Programmieren mit VBA

(Entwurf)

Hans-Georg Eßer
Hochschule München
Informatik-Grundlagen, WS 2008/09

Inhaltsverzeichnis

1. Arbeiten mit Variablen

Wenn Sie Makros mit dem Makrorekorder aufnehmen, werden Ihnen keine Variablen begegnen – die tauchen erst auf, wenn Sie eigene Makroprogramme schreiben. Warum ist es hilfreich, mit Variablen zu arbeiten? Betrachten Sie dazu folgendes Beispiel, das – ohne die Hilfe von Variablen – die Zellen *A1* bis *A6* addiert und das Ergebnis in die Zelle *B1* schreibt:

```
Cells(1,2) = Cells(1,1)+Cells(2,1)+Cells(3,1)+
  Cells(4,1)+Cells(5,1)+Cells(6,1)
```

Zellen sprechen Sie in Excel-Makros immer mit Hilfe von

```
Cells(Zeile, Spalte)
```

an – merken Sie sich einfach, dass die Zeile immer vorne steht (während in einer „normalen“ Adressangabe wie *A3* zunächst die Spalte *A* und dann die Zeile *3* genannt wird).

Das ist zwar einfach zu verstehen, aber doch recht umständlich. Der folgende Code verwendet eine Hilfsvariable Summe und eine `For`-Schleife (mehr über `For`-Schleifen weiter unten):

```
Summe = 0
For i = 1 To 6
  Summe = Summe + Cells(1,i)
Next
Cells(1,2) = Summe
```

Dieser Code ist zwar länger als der ursprüngliche, aber er ist besser verständlich, und vor allem lässt er sich leicht so erweitern, dass Sie damit nicht die ersten sechs, sondern die ersten 100 Zellen addieren. Auch können Sie durch ändern eines einzigen Zeichens aus der Summe ein Produkt machen, indem Sie im Befehl

```
  Summe = Summe + Cells(1,i)
```

Aus dem „+“ ein „*“ machen. (Allerdings sollten Sie dann auch die Variable `Summe` in `Produkt` umbenennen.)

1.1 Zuweisungen

Was passiert in diesem Code? Das zentrale Element (neben der `For`-Schleife) ist die Zuweisung, die Sie am Gleichheitszeichen („=“) erkennen: Wenn Sie in einem Makro etwas in der Form

`Linke Seite = Rechte Seite`

sehen, wird immer ein Wert zugewiesen. Typischerweise finden Sie auf der linken Seite den Namen einer Variable und rechts einen „Ausdruck“, der eventuell auszurechnen ist. Betrachten Sie die folgenden einfachen Beispiele:

```
Wert = 19
```

Dieser Befehl führt dazu, dass die Variable mit Namen `Wert` als Inhalt die Zahl 19 speichert. 19 ist eine Integer-Zahl, d. h., sie hat keine Nachkommastellen (anders, als etwa 19,123). Alle folgenden Befehle können den Variableninhalt abfragen, z. B. in der Form

```
Debug.Print Wert
```

Dieser Befehl wird den Inhalt der Variable `Wert` im Direktbereich ausgeben. (Das ist das Fenster, das Sie aus dem VBA-Makro-Editor heraus mit [Strg-G] aufrufen.) Sie können Variablen aber auch in Ausdrücken auf der rechten Seite einer Zuweisung verwenden. Haben Sie z. B. in den beiden Variablen `Wert1` und `Wert2` bereits zwei Zahlen abgelegt, schreiben Sie mit dem folgenden Befehl die Summe der beiden in eine neue Variable `Wert3`:

```
Wert3 = Wert1 + Wert2
```

Diese Zuweisung ist nicht viel anders als das einfachere Beispiel `Wert = 19`, auch sie legt einen neuen Wert in der Variablen auf der linken Seite (hier: `Wert3`) ab, auf der rechten Seite steht aber nicht einfach eine Zahl, sondern ein (mathematischer) Ausdruck: eine Summe. Summiert werden dabei die Inhalte der Variablen `Wert1` und `Wert2`.

Variablen dürfen auch gleichzeitig auf der linken **und** der rechten Seite einer Zuweisung auftauchen, das ist etwa dann hilfreich, wenn Sie den Wert einer Variablen um 1 erhöhen wollen (es Sie dabei aber nicht interessiert, welchen Inhalt sie gerade hat). Das geht dann so:

```
Zaehler = Zaehler+1
```


Dieser Befehl addiert zur Variablen `Zaehler` den Wert 1. Hatte Sie vorher den Wert 5, so steht nach dem Additionsbefehl der Wert 6 darin. Probieren Sie das ruhig im Direktbereich aus: Öffnen Sie ihn mit [Strg-G] und geben Sie die folgenden Befehle ein:

```
Zaehler = 5
Print Zaehler
5
Zaehler = Zaehler + 1
Print Zaehler
6
```

Sie sehen: Das funktioniert sehr gut. Eingaben im Direktbereich finden Sie in diesem Skript immer **fett** gedruckt, damit Sie diese von den Ausgaben unterscheiden können, die hier ebenfalls zu sehen sind: Im obigen Beispiel sind 5 und 6 die Ausgaben von VBA, die restlichen Zeilen enthalten die Befehle, die Sie eingeben müssen.

