PEARSON Education

Markt+Technik

jetzt lerne ich

VBA mit Access

Der einfache Einstieg in die Makro- und Datenbankprogrammierung

SAID BALOUI

Markt+Technik

Kapitel 3

jetzt lerne ich

Grundlegende Sprachelemente

Den Umgang mit der Programmierumgebung beherrschen Sie inzwischen; kommen wir zum Programmieren selbst.

Ich erläutere zunächst die grundlegenden VBA-Sprachelemente, angefangen bei Anweisungen wie *Print*, *MsgBox* und *InputBox* zur Ausgabe von Daten auf dem Bildschirm bzw. zur Eingabe von Daten per Tastatur.

Die danach besprochenen Möglichkeiten, Textfelder in Formularen zur Ausgabe von Daten zu benutzen, verleihen Ihren Programmen bereits eine professionellere Optik.

Nach diesen Ein-/Ausgabegrundlagen wird die Sprache selbst systematisch dargestellt. Das beginnt mit der Erläuterung der verschiedenen Bestandteile von Prozeduren wie Code- und Kommentarzeilen.

Danach dreht sich ein Großteil des folgenden Kapitels um Variablen, da es ohne den ständigen Einsatz von Variablen nicht möglich ist, wirklich sinnvolle Programm zu schreiben. Ich erläutere die verschiedenen verfügbaren Datentypen und zeige Ihnen, wie Sie Variablen des betreffenden Typs deklarieren und definieren – und wie die damit verwandten Konstanten deklariert werden.

Außer auf eigendefinierte Konstanten gehe ich auch auf die in Access und VBA vordefinierten Konstanten ein, deren Anwendung Programme übersichtlicher und besser lesbar macht, da sie es ermöglichen, einen nichtssagenden Ausdruck wie

```
MsgBox("Dateiname:", 1 + 32)
```

durch die folgende weit aussagekräftigere Variante zu ersetzen:

```
MsgBox("Dateiname:", vbOKCancel + vbQuestion)
```

Der umfangreiche Rest des Kapitels widmet sich Prozeduren. Als Einführung erläutere ich zunächst Funktionsprozeduren wie die in VBA eingebauten Funktionen *Sqr* (Wurzelberechnung) oder *InStr* (prüfen, ob sich eine bestimmte Zeichenfolge in einer Zeichenkette befindet).

Dann geht es um die Definition eigener Prozeduren, salopp ausgedrückt: um das Verpacken Ihrer Codezeilen in nette Kästchen, deren teilweise höchst unterschiedliche Eigenschaften Sie im Detail kennen müssen, um effiziente Programme schreiben zu können.

Ich erläutere die verschiedenen Typen eigendefinierter Prozeduren und zeige Ihnen, welche Möglichkeiten VBA zur Verfügung stellt, um Variablen und Konstanten an Prozeduren zu übergeben, angefangen bei benannten Argumenten bis hin zur Übergabe als Wert oder als Referenz.

Anschließend geht es um die Geltungsbereiche sowohl von Prozeduren als auch von Variablen und um die Möglichkeiten, private bzw. öffentliche Prozeduren zu erstellen bzw. private und öffentliche Variablen zu deklarieren.

Ich erläutere, welche Probleme es gibt, wenn mehrere gleichnamige Prozeduren in einer Datenbank existieren oder gar eine Prozedur aufgerufen werden soll, die sich in einer anderen Datenbank befindet, und zeige, wie diese Probleme gelöst werden.

3.1 Ein- und Ausgabeanweisungen

3.1.1 Bildschirmausgaben – Debug.Print und MsgBox

Die wohl grundlegendste Ausgabeanweisung ist die *Print*-Methode. Wie Sie wissen, können Sie Aufrufe dieser Methode im Testfenster eintippen, um damit in diesem Fenster Daten auszugeben.

Debug.Print

Zusätzlich können Sie diese Anweisung auch in Ihren Programmen verwenden, um Daten im Testfenster auszugeben, während ihr Programm abgearbeitet wird. Dann müssen Sie jedoch angeben, dass als »Ausgabeobjekt« das Testfenster gemeint ist, und folgende Form verwenden:

```
Debug.Print Ausdruck
```

»Debug« repräsentiert das Testfenster. *Debug.Print* bedeutet somit, dass die im nachfolgenden *Ausdruck* angegebenen Daten in diesem Fenster ausgegeben werden sollen. Das passiert, auch wenn das Testfenster gerade geschlos-

sen ist: Öffnen Sie es, nachdem Ihre Prozedur beendet ist, sehen Sie darin die zuvor von der Prozedur im Testfenster ausgegebenen Daten.

Jedem Aufruf von *Print* folgt ein Zeilenumbruch.

Zwei aufeinander folgende Aufrufe wie

```
Debug.Print "Test"
Debug.Print "Noch ein Test"
```

geben daher die beiden Zeichenketten »Test« und »Noch ein Test« nicht neben-, sondern untereinander aus.

Endet *Ausdruck* mit einem Semikolon, wird dieser Zeilenumbruch unterdrückt.

Daher gibt

```
Debug.Print "Test";
Debug.Print "Noch ein Test"
```

im Testfenster die Zeichenkette »TestNoch ein Test« aus.

Endet *Ausdruck* mit einem Komma, wird der Zeilenvorschub ebenfalls unterdrückt, die folgende Ausgabe jedoch in die nächste Bildschirmzone verlegt, wodurch für ein wenig Abstand zwischen aufeinander folgenden Ausgaben gesorgt wird.

Ein Beispiel:

```
Debug.Print "Test",
Debug.Print "Noch ein Test"
```

gibt somit »Test Noch ein Test« aus.

Mit der *Print*-Methode können beliebig viele Ausgaben aneinander gereiht werden.

Beispielsweise gibt

```
Debug.Print "Ein"; " "; "Test"
```

die Zeichenkette »Ein Test« aus und

```
Debug.Print "MWSt:"; x
```

die Zeichenkette »MWSt: 5«, falls die Variable *x* momentan den Wert 5 repräsentiert.

Um Ausgaben exakt zu positionieren, verwenden Sie die *Tab*-Anweisung:

```
Debug.Print Tab(x); "Zeichenkette"
```

Tab positioniert den Cursor vor der nächsten Ausgabe auf das Zeichen Nummer *x* der Spalte. *Tab*

Beispielsweise gibt

```
Debug.Print Tab(10); "Willi"; Tab(20); "Maier"
```

ab Spalte 10 die Zeichenkette »Willi« und ab Spalte »20« die Zeichenkette »Maier« aus.

MsgBox

Debug.Print erfordert, dass Sie das Testfenster öffnen, um sich die Ausgaben Ihres Programms anzuschauen. Die Funktion *MsgBox* ist wesentlich angenehmer zu handhaben, da die Ausgaben Windows-gerecht in einem kleinen Dialogfeld präsentiert werden. Die Syntax (leicht vereinfacht):

```
MsgBox(Meldung [,Schaltflächen] [,Titel])
```

»Meldung« ist die Zeichenkette, die im Dialogfeld erscheinen soll. Sie kann maximal 1024 Zeichen enthalten und wird von Access bei Bedarf automatisch umbrochen.

Ein Beispiel (Bild 3.1).

Bild 3.1: Prozedur »Meldungsdialogfeld«

Um die abgebildete Funktion *Meldungsdialogfeld* aufzurufen, öffnen Sie das Direktfenster und geben darin *Call Meldungsdialogfeld* ein. Daraufhin öffnet die in der Prozedur aufgerufene *MsgBox*-Funktion das abgebildete Dialogfeld (auf den nach dem Schließen des Dialogfelds im Direktfenster ausgegebenen Funktionswert 1 gehe ich in Kürze ein).

Zeilenumbruch

Durch Einfügen des ASCII-Codes 13 (Wagenrücklauf) in die Zeichenkette können Sie Zeilenumbrüche selbst definieren. Dazu fügen Sie dort, wo eine neue Ausgabezeile beginnen soll, den ASCII-Code 13 ein und benutzen dazu folgende Syntax:

```
MsgBox("Zeichenkette1" + Chr(13) + "Zeichenkette2" + Chr(13) +
"Zeichenkette3" + Chr(13) + " ... ")
```

»Titel« ist eine Zeichenkette, die den Inhalt der Titelzeile des Dialogfelds bestimmt, beispielsweise »Mein Dialogfeld«. Ohne dieses Argument verwendet Access immer die Überschrift »Microsoft Access«. *Titel*

»Schaltflächen« ist eine Zahl, die bestimmt, welche Schaltflächen das Dialogfeld enthält. Die Zahl ergibt sich aus der Addition einzelner Werte, wobei folgende Werte zur Verfügung stehen: *Schaltflächen*

Tabelle 3.1: Argument »Schaltflächen«

Wert	Bedeutung
Schaltflächen	
0	Nur Schaltfläche »OK« anzeigen
1	»OK« und »Abbrechen« anzeigen
2	»Abbrechen«, »Wiederholen« und »Ignorieren« anzeigen
3	»Ja«, »Nein« und »Abbrechen« anzeigen
4	»Ja« und »Nein« anzeigen
5	»Wiederholen« und »Abbrechen« anzeigen
Symbolart	
0	Kein Symbol
16	Stopp-Symbol
32	Fragezeichen-Symbol
48	Ausrufezeichen-Symbol
64	Informations-Symbol
Standardschaltfläche	
0	Erste Schaltfläche
256	Zweite Schaltfläche
512	Dritte Schaltfläche
768	Vierte Schaltfläche
Bindungsart	
0	Anwendungsgebunden (bis zur Beantwortung nur Wechsel zu anderer Anwendung möglich)
4096	Systemgebunden (auch kein Wechsel zu anderer Anwendung möglich)

Ohne das Argument »Schaltflächen« bzw. mit dem Wert 0 wird nur die »OK«-Schaltfläche angezeigt. Der Wert 1 würde beispielsweise zusätzlich die Schaltfläche »Abbrechen« einfügen.

Soll nicht die erste Schaltfläche »OK«, sondern die zweite Schaltfläche »Abbrechen« die durch ↵ aktivierte Standardschaltfläche sein, addieren Sie zu dieser 1 den Wert 256, übergeben also 257. *Standardschaltfläche*

Symbole

Soll zusätzlich ein Fragezeichensymbol im Dialogfeld erscheinen, addieren Sie zusätzlich 32 und übergeben somit 289:

```
MsgBox("Demo" + Chr(13) + " der Funktion MsgBox", 289, "Test")
```

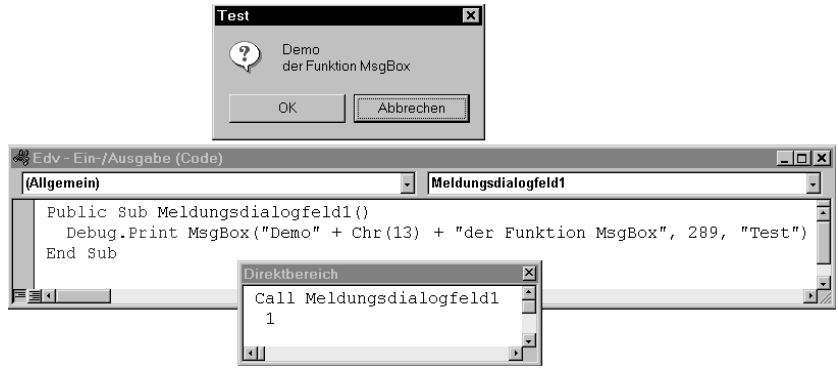


Bild 3.2: Prozedur »Meldungsdialogfeld1«

Das Argument »Test« für »Titel« gibt die Zeichenkette »Test« in der Titelleiste aus.

Die Funktion *MsgBox* übergibt nach dem Schließen des Dialogfelds einen der folgenden Funktionswerte, der im Beispiel nach dem Schließen des Dialogfelds mit *Debug.Print* im Direktfenster ausgegeben wird:

Tabelle 3.2: Übergebener Funktionswert

Wert	Schaltfläche
1	»OK«
2	»Abbrechen«
3	»Abbruch«
4	»Wiederholen«
5	»Ignorieren«
6	»Ja«
7	»Nein«

3.1.2 Benutzereingaben – InputBox und Formularfelder

InputBox

Ebenso einfach anzuwenden ist die Funktion *InputBox*. Mit ihr können Sie Eingaben in einem Dialogfeld entgegennehmen:

```
InputBox(Eingabeaufforderung [, [,Titel][,[Standard][,XPosition,
YPosition]]])
```

»Eingabeaufforderung« ist wieder eine im Dialogfeld anzuzeigende Zeichenkette und »Titel« erneut die Dialogfeldüberschrift.

»Standard« ist eine Zeichenkette, die im Eingabefeld vorgegeben werden soll.

»XPosition« und »YPosition« legen den horizontalen/vertikalen Abstand des linken/oberen Dialogfeldrands vom linken/oberen Bildschirmrand fest.

Geben Sie »Titel« nicht an, bleibt die Titelleiste leer. Ohne Positionsangaben wird das Dialogfeld in der Bildschirmmitte zentriert (Bild 3.3).

