Kapitel 3
Grundlegende Sprachelemente

Nachdem Sie nun grundlegend mit der VBA-Oberfläche umgehen können, wird es Zeit, sich mit der Sprache selbst zu beschäftigen – die übrigens für alle Excel-Versionen identisch ist!

Das folgende Kapitel beschäftigt sich zunächst mit den grundlegenden Elementen, mit Anweisungen zur Ausgabe von Daten auf dem Bildschirm und zur Eingabe von Daten über die Tastatur. Das versetzt Sie in die Lage, Programme zu schreiben, die den Anwender zur Eingabe bestimmter Texte oder Zahlen auffordern, diese Eingaben weiterverarbeiten und die Ergebnisse auf dem Bildschirm ausgeben.

Wirklich sinnvolle Programme zu schreiben, erfordert zusätzlich jedoch den Einsatz von »Variablen«, in denen Eingaben, Zwischenergebnisse oder Resultate bis zur Weiterverarbeitung gespeichert werden. Also erläutere ich anschließend die Eigenschaften der verschiedenen Daten- und Variablentypen, die Sie dazu verwenden können.

Danach geht es um »Konstanten«, vor allem um die Unmenge an bereits vordefinierten Excel- und VBA-Konstanten, die das Programmiererleben ungemein vereinfachen.

Das wichtigste Objekt beim Programmieren sind Prozeduren. Ich erläutere im Detail, welche Prozedurarten es gibt und welche Eigenschaften sie besitzen. Vor allem gehe ich im Detail auf die verschiedenen Möglichkeiten der »Parameterübergabe« (als Referenz, als Wert etc.) ein und auf den Geltungsbereich sowohl von Variablen als auch von Prozeduren (lokale, globale und statische Variablen bzw. Prozeduren). All das ermöglicht es Ihnen, auch komplexeste Programme »wohlstrukturiert« und somit gut änderungs- und erweiterbar zu erstellen.

Danach sind Sie in der Lage, Excel mit eigenerstellten »Funktionsprozeduren« um zusätzliche Funktionen zu erweitern, die Sie wie die eingebauten Funktionen (*Summe*, *Mittelwert* etc.) benutzen können.

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 Direktfenster eintippen, um damit in diesem Fenster Daten auszugeben.

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

```
Debug.Print Ausdruck
```

»Debug« repräsentiert das Direktfenster. *Debug.Print* bedeutet somit, daß die im nachfolgenden *Ausdruck* angegebenen Daten in diesem Fenster ausgegeben werden sollen. Das passiert, auch wenn das Direktfenster gerade geschlossen ist: Öffnen Sie es, nachdem Ihre Prozedur beendet ist, sehen Sie darin die zuvor von der Prozedur im Direktfenster ausgegebenen Daten.

Jedem Aufruf von *Print* folgt ein Zeilenumbruch. Zwei aufeinanderfolgende Aufrufe wie

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

geben daher die beiden Zeichenketten »Test« und »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 Direktfenster 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 aufeinanderfolgenden Ausgaben gesorgt wird:

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

gibt somit »Test Noch ein Test« aus.

Mit der *Print*-Methode können beliebig viele Ausgaben aneinandergereiht 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

Tab positioniert den Cursor vor der nächsten Ausgabe auf das Zeichen Nummer *x* der Spalte. 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 Direktfenster öffnen, um sich die Ausgaben Ihres Programms anzuschauen. Die Funktion *MsgBox* ist wesentlich angenehmer zu handhaben, da die Ausgaben »Windowsgerecht« 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 Excel bei Bedarf automatisch umbrochen. Ein Beispiel:

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 übergebenen und dank *Debug.Print* im Direktfenster ausgegebenen Funktionswert 1 gehe ich in Kürze ein.)

Das Dialogfeld erscheint übrigens nicht im VBA-, sondern im Excel-Fenster, auf das dabei automatisch umgeschaltet wird.

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. Benutzen Sie dazu folgende Syntax:

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

Titel

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

Schaltflächen

»Schaltflächen« ist eine Zahl, die bestimmt, welche Schaltflächen das Dialogfeld enthält. Die Zahl ergibt sich aus der Addition folgender einzelner Werte:

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

Wert	*Bedeutung*
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)

Tabelle 3.1: Argument »Schaltflächen« (Fortsetzung)

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.

Standardschaltfläche

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.

Symbole

Soll zusätzlich ein Fragezeichensymbol im Dialogfeld erscheinen, addieren Sie zusätzlich 32 und übergeben somit 289 (Bild 3.2):

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

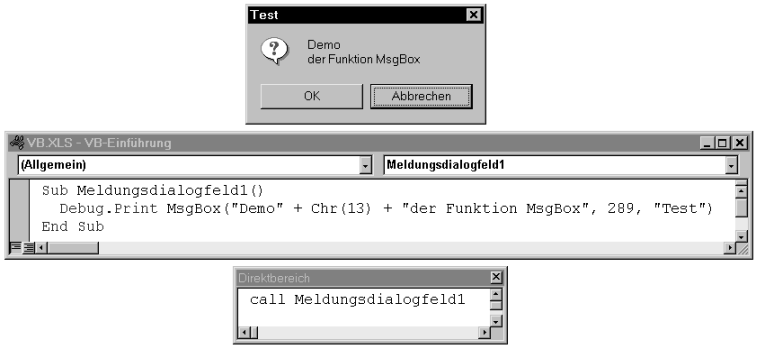


Bild 3.2: Prozedur »Meldungsdialogfeld1«

Da unter anderem eine 256 addiert wurde, ist die zweite Schaltfläche »Abbrechen« vorselektiert und kann mit ↵ aktiviert werden.

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*

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.

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

»Titel« bestimmt wieder den Inhalt der Titelzeile des Dialogfelds und ist eine Zeichenkette wie »Mein Dialogfeld«. Ohne Positionsangaben wird das Dialogfeld in der Bildschirmmitte zentriert (Bild 3.3).

InputBox fordert den Anwender zu einer Eingabe auf. Gibt er wie hier die Zeichenkette »Maier« ein und aktiviert »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 statt dessen beispielsweise in einer Tabelle speichern.

