

Inhalt

Vorbemerkung

Was ist VBA und wofür ist es gut?

Bestandteile von VBA-Programmen

Was ist ein VBA-Programm?

Benutzerdefinierte Funktionen

Programme mit Prozeduren

Kommentare

Funktionen

Datentypen

Variablen

Der Begriff der Variablen

Deklaration und Gebrauch von Variablen

Der Gültigkeitsbereich einer Variablen

Option Explicit

Operatoren und Ausdrücke

Arithmetische Operatoren und arithmetische Ausdrücke

Funktionsaufrufe in Ausdrücken

Logische Ausdrücke, Vergleichsoperatoren, logische Operatoren

Zusammenfassende Übersicht der Operatorprioritäten

Die Verwendung von Variablen in Ausdrücken

Zuweisungen

Einfache Funktionen verschiedener Datentypen

Rückgriff auf bereits programmiertes

Steueranweisungen

Die Verzweigung

Die Funktion If

Geschachtelte Verzweigungen

Mehrfachauswahl

Die Schleife mit Do While ... Loop

Die Schleife mit For ... Next (Zählschleife)

Exkurs: Rekursion

Datentypumwandlungen

Der Datentyp String (Vertiefung)

- Deklaration von String-Variablen

- Funktionen zur String-Bearbeitung

- Vordefinierte String-Konstanten

- ASCII und der Vergleich von Zeichenketten

- Umwandlung von Zeichen in ihre ASCII-Entsprechung und umgekehrt

- Umwandlung von Zeichenketten in Zahlen

Prozeduren (Sub)

- Aufruf von Funktionen aus Prozeduren

- Ein- und Ausgabe

- ByVal und ByRef

Arrays

- Grundlagen

- Eine Funktion mit einem Array als Parameter

- Eine Funktion, die ein Array liefert

- ReDim Preserve

- Mehrdimensionale Arrays

- Arrays und benutzerdefinierte Funktionen

Benutzerdefinierte Datentypen

Ereignisgesteuerte Programmierung

- Begriff

- Ablauf eines ereignisgesteuerten Programms

- Programmierung von Ereignisprozeduren

Vorbemerkung

Diese VBA-Einführung fasst in kompakter Form die in der Lehrveranstaltung Informatik 1 vermittelten Kenntnisse der Programmiersprache zusammen. Einige zusätzliche Punkte sind ebenfalls enthalten; sie sollten leicht verständlich sein, wenn man das Übrige verstanden hat.

Der Umgang mit der Entwicklungsumgebung (VBE) wird nicht behandelt. Zu Beginn der Programmierausbildung wird in Vorlesung und Übung eine kleine Einführung in die Entwicklungsumgebung gegeben. Die eigentliche Beherrschung kommt dann mit der Arbeit am System (learning by doing).

Grundsätzlich ist der Aufbau von VBA in allen Office-Anwendungen derselbe. Eine Ausnahme bilden die sog. benutzerdefinierten Funktionen (s. unten), die es nur in Verbindung mit Excel gibt.

Was ist VBA und wofür ist es gut?

VBA (Visual Basic for Applications) ist eine Variante der Programmiersprache Visual Basic, welche von Microsoft innerhalb der Office-Anwendungen (Excel, Word, Powerpoint, ...) zur Verfügung gestellt wird. VBA soll dazu benutzt werden,

- Office-Anwendungen um weitere benötigte Funktionalität zu ergänzen und
- um Schritte bei der Arbeit mit Office-Anwendungen zu automatisieren, die sonst manuell durchgeführt werden müssten.

Die professionelle Benutzung von Excel gehört für viele Ingenieure zur täglichen Arbeit. Schon im Praxissemester sind viele Studenten damit befasst, die Funktionalität von Excel mit Hilfe von VBA zu erweitern.

Es bietet sich daher an, die Programmierausbildung von Ingenieuren anhand von Excel-VBA durchzuführen. Allerdings wäre es falsch, daraus eine reine Excel-Schulung zu machen, denn (auch unabhängig von Excel) programmieren zu können ist für alle Ingenieure unabdingbar. Für den Lehrenden ist es deshalb wichtig, dem Unterricht die allgemein anerkannten Prinzipien guter Programmierung zugrunde zu legen, für den Lernenden, sie konsequent anzuwenden.

Bestandteile von VBA-Programmen

Was ist ein VBA-Programm?

Unter einem VBA-Programm wollen wir etwas in VBA Programmiertes verstehen, das man ausführen lassen kann. Die beiden für uns wichtigen Erscheinungsformen werden nun kurz beschrieben.

Benutzerdefinierte Funktionen

Benutzerdefinierte Funktionen sind die kleinste Form von Programmen im Sinne der obigen Definition, und es gibt sie nur in Excel. Wie Sie wissen, ist Excel mit vielen Funktionen ausgestattet, die Sie bei der Arbeit in den Tabellenblättern benutzen können. Wenn Ihnen diese Funktionen nicht ausreichen, können Sie selbst weitere in VBA programmieren. Diese Funktionen stehen dann anschließend im Funktionsassistenten unter der Rubrik „benutzerdefinierte Funktionen“ zur Verfügung und können wie die originären Excel-Funktionen in Formeln benutzt werden.

Um eine solche Funktion zu schreiben, muss man zunächst im VBE ein **Modul** anlegen (wichtig !!!). In dieses Modul kann man dann die gewünschte Funktion (und bei Bedarf weitere) schreiben.

Benutzerdefinierte Funktionen eignen sich besonders für die ersten Schritte in der Programmierung. Man kann sie direkt in den Arbeitsblättern benutzen, braucht also keine weiteren Programmteile, um sie ablaufen zu lassen.

Programme mit Prozeduren

Diese Art von Programmen gibt es in mehreren Unterformen. Gemeinsam haben diese Unterformen, dass der Programmablauf von Prozeduren koordiniert wird. Prozeduren sind Programmeinheiten, welche vom Aufbau her Funktionen ähneln, im Unterschied zu diesen aber keinen Wert liefern. Eine Prozedur kann man starten, d.h. direkt ausführen lassen, während dies für eine Funktion nicht möglich ist. Der Start einer Prozedur kann auf unterschiedliche Weise geschehen, beispielsweise über die Menüoption Ausführen im VBE oder, bei der ereignisorientierten Programmierung, durch das Drücken einer Schaltfläche in einem Formular.

Auch Prozeduren sind in Modulen enthalten. Gewöhnlich packt man Prozeduren und Funktionen, die inhaltlich zusammengehören, in ein Modul.

Wir werden uns im Folgenden zunächst mit benutzerdefinierten Funktionen beschäftigen und erst später Programme mit Prozeduren schreiben (s. Abschnitt Prozeduren).

Kommentare

Kommentare sind Anmerkungen im Programmtext, die der Programmierer für sich selbst und andere Programmierer macht, die ebenfalls das Programm bearbeiten. In VBA wird ein Text zum Kommentar, wenn man ihm ein Apostroph (') voranstellt. Für den Compiler ist ein solcher Apostroph ein Zeichen dafür, dass der folgende Text (bis zum Zeilenende) nicht übersetzt werden soll. Kommentare werden vor allem eingesetzt, um das Programm verständlicher zu machen und um den Stand der Bearbeitung zu dokumentieren, wie das folgende Beispiel zeigt:

```
'Dieses Modul enthält betriebsspezifische Funktionen  
'Autor:      Fritz Findig, OA1  
'Stand:      01.10.2012
```

```
'liefert das Jahr der Betriebsgründung
Public Function Betriebsgruendung() AS Integer
    Betriebsgruendung = 1887
End Function
```

Daneben können Kommentare hilfreich sein, wenn man in der Phase des Testens zeitweilig einen Teil des Codes deaktivieren will, ohne ihn gleich zu löschen. Bequem ist diese Technik allerdings nur, wenn man vorher den VBE mit den Icons zur Kommentierung und Entkommentierung ausgestattet hat (Option Anpassen).

Funktionen

Der Aufbau einer Funktion ist immer gleich, unabhängig davon, ob die Funktion als benutzerdefinierte Funktion verwendet wird oder ob sie in ein größeres Programm eingebettet ist.

```
Public Function xHoch5 (ByVal x As Integer) As Double
    xHoch5 = x^5
End Function
```

Eine Funktion besteht aus einer **Kopfzeile**, einer **Fusszeile (Abschlusszeile)** und einem **Rumpf**, der durch Kopf- und Abschlusszeile begrenzt ist. Die Kopfzeile besteht hier aus:

- Der **Zugriffsspezifikation** Public, welche aussagt, dass die Funktion auch von außerhalb des Moduls aufgerufen werden kann. Die Alternative zu Public ist Private. Funktionen, die mit Private ausgezeichnet sind, können nur innerhalb ihres Moduls verwendet werden, und sie können auch nicht als benutzerdefinierte Funktionen benutzt werden.
- dem Schlüsselwort **Function**,
- dem **Namen** der Funktion (wird vom Programmierer festgelegt),
- in der Klammer: Definition eines **Parameters**, welcher der Funktion bei ihrem Aufruf übergeben wird (entspricht dem Argument der Funktion). Eine Erklärung für den Zusatz ByVal finden Sie weiter unten. Sie müssen vorerst nicht wissen, was er bedeutet.
- der Angabe des **Resultat-Datentyps**.

Der Rumpf der Funktion besteht im Beispiel nur aus einer **Zuweisung** an den Funktionsnamen. Diese Zuweisung bewirkt, dass die Funktion den Wert des zugewiesenen Ausdrucks an die aufrufende Stelle meldet.

Datentypen

Datentypen haben einen doppelten Zweck:

- Sie sorgen für einen ökonomischen Gebrauch des Speichers. Es wird nicht mehr Speicherplatz verbraucht als notwendig.

- Sie verhindern, dass Daten unsachgemäß benutzt werden. Das Programmiersystem kann sicher stellen, dass der Programmierer die Daten nur für den vorhergesehenen Zweck einsetzt.

Die folgende Tabelle zeigt die wichtigsten Datentypen

Typ	Verwendung	Beispielwert
Integer	Ganzzahl	20233
Double	Fließkommazahl	36489.645
String	Zeichenkette	"abc233!!öö"
Date	Datum	#12/24/2011#
Boolean	Logischer Wert	True
Variant	Möglichst nicht verwenden!	

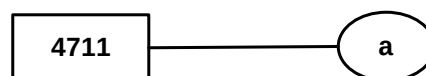
Variant bedeutet, dass der Datentyp nicht festgelegt ist, kennzeichnet also eigentlich die Absenz eines Datentyps.

Variablen

Der Begriff der Variablen

Eine Variable ist ein vom Programmierer vergebener Name, dem ein Wert zugewiesen werden kann. Im Verlauf der Programmausführung kann sich dieser Wert ändern. Mit anderen Worten: er ist variabel.

Programmieranfängern hilft oft eine Analogie. Demnach ist eine Variable ein benanntes Fach im Arbeitsspeicher, in dem ein Wert abgelegt werden kann. Über die Benennung des Fachs (= Name der Variablen) kann das Programm auf den dort abgelegten Wert zugreifen. Das folgende Bild zeigt eine ganzzahlige Variable mit dem Namen a und dem aktuellen Wert 4711



Deklaration und Gebrauch von Variablen

Die folgende, etwas umständlich programmierte Funktion zeigt, wie Variablen angewendet werden:

```

Public Function xHochy (ByVal x As Integer, ByVal y as Integer) As Double
    Dim erg As Double
    erg = x^y
    xHochy = erg
End Function
  
```

In der Funktion werden gleich drei Variablen benutzt: die beiden Parametervariablen x und y, die im Funktionskopf deklariert sind, und eine weitere Variable erg, die im Rumpf der Funktion deklariert

ist. Sie dient hier der Zwischenspeicherung des Ergebnisses der Operation x^y , bevor dieses dem Funktionsnamen zugewiesen wird.

Mit dem Befehl in der ersten Zeile des Rumpfs wird die Variable `erg` definiert („deklariert“). Durch diesen Befehl wird, um beim oben genannten Bild zu bleiben, VBA veranlasst, ein Fach mit der Benennung `erg` zu reservieren, in dem ein Wert vom Typ `Double` abgelegt werden kann.

Tatsächlich abgelegt wird der Wert aber erst mit dem Befehl in der zweiten Zeile. Nach Ausführung dieses Befehls liegt das Ergebnis des Ausdrucks x^y im Fach mit der Benennung `erg`.

Mit dem letzten Befehl des Funktionsrumpfes wird schließlich der in `erg` abgelegte Wert wieder gelesen und dem Funktionsnamen zugewiesen, so dass das Ergebnis an die Stelle geliefert werden kann, von der die Funktion aufgerufen wurde.