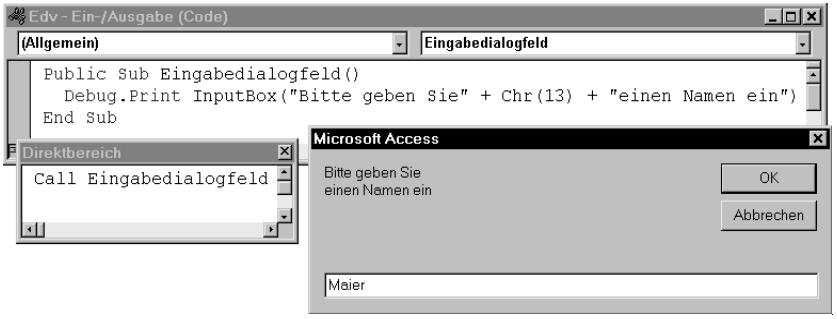


Bild 3.3: Prozedur »Eingabedialogfeld«

InputBox fordert den Anwender zu einer Eingabe auf. Gibt er wie hier die Zeichenkette »Maier« ein und aktiviert er »OK«, übergibt *InputBox* die eingegebene Zeichenkette als Funktionswert.

In diesem Beispiel wird die Eingabe anschließend mit *Debug.Print* im Direktfenster ausgegeben. In der Praxis könnten Sie den eingegebenen Namen stattdessen beispielsweise in einer Tabelle speichern.

Gibt der Anwender nichts ein oder bricht er die Eingabe mit »Abbrechen« ab, übergibt *InputBox* eine leere Zeichenkette »""« der Länge 0.

InputBox gibt immer eine Zeichenkette zurück (den Datentyp *String*) – auch dann, wenn der Anwender eine Zahl wie 22,3 eingab!

Benötigen Sie die Zahl als echte *Zahl* und nicht als Folge einzelner Zeichen, weil Sie sie anschließend einer so genannten numerischen Variablen zuweisen wollen, müssen Sie sie mit der *Value*-Funktion in eine echte Zahl umwandeln.

Beispielsweise ermittelt *x# = Value(s$)* den numerischen Wert der in einer Stringvariablen *s$* gespeicherten Zeichenfolge wie »23,4« und weist der numerischen Variablen *x#* daher den Wert 23,3 zu.

Formularfelder

Statt für Ausgaben das Textfenster zu benutzen oder immerhin wesentlich komfortabler Ein-/Ausgaben über ein kleines Dialogfeld abzuwickeln, gibt es für beides eine weitere Alternative: Sie können auch Textfelder von Formularen für Ein- oder Ausgaben verwenden.

Besitzt ein Textfeld eines Formulars namens »Ein-/Ausgabe« den Steuerelementnamen (Eigenschaft »Name«) »Eingabe«, repräsentiert der Ausdruck

```
Forms![Ein-/Ausgabe]![Eingabe]
```

den Inhalt dieses Steuerelements, also die Zeichenkette, die der Benutzer in dieses Textfeld eingab.

Sie können mit dieser Zeichenkette nun anstellen, was Sie wollen; Sie könnten sie beispielsweise mit

```
Debug.Print Forms![Ein-/Ausgabe]![Eingabe]
```

im Direktfenster ausgeben oder mit

```
MsgBox Forms![Ein-/Ausgabe]![Eingabe]
```

in einem Dialogfeld ausgeben; oder sie in einer Tabelle speichern.

Umgekehrt können Sie Textfelder verwenden, um Ausgaben Ihres Programms darin anzuzeigen. Um im Textfeld »Ausgabe« den Text »Hallo« anzuzeigen, verwenden Sie den Ausdruck

```
Forms![Ein-/Ausgabe]![Ausgabe] = "Hallo"
```

Dieser Ausdruck »weist« dem Textfeld die Zeichenkette »Hallo« als neuen Inhalt zu.

Ein Beispiel dafür enthält das folgende Formular (Bild 3.4).

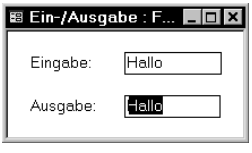


Bild 3.4: Formular »Ein-/Ausgabe«

Geben Sie im oberen Textfeld (Steuerelementname »Eingabe«) einen Text ein, erscheint er nach dem Beenden der Eingabe automatisch auch im unteren Textfeld (Steuerelementname »Ausgabe«).

Dafür verantwortlich ist die folgende Ereignisprozedur, die an die Eigenschaft »NachAktualisierung« des oberen Textfelds gebunden ist (Bild 3.5).

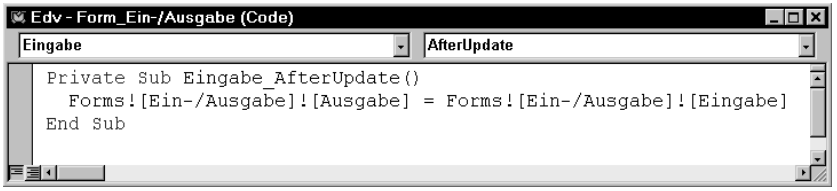


Bild 3.5: KlassenProzedur »Eingabe_AfterUpdate«

Der erste Teil *Forms![Ein-/Ausgabe]![Ausgabe]* = des Ausdrucks weist dem Textfeld »Ausgabe« einen Wert zu; und zwar jenen Wert, den der Ausdruck *Forms![Ein-/Ausgabe]![Eingabe]* besitzt, der sich rechts vom Gleichheitszeichen befindet: also den aktuellen Inhalt des Textfelds »Eingabe«.

3.2 Programm- und Kommentarzeilen

Grundlage von VBA-Programmen sind einzelne Codezeilen, die eine oder mehrere durch Doppelpunkte voneinander getrennte Anweisungen enthalten:

```
Debug.Print "Ein Test": Debug.Print "Noch ein Test"
```

Beide Anweisungen werden nacheinander ausgeführt, so als würden sie sich in zwei getrennten Programmzeilen befinden.

Kommentare

Nicht jede Anweisung bewirkt etwas. Die Anweisung *Rem* leitet ebenso wie das Zeichen »'« (⇧+#) einen Kommentar ein:

```
Rem Dies ist ein Kommentar
'   Dies ist ein Kommentar
```

Trifft Access beim Interpretieren einer Programmzeile auf eines dieser beiden Kommentarzeichen, ignoriert es den gesamten Rest der Zeile. Diese Eigenschaft betrifft jedoch nicht den Programmtext *vor* dem Kommentarzeichen. Daher können Sie eine Programmzeile zum Beispiel so kommentieren:

```
Debug.Print "Gleich kommt ein Kommentar"  :   Rem Ein Kommentar
```

oder

```
Debug.Print "Gleich kommt ein Kommentar"      'Ein Kommentar
```

Einen mit dem Kommentarzeichen »'« eingeleiteten Kommentar müssen Sie ausnahmsweise nicht per Doppelpunkt von dem zu kommentierenden Programmcode trennen, so dass es sich aus optischen Gründen anbietet, statt *Rem* dieses Zeichen zu verwenden.

DoCmd

Die Anweisung *DoCmd* (DoCommand = Kommando ausführen) können Sie benutzen, um eine beliebige Access-Aktion auszuführen, wobei Sie im Gegen-

satz zu Makros jedoch die englischen Aktionsbezeichnungen verwenden müssen:

```
DoCmd Aktionsname Argumente
```

Beachten Sie, dass die einer Aktion übergebenen Argumente im Gegensatz zu Funktionsargumenten nicht in Klammern gesetzt werden müssen!

Ein Beispiel: Die Syntax der Aktion *ApplyFilter*, die ein Filter auf ein Formular anwendet (eine Abfrage oder eine SQL-WHERE-Klausel), lautet

```
DoCmd ApplyFilter [Filtername] [, Bedingung]
```

Um diese Aktion während des Ablaufs einer Prozedur auszuführen und dabei im aktiven Formular nur Datensätze anzuzeigen, deren Postleitzahl größer oder gleich 50000 ist, verwenden Sie folgende Anweisung:

```
DoCmd ApplyFilter , "[Plz] >= 50000"
```

Das Komma repräsentiert das nicht angegebene optionale Argument *Filtername*. Lassen Sie optionale Argumente weg, geben Sie jedoch in der Syntaxliste danach folgende Argumente an, muss jedes der vorangehenden und nicht benutzten Argumente durch ein Komma ersetzt werden!

3.3 Variablen und Konstanten

Ist im Dialogfeld des Befehls EXTRAS | OPTIONEN... die Option »Variablen-Deklaration erforderlich« aktiviert, wird in den Deklarationsbereich jedes neu erstellten Moduls die Anweisung

```
Option Explicit
```

eingefügt, die Sie *zwingt*, alle darin verwendeten Variablen mit *Dim* zu deklarieren! Da ich *Dim* jedoch erst im Kapitel 3.3.2, »Variablen deklarieren«, erläutere, sollten Sie diesen »Deklarationszwang« bis dahin ausschalten, indem Sie

- entweder die Option »Variablen-Deklaration erforderlich« deaktivieren und danach ein neues Modul erstellen
- oder diese Anweisung im Deklarationsbereich des Moduls, das Sie momentan verwenden, löschen.

Jede Programmiersprache kennt zwei grundlegende Datentypen, Zahlen und Zeichen, besser gesagt Zeichen*ketten*, so genannte »Strings« (»Maier«, »Willi« etc.), die in Anführungszeichen eingeschlossen werden:

```
Test.Drucke 10
Test.Drucke "Test"
```

Zuweisungen

Auf beide Datenarten kann auch indirekt über Variablen zugegriffen werden. Dazu werden die Daten irgendwo im Speicher mit der *Let*-Anweisung unter einem frei wählbaren Namen abgelegt, dem Variablennamen:

```
Let Variablenname = Ausdruck
```

Man sagt, der Variablen *Variablenname* wird der Wert *Ausdruck* zugewiesen. Anschließend können Sie in beliebigen Anweisungen unter Angabe dieses Namens auf die gespeicherten Daten zugreifen, sie lesen oder verändern. Das Schlüsselwort *Let* ist optional und wird meist weggelassen, so dass sich folgende Syntax ergibt:

```
Variablenname = Ausdruck
```

Variablennamen

Variablennamen

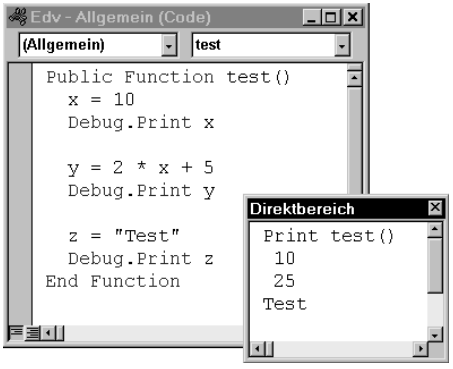
- müssen mit einem Buchstaben beginnen,
- dürfen nur Buchstaben, Ziffern und das Zeichen »_« enthalten,
- können maximal 200 Zeichen lang sein,
- dürfen keine Satzzeichen (».«, »;« etc.) und keine Leerzeichen enthalten
- und dürfen nicht mit einem reservierten Schlüsselwort identisch sein (versuchen Sie also nicht, eine Variable *Print*, *Debug*, *MsgBox* oder ähnlich zu nennen).

Die gleichen Regeln gelten übrigens auch für Konstanten- und Prozedurnamen!

Ein kleines Beispiel für die Anwendung von Variablen (das nur funktioniert, wenn, wie zu Beginn dieses Kapitels erläutert, der Zwang zur Deklaration von Variablen mit *Dim* aufgehoben wurde!) (Bild 3.6).

Nach dem Aufruf der Funktion *Test* im Direktfenster mit *Print Test()* weist die Anweisung *x = 10* der Variablen *x* den Wert 10 zu, den Access irgendwo im Rechnerspeicher ablegt. *Debug.Print x* greift unter Angabe dieses Namens auf den zugehörigen Wert zu und gibt ihn im Direktfenster aus.

Bild 3.6: Prozedur »Test«



Die zweite Anweisung *y = 2 * x + 5* zeigt, dass bei einer Zuweisung auch rechts vom Gleichheitszeichen, also im Zuweisungsausdruck, der den zu speichernden Wert festlegt, Variablen verwendet werden können: *x* besitzt nach der vorhergehenden Zuweisung den Wert 10. Der Ausdruck *2 * x + 5* besitzt daher den Wert 25. Er wird einer weiteren Variablen *y* zugewiesen und der dadurch definierte Inhalt 25 dieser Variablen wird ebenfalls im Direktfenster ausgegeben.

Die folgende Anweisung zeigt, dass Sie einer Variablen auch Zeichenketten zuweisen können – und dass die betreffende Zeichenkette dabei in Anführungszeichen eingeschlossen werden muss!

Jede Zuweisung an eine bereits bestehende Variable überschreibt den momentan darin gespeicherten Wert! Weisen Sie mit *x = 10* der Variablen *x* den Wert 10 und sofort danach mit *x = 20* den Wert 20 zu, würde eine darauf folgende Anweisung *Debug.Print x* die Zahl 20 ausgeben, also jene Zahl, die *x* zuletzt zugewiesen wurde und die daher den zuvor in *x* gespeicherten Wert 10 überschrieb.