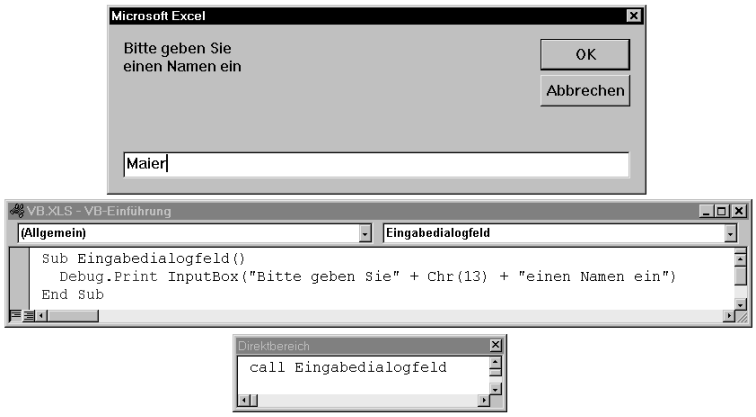


Bild 3.3: Prozedur »Eingabedialogfeld«

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 sogenannten »numerischen Variablen« zuweisen wollen, müssen Sie sie mit der *Val*-Funktion in eine echte Zahl umwandeln.

Beispielsweise ermittelt *x# = Val(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.

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 VBA beim Interpretieren einer Programmzeile auf die Kommentaranweisung *Rem* oder auf das Zeichen »'«, 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.

3.3 Variablen und Konstanten

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

```
Debug.Print 10
Debug.Print "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 (Bild 3.4).

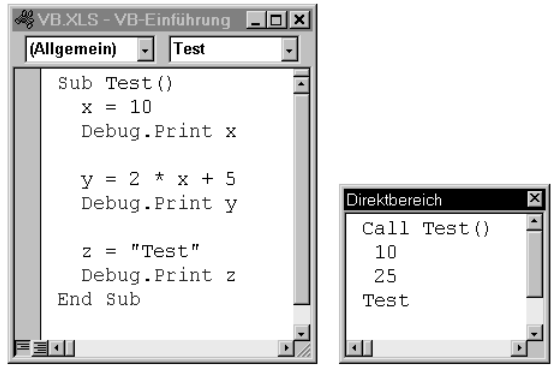


Bild 3.4: Prozedur »Test«

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

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

Ist vor dem Anlegen eines neuen Moduls die Option »Variablen-Deklaration erforderlich« nicht aktiviert (Register »Editor« des Befehls EXTRAS|OPTIONEN...), können Sie darauf verzichten, Variablen, wie in Kürze erläutert, mit dem Befehl DIM zu »deklarieren«.

Variant

Ohne eine solche Deklaration und ohne zusätzliches »Typdeklarationszeichen« besitzt eine Variable immer den Standard-Datentyp *Variant*, der beliebige Datenarten aufnehmen kann: Zahlen und Zeichen, ein Datum oder eine Zeitangabe.

Dieser Typ ist äußerst komfortabel, da VBA – falls notwendig – automatisch einen Datentyp in einen benötigten anderen Typ umwandelt. Ein Beispiel:

Listing 3.1: Typkonvertierung

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

x = 12 weist der Variablen *x* die Zeichenkette »12« zu; *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, daß die Datentypen unverträglich sind.

Der Datentyp *Variant* ist zwar sehr komfortabel, aber VBA 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 VBA in einem Ausdruck der Art *y = x – 10* die in *x* enthaltene Datenart nicht mehr überprüfen, um beispielsweise festzustellen, daß der Inhalt von *x* möglicherweise eine Zeichenkette wie »12« ist, die erst in die Zahl 12 umgewandelt werden muß.

Typdeklarationszeichen

Aus Effizienzgründen sollten Sie nicht ständig den Datentyp *Variant* verwenden, sondern mit einem »Typdeklarationszeichen« bekanntgeben, welche Datenarten die betreffende Variable aufnehmen soll (bei den Datentypen ohne spezielles Typdeklarationszeichen müssen Sie die im folgenden Kapitel erläuterte Deklarationstechnik verwenden).

Tabelle 3.3: Datentypen

Datentypname	Zeichen	Zulässige Werte	Belegter Platz
Byte	keines	0 bis 255 (nur ganze Zahlen)	1 Byte
Boolean	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

Tabelle 3.3: Datentypen (Fortsetzung)

Datentypname	Zeichen	Zulässige Werte	Belegter Platz
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/Uhrzeiteingabe

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« (Zeichenkettenvariablen) *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 VBA 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. Statt dessen erhalten Sie eine Fehlermeldung, die darauf beruht, daß es nur eine Variable namens *x* geben kann!

3.3.2 Variablen deklarieren

Sie können Variablen in einer Prozedur ohne besondere Umstände einfach benutzen und den Typ der Variablen mit dem zugehörigen Typkennzeichen festlegen, wenn Sie wollen (vorausgesetzt, vor dem Anlegen des Moduls war im Register »Editor« des Befehls EXTRAS| OPTIONEN... die Option »Variablen-Deklaration erforderlich« deaktiviert!).

Dim

Alternativ dazu können Sie eine Variable jedoch auch vor der ersten Benutzung mit einer der Anweisungen *Dim*, *Public, Private* oder *Static* »deklarieren«, wobei *Dim* die »gebräuchlichste« Deklarationsanweisung ist:

```
Dim Variablenname[Typkennzeichen]
```

Zum Beispiel am Anfang einer Prozedur:

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.

Ohne Typkennzeichen wird eine Variable vom Standardtyp *Variant* deklariert. *Dim x* deklariert daher eine Variable namens *x*, die den Typ *Variant* besitzt.

Dim As

Wirklich nützlich 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 Benutzung der Variablen 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 VBA 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 VBA 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 statt dessen »Stringvariablen fester Länge« verwenden, die mit der Anweisung

```
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 VBA 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.

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* eine Stringvariable variabler Länge und *z* eine Stringvariable mit einer festen Länge von zehn Zeichen.

Deklarationszwang

Die explizite Deklaration von Variablen scheint zunächst umständlich zu sein, besitzt in der Praxis jedoch nur Vorteile. Ich empfehle Ihnen dringend, dafür zu sorgen, daß VBA Sie dazu zwingt!