Der Gültigkeitsbereich einer Variablen

Wir wollen annehmen, dass in dem Modul, in dem sich die Funktion `xHochy` befindet, noch eine zweite Funktion `maxVon2` enthalten ist, welche die größere von zwei Zahlen liefert. Das Modul hat also folgenden Wortlaut:

```
Public Function xHochy (ByVal x As Integer, ByVal y As Integer) As Double
    Dim erg As Double
    erg = x^y
    xHochy = erg
End Function
```

```
Public Function maxVon2 (ByVal x As Double, ByVal y As Double) As Double
    Dim erg As Double
    erg = 0.5 * (x + y + Abs(x - y)) 'Abs errechnet den Betrag einer Zahl
    maxVon2 = erg
End Function
```

Dass die Variablen in den beiden Funktionen dieselben Namen tragen, ist eine Entscheidung des Programmierers, die beim Programmieranfänger vielleicht Verwirrung stiften kann, im Übrigen aber überhaupt keinen Effekt auf die Arbeit der Funktionen hat. Die Variable `erg` der ersten Funktion hat nämlich überhaupt nichts zu tun mit der gleichnamigen Variablen in der zweiten Funktion. Es handelt sich um zwei verschiedene Variablen, die nur zufällig denselben Namen tragen. Es ist auch nicht zu befürchten, dass VBA aufgrund der Namensgleichheit Probleme hätte, die Variablen zu unterscheiden, denn das `erg` der ersten Funktion ist nur dort bekannt, genau so wie das `erg` der zweiten Funktion auch nur in der zweiten Funktion bekannt ist. Man spricht vom **Gültigkeitsbereich** (engl.: *scope*) der Variablen. Der Gültigkeitsbereich des `erg` der ersten Funktion ist auf diese erste Funktion beschränkt, der Gültigkeitsbereich des `erg` der zweiten Funktion auf die zweite Funktion.

Dasselbe gilt übrigens auch für die Parametervariablen x und y. Auf das x der ersten Funktion kann in der zweiten Funktion nicht zugegriffen werden, weil sich ihr Gültigkeitsbereich auf die erste Funktion beschränkt.

Option Explicit

Bei der Option Explicit handelt es sich um eine Anweisung an den Compiler, bei der Übersetzung bzw. Ausführung eines Programms darauf zu achten, dass alle Variablen ordnungsgemäß deklariert sind. Man sollte diese Anweisung als erste Zeile in jedes Modul setzen. Dies erspart viel Zeit, die man sonst bei der Suche nach schwer auffindbaren Fehlern verbringt. Die folgende (fehlerhafte) Variante der Funktion xHochy soll dies illustrieren:

```
Public Function xHochy (ByVal x As Integer, ByVal y as Integer) As Double
    Dim erg As Double
    erg = x^y
    xHochy = er
End Function
```

Hat der Programmierer die Option Explicit nicht gesetzt, so wird die Funktion vom Compiler ohne Widerrede übersetzt. Nichtsdestotrotz kann der Programmierer nicht zufrieden sein, denn die Funktion liefert immer den Wert 0, gleichgültig, wie hoch die Werte für x und y sind.

Wo liegt der Fehler? Der Programmierer hat sich verschrieben. In der dritten Zeile hat er dem Funktionsnamen den Wert einer Variablen er zugewiesen. Es hätte erg heißen müssen! Weil aber die Option Explicit nicht gesetzt war, hat der Compiler er als zusätzliche, nicht explizit deklarierte Variable interpretiert. Er hat diese Variable mit dem Standardwert 0 vorbesetzt, was das konstante Ergebnis der Funktion erklärt.

Hätte der Programmierer die Option Explicit gesetzt, so hätte der Compiler den Übersetzungslauf mit dem Hinweis abgebrochen, dass das Programm eine nicht deklarierte Variable er enthält. Der Programmierer hätte damit auch schon seinen Fehler gefunden gehabt.

Abschließende Bemerkung zu den beiden Funktionen

Die Hilfsvariablen erg in den beiden Funktionen sind entbehrlich. Sie waren nur für die Erläuterung des Gültigkeitsbereichs wichtig. Eine kürzere Version ist:

```
Public Function xHochy (ByVal x As Integer, ByVal y as Integer) As Double
    xHochy = x^y
End Function
```

```
Public Function maxVon2 (ByVal x As Double, ByVal y As Double) As Double
    maxVon2 = 0.5 * (x + y + Abs(x - y))
End Function
```

Operatoren und Ausdrücke

Arithmetische Operatoren und arithmetische Ausdrücke

Häufig müssen in Funktionen oder größeren Programmen aus numerischen Werten andere berechnet werden. Hierzu stehen die Operatoren zur Verfügung, die in der folgenden Tabelle aufgeführt sind:

-	Negation
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
^	Potenzierung
\	Ganzzahliges Ergebnis der Division
Mod	Divisionsrest

Für die Negation und die Subtraktion wird, wie wir dies aus der Mathematik kennen, dasselbe Symbol benutzt (-). Dies ist unproblematisch, weil VBA im konkreten Fall stets feststellen kann, welcher der beiden Operatoren gemeint ist.

Verknüpft man numerische Werte durch Operatoren aus der Tabelle, so entsteht ein **arithmetischer Ausdruck**. Ein solcher Ausdruck besitzt einen Wert (ein Ergebnis). Die verknüpften Werte nennen wir **Operanden** des Ausdrucks.

Die Mehrzahl der Operatoren ist für Werte aller numerischen Datentypen definiert. Lediglich für die letzten beiden der obigen Tabelle gilt dies nicht; sie sind nur für Ganzzahlen (Datentyp Integer, Long, ...) möglich. Die folgende Tabelle zeigt links zulässige Ausdrücke und rechts das dazu gehörige Ergebnis:

	Ausdruck	Ergebnis
1	3 + 2	5
2	3.1 + 2	5.1
3	3.0 + 2	5.0
4	6 / 3	2.0
5	2^4	16.0
6	6 \ 4	1
7	6 Mod 4	2

Beachten Sie, dass die rechte Spalte nicht nur die Höhe des Ergebnisses zeigt, sondern auch dessen Datentyp (Integer oder Double). Bei Addition, Subtraktion und Multiplikation hängt der Datentyp vom Datentyp der Operanden ab. Sind beide Operanden vom Typ Integer, wie in Zeile 1, so ist auch das Ergebnis vom Typ Integer. Wenn aber einer der Operanden oder beide vom Typ Double sind, so ist auch das Ergebnis vom Typ Double (Zeile 2). Dabei kommt es nicht auf den Wert der Operanden an, sondern nur auf ihren Datentyp, wie Zeile 3 zeigt. Obwohl dort beide Operanden ganzzahlige Werte haben, ist das Ergebnis nicht vom Typ Integer.

Bei den Operatoren / und ^ ist der Datentyp des Ergebnisses leicht zu bestimmen: es ist immer vom Typ Double, unabhängig vom Typ der Operanden. Ähnlich einfach gestaltet sich die Lage bei den Operatoren \ und Mod. Das Ergebnis ist stets vom Typ Integer.

Verknüpft man in einem Ausdruck mehr als zwei Operanden, und kommen dabei verschiedene Operatoren zum Einsatz, so muss man die **Vorrangregeln** beachten. Für die Operatoren der obigen Tabelle gilt dabei folgende Reihenfolge:

^
 - (Negation)
 *, /
 \
 Mod
 +, -

Will man in einem konkreten Ausdruck, dass er nicht im Sinne dieser Vorrangregeln interpretiert werden soll, so kann man Klammern setzen. Die folgende Tabelle zeigt einige Ausdrücke und ihr Ergebnis:

	Ausdruck	Ergebnis
1	3 + 2 * 2	7
2	(3 + 2) * 2	10
3	- 2^2	-4.0
4	(-2)^2	4.0
5	2^4	16.0
6	12 \ 3 Mod 2	0
7	12 \ (3 Mod 2)	12

Die Verwendung von Funktionsaufrufen in Ausdrücken

Zusätzlich zu den dargestellten Operatoren gibt es in VBA noch eine große Anzahl „eingebauter“ Funktionen, die der Programmierer in arithmetischen Ausdrücken verwenden kann. Wir wollen hier beispielhaft die Funktion Abs betrachten. Abs liefert den absoluten Wert (den Betrag) einer Zahl. Mit anderen Worten, Abs entfernt ein eventuell vorhandenes Vorzeichen. Der numerische Ausdruck

Abs (3 – 5)

hat demnach den Wert 2.

Weitere eingebaute Funktionen finden Sie leicht im VBA-Sprachverzeichnis. Genau so wie die eingebauten Funktionen kann man aber auch Aufrufe selbst programmierter Funktionen in Ausdrücken verwenden. Wir kommen darauf später zurück.

Logische Ausdrücke, Vergleichsoperatoren, logische Operatoren

Logische Ausdrücke sind Ausdrücke, die einen Wahrheitswert haben. Anders ausgedrückt: ein logischer Ausdruck repräsentiert eine Aussage, von der man sagen kann, dass sie wahr oder falsch ist.

Logische Ausdrücke spielen eine wichtige Rolle in Verbindung mit Verzweigungen und Schleifen (s. weiter unten). Generell können logische Ausdrücke auch all die Operatoren enthalten, welchen in numerischen Ausdrücken vorkommen. Darüber hinaus kommen darin aber noch zwei weitere Arten von Operatoren vor: Vergleichsoperatoren und logische Operatoren.

Häufig beruhen logische Ausdrücke auf **Vergleichsoperatoren**. Beispielsweise hat der Ausdruck

$$3 > 5$$

den Wahrheitswert False, weil die Aussage offensichtlich falsch ist. Demgegenüber hat der Ausdruck

$$2 - 2 = 0$$

den Wahrheitswert True, weil die Aussage zweifelsohne richtig ist. Die folgende Tabelle zeigt die Vergleichsoperatoren, die der Programmierer in VBA zur Verfügung hat:

<	kleiner
<=	kleiner oder gleich
>	größer
>=	größer oder gleich
=	gleich
<>	ungleich

Mit Hilfe von **logischen Operatoren** kann man aus logischen Ausdrücken größere, **zusammengesetzte** logische Ausdrücke formen. Verbinden wir z.B. die beiden obigen Ausdrücke mit Hilfe des Operators AND, so erhalten wir

$$3 > 5 \text{ AND } 2 - 2 = 0$$

Dieser Ausdruck hat den Wahrheitswert False, weil der Operator AND nur dann True ergibt, wenn beide verbundenen Einzelaussagen wahr (True) sind. Die folgende Tabelle zeigt die in VBA verfügbaren logischen Operatoren:

NOT	Verneinung
AND	Und
OR	Oder
XOR	ausschließendes Oder

Wie bei den arithmetischen Operatoren auch, sind bei den logischen Operatoren Prioritätsregeln zu beachten. Es gilt

NOT vor AND vor OR, XOR

Zusammenfassende Übersicht der Operatorprioritäten

Da in logischen Ausdrücken viele verschiedene Operatoren vorkommen können, müssen die Prioritäten aller möglichen Operatoren geregelt sein. Die folgende Tabelle zeigt die gängigen

Operatoren mit von oben nach unten abnehmender Priorität. Die Tabelle gilt sowohl für arithmetische als auch für logische Ausdrücke:

$\wedge$	Potenzierung
-	Negation
*, /	Multiplikation, Division
\	Ganzzahlige Division
Mod	Divisionsrest
+, -	Addition, Subtraktion
&	Stringverkettung
=, <, <=, >, >=, <>	Vergleichsoperatoren
Not	Verneinung
And	Und
Or, Xor	Oder, ausschließendes Oder

Verwendung von Variablen in Ausdrücken

Ausdrücke wie die bisher in diesem Abschnitt vorgestellten, in denen alle Operanden bekannte Werte sind, kommen in realen Programmen selten vor. Viel häufiger sind Ausdrücke, in denen anstelle bekannter Werte Variablen verwendet werden. Dies ist möglich, denn Variable repräsentieren die Werte, die ihnen zugewiesen wurden. Wir betrachten hierzu noch einmal die Funktion maxVon2 in ihrer kürzeren Fassung:

```
Public Function maxVon2 (ByVal x As Double, ByVal y As Double) As Double
    maxVon2 = 0.5 * (x + y + Abs(x - y))
End Function
```

Die einzige Zeile im Rumpf dieser Funktion enthält auf der rechten Seite einen Ausdruck, in dem zwei Variablen und der Betrag der Differenz der beiden Variablen summiert werden.