1.1.1 Zuweisungen sind keine Gleichungen

Auch wenn Zuweisungen wie (mathematische) Gleichungen aussehen, haben Sie nichts mit einer Gleichheitsaussage zu tun, was schon an einer gültigen Zuweisung wie `Zaehler = Zaehler + 1` deutlich wird, die nie eine gültige Gleichung sein könnte. Darum ist es auch nicht möglich, die linke und rechte Seite in einer Zuweisung zu vertauschen, ohne die Bedeutung zu ändern: Der VBA-Befehl `x = y` hat eine andere Bedeutung als der Befehl `y = x`, und etwas wie `Zaehler + 1 = Zaehler` dürfen Sie gar nicht schreiben, weil auf der linken Seite immer eine einzelne Variable (und kein zu berechnender Ausdruck) stehen muss.

Einige Programmiersprachen verwenden statt = das Doppelsymbol `:=` als Zuweisungsoperator, um zwischen dem Test auf Gleichheit (=) und der Zuweisung zu unterscheiden. In VBA können Sie einen Befehl der Form `c=a=b` schreiben, der `a` und `b` auf Gleichheit testet und den resultierenden Wahrheitswert (`True` oder `False`) in `c` speichert:

```
a=9
b=8
c=a=b
print c
Falsch
```

1.2 Mehr zu Ausdrücken

Was kann nun ein Ausdruck (auf der rechten Seite einer Zuweisung) für Elemente enthalten? Zunächst können Sie alle Formen mathematischer Ausdrücke verwenden, dürfen also Zahlen und Variablen (die Zahlen enthalten), addieren, subtrahieren, multiplizieren und auch dividieren. Dazu verwenden Sie die mathematischen „Operatoren" +, -, * und /. Außerdem können Sie mit Klammern die Reihenfolge der Berechnung festlegen, so wie Sie es aus mathematischen Formeln kennen. Ein kleines Beispiel, das zunächst auf der rechten Seite nur Zahlen verwendet:

```
Ergebnis = (3+5)*4-2/2
```

Mit diesem Befehl summieren Sie zunächst 3 und 5 (ergibt 8), multiplizieren dieses Zwischenergebnis mit 4 (ergibt 32) und ziehen anschließend das Ergebnis der Division von 2 durch 2 (also 1) davon ab: Die Variable Ergebnis enthält nun den Wert 31.

Wollen Sie das Ganze etwas übersichtlicher gestalten, könnten Sie zusätzliche Variablen benutzen, welche die Zwischenergebnisse aufnehmen:

```
Summe = 3+5
Produkt = Summe * 4
Division = 2/2
Ergebnis = Produkt - Division
```

In diesem recht einfachen und übersichtlichen Beispiel ist das natürlich unsinnig, bei komplexeren Berechnungen kann es aber helfen – vor allem dabei, einige Wochen, Monate oder Jahre später noch zu verstehen, welchen Sinn Ihre Berechnungen haben.

Betrachten Sie dazu etwa folgendes kleines Beispiel:

… irgendwas mit Zinsen, komplizierte Formel ;)

Bisher haben Sie in Variablen und Berechnungen nur Integer-Zahlen gesehen, Sie können aber auch mit Strings, also Textketten, „rechnen".

Zwar hat es keinen Sinn, einen String vom anderen abzuziehen oder beide zu multiplizieren, aber es gibt eine Addition für Strings: Die hängt beide einfach aneinander. Betrachten Sie folgendes Beispiel:

```
Text1 = "Hallo "
Text2 = "Welt"
GesamtText = Text1 + Text2
Print GesamtText
Hallo Welt
```

2. Schleifen

Solange Sie einfach nur verschiedene Befehle nacheinander ausführen wollen, benötigen Sie keine Schleifen. Die kommen erst ins Spiel, wenn Sie Dinge wiederholt erledigen wollen – und dabei vielleicht noch gar nicht wissen, wie oft Sie etwas wiederholen wollen.