Dazu fügen Sie in den Deklarationsabschnitt eines Moduls (Excel 97-2002: Eintrag »(Deklarationen)« im Prozeduren-Listenfeld 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.

Ist das Kontrollkästchen »Variablen-Deklaration erforderlich« im Register »Editor« des Befehls EXTRAS|OPTIONEN... aktiviert, fügt VBA diese Anweisung sogar automatisch in jedes neue Modul ein.

Dieser freiwillige Zwang zur Deklaration ist äußerst sinnvoll, wenn Sie Wert auf übersichtliche und gut strukturierte Programme legen, in denen unnötige Fehler von vornherein vermieden werden!

Geltungsbereich

Werden Variablen am Anfang einer Prozedur mit *Dim* deklariert, sind sie nur dieser einen Prozedur »bekannt« und können nur in ihr verwendet werden. Alternativ dazu können Sie, wie in Kürze erläutert, auch »öffentliche« Variablen deklarieren.

3.3.3 Konstanten definieren

Jede Zahl oder Zeichenkette, die sich in Ihrem Programmtext befindet, ist eine literale Konstante. Die Anweisung

```
Debug.Print "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«, also 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 und danach mit einer Anweisung *Const x = 10* einen anderen Wert zuzuweisen!

Diese Eigenschaft von Konstanten ist immer dann nützlich, wenn Sie mit Werten hantieren (Zahlen oder Zeichenketten), die tatsächlich 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 auch in Abhängigkeit von anderen Konstanten deklarieren; beispielsweise mit *Const A = 20* eine Konstante *A* und danach mit *Const B = A / 2* eine Konstante *B*, die den zuvor *A* zugewiesenen Wert benutzt. Ändern Sie irgendwann die Zuweisung an die Konstante *A*, ändert sich automatisch der *B* zugewiesene Wert entsprechend.

Geltungsbereich

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

3.3.4 Vordefinierte Konstanten

Excel und VBA stellen Ihnen eine Unmenge vordefinierter und »global« gültiger Konstanten zur Verfügung. Damit sind Konstanten gemeint, die einen von VBA oder von Excel vorgegebenen Wert besitzen und die Sie in beliebigen Prozeduren anstelle dieses Werts einsetzen können.

Die Namen dieser Konstanten beginnen immer mit zwei Buchstaben, die die »Objektbibliothek« bezeichnen, die sie zur Verfügung stellt. Excel-Konstanten beginnen beispielsweise mit »xl...« 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)
```

Sollen 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)
```

Eine Übersicht über die vordefinierten *MsgBox*-Konstanten erhalten Sie im »Objektkatalog« (Erläuterung im Anhang), indem Sie darin die interessierende Objektbibliothek auswählen, im Beispiel »VBA«, und im Listenfeld darunter »vbMsgBoxResult« oder »vbMsgBox-Style« selektieren. Im rechten Listenfeld werden daraufhin die von der betreffenden Bibliothek zur Verfügung gestellten Konstanten aufgelistet (Bild 3.5).

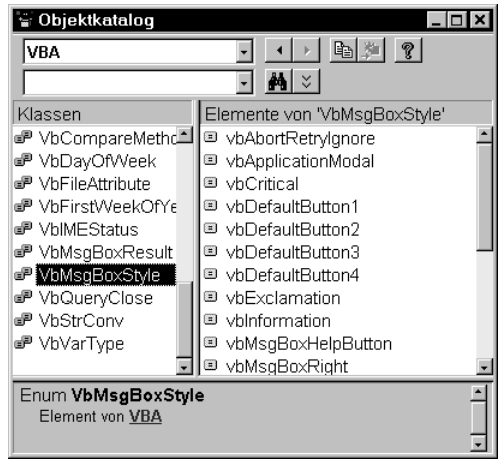


Bild 3.5: Vordefinierte MsgBox-Konstanten

3.4 Test

1. Ein Programm soll im Direktfenster (Excel 5.0: Testfenster) 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 wie geben Sie seine Eingabe anschließend im Direktfenster (Excel 5.0: Testfenster) 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 aufeinanderfolgenden Programmzeilen *x = 10* und *x = 20* in *x*?
7. Welchen Wert speichern die beiden aufeinanderfolgenden 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 Lösungen

1. a) *Debug.Print 100, 200, 300* gibt die zweite und dritte Zahl jeweils in der nächsten »Bildschirmzone« aus,
   b) *Debug.Print 100; " ";200; " "; 300* gibt die Zahlen direkt aufeinanderfolgend aus, fügt dazwischen aber jeweils zwei Leerzeichen ein, und
   c) *Debug.Print 100; Tab(10); 200; Tab(20); 300* gibt die beiden letzten Zahlen an genau definierten Spaltenpositionen aus.
2. *MsgBox("Okay, hat geklappt!", 0)*
3. Indem Sie die 0 durch die vordefinierte VBA-Konstante *vbOKOnly* ersetzen: *MsgBox("Okay, hat geklappt!", vbOKOnly)*.
4. *Debug.Print InputBox("Bruttobetrag?")*
5. *InputBox* übergibt immer eine Zeichenkette, auch wenn der Benutzer eine Zahl eingibt, beispielsweise den Bruttobetrag 100. Um das Doppelte davon in einer numerischen Variablen wie *x#* zu speichern, muß die übergebene Zeichenkette »100« mit der *Val*-Funktion in die Zahl 100 umgewandelt werden (*Val(InputBox("Bruttobetrag?"))*). Anschließend können Sie damit rechnen: *x# = 2 * Val(InputBox("Bruttobetrag?"))*.
6. 20, da jede Zuweisung den zuvor in der betreffenden Variablen enthaltenen Wert einfach überschreibt.
7. Ebenfalls 20.
8. 10 Zeichen, entsprechend der Deklaration dieser »Stringvariablen fester Länge«; daran ändert auch die Zuweisung der nur sieben Zeichen langen Zeichenkette »Porsche« nichts (die restlichen drei Zeichen sind »aufgefüllte« Leerzeichen).
9. 7 Zeichen, da in »Stringvariablen variabler Länge« immer nur die der Variablen explizit zugewiesenen Zeichen enthalten sind.

3.6 Prozeduren

3.6.1 Vorgegebene Funktionsprozeduren

Excel (beziehungsweise VBA) stellt Ihnen eine ganze Menge eingebauter Funktionsprozeduren zur Verfügung, und zwar »numerische Funktionen« und »Stringfunktionen«:

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

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

```
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*: *Len*

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

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.