Zuweisungen

Mit Hilfe der Zuweisung wird einer Variablen oder einem Funktionsnamen ein Wert zugewiesen. Das Symbol für die Zuweisung ist „=“. In dem folgenden Codeabschnitt wird zuerst eine Variable a deklariert, dann wird dieser Variablen ein Wert zugewiesen:

```
Dim a As Integer
a = 12
```

Auf der rechten Seite einer Zuweisung steht immer ein Ausdruck oder, wie im obigen Beispiel, ein einzelner Wert. Der Datentyp der Variablen auf der linken Seite sollte mit dem Datentyp des Werts bzw. Ausdrucks auf der rechten Seite harmonisieren.

Beachten Sie, dass die Zuweisung **immer von rechts nach links geschieht**. Die Nichtbeachtung dieser Regel ist ein häufiger Fehler bei Programmieranfängern. Der folgende Zuweisungsversuch funktioniert also nicht:

```
Dim a As Integer
12 = a                'fehlerhafte Zuweisung!!!
```

Nicht ganz unschuldig an den Problemen sind die Schöpfer von VBA, denn sie haben für die Zuweisung dasselbe Symbol vorgesehen wie für den Vergleichsoperator. Ein und dasselbe Symbol bezeichnet also ganz verschiedene Dinge. Bei richtig geschriebenen Programmen führt dies nicht zu Fehlern, da für VBA stets aus dem Kontext ersichtlich ist, welche der beiden Bedeutungen des Symbols gerade gemeint ist. Dem Lernenden, der anfangs ohnehin Schwierigkeiten hat, den Unterschied zwischen Zuweisung und Vergleichsoperator zu erfassen, ist dies nur ein schwacher Trost. Andere Programmiersprachen sind in dieser Hinsicht besser konzipiert.

Wir betrachten nun einige Beispiele, in denen das Symbol „=" sowohl für die Zuweisung als auch für die Prüfung auf Gleichheit verwendet wird, um den Unterschied herauszuarbeiten. Im ersten Beispiel wird einer Variablen b vom Typ Boolean das Ergebnis des logischen Ausdrucks $3 = 4$ zugewiesen. Das erste „=" ist demnach eine Zuweisung, das zweite ein Gleichheitsoperator, also eine Prüfung auf Gleichheit. Da 3 und 4 nicht gleich sind, hat der logische Ausdruck den Wert False. Nach Ausführung dieser Codezeile hat b demnach den Wert False.

```
Dim b As Boolean
b = (3 = 4)
```

Die Klammer im Beispiel gibt dem Leser eine Interpretationshilfe; sie trennt den logischen Ausdruck visuell von der Zuweisung. Nötig ist sie aber nicht, denn die Zuweisung hat geringere Priorität als der Gleichheitsoperator (und alle anderen Operatoren). Die folgende Formulierung hat daher denselben Effekt:

```
Dim b As Boolean
b = 3 = 4
```

Wir erweitern nun das obige Beispiel, um es noch etwas komplizierter zu machen:

```
Dim b As Boolean
b = 3 = 4
Dim c As Boolean
c = (b = (5 = 5))
```

Wie schon oben ausgeführt, hat nach Durchführung der zweiten Zeile die Variable b den Wert False. In der dritten Zeile wird eine zweite Variable c deklariert, der dann in der vierten Zeile ein Wert zugewiesen wird. Welcher Wert ist das? Der Text rechts vom Zuweisungszeichen ist ein logischer Ausdruck. Darin wird der Wert der Variablen b mit dem Wert des logischen Ausdrucks $5 = 5$ verglichen. Da $5 = 5$ offensichtlich wahr (True) ist, b aber den Wert False hat, reduziert sich der gesamte logische Ausdruck rechts des Zuweisungszeichens zu $\text{False} = \text{True}$, was zweifelsohne falsch (False) ist. Der Variablen c wird demnach der Wert False zugewiesen.

Neben der Zuweisung von Werten an Variablen hat die Zuweisung in VBA noch einen zweiten Anwendungsbereich: die **Zuweisung des Funktionsergebnisses an den Funktionsnamen**. Diese Zuweisung bewirkt, dass der Stelle, welche die Funktion aufgerufen hat, das Funktionsergebnis geliefert wird. Wir betrachten noch einmal die Funktion maxVon2:

```
Public Function maxVon2 (ByVal x As Double, ByVal y As Double) As Double
    maxVon2 = 0.5 * (x + y + Abs(x - y))
End Function
```

In längeren Funktionen, wo sich die Ermittlung des Funktionsergebnisses über mehr als eine Zeile erstreckt, erfolgt die Zuweisung an den Funktionsnamen gewöhnlich in der letzten Zeile des Funktionsrumpfs.

Einfache Funktionen verschiedener Datentypen

Wir betrachten nun einige Beispiele einfacher Funktionen verschiedener Datentypen. Diese Funktionen haben alle nur einen Rumpf von einer Zeile. Wir können aber an ihnen gut die Verwendung von Variablen, die Bildung von Ausdrücken und die Zuweisung an den Funktionsnamen studieren. Alle Funktionen, die wir betrachten, sind auch als benutzerdefinierte Funktionen aus einem Tabellenblatt aufrufbar.

Die beiden folgenden Funktionen sind vom Typ **Double**, liefern also einen Double-Wert. Während bei der ersten Funktion der Mehrwertsteuersatz fest in den Programmtext eingearbeitet ist, wird bei der zweiten Funktion der Satz als Parameterwert entgegengenommen. Die zweite Funktion kann daher mit unterschiedlichen Steuersätzen arbeiten, während dies bei der ersten nicht der Fall ist.

```
Public Function brutto(ByVal netto As Double) As Double
    brutto = netto * 1.19
End Function
```

```
Public Function brutto(ByVal netto As Double, ByVal mwst As Double) As Double
    brutto = netto * (1 + mwst/100)
End Function
```

Die beiden Beispiele illustrieren ein häufiges Dilemma, dem Programmierer ausgesetzt sind: programmiert man eine Funktion so, dass sie nur für einen speziellen Fall einsetzbar ist, so ist sie relativ einfach aufgebaut. Schreibt man die Funktion aber so, dass sie möglichst flexibel eingesetzt werden kann, so hat sie einen komplizierteren Aufbau.

Die nächste Funktion ist vom Typ Boolean, liefert also einen Wahrheitswert. Es handelt sich um die logische Operation Nand, die im Excel-Funktionsassistenten nicht aufgeführt ist. Wir realisieren sie hier, indem wir sie auf die logischen Basisfunktionen Not und And zurückführen. Bekanntlich kann man alle möglichen logischen Funktionen auf die Grundfunktionen Not, And und Or zurückführen.

```
Public Function Nand(ByVal A As Boolean, ByVal B As Boolean) As Boolean
    Nand = Not (A And B)
End Function
```

Wir betrachten nun eine Funktion näher, die schon weiter oben einmal aufgetaucht ist. Neben der Tatsache, dass diese Funktion vom Typ **Integer** ist, hat sie noch eine weitere Besonderheit: sie benötigt keinen Parameter. Deshalb ist die Parameter-Klammer in der Kopfzeile leer.

```
Public Function Betriebsgruendung() AS Integer
    Betriebsgruendung = 1887
End Function
```

Offensichtlich liefert die Funktion bei jedem Aufruf denselben Wert (1887). Man kann Funktionen in dieser Weise gebrauchen, um häufig benötigte Konstanten bequem abrufbar zu machen. Die folgende Funktion zeigt dieselbe Technik anhand einer Funktion des Typs **Date**. Beachten Sie die Schreibweise des Datumswerts (31. Dezember 1847).

```
Public Function meinGeburtstag() As Date
    meinGeburtstag = #12/31/1847#
End Function
```

Damit der Funktionsaufruf aus dem Tabellenblatt richtig funktioniert, müssen Sie der Zelle, in der sich die Formel mit dem Funktionsaufruf befindet, das Format Datum zuweisen. Tun Sie das nicht, so wird anstelle eines Datums eine ganze Zahl ausgegeben. Dasselbe gilt auch für die folgende Funktion:

```
Public Function thedayafter() As Date
    thedayafter = #12/31/1847# + 1
End Function
```

Zum Schluss betrachten wir noch eine Funktion, welche einen **String** (eine Zeichenkette) liefert. Die Funktion nimmt zwei Parameter entgegen, von denen wir annehmen wollen, dass sie die Namen von Personen sind. In dem Ausdruck auf der rechten Seite des Zuweisungsoperators wird zweimal der Verkettungsoperator & verwendet. Für den Fall, dass p1 gleich „Franz“ ist und p2 gleich „Zensi“, so hat der ganze Ausdruck den Wert „Franz liebt Zensi“.

```
Public Function Liebe(ByVal p1 As String, ByVal p2 As String) As String
    Liebe = p1 & " liebt " & p2
End Function
```

Rückgriff auf bereits Programmiertes

In den Beispielen einfacher Funktionen des vorangegangenen Abschnitts bestand der Rumpf der Funktion jeweils aus einer Zuweisung, in der dem Funktionsnamen das Ergebnis eines Ausdrucks zugewiesen wurde. Bereits bei der Behandlung der Ausdrücke weiter oben wurde erwähnt, dass Ausdrücke auch Funktionsaufrufe enthalten können.

Solche Funktionsaufrufe beschränken sich nicht auf die eingebauten VBA-Funktionen, wie z.B. Abs. Es ist möglich, auch Aufrufe selbst programmierter Funktionen in Ausdrücken zu verwenden. Um dies zu zeigen, greifen wir auf eine bereits weiter oben betrachtete Funktion zurück, welche die größere von zwei Zahlen ermittelt:

```
Public Function maxVon2 (ByVal x As Double, ByVal y As Double) As Double
    maxVon2 = 0.5 * (x + y + Abs(x - y))
End Function
```

Wir können nämlich auf der Basis dieser Funktion sehr leicht eine Funktion maxVon3 formulieren, welche das Maximum dreier Zahlen liefert. Wir wollen dabei annehmen, dass sich maxVon2 in demselben Modul befindet wie maxVon3:

```
Public Function maxVon3 (ByVal x As Double, ByVal y As Double, _
                        ByVal z As Double) As Double
    maxVon3 = maxVon2(maxVon2(x, y), z)
End Function
```

Eine Funktion, welche das Maximum von vier Zahlen ermittelt, ist ebenso leicht zu realisieren:

```
Public Function maxVon4 (ByVal x As Double, ByVal y As Double, _
                        ByVal z As Double, ByVal v As Double) As Double
    maxVon4 = maxVon2(maxVon2(x, y), maxVon2(z, v))
End Function
```

Dies sind natürlich Beispiele, bei denen die Arbeitersparnis durch Verwendung bereits programmierter Funktionen nicht bedeutend ist. Richtig lohnend wird diese Technik, wenn es darum geht, umfangreiche Berechnungen anzustellen. In dem Maß, in dem Ihre Programmierfertigkeiten sich weiter entwickeln und Sie größere Aufgaben angehen können, werden Sie auch von dieser Technik profitieren können.

Steueranweisungen

Die Verzweigung

Verzweigungen werden benötigt, wenn der Fortgang der Verarbeitung vom Vorliegen bestimmter Bedingungen abhängig ist. Weil Verzweigungen in fast allen Programmiersprachen mit dem Schlüsselwort „If“ eingeleitet werden, spricht man auch häufig von der „If-Anweisung“. Wir unterscheiden zwei Varianten davon: die vollständige Verzweigung und die unvollständige Verzweigung. Wir beginnen mit der **vollständigen Verzweigung**. Die folgende Funktion ist eine alternative Lösung für das Problem, aus zwei Zahlen die größere herauszusuchen:

```
Public Function maxVon2(ByVal z1 As Double, ByVal z2 As Double) As Double
    If z1 > z2 Then
        maxVon2 = z1
    Else
```

```
        maxVon2 = z2
    End If
End Function
```

Der Sinn der If-Anweisung ist hier nicht schwer zu verstehen: Wenn z1 größer als z2 ist, so soll die Funktion z1 liefern, ansonsten z2. Beachten Sie, dass die Anweisung sich im vorliegenden Fall über fünf Zeilen hinzieht; die Schlüsselwörter If und End If wirken, bildlich gesprochen, wie eine Klammer um die gesamte Anweisung. Es gibt noch eine zweite Schreibweise, die man verwenden kann, wenn die If-Anweisung wenig Text enthält, so dass man alles bequem in einer Zeile unterbringen kann:

```
Public Function maxVon2(ByVal z1 As Double, ByVal z2 As Double) As Double
    If z1 > z2 Then maxVon2 = z1 Else maxVon2 = z2
End Function
```

In dieser Version ist das End If nicht notwendig, denn es ist auch so klar, wo die Anweisung endet, nämlich am Ende der Zeile.