3.3.1 Die verfügbaren Datentypen

Variant

Benutzen Sie eine Variable, ohne sie zuvor mit der Anweisung *Dim* zu deklarieren (dazu muss der Deklarationszwang deaktiviert werden, siehe Kapitel 3.3, »Variablen und Konstanten«) und ohne ein zusätzliches Typdeklarationszeichen zu verwenden, besitzt sie den Standard-Datentyp *Variant*.

Dieser Datentyp kann beliebige Datenarten aufnehmen, Zahlen und Zeichen, ein Datum oder eine Zeitangabe. Er ist außerdem sehr komfortabel einzusetzen, da VBA – falls notwendig – einen Datentyp automatisch in einen benötigten anderen Typ umwandelt.

Ein Beispiel:


```
x = "12"
y = x – 10
Debug.Print y
```

Listing 3.1: Typkonvertierung

x = "12" weist der Variablen *x* die Zeichenkette »12« zu, und *y = x – 10* subtrahiert davon 10 und weist das Ergebnis der Variablen *y* zu – subtrahiert also eine Zahl von einer Zeichenkette!

Typkonvertierung

Diesen Unfug würde außer VBA jede andere Programmiersprache mit einer Fehlermeldung ahnden. VBA nimmt jedoch eine automatische Typkonvertierung vor: Der Inhalt von *x* soll bei dieser Subtraktion offenbar als Zahl verwendet werden. Daher wandelt VBA die *Zeichenkette* »12« in die *Zahl* 12 um, von der nun problemlos die Zahl 10 subtrahiert werden kann.

Eine Fehlermeldung erhalten Sie nur mit Ausdrücken, bei denen eine solche Konvertierung nicht möglich ist:

```
x = "Otto"
y = x – 10
Debug.Print y
```

Listing 3.2: Typkonvertierung nicht möglich

Dieser Versuch, von der Zeichenkette »Otto« die Zahl 10 zu subtrahieren, ist selbst VBA zuviel und Sie erhalten die Fehlermeldung »Typen unverträglich«.

Der Datentyp *Variant* ist zwar sehr komfortabel, aber Access ist ständig mit der Überprüfung von Datentypen und gegebenenfalls mit entsprechenden Typumwandlungen beschäftigt, was Rechenzeit kostet.

All das entfällt mit eingeschränkteren Variablentypen. Geben Sie von vornherein bekannt, dass in *x* nur Zahlen gespeichert sind, muss Access in einem Ausdruck der Art *y = x – 10* die in *x* enthaltene Datenart nicht mehr überprüfen, um beispielsweise festzustellen, dass der Inhalt von *x* eine Zeichenkette wie »12« ist, die erst in die Zahl 12 umgewandelt werden muss, um davon die Zahl 10 subtrahieren zu können.

Typdeklarationszeichen

Da diese zusätzlichen Prüfungen und Umwandlungen Ihr Programm verlangsamen, sollten Sie aus Effizienzgründen nicht ständig den Datentyp *Variant* verwenden.

Geben Sie Access stattdessen entsprechend der folgenden Tabelle mit einem Typdeklarationszeichen bekannt, welche Datenarten die betreffende Variable aufnehmen soll (bei den Datentypen ohne spezielles Typdeklarationszeichen

müssen Sie die im Kapitel 3.3.2, »Variablen deklarieren«, erläuterte Technik verwenden):

Tabelle 3.3: Datentypen

Datentypname	Zeichen	Zulässige Werte	Belegter Platz
Byte	keines	0 bis 255 (nur ganze Zahlen)	1 Byte
Boolean (Access 2.0: nicht vorhanden)	keines	Einer der beiden Wahrheitswerte *True* und *False*	2 Byte
Integer	%	–32.768 bis 32.767 (nur ganze Zahlen)	2 Byte
Long	&	–2.147.483.648 bis 2.147.483.647 (nur ganze Zahlen)	4 Byte
Single	!	–3,402823E38 bis 3,402823E38 (auch Dezimalzahlen; sechsstellige Genauigkeit)	4 Byte
Double	#	–1,79769313486232E308 bis 1,79769313486232E308 (auch Dezimalzahlen; zehnstellige Genauigkeit)	8 Byte
Currency	@	Zwischen –922337203685477,5808 und 922337203685477,5808	8 Byte
Date	keines	1.Jan 100 bis 31.Dez 9999	8 Byte
Object	keines	Verweis auf ein beliebiges Objekt	4 Byte
String variabler Länge	$	Zeichenkette, max. 2 Milliarden Zeichen	10 Byte plus die Textlänge
String fester Länge	keines	Zeichenkette, max. 2 Milliarden Zeichen	die Textlänge
Variant	keines	beliebige Daten	je nach Datenart (Zahlen: 16 Byte; Zeichenketten: 22 Byte plus die Textlänge)

Datum/Uhrzeit-Eingabe

Beachten Sie, dass ein Datum oder eine Uhrzeit bei der Zuweisung in das Zeichen »#« eingeschlossen werden muss, um als Datum oder Uhrzeit erkannt zu werden:

```
Falsch: x = 1.Jan 1999

Richtig: x = #1.Jan 1999#
```

Zeichenketten

Zeichenketten müssen wie bereits erwähnt in Anführungszeichen eingeschlossen werden:

```
Falsch: x$ = Hallo

Richtig: x$ = "Hallo"
```

Sie können mehrere Zeichenketten zu einer einzigen verknüpfen. Entweder mit dem Zeichen »&« oder mit dem Operator »+«.

Die Ausdrücke

```
x$ = "Gerd" & " " & "Maier"
```

und

```
x$ = "Gerd" + " " + "Maier"
```

sind funktional äquivalent. Beide weisen der Stringvariablen (Zeichenkettenvariable) *x$* die Zeichenkette »Gerd Maier« zu.

Analog zum Tabellenentwurf sollten Sie immer jenen Typ wählen, der für die betreffende Anwendung gerade noch ausreicht. Verwenden Sie beispielsweise keine *Long*-Variable, um ganze Zahlen zwischen 100 und 200 zu speichern, sondern benutzen Sie dazu den Typ *Integer*.

Die einfachste Möglichkeit zur Festlegung des gewünschten Variablentyps besteht darin, an den Variablennamen das betreffende Typkennzeichen anzuhängen und die Variable einfach zu verwenden, wo sie gebraucht wird. Im gleichen Moment, in dem der betreffende Variablenname zum erstenmal benutzt wird, reserviert Access im Rechnerspeicher den dafür benötigten Platz.

Im Gegensatz zu vielen anderen Programmiersprachen können Sie nicht mit

```
x% = 10
x$ = "Test"
```

der Integervariablen *x%* die Zahl 10 und der Stringvariablen *x$* die Zeichenkette »Test« zuweisen. Access teilt Ihnen mit einer Fehlermeldung mit, dass die Variable *x* bereits benutzt wird, und verweigert die Ausführung der Zuweisung *x$ = "Test"*!

3.3.2 Variablen deklarieren

Sie können Variablen in einer Prozedur ohne besondere Umstände einfach benutzen und den Typ der Variablen bei Bedarf mit dem zugehörigen Typkennzeichen festlegen. Dazu muss jedoch der Deklarationszwang deaktiviert werden (siehe Kapitel 3.3, »Variablen und Konstanten«).

Dim Meist ist es jedoch äußerst sinnvoll, eine Variable noch vor der ersten Benutzung mit einer der Anweisungen *Dim*, *Public*, *Private* oder *Static* zu deklarieren.

Dim ist die gebräuchlichste Deklarationsanweisung (die restlichen Anweisungen erläutere ich im Kapitel 3.6.2, »Geltungsbereiche von Variablen und Konstanten«):

```
Dim Variablenname[Typkennzeichen]
```

Die nach diesem Schema gebildete Deklaration wird als erste Zeile am Anfang einer Prozedur eingefügt:

Listing 3.3: Variablen deklarieren

```
Function Test ()
  Dim x%

  x% = 10
  Debug.Print x%
End Function
```

Die *Dim*-Anweisung reserviert den für die Variable benötigten Platz im Rechnerspeicher, obwohl er erst in der nächsten Programmzeile benötigt wird.

Ohne das optionale Typkennzeichen erhält die deklarierte Variable den Standardtyp *Variant*. *Dim x* deklariert daher eine *Variant*-Variable namens *x*.

Dim As Wirklich sinnvoll ist die Deklaration mit *Dim* jedoch erst mit folgender Syntax:

```
Dim Variablenname As Typ
```

Damit ist es möglich, den Datentyp einer Variablen festzulegen und dennoch nicht bei jeder späteren Benutzung der Variablen jedes Mal das Typkennzeichen angeben zu müssen:

Listing 3.4: Deklaration mit As

```
Function Test ()
  Dim x As Integer

  x = 10
  Debug.Print x
End Function
```

Nach der Deklarierung mit *Dim x As Integer* steht für Access ein für allemal fest, dass die in dieser Prozedur verwendete Variable *x* eine Integervariable ist, und Sie können bei folgenden Bezügen auf diese Variable auf das Typkennzeichen verzichten.

Im Gegensatz zu numerischen Variablen besitzen Stringvariablen normalerweise keine feste Länge. Nach der Deklaration einer Stringvariablen *s* mit

```
Dim s As String
```

speichert eine Zuweisung wie *s = "Test"* darin vier Zeichen und eine Anweisung wie *s = "Gerd Maier"* zehn Zeichen. Die Größe der Stringvariablen und der von ihr belegte Speicherplatz wird von Access bei jeder neuen Zuweisung entsprechend angepasst – was allerdings viel Zeit kostet!

Strings fester Länge

Wissen Sie genau, wie viele Zeichen eine bestimmte Stringvariable maximal aufnehmen wird, können Sie stattdessen Stringvariablen fester Länge verwenden, die mit einer Anweisung der Art

```
Dim Variablenname As String * Länge
```

deklariert werden.

Zum Beispiel deklariert

```
Dim s As String * 10
```

eine Stringvariable *s* mit einer Länge von zehn Zeichen. Weisen Sie ihr mit *s = "Otto"* nur vier Zeichen zu, füllt Access die restlichen sechs Zeichen mit Leerzeichen auf, so dass darin tatsächlich die Zeichenkette »Otto « gespeichert ist. Weisen Sie ihr mit *s = "Otto Maierbach"* mehr als zehn Zeichen zu, werden die überzähligen Zeichen abgeschnitten und darin nur »Otto Maier« gespeichert.

Stringvariablen fester Länge eignen sich vor allem zur Aufnahme der Inhalte von Tabellenfeldern vom Typ *Text*, die ja ebenfalls nur die beim Tabellenentwurf festgelegte Zeichenanzahl speichern können.

Sie können mit *Dim* auch mehrere Variablen auf einmal deklarieren:

```
Dim Variablenname As Datentypname, Variablenname As Datentypname,
Variablenname As Datentypname, ...
```

Zum Beispiel deklariert die Anweisung

```
Dim x As Double, y As String, z As String * 10
```

drei Variablen: *x* ist eine Gleitkommavariable doppelter Genauigkeit, *y* ist eine Stringvariable variabler Länge und *z* ist eine Stringvariable mit einer festen Länge von zehn Zeichen.

Deklarationszwang

Die explizite Deklaration von Variablen mit *Dim* scheint zunächst umständlich zu sein, besitzt in der Praxis jedoch nur Vorteile. Speziell größere Programme werden dadurch übersichtlicher und besser strukturiert und dadurch werden wiederum viele unnötige Fehler von vornherein vermieden! Ich empfehle Ih-

nen dringend, dafür zu sorgen, dass Access Sie dazu *zwingt*, Variablen zu deklarieren!

Dazu fügen Sie in den Deklarationsabschnitt eines Moduls (im Prozeduren-Listenfeld den Eintrag »(Deklarationen)« selektieren) die Anweisung *Option Explicit* ein. Wird irgendwo in diesem Modul eine nicht deklarierte Variable verwendet, erhalten Sie später bei der Ausführung der betreffenden Prozedur eine entsprechende Fehlermeldung.

Wie bereits erläutert fügt VBA diese Anweisung sogar automatisch in jedes neue Modul ein, wenn das Kontrollkästchen »Variablen-Deklaration erforderlich« im Befehl EXTRAS | OPTIONEN... aktiviert ist.

3.3.3 Konstanten deklarieren

»Literale Konstanten« kennen Sie. Jede Zahl oder Zeichenkette, die sich in Ihrem Programmtext befindet, ist eine literale Konstante.

Die Anweisung

```
Test.Drucke "Hallo"; 3 * 7
```

enthält drei literale Konstanten: die Stringkonstante »Hallo« und die beiden Zahlenkonstanten 3 und 7.