Betrachten Sie zunächst folgendes einfache Beispiel: Sie haben die Aufgabe, 10x hintereinander den Text „Ich soll mir nicht zu viel Mühe machen." In das Direktfenster zu schreiben. Als kleines Makroprogramm sieht das wie folgt aus:

```
Sub ZehnMalAusgabe()
  Debug.Print "Ich soll mir nicht zu viel Mühe machen."
  Debug.Print "Ich soll mir nicht zu viel Mühe machen."
  Debug.Print "Ich soll mir nicht zu viel Mühe machen."
  Debug.Print "Ich soll mir nicht zu viel Mühe machen."
  Debug.Print "Ich soll mir nicht zu viel Mühe machen."
  Debug.Print "Ich soll mir nicht zu viel Mühe machen."
  Debug.Print "Ich soll mir nicht zu viel Mühe machen."
  Debug.Print "Ich soll mir nicht zu viel Mühe machen."
  Debug.Print "Ich soll mir nicht zu viel Mühe machen."
  Debug.Print "Ich soll mir nicht zu viel Mühe machen."
End Sub
```

Dieser Programmcode sieht nicht besonders elegant aus, und stellen Sie sich die Variante mit 100- oder 1000-facher Ausgabe vor… Zwar können Sie mit Copy & Paste (Kopieren und Einfügen) mühelos die `Debug-Print`-Befehlszeile mehrfach im VBA-Editor einfügen, aber das Ergebnis ist kaum befriedigend.

2.1 For-Schleifen

Hier kommen die For-Schleifen in ihrer einfachsten Variante ins Spiel: als schnelle Möglichkeit, ein identisches Kommando mehrfach auszuführen. Das 10-Zeilen-Ausgabeprogramm können Sie nämlich einfach wie folgt schreiben:

```
Sub ZehnFachSchleife()
  Dim i As Integer
  For i = 1 To 10
    Debug.Print "Ich soll mir nicht zu viel Mühe machen."
  Next
End Sub
```

Hier ist `i` eine Variable, die in der `For`-Schleife als Zähler verwendet wird. VBA wird ihren Wert zunächst auf 1 setzen, dann das Innere der Schleife (hier nur die `Print`-Anweisung) ausführen, dann den Zähler zunächst von 1 auf 2 erhöhen, erneut den `Print`-Befehl ausführen usw. Das Ganze endet erst, wenn der Zähler den Wert 10 erreicht: Dann läuft ein letztes Mal der `Print`-Befehl, und danach ist die Schleife beendet.

Sie können also allgemein mit einer Schleife der Form

```
For i = 1 To n
  Befehle
Next
```

erreichen, dass die *Befehle* *n*-mal ausgeführt werden. Ich habe diese Schleifenart als „einfachste“ Variante bezeichnet, weil Sie hier exakt denselben Befehl mehrfach ausführen, d. h.: Es interessiert Sie in diesem Beispiel gar nicht, im wievielten Durchlauf Sie sich gerade befinden. Statt die Zählvariable von 1 bis 10 laufen zu lassen, könnte sie auch von 1001 bis 1010 laufen – das würde keinen Unterschied ergeben, denn es kommt hier nur auf die Reihenfolge an.

Spannender wird es, wenn Sie in jedem Schleifendurchlauf auch den aktuellen Wert der Zählvariable auswerten. Betrachten Sie dazu das folgende, nur leicht veränderte Beispiel:

```
Sub ZehnFachSchleife()
  Dim i As Integer
  For i = 1 To 10
    Debug.Print "Das ist Schleifendurchgang Nr. " & i
  Next
End Sub
```


Die wesentliche Veränderung ist hier, dass Sie nun in der Ausgabe mit `Debug.Print` auch den jeweils aktuellen Wert der Zählervariable `i` ausgeben. Das Makro erzeugt, wenn Sie es starten, die folgende Ausgabe:

```
Das ist Schleifendurchgang Nr. 1
Das ist Schleifendurchgang Nr. 2
Das ist Schleifendurchgang Nr. 3
Das ist Schleifendurchgang Nr. 4
Das ist Schleifendurchgang Nr. 5
Das ist Schleifendurchgang Nr. 6
Das ist Schleifendurchgang Nr. 7
Das ist Schleifendurchgang Nr. 8
Das ist Schleifendurchgang Nr. 9
Das ist Schleifendurchgang Nr. 10
```

Dabei können Sie den Zähler nicht nur ausgeben, sondern Sie können auch mit ihm rechnen oder ihn für den Zugriff auf Tabellenzellen verwenden. Die folgende Schleife gibt eine Multiplikationstabelle der Vielfachen von 7 aus:

```
For i = 1 to 10
  Debug.Print "7 x " & i & " = " & 7*i
Next
```