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

```
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

Mit Hilfe von Stringfunktionen können Sie prüfen, ob in einer Zeichenkette eine bestimmte andere Zeichenkette enthalten ist, Teile aus einer Zeichenkette heraustrennen und so weiter, kurz: Zeichenketten 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 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 *Right*

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

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

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 *Mid*

```
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«?

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

```
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!

3.6.2 Prozeduren selbst erstellen

Excel 5.0 Wie Sie wissen, erstellen Sie in Excel 5.0 eine neue Prozedur, indem Sie einfach die erste Zeile in einem Modulblatt eintippen. Was Sie noch nicht wissen: daß Sie *vor* dem Schlüsselwort *Sub* bzw. *Function*, das den Prozedurtyp angibt, zusätzlich eines der beiden Schlüsselworte *Private* und *Static* eingeben können (oder auch beide zusammen, z.B. in der Form *Private Static Sub Test*).

In Excel 97-2002 sind die gleichen Schlüsselwörter verwendbar, sie werden aber bei Bedarf automatisch eingefügt. Zunächst erstellen Sie im momentan aktiven Modul mit EINFÜGEN|PROZEDUR... bzw. durch Klicken auf das zugehörige Symbol eine neue Prozedur (Bild 3.6).

Bild 3.6: Prozedur erstellen

Typ und Geltungsbereich

Wie der automatisch eingefügte »Prozedurrumpf« aussieht, hängt von den ausgewählten Optionen ab. Die unter »Typ« gewählte Option bestimmt die Art der Prozedur: Je nach Option fügt VBA vor einem Prozedurnamen wie *Test* eines der drei Schlüsselwörter *Sub* (Sub-Prozedur), *Function* (Funktionsprozedur) oder *Property* (Eigenschaftsprozedur) ein.

Gleiches gilt für die unter »Geltungsbereich« gewählte Option: »Öffentlich« fügt zusätzlich das Schlüsselwort *Public* ein, »Privat« 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 »Funktion« und »Privat«, und aktivieren Sie zusätzlich das Kontrollkästchen »Alle lokalen Variablen...«, lautet der eingefügte »Prozedurrumpf« (den Sie in Excel 5.0 statt dessen per Hand eintippen):

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

Damit wird eine »private Funktionsprozedur« namens *Test* deklariert, deren Variablen »statisch« sind. Die drei Schlüsselwörter *Public*, *Private* und *Static* sind jedoch alle »optional«, können also auch weggelassen werden.

Der Beweis dafür ist einfach zu führen: In Excel 97-2002 fügt VBA automatisch zumindest eines der beiden Schlüsselwörter *Public* oder *Private* ein; löschen Sie das betreffende Schlüsselwort anschließend, funktioniert die betreffende Prozedur genauso wie zuvor!

Ich will damit keinesfalls sagen, daß diese Schlüsselwörter bedeutungslos seien. Da die Schlüsselwörter *Public*, *Private* und *Static* aber zumindest nicht gerade »lebenswichtig« für eine Prozedur sind, bespreche ich sie später im Kapitel 3.7.1, »Geltungsbereiche von Prozeduren«.

Das reduziert die Syntax einer Prozedur zunächst auf die beiden folgenden wesentlich »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
```

Sowohl Sub-Prozeduren als auch Funktionsprozeduren bestehen aus einem »Prozedurrumpf«, zwischen dem sich beliebig viele auszuführende Anweisungen befinden können. Der entscheidende Unterschied ist der zusätzliche Ausdruck *Prozedurname = Ausdruck*, der in einer Funktionsprozedur an einer beliebigen Stelle vorkommen darf.

Funktionsprozeduren

Mit diesem Ausdruck wird in einer Funktionsprozedur der Funktionswert definiert, den die Funktion 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 festlegen:

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

Beispielsweise übergibt die Funktion

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

einen *Double*-Wert als Funktionswert.

Alternativ dazu können Sie nach dem Funktionsnamen das zugehörige Typkennzeichen des Funktionswerts 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 Aufgabe eine Sub-Prozedur zu erstellen, die die benötigten Ausgabeanweisungen enthält. Immer dann, wenn eine der Funktionsprozeduren wieder einmal die betreffenden Texte ausgeben muß, ruft sie einfach mit der Anweisung *Call* die dafür zuständige Sub-Prozedur auf:

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

Beispielsweise ruft

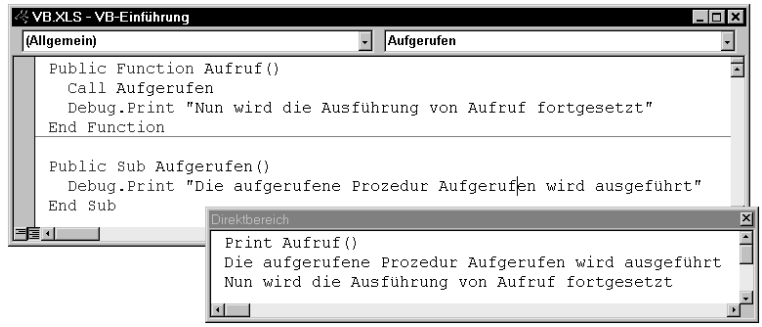
```
Call Aufgerufen
```

Prozeduraufrufe

die Sub-Prozedur *Aufgerufen* auf. Enthält eine Funktionsprozedur diese Anweisung, »verzweigt« der Programmfluß zu dieser 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 3.7).

Bild 3.7: 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. Dadurch wird die Ausführung von *Aufruf* vorübergehend unterbrochen und nun die Sub-Prozedur *Aufgerufen* ausgeführt, die die Zeichenkette »Die aufgerufene Prozedur Aufgerufen wird ausgeführt« ausgibt.

