach der Deklaration notwendig, sondern auch vor deren Nutzung.

```
    For i = LBound(MyArray) To UBound(MyArray)
        s = s & MyArray(i)
    Next
```

Wie kann man sicher sein, dass in die Variable s zu Beginn der Schleife leer ist? Vor allem in großen Codemodulen kann es schon mal passieren, dass eine Variable mehrfach benutzt wird. Eine simple Initialisierung schafft hier Abhilfe:

```
    s = ""
    For i = LBound(MyArray) To UBound(MyArray)
        s = s & MyArray(i)
    Next
```

Hinweis: Objektvariable sollten direkt nach ihrer Deklaration mit Nothing initialisiert werden.

```
    Dim objMyObject As Object
    Set objMyObject = Nothing
```

Nutze TypeName, VarType und TypeOf

Diese drei Visual Basic Funktionen sind bei nativen Varianten und Objektvariablen sehr nützlich:

```
Sub ShowUsers(User As Object)
    If TypeName(User) = "UserObject" Then
        MsgBox "The User Name is: " & User.Name
    End If
End Sub

Sub ShowUsers(UserName As Variant)
    If VarType(UserName) = vbString Then
        MsgBox "The User Name is: " & UserName
    End If
End Sub

Sub ShowUsers(User As Object)
    If TypeOf User Is UserObject Then
        MsgBox "The User Name is: " & User.Name
    End If
End Sub
```

Argumente

Nutze immer ByRef und ByVal

Argumente (Parameter) werden in Visual Basic implizit ByRef – als Referenz – übergeben.

Die so übergebenen Argumente können durch die aufgerufene Routine verändert werden. Dies kann gewollt sein, oder auch nicht.

Werden Argumente explizit ByVal übergeben, so bleibt die ursprüngliche Variable unverändert und es wird eine Kopie an die aufgerufene Routine übergeben.

```
Sub Arguments()
    Dim sMyValue As String

    sMyValue = "Hello World!"
    Debug.Print sMyValue        '-> Hello World!

    Call ByRefSample(sMyValue)
    Debug.Print sMyValue        '-> Result from ByRefSample

    Call ByValSample(sMyValue)
    Debug.Print sMyValue        '-> Result from ByRefSample
End Sub

Sub ByRefSample(ByRef sMyValue As String)
    sMyValue = "Result from ByRefSample"
End Sub

Sub ByValSample(ByVal sMyValue As String)
    sMyValue = "Result from ByValSample"
End Sub
```

Grundsätzlich sollte man ByRef-Argumente vermeiden und Argumente explizit ByVal übergeben, sofern keine triftigen Gründe dagegen sprechen.

Nutze explizite Typ-Argumente

Auch bei Argumenten sollten explizite Datentypen genannt werden.

```
Sub ShowUsers(ByVal UserCount)
```

In diesem Beispiel könnte es sich bei UserCount um einen Varianten handeln, da dies standardmäßig der Fall ist. Es könnte aber auch aufgrund einer DefType-Anweisung ein ganz anderer Datentyp sein. Deutlich stabiler wird diese Routine durch explizite Typ-Definition des Arguments UserCount; z.B.:

```
Sub ShowUsers(ByVal UserCount As Variant)
```

Setzte explizit Standardwerte für optionale Argumente

Optionale Argumente ohne Standardwerte können bzw. sollten vor ihrer Verwendung geprüft werden:

```
Sub ShowUserName(Optional ByVal UserCount As Variant)
    If IsMissing(UserCount) Then UserCount = 0
End Sub
```

Durch die Vergabe eines Standardwerts ist zum einen die Überprüfung mit IsMissing nicht notwendig zum anderen wird – zumindest im Falle eines Varianten – dessen Typ klar(er).

```
Sub ShowUserName(Optional ByVal UserCount As Variant = 0)
```

In diesem Beispiel wird klar, dass es sich um ein numerisches Argument mit dem Standardwert 0 handelt.

Validiere alle Argumente

Jede Routine sollte die Verantwortung für die Validierung ihrer Argumente haben:

- ➔ Argumente sind optional oder nicht
- ➔ Gültige Argumente besitzen den korrekten Datentypen
- ➔ Numerische Argumente liegen im erwarteten Wertebereich etc.

Grundsätzlich sollten keine Annahmen über die Gültigkeit von Argumenten getroffen werden!

Eine Routine kann/sollte nicht wieder verwendet werden, wenn es nicht seine Argumente überprüft!

Benutze benannte Argumente

Code wird deutlich lesbarer, wenn Argumente beim Aufruf einer Routine benannt werden. Zudem vereinfacht dies die Handhabung von Routinen mit vielen optionalen Argumenten drastisch; gleichermaßen verringert sich die Fehleranfälligkeit durch falsch benutzte Argumente.

Die Codezeile

```
If bGetUserInfo(sUserName, False) Then
```

wird durch Nennung der Argumente deutlich lesbarer

```
If bGetUserInfo(sUserAcountName:=sUserName, _
                bShowPasswortDialog:=False) Then
```

Arrays (Datenfelder)

Datenfeld Operationen sind ein weiteres Spektrum, Fehlern durch „implizites Verhalten“ die Türen zu öffnen. Nachfolgende werden einige implizite Verhaltensweisen dargestellt, die vermieden werden sollten.

Unterstelle niemals untere Array Grenzen

Aufgrund der Standardgrenze für die untere Array Grenze (=0) läuft man Gefahr, diese untere Grenze grundsätzlich als gesetzt anzunehmen. Bereits eine globale Abweichung dieser Grenze durch eine **Option Base** Anweisung führt hier bereits zu potentiellen Fehlern.

Daher sollte man grundsätzlich die untere Grenze explizit setzen, worauf dann auch eine **Option Base** Anweisung keinen Einfluss mehr nimmt.

```
Option Explicit


Sub CountDown()

    Dim i           As Integer
    Dim sNums(10)   As String

    sNums(10) = "Zehn"
    sNums(9) = "Neun"
    sNums(8) = "Acht"
    sNums(7) = "Sieben"
    sNums(6) = "Sechs"
    sNums(5) = "Fünf"
    sNums(4) = "Vier"
    sNums(3) = "Drei"
    sNums(2) = "Zwei"
    sNums(1) = "Eins"
    sNums(0) = "Null"

'// Countdown von 10 bis 0
    For i = UBound(sNums) To _
            LBound(sNums) Step -1
        Debug.Print sNums(i)
    Next

End Sub
```

Dieses Beispiel funktioniert, da implizit die untere Array-Grenze auf 0 gesetzt wird.

```
Option Explicit
Option Base 1

Sub CountDown()

    Dim i           As Integer
    Dim sNums(10)   As String

    sNums(10) = "Zehn"
    sNums(9) = "Neun"
    sNums(8) = "Acht"
    sNums(7) = "Sieben"
    sNums(6) = "Sechs"
    sNums(5) = "Fünf"
    sNums(4) = "Vier"
    sNums(3) = "Drei"
    sNums(2) = "Zwei"
    sNums(1) = "Eins"
    sNums(0) = "Null"

'// Countdown von 10 bis 0
    For i = UBound(sNums) To _
            LBound(sNums) Step -1
        Debug.Print sNums(i)
    Next

End Sub
```

Dieses Beispiel führt zu einem Fehler, da die Anweisung Option Base 1 alle unteren Arraygrenzen grundsätzlich auf 1 stellt, sofern diese nicht explizit gesetzt werden.