Wesentlich interessanter sind »symbolische Konstanten«, die mit der Anweisung *Const* deklariert werden:

```
Const Konstantenname = Ausdruck
```

Das Argument »Konstantenname« unterliegt den gleichen Regeln wie ein Variablenname. Ohne Typkennzeichen wird der passendste Datentyp verwendet. »Ausdruck« ist ein beliebiger String- oder numerischer Ausdruck, der den Wert der Konstanten angibt, zum Beispiel *"Hallo"* oder *3 * 7*.

Ein Beispiel, in dem zwei Konstanten deklariert werden:

```
Const a = "Hallo"
Const x = 10 * 5
```

Wie bei *Dim* können Sie den Zusatz *As Typ* verwenden, um den Typ der Konstanten festzulegen:

```
Const Konstantenname As Typ = Ausdruck
```

Beispielsweise deklariert

```
Const a As String = "Hallo"
```

eine Stringkonstante.

Wie der Name bereits sagt, ist der Wert einer Konstanten konstant, das heißt unveränderlich. Im Gegensatz zu einer Variablen kann er nach der Festlegung nicht mehr verändert werden. Es ist daher nicht möglich, einer Konstanten einmal mit der Anweisung *Const x = 5* den Wert 5 zuzuweisen und mit einer folgenden Anweisung *Const x = 10* einen anderen Wert zuweisen zu wollen!

Diese Eigenschaft von Konstanten ist immer dann nützlich, wenn Sie mit Werten hantieren (Zahlen oder Zeichenketten), die unveränderlich sind. Dann definieren Sie eine Konstante mit dem entsprechenden Wert und verwenden in Ihrem Programm statt der Zahl/Zeichenkette selbst den bequemer zu benutzenden symbolischen Namen, den Sie der Konstanten gaben.

Ein Beispiel für einen solchen normalerweise unveränderlichen Wert ist der Mehrwertsteuersatz. Es bietet sich an, für entsprechende Berechnungen am Prozeduranfang eine Konstante zu verwenden, die folgendermaßen deklariert wird:

```
Const MWSt = 16
```

In Ihrem Programm können Sie nun in Berechnungen statt der Zahl 16 den Konstantennamen *MWSt* verwenden. Ändert sich irgendwann der Mehrwertsteuersatz, müssen Sie nur diese Deklaration ändern.

Sie können Konstanten in Abhängigkeit von anderen Konstanten deklarieren. Beispielsweise mit *Const A = 20* eine Konstante *A* und danach mit *Const B = A / 2* eine Konstante *B* deklarieren, die den zuvor *A* zugewiesenen Wert benutzt. Ändern Sie irgendwann die Zuweisung an die Konstante *A*, ändert sich automatisch der *B* zugewiesene Wert entsprechend.

3.3.4 Vordefinierte Konstanten

Access stellt Ihnen eine Unmenge vordefinierter und global gültiger Konstanten zur Verfügung. Damit sind Konstanten gemeint, die einen bereits vorgegebenen Wert besitzen und die Sie in beliebigen Prozeduren Ihrer Programme anstelle dieses Werts einsetzen können.

Die Namen dieser vordefinierten Konstanten beginnen übrigens immer mit zwei Buchstaben, die die Objektbibliothek bezeichnen, die sie zur Verfügung stellt. Access-Konstanten beginnen beispielsweise mit »ac…«, »DAO-Konstanten« (DAO = Datenzugriffsobjekte) mit »db…« und VBA-Konstanten mit »vb…«.

Die Frage ist, was Sie mit diesen Konstanten anfangen können. Nehmen wir an, Sie benutzen häufig die Funktion *MsgBox*, der wie erläutert unterschiedliche Zahlenwerte übergeben werden, um ihr Verhalten näher festzulegen.

Diese Funktion wird von der VBA-Objektbibliothek zur Verfügung gestellt, die zusätzlich mehrere Konstanten definiert, die den Umgang mit diesen Zahlenwerten erleichtert und Ihre Programme wesentlich lesbarer macht.

Beispielsweise besitzt die VBA-Konstante *vbOKOnly* den Wert 0, die Konstante *vbOKCancel* den Wert 1 und *vbQuestion* den Wert 32. Wollen Sie die Funktion *MsgBox* aufrufen und soll darin nur die »OK«-Schaltfläche erscheinen, können Sie daher statt

```
MsgBox("Dateiname:", 0)
```

alternativ folgenden Ausdruck verwenden:

```
MsgBox("Dateiname:", vbOKOnly)
```

Soll die »OK«- und die »Abbrechen«-Schaltfläche und zusätzlich ein Fragezeichensymbol erscheinen, können Sie statt

```
MsgBox("Dateiname:", 1 + 32)
```

beziehungsweise

```
MsgBox("Dateiname:", 33)
```

entsprechend folgende aussagekräftigere Alternative verwenden:

```
MsgBox("Dateiname:", vbOKCancel + vbQuestion)
```

Mithilfe des Objektkatalogs (siehe Kapitel 10.1.1, »Nutzung des Objektkatalogs«) können Sie sich eine Übersicht über die vordefinierten *MsgBox*-Konstanten verschaffen. Dazu wählen Sie die interessierende Objektbibliothek aus, im Beispiel »VBA«, und selektieren im Listenfeld darunter »vbMsgBoxResult« oder »vbMsgBoxStyle«. Im rechten Listenfeld werden daraufhin die von der betreffenden Bibliothek zur Verfügung gestellten Konstanten aufgelistet (Bild 3.7).

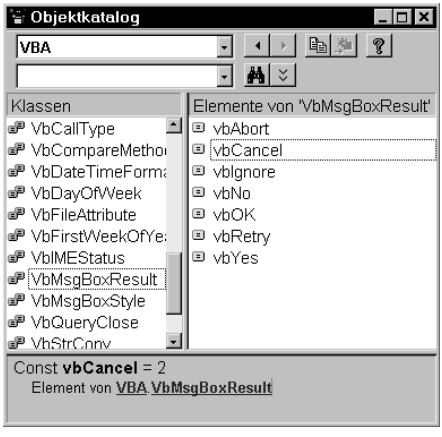


Bild 3.7: MsgBox-Konstanten

3.4 Test 1

1. Ein Programm soll im Direktfenster die Zahlen 100, 200 und 300 ausgeben, mit ein wenig Abstand zwischen den Zahlen. Welche Möglichkeiten fallen Ihnen ein?
2. Welche Anweisung wird benötigt, um in einem Dialogfeld dem Benutzer mitzuteilen »Okay, hat geklappt!« und eine »OK«-Schaltfläche im Dialogfeld anzuzeigen?
3. Und wie schreiben Sie das Ganze lesefreundlicher?
4. Mit welcher Anweisung fordern Sie den Benutzer in einem Dialogfeld mit dem Titel »Eingabe« dazu auf, einen Bruttobetrag einzugeben, und geben seine Eingabe anschließend im Direktfenster aus?
5. Welchen Datentyp hat die in der letzten Aufgabe von VBA übergebene Benutzereingabe und wie speichern Sie das Doppelte des eingegebenen Bruttobetrags in einer numerischen Variablen?
6. Welchen Wert speichern die beiden aufeinander folgenden Programmzeilen *x = 10* und *x = 20* in *x*?
7. Welchen Wert speichern die beiden aufeinander folgenden Programmzeilen *y = 10* und *x = 2 * y* in *x*?
8. Wie lang ist die mit *Dim s As String * 10* deklarierte Stringvariable *s* nach der Zuweisung *s = "Porsche"*?
9. Und wie lang ist *s*, wenn die Deklaration einfach *Dim s* lautete?

3.5 Prozeduren

Ich zeige gleich, wie Sie eigendefinierte Prozeduren unterschiedlichsten Typs erstellen. Zuvor möchte ich jedoch auf die Unmenge an bereits vorgegebenen Prozeduren eingehen.

3.5.1 Vorgegebene Funktionsprozeduren

VBA stellt Ihnen eine ganze Menge eingebauter Prozeduren in Form von Funktionsprozeduren zur Verfügung, und zwar numerische Funktionen und Stringfunktionen:

- Numerische Funktionen übergeben als Funktionswert eine Zahl.
- Stringfunktionen übergeben eine Zeichenkette.

Sqr Ein praktisches Beispiel für eine numerische Funktion ist *Sqr*, die die Quadratwurzel einer Zahl ermittelt:

```
v = Sqr(x)
```

Sqr wird als Argument eine Zahl »x« übergeben. *Sqr* ermittelt die Quadratwurzel dieser Zahl und übergibt sie als Funktionswert.

Eine ebenfalls sehr häufig benutzte Funktion ist *Len*:

```
v = Len(x$)
```

Len *Len* ermittelt die Länge einer Zeichenkette, eines Strings, und übergibt als Funktionswert die Anzahl der im betreffenden String enthaltenen Zeichen. Zum Beispiel ermittelt *Len("Test")* die Länge der Zeichenkette »Test« und übergibt als Funktionswert die Zahl 4.

Int Der Funktion *Int* wird als Argument ein numerischer Wert »x« übergeben:

```
v = Int(x)
```

Als Funktionswert »v« liefert *Int* wieder einen numerischen Wert (fehlendes Dollarzeichen), den »Integer-« oder »ganzzahligen« Anteil der übergebenen Zahl: die größte ganze Zahl, die kleiner oder gleich »x« ist.

Nehmen wir als Beispiel die Zahl 3.45. Die größte ganze Zahl, die kleiner oder gleich 3.45 ist, ist 3, denn 4 ist bereits größer. Oder nehmen wir 167.12345. Der ganzzahlige Anteil dieser Zahl ist 167.

Bei negativen Zahlen wird ebenfalls der Nachkommaanteil entfernt. Aus –23.9 macht *Int* daher –24.

Stringfunktionen Mithilfe von Stringfunktionen können Sie prüfen, ob in einer Zeichenkette eine bestimmte andere Zeichenkette enthalten ist, Teile aus einer Zeichenkette heraustrennen und so weiter, kurz: Sie können Zeichenketten damit auf vielfältigste Art und Weise manipulieren.

Die drei am häufigsten verwendeten Stringfunktionen sind *Mid*, *Left* und *Right*:

```
v = Left(x$, n)
v = Right(x$, n)
v = Mid(x$, n [,m])
```

Einige Funktionen, die Zeichenfolgen liefern, gibt es in zwei Versionen: einmal als Funktion des Typs *Variant* und ein weiteres Mal als Funktion vom Typ *String*, die explizit eine Zeichenkette übergibt. *Left*, *Right* und *Mid* sind Beispiele für derartige Funktionen. Hängen Sie an den Funktionsnamen ein Dollarzeichen »$« an (*Left$*, *Right$*, *Mid$*), wird der Datentyp *String* statt *Variant* zurückgegeben.

Left

Die *Left*-Funktion übergibt die ersten »n« Zeichen der Zeichenkette »x$«. Wie bei jeder Funktion kann das numerische Argument »n« eine Konstante (3, 7, 23), eine numerische Variable (*x*, *zahl%*) oder ein komplexer Ausdruck sein (*3 + x*, *7 * zahl%*).

Der Stringausdruck »x$« darf ebenfalls eine Stringkonstante ("Maier", "Müller"), eine Stringvariable (*x$*, *name$*) oder ein komplexer Ausdruck sein (*"Müller" + x$* oder *a$ + b$ + c$*).

Zum Beispiel gibt der Aufruf

```
Debug.Print Left("Hallo", 1)
```

das erste Zeichen der Zeichenkette »Hallo« im Direktfenster aus, also »H«.

Und der Aufruf

```
Debug.Print Left("Hallo", 2)
```

gibt entsprechend die ersten zwei Zeichen von »Hallo« aus, also »Ha«. Interessant wird es, wenn das Argument »n« größer ist als die Länge der Zeichenkette. Die Anweisung

```
Debug.Print Left("Hallo", 6)
```

soll VBA veranlassen, die ersten sechs Zeichen einer Zeichenkette auszugeben, die aus nur fünf Zeichen besteht. Das Resultat: Es werden einfach nur die fünf möglichen Zeichen ausgegeben, also »Hallo«.

Right

Right wird mit den gleichen Argumenten aufgerufen wie *Left*, übergibt jedoch nicht den linken, sondern den rechten Teil der Zeichenkette »x$« in der Länge »n«. Die Anweisung

```
Debug.Print Right("Hallo", 3)
```

gibt die letzten drei Zeichen von »Hallo« aus, »llo«.

Mid

Mid übergibt die ersten »m« Zeichen des Strings »x$« ab Position »n«. Zur Position ist zu sagen, dass das erste Zeichen in einem String die Nummer 1 besitzt, das Zeichen rechts davon die Nummer 2 und so weiter.

Die Anweisung

```
Debug.Print Mid("Hallo", 3, 2)
```

übergibt die beiden Zeichen (*m = 2*), die sich ab dem dritten Zeichen (*n = 3*) im String »Hallo« befinden, also »ll«.