End Sub beendet die Ausführung der Sub-Prozedur. VBA 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* ebensogut einfach mit *Aufgerufen* ausgeführt werden (die »Argumentliste« darf dann jedoch nicht von Klammern umgeben sein!).

Denken Sie daran, daß Sie eine Sub-Prozedur zu Testzwecken statt mit *Call* auch ausführen können, indem Sie den Cursor in eine beliebige Prozedurzeile setzen und AUSFÜHREN|SUB/USERFORM AUSFÜHREN wählen oder auf das abgebildete Symbol klicken.

3.6.3 Lokale Variablen in Prozeduren

Nun ein praktisches Beispiel: Eine Sub-Prozedur soll, basierend auf einer Zahl *x*, die darin enthaltene Mehrwertsteuer und den Nettobe-

trag 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 3.8).

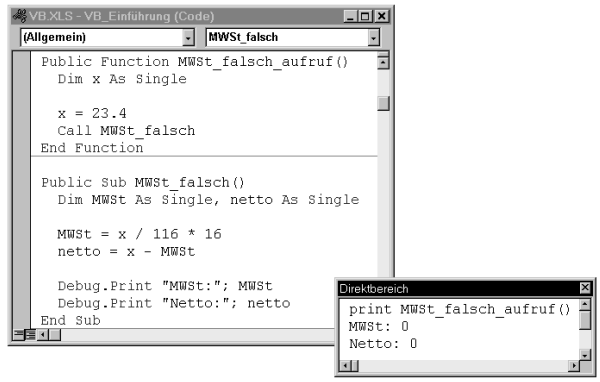


Bild 3.8: Prozeduren »MWSt_falsch_aufruf« und »MWSt_falsch«

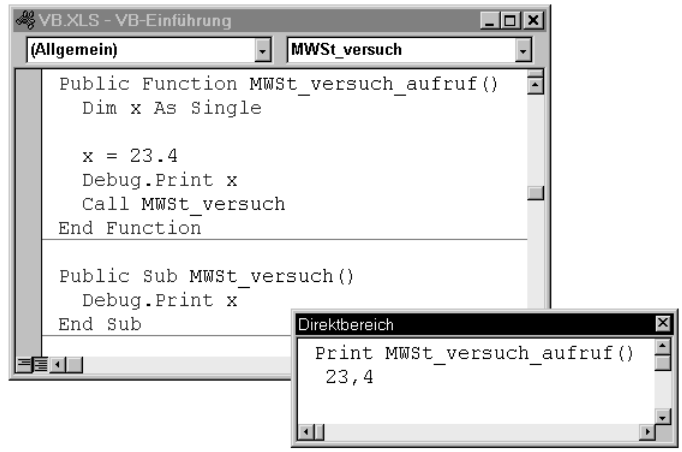
MWSt_falsch_aufruf weist der *Single*-Variablen *x* den Bruttobetrag 23.4 zu und ruft *MWSt_falsch* auf. *MWSt_falsch* ermittelt mit *mwst = x / 116 * 16* die in *x* enthaltene Mehrwertsteuer und weist sie der *Single*-Variablen *mwst* zu. Danach ermittelt *netto = x – mwst* den Nettobetrag und speichert 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 3.9).

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! Damit kennen wir nun auch die Ursache der fehlgeschlagenen Mehrwertsteuerberechnung:

Bild 3.9: Prozeduren »MWSt_versuch_aufruf« und »MWSt_versuch«



Prozeduren können nur Variablen benutzen, die in der betreffenden Prozedur deklariert werden! Ursache dafür ist die »Lokalität von Prozedurvariablen«: Eine Prozedurvariable ist immer »lokal« zur Prozedur, in der sie deklariert wird, und nur dieser einen Prozedur bekannt (tatsächlich existiert sie sogar nur während der Ausführung der betreffenden Prozedur und wird beim Verlassen der Prozedur wieder gelöscht). Für andere Prozeduren ist die betreffende Variable nicht existent!

Daher ist eine Variable *x*, die in einer Prozedur *Berechnen* deklariert wurde, für eine Prozedur *Test* »nicht sichtbar« und kann nicht von ihr beeinflußt werden. Wird in *Test* ebenfalls eine Variable *x* deklariert und benutzt, wird die gleichnamige Variable *x* der Prozedur *Berechnen* dadurch nicht beeinflußt.

In großen Programmen ist das ein enormer Sicherheitsvorteil, da dadurch ausgeschlossen wird, daß durch eine Zuweisung wie *x = 10* in einer Prozedur *Test* der zuvor in einer Prozedur *Berechnen* mühsam errechnete Wert von *x* überschrieben wird, nur weil Sie in *Test* versehentlich den gleichen Variablennamen benutzen.

3.6.4 Argumente an Prozeduren übergeben

Diese »Lokalität« ist jedoch ein Problem, wenn, wie bei der geplanten Mehrwertsteuerberechnung, der Inhalt einer Variablen ganz bewußt einer anderen Prozedur übergeben werden soll. Glücklicherweise stellt VBA dafür 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« sind Variablen, jeweils durch ein Komma getrennt. Darin 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* bekanntgeben:

```
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 3.10).

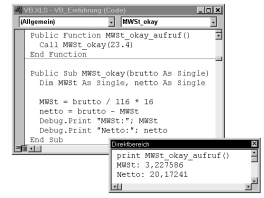


Bild 3.10: Prozeduren »MWSt_okay_aufruf« und »MWSt_okay«

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.

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 die Deklaration von *Stringtest* die Übergabe einer Stringvariablen enthalten, in der diese Zeichenkette bei der Übergabe gespeichert wird:

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

oder

```
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.

3.6.5 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 VBA 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* und ohne Klammern verwenden:

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

Angewandt auf das Beispiel:

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

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
Verkaufspreis 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.

3.6.6 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, Informationen an die aufrufende Prozedur *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 als Referenz. 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 entsprechend zwei Variablen übergeben werden (Bild 3.11).