Aus diesem Grund sollten untere Arraygrenzen grundsätzlich deklariert werden:

```
Dim sNums(0 To 10) As String
```

Vermeide hart-codierte Array Grenzen

Hart codierte Array-Grenzen erschweren die Pflege des Codes. Wenn sich später die Grenzen ändern, müssen alle Stellen, an denen auf die Array-Grenzen zugegriffen werden, gesucht und überarbeitet werden.

Werden stattdessen Konstanten genutzt, müssen lediglich die Konstanten geändert werden.

```
Option Explicit

Sub CountDown()

 Dim i               As Integer
 Dim sNums(0 To 10)  As String

 sNums(10) = "Zehn"
 sNums(9) = "Nen"
 sNums(8) = "Acht"
 sNums(7) = "Sieben"
 sNums(6) = "Sechs"
 sNums(5) = "Fünf"
 sNums(4) = "Vier"
 sNums(3) = "Drei"
 sNums(2) = "Zwei"
 sNums(1) = "Eins"
 sNums(0) = "Null"

'// Countdown von 10 bis 0
 For i = 10 To 0 Step -1
     Debug.Print sNums(i)
 Next

End Sub
```

```
Option Explicit

Sub CountDown()

 Const COUNT_MIN  As Integer = 0
 Const COUNT_MAX  As Integer = 10

 Dim i            As Integer
 Dim sNums(COUNT_MIN To _
           COUNT_MAX)    As String

 sNums(10) = "Zehn"
 sNums(9) = "Neun"
 sNums(8) = "Acht"
 sNums(7) = "Sieben"
 sNums(6) = "Sechs"
 sNums(5) = "Fünf"
 sNums(4) = "Vier"
 sNums(3) = "Drei"
 sNums(2) = "Zwei"
 sNums(1) = "Eins"
 sNums(0) = "Null"

'// Countdown von 10 bis 0
 For i = COUNT_MAX To COUNT_MIN Step -1
   Debug.Print sNums(i)
 Next

End Sub
```

Vermeide die Option Base Anweisung

Wie bereits zuvor geschildert, führt die Option Base Anweisung zu potentiellen Fehlersituationen.

Da die explizite Deklaration der Array-Grenzen zu bevorzugen ist, sollte auch auf die Option Base Anweisung grundsätzlich verzichtet werden, da diese bei expliziter Grenzenbestimmung keine Wirkung mehr hat.

Kapitel 2

CODIER-EMPFEHLUNGEN

Die nachfolgenden Kapitel beinhalten Empfehlungen für ein explizites Codieren, um den Code zu verbessern und potentielle Fehlerquellen auszuschalten.

Implementiere immer einen Else-Fall

In jedem If-Then und Select Case Block sollte immer ein Else Zweig implementiert werden, auch wenn man sich sicher ist, das kein anderer Fall eintreten kann. Spätestens bei Erweiterungen des Codes sind diese Blöcke hilfreich, um „Grenzfälle“ abzufangen:

Beispiel:

```
Sub HandleMenu(mnuChoice As Integer)

    Select Case mnuChoice
           Case MNU_OPEN
                OpenFile
           Case MNU_CLOSE
                CloseFile
           Case MNU_SAVE
                SaveFile
           Case Else
                MsgBox "Unbekannter Menübefehl!"
    End Select

End Sub
```

bzw.

```
Sub HandleMenu(mnuChoice As Integer)

    If mnuChoice = MNU_OPEN Then
        OpenFile
    ElseIf mnuChoice = MNU_CLOSE Then
        CloseFile
    ElseIf mnuChoice = MNU_SAVE Then
        SaveFile
    Else
        MsgBox "Unbekannter Menübefehl!"
    End If

End Sub
```

Wird für diese Routinen das zugrunde liegende Menü erweitert, ist schnell klar, an welcher Stelle die Codeblöcke nachbearbeitet werden müssen. Ohne diese Else-Zweige würde gar nichts passieren und es wäre nicht unbedingt immer direkt offensichtlich, an welcher Stelle „nachzubessern“ ist.

Vermeide es, Standardeigenschaften zu benutzen

Sehr oft werden Standardeigenschaften von Objekten genutzt, ohne diese explizit zu (be)nennen.

Der Code wird für manche dadurch schlanker und „lesbarer“. Die beiden nachfolgenden Zeilen zeigen die gleiche Information:

```
    MsgBox Err.Number
    MsgBox Err
```

Der Code funktioniert, da „**Number**“ die Standardeigenschaft des Err-Objekts ist.

Standardeigenschaften können sich in Nachfolgeversionen ändern. Zudem ist das Weglassen dieser (Standard)-Eigenschaften nicht wirklich der Lesbarkeit dienlich. Darüber hinaus ist es auch nicht immer direkt ersichtlich, ob man ein Objekt referenziert und lediglich dessen Standardwert abfragt.

Ein weiterer technischer Grund für die explizite Nutzung/Nennung der Standardeigenschaften, sind bekannte Speicherprobleme mit manchen COM-Objekten bei der impliziten Nutzung der Standardeigenschaften.

Der wohl am meisten implizit genutzte Eigenschaftswert ist der „**Field Value**“ bei Recordsets. Viele Programmierer nutzen diese Eigenschaft ungefähr wie folgt:

```
rs("FieldName")
```

Hier liegt sogar ein doppelter impliziter Standard vor. Die vollständige – und empfohlene – Referenz sollte wie folgt aussehen:

```
rs.Fields("FieldName").Value
```

Vermeide vermischte Datentypen in Ausdrücken

Visual Basic erlaubt vermischte Ausdrücke wie z.b.:

```
CurDate = Now + 1
```

Derartige Konstrukte sollten vermieden werden, da sie voraussetzen, dass

- ➔ der Programmierer diese VB/A spezifischen impliziten Ausdrücke kennt
- ➔ die Portierbarkeit des Codes erschwert wird
- ➔ es nicht gewährleistet wird, dass implizite Unterstützung immer gültig ist

Es ist daher besser, Funktionen zu nutzen, die für den jeweiligen Fall ausgelegt sind; z.B.:

```
CurDate = DateAdd("d", 1, Now)
```

Benutze Konstanten

Nutze niemals „hart-codierte“ Elemente wie z.b. Array Grenzen, Control Indizes (nur VB) oder Spalten/Zeilenpositionen in Matrix-Objekten (Grids etc.).