Wenn der Zweck des Programms es nicht erfordert, kann man den Else-Teil der Verzweigung auch weglassen. Man spricht dann von einer **unvollständigen Verzweigung**. Die folgende Funktion ermittelt den Betrag einer ganzen Zahl:

```
Public Function Betrag (ByVal z As Integer) As Integer
    If z < 0 Then
        z = -z
    End If
    Betrag = z
End Function
```

Bemerkenswert in dieser Funktion ist auch die Zeile `z = -z`. Wie Sie denken können, steht das Symbol „`=`“ hier nicht für die Gleichheitsprüfung, sondern für die Zuweisung. Verbal lässt sich der Inhalt der Anweisung so ausdrücken: der neue Wert von z ergibt sich aus dem alten Wert von z, wenn man diesen negiert.

Auch die unvollständige Verzweigung kann man in eine einzige Zeile schreiben, wenn der Text hinreichend kurz ist. Hier ist dies der Fall, so dass wir die Funktion auch kompakter wie folgt schreiben können:

```
Public Function Betrag (ByVal z As Integer) As Integer
    If z < 0 Then z = -z
    Betrag = z
End Function
```

In vielen Fällen kann man eine Lösung, die auf einer unvollständigen Verzweigung basiert, in eine Lösung umwandeln, die auf einer vollständigen Verzweigung beruht, und umgekehrt. Dies zeigt die folgende Version der Funktion Betrag:

```
Public Function Betrag (ByVal z As Integer) As Integer
    If z < 0 Then Betrag = -z Else Betrag = z
End Function
```

Die Funktion IIf

Sie entspricht der aus Excel bekannten Wenn-Funktion und ist immer dann anwendbar, wenn der Zweck einer Verzweigung nur daraus besteht, einen einzigen Wert zu ermitteln.

Die größere von zwei Zahlen lässt sich z.B. wie folgt ermitteln:

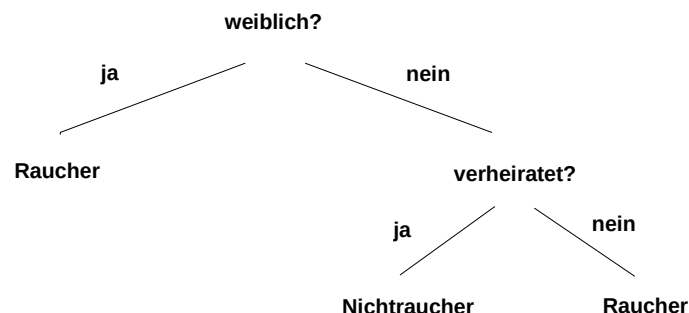
```
Public Function maxVon2(ByVal z1 As Double, ByVal z2 As Double) As Double
    maxVon2 = IIf(z1 > z2, z1, z2)
End Function
```

Auch die Funktion, welche den Betrag liefert, lässt sich damit sehr kurz formulieren:

```
Public Function Betrag (ByVal z As Integer) As Integer
    Betrag = IIf(z < 0, -z, z)
End Function
```

Geschachtelte Verzweigungen

Mehrstufige Entscheidungen können mit geschachtelten Verzweigungen realisiert werden. Das unten stehende Bild zeigt das Ergebnis einer Data-Mining-Studie. In einer Lernphase wurden zunächst die Kriterien ermittelt, die es ermöglichen, vorherzusagen, ob eine Person Raucher ist oder nicht. Das Ergebnis dieser Lernphase wird durch den abgebildeten **Entscheidungsbaum** repräsentiert. Dieser Entscheidungsbaum kann nun in der Anwendungsphase dazu verwendet werden, für konkrete Personen zu prognostizieren, ob sie Raucher sind.



Die folgende Funktion liefert True, wenn aufgrund der übermittelten Eigenschaften zu vermuten ist, dass es sich um einen Raucher handelt, sonst False:

```
Public Function Raucher(ByVal weiblich As Boolean, ByVal verh As Boolean) _
    As Boolean
    If weiblich Then
```

```

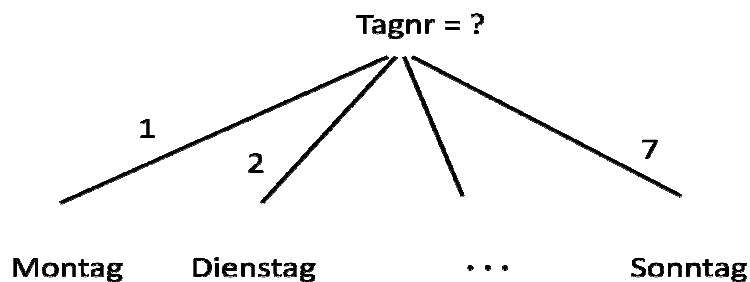
        Raucher = True
Else
    If verh Then
        Raucher = False
    Else
        Raucher = True
    End If
End If
End Function

```

Man kann leicht erkennen, dass der Aufbau der geschachtelten Verzweigung in der Funktion genau dem Aufbau des Entscheidungsbaums entspricht. Das Beispiel zeigt aber auch, wie wichtig korrektes Einrücken bei der Programmierung ist. Nur so lassen sich Entscheidungsbaum und Code gut vergleichen.

Mehrfachauswahl

Von einer Mehrfachauswahl spricht man, wenn es in Abhängigkeit vom Wert einer Variablen mehr als zwei Möglichkeiten zur Fortsetzung gibt, jeweils aber nur eine dieser Möglichkeiten zum Zuge kommen kann. Ein nichtbinärer Entscheidungsbaum zeigt das Prinzip einer solchen Mehrfachauswahl:



In VBA gibt es für die Mehrfachauswahl zwei Lösungsansätze. Der erste ist eine Variante der bereits bekannten If-Anweisung:

```

If ... Then
    ...
ElseIf ...Then
    ...
End If.

```

Die folgende Funktion liefert den ausgeschriebenen Wochentag für die übergebene Tagesnummer. Anders als bei der gewöhnlichen geschachtelten If-Anweisung verwendet man hier keine verschieden starken Einrückungen, weil nicht tatsächlich eine mehrstufige Entscheidung vorliegt.

```

Public Function wochentg(ByVal tagnr As Integer) As String
    If tagnr = 1 Then
        wochentg = "Montag"
    ElseIf tagnr = 2 Then
        wochentg = "Dienstag"

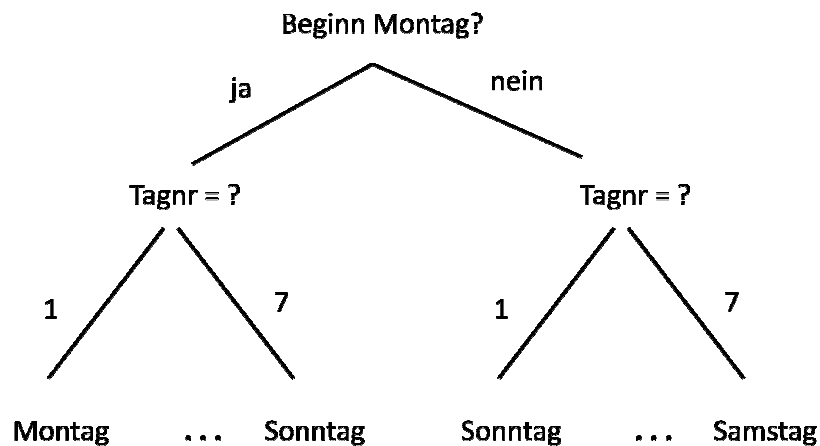
```

```
ElseIf tagnr = 3 Then
    wochentg = "Mittwoch"
ElseIf tagnr = 4 Then
    wochentg = "Donnerstag"
ElseIf tagnr = 5 Then
    wochentg = "Freitag"
ElseIf tagnr = 6 Then
    wochentg = "Samstag"
ElseIf tagnr = 7 Then
    wochentg = "Sonntag"
Else
    wochentg = "Fehler"
End If
End Function
```

Verwendet man die Anweisung **Select Case**, so ergibt sich folgende Lösung:

```
Public Function wochentgSC(ByVal tagnr As Integer) As String
    Select Case tagnr
        Case 1
            wochentgSC = "Montag"
        Case 2
            wochentgSC = "Dienstag"
        Case 3
            wochentgSC = "Mittwoch"
        Case 4
            wochentgSC = "Donnerstag"
        Case 5
            wochentgSC = "Freitag"
        Case 6
            wochentgSC = "Samstag"
        Case 7
            wochentgSC = "Sonntag"
        Case Else
            wochentgSC = "Fehler"
    End Select
End Function
```

Mehrfachauswahl und normale Verzweigung können auch in Kombination auftreten. Das folgende Bild beruht auf der Tatsache, dass es hinsichtlich der Nummerierung der Wochentage unterschiedliche Konventionen gibt. Teilweise wird die Tagnummer 1 mit dem Montag verbunden, z.T. aber auch mit dem Sonntag. Wenn man eine Funktion schreiben will, welche mit beiden Konventionen arbeiten kann, muss man einen zweiten Parameter zur Verfügung stellen, der angibt, welcher Tag der Beginnstag ist. Erst mit der Kenntnis des Beginntags kann der übermittelten Tagnummer eindeutig ein Wochentag zugeordnet werden.



Die Kopfzeile der Funktion könnte lauten:

```
Public Function wochentg(ByVal begMontag As Boolean, ByVal tagnr As Integer) _
    As String
```

Das Programmieren der Funktion sei Ihnen als Übung überlassen. Achten Sie dabei unbedingt auf korrekte Einrückungen.

Die Schleife mit Do While ... Loop

Schleifen dienen der Programmierung von Wiederholungen. In VBA gibt es eine Vielzahl von verschiedenen Schleifentypen. Wir werden uns auf zwei davon beschränken, die Schleife mit Do While ... Loop und die Zählschleife mit For ... Next. Jede Schleife in VBA besteht aus einer **Kopfzeile**, einem **Rumpf** und einer **Fußzeile**.

Die Schleife mit Do While ... Loop ist eine sog. **kopfgesteuerte** oder **abweisende** Schleife. Jedes Mal, wenn das Programm an den Beginn der Schleife gelangt, wird geprüft, ob die Bedingung für ihre Durchführung noch gegeben ist. Besonders abweisend zeigt sich die Schleife, wenn die Durchführungsbedingung von Anfang an nicht gegeben ist, denn dann wird sie kein einziges Mal durchlaufen.

Die folgende Funktion berechnet die Summe der ganzen Zahlen von 1 bis n, wobei n der an die Funktion übergebene Parameter ist.

```
Public Function Summe1BisN(ByVal n As Integer) As Integer
    Dim summe As Integer
    summe = 0
    Dim i As Integer
    i = 1
    Do While i <= n
        summe = summe + i
        i = i + 1
    Loop
```

```
Summe1BisN = summe
End Function
```

Die Variable *summe* ist eine Hilfsvariable, die dem Aufsummieren der Zahlen dient. Sie wird gleich nach der Deklaration mit 0 vorbesetzt („initialisiert“). Anschließend werden mit Hilfe einer Schleife die ganzen Zahlen im gewünschten Bereich addiert. Bei jedem Durchlauf der Schleife wird der Variablen *summe* eine ganze Zahl hinzugefügt. Dabei hat die Variable *i* eine wichtige Funktion. Sie sorgt zum Einen dafür, dass die Schleife so lange wie nötig durchlaufen wird, nämlich solange $i \leq n$. Zum Anderen entspricht sie in jedem Schleifendurchlauf der Zahl, die zu der Summe addiert werden soll. Hierzu muss *i* am Ende jedes Schleifendurchlaufs fortgeschrieben, d.h. um 1 erhöht werden.

Die folgende Tabelle illustriert für $n = 4$, wie sich die Variablen mit den Schleifendurchläufen verändern. Jede Zeile entspricht einem Durchlauf der Schleife. Die Tabelle zeigt, dass die Schleife insgesamt viermal durchlaufen wird. Sie wird kein fünftes Mal betreten, weil mit $i = 5$ die Schleifenbedingung $i \leq n$ nicht mehr gegeben ist. Das Ergebnis der Summierung (10) entspricht dem Wert der Variablen *summe* nach dem letzten Schleifendurchlauf.

Durchlauf Nr	i am Anfang	summe am Anfang	summe am Ende	i am Ende
1	1	0	1	2
2	2	1	3	3
3	3	3	6	4
4	4	6	10	5

Die Schleife mit For ... Next (Zählschleife)

Die Schleife mit For ... Next kann nicht mehr als die Schleife mit Do While ... Loop, ist jedoch in vielen Fällen einfacher zu benutzen.

Man kann sie verwenden, wenn die Anzahl der Schleifendurchläufe schon bekannt ist, wenn die Programmdurchführung die Schleife erreicht. Bei der Zählschleife erübrigt sich die explizite Fortschreibung der Schleifenvariablen; sie erfolgt automatisch am Ende eines jeden Schleifendurchlaufs.