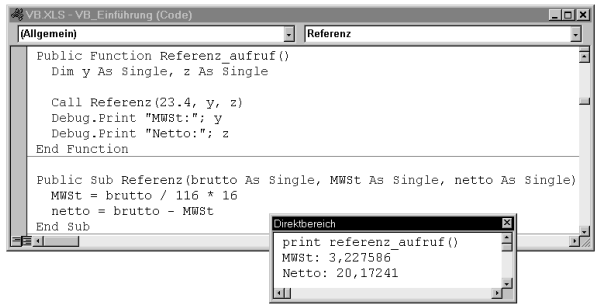


Bild 3.11: Prozeduren »Referenz_aufruf« und »Referenz«

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 *y* 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 Argumentliste der aufgerufenen Prozedur eine Variable namens *mwst*, *xyz* oder einfach eine gleichnamige Variable *y* zuordnen.

VBA interessiert sich ausschließlich für die *Reihenfolge* in der Argument- und der Parameterliste. Wird in einer aufgerufenen Prozedur die zweite Variable der Argumentliste 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)
```

3.6.7 Variablen »als Wert« übergeben

Die Übergabe einer Variablen ermöglicht es der aufgerufenen Prozedur, diese Variable zu verändern und so 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 statt dessen »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 schreibend auf diese Variable zugreifen.

Ein weiteres Beispiel:

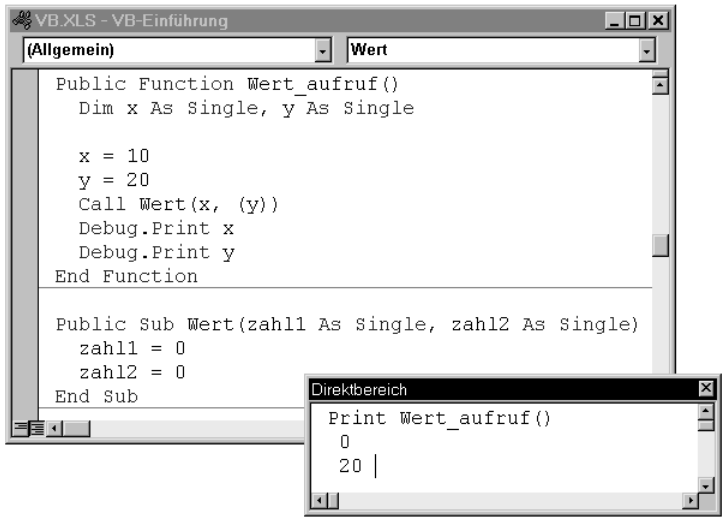


Bild 3.12: Referenz- und Wert-Übergabe

In diesem Beispiel wird der Prozedur *Wert* von *Wert_aufruf* die Variable *x* 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.

3.7 Geltungsbereiche von Prozeduren und Variablen

3.7.1 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* (*Public Sub Test()*) deklariert »öffentliche« Prozeduren. Öffentliche Prozeduren können von allen Prozeduren aus aufgerufen werden, egal in welchen Modulen sie sich befinden.

Sub- und *Function*-Prozeduren sind jedoch auch ohne Schlüsselwort öffentliche Prozeduren, so daß Sie es bei diesen Prozeduren einfach weglassen können.

Die große Ausnahme: Ereignisprozeduren sind standardmäßig privat (in Deklarationen von Ereignisprozeduren fügt VBA automatisch das Schlüsselwort *Private* ein).

Private *Private* (*Private Sub Test()*) deklariert »private« Prozeduren. Das bedeutet, daß die betreffende Prozedur nur von Prozeduren aus aufgerufen werden kann, 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 3.13).

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* statt dessen beim Verlassen der Prozedur gelöscht und beim nächsten Aufruf neu angelegt und wieder mit 0 initialisiert werden, so daß statt 5, 7 und 15 einfach die übergebenen Zahlen 5, 2 und 8 ausgegeben würden.

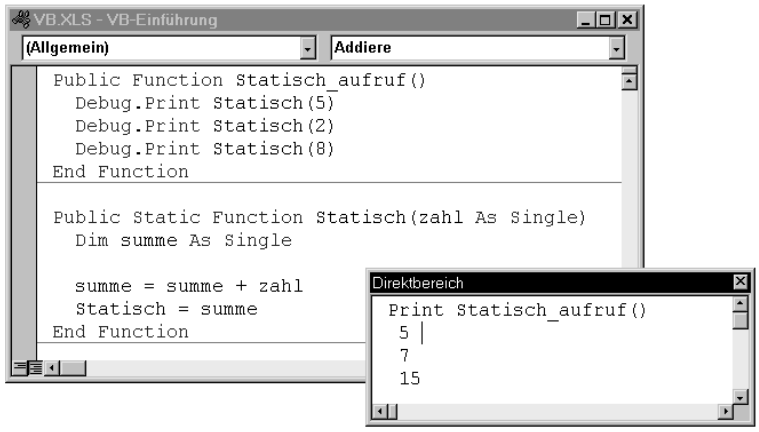


Bild 3.13: Statische Prozeduren

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.

3.7.2 Geltungsbereiche von Variablen und Konstanten

Normalerweise sind Prozedurvariablen »lokal«, daß 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«, 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 3.7: Wiederholte Variablenübergabe