Wenn sich deren Werte ändern, kann es u.U. äußerst schwierig sein, alle betroffenen Stellen zu korrigieren.

```
Hard-codierte Elemente
    Dim nums(0 To 10)
    grid.Col(4).Text = "12345"
    txtInput(4).Text = "12345"
```

```
Nutzung von Konstanten
    Dim nums(NUMS_LOWER To NUMS_UPPER)
    grid.Col(GRID_ZIP).Text = "12345"
    txtInput(TEXT_SSN).Text = "12345"
```

Vermeide Operatoren-Prioritäten

Grundsätzlich sollte man nicht davon ausgehen, dass mathematische Regeln wie „Punkt vor Strich“ korrekt umgesetzt werden. Aus diesem Grund sollte man immer mit entsprechenden Klammern arbeiten:

```
statt  DblResult = dblBase + dblCost * dblUnits
besser DblResult = dblBase + (dblCost * dblUnits)
```

oder

```
statt  DblResult = dblBase + dblCost * dblUnits + 4.6 / dblAdj
besser DblResult = dblBase + (dblCost * dblUnits) + (4.6 / dblAdj)
```

Prüfe String-Längen

Je nach VB/VBA Version werden ungültige String-Längen mal als Fehler und mal nicht als Fehler behandelt.

```
Function GetZipOnly(strZip As String) As String
    Const ZIP_BASELENGTH = 5
    GetZipOnly = Left$(strZip, ZIP_BASELENGTH)
End Function
```

Was passiert, wenn der Wert des Arguments strZip kleiner als 5 Zeichen lang ist? Je nach VB/A-Version wird ein Laufzeitfehler erzeugt oder aber auch nicht. Darauf sollte man sich nicht verlassen, und grundsätzlich String-Längen prüfen; z.B.:

```
Function GetZipOnly(strZip As String) As String
    Const ZIP_BASELENGTH = 5
    If Len(strZip) < ZIP_BASELENGTH + 1 Then
        GetZipOnly = strZip
    Else
        GetZipOnly = Left$(strZip, ZIP_BASELENGTH)
    End If
End Function
```

Schließe alle offenen Objekte

Grundsätzlich sollten alle Objekte nach deren Nutzung explizit geschlossen werden. Auch wenn VB/A vermeintlich Objekte automatisch schließt, wenn eine Routine verlassen wird, so sollte man sich nicht darauf verlassen. Zudem kann das Nicht-Schließen von Objekten bei späteren Erweiterungen des Codes zu Fehlern führen, wenn dieser Umstand nicht berücksichtigt wird.

```
    Set rs = OpenRecordset(db, sql)
    sName = rs.Fields("Name").Value
    rs.Close
```

Das explizite Schließen von Objekte sollte jeweils durchgeführt werden; dies gilt auch für Dateioperationen, bei denen die betroffenen Dateien stets nach deren Nutzung geschlossen werden!

Setze Objekte auf Nothing

Alle Objekte, die nicht mehr genutzt werden, sollten explizit auf „**Nothing**“ gesetzt werden. Dieser Punkt wird sehr oft diskutiert und es hält sich hartnäckig das Argument, dass Objekte automatisch auf „**Nothing**“ gesetzt werden, wenn sie sich nicht mehr „im aktuellen Gültigkeitsbereich“ befinden.

Das mag zum Teil stimmen, aber was spricht dagegen, auch diese Annahme durch explizites „Zerstören“ von Objekten zur Gewissheit werden zu lassen?

Schalte Fehlerhandlings explizit (wieder) aus

Eine On Error Anweisung schaltet VB/A explizit in einen „Fehler-Handling-Modus“. Daher sollte grundsätzlich nach jedem logischen Block das Error-Handling auch wieder explizit ausgeschaltet werden.

```
    Dim rs As Recordset
    Dim sName As String

    On Error Resume Next
    Set rs = OpenRecordset(db, sql)
    If Err.Number <> 0 Then Exit Sub
    sName = rs.Fields("Name").Value
    rs.Close
```

In diesem Beispiel wird zwar ein Fehler-Handling aktiviert aber nicht wieder ausgeschaltet. Es ist besser, das Error-Handling entsprechend zu beenden:

```
    Dim rs As Recordset
    Dim sName As String
    Dim intError As Integer
    On Error Resume Next
    Set rs = OpenRecordset(db, sql)
    intError = Err.Number
    On Error GoTo 0
    If intError <> 0 Then Exit Sub
    sName = rs.Fields("Name").Value
    rs.Close
```

Treffe niemals Annahmen über die „externe“ Welt

Man sollte grundsätzlich keine Annahmen über die „**Welt**“ außerhalb des eigenen Programms treffen, die man nicht beeinflussen kann. Beim Lesen von Dateien oder Datenbanken und während der Kommunikation mit Schnittstellen kann man nie den jeweiligen Status voraussetzen. Dateien können schreibgeschützt, Datenbanken exklusiv geöffnet und Schnittstellen aktuell nicht verfügbar sein.

Es liegt jeweils in der Verantwortung des „**zugreifenden**“ Programms mit möglichen Fehlern beim Zugriff intelligent umzugehen und niemals den gewünschten Status vorauszusetzen.

Codiere niemals mit „Kopieren und Einfügen“

Eine der größten „**Programmkrankheiten**“ hat ihren Ursprung im Kopieren und Einfügen von Codeteilen aus anderen Programmen. Hierbei werden nicht nur bestehende bzw. potentielle Fehlerquellen gleich mit kopiert, sondern der Änderungsaufwand wird oft unterschätzt, so dass in vielen Fällen die Erstellung eines neuen, robusten Codes effektiver ist.

Benutze + und & korrekt

Obwohl VB/A das Plus-Zeichen zum Konkatenieren (Verknüpfen) von Strings unterstützt, sollte hierzu grundsätzlich das kaufmännische UND benutzt werden. Das Plus-Zeichen sollte ausschließlich für mathematische Operationen verwendet werden.

Setze Eigenschaften zur Laufzeit

Das Setzen von Control-Eigenschaften zur Design-Zeit erscheint schnell und bequem vonstatten zu gehen. Dies sollte man trotzdem vermeiden und alle Eigenschaften zur Laufzeit explizit über die Control-/Objekteigenschaften setzen. Dies macht den Code lesbarer und wartbarer; zudem können beispielsweise Control-Updates nicht mehr „störend“ auf diese Eigenschaften einwirken.

Kapitel 3

„ERROR CODING“ MECHANISMEN

Um die interne Fehlerbehandlung zu aktivieren, muss eine der beiden On Error Anweisungen genutzt werden; **On Error Resume Next** oder **On Error Goto**.

On Error Resume Next

Die Anweisung On Error Resume Next ist die erste Anweisung zum Schutz vor Fehlern. Dabei wird VB/A veranlasst, bei einem Fehler in der Zeile den Code fortzuführen, die nach der Zeile kommt, die den Fehler verursacht hat. Belässt man es bei dieser Anweisung, handelt es sich nicht um Fehlerbehandlung, sondern um Fehlerunterdrückung. Fehlerunterdrückung ist sehr gefährlich. Es versteckt Fehler vor dem Programmierer, dem Tester und letztendlich dem Benutzer.

Es gibt nur sehr wenige Situationen, in denen diese Fehlerunterdrückung legitim ist.