Das folgende Beispiel demonstriert den Standardfall, bei dem die Schleifenvariable nach jedem Durchlauf automatisch um 1 erhöht wird.

```
Public Function Summe1BisN(ByVal n As Integer) As Integer
    Dim summe As Integer
    summe = 0
    Dim i As Integer
    For i = 1 To n
        summe = summe + i
    Next i
    Summe1BisN = summe
End Function
```

Benötigt man eine andere Fortschreibung, so benutzt man die Erweiterung mit Step. Auf diese Weise könnten wir die Summierung auch in umgekehrter Richtung durchführen:

```
Public Function Summe1BisN(ByVal n As Integer) As Integer
    Dim summe As Integer
    summe = 0
    Dim i As Integer
    For i = n To 1 Step -1
        summe = summe + i
    Next i
    Summe1BisN = summe
End Function
```

Exkurs: Rekursion

Das Wort Rekursion ist abgeleitet vom lateinischen recurrere (deutsch: zurücklaufen) und bedeutet in der Programmierung, dass eine Programmeinheit (hier eine Funktion) sich selbst aufruft. Die Rekursion bewirkt eine Wiederholung der Funktion, ohne dass hierzu eine Schleife benötigt würde. Die rekursive Programmierung ist damit eine Alternative zur Programmierung mit Schleifen, der sog. iterativen Programmierung.

Die folgende Funktion führt eine rekursive Berechnung der Summe der ganzen Zahlen von 1 bis n durch. Wie das Beispiel zeigt, sind induktive Formulierungen von Funktionen meist sehr kompakt:

```
Public Function su1n (ByVal n As Integer) As Integer
    If n = 1 Then su1n = 1 Else su1n = n + su1n(n - 1)
End Function
```

Da die Rekursion nicht zum Inhalt des Kurses gehört, belassen wir es hier bei diesem Beispiel. Für den etwas fortgeschrittenen Studenten lohnt es sich aber in jedem Fall, sich näher damit zu beschäftigen, denn die Rekursion ist eine fundamentale Technik der Informatik und der Mathematik.

Datentypumwandlungen (Datentypkonversionen)

Mit Datentypen muss man exakt umgehen

Geht man als Programmierer mit Datentypen schludrig um, so kann dies böse Folgen haben. Zwar versucht der Compiler, die Nachlässigkeiten der Programmierer abzufangen, aber dies gelingt häufig nicht. Das Programm wird dann mit entweder mit einer Fehlermeldung abgebrochen oder, noch schlimmer, es wird ein falsches Ergebnis ermittelt. Wir betrachten hierzu einige Beispiele:

Beispiel 1:

Einer Variablen vom Typ Integer wird das Ergebnis des Aufrufs einer Funktion zugewiesen, welche einen Double-Wert liefert:

```
Dim z As Integer  
z = FunktionDieDoubleLiefert(einParameter)
```

Angenommen, der Funktionsaufruf ergibt den Wert 135.789. Da die Nachkommastellen bei der Zuweisung an die Integer-Variable nicht mitgegeben werden können, wird VBA den Wert auf 136 runden, also stillschweigend eine Typumwandlung durchführen, und den gerundeten Wert dann der Variablen zuweisen. Es tritt also ein Informationsverlust auf.

Beispiel 2:

Manche Operatoren sind für mehrere Datentypen definiert. Dieser Umstand kann in Verbindung mit Nachlässigkeiten im Umgang mit den Datentypen zu schweren Fehlern führen. Wir betrachten hierzu zwei Codeausschnitte, in denen der Vergleichsoperator > verwendet wird.

```
Dim a As Integer, b As Integer  
Dim a_groesser_b as Boolean  
a = 100  
b = 99  
a_groesser_b = Iif(a > b, True, False)
```

Nachdem die letzte Zeile ausgeführt ist, hat die Variable a_groesser_b den Wert True, was sicherlich den Erwartungen entspricht. Wir betrachten nun noch eine zweite Variante:

```
Dim a As String, b As String  
Dim a_groesser_b as Boolean  
a = 100  
b = 99  
a_groesser_b = IIf(a > b, True, False)
```

In diesem Fall hat Variable a_groesser_b nach Ausführung der letzten Anweisung den Wert False, was unter Umständen nicht dem Willen des Programmierers bzw. des Benutzers entspricht. Was ist passiert? Der Programmierer hat a und b als Stringvariablen deklariert, danach diesen Variablen aber Zahlenwerte zugewiesen. Der Compiler versucht, daraus das Beste zu machen; er lässt die beiden Werte stillschweigend in Strings konvertieren, bevor sie den Variablen zugewiesen werden. Der abschließende Wert von a_groesser_b beruht daher auf dem Ausdruck "100" > "99", welcher den Wert False hat.

Wann braucht man Konvertierungen?

In den beiden vorgestellten Fällen wäre es sicher besser gewesen, wenn der Programmierer für die Variablen stets den richtigen Typ gewählt und sich bei der Zuweisung von Werten konsequent an diesen Typ gehalten hätte. Für den Compiler hätte es dann auch keinen Anlass zu Typumwandlungen gegeben.

Es gibt jedoch Anwendungsfälle, in denen eine Typumwandlung durchaus sinnvoll ist und, bei exakter Programmierung, auch nicht zu Fehlern führt. Im Wesentlichen handelt es sich um drei Typen von Situationen:

1. Man möchte ein Programm schreiben, das einen Wert eines bestimmten Datentyps d1 ermittelt und hat bereits eine Funktion, die genau die gewünschte Berechnung durchführt, aber den Datentyp d2 liefert. Hier bietet sich natürlich an, die bereits vorliegende Funktion zu nutzen und den von ihr gelieferten Wert in den Datentyp d1 zu transformieren.
2. Umwandlung von Benutzereingaben in Werte, mit denen man rechnen kann.
Benutzereingaben, z.B. mit Hilfe von Formularen, werden grundsätzlich als Strings eingelesen, weil dann Fehleingaben leichter abgefangen werden können. Da man aber mit Strings nicht rechnen kann, muss man sie gegebenenfalls vor der Weiterverarbeitung in ein numerisches Datenformat umwandeln.
3. Manche Aufgaben lassen sich leichter lösen, wenn man den Datentyp wechselt. Zum Beispiel kann man leichter auf die einzelnen Ziffern einer Zahl zugreifen, wenn man sie vorher in einen String verwandelt (s. Abschnitt Strings, weiter unten).

Typumwandlungen immer explizit vornehmen

Zwar versucht der Compiler, Typumwandlungen vorzunehmen, wo dies nötig erscheint, um ein Abbrechen des Programms zu verhindern, aber solche stillschweigenden Umwandlungen gelingen nicht in allen Fällen und haben auch nicht immer den von Programmierer und Benutzer gewünschten Effekt.

Typumwandlungen sollen daher immer explizit vorgenommen werden. VBA hält zu diesem Zwecke eine Reihe von Konvertierungsfunktionen bereit. Die wichtigsten sind:

CInt	Umwandlung nach Integer
CDbl	Umwandlung nach Double
CLng	Umwandlung nach Long
CStr	Umwandlung nach String
CDec	Umwandlung nach Decimal

Alle diese Funktionen beginnen mit C (für change) und sind daher im Verzeichnis der Funktionen der VBA-Hilfe leicht zu finden.

Wir greifen das erste Beispiel dieses Abschnitts noch einmal auf und betrachten eine Fassung mit expliziter Umwandlung des von der Funktion gelieferten Double-Werts in einen Integer-Wert:

```
Dim z As Integer
z = CInt(FunktionDieDoubleLiefert(einParameter))
```

Beachten Sie, dass auch bei dieser Version ein Informationsverlust in Kauf genommen werden muss, wenn der gelieferte Funktionswert signifikante Nachkommastellen besitzt. Im Unterschied zur ursprünglichen Version wird aber hier jedem Leser des Codes klar, dass durch die Umwandlung ein Informationsverlust droht.

Der Datentyp String (Vertiefung)

Deklaration von String-Variablen

In VBA wird, anders als in VB.net, kein formaler Unterschied zwischen einem einzelnen Zeichen und einer Zeichenkette gemacht. Beide gehören zum Datentyp String.

Wenn man eine Variable vom Typ String deklariert, hat man die Wahl zwischen einem String fester Länge (maximal 65535 Zeichen) und einem String variabler Länge (bis 2 Milliarden Zeichen). Unabhängig davon werden Werte von Strings immer in doppelte Anführungszeichen gesetzt. Der folgende Codeausschnitt zeigt zuerst die Deklaration einer String-Variablen für einen String variabler Länge, dann die Deklaration einer Variablen für einen String der festen Länge 1, also für ein einzelnes Zeichen.

```
Dim s1 As String
s1 = "Dies ist ein String variabler Länge"
```

```
Dim s2 As String * 1    'für ein einzelnes Zeichen
s2 = "X"
```

Funktionen zur String-Bearbeitung

Für Strings gibt es in VBA bereits viele nützliche eingebaute Funktionen. Die wichtigsten sind in der folgenden Tabelle aufgeführt.

Funktion	Effekt
Len(s)	Länge von s
Left(s, n)	Die ersten n Zeichen von s
Right(s, n)	Die letzten n Zeichen von s
Mid(s, i, n)	Liefert n Zeichen, beginnend mit dem i-ten
s & d	Verkettet s mit dem Wert d zu einem String
Trim(s)	Entfernt Leerzeichen am Anfang u. am Ende
UCase(s)	Wert von s in Großbuchstaben
LCase(s)	Wert von s in Kleinbuchstaben
Replace(s, z1, z2)	ersetzt Zeichen z1 durch z2
StrReverse(s)	liefert String in umgekehrter Reihenfolge

Besonders nützlich sind die Funktionen Len und Mid. Sie erlauben uns, alle Zeichen eines Strings einzeln zu betrachten. Die folgende Funktion liefert die Anzahl der „e“ im String s:

```
Public Function anzahl_e(ByVal s As String) As Integer
    Dim a As Integer
    a = 0
    Dim i As Integer
    For i = 1 To Len(s)
```

```

        If Mid(s, i, 1) = "e" Then a = a + 1
    Next i
    anzahl_e = a
End Function

```

Eine Besonderheit von Strings ist, dass man sie nicht ändern kann. Schreibt man daher eine Funktion, die einen String als Parameter empfängt und diesen in veränderter Form zurückliefern soll, dann kann diese Funktion nicht den Original-String verändern, sondern sie muss einen neuen aufbauen. Wir betrachten hierfür eine Funktion, welche in einem String alle „u“ durch „x“ ersetzt:

```

Public Function xFueru(ByVal s As String) As String
    Dim n As String
    n = ""                'leerer String
    Dim i As Integer
    For i = 1 To Len(s)
        If Mid(s, i, 1) = "u" Then
            n = n & "x"
        Else
            n = n & Mid(s, i, 1)
        End If
    Next i
    xFueru = n
End Function

```

Bitte studieren Sie die Funktion genau, denn sie enthält ein Prinzip, das in vielen ähnlichen Funktionen zur Anwendung kommt: zunächst wird eine String-Variable n deklariert, dann wird dieser Variablen ein sog. „leerer String“ zugewiesen, also ein String, der keine Zeichen enthält. Sodann wird in einer Schleife der String s Zeichen für Zeichen durchgegangen. Bei jedem Zeichen wird geprüft, ob es sich um ein „u“ handelt. Wenn dies der Fall ist, wird an den Wert von n ein „x“ angehängt, wenn nicht, wird das gerade betrachtete Zeichen unverändert angehängt.

In der Anweisung

```
n = n & "x"
```

wird der Verkettungsoperator & benutzt, um der Hilfsvariablen n einen String zuzuweisen, der aus dem String besteht, auf den die Variable n vorher verwies und einem angehängten „x“. Analog ist die Anweisung `n = n & Mid(s, i, 1)` zu interpretieren. Mit anderen Worten: mit jedem neuen Durchlauf der Schleife wird der Variablen n ein neuer, längerer String zugewiesen.

Es ist Ihnen sicher aufgefallen, dass die Funktion xFueru nur ein Spezialfall der allgemeineren Funktion Replace ist, die in der Tabelle weiter oben aufgeführt ist. Es ist eine gute Übung, wenn Sie die Funktion Replace nun selbst programmieren.