Hier gibt das Print-Kommando in jedem Schleifendurchgang einen String aus, der über `&` aus vier Teilausdrücken zusammengesetzt wird; diese sind der Reihe nach:

- `"7 x "` ist einfach der Text „7 x “ (mit einem abschließenden Leerzeichen)
- `i` ist der aktuelle Inhalt der Variable `i`, z. B. 1, 2, 3, …
- `" = "` ist wieder ein Stück Text: das Gleichheitszeichen mit je einem Leerzeichen davor und dahinter
- `7*i` wird ausgerechnet; das Sternchen ist der Multiplikationsoperator. Im ersten Schleifendurchlauf (mit `i=1`) ergibt sich also 7*1, d. h. 7; im zweiten Durchlauf 7*2, also 14, usw.

Insgesamt erhalten Sie so die folgende Ausgabe:

```
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
```

```
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
```

Genauso leicht ist der automatisierte Zugriff auf Zelleninhalte. Nehmen Sie an, dass in den Zellen A1 bis A10 Integer-Zahlen stehen, die Sie dort eingetragen haben. In der Spalte daneben sollen nun die mit 7 multiplizierten Zahlen stehen. Das leistet folgendes Makroprogramm:

```
Sub Versiebenfachen()
  Dim i As Integer
  For i = 1 To 10
    Cells(i,2) = Cells(i,1) * 7
  Next
End Sub
```

In jedem Schleifendurchgang liest VBA den Wert, der in der `i`-ten Zeile und 1. Spalte steht, also `Cells(i,1)`, multipliziert ihn mit 7 und schreibt das Ergebnis in die `i`-te Zeile und 2. Spalte, also `Cells(i,2)`.

2.2 Andere Schleifen

Manchmal kommen Sie mit `For`-Schleifen nicht weiter: Betrachten Sie etwa die Aufgabe, alle Potenzen von 2 (also 2, 4, 8, 16, 32, …) auszugeben, die kleiner als 1000 sind. Wenn Sie bereits wissen, dass es insgesamt 9 solche Werte gibt (2, 4, 8, 16, 32, 64, 128, 256, 512), könnten Sie die Aufgabe wie folgt mit einer `For`-Schleife lösen:

```
Dim nWert As Integer
nWert = 2
For i = 1 To 9
  Debug.Print nWert
  nWert = nWert * 2  ' nächste Potenz berechnen
Next
```


Das klappt aber nur, weil Sie in diesem Fall die Aufgabe schon „von Hand“ gelöst haben. Einfacher wird es, wenn Sie einen anderen Schleifentyp verwenden, der nicht auf eine feste Zahl von Ausführungen festgelegt ist, z. B. eine `While`-Schleife, die solange läuft, wie eine bestimmte Bedingung erfüllt ist. Betrachten Sie dazu den folgenden Code:

```
Dim nWert as Integer
nWert = 2
While nWert <= 1000
  Debug.Print nWert
  nWert = nWert * 2  ' nächste Potenz berechnen
Wend
```

Diese Schleife leistet das Gleiche wie die vorherige `For`-Schleife, funktioniert aber auch dann noch, wenn sich die Aufgabenstellung ändert und Sie z. B. alle 2er-Potenzen bis 5000 (statt 1000) ausgeben sollen; dann ändern Sie einfach in der `While`-Bedingung `nWert <= 1000` in `nWert <= 5000`.

2.3 Verschachtelte Schleifen

Hinter dem Schlüsselwort `Next` darf der Name der Zählervariable stehen, muss es aber nicht: VBA weiß auch ohne Hilfe, in welcher Schleife es sich gerade befindet.

Es geht also sowohl

```
For i = 1 To 10
  Debug.Print i
Next
```

als auch

```
For i = 1 to 10
  Debug.Print i
Next i
```

Nun ist es möglich, Schleifen auch zu schachteln, also innerhalb einer Schleife eine weitere zu beginnen. Wollen Sie beispielsweise Multiplikationstabellen mit den ersten zehn Vielfachen von mehreren

Zahlen in einem Rutsch erstellen, können Sie eine solche geschachtelte Schleife verwenden:

```
For nZahl = 5 To 8
  For nMultiplikator = 1 To 10
    Debug.Print nZahl & " x " & nMult & " = " & nZahl * nMult
  Next
Next
```