Angenommen, Sie speichern in der Variablen *a$* die Zeichenkette »Mein Personal Computer ist der beste«. Welche Anweisung übergibt »Personal Computer«? Eine *Mid*-Anweisung, in der als Argument »x$« die Variable *a$*, als Argument »n« der Wert 6 (die betreffende Teilzeichenkette beginnt ab dem

sechsten Zeichen von *a$*) und als Argument »m« der Wert 17 übergeben wird (»Personal Computer« besteht aus 17 Zeichen):

```
a$ = "Mein Personal Computer ist der beste"
Debug.Print Mid(a$, 6, 17)
```

Wenn das Argument »m« – die optionale Längenangabe – entfällt, übergibt *Mid* einfach den gesamten Rest der Zeichenkette ab der angegebenen Position »n«. Daher übergibt *Mid("Hallo", 3)* die Zeichenkette »llo«, alle Zeichen ab Position 3 bis zum Ende des Strings.

InStr *InStr* ist eine numerische Funktion (fehlendes Dollarzeichen), die angibt, ob und – wenn ja – wo in einer Zeichenkette *x$* eine zweite Zeichenkette *y$* enthalten ist.

```
v = InStr([n,] x$, y$)
          ¦   ¦   ¦
   Suchstart  ¦   String, der gesucht wird
              ¦
        String, der
        durchsucht wird
```

Ohne den optionalen Parameter »n« ist die Anwendung der *InStr*-Funktion einfach: *Instr("Hallo", "al")* übergibt den numerischen Wert 2, da die Zeichenkette »al« ab Position Nummer 2 im String »Hallo« enthalten ist. Und *InStr("Dies ist ein Test", "kein")* übergibt 0, da die Zeichenkette »kein« in »Dies ist ein Test« nicht enthalten ist (mit »ein« statt »kein« als gesuchter Zeichenkette übergibt *Instr* als Resultat 10).

Mit dem Parameter »n« können Sie festlegen, wo die Suche beginnt. Ohne diesen Parameter beginnt sie am Anfang des zu durchsuchenden Strings, gesucht wird also ab dem ersten Zeichen.

»n« ist eine beliebige Zahl zwischen 1 und der Länge des durchsuchten Strings. Zum Beispiel durchsucht *InStr(2, "Ein Test", "Ein")* die Zeichenkette »Ein Test« ab dem zweiten Zeichen nach »Ein« und übergibt daher 0.

Str Die Funktion *Str* übergibt einen String, der die einzelnen Ziffern der Zahl »x« enthält.

Zum Beispiel weist die Anweisung

```
a$ = Str(23)
```

der Stringvariablen *a$* die Zeichenkette » 23« zu. Beachten Sie bitte das erste Zeichen dieses Strings, das Leerzeichen vor der eigentlichen Zahl!

3.5.2 Prozeduren selbst erstellen

Wie Sie wissen, erstellt EINFÜGEN | PROZEDUR... bzw. das zugehörige Symbol eine neue Prozedur (Bild 3.8).

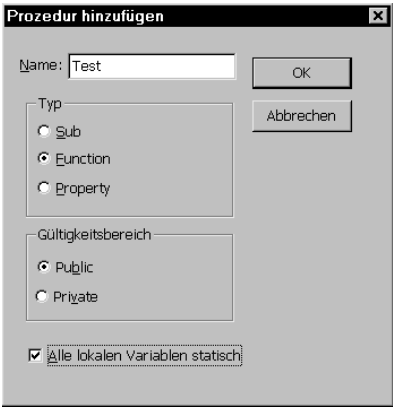


Bild 3.8: Prozedur erstellen

Typ und Geltungsbereich

Wie der eingefügte Prozedurrumpf aussieht, hängt von den ausgewählten Optionen ab.

Die unter »Typ« gewählte Option bestimmt die Art der Prozedur: Je nach Option fügt Access vor einem Prozedurnamen wie *Test* eines der drei Schlüsselwörter *Sub* (Sub-Prozedur), *Function* (Funktionsprozedur) oder *Property* (Eigenschaftsprozedur) ein.

- Gültigkeitsbereich: »Public« fügt zusätzlich das Schlüsselwort *Public* ein, »Private« das Schlüsselwort *Private*.
- Aktivieren Sie »Alle lokalen Variablen statisch«, wird darüber hinaus noch das Schlüsselwort *Static* eingefügt.

Prozedurdeklarationen können also recht komplex sein. Geben Sie Ihrer Prozedur den Namen *Test*, wählen Sie »Function« und »Private« und aktivieren Sie zusätzlich das Kontrollkästchen »Alle lokalen Variablen statisch«, lautet der eingefügte Prozedurrumpf:

```
Private Static Function Test()
```

Private Static Function Test() deklariert eine private Funktionsprozedur namens *Test*, deren Variablen statisch sind. Was das bedeutet, müssen Sie jetzt noch nicht wissen. Die drei Schlüsselwörter *Public*, *Private* und *Static* sind alle optional, das heißt, sie können weggelassen werden.

Der Beweis dafür ist einfach zu führen: Access fügt zwar automatisch zumindest *Public* oder *Private* ein; löschen Sie das betreffende Schlüsselwort, funktioniert die betreffende Prozedur jedoch genauso wie zuvor!

Ich will damit nicht sagen, dass diese Schlüsselwörter bedeutungslos seien. Da *Public*, *Private* und *Static* aber zumindest nicht »lebenswichtig« für eine Prozedur sind, bespreche ich sie erst später, im Kapitel 3.6.1, »Geltungsbereiche von Prozeduren«. Auch auf *Property* werde ich erst später eingehen.

Das reduziert die Syntax einer Prozedur zunächst auf die beiden folgenden harmloseren Varianten:

Listing 3.5: Sub-Prozeduren

```
'Syntax einer Sub-Prozedur
'-------------------------

Sub Prozedurname [(Argumentliste)]
  [Anweisung 1]
  [Anweisung 2]
  ...
  ...
  [Anweisung N]
End Sub
```

Listing 3.6: Funktionsprozeduren

```
'Syntax einer Funktionsprozedur
'------------------------------

Function Prozedurname [(Argumentliste)]
  [Anweisung 1]
  [Anweisung 2]
  ...
  ...
  [Prozedurname = Ausdruck]
  ...
  ...
  [Anweisung N]
End Function
```

Beide bestehen aus einem Prozedurrumpf, zwischen dem sich beliebig viele auszuführende Anweisungen befinden können.

Der entscheidende Unterschied ist der zusätzliche Ausdruck *Prozedurname = Ausdruck*, der sich in einer Funktionsprozedur an einer beliebigen Stelle befinden darf.

Funktionsprozeduren

Er definiert den Funktionswert, den die Funktionsprozedur übergibt. Heißt die Funktion beispielsweise *Test*, weist ihr der Ausdruck *Test = 20* den Funktionswert 20 zu.

Funktionsprozeduren übergeben prinzipiell einen Funktionswert vom allgemeinen Datentyp *Variant*. Sie können den Funktionstyp, also den Datentyp des Funktionswerts, jedoch mit einem zusätzlichen *As Datentyp*-Ausdruck individuell festlegen:

```
Function Prozedurname [(Argumentliste)] As Datentyp
```

Beispielsweise übergibt die Funktion


```
Function Test() As Double
```

einen *Double*-Wert als Funktionswert.

Alternativ können Sie hinter dem Funktionsnamen das betreffende Typkennzeichen angeben:

```
Function Test#()
```

Sub-Prozeduren

Sub-Prozeduren übergeben zwar keinen Funktionswert, besitzen aber dennoch ihren Sinn. Stellen Sie sich ein größeres Programm mit Unmengen an Funktionsprozeduren vor, die immer wieder irgendwelche Meldungen auf dem Bildschirm ausgeben müssen, entweder im Direktfenster oder in einem kleinen Dialogfeld (*MsgBox*).

Natürlich können Sie in jeder dieser Funktionsprozeduren immer wieder die gleichen dazu benötigten Anweisungen eintippen. Allerdings ist es einfacher, für diese immer gleiche Aufgabe eine Sub-Prozedur zu erstellen, die die benötigten Ausgabeanweisungen enthält. Immer dann, wenn eine der Funktionsprozeduren wieder 'mal die betreffenden Texte ausgeben muss, ruft sie dazu einfach mit der Anweisung *Call* die dafür zuständige Sub-Prozedur auf:

```
Call Prozedurname [(Argumentliste)]
```

Beispielsweise ruft


```
Call Aufgerufen
```

Prozeduraufrufe

die Sub-Prozedur *Aufgerufen* auf. Enthält eine Funktionsprozedur diese Anweisung, verzweigt der Programmfluss zu jener Prozedur, das heißt, nun werden zunächst die in *Aufgerufen* enthaltenen Anweisungen ausgeführt.

Ist das Ende der Prozedur erreicht, wird zur aufrufenden Prozedur zurückgekehrt und die Anweisung ausgeführt, die *Call* folgt (Bild 3.9).

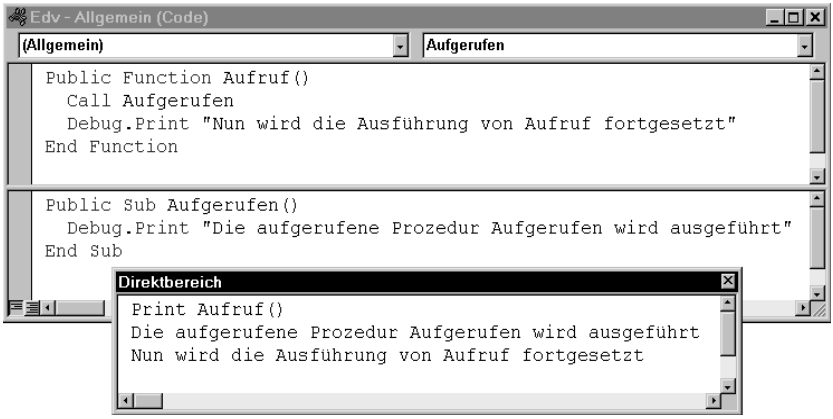


Bild 3.9: Prozeduren »Aufruf« und »Aufgerufen«

Die Funktion *Aufruf* wird im Direktfenster mit *Print Aufruf()* ausgeführt. Die erste darin enthaltene Anweisung *Call Aufgerufen* ruft die Sub-Prozedur *Aufgerufen* auf: Die Ausführung von *Aufruf* wird vorübergehend unterbrochen und nun wird die Sub-Prozedur *Aufgerufen* ausgeführt, die die Zeichenkette »Die aufgerufene Prozedur Aufgerufen wird ausgeführt« ausgibt.

End Sub beendet die Ausführung der Sub-Prozedur. Access kehrt zur »aufrufenden« Funktionsprozedur *Aufruf* zurück und setzt deren Ausführung mit der folgenden Anweisung fort, mit

```
Debug.Print "Nun wird die Ausführung von Aufruf fortgesetzt"
```

Allgemein: Die Anweisung *Call* unterbricht die Ausführung einer Prozedur und führt die angegebene Prozedur aus. Ist deren Ende erreicht, wird zur aufrufenden Prozedur zurückgekehrt und dort die nächste Anweisung ausgeführt.

Call ist übrigens optional, das heißt, eine Prozedur namens *Aufgerufen* kann statt mit *Call Aufgerufen* ebenso gut einfach mit *Aufgerufen* ausgeführt werden. Die Argumentliste darf dann jedoch nicht von Klammern umgeben sein!

3.5.3 Lokale Variablen in Prozeduren

Nun ein praktisches Beispiel: Eine Sub-Prozedur soll, basierend auf einer Zahl *x*, die darin enthaltene Mehrwertsteuer und den Nettobetrag berechnen. Dazu muss die aufrufende Prozedur der aufgerufenen Sub-Prozedur mitteilen, welche Zahl für diese Berechnung verwendet werden soll.

Die einfachste Lösung scheint zunächst darin zu bestehen, die Zahl in einer Variablen *x* zu speichern, die die aufgerufene Sub-Prozedur anschließend weiterverwendet, etwa so (Bild 3.10).

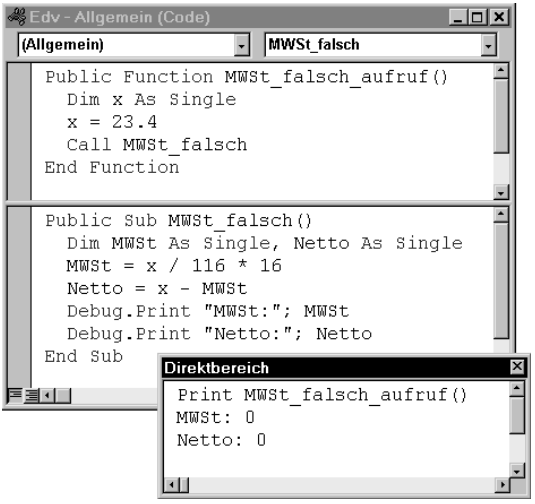


Bild 3.10: Prozeduren »MWSt_falsch_aufruf« und »MWSt_falsch«

MWSt_falsch_aufruf weist der *Single*-Variablen *x* den Bruttobetrag 23.4 zu und ruft *MWSt_falsch* auf. *MWSt_falsch* ermittelt mit *MWSt = x / 116 * 16* die in *x* enthaltene Mehrwertsteuer und weist sie der *Single*-Variablen *MWSt* zu. Danach ermittelt *Netto = x – MWSt* den Nettobetrag und speichert ihn in der *Single*-Variablen *Netto*.