Vordefinierte String-Konstanten

Eine Besonderheit von Strings ist, dass sie auch sog. Steuerzeichen enthalten können, wie z.B. einen Zeilenvorschub oder einen Seitenvorschub. Indem man solche Zeichen in Strings einbaut, kann man

deren Aussehen bei der Ausgabe auf dem Bildschirm (s. weiter unten, Ein- und Ausgabe) oder auf Papier gestalten. In der folgenden Funktion untereinander wird zwischen die einzelnen Zeichen des Strings *s* jeweils eine Zeilenvorschub (Konstante *vbLf*) eingefügt, so dass die Zeichen bei einer Ausgabe auf dem Bildschirm in einer MsgBox untereinander angezeigt werden.

```
Public Function untereinander(ByVal s As String) As String
    Dim n As String
    n = ""
    Dim i As Integer
    For i = 1 To Len(s)
        n = n & Mid(s, i, 1) & vbLf
    Next i
    untereinander = n
End Function
```

Anmerkung: funktioniert nicht bei Aufruf aus einem Tabellenblatt.

ASCII und der Vergleich von Zeichenketten

Die Zeichen, die in Strings enthalten sind, werden von VBA intern als Zahlen dargestellt. Welche Zahlen den einzelnen Zeichen entsprechen, ist in der sogenannten ASCII-Tabelle (American Standard Code for Information Interchange) festgelegt. Seit einigen Jahren gibt es unter dem Namen Unicode eine erheblich umfangreichere Tabelle, die viele Schriftzeichen aller Nationalitäten umfasst. Die alte ASCII-Tabelle ist jedoch darin enthalten, so dass man sie immer noch benutzen kann. Die folgende Tabelle enthält einen Ausschnitt der ASCII-Tabelle:

Zeichen	ASCII	Zeichen	ASCII
...	...	...	...
0	48	Z	90
1	49	...	...
...	...	a	97
9	57	b	98
...	...	...	...
A	65	z	122
B	66	...	...

Die Zahlenentsprechungen der Zeichen sind hier im Dezimalsystem angegeben. Häufig findet man auch Tabellen, in denen die Zahlenäquivalente im Hexadezimalsystem (Basis 16) angegeben sind. Die Pünktchen kennzeichnen Auslassungen. In diesen Bereichen finden sich Sonderzeichen, wie „!“ und „?“ und Umlaute, wie „ä“ und „ü“.

Die Zahlenentsprechungen dienen nicht nur der internen Speicherung der Zeichen, sondern auch als Basis bei Größenvergleichen. Die Vergleichsoperatoren, die für numerische Datentypen erlaubt sind, sind nämlich auch für einzelne Zeichen und Zeichenketten zugelassen. Ob ein Zeichen größer oder kleiner als ein anderes ist, entscheidet die Position der Zeichen in der ASCII-Tabelle. Aus der Tabelle können wir direkt ersehen, dass folgende Aussagen gelten:

```

"0" < "1"
"A" < "B"
"A" < "a"
"a" < "z"

```

Auch beim Vergleich längerer Zeichenketten dient die Tabelle als Grundlage. Zunächst werden die Zeichen am weitesten links verglichen. Sind diese gleich, so wird das nächste Zeichen verglichen usw. Nach dieser Methode kommen wir beispielsweise zu folgenden Aussagen:

```

"Aber" < "aber"
"EVA" < "Eva"
"123" < "abc"
"Anton" < "Berta"

```

Dies ist nicht unbedingt die Ordnung, die wir von Wörterbüchern kennen. Dort werden „A“ und „a“ gleichgesetzt bzw. beim Größenvergleich nicht unterschieden. Will man einen Größenvergleich in dem Sinne vornehmen, wie dies in Wörterbüchern geschieht, so hat man in VBA zwei Möglichkeiten:

- man verwandelt beide Strings, die man vergleichen will, in Großbuchstaben (oder in Kleinbuchstaben). Hierfür kann man die Funktionen UCase und LCase (s. oben) verwenden.
- man setzt an den Anfang des Moduls (außerhalb von Funktionen und Subs) die Zeile
Option Compare Text

In der folgenden Funktion kommt es auf die Unterscheidung von Groß- und Kleinbuchstaben an. Aus einem String sollen alle Großbuchstaben und Sonderzeichen entfernt werden. Wir erreichen dies, indem wir in den neu zu bildenden String nur die Zeichen übernehmen, welche im Bereich der Kleinbuchstaben liegen:

```

Public Function nurKlein (ByVal s As String) As String
    Dim n As String
    n = ""
    Dim i As Integer
    For i = 1 to Len(s)
        If Mid(s,i,1) >= "a" And Mid(s,i,1) <= "z" Then
            n = n & Mid(s,i,1)
        End If
    Next i
    nurKlein = n
End Function

```

Umwandlung von Zeichen in ihre ASCII-Entsprechung und umgekehrt

In manchen Anwendungen (z.B. in der Kryptographie) muss man Zeichen in ihre Zahlenentsprechungen oder aber die Zahlenentsprechungen zurück in Zeichen verwandeln. Hierfür gibt es die eingebauten Funktionen

Asc	liefert den ASCII-Wert eines Zeichens
AscW	wie Asc
Chr	liefert das Zeichen für einen ASCII-Wert
ChrW	wie Chr

Die folgende Funktion verwandelt einen String *s* zunächst in Großbuchstaben und ersetzt dann alle Buchstaben in ihre Zahlenentsprechungen. Wir wollen annehmen, dass *s* nur Buchstaben enthält, also keine Sonderzeichen und sonstigen Zeichen.

```
Public Function inZahlen(ByVal s As String) As String
    s = UCase(s)
    Dim n As String
    n = ""
    Dim i As Integer
    For i = 1 To Len(s)
        n = n & Asc(Mid(s, i, 1))
    Next i
    inZahlen = n
End Function
```

Beachten Sie, dass die Zahlenentsprechungen in einem String zusammengefasst werden. Ist *s* beispielsweise gleich „ABC“, so liefert die Funktion den String „656667“. Für „abc“ wird derselbe String geliefert, weil vor der Umwandlung in Zahlen alle Buchstaben in Großbuchstaben konvertiert werden.

Umwandlung von Zeichenketten in Zahlen

Wir wollen noch eine andere Art der Umwandlung in Zahlen betrachten. Wie bereits erwähnt, werden Eingaben des Benutzers gewöhnlich als Strings entgegengenommen. Gibt der Benutzer also die Zahl 345 in ein Formular ein, so wird daraus im Formular der String „345“. Da man mit einem String nicht rechnen kann, muss er vor der Weiterverwendung in eine Zahl zurück verwandelt werden. Dies kann mit Hilfe der Funktionen `CInt` bzw. `CDBl` geschehen, wie der folgende Codeausschnitt zeigt:

```
Dim s As String
s = "345"
Dim z As Integer
z = CInt(s)

s = "455.17"
Dim d As Double
d = CDBl(s)
```

Beachten Sie, dass es hier nicht sinnvoll wäre, die Ziffern der Zahlenstrings in ihre ASCII-Äquivalente zu verwandeln, denn die Entsprechung von „3“ ist nicht 3, sondern 51.

Es besteht auch die Möglichkeit, aus einem gemischten String, der vorne Ziffern und danach andere Zeichen enthält, den Zahlenanteil herauszuziehen. Man verwendet hierzu die Funktion Val, welche die betreffende Zahl als Double liefert:

```
Dim s As String
s = "123abcdefg"
Dim z As Integer
z = CInt(Val(s))

s = "44.35abc"
Dim d As Double
d = Val(s)
```

Prozeduren (Sub-Prozeduren)

Prozeduren sind (ähnlich wie Funktionen) abgeschlossene Programmeinheiten, die sich in Modulen befinden. Anders als Funktionen liefern Prozeduren aber keinen Wert. Häufig werden Prozeduren dafür eingesetzt, den Programmablauf zu koordinieren und um Eingaben und Ausgaben anzustoßen.

Prozeduren können Funktionen und auch andere Prozeduren aufrufen, um Teilaufgaben zu delegieren. Wir haben auch die Möglichkeit, mit Hilfe von Prozeduren Funktionen zu testen, ohne dass wir hierfür in ein Tabellenblatt wechseln müssen. Dies ist besonders dann wichtig, wenn wir Funktionen testen wollen, die nicht als benutzerdefinierte Funktionen zugelassen sind.

Aufruf von Funktionen aus Prozeduren

Das folgende Beispiel besteht aus einer Funktion gespreizt und einer Prozedur StringSpreizen, die sich in unterschiedlichen Modulen befinden. Die Funktion nimmt einen String s entgegen und liefert einen String, der die Zeichen von s enthält, in den aber zusätzlich Leerzeichen zwischen die übrigen Zeichen eingefügt sind (gespreizte bzw. gesperrte Darstellung des Worts).

Die Prozedur koordiniert den Ablauf. Sie liest zunächst mittels einer Input-Box das zu spreizende Wort vom Benutzer ein. Mit Hilfe eines Aufrufs der Funktion spreizen lässt sie das Wort spreizen und gibt es dann in einer MsgBox am Bildschirm aus.

Um das Programm laufen zu lassen, muss man die Prozedur im VBE starten.

```
Public Function gespreizt(ByVal s As String) As String
    Dim n As String
    n = ""
    Dim i As Integer
    For i = 1 To Len(s) - 1
        n = n & Mid(s, i, 1) & " "
    Next i
    n = n & Mid(s, Len(s), 1)
```

```

    gespreizt = n
End Function

```

```

Public Sub StringSpreizen()
    Dim s As String
    s = InputBox("Wort eingeben: ")
    MsgBox StrBea.gespreizt(s)
End Sub

```

Beachten Sie, dass dem Aufruf der Funktion spreizen in der Prozedur StringSpreizen der Name des Moduls, in dem sich die Funktion befindet (StrBea), vorangestellt ist. Dies ist nötig, weil sich die Funktion in einem anderen Modul befindet, als die Prozedur.

Beachten Sie auch, dass die Kopfzeile der Prozedur nach dem Prozedurnamen eine Parameterklammer enthält. In diesem Beispiel ist sie leer, weil die Prozedur keine Parameter benötigt. Es gibt aber auch Prozeduren, die Parameter entgegennehmen. Ein Beispiel hierfür ist die eingebaute Prozedur MsgBox. Sie empfängt den auf den Bildschirm auszugebenen Text als Parameter.

Das Beispiel demonstriert noch einen subtilen Unterschied, der zwischen Funktionen und Prozeduren gilt, wenn beim Aufruf Parameter übergeben werden müssen. Die Prozedur StringSpreizen enthält zwei Funktionsaufrufe, den Aufruf der Funktion InputBox und den der Funktion gespreizt. Bei beiden wird ein Parameter übergeben und dieser Parameter ist jeweils in Klammern gesetzt. Auch beim Aufruf der Prozedur MsgBox wird ein Parameter übergeben (s. oben). Dieser wird aber, anders als bei einem Funktionsaufruf, nicht in Klammern gesetzt.

ByVal und ByRef

Wir können jetzt auch ein Thema ansprechen, das wir bisher vor uns hergeschoben haben: Was bedeutet es, wenn wir Parameter ByVal (by value) übergeben? Und: gibt es noch eine andere Möglichkeit? Die Antwort auf die zweite Frage zuerst: ja, es gibt eine andere Möglichkeit, nämlich die Parameterübergabe ByRef (by reference). Das folgende Beispiel in zwei Varianten soll den Unterschied zeigen. Das Programm besteht aus zwei Prozeduren Sub1 und Sub11, die sich in demselben Modul befinden. Sub11 ruft Sub1 auf und übergibt ihr einen Integer-Wert als Parameter.

In der ersten Variante (oben) wird der Parameter in gewohnter Weise ByVal (by value) übergeben. Startet man Sub11, so wird am Ende der Ausführung die MsgBox sichtbar, die den Wert von a meldet. Wie nicht anders zu erwarten, ist dieser Wert 3.

```

Public Sub Sub1(ByVal x As Integer)
    x = 3 * x
End Sub

Public Sub Sub11()
    Dim a As Integer
    a = 3
    Sub1 a
    MsgBox a
End Sub

```

Die zweite Variante unterscheidet sich von der ersten dadurch, dass der Parameter an Sub1 ByRef (by reference) übergeben wird. Ansonsten ist der Wortlaut gleich.

```
Public Sub Sub1(ByRef x As Integer)
    x = 3 * x
End Sub

Public Sub Sub11()
    Dim a As Integer
    a = 3
    Sub1 a
    MsgBox a
End Sub
```

Erstaunlicherweise meldet jetzt die MsgBox für a den Wert 9. Wie ist dies zu erklären? Der Vermerk ByRef in der Parameterklammer von Sub1 sorgt dafür, dass Sub11 beim Aufruf von Sub1 nicht den Wert 3 übergibt, sondern einen Verweis (eine Referenz) auf die Variable a. Da die Parametervariable auf der Seite von Sub1 x heißt, kann man nun innerhalb von Sub1 über den Namen x auf die Variable a von Sub11 zugreifen. Wird jetzt also in Sub1 der Wert von x verändert (Multiplikation mit 3), so wird tatsächlich die Variable a von Sub11 verändert.