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

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

Dim und Private

Die Deklaration muß außerhalb der Prozeduren erfolgen, im Deklarationsbereich des Moduls (Excel 97-2002: im Prozeduren-Listenfeld den Eintrag »(Deklarationen)« wählen).

In diesem Bereich deklarieren Sie die Variable wahlweise mit *Dim Variablenname As Datentyp* oder mit *Private Variablenname As Datentyp* (Bild 3.14).

Bild 3.14: Variablendeklaration im Deklarationsabschnitt eines Moduls

Die Variable *MWSt_privat* ist nun eine private Variable aller Prozeduren des betreffenden Moduls, steht also all diesen Prozeduren uneingeschränkt zur Verfügung. Um diesen Sachverhalt klar auszudrücken, sollten Sie entgegen der Abbildung statt *Dim* besser das Schlüsselwort *Private* verwenden.

Public

Deklarieren Sie eine Variable im Deklarationsbereich mit dem Schlüsselwort *Public*, ist sie dadurch keine private Variable des be-

treffenden Moduls mehr, sondern eine »öffentliche« Variable, die nun sogar allen Prozeduren *aller Module* zur Verfügung steht!

Übrigens behalten auf Modulebene (im Abschnitt »(Deklarationen)«) deklarierte Variablen ihren Wert unverändert, bis der Befehl AUSFÜHREN|BEENDEN (Excel2000/2002: 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 entsprechend 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 dagegen im Deklarationsbereich eines Moduls, sind sie allen in diesem Modul enthaltenen Prozeduren bekannt und können von ihnen verwendet werden.

Public

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

Zum Beispiel deklariert

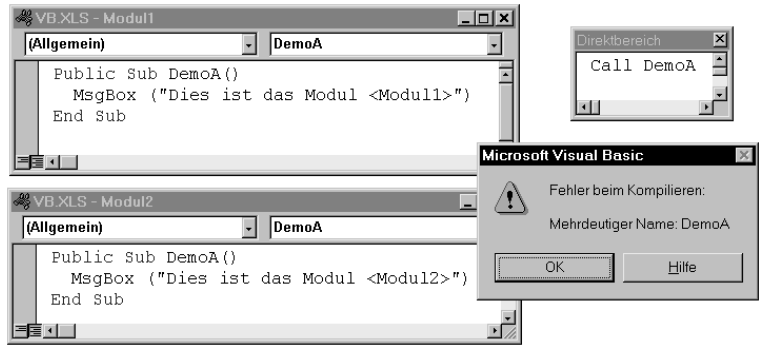
```
Public Const MWSt = 16
```

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

3.7.3 Externe Bezüge auf Bibliotheken und andere Module

Öffentliche Prozeduren – Prozeduren ohne den Zusatz *Private* – können von allen Prozeduren aller Module aufgerufen werden. Trotzdem kann es beim Aufruf ein Problem geben (Bild 3.15).

Bild 3.15: Mehrdeutiger Prozedurname



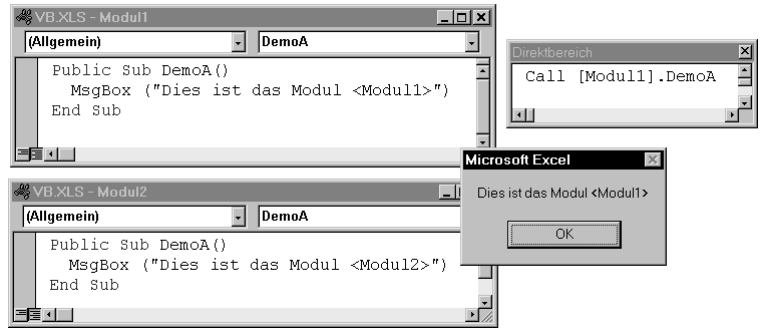
Im Direktfenster wird die Prozedur *DemoA* aufgerufen. Für Excel ist der Prozedurname *DemoA* jedoch nicht eindeutig, da es zwei Prozeduren mit diesem Namen gibt: Sowohl das Modul »Modul1« als auch das »Modul2« enthalten jeweils eine Prozedur *DemoA*!

Sie lösen dieses Problem durch einen Prozeduraufruf der Form

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

»Modulname« ist der Name des Moduls, in dem sich die aufzurufende Prozedur befindet (Bild 3.16).

Bild 3.16: Externer Prozedurbezug



Der Aufruf