On Error Goto

Die Anweisung On Error Goto ist die zweite Anweisung zum Schutz vor Fehlern. Diese Anweisung veranlasst VB/A bei einem Fehler zu einer bestimmten Marke in der Routine zu springen, in der sich eine Fehlerbehandlung befindet.

```
Fehlerunterdrückung
    Dim i As Integer
    On Error Resume Next
    i = 8 / 0
```

```
Fehlerbehandlung
    Dim i As Integer
    On Error GoTo ErrTrap
    i = 8 / 0
    Exit Sub

ErrTrap:
    MsgBox "Es ist ein Fehler aufgetreten."
```

Programmablauf wiederaufnehmen

Sobald der interne Error-Handler aktiviert wurde, muss das Programm zum „normalen Ablauf“ zurückkehren. Hierfür stellt VB/A die Anweisung Resume (Wiederaufnehmen) zur Verfügung.

Man unterscheidet drei Varianten:

Resume

Die Anweisung Resume entspricht einer „Retry“-Anweisung, also die Wiederholung der Codezeile, die den Fehler verursacht hat. Das macht nur dann Sinn, wenn in der Fehlerbehandlung die Voraussetzungen erfüllt wurden, um das Programm ab der entsprechenden Codezeile – jetzt ohne Fehler – fortführen zu können; z.B.:

```
Sub DivideByZero()
    Dim i As Integer
    Dim intDenom As Integer
    On Error GoTo ErrHandler
    intDenom = 0
    i = 8 / intDenom
    Exit Sub
ErrHandler:
    intDenom = InputBox("Bitte gültigen Divisor eingeben (>0)")
    Resume 'Retry
End Sub
```

Resume Next

Die Resume Next-Anweisung ist ähnlich der Resume-Anweisung, wobei hier nun in der Codezeile das Programm fortgeführt wird, die der Fehler verursachenden Codezeile folgt; z.B.:

```
Sub DivideByZero()
    Dim i As Integer
    Dim intDenom As Integer
    On Error GoTo ErrHandler
    intDenom = 0
    i = 8 / intDenom
    Exit Sub
ErrHandler:
    MsgBox "Division wurde abgebrochen!"
    Resume Next
End Sub
```

oder

```
Sub DivideByZero()
    Dim i As Integer
    Dim intDenom As Integer
    On Error Resume Next
    intDenom = 0
    i = 8 / intDenom
    If Err.Number <> 0 Then
        MsgBox "Division wurde abgebrochen!"
    End If
End Sub
```

Resume Line

Die dritte Version der Resume Anweisung verweist mit Line auf ein gültiges Programm-Label oder eine gültige Zeilennummer; z.B.:

```
Sub DivideByZero()
    Dim i As Integer
    Dim intDenom As Integer
    On Error GoTo ErrHandler
    intDenom = 0
    i = 8 / intDenom
    Exit Sub
AltCalc:
    i = ERR_VALUE
    Exit Sub
ErrHandler:
    MsgBox "Division wurde abgebrochen!"
    Resume AltCalc
End Sub
```

Bei der Nutzung der Resume **Line** Anweisung sollte man sehr vorsichtig sein. Wenn die Resume-Anweisung aufgerufen wird, ohne das ein Error-Handler aktiv ist, führt dies wiederum zu dem Fehler 20 („Resume ohne Fehler!“).

Mehrere Error Handles

In einer Prozedur kann jeweils immer nur ein Error-Handler aktiviert werden. Zudem gilt ein Error-Handler immer nur in einer Prozedur. Demzufolge gibt es keinen globalen Error-Handler, der über mehrere Prozeduren „wacht“. Um also jede Prozedur in einem Programm vor Fehlern zu schützen, muss in jeder Prozedur ein eigener Error-Handler implementiert werden.

Das nachfolgende Beispiel besitzt sog. Zeilennummern – ein Relikt, dass seit der ersten Basic Version existiert und bis zur aktuellsten VB/VBA-Version beibehalten wurde.

```
Sub DivideByZero()

1   On Error Resume Next
2   i = 8 / 0   'Diese Zeile wird nicht bewertet!
3   DivideByZeroAgain

7   End Sub

Sub DivideByZeroAgain()

4   On Error Resume Next
5   i = 8 / 0   'Diese Zeile wird nicht bewertet!

6   End Sub
```

In beiden Prozeduren werden Fehler jeweils durch die Error-Handler abgefangen. Dieses Beispiel soll lediglich aufzeigen, in Gültigkeit der Error-Handler darstellen.

Das Error-Objekt

Grundsätzlich besitzt jede VB/VBA-Routine automatisch eine Referenz auf das sog. Error-Objekt. Durch die Nutzung einer On Error – Anweisung wird in der jeweiligen Routine der Error-Handler aktiviert, welcher wiederum das Error-Objekt benutzt, um nähere Informationen zu dem aufgetretenen Fehler zu dokumentieren.

Das Error-Objekt besitzt folgende Eigenschaften

Err	das ErrorObjekt selbst
.Number	Gibt einen numerischen Wert zurück oder legt einen numerischen Wert fest, der einen Fehler angibt. Number ist die Standardeigenschaft des Err-Objekts.
.Description	Die Einstellung für die Description-Eigenschaft enthält eine kurze Beschreibung des Fehlers. Mit dieser Eigenschaft kann man für Benutzer eine Warnmeldung zu einem Fehler ausgeben, den man nicht verarbeiten kann oder will.
.LastDllError	Enthält unter Betriebssystemen vom Typ 32-Bit Microsoft Windows nur den Systemfehler-Code für den letzten Aufruf einer Dynamic Link Library (DLL).
.Source	Die Source-Eigenschaft gibt einen Zeichenfolgenausdruck an, der das Objekt darstellt, das den Fehler ursprünglich ausgelöst hat. Der Ausdruck ist normalerweise der Name der Klasse des Objekts oder die Ressource-ID.
.HelpFile	Wenn eine Microsoft Windows-Hilfedatei in HelpFile angegeben ist, wird sie automatisch aufgerufen, wenn der Benutzer im Dialogfeld mit der Fehlermeldung die Schaltfläche Hilfe wählt (oder die F1-Taste drückt).
.HelpContext	Die HelpContext-Eigenschaft wird verwendet, um automatisch das Hilfethema anzuzeigen, das in der HelpFile-Eigenschaft angegeben ist.
.Raise	Raise wird zum Erzeugen von Laufzeitfehlern verwendet und kann anstelle der Error-Anweisung verwendet werden. Raise ist hilfreich beim Erzeugen von Fehlern, wenn man Klassenmodule schreiben, da das Err-Objekt mehr Informationen zur Verfügung stellt als die Error-Anweisung. Mit der Raise-Methode kann zum Beispiel der ursprüngliche Auslöser des Fehlers in der Source-Eigenschaft angegeben werden, oder man kann auf die Hilfe für den Fehler verweisen usw.
.Clear	Mit Clear löscht man das Err-Objekt explizit, nachdem ein Fehler verarbeitet wurde. Dies ist zum Beispiel erforderlich, wenn man On Error Resume Next verwendet und die Fehler erst später verarbeitet. Die Clear-Methode wird automatisch aufgerufen, sobald eine der folgenden Anweisungen ausgeführt wird: ➔ Alle Arten von Resume-Anweisungen ➔ Exit Sub, Exit Function, Exit Property ➔ Jede On Error-Anweisung

Überprüfen der Fehlernummer

Bei der Überprüfung von Fehlernummern sollte in jedem Fall vermieden werden, logische Funktionen anzuwenden.