Schon an diesem kleinen Beispiel wird deutlich, wie gefährlich eine solche Art der Programmierung sein kann. Bisher (Übergabe ausschließlich ByVal) konnten wir davon ausgehen, dass die Variablen einer Funktion oder Prozedur mit den Variablen einer anderen Funktion oder Prozedur nichts zu tun haben. Wir konnten sie isoliert voneinander betrachten. Gab es einen Fehler, so genügte es, sich bei der Fehlersuche auf die betreffende Funktion oder Prozedur zu beschränken. Dies ändert sich grundlegend, wenn man von der Parameterübergabe ByRef Gebrauch macht. Nun kann es sein, dass Veränderungen von Variablen ihre Ursache in ganz anderen Prozeduren oder Funktionen haben.

Um diese Probleme zu vermeiden, **sollte man die Variablenübergabe ByRef unbedingt vermeiden!** Es gibt einige Ausnahmen, die von VBA erzwungen werden. Beispielsweise muss die Übergabe von Array-Parametern (s. unten) in VBA immer ByRef erfolgen. Hier müssen wir uns fügen. Eine zweite Ausnahme gibt es bei der sog. objektorientierten Programmierung, die in unserem Kurs aber keine Rolle spielt.

Arrays

Grundlagen

Im Deutschen werden Arrays häufig auch als Felder bezeichnet. Wir wollen aber bei der englischen Bezeichnung bleiben, weil Feld eine Bezeichnung ist, die für viele verschiedene Dinge verwendet wird.

Ein Array bezeichnet eine Sammlung von Werten oder Objekten, die alle denselben Datentyp haben. Die einzelnen Elemente in einem Array können über Indizes angesprochen werden. Den Arrays der Informatik entsprechen in der Mathematik die indizierten Variablen.

Man kann sich ein Array als ein **Regal** vorstellen, welches vom Programm zur Speicherung von Daten verwendet werden kann. Die einfachsten dieser Regale haben nur eine Reihe von Fächern. Diese Fächer sind durchnummeriert, damit man gezielt auf den Inhalt einzelner Fächer zugreifen kann, und jedes der Fächer kann nur einen Wert aufnehmen. Die Nummerierung kann man frei wählen, aber wir wollen sie der Einfachheit halber immer bei 1 beginnen lassen.

Wir legen ein solches „Regal“ an, indem wir eine Array-Variable deklarieren. Für ein Regal mit 5 Fächern, das zur Aufnahme von fünf Integer-Werten bestimmt ist, lautet eine Deklaration z.B. :

```
Dim a (1 To 5) As Integer
```

wobei a ein frei vom Programmierer gewählter Name ist. Das folgende Bild zeigt das Regal bzw. Array nach dieser Deklaration. Es enthält noch keine Werte.

1		2		3		4		5	

Im Verlauf des Programmablaufs können den Fächern des Regals dann Werte zugewiesen werden. Dies könnte beispielsweise durch folgende Anweisungen geschehen:

```
a(1) = 32
a(2) = -315
a(3) = 55
a(4) = 27
a(5) = 100
```

Das nächste Bild zeigt das Array-Regal nach Durchführung dieser Zuweisungen.

1	32	2	-315	3	55	4	27	5	100

Wenn für die Zwecke des Programms ein solches einfaches Regal nicht ausreicht, kann man auch eines verwenden, das aus mehr als einer Reihe besteht. Man spricht dann von einem **zweidimensionalen Array**.

1,1		1,2		1,3		1,4		1,5	
2,1		2,2		2,3		2,4		2,5	
3,1		3,2		3,3		3,4		3,5	

Gegenüber dem einfachen Regal von oben (dem eindimensionalen Array) ist bei dieser Form die Nummerierung der Fächer etwas komplizierter. Jedes Fach wird durch zwei Ziffern identifiziert, die durch Kommata voneinander getrennt sind. Die erste der beiden Ziffern steht für die Zeile, die zweite

für die Spalte des Regals. Ein Array wie das oben dargestellte können wir z.B. durch folgende Deklaration anlegen:

```
Dim a(1 To 3, 1 To 5) As Integer
```

Beim Besetzen eines bestimmten Fachs verwenden wir beide Nummerierungsziffern (Indizes). Mit der Zuweisung

```
a(2,3) = 620
```

können wir beispielsweise das Fach in der zweiten Zeile und der dritten Spalte mit dem Wert 620 besetzen.

Beispiel 1: eine Prozedur, die ein eindimensionales Array benutzt

Wir beginnen mit einem einfachen Beispiel, einer Prozedur, in der wir zunächst ein Array aufbauen, dann die einzelnen Elemente in einen String schreiben, um sie schließlich in einer MsgBox auszugeben. Die Prozedur hat keinen Zweck außer der Demonstration der Array-Benutzung.

```
Public Sub ArrayEinf()  
    Dim a(1 To 3) As Integer  
    a(1) = 4  
    a(2) = -2  
    a(3) = 7  
    Dim s As String  
    s = ""  
    Dim i As Integer  
    For i = 1 To 3  
        s = s & a(i) & " "  
    Next i  
    MsgBox s  
End Sub
```

Beispiel 2: Eine Funktion mit einem Array als Parameter

Wir betrachten nun eine Funktion, die ein Double-Array als Parameter empfängt und den maximalen Wert der darin enthaltenen Double-Werte liefert.

Die Funktion soll natürlich nicht auf Arrays mit einer bestimmten Ausdehnung beschränkt sein, sondern soll beliebige Arrays verarbeiten können. Dies stellt uns vor ein Problem, denn bei der Suche nach dem Maximum sollten wir wissen, wie viele Elemente in dem übergebenen Array enthalten sind. Glücklicherweise gibt es eine Funktion, welche uns genau diese Information bereit stellen kann. Sie heißt UBound (für upper bound bzw. Obergrenze) und liefert uns den höchsten vorkommenden Index im Array. Gehen wir davon aus, dass die Indizes bei 1 beginnen, so lautet unsere Funktion zur Bestimmung des Maximums wie folgt:

```

Public Function maxw(ByRef a() As Double) As Double
    Dim m As Double
    Dim i As Integer
    m = a(1)
    For i = 2 To UBound(a)
        If a(i) > m Then m = a(i)
    Next i
    maxw = m
End Function

```

Dass der Parameter ein Array ist, erkennt VBA an der leeren Klammer hinter dem Namen des Parameters. Es ist nicht nötig und auch nicht möglich, hier eine konkrete Obergrenze anzugeben. Dies würde auch die Funktion in ihrer Anwendung sehr beschränken. Ohne Angabe einer Obergrenze kann sie Arrays beliebiger Größe verarbeiten.

Beachten Sie, dass die Übergabe eines Array-Parameters in VBA immer ByRef erfolgen muss.

Leider kann die obige Funktion nicht als benutzerdefinierte Funktion verwendet werden. Aber wir können sie von einer Prozedur aus aufrufen und testen. Wir wollen hierfür annehmen, dass die Funktion sich in einem anderen Modul befindet als die Prozedur, und dass dieses Modul AU heißt.

```

Public Sub MaxwTest()
    Dim f(1 To 3) As Double
    f(1) = 4
    f(2) = -2
    f(3) = 3
    MsgBox AU.maxw(f)
End Sub

```

Beispiel 2a: eine universell verwendbare Version der Funktion

Wenn sich alle beteiligten Programmierer an die Konvention halten, die Indizes der Arrayelemente bei 1 beginnen zu lassen, ist die eben betrachtete Funktion keinen Einsatzbeschränkungen unterworfen. Dies gilt nicht, falls die Untergrenze beliebig sein kann. Für diesen Fall müssen wir die Funktion noch allgemeiner formulieren.

Eine universell einsetzbare Version bekommen wir, wenn wir als Index-Untergrenze nicht 1 verwenden, sondern die Funktion LBound (für lower bound bzw. Untergrenze):

```

Public Function maxw(ByRef a() As Double) As Double
    Dim m As Double
    Dim i As Integer
    m = a(LBound(a))
    For i = LBound(a) + 1 To UBound(a)
        If a(i) > m Then m = a(i)
    Next i
    maxw = m
End Function

```

Eine Testprozedur für diese Version der Funktion ist z.B. folgende:

```
Public Sub MaxwTest()
    Dim f(-27 To -25) As Double
    f(-27) = 4
    f(-26) = -2
    f(-25) = 3
    MsgBox AU.maxw(f)
End Sub
```

Beispiel 3: Eine Funktion, die ein Array liefert

Wir betrachten nun eine Funktion, die ein Array liefert. Die Funktion nimmt zwei Integer-Werte als Parameter entgegen, wovon der erste kleiner oder gleich dem zweiten sein muss. Sie liefert die ganzen Zahlen in dem Bereich vom ersten bis zum zweiten Wert, die Grenzen eingeschlossen. Die Kopfzeile der Funktion lautet:

```
Public Function vonUGbisOG(ByVal ug As Integer, ByVal og As Integer) _
    As Integer()
```

Der Parameter ug gibt dabei den Startwert an (Untergrenze), og steht für den Endwert (Obergrenze). Für ug = 3 und og = 7 soll demnach die Folge 3, 4, 5, 6, 7 geliefert werden; für ug = -2 und og = 3 ergibt sich die Folge -2, -1, 0, 1, 2, 3. Gehen wir davon aus, dass der Index der Elemente bei 1 beginnt, so beträgt der letzte Index og – ug + 1. Wir betrachten nun die Funktion:

```
Public Function vonUGbisOG(ByVal ug As Integer, ByVal og As Integer)
    As Integer()

    Dim i As Integer
    Dim f() As Integer
    ReDim f(1 To og - ug + 1)
    For i = ug To og
        f(i - ug + 1) = i
    Next i
    vonUGbisOG = f
End Function
```

Beachten Sie die Klammer bei der Angabe des Funktionstyps; sie zeigt an, dass nicht ein einfacher Integer-Wert, sondern ein Integer-Array geliefert wird.

Im Rumpf der Funktion wird zunächst eine Array-Variable f deklariert, welche das zu liefernde Array aufnehmen soll. Dabei werden keine Grenzen angegeben, sondern die Klammer wird leer gelassen. Dies ist immer dann erforderlich, wenn die Größe des Arrays entweder noch nicht bekannt ist oder sie aus Parametern errechnet werden muss. Im Beispiel ist das Letztere der Fall, den wir können die Größe des Arrays aus den Werten der beiden Parameter ermitteln. Die Ausdehnung des Arrays wird dann mit Hilfe einer ReDim-Anweisung "nachdeklariert". Beachten Sie, dass bei der Zuweisung der Array-Variablen an den Funktionsnamen in der letzten Zeile die Variable f ohne Klammer benutzt wird.

Um die Funktion zu testen, schreiben wir eine kleine Prozedur. Darin deklarieren wir zunächst ein Integer-Array a unbestimmter Größe. Dieser Variablen weisen wir dann das Ergebnis des Aufrufs der Funktion vonUGbisOG zu. Anschließend können wir in einer MsgBox das erste und letzte Element von a ausgeben, durch einen Zeilenvorschub getrennt:

```
Public Sub vonUGbisOGTest()
    Dim a() As Integer
    a = vonUGbisOG(4, 17)
    MsgBox "erstes Element " & a(LBound(a)) _
        & vbCrLf & "letztes Element " & a(UBound(a))
End Sub
```

ReDim Preserve

Wir betrachten nun noch einen anderen Fall einer Funktion, die ein Array liefert. Nicht immer ist es so leicht, den notwendigen Umfang des zu liefernden Arrays zu ermitteln wie bei vonUGbisOG. Unsere Funktion soll die Teilermenge einer ganzen Zahl z als Array liefern. Das Problem ist, dass wir erst wissen, wie viele Teiler es werden, wenn wir sie ermittelt haben.

Wir wenden folgende Strategie an: Zunächst definieren wir ein unbestimmtes Array für die Teiler, also eines ohne Angabe der Ausdehnung. Dann ermitteln wir die Teiler mit Hilfe einer Schleife, und jedes Mal, wenn wir einen zusätzlichen Teiler gefunden haben, erweitern wir das Array um ein Element und speichern darin den neu gefundenen Teiler. Damit beim Ausdehnen des Arrays die vorher darin gespeicherten Werte nicht verloren gehen, müssen wir nun ReDim mit dem Zusatz Preserve verwenden.

```
Public Function TeilerB1(ByVal z As Integer) As Integer()
    Dim i As Integer
    Dim t() As Integer
    Dim umfang As Integer
    umfang = 0
    For i = 1 To z
        If z Mod i = 0 Then
            umfang = umfang + 1
            ReDim Preserve t(1 To umfang)
            t(umfang) = i
        End If
    Next i
    TeilerB1 = t
End Function
```

Mehrdimensionale Arrays

In VBA kann man Arrays mit sehr vielen Dimensionen bzw. Indices ausstatten. Diese Möglichkeit wird allerdings selten ausgenutzt. Neben den eindimensionalen Arrays werden in der Praxis vorwiegend

zweidimensionale Arrays verwendet. Man braucht sie, um Tabellen bzw. Matrizen speichern und bearbeiten zu können.