Debug.Print "MWST:"; mwst gibt die Zeichenkette »MWSt« und dahinter den Inhalt der Variablen *MWSt* aus, *Debug.Print "Netto:"; Netto* entsprechend »Netto« und den Inhalt der Variablen *Netto*.

Offenbar enthalten jedoch beide Variablen den Wert 0. Die Ursache dafür klärt ein kleiner Versuch (Bild 3.11).

Der in *MWSt_versuch_aufruf* deklarierten Variablen *x* wurde der Wert 23.4 zugewiesen, den *MWSt_versuch_aufruf* auch korrekt ausgibt.

Anschließend wird *MWSt_versuch* aufgerufen. Dort gibt *Debug.Print x* statt 23,4 jedoch überhaupt nichts aus. Offenbar ist die Variable *x* für die Prozedur *MWSt_versuch* nicht existent!

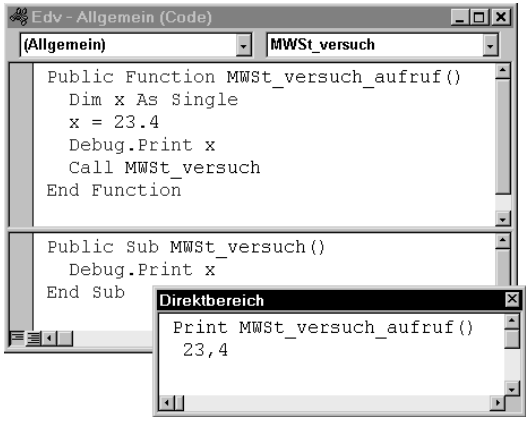


Bild 3.11: Prozeduren »MWSt_versuch_aufruf« und »MWSt_versuch«

Damit kennen wir nun auch die Ursache der fehlgeschlagenen Mehrwertsteuerberechnung:

Eine Variable ist immer lokal zur Prozedur, in der sie deklariert wurde und nur dieser Prozedur und keiner anderen bekannt! Für andere Prozeduren ist die betreffende Variable nicht existent. Tatsächlich existiert sie nur während der Ausführung der betreffenden Prozedur und wird danach wieder gelöscht.

Daher können Sie ohne die geringsten Probleme in beliebig vielen Prozeduren gleichnamige Variablen deklarieren und verwenden, ohne dass sich diese Variablen gegenseitig beeinflussen. Jede der Prozeduren sieht nur ihre eigene Variable *x* und weiß nichts von der Existenz einer gleichnamigen Variablen in einer anderen Prozedur.

Diese Eigenschaft von Variablen ist vor allem in großen und komplexen Programmen ein enormer Vorteil, da dadurch versehentliche Veränderungen von Variablen durch die Verwendung gleichnamiger Variablen in anderen Prozeduren von vornherein ausgeschlossen sind.

3.5.4 Argumente an Prozeduren übergeben

Sie ist jedoch von Nachteil, wenn wie bei der geplanten Mehrwertsteuerberechnung der Inhalt einer Variablen ganz bewusst an eine andere Prozedur übergeben werden soll. Glücklicherweise stellt Access für die Variablenübergabe einen eigenen Mechanismus zur Verfügung.

Argumentliste

Schauen Sie sich noch einmal die vollständige Syntax der *Function* und der *Sub*-Anweisung an:

```
Function Prozedurname [(Argumentliste)]
```

```
Sub Prozedurname [(Argumentliste)]
```

»Argumentliste« ist eine Anzahl von Variablen, jeweils durch ein Komma getrennt. In diesen Variablen speichert die aufgerufene Prozedur die von der aufrufenden Prozedur übergebenen Werte. Aus der Liste muss eindeutig hervorgehen, welchen Datentyp die übergebene Information besitzt.

Zum Beispiel wird der Prozedur

```
Sub Test (x!)
```

ein *Single*-Wert übergeben, den diese Prozedur in der Variablen *x!* speichert. *x!* ist die lokale Prozedurvariable, die die übergebene Information aufnimmt.

As

Alternativ dazu können Sie den Typ der übergebenen Variablen ähnlich wie in der *Dim*-Anweisung mit dem Zusatz *As Typ* bekannt geben:

```
Sub Test (x As Single)
```

Diese Methode werde ich bevorzugen, da sie klarer ist und eher allgemeinen Standards entspricht, die beispielsweise auch für Pascal oder C gelten.

Damit ist unser Mehrwertsteuerproblem gelöst (Bild 3.12).

Der Aufruf *Call MWSt_okay(23.4)* übergibt der Prozedur *MWSt_okay* den Wert 23.4, den diese in der lokalen Variablen *brutto* übernimmt, die anschließend zur Ermittlung der darin enthaltenen Mehrwertsteuer und des zugehörigen Nettobetrags benutzt wird.

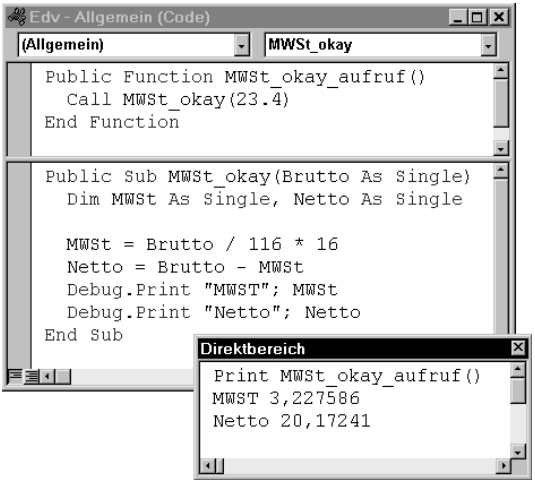


Bild 3.12: Prozeduren »MWSt_okay_aufruf« und »MWSt_okay«

Sie können einer Prozedur beliebig viele Argumente übergeben. Die folgende Prozedur erwartet beispielsweise die Übergabe zweier Zahlen:

```
Sub Testproc(Einkaufspreis As Single, Verkaufspreis As Single)
  Debug.Print "Gewinn:", Verkaufspreis – Einkaufspreis
End Sub
```

Um sie aufzurufen, verwenden Sie einen Ausdruck wie

```
Call Testproc(200, 300)
```

Nach dem Aufruf befindet sich in der Variablen *Einkaufspreis* der Prozedur der Wert 200 und in der Variablen *Verkaufspreis* der Wert 300. Die Prozedur verwendet diese Variablen, um den Gewinn zu ermitteln und im Direktfenster auszugeben.

Konstanten übergeben

Die Übergabe von Konstanten ist der einfachste Fall. Sie können einer Prozedur beliebige Ausdrücke übergeben, Konstanten, Variablen oder Mischungen von beidem.

Zum Beispiel übergibt der Aufruf

```
Call MWSt_okay(20 + 3.4)
```

der Prozedur *MWSt_okay* ebenfalls die Zahl 23.4, nur etwas umständlicher formuliert.

Auch

```
x = 23.4
Call MWSt_okay(x)
```

übergibt diese Werte, ebenso wie

```
x = 20
Call MWSt_okay(x + 3.4)
```

Zeichenketten übergeben

Natürlich können Sie auch Stringargumente übergeben.

Der Aufruf

```
Call Stringtest ("Hallo")
```

übergibt einer Prozedur *Stringtest* die Zeichenkette »Hallo«.

Ebenso wie der Aufruf

```
a$ = "Hallo"
Call Stringtest (a$)
```

Entsprechend muss in der Deklaration von *Stringtest* die Übergabe einer Stringvariablen vorgesehen werden, in der diese Zeichenkette bei der Übergabe gespeichert wird. Entweder so:

```
Sub Stringtest (s$)
```

Oder so:

```
Sub Stringtest (s As String)
```

Übergabeausdrücke dürfen beliebig komplex sein:


```
Call Stringtest(a$ + "x" + b$ + "y")
```

Hier wird der Prozedur *Stringtest* eine Zeichenkette übergeben, die durch die Verkettung mehrerer Stringvariablen und Stringkonstanten entsteht.

3.5.5 Benannte und optionale Argumente

Wenn Sie wollen, müssen Sie sich beim Aufruf einer Prozedur nicht unbedingt an die in der Deklaration festgelegte Reihenfolge halten. Angenommen, eine Prozedur erwartet zwei Zahlen:

```
Sub Testproc(Einkaufspreis As Single, Verkaufspreis As Single)
```

Normalerweise rufen Sie diese Prozedur etwa so auf:

```
Call Testproc(200, 300)
```

Dann ist 200 der übergebene Einkaufspreis und 300 der übergebene Verkaufspreis.

Benannte Argumente

Wenn Sie wollen, können Sie die Reihenfolge der Argumente bei der Übergabe vertauschen. Damit Access weiß, welche der beiden Zahlen in der Prozedurvariablen *Einkaufspreis* gespeichert werden soll und welche in der Prozedurvariablen *Verkaufspreis*, müssen Sie die Argumente jedoch entsprechend benennen und zum Aufruf folgende Syntax *ohne Call* verwenden:

```
Prozedurname Variablenname1:=Wert, Variablenname2=Wert, ...
```

Angewandt auf das Beispiel:


```
Testproc Einkaufspreis:=200, Verkaufspreis:=300
```

Optionale Argumente

Sollen Argumente optional sein, beim Aufruf also nicht unbedingt angegeben werden müssen, stellen Sie ihnen das Schlüsselwort *Optional* voran. Beachten Sie dabei, dass optionale Argumente vom Typ *Variant* sein müssen!

```
Sub Testproc(Optional Einkaufspreis As Variant, Optional Verkaufs-
preis As Variant)
```

Wenn Sie wollen, können Sie nun beim Aufruf nur das zweite Argument angeben und für das erste Argument als Stellvertreter einfach ein Komma einsetzen:

```
Call Testproc(, 300)
```

Wenn Sie das Schlüsselwort *Optional* einsetzen, müssen Sie es übrigens *allen* Argumenten voranstellen!

Die aufgerufene Prozedur kann mit der Funktion *IsMissing* und der in Kürze erläuterten *If*-Anweisung ermitteln, ob ein optionales Argument übergeben oder ausgelassen wurde. Beispielsweise gibt

```
If Not IsMissing(Einkaufspreis) Then Debug.Print Einkaufspreis
```

nur dann den Einkaufspreis im Direktfenster aus, wenn diese Variable auch tatsächlich übergeben wurde.

3.5.6 Variablen »als Referenz« übergeben

Wie Sie vor kurzem sahen, kann beim Aufruf einer Prozedur auch eine Variable übergeben werden. Dann ist das Lokalitätsprinzip jedoch durchbrochen: Wird jene Variable in der Argumentliste der aufgerufenen Prozedur verändert, in der das Argument übernommen wird, wird dadurch auch die korrespondierende Variable der aufrufenden Prozedur verändert.

Nehmen Sie als Beispiel folgende Prozedur *Testproc*:

```
Sub Testproc(zahl)
  zahl = zahl + 10
End Sub
```

Der Aufruf aus einer anderen Prozedur heraus mit

```
x = 50
Call Testproc(x)
Debug.Print x
```

übergibt *Testproc* in der Prozedurvariablen *zahl* den Wert 50. *Testproc* addiert 10 und weist das Resultat 60 wieder *zahl* zu. Diese Zuweisung betrifft jedoch gleichzeitig die korrespondierende Variable *x* der aufrufenden Prozedur, so dass *Debug.Print x* nun entsprechend den Wert 60 ausgibt und damit beweist, dass *x* jetzt den Wert 60 enthält, diese Variable also durch die aufgerufene Prozedur beeinflusst wurde!

Diesen Mechanismus nennt man Variablenübergabe als Referenz. Er kann eingesetzt werden, um es einer aufgerufenen Prozedur zu ermöglichen, an die aufrufende Prozedur Informationen *zurück*zugeben.

Angenommen, eine Prozedur soll zwar die Mehrwertsteuer und den Bruttobetrag ermitteln, die Ergebnisse jedoch nicht auf dem Bildschirm ausdrucken, sondern nur der aufrufenden Prozedur zurückübergeben.

Dazu übergeben Sie beim Aufruf eine Variable. Jede Veränderung der zugehörigen Variablen in der aufgerufenen Prozedur wirkt sich in identischer Weise auf die beim Aufruf angegebene Variable aus. Soll die aufgerufene Prozedur wie *Mehrwertsteuer* zwei Informationen zurückgeben, müssen ihr wie im folgenden Beispiel entsprechend zwei Variablen übergeben werden (Bild 3.13).