```
Call [Modul1].DemoA
```

bezieht sich eindeutig auf die in »Modul1« enthaltene Prozedur *DemoA*.

Bezüge auf externe Bezüge

Sie können sogar Prozeduren aufrufen, die sich nicht in der gleichen, sondern in anderen Arbeitsmappen befinden – sogar wenn diese Mappen momentan nicht geöffnet sind! Dazu müssen Sie mit dem Befehl EXTRAS|VERWEISE... einen »Verweis« auf diese Mappe einfügen und die betreffende Mappe auf diese Weise als zusätzliche »Bibliothek« bekanntgeben. Anschließend können öffentliche Variablen und Prozeduren dieser Bibliothek auch von Prozeduren der aktuellen Mappe benutzt werden (Bild 3.17).

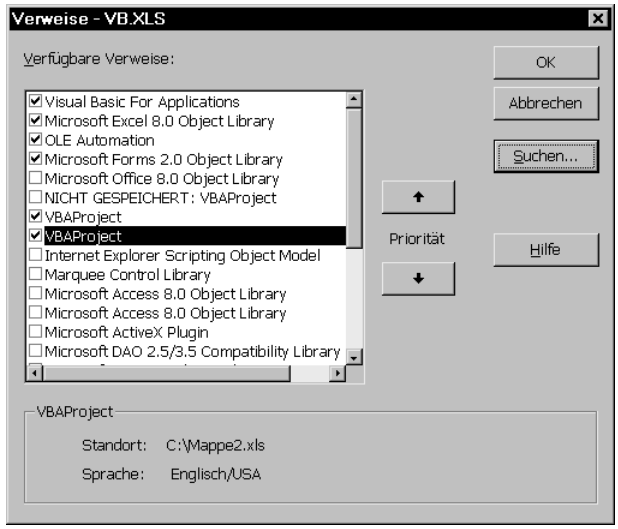


Bild 3.17: Verweise auf Bibliotheken

Dieses – in Excel 5.0 ähnliche – Dialogfeld enthält immer »aktive« Verweise auf die »Standard-Bibliotheken«, die in jedem Projekt 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.

Wie Sie wissen, können Sie mit VBA mehrere Projekte gleichzeitig bearbeiten. Tun Sie das gerade, enthält das Listenfeld für jedes geöffnete Projekt (außer dem aktuellen) einen eigenen Eintrag »VBAProject« bzw. – falls das Projekt noch nicht gespeichert wurde –

»NICHT GESPEICHERT: VBAProject« (dann muß das Projekt vor dem nächsten Schritt zunächst gespeichert werden).

Jeder dieser Einträge steht stellvertretend für das betreffende Projekt. Aktivieren Sie das zugehörige Kontrollkästchen, wird das betreffende Projekt von VBA in seine Liste »zu durchsuchender Bibliotheken« aufgenommen. Rufen Sie eine Prozedur wie *DemoA* auf, wird daher ab jetzt auch in diesem Projekt nach der betreffenden Prozedur »gefahndet«. Allgemein ausgedrückt: Alle im betreffenden Projekt enthaltenen öffentlichen Prozeduren und Variablen können ab jetzt auch vom aktuellen Projekt benutzt werden.

Ist das Projekt (die Excel-Mappe), das auf diese Weise öffentlich zugänglich gemacht werden soll, momentan nicht geöffnet, klicken Sie einfach auf »Suchen...«: Das Dateiauswahl-Dialogfeld erscheint, und Sie wählen darin die Mappe aus, die das betreffende Projekt enthält, beispielsweise MAPPE2.XLS. Im Listenfeld erscheint ein zusätzlicher Eintrag »VBAProject«, der automatisch aktiviert wird.

Ist ein Eintrag selektiert, erscheint übrigens unterhalb des Listenfelds eine Kurzbeschreibung des Verweises, beispielsweise der Name der zugehörigen Arbeitsmappe. Dadurch wird der »Durchblick« bei mehreren auf diese Weise eingefügten Verweisen, die allesamt »VBAProject« heißen, deutlich erleichtert.

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 VBA alle Bibliotheken nach dieser Prozedur/Variablen durchsuchen und hält sich bei der Suche an die Reihenfolge der Bibliotheken in diesem Listenfeld.

Verwenden Sie Objekte einer bestimmten Bibliothek sehr häufig, können Sie die Suche daher beschleunigen, indem Sie der betreffenden Bibliothek eine höhere Priorität geben, so daß sie als eine der ersten Bibliotheken durchsucht wird – beispielsweise, indem Sie den Eintrag »VBAProject« selektieren, der für MAPPE2.MDB steht, und ihn mit dem nach oben weisenden Pfeil ein wenig nach oben verschieben.

Klassenmodule

Prozeduren, die sich in einem Klassenmodul befinden, können nicht von anderen Projekten benutzt werden! Das gleiche gilt für Variablen, die in Klassenmodulen deklariert werden!

3.8 Test

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 wie weisen Sie 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 »Prozedurbibliotheken«, sprich: Module, die allgemein verwendbare Prozeduren enthalten, die jederzeit in all ihren Projekten aufgerufen werden können, an denen Sie gerade arbeiten – und zwar ohne daß erst umständlich die Excel-Mappe geöffnet wird, die diese Prozeduren enthält?

3.9 Lösungen

1. Der Ausdruck *Instr(1, s, " ")* prüft, ob in *s* irgendwo ab Position 1 (also ab dem ersten Zeichen) ein Leerzeichen enthalten ist. Gibt es darin tatächlich wie in »Hans Maier« ein Leerzeichen, übergibt die *Instr*-Funktion dessen Position, im Beispiel also die Zahl 5.

2. Vorname behandeln: *vorname = Left(s, Instr(1, s, " ") – 1)*, denn *Instr(1, s, " ")* übergibt im Beispiel »Hans Maier« den Funktionswert 5, der Ausdruck auf der rechten Seite vom Gleichheitszeichen ist daher gleichbedeutend mit *Left(s, 5 – 1)* bzw. *Left(s, 4)* und übergibt die ersten vier Zeichen »Hans«. Nachname behandeln: *nachname = Right(s, Len(s) – Instr(1, s, " "))*, denn *Right(s, Len(s) – Instr(1, s, " "))* ist wie erläutert gleichbedeutend mit *Right(s, Len(s) – 5)*, und da *Len(s)* 10 übergibt, ist der Ausdruck gleichbedeutend mit *Right(s, 10 – 5)*, übergibt also die rechten 5 Zeichen »Maier«. Einfachere Alternative: *Mid(s, Instr(1, s, " ") + 1)* übergibt den gesamten Rest der Zeichenkette ab der Position *Instr(1, s, " ") + 1*, also ab der dem Leerzeichen folgenden Position 6.

3. In Prozeduren deklarierte Variablen sind nur dieser und keiner anderen Prozedur bekannt, können also von anderen Prozeduren nicht beeinflußt werden. Daran ändern auch mehrere gleichnamige Variablen *x* in mehreren Prozeduren nichts, da es sich gemäß diesem Prinzip um verschiedene Variablen handelt, auch wenn sie zufällig den gleichen Namen besitzen.

4. Indem *Berechnen* die Variable *x* beim Aufruf der Prozedur *Test* als Argument übergibt. *Test* nimmt den in *x* übergebenen Wert in einer lokalen Prozedurvariablen entgegen (die ebenfalls *x* heißen kann, ebensogut aber auch *y* oder *z*) und kann nun mit diesem Wert arbeiten.

5. *Function Test (x As Double, y As Double) As String*

6. Übergibt eine Prozedur einer anderen beim Aufruf eine Variable *x* als Argument (*Call Test(x)*), ist das »Lokalitätsprinzip« durch diese Übergabe der Variablen »als Referenz« durchbrochen. Verändert die aufgerufene Prozedur die Variable, in der sie diesen Wert entgegennimmt, wird dadurch der Wert der Variablen *x* der aufrufenden Prozedur in gleicher Weise verändert. Soll das vermieden werden, muß die betreffende Variable statt dessen »als Wert« übergeben werden (*Call Test((x))*). Die Klammerung

schützt *x* vor Veränderungen: die aufgerufene Prozedur kann den Inhalt der übergebenen Variablen zwar in einer lokalen Prozedurvariablen entgegennehmen und benutzen, die »Originalvariable« jedoch nicht verändern!

7. Mit dem Schlüsselwort *Static* (*Static x As Integer*).
8. a) Im Deklarationsbereich des Moduls, wahlweise mit *Dim* oder – klarer – mit *Private* (*Private x As Integer*) bzw. b) im Deklarationsbereich des Moduls, mit *Public* (*Public x As Integer*).
9. *Public Static Function Test (x As Single, y As Single) As String*
10. Indem Sie mit EXTRAS|VERWEISE... einen »Verweis« auf die Mappe einfügen, die die betreffenden Prozeduren enthält, und sie dadurch als zusätzliche »Bibliothek« bekanntgeben (das zugehörige Kontrollkästchen muß aktiviert sein, damit die betreffende Mappe auch wirklich nach den aufgerufenen Prozeduren durchsucht wird!).