Folgendes Beispiel funktioniert:

```
Sub DivideByZero()

    Dim i As Integer
    On Error Resume Next
    i = 8 / 0
    If Err.Number Then
        MsgBox "Es trat ein Fehler auf: " & Err.Description
    End If

End Sub
```

Demzufolge sollte man annehmen, dass folgendes Beispiel ebenfalls funktioniert:

```
Sub DivideByZero()

    Dim i As Integer
    On Error Resume Next
    i = 8 / 0
    If Not Err.Number Then
        MsgBox "Es trat kein Fehler auf!"
    End If

End Sub
```

In diesem zweiten Beispiel wird die Meldung **"Es trat kein Fehler auf!"** auch dann angezeigt, wenn es zu einem Fehler kommt.

Für die **Not** Funktion der **If**-Anweisung ist nur **0 = False** und **–1 = True**.

Das heißt, dass in einer If-Anweisung eine 0 stets als False und ein Wert ungleich 0 stets als True gewertet werden.

Der Wert Err.Number in dem Beispiel besitzt den Wert 11 („Division durch Null“); dieser Wert ist ungleich 0 und führt demnach in der If Not-Anweisung zu True!!!

Fazit: Man sollte grundsätzlich auf den Wert (bzw. auf 0) prüfen und nicht mit logischen Funktionen die Fehlersituation überprüfen!

Bereinigen des Error-Objekts

Wenn ein Fehler auftritt, behält das Fehlerobjekt solange die Fehlerdaten, bis das Fehlerobjekt bereinigt wird.

```
Sub DivideByZero()

    Dim i As Integer
    On Error Resume Next
    i = 8 / 0
    If Err.Number <> 0 Then
        MsgBox "Es trat ein Fehler auf!"
    End If
    i = 8 / 4
    If Err.Number <> 0 Then
        MsgBox "Es trat ein Fehler auf!"
    End If

End Sub
```

In diesem Beispiel wird zweimal ein Fehlerhinweis angezeigt, obwohl nur die erste Division zu einem Fehler führt!

Das Fehlerobjekt sollte demnach sofort nach dem Fehler-Handling bereinigt werden; hierzu stellt und das Error.Objekt die Clear-Methode zur Verfügung:

```
Sub DivideByZero()

    Dim i As Integer
    On Error Resume Next
    i = 8 / 0
    If Err.Number <> 0 Then
        MsgBox "Es trat ein Fehler auf!"
    End If
    Err.Clear
    i = 8 / 4
    If Err.Number <> 0 Then
        MsgBox "Es trat ein Fehler auf!"
    End If
    Err.Clear

End Sub
```

Das Error-Objekt wird ebenfalls nach jeder Resume-Anweisung initialisiert.

Ausschalten des Error-Handlers

Ob man innerhalb einer Routine ein Fehlerhandling an und wieder ausschaltet, ist eher eine philosophische Frage. Grundsätzlich wird das Fehlerhandling durch die Anweisung **On Error Goto 0** ausgeschaltet:

```
Sub DivideByZero()

    Dim i As Integer
    On Error Resume Next
    i = 8 / 0       'Fehler wird unterdrückt
    On Error GoTo 0
    i = 8 / 0       'Fehler wird nicht behandelt

End Sub
```

In jedem Fall sollte der Error-Handler vor Beendigung der Routine explizit ausgeschaltet werden.

Gültigkeitsbereich von Error-Handlern

Ein Error-Handler wird beim Verlassen einer Sub-Routine, einer Function oder einer Property deaktiviert. Die On Error Anweisung ist beim Rücksprung in die aufrufende Prozedur nicht länger gültig.

```
Sub FirstSub()
    Dim i As Integer
    SecondSub
    i = 8 / 0 'hier kommt es zum behandelten Fehler!
End Sub

Sub SecondSub()
    Dim i As Integer
    On Error Resume Next
    i = 8 / 0 'dieser Fehler wird unterdrückt
End Sub
```

In nachfolgendem Beispiel wird trotz Fehlerunterdrückung die Fehlernachricht angezeigt:

```
Sub FirstSub()
    Dim i As Integer
    On Error Resume Next
    SecondSub
    i = 8 / 4
    If Err.Number <> 0 Then MsgBox "Fehler!"
End Sub

Sub SecondSub()
    Dim i As Integer
    On Error Resume Next
    i = 8 / 0
End Sub
```

Grund für diese Fehlermeldung ist ein nicht initialisiertes Error-Objekt beim Verlassen der Routine SecondSub. Dort wird zwar der Fehler 11 unterdrückt, aber der Fehler bleibt solange im Fehlerobjekt erhalten, bis das Objekt durch Err.Clear (oder eine entsprechenden Resume-Anweisung) initialisiert wird.

Die Anweisung i = 8 / 4 in der Routine FirstSub führt zwar nicht zu einem Fehler, aber das Error-Objekt besitzt noch die Fehlerinformationen aus der zuvor aufgerufenen SecondSub, so dass Err.Number immer noch den Wert 11 besitzt und es somit zur Anzeige der Fehlermeldung kommt! Die korrekte Implementierung der Error-Handler sieht wie folgt aus:

```
Sub FirstSub()

    Dim i As Integer
    Dim lngError As Long

    SecondSub

    On Error Resume Next
    i = 8 / 4
    lngError = Err.Number
    On Error GoTo 0
    If lngError <> 0 Then MsgBox "Fehler!"

End Sub

Sub SecondSub()

    Dim i As Integer
    On Error Resume Next
    i = 8 / 0
    On Error GoTo 0

End Sub
```

Fehlercodes im Aufrufstapel

Es ist nicht unbedingt notwendig, in jeder Routine ein Fehler-Handling zu implementieren, um eine Fehlermeldung zu erhalten. VB/A sucht bei einem Fehler den Aufrufstapel solange durch, bis es eine Fehlerbehandlung findet:

```
Sub MainSub()

1   On Error Resume Next
2   FirstSub
5   If Err.Number <> 0 Then MsgBox "Fehler!"
6   On Error GoTo 0

End Sub

Sub FirstSub()

3   SecondSub
    i = 8 / 4

End Sub

Sub SecondSub()

4   i = 8 / 0 'dieser Fehler führt zum Rücksprung in MainSub

End Sub
```

In diesem Beispiel kann man anhand der Zeilennummer verfolgen, wie das Programm abläuft.

Da in MainSub die Fehlerbehandlung unterdrückt wird, erfolgt nach Zeile 4 ein Rücksprung in MainSub und Anzeige der Fehlermeldung. Die Zeile **i = 8 / 4** in der Routine FirstSub wird in diesem Zusammenhang gar nicht durchgeführt!

Bei allen Möglichkeiten zur Fehlerbehandlung ist es immer besser, Fehler zu vermeiden anstelle Fehler abzufangen und anzuzeigen.