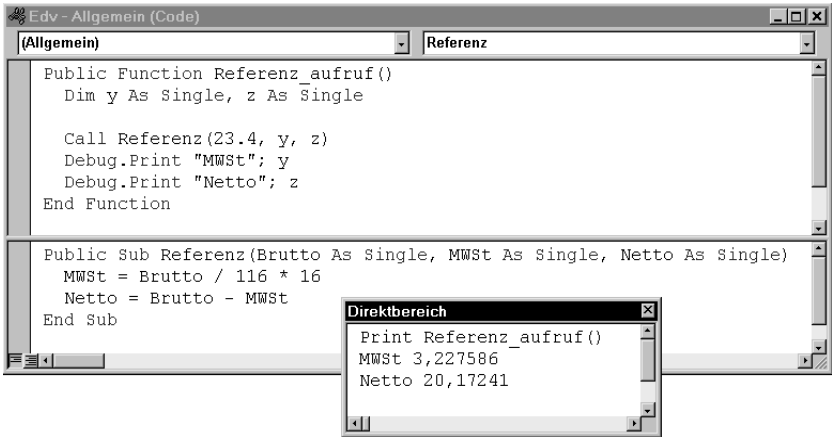


Bild 3.13: Prozeduren »Referenz_aufruf« und »Referenz«

Referenz_aufruf übergibt *Referenz* außer dem Bruttobetrag 23.4 die beiden zuvor deklarierten *Single*-Variablen *y* und *z*, die in diesem Moment keinerlei definierten Wert enthalten.

Jede Änderung einer der in der Argumentliste von *Referenz* aufgeführten Variablen wirkt auf die korrespondierende Variable der aufrufenden Prozedur zurück: Jede Veränderung von *mwst* führt zur gleichen Veränderung bei *x* und jede Veränderung von *netto* zur entsprechenden Beeinflussung von *z*.

Die Zuweisung *mwst = brutto / 116 * 16* weist daher nicht nur der Variablen *mwst* von *Referenz* einen Wert zu, sondern gleichzeitig der korrespondierenden Variablen *y* der Prozedur *Referenz_aufruf*. Und jede Veränderung von *netto* weist den betreffenden Wert entsprechend der zugehörigen Variablen *z* der Prozedur *Referenz_aufruf* zu.

Allgemein: Wird beim Aufruf einer Prozedur in der Argumentliste eine Variable verwendet, verändert jede Zuweisung an die zugehörige Variable der Parameterliste diese Variable des Aufrufers.

Für die übernehmende Prozedurvariable muss nicht unbedingt wie im Beispiel ein anderer Name als für die übergebene Variable verwendet werden. Der

Name ist bedeutungslos. Es spielt keine Rolle, ob Sie der übergebenen Variablen *y* in der aufgerufenen Prozedur eine Variable namens *mwst*, *xyz* oder einfach eine gleichnamige Variable *y* zuordnen.

Access interessiert sich ausschließlich für die *Reihenfolge* in der Argument- und der Parameterliste. Wird in einer aufgerufenen Prozedur die zweite angegebene Variable beeinflusst, hat das Auswirkungen auf die zweite beim Aufruf übergebene Variable, unabhängig von den verwendeten Variablennamen.

ByRef Um die Übergabe als Referenz ausdrücklich festzulegen (und Ihre Programme lesbarer zu machen), können Sie das Schlüsselwort *ByRef* verwenden:

```
Sub Testproc(ByRef zahl As Single)
```

3.5.7 Variablen »als Wert« übergeben

Die Übergabe einer Variablen ermöglicht der aufgerufenen Prozedur, diese Variable zu verändern und auf diese Weise Informationen zurückzugeben. Geschieht das bewusst, ist alles in Ordnung. Allerdings ermöglicht dieser Mechanismus auch ungewollte Veränderungen, da die lokale Abschottung der Variablen verschiedener Prozeduren durchbrochen wird!

Ruft eine Prozedur eine andere auf und übergibt ihr eine Variable, die diese unbeabsichtigt verändert, sind ungewollte Nebenwirkungen unvermeidlich. Um derartige Effekte zu vermeiden, kann eine Variable »als Wert« übergeben werden, indem sie beim Aufruf in Klammern gesetzt wird:

```
Call Wert((x))
```

Nehmen wir an, *x* ist eine *Single*-Variable. Dann muss im Prozedurkopf von *Wert* ebenfalls eine *Single*-Variable deklariert werden:

```
Sub Wert (zahl As Single)
```

Wird in der Prozedur *Wert* der Inhalt von *x* verändert, zum Beispiel durch die Zuweisung *zahl = 10*, bleibt die Variable *x* der aufrufenden Prozedur diesmal unbeeinflusst. Die Klammerung schützt *x* vor jeder Veränderung durch die aufgerufene Prozedur. *Wert* kann zwar lesend, aber nicht verändernd auf diese Variable zugreifen.

Ein weiteres Beispiel (Bild 3.14).

In diesem Beispiel wird *x* der Prozedur *Wert* von *Wert_aufruf* als Referenz übergeben (keine Klammern), *y* dagegen als Wert (Klammern).

Die korrespondierenden Variablen von *Wert* heißen *zahl1* und *zahl2*. Beiden wird der Wert 0 zugewiesen. Die als Referenz übergebene und daher nicht geschützte korrespondierende Variable *x* der aufrufenden Prozedur wird dadurch verändert, die als Wert übergebene Variable *y* jedoch nicht.

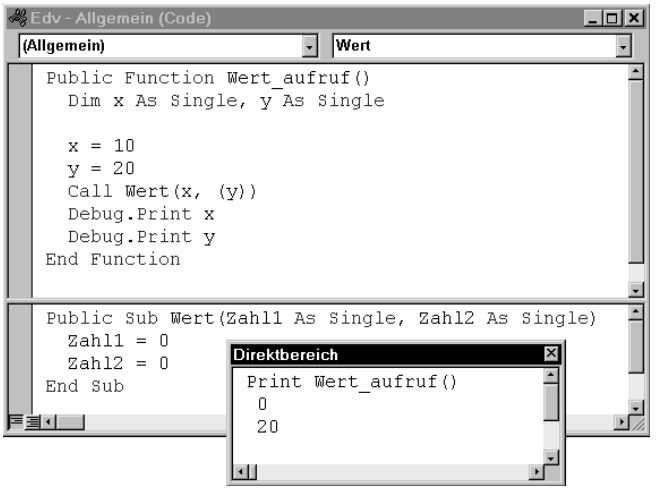


Bild 3.14: Prozeduren »Wert_aufruf« und »Wert«

Eine weitere Möglichkeit, Variablen als Wert zu übergeben, besteht in der Verwendung des Schlüsselworts *ByVal* in der Argumentliste der aufgerufenen Prozedur:

```
ByVal Variablenname As Datentyp
```

Beispielsweise deklariert

```
Sub Wert(zahl1 As Single, ByVal zahl2 As Single)
```

die Prozedur *Wert*, der ein Argument *zahl1* als Referenz oder als Wert übergeben wird, abhängig vom Aufruf. Und ein Argument *zahl2*, das *in jedem Fall als Wert* übergeben wird, unabhängig von der Art des Aufrufs.

Objekte übergeben

Sie können einer Prozedur auch Objekte als Argumente übergeben. Dazu müssen Sie in der Prozedurdeklaration analog zum Variablentyp den Objekttyp angeben. Beispielsweise deklariert

```
Sub Test (x As Control)
```

eine Prozedur *Test*, der ein Verweis auf ein Steuerelement eines Formulars oder Berichts übergeben wird. Es wird in der lokalen Variable *x* übernommen, die vom Typ *Control* ist und somit auf ein Steuerelement verweisen kann. Aufgerufen wird diese Prozedur beispielsweise mit *Call Test(Forms![Aufträge]![Nachname])*, um ihr den aktuellen Inhalt des Steuerelements »Nachname« des Formulars »Aufträge« zu übergeben.

Objekte wie beispielsweise Steuerelemente können auch in Variablen des Typs *Variant* übernommen werden, so dass die Prozedurdeklaration alternativ so formuliert werden könnte:

```
Sub Test (x As Variant)
```

Da die Variable als Referenz übergeben wird, kann die aufgerufene Prozedur das Objekt verändern:

Listing 3.7: Manipulation von Objekteigenschaften

```
Sub Test (x As Control)
  x.Sichtbar = "Nein"
End Sub
```

Diese Prozedur setzt die Eigenschaft »Sichtbar« des übergebenen Steuerelements auf »Nein« und macht es damit unsichtbar.

Umgekehrt kann eine Funktionsprozedur nur die Grunddatentypen (*Integer*, *Single*, *String* etc.), aber keine Objekte als Funktionswert übergeben.

3.6 Geltungsbereiche von Prozeduren und Variablen

3.6.1 Geltungsbereiche von Prozeduren

Die vollständige Syntax der *Sub*- und *Function*-Anweisungen lautet

```
[Public | Private] [Static] Sub Prozedurname [(Argumentliste)]
[Public | Private] [Static] Function Prozedurname [(Argumentliste)] As Typ
```

Public *Public* (Bsp.: *Public Sub Test()*) deklariert öffentliche Prozeduren, die von allen Prozeduren aus aufgerufen werden können, egal in welchen Modulen sie sich befinden.

Prozeduren sind auch ohne dieses Schlüsselwort öffentlich, so dass Sie es weglassen können.

Ausnahme: Ereignisprozeduren sind standardmäßig privat: In Deklarationen von Ereignisprozeduren fügt Access automatisch das Schlüsselwort *Private* ein!

Private *Private* (Bsp.: *Private Sub Test()*) deklariert »private« Prozeduren, die nur von Prozeduren aus aufgerufen werden können, die sich *im gleichen Modul* befinden.

Static Egal, ob Sie eine Prozedur als *Public* oder als *Private* deklarieren, in jedem Fall dürfen Sie das Schlüsselwort *Static* hinzufügen (*Public Static Sub Test()* oder *Private Static Sub Test()*).

Es bewirkt, dass die Inhalte aller lokalen Prozedurvariablen bis zum nächsten Aufruf der Prozedur unverändert erhalten bleiben und nicht wie sonst nach dem Beenden der Prozedur gelöscht werden.

Dieses Schlüsselwort ermöglicht Prozeduren wie die folgende (Bild 3.15).

Der Funktionsprozedur *Statisch* wird ein *Single*-Argument übergeben, das sie zur aktuellen Zwischensumme *Summe* addiert, einer *Single*-Variablen. Das neue Zwischenergebnis wird als Funktionswert übergeben.

Dank des Schlüsselworts *Static* bleibt der aktuelle Inhalt von *Summe* zwischen den einzelnen Aufrufen weiterhin erhalten. Ohne dieses Schlüsselwort würde *Summe* stattdessen jedes Mal nach dem Verlassen der Prozedur gelöscht und beim nächsten Aufruf neu angelegt und mit 0 initialisiert werden, so dass statt 5, 7 und 15 einfach die übergebenen Zahlen 5, 2 und 8 ausgegeben würden.

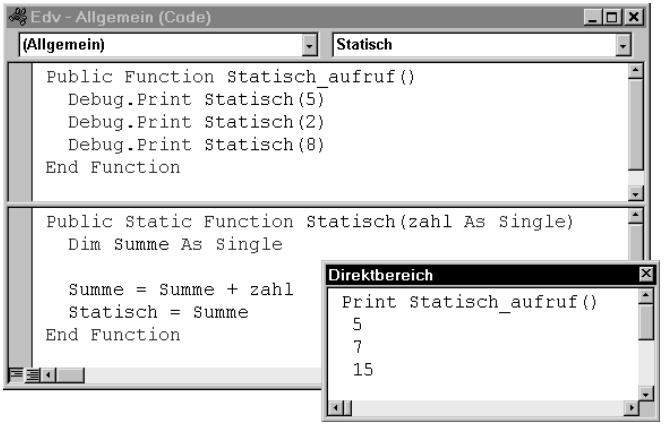


Bild 3.15: Prozeduren »Statisch_aufruf« und »Statisch«

Das Schlüsselwort *Static* kann mit *Private* oder *Public* kombiniert werden!

Beispielsweise deklariert

```
Private Static Sub Demo()
```

eine statische Prozedur *Demo*, die nur von den Prozeduren jenes Moduls verwendet werden kann, in dem sie deklariert ist.

3.6.2 Geltungsbereiche von Variablen und Konstanten

Prinzipiell sind Prozedurvariablen lokal, das heißt, der Geltungsbereich einer Variablen ist auf die Prozedur beschränkt, in der sie deklariert wurde.

Zusätzlich können Variablen jedoch auch öffentlich oder privat sein. Ebenso wie bei Prozeduren bedeutet privat, dass alle Prozeduren des betreffenden Moduls die Variable benutzen können. Und öffentlich heißt, dass wirklich alle Prozeduren die Variable nutzen können, auch wenn sich die betreffenden Prozeduren in einem anderen Modul befinden.

Ein Beispiel: Möglicherweise soll ein und dieselbe Variable in einem umfangreichen Programm in vielen verschiedenen Prozeduren verwendet werden, beispielsweise eine Variable *MWSt*, der mit

```
MWSt = 16
```

der aktuelle Mehrwertsteuersatz zugewiesen wird. Natürlich können Sie diese Variable beim Aufruf der verschiedenen Prozeduren immer wieder explizit übergeben:

Listing 3.8: Wiederholte Variablenübergabe

```
Call Prozedur1 (MWSt)
...
...
Call Prozedur2 (MWSt)
...
...
Call ProzedurN (MWSt)
```

Sie können sich diese Arbeit jedoch ersparen, wenn Sie die Variable als privat deklarieren, also als Variable, die im Gegensatz zu lokalen Variablen nicht nur einer, sondern allen Prozeduren des betreffenden Moduls zur Verfügung steht. Entscheidend dafür ist der Deklarationsort.