In der folgenden kleinen Prozedur wird ein zweidimensionales Array deklariert und mit Werten gefüllt. Anschließend werden die Werte einzeln in MsgBoxes ausgegeben. Zur Ausgabe brauchen wir eine Doppelschleife. Die Schleifenvariable *i* geht dabei die Zeilen der Tabelle durch, die Schleifenvariable *j* jeweils die Spalten innerhalb einer Zeile. Auch bei diesem Beispiel gilt: es hat keinen Zweck, außer den Umgang mit zweidimensionalen Arrays zu demonstrieren.

```
Public Sub Tabelle()  
    Dim a(1 To 3, 1 To 2) As Integer  
    a(1, 1) = 135  
    a(1, 2) = -34  
    a(2, 1) = 2759  
    a(2, 2) = 2345  
    a(3, 1) = 9  
    a(3, 2) = -365  
    Dim i As Integer, j As Integer  
    For i = 1 To 3  
        For j = 1 To 2  
            MsgBox a(i, j)  
        Next j  
    Next i  
End Sub
```

Auch bei mehrdimensionalen Arrays stehen die Funktionen LBound und UBound zur Verfügung, allerdings in einer erweiterten Form mit zwei Parametern. Der zweite Parameter steht für die Nummer der Dimension (Zählung bei 1 beginnend). Die folgende Variante des Beispiels macht von den Funktionen Gebrauch:

```
Public Sub Tabelle()  
    Dim a(1 To 3, 1 To 2) As Integer  
    a(1, 1) = 135  
    a(1, 2) = -34  
    a(2, 1) = 2759  
    a(2, 2) = 2345  
    a(3, 1) = 9  
    a(3, 2) = -365  
    Dim i As Integer, j As Integer  
    For i = LBound(a, 1) To UBound(a, 1)  
        For j = LBound(a, 2) To UBound(a, 2)  
            MsgBox a(i, j)  
        Next j  
    Next i  
End Sub
```

Wir schließen diesen Abschnitt mit dem Beispiel einer Funktion, welche einen Spaltenvektor eines zweidimensionalen Arrays liefert. Wir setzen dabei voraus, dass die Indizes mit 1 beginnen.

```
Public Function Spaltenvektor (ByRef m() As Integer, _  
                                ByVal Spalte As Integer) As Integer()  
    Dim v() As Integer  
    ReDim v(1 To UBound(m, 1))  
    Dim i As Integer  
    For i = 1 To UBound(m, 1)  
        v(i) = m(i, Spalte)  
    Next i  
    Spaltenvektor = v  
End Function
```

Benutzerdefinierte Datentypen

Während man mit einem Array eine Sammlung von Elementen desselben Datentyps verwalten kann, gestatten benutzerdefinierte Datentypen, Elemente verschiedenen Typs zu bündeln. Dazu können auch Arrays gehören.

Ein Beispiel soll den Nutzen und den Gebrauch dieser Datentypen veranschaulichen:

Mit Hilfe von Funktionen soll eine Menge von n Zufallszahlen erzeugt werden.
Auch soll das Maximum dieser Zahlen ermittelt werden.

Die Aufgabe ist überschaubar, stellt aber trotzdem vor Probleme. Wir könnten eine Funktion schreiben, welche eine Menge von n Zufallszahlen in Form eines Arrays liefert. Da aber eine Funktion immer nur ein Ergebnis liefert, müssten wir eine zweite Funktion schreiben, welche aus den erzeugten Zahlen das Maximum ermittelt. Nun führt dieses Vorgehen sicher zum gewünschten Ergebnis, aber es ist ineffizient. Es wäre nämlich prinzipiell möglich, in einem Aufwasch (d.h. mit einer einzigen Schleife) die Zahlen zu erzeugen und das Maximum daraus zu ermitteln. Wenn wir aber die beiden Aufgaben durch separate Funktionen durchführen lassen, brauchen wir zwei Durchgänge durch die Zahlenfolge.

Hier schafft nun ein benutzerdefinierter Datentyp Abhilfe. In einem solchen Datentyp können wir die Zahlenfolge und den Maximalwert zusammenfassen. Wir zeigen im Folgenden, wie dies geschehen kann.

Ein benutzerdefinierter Datentyp wird auf Modulebene deklariert, also außerhalb der im Modul befindlichen Funktionen und Sub-Prozeduren. Falls man ihren Gültigkeitsbereich nicht explizit beschränkt (mit Private), sind diese Datentypen in der ganzen Anwendung bekannt und gültig.

Es wird zunächst ein Typ ZZ deklariert, der aus einem Double-Array r und einem einzelnen Double-Wert maxW besteht. Die darunter stehende Funktion zufall liefert ein Exemplar des Datentyps ZZ. Sie befüllt das Array r mit n Zufallszahlen und setzt maxW gleich dem Maximum aus diesem Array.

Beachten Sie die Punktnotation: mit z.maxW wird die Variable maxW innerhalb der Variablen z angesprochen.

Wie die Funktion zufall verwendet werden kann, zeigt die Sub-Prozedur zufallTest. Von dort soll die Funktion aufgerufen werden. Damit das Ergebnis des Aufrufs auch von der Prozedur entgegen genommen werden kann, wird zunächst eine Variable t vom Typ ZZ deklariert. Ihr wird dann das Ergebnis des Funktionsaufrufs zugewiesen. Anschließend werden mit Hilfe von MsgBoxes ausgewählte Werte daraus ausgegeben.

```

Type ZZ
    r() As Double
    maxW As Double
End Type

Public Function zufall(ByVal n As Integer) As ZZ
    Dim z As ZZ
    ReDim z.r(1 To n)
    z.maxW = 0
    Dim i As Integer
    For i = 1 To n
        z.r(i) = Rnd
        If z.r(i) > z.maxW Then z.maxW = z.r(i)
    Next i
    zufall = z
End Function

Public Sub zufallTest()
    Dim t As ZZ
    t = zufall(3)
    MsgBox t.r(LBound(t.r)) & " " & t.r(UBound(t.r))
    MsgBox t.maxW
End Sub

```

Funktionen, welche einen benutzerdefinierten Datentyp liefern, kann man leider nicht als benutzerdefinierte Funktionen einsetzen. Excel kann einen solchen Typ nicht in der Zelle eines Tabellenblatts darstellen.

Ereignisgesteuerte Programmierung

Begriff

Bei der ereignisgesteuerten Programmierung benutzt man Formulare (Masken) zur Ein- und Ausgabe der Daten. Aktionen des Benutzers, wie im folgenden Beispiel das Klicken auf die Schaltfläche <Wort spreizen>, stellen Ereignisse dar, auf die das Programm adäquat reagieren muss. Dies geschieht durch Ausführung einer Ereignisprozedur, die speziell auf das jeweilige Ereignis zugeschnitten ist.

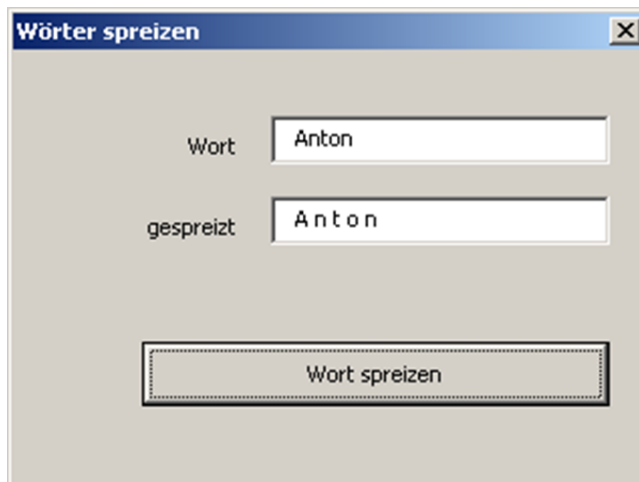
Ablauf eines ereignisgesteuerten Programms

Das erste Bild zeigt das Formular, nachdem der Benutzer das zu spreizende Wort eingegeben hat. Er hat jedoch noch nicht die Schaltfläche (Button) mit der Aufschrift „Wort spreizen“ gedrückt.



The screenshot shows a VBA UserForm titled "Wörter spreizen". It has two text boxes: "Wort" and "gespreizt". The "Wort" box contains the text "Anton". The "gespreizt" box is empty. Below the text boxes is a button labeled "Wort spreizen".

Das zweite Bild zeigt das Formular nach dem Anklicken der Schaltfläche „Wort spreizen“ und der darauf folgenden Verarbeitung dieses Ereignisses.

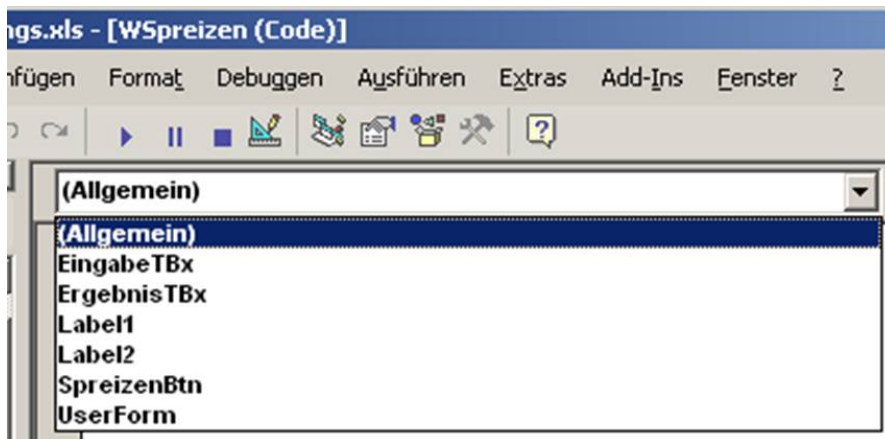


The screenshot shows the same VBA UserForm "Wörter spreizen". The "Wort" box still contains "Anton". The "gespreizt" box now contains "A n t o n", with spaces between the letters. The "Wort spreizen" button is highlighted with a dashed border, indicating it was just clicked.

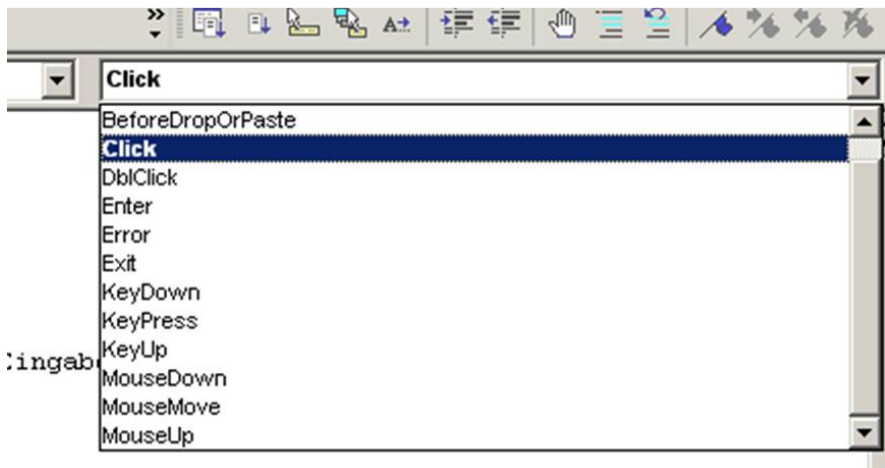
Programmierung von Ereignisprozeduren

Im Folgenden wird gezeigt, wie die Ereignisprozedur angelegt werden kann, welche sich auf das Anklicken des Buttons mit der Aufschrift „Wort spreizen“ bezieht.

Im ersten Schritt wird aus der Combox über dem Codefenster das Formularelement ausgewählt, für das eine Ereignisprozedur erstellt werden soll. In diesem Fall ist dies SpreizenBtn



Im zweiten Schritt wird aus der rechten Combobox über dem Codefenster das Ereignis ausgewählt, auf das sich die Prozedur beziehen soll. Hier ist es das Ereignis Click.



Daraufhin generiert die Entwicklungsumgebung den Rahmen für die Ereignisprozedur:

```
Private Sub SpreizenBtn_Click()

End Sub
```

Nun ist der Programmier gefragt. Er füllt den Rumpf aus:

```
Private Sub SpreizenBtn_Click()
    Me.ErgebnisTBx = StrBea.gespreizt(Me.EingabeTBx)
End Sub
```

Das war's dann auch schon.