Die Möglichkeiten Fehler zu vermeiden sind sehr vielfältig und würden den Rahmen dieses Workshops sprengen.

Grundsätzlich sollte man aus „seinen“ Fehlern lernen und entsprechende Sicherheitsmechanismen einbauen, um einen einmal aufgetretenen Fehler möglichst nicht ein weiteres Mal auftreten zu lassen.

Kapitel 4

DEBUGGING IN VBA

Exkurs: Ein Bug ist ein Programmfehler. Der Name Bug (engl. Käfer, Wanze) geht auf die durch Insekten verursachten Fehler in Rechenmaschinen zurück.

In der Frühzeit der Datenverarbeitung, als Computer noch mit Relais als Schaltelementen funktionierten, störten manchmal Insekten die Datenverarbeitung. Sie krabbelten in die damals noch riesigen Computergehäuse und wurden zwischen den Schaltern zerquetscht. Die Bezeichnung Bug für Programmfehler wurde auch bei den heutigen Computern, die auf integrierten Schaltungen basieren, beibehalten. Gemeinhin wird die Bezeichnung Bug der Entwicklerin Grace Hopper zugewiesen, die 1945 tatsächlich einen Käfer in einem Relais entdeckt haben soll. Die Motte, die schuld war an einem Defekt, ist heute im Marinemuseum in Dahlgren, Virginia/USA ausgestellt. Jedoch wurde der Ausdruck Bug bereits im 19 Jh. zur Bezeichnung von mechanischen Defekten verwendet. (So 1878 schriftlich belegt in einem Brief von Thomas Alva Edison).

Selbst das beste Fehlerhandling reicht nicht immer aus, die Ursachen für Fehler herauszufinden.

Um nun solche Bugs (=Fehler) analysieren zu können, stellt uns VB bzw. VBA verschiedene sog. Debugging Mechanismen und –Tools zur Verfügung. Leider wird davon heutzutage viel zu wenig bis gar nicht Gebrauch gemacht. Es gibt tatsächlich VB/VBA-Programmierer, die nicht einmal diese Werkzeuge kennen!

Die konsequente Nutzung von Error-Handlern im Programmcode ist sicherlich wichtig und notwendig, doch durch das Abfangen von Fehlern und der Anzeige von Fehlernachrichten wird ein Programm auf lange Sicht nicht wirklich besser. Aus diesem Grund sollte man sich mit jedem auftretenden Fehler intensiv beschäftigen, um Maßnahmen zu ergreifen, dass dieser Fehler möglichst nicht mehr auftritt.

Debugging erfolgt im Code

Obwohl über optional einblendbare Symbolleisten Debugger-Funktionen auswählbar sind, werden nachfolgend die verfügbaren Tastenkombinationen aufgelistet die direkt oder indirekt zum Thema Debugging zur Verfügung stehen:

Um das zu tun	Muss das gedrückt werden
Eine Codezeile ausführen (Einzelschritt)	F8
Anweisungen zeilenweise ausführen, ohne Prozeduren aufzurufen (Prozedurschritt)	Umschalt + F8
Ausführen bis Cursor-Position	Strg + F8
Haltepunkt in der Cursor-Zeile ein- und ausschalten (Haltepunkt ein/aus)	F9
Alle Haltepunkte löschen	Strg + Umschalt + F9
Nächste Anweisung festlegen	Strg + F9
(Aktuellen Wert anzeigen)	Umschalt + F9
Den Fehlerbeseitigungscode ausführen oder den Fehler an die aufrufende Prozedur zurückgeben	Alt + F5
In die Fehlerbehandlungroutine einsteigen oder den Fehler an die aufrufende Prozedur zurückgeben	Alt + F8
Prozedur abschließen	Strg + Umschalt + F8
Typ und Gültigkeit einer Variablen anzeigen	Strg + I
Aufruf/Anzeige des Direktfensters	Strg + G
Sprung vor Variablen-/Funktionsdeklaration	Umschalt + F2
Rücksprung zur letzten Position	Strg + Umschalt + F2
(Anzeige der Aufrufliste)	Strg + L
Wechsel vom Objekt (Bsp.: UserForm) zum Codebereich	F7
Rücksprung vom Codebereich zum Objekt	Umschalt + F7
Aufruf des Objektkatalogs	F2

Je nach VB/VBA-Version stehen ggf. weitere Tastenkombinationen zur Verfügung.

Der Schlüssel zur Behebung von Programmfehlern ist der VB/VBA Haltemodus. Im Haltemodus läuft das Programm im Prinzip, wird aber an einer bestimmten Anweisung im Code in der Schwebe gehalten. Da das Programm aber dennoch aktiv ist, kann man sich in aller Ruhe die aktuellen Werte aller Variablen ansehen.

Praktischerweise kann man im Haltemodus den Code bearbeiten, notwendige Änderungen vornehmen oder spontan neue Zeilen hinzufügen, und das alles bei laufendem Programm.

Das Direktfenster

Mithilfe der Tastenkombination Strg + G wird das Direktfenster eingeblendet. In diesem Direktfenster können Codesegmente oder aber auch direkte mathematische Operationen ausgeführt werden.

Die Debug.Print Anweisung im Code veranlasst VB/A dazu, den entsprechenden Wert im Direktfenster anzuzeigen.

Variablen können im Direktfenster andere Werte zugewiesen werden.

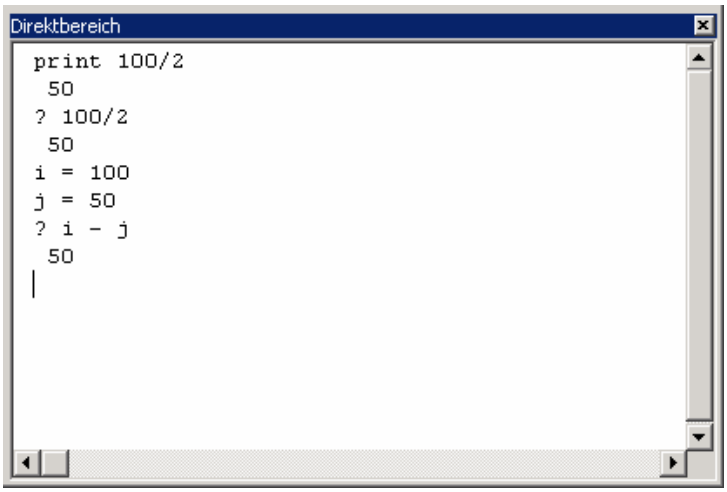

Abbildung 1 - Das Direktfenster

Das Lokal-Fenster

Wenn man genügend Platz zur Verfügung hat, sollte man das Lokal-Fenster während des Debugging im Haltemodus immer auf dem Bildschirm haben (Ansicht – Lokal-Fenster). Das Lokal-Fenster zeigt automatisch alle in der laufenden Prozedur zugänglichen Variablen, ihren Namen, Werte und Datentypen an. Weil der VBA-Editor diese Informationen nach jeder Ausführung einer Anweisung aktualisiert, bildet das Lokal-Fenster immer die aktuellen Werte ab.