Wenn Sie mit geschachtelten Schleifen arbeiten, kann es aber sinnvoll sein, die Variable anzugeben – einfach der Übersicht wegen:

```
For i = 1 To 3
  For j = 8 To 9
    Print i & "/" & j,
  Next j
  Print  ' Zeilenumbruch
Next i
```

erzeugt folgende Ausgabe:

```
1/8  1/9
2/8  2/9
3/8  3/9
```

Bei dieser Verschachtelung von Schleifen spricht man von „inneren“ und „äußeren“ Schleifen – im Beispiel ist die Schleife mit Zählervariable `i` die äußere Schleife, und die `j`-Schleife ist die innere. Welche Schleife innen und welche außen läuft, ist wichtig für die Reihenfolge der Abarbeitung: Wenn Sie die beiden `For`-Zeilen am Code-Anfang austauschen, wird VBA zwar immer noch alle gewünschten Kombinationen der Werte für `i` und `j` bearbeiten, aber in anderer Reihenfolge. Die Ausgabe von

```
For j = 8 To 9
  For i = 1 To 3
    Print i & "/" & j,
  Next i
  Print  ' Zeilenumbruch
Next j
```

ist nämlich:

```
1/8  2/8  3/8
1/9  2/9  3/9
```

Vergleichen Sie das mit der Ausgabe weiter oben: Mit etwas Phantasie (und Kenntnis von Matrix-Operationen aus der Linearen Algebra) sehen

Sie hier eine transponierte Matrix, also eine Matrix, in der Spalten und Zeilen vertauscht wurden.

3. Funktionen und Subs

Mit `Sub`-Prozeduren und Funktionen untergliedern Sie eine komplexe Aufgabe in ihre Teilprobleme. Aber wann ist welche Methode die richtige?

- Funktionen verwenden Sie z. B. für Berechnungen, also etwa dann, wenn Sie eine mathematische Funktion benötigen. Auch „Berechnungen“ mit Strings, wie das Verbinden mehrerer Strings zu einem einzigen oder Veränderungen in Strings, sind Kandidaten für Funktionen.
- Sub-Prozeduren kennen Sie bereits als Makros: Jedes mit dem Makro-Rekorder aufgenommene Makro ist eine solche Sub-Prozedur. Sie können aber im VBA-Editor auch weitere Sub-Prozeduren programmieren und dann über den Call-Befehl aus anderen Makros heraus aufrufen.

Ein wesentlicher Unterschied zwischen Subs und Funktionen ist, dass Funktionen immer ein Ergebnis, den so genannten Rückgabewert, erzeugen, während Subs einfach die vorgesehene Aufgabe erledigen und sich dann (ohne Rückgabewert) beenden.

Gemeinsam ist beiden Unterprogrammarten, dass beim Aufruf einer Sub-Prozedur oder einer Funktion der Kontrollfluss innerhalb des aufrufenden Makros unterbrochen wird: Die Ausführung geht nun innerhalb der aufgerufenen Sub-Prozedur/Funktion weiter, bis diese beendet wird – dann geht es im aufrufenden Makro unmittelbar hinter dem Aufrufbefehl weiter.

3.1 Funktionen

Die klassischen Beispiele für VBA-Funktionen sind mathematische Funktionen. Ein ganz einfaches Beispiel wäre die Funktion f(x) = x+1, die zu jedem x den um 1 größeren Wert x+1 berechnet. In VBA programmieren Sie diese wie folgt:

```
Function f(x)
```

```
  f = x + 1
End Function
```

Funktionsdefinitionen sind immer so aufgebaut, dass hinter dem Schlüsselwort Function zunächst der Name der Funktion (hier: `f`) und dann in Klammern eine Liste aller Argumente (hier nur: `x`) steht. Sie können (und sollten) für das Argument `x` einen Typ angeben, dann sieht das Ganze wie folgt aus:

```
Function f(x as Integer)
  f = x + 1
End Function
```

Die Syntax erinnert (mit Absicht) an das Dimensionieren einer Variablen (`Dim x As Integer`). Eine Funktionsdefinition endet mit `End Function`, und zwischen diesen beiden Zeilen steht die Berechnungsvorschrift. In diesem einfachen Beispiel hat sie die Form `f = x + 1`. Die Angabe des Funktionsergebnisses erfolgt also immer in Form einer Zuweisung, wobei hier auf der linken Seite kein Variablenname, sondern der Name der Funktion selbst steht. Es ist auch möglich, mehrere solche Ergebniszuweisungen zu verwenden, z. B., wenn Sie mit Fallunterscheidungen arbeiten. Betrachten Sie dazu das folgende Beispiel:

```
Function mitFallunterscheidung (nWert As Integer)
  If nWert<0 Then
    mitFallunterscheidung = 0
  Else
    mitFallunterscheidung = nWert + 1
  End If
End Function
```

Das berechnet die folgende Funktion f:

$$f(x) = \begin{cases} x+1, & x >= 0 \\ 0, & x < 0 \end{cases}$$

Für Werte x >= 0 ist das Ergebnis also x+1 (wie in der ersten Funktion); und für Werte x<0 ist das Ergebnis 0.