Dim

Um eine private Variable zu deklarieren, muss die Deklaration mit *Dim* außerhalb der Prozeduren erfolgen, im Deklarationsbereich des Moduls. Dazu wählen Sie im Prozeduren-Listenfeld den Eintrag »(Deklarationen)«.

Im zugehörigen Abschnitt deklarieren Sie die Variable beispielsweise mit *Dim MWSt_privat As Integer* (Bild 3.16).

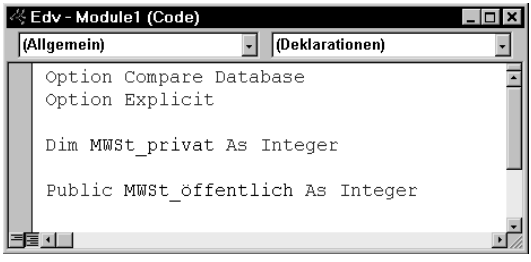


Bild 3.16: Variablendeklaration im Deklarationsabschnitt eines Moduls

Die *Integer*-Variable *MWSt_privat* steht nun allen Prozeduren des betreffenden Moduls zur Verfügung.

Public/Global

Deklarieren Sie eine Variable entsprechend der vorhergehenden Abbildung im Deklarationsbereich statt mit *Dim* mit dem Schlüsselwort *Public*, wird sie dadurch zur »öffentlichen« Variablen und steht allen Prozeduren aller Module der Datenbank zur Verfügung.

Auf Modulebene (im Abschnitt »(Deklarationen)«) deklarierte Variablen behalten ihren Wert (bis die Datenbank geschlossen oder der Befehl AUSFÜHREN | ZURÜCKSETZEN gewählt wird.

Static

Das Gleiche erreichen Sie für lokale Variablen, die Sie innerhalb einer Prozedur deklarieren, wenn Sie dazu das Schlüsselwort *Static* verwenden:

```
Static Variable As Datentyp
```

Der Inhalt von auf diese Weise deklarierten statischen Variablen bleibt bis zum nächsten Aufruf der Prozedur unverändert erhalten.

Das Ganze ähnelt der Deklaration einer Prozedur mit dem Schlüsselwort *Static* (siehe vorhergehenden Abschnitt), im Gegensatz dazu werden jedoch alle anderen Prozedurvariablen, die nicht mit *Static* deklariert wurden, weiterhin bei Erreichen des Prozedurendes gelöscht.

Die Deklaration einer Variablen mit *Static* bezieht sich daher im Gegensatz zur Deklaration einer Prozedur mit *Static* nur auf die deklarierte Variable und nicht auf alle Prozedurvariablen gleichzeitig!

Const

Für den Geltungsbereich von Konstanten gilt das Gleiche wie bei Variablen: Werden sie mit *Const* am Anfang einer Prozedur deklariert, sind sie nur dieser einen Prozedur bekannt.

Deklarieren Sie Konstanten im Deklarationsbereich eines Moduls, sind diese Konstanten allen in diesem Modul enthaltenen Prozeduren bekannt und können von ihnen verwendet werden.

Public/Global

Deklarieren Sie die Konstanten im Deklarationsbereich mit *Public* statt mit *Const*, stehen sie jedem einzelnen Modul und allen darin enthaltenen Prozeduren zur Verfügung (Ausnahme: so genannte »Klassenmodule«).

Zum Beispiel deklariert

```
Public Const MWSt = 16
```

eine Konstante namens *MWSt*, die stellvertretend für die Zahl 16 steht und von allen Prozeduren Ihres Access-Programms verwendet werden kann, egal in welchen Modulen der Datenbank sie sich befinden.

Klassenmodule

Deklarieren Sie in einem Klassenmodul (Formular- oder Berichtsmodul) Variablen auf Modulebene, sind diese Variablen nur in der Formular- bzw. in der Seitenansicht für andere Module verfügbar!

3.6.3 Externe Bezüge auf Bibliotheken und andere Module

Öffentliche Prozeduren können von allen Prozeduren aller Module aufgerufen werden. Allerdings kann es dabei ein Problem geben (Bild 3.17).

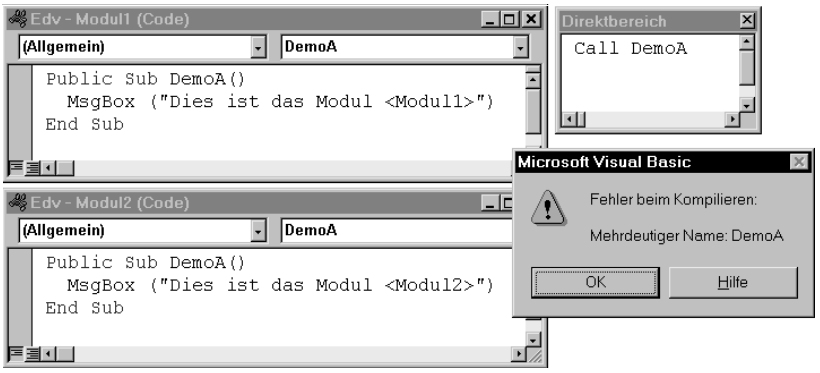


Bild 3.17: Module »Modul1« und »Modul2«, Prozeduren »DemoA«

Im Direktfenster wird die Prozedur *DemoA* aufgerufen. Für Access ist der Prozedurname *DemoA* jedoch nicht eindeutig, da es zwei Prozeduren mit diesem Namen gibt: Sowohl das Modul »Modul1« als auch das »Modul2« enthalten jeweils eine Prozedur *DemoA*!

Sie lösen dieses Problem durch einen Prozeduraufruf der Form

```
Call [Modulname].Prozedurname
```

»Modulname« ist dabei der Name des Moduls, in dem sich die aufzurufende Prozedur befindet (Bild 3.18).

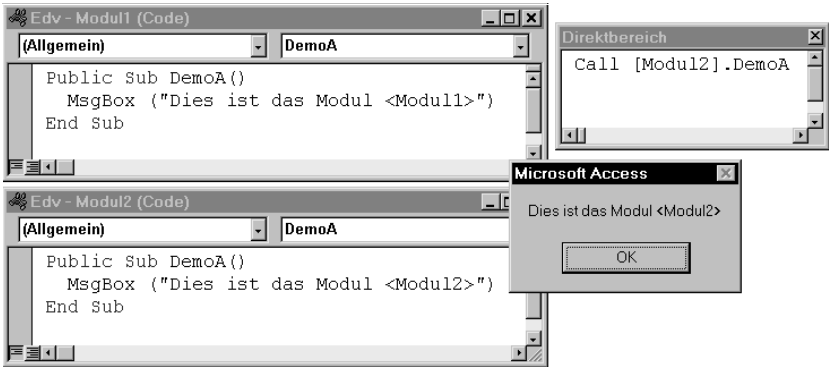


Bild 3.18: Externer Prozedurbezug

Der Aufruf

```
Call [Modul2].DemoA
```

bezieht sich eindeutig auf die in »Modul2« enthaltene Prozedur *DemoA*.

Sie können sogar Prozeduren aufrufen, die sich nicht in der gleichen, sondern in anderen Datenbanken befinden. Dazu müssen Sie mit dem Befehl EXTRAS | VERWEISE... einen Verweis auf diese Datenbank einfügen, das heißt die betreffende Datenbank als zusätzliche Bibliothek bekannt geben. Anschließend können öffentliche Variablen und Prozeduren dieser Bibliotheksdatenbank auch von der aktuellen Datenbank benutzt werden (Bild 3.19).

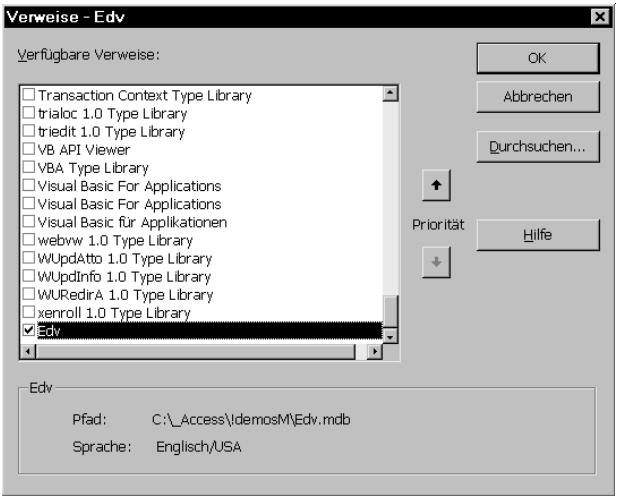


Bild 3.19: Verweise auf Bibliotheken

Dieses Dialogfeld enthält immer aktive Verweise auf die Standard-Bibliotheken, die in jeder Access-Datenbank benutzt werden können. Zusätzlich befinden sich darin inaktive Verweise auf alle möglichen anderen Objektbibliotheken. Um einen dieser Verweise zu aktivieren, aktivieren Sie einfach das zugehörige Kontrollkästchen.

Um eine Ihrer Datenbanken, die wichtige Prozeduren/Variablen enthält, als zusätzliche Bibliothek bekannt zu geben, klicken Sie auf »Durchsuchen...«. Das Dateiauswahl-Dialogfeld erscheint und Sie wählen darin die gewünschte Datenbank aus, beispielsweise EDV.MDB.

Anschließend wird am Ende des Listenfelds ein entsprechender Bibliotheksverweis eingefügt und automatisch aktiviert. Ab jetzt können Sie in der aktuellen Datenbank auch die in EDV.MDB enthaltenen öffentlichen Prozeduren/Variablen verwenden.

Mit den beiden nach oben bzw. nach unten weisenden Pfeilen geben Sie die Priorität einer Bibliothek an. Rufen Sie eine Prozedur wie *DemoA* auf, muss Access alle Bibliotheken nach einer dieser Prozedur/Variablen absuchen, wobei sich Access bei der Suche an die Reihenfolge der Bibliotheken in diesem Listenfeld hält.

Verwenden Sie Objekte einer bestimmten Bibliothek sehr häufig, können Sie die Suche nach diesen Objekten beschleunigen, indem Sie der betreffenden Bibliothek eine höhere Priorität geben. Beispielsweise, indem Sie EDV.MDB selektieren und diesen Eintrag mit dem nach oben weisenden Pfeil ein wenig nach oben verschieben.

Klassenmodule

Prozeduren, die sich in einem Klassenmodul (Formular- oder Berichtsmodul) befinden, können nicht von anderen Datenbanken benutzt werden! Das Gleiche gilt für Variablen, die in Klassenmodulen deklariert werden!

3.7 Test 2

1. Angenommen, der Benutzer wird mit *s = InputBox("Name")* aufgefordert, einen Namen einzugeben, der dann der Variablen *s* zugewiesen wird. Mit welchem Ausdruck prüfen Sie, ob in der Stringvariablen *s* nur ein Nachname enthalten ist (z.B. »Maier«) oder Vor- und Nachname, durch ein Leerzeichen voneinander getrennt (z.B. »Hans Maier«)?
2. Angenommen, *s* enthält tatsächlich einen Vor- und einen Nachnamen. Wie trennen Sie beide voneinander und weisen sie den Variablen *vorname* bzw. *nachname* zu?
3. Erläutern Sie die Bedeutung der Lokalität von Prozedurvariablen.
4. Wie können Sie dennoch erreichen, dass eine Prozedur *Test* auf einen in einer anderen Prozedur *Berechnen* ermittelten Wert *x* zugreifen und mit diesem weiterarbeiten kann?
5. Wie deklarieren Sie eine Funktionsprozedur namens *Test*, der zwei *Double*-Argumente übergeben werden und die als Funktionswert ein *String*-Argument zurückgibt?
6. Was ist der Unterschied zwischen der Übergabe von Variablen als Referenz bzw. als Wert?
7. Wie deklarieren Sie eine Prozedurvariable, die auch nach Verlassen der Prozedur bis zum nächsten Aufruf unverändert erhalten bleiben soll?
8. Wie und wo deklarieren Sie eine Variable, die a) allen Prozeduren eines Moduls uneingeschränkt (Lese- und Schreibzugriff möglich) zur Verfügung stehen soll bzw. b) allen Prozeduren *aller* Module?

9. Wie deklarieren Sie eine Funktionsprozedur *Test*, die a) alle Prozeduren aller Module aufrufen können, deren b) lokale Prozedurvariablen bis zum nächsten Aufruf unverändert erhalten bleiben, der c) zwei Fließkommazahlen einfacher Genauigkeit übergeben werden und die d) als Funktionswert eine Zeichenkette übergibt?
10. Wie erstellen Sie in Access-Prozedurbibliotheken, sprich: Module, die allgemein verwendbare Prozeduren enthalten, die jederzeit in all Ihren Projekten aufgerufen werden können, an denen sie gerade arbeiten?