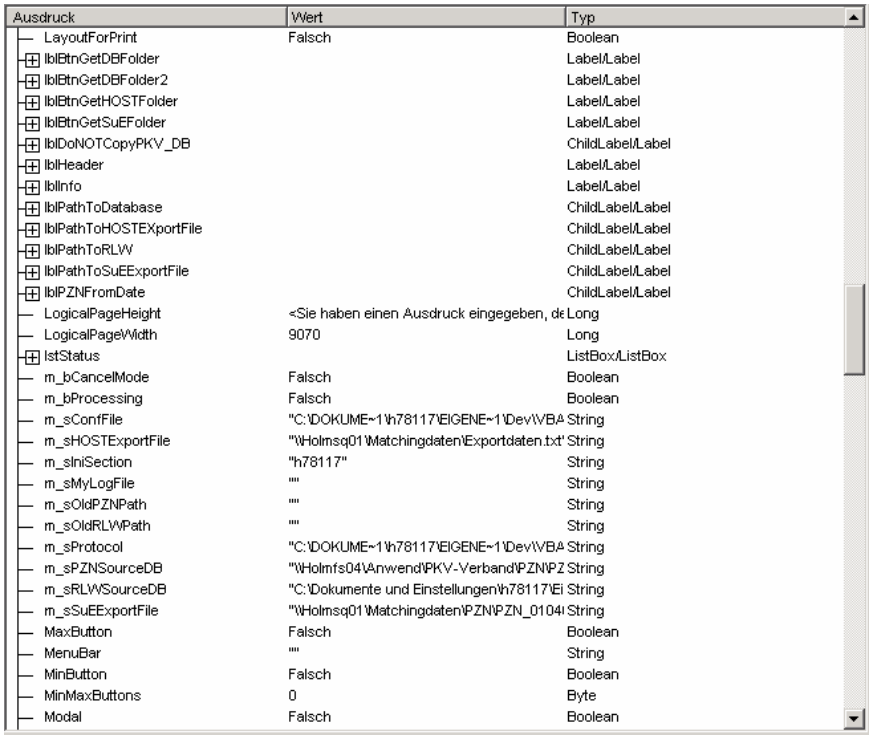

Abbildung 2 - Das Lokal-Fenster

Ausschnitt aus dem „Lokal-Fenster“ während dem Debugging einer Access-Applikation.

Das Überwachungsfenster

Das Überwachungsfenster funktioniert ähnlich dem Lokal-Fenster, nur das man hier selbst bestimmt, welche Werte angezeigt werden.

Dieses Fenster erscheint automatisch, wenn man einen **Überwachungsausdruck** definiert hat.
Um eine Variable für die Überwachung zu berücksichtigen, ist diese im Code zu markieren und im Kontextmenü (rechte Maustaste) der Menüpunkt „Überwachung hinzufügen...“ auszuwählen:

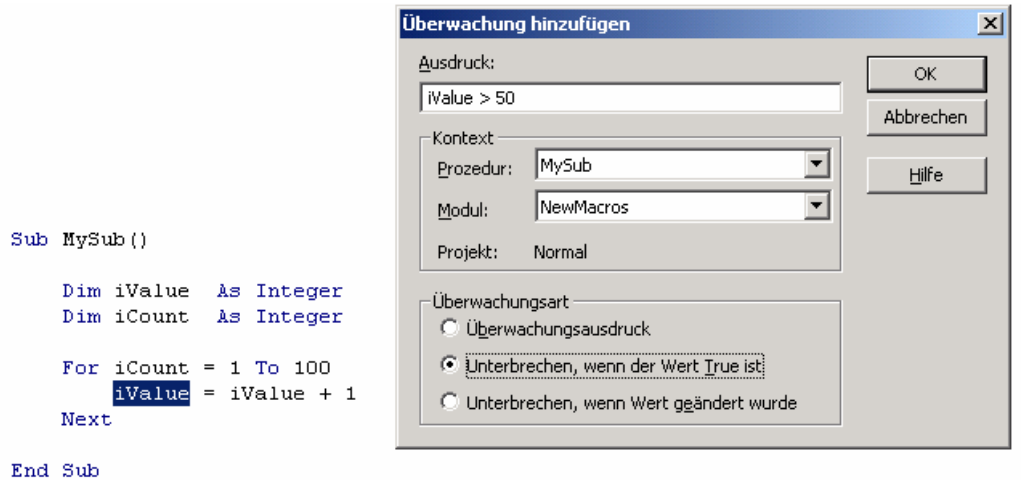

Abbildung 3 - Das Überwachungsfenster

Nun kann man nicht nur die Variable selbst ähnlich wie im Lokal-Fenster überwachen, sondern auch „Ausdrücke“ hinzufügen. In diesem Beispiel wurde festgelegt, dass die Programmausführung gestoppt wird, wenn den Wert der Variablen iValue > 50 ist.

Der Gültigkeitsbereich dieser Überwachung kann auf Prozedur- bzw. Modulebene entsprechend erweitert werden; dies führt jedoch u.u. zu einem deutlich langsameren Programmablauf.

Aber für das Aufspüren von Fehler wartet man ja gerne bisschen...

Kapitel 5

WISSENSWERTES

Gültigkeitsbereiche und Lebensdauer von Variablen

Jede Sprache hat Regeln für die Definition des Gültigkeitsbereichs und der Lebensdauer von Variablen und Prozeduren. Basic macht da keine Ausnahme, obwohl die Regeln sich leider anscheinend mit jeder Version ändern. Während der verwickelten Geschichte der Sprache haben alle die Versuche, neue Zusätze einzufügen, ohne wiederum dafür mit alten zu brechen, ein heilloses Durcheinander angerichtet. Die Basic-Landschaft ist übersät von aufgegebenen Modifikatoren für Reichweite und Gültigkeitsbereich, wie **Shared**, **Common** und jetzt auch noch **Global**. In der aktuellen Version haben selbst die besten Absichten die Angelegenheit nicht durchschaubarer gemacht:

Das Schlüsselwort **Dim** (das Wort mit der absolut schlechtesten Gedächtnisstütze für die Bedeutung) erstellt eine Variable mit lokalem Gültigkeitsbereich und vorübergehender Lebensdauer, wenn es innerhalb einer Prozedur verwendet wird (außer dass die Lebensdauer unbegrenzt ist, wenn die Prozedur als statisch deklariert wird). Wenn es aber außerhalb einer Prozedur verwendet wird, erstellt **Dim** eine Variable mit dem Gültigkeitsbereich **Private** und unbegrenzter Lebensdauer (trotz der Tatsache, dass die Lebensdauer in einem Standardmodul wiederum wichtig ist).

Private und **Public** sind Deklaratoren für Variablen; für Konstanten, Deklarationen, Typen und Prozeduren sind es aber Modifikatoren.