Sie können auch die Fakultätsfunktion f(x) = x! = x (x-1) (x-2) … 2 . 1 mit folgender Funktionsdefinition erzeugen:


```
Function fakultaet(n)
  If n = 1 Then
    fakultaet = 1
  Else
    fakultaet = n * fakultaet (n-1)
  End If
End Function
```

Beachten Sie, dass diese Funktion rekursiv ist, d. h., sie ruft sich selbst auf – es gilt nämlich für die Fakultät n!: 1!=1 und n! = n (n-1)!.

Innerhalb eines Makros, das den Wert `n` abfragt und dann die Fakultät `n!` ausgibt, können Sie die neu definierte Funktion wie folgt verwenden:

```
Sub FakultaetBerechnen()
  Dim n,f As Integer
  Dim sAusgabe As String
  n = InputBox ("Geben Sie n ein.")
  f = fakultaet(n)
  sAusgabe = n & "! = " & f
  MsgBox sAusgabe
End Sub
```

Dieses Makro öffnet zunächst eine InputBox und fragt darin den Wert für `n` ab. Dann berechnet es die Fakultät, indem es die Funktion `fakultaet()` verwendet – dabei gibt es die Variable `n` als Argument an und speichert das Ergebnis der Funktionsberechnung im Rückgabewert `f`, also `f = fakultaet(n)`. Schließlich setzt das Makro in `sAusgabe` einen Ausgabetext zusammen und gibt ihn mit einer MsgBox aus. Haben Sie in einem Aufruf den Wert 4 eingegeben, erscheint als Ausgabe

```
4! = 24
```

Auch Funktionen, die mit Strings arbeiten, sind leicht erstellt. Betrachten Sie das folgende Beispiel, das einen String verdreifacht:

```
Function verdreifachen (s as String)
  verdreifachen = s & s & s
End Function
```

Ein Makro, das diese Funktion verwendet, könnte folgendermaßen aussehen:

```
Sub TestVerdreifachen()
  Dim s As String
  s = "Test"
  MsgBox verdreifachen(s)
End Sub
```

Wenn Sie `TestVerdreifachen()` aufrufen, erscheint eine MsgBox mit dem Inhalt „TestTestTest“.

3.1.1 Funktionen ohne Argument

In der Mathematik sind Funktionen ohne Argument Konstanten: Da es kein Argument gibt, kann das Ergebnis auch nicht von einem Argument abhängen, sondern muss immer gleich sein.

Solche konstanten Funktionen können Sie auch in VBA definieren, sie sind aber nicht sonderlich sinnvoll:

```
Function neun()
  neun = 9
End Function
```

Eine Funktion ohne Argument können Sie beim Aufruf wahlweise mit leeren Klammern schreiben, oder Sie lassen die Klammern gleich ganz weg (was keine gute Idee ist, weil dann der Funktionsaufruf wie die Verwendung einer Variablen aussieht):

```
Print neun()
 9
Print neun
 9
```

3.2 Sub-Prozeduren

3.2.1 Subs mit Parametern

Ähnlich wie Funktionen können auch Sub-Prozeduren mit Argumenten definiert werden.

3.3 Call by Reference und Call by Value

Wenn Sie eine Sub-Prozedur mit Parametern aufrufen und dabei als Argument eine Variable einsetzen, dann kann die Sub-Prozedur diese ursprüngliche Variable verändern – selbst, wenn innerhalb der Sub-Prozedur ein anderer Name für den Parameter verwendet wird. Ein Beispiel für dieses Phänomen ist folgender Code:

```
Sub AendereWert(n As Integer)
  Debug.Print "AendereWert aufgerufen"
  n = 99
End Sub

Sub Test()
  Dim nWert As Integer
  nWert = 3
  Debug.Print "nWert vor  AendereWert: " & nWert
  Call AendereWert(nWert)
  Debug.Print "nWert nach AendereWert: " & nWert
End Sub
```

Beim Aufruf der Sub-Prozedur `AendereWert` mit dem Argument `nWert` reicht VBA die Speicheradresse von `nWert` an `AendereWert` weiter – innerhalb von `AendereWert` sprechen Sie also über den Namen `n` exakt dieselbe Variable an wie in der Prozedur `Test` über den Namen `nWert`. Dass ein Ding zwei Namen hat, ist für viele Programmieranfänger ein verwirrendes Konzept. „Freundlicherweise“ gibt es auch das umgekehrte Phänomen: Unter demselben Namen können Sie, abhängig vom Ort, verschiedene Dinge ansprechen. Informatiker definieren den **Gültigkeitsbereich** (engl.: Scope) einer Variablen, der beschreibt, in welchen Code-Blöcken ein Variablenname ein und demselben Objekt (also: der oder den gleichen Speicherzellen) zugeordnet ist.

3.4 Ereignisprozeduren

(Arbeiten mit Forms)

4. Felder

Variablen sind sehr nützlich, um Werte zwischen zu speichern und damit zu rechnen – aber was tun Sie, wenn Sie sehr viele Variablen benötigen? Wollen Sie z. B. die ersten 15 Zellen der Spalte A auslesen, addieren und anschließend in Zelle A16 schreiben, könnten Sie das mit folgendem Code tun:

```
Sub AddierenUmstaendlich()
  Dim Wert1, Wert2, Wert3, Wert4, Wert5 As Integer
  Dim Wert6, Wert7, Wert8, Wert9, Wert10 As Integer
  Dim Wert11, Wert12, Wert13, Wert14, Wert15 As Integer
  Dim Ergebnis As Integer
  Wert1  = Cells(1,1)  :  Wert2  = Cells(2,1)
  Wert3  = Cells(3,1)  :  Wert4  = Cells(4,1)
  Wert5  = Cells(5,1)  :  Wert6  = Cells(6,1)
  Wert7  = Cells(7,1)  :  Wert8  = Cells(8,1)
  Wert9  = Cells(9,1)  :  Wert10 = Cells(10,1)
  Wert11 = Cells(11,1) :  Wert12 = Cells(12,1)
  Wert13 = Cells(13,1) :  Wert14 = Cells(14,1)
  Wert15 = Cells(15,1)
  Ergebnis = Wert1 + Wert2 + Wert3 + Wert4 + Wert5 + Wert6 _
    + Wert7 + Wert8 + Wert9 + Wert10 + Wert11 + Wert12 _
    + Wert13 + Wert14 + Wert15
  Cells(16,1) = Ergebnis
End Sub
```

Das funktioniert, ist aber – wie der Name des Makros schon verspricht – sehr umständlich. Hilfreicher wäre es, wenn Sie `Wert1`, `Wert2` usw. mit Hilfe einer Zählvariablen `i` ansprechen könnten – also `Wert_i`, wobei Sie mit jedem `i` eine andere, zugehörige Variable `Wert_i` ansprechen.

VBA bietet Ihnen hier das Konzept der **Felder**: Ein Feld besteht aus vielen Variablen, die Sie über einen gemeinsamen Namen (den Feldnamen) und eine Indexnummer ansprechen können. Die Syntax für den Zugriff auf eine einzelne Variable ist dann `Feldname(Indexnummer)`, im Beispiel etwa `Wert(i)`. für die `i`-te Variable im Feld `Wert`.

Anders als Variablen, die Sie nicht zwingend mit `Dim` deklarieren müssen (aber sollten), können Sie mit Feldern erst arbeiten, wenn Sie VBA explizit mitgeteilt haben, dass Sie ein Feld verwenden wollen – und wie groß es ist: Das Festlegen der Größe heißt auch Dimensionieren des Felds.

4.1 Felder dimensionieren (Dim)

Bevor Sie ein Feld verwenden können, legen Sie mit `Dim` seine Größe (und meist auch den Typ der enthaltenen Variablen) fest. Um etwa zu obigem Addierbeispiel ein Feld namens `Wert` mit 15 Elementen vom Typ `Integer` zu erzeugen, benutzen Sie folgenden Befehl:

```
Dim Wert(1 To 15) As Integer
```

Das `Dim`-Kommando ähnelt dem `Dim`-Befehl, mit dem Sie den Typ einer einzelnen Variablen festlegen (`Dim n As Integer`), der Unterschied liegt in der in Klammern stehenden Größenangabe `1 To 15`, die festlegt, dass es ein Feld mit Indexnummern 1 bis 15 wird.

Mit `Dim` legen Sie hier also fest, wie groß das Feld ist – Sie dimensionieren es (bestimmen seine Dimension), und daher kommt auch ursprünglich der `Dim`-Befehl: In älteren BASIC-Versionen diente er nur zum Festlegen von Feldgrößen, während einfache Variablen gar nicht deklariert wurden.

Gleich nach dem Dimensionieren dürfen Sie auf die Feldinhalte zugreifen, und das Addierprogramm von oben sieht mit Hilfe von zwei kleinen `For`-Schleifen nun wie folgt aus:

```
Sub AddierenBesser()
  Dim Wert(1 To 15) As Integer
  Dim i, Ergebnis As Integer
  For i = 1 To 15
    Wert(i) = Cells(i,1)
  Next i
  Ergebnis = 0
  For i = 1 To 15
    Ergebnis = Ergebnis + Wert(i)
  Next i
  Cells(16,1) = Ergebnis
End Sub
```

4.1.1 Mehrdimensionale Felder

Felder dürfen auch mehr als eine Dimension haben, also über mehrere Indexnummern ansprechbar sein: Die bisher vorgestellten Felder heißen eindimensional. Ein eindimensionales Feld `Wert`, das Sie über

```
Dim Wert(1 To 10) As Integer
```

definiert haben und dessen Elemente Sie einzeln als `Wert(1)`, `Wert(2)`, …, `Wert(10)` ansprechen, entspricht mathematisch einem Vektor

$\mathbf{w} = (w_1, w_2, w_3, w_4, w_5, w_6, w_7, w_8, w_9, w_{10})$

den Sie sich wahlweise in Zeilen- oder Spaltennotation vorstellen können. So wie Sie in der Mathematik von eindimensionalen Vektoren zu zweidimensionalen Matrizen übergehen können, erzeugen Sie auch in VBA ein zweidimensionales Feld, indem Sie einen weiteren Index einführen:

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

definiert ein Feld, das 15 (3 x 5) einzelne Elemente besitzt, die Sie sich wie folgt in einer Matrix angeordnet denken können:

M(1,1)	M(1,2)	M(1,3)	M(1,4)	M(1,5)
M(2,1)	M(2,2)	M(2,3)	M(2,4)	M(2,5)
M(3,1)	M(3,2)	M(3,3)	M(3,4)	M(3,5)

4.1.2 Das „Feld“ Cells()

Sicher ist Ihnen aufgefallen, dass der Zugriff auf zweidimensionale Felder exakt so funktioniert wie der Zugriff auf Zelleninhalte der Excel-Tabelle. So kopiert beispielsweise der Code

```
Sub CellsKopieren()
  Dim Werte(1 To 10, 1 To 10) As Variant
  For i = 1 To 10
    For j = 1 To 10
      Werte(i,j) = Cells(i,j)
    Next j
  Next i
End Sub
```

die ersten zehn Spalten und Zeilen in ein passend dimensioniertes Feld. Cells ist aber dennoch kein Feld, sondern ein Objekt (mit vielen Eigenschaften eines zweidimensionalen Felds). Den Unterschied erkennen Sie u. a. daran, dass Sie auf `Cells` Methoden anwenden können, etwa das Ändern der Schriftart:

```
Cells(1,1).Font.Size=10
```


Mit einem „normalen“ Feld funktioniert das nicht; ein Aufruf der Form `Werte(1,1).Font.Size = 10` würde eine Fehlermeldung verursachen („Laufzeitfehler ‚424’: Objekt erforderlich“).

4.2 Felder neu dimensionieren (ReDim)

Ddd

4.2.1 Feldinhalte bewahren (Preserve)

Bla

5. Beispielaufgaben und Musterlösungen

Aufgabe 1 (Folie 6, 2008/12/09)

Schreiben Sie ein kleines Makro, das folgende Aufgabe erfüllt:

- Es liest die Werte in den Zellen A1, A2 und A3
- Wenn in A1 das Wort „ADD“ steht, dann addiert es A2 und A3 (und speichert das Ergebnis in einer neuen Variable)
- Wenn in A1 das Wort „SUB“ steht, dann subtrahiert es A3 von A2 – bildet also A2-A3 – (und speichert das Ergebnis in einer neuen Variable)
- Das Ergebnis der Berechnung landet in Zelle A4
- Hinweis: Nutzen Sie `Variable = Cells (......)`

Lösung:

```
Sub Aufgabe1()
  Dim sOperation As String
  Dim nWert1, nWert2, nErgebnis As Integer
  nWert1 = Cells(2,1)
  nWert2 = Cells(3,1)
  Operator = Cells(1,1)
  If Operator = "ADD" Then
    nErgebnis = nWert1 + nWert2
  ElseIf Operator = "SUB" Then
    nErgebnis = nWert1 - nWert2
  End If
  Cells(4,1) = nErgebnis
End Sub
```

Aufgabe 2 (Folie 17, 2008/12/11)

Lesen Sie aus A1, A2 und A3 je eine Zahl in eine Variable ein. A1 ist der erste, A2 der zweite Operand einer Modulooperation – rechnen Sie also A1 Mod A2 aus.
Wenn der Inhalt von A3 gerade ist, soll die Ausgabe in einer Messagebox erfolgen, anderenfalls in Zelle A4. Die Ausgabe soll in jedem Fall

vernünftig lesbar sein, z.B. in der Form „12 Mod 5 ist 2“. Verwenden Sie sprechende Bezeichner für Ihre