Falls du das durchschaust und **Dim** immer richtig verwendest, brauchst du diesen Kurs nicht. Für alle anderen könnte es hilfreich sein, sich einen Mythos zu erschaffen und vorzugeben, daran zu glauben. Hier sind die Geboten für einen möglichen Basic-Daten-Mythos:

1. Das Wort **Dim** bedeutet in der Eingeborenensprache eines Basic-Volksstammes im nordöstlichen Cathistan tatsächlich **Lokal**. Deshalb sollte man **Dim** nur für lokale Variablen verwenden. Es sieht zwar so aus, als wenn Basic **Dim** auch in anderen Zusammenhängen zulässt, der Code wird dann aber von bösen Geistern heimgesucht.
2. Verwende **Static** für lokale Variablen mit unbegrenzter Lebensdauer. Deklariere niemals Funktionen als **Static**, weil es die Bedeutung von **Dim** verändert. Außerdem wird man davon Warzen bekommen.
3. Verwende **Private** für Variablen mit modularem Gültigkeitsbereich. Widerstehe der Versuchung, in diesem Zusammenhang **Dim** zu verwenden, obwohl es für einige Zeit zu funktionieren scheint.
4. Verwende **Public** für Variablen mit globalem Gültigkeitsbereich. Vor langer, langer Zeit (nach anderen Quellen in der vorletzten Version) bedeutete **Global Public**. Einige Abweichler vom Wahren Glauben behaupten, dass es immer noch als eine Art

Huldigung an die böse Gottheit Kompatibilität funktioniert. Ignoriere diese Gerüchte.

5. Deklariere dynamische (in der Größe veränderbare) Datenfelder immer mit leeren Klammern und dem richtigen Variablen-Schlüsselwort. Die Behauptung, dass **Redim** ohne vorherige Deklaration auf lokale Variablen angewendet werden kann, führt unweigerlich in den Wahnsinn.
6. Gib immer **Public** oder **Private** für benutzerdefinierte Typen und Deklarationsanweisungen an. Bei Klassen- und Formularmodulen erzwingt Basic die Deklaration, bei Standardmodulen aber bietet es eine automatische Voreinstellung. Einige Experten glauben, dass sie diese Regel auch bei Konstanten und Prozeduren befolgen sollten. Wenn man jedoch die Standardeinstellung verwendet, wird man trotzdem weiterleben.

Gültigkeitsbereiche: Die nackten Fakten

Für all die eher prosaischen Leser, die mehr an Fakten als an Meinungen interessiert sind, sind in der nachfolgenden Tabelle die Standard-Gültigkeitsbereiche für verschiedene Elemente aufgelistet, die man in seinem Programm deklarieren kann:

	Standardmodule	Form- und Klassenmodule
Konstanten	Standard: **Private**	Standard: **Private**
	Private OK	**Private** OK
	Public OK	**Public** illegal
Benutzdefinierte Typen	Standard: **Public**	Standard: illegal
	Private OK	**Private** erforderlich
	Public OK	**Public** illegal
Deklarationsanweisungen	Standard: **Public**	Standard: illegal
	Private OK	**Private** erforderlich
	Public OK	**Public** illegal
Variablen	Standard: **Private** *)	Standard: **Private** *)
	Private OK	**Private** OK
	Public OK	**Public** definiert Eigenschaft
Funktionen und Sub-Prozeduren	Standard: **Public**	Standard: definiert Methode
	Private OK	**Private** OK
	Public OK	**Public** erstellt Methode
Eigenschaften	Standard: **Public**	Standard: **Public**
	Private OK	**Private** OK
	Public OK	**Public** OK

*) Standard bedeutet Deklaration mit **Dim**, nicht mit **Private** oder **Public**.

Pass auf Deine Typen auf...

Einer der gefährlichsten Fehler in Visual Basic (For Applications) tritt auf, wenn man versucht, mehrere Deklarationen in einer Zeile unterzubringen. Zum Beispiel:

```
Dim c, i, h As Long
```

Man könnte jetzt glauben, drei Longinteger definiert zu haben. Falsch! Man hat stattdessen zwei Varianten (der Standardtyp für die ersten beiden) und eine explizit deklarierte Longinteger-Variable geschaffen. Hier eine andere Variante:

```
Dim c As Long, i, h
```

Richtig ist:

```
Dim c As Long, i As Long, h As Long
```

bzw.:

```
Dim c As Long
Dim i As Long
Dim h As Long
```

Datentypen (Zusammenfassung)

Die folgende Tabelle enthält die von Visual Basic unterstützten Datentypen sowie deren Speicherbedarf und Wertebereiche.

Datentyp	Speicherbedarf	Wertebereich
Byte	1 Byte	0 bis 255
Boolean	2 Bytes	True oder False
Integer	2 Bytes	-32.768 bis 32.767
Long (lange Ganzzahl)		-2.147.483.648 bis 2.147.483.647
Single (Gleitkommazahl mit einfacher Genauigkeit)	4 Bytes	-3,402823E38 bis -1,401298E-45 für negative Werte; 1,401298E-45 bis 3,402823E38 für positive Werte.
Double (Gleitkommazahl mit doppelter Genauigkeit)	8 Bytes	-1,79769313486232E308 bis -4,94065645841247E-324 für negative Werte; 4,94065645841247E-324 bis 1,79769313486232E308 für positive Werte.
Currency (skalierte Ganzzahl)	8 Bytes	-922.337.203.685.477,5808 bis 922.337.203.685.477,5807

Datentyp	Speicherbedarf	Wertebereich
Decimal	14 Bytes	+/-79.228.162.514.264.337.593.543.950.335 ohne Dezimalzeichen; +/-7,9228162514264337593543950335 mit 28 Nachkommastellen; die kleinste Zahl ungleich Null ist +/-0,0000000000000000000000000001.
Date	8 Bytes	1. Januar 100 bis 31. Dezember 9999.
Object	4 Bytes	Beliebiger Verweis auf ein Objekt vom Typ Object.
String (variable Länge)	10 Bytes plus Zeichenfolgenlänge	0 bis ca. 2 Milliarden.
String (feste Länge)	Zeichenfolgenlänge	1 bis ca. 65.400
Variant (mit Zahlen)	16 Bytes	Numerische Werte im Bereich des Datentyps Double.
Variant (mit Zeichen)	22 Bytes plus Zeichenfolgenlänge	Wie bei String mit variabler Länge.
Benutzerdefiniert (mit **Type**)	Zahl ist von Elementen abhängig	Der Bereich für jedes Element entspricht dem Bereich des zugehörigen Datentyps.

Datenfelder eines beliebigen Datentyps benötigen 20 Bytes im Speicher, vier Bytes für jede Datenfelddimension und die Anzahl an Bytes, die für die eigentlichen Daten benötigt werden. Der für die Daten benötigte Speicher kann durch Multiplikation der Anzahl an Datenelementen mit der Größe eines einzelnen Elements ermittelt werden. Die Daten in einem eindimensionalen Datenfeld, das vier Elemente vom Typ Integer mit jeweils zwei Bytes enthält, belegen zum Beispiel acht Bytes. Insgesamt benötigt das Datenfeld die acht Bytes für die Daten zuzüglich 24 Bytes für Verwaltung, also 32 Bytes.

Ein Wert vom Typ Variant, der ein Datenfeld enthält, benötigt 12 Bytes zusätzlich zu dem Speicher, der für das Datenfeld alleine benötigt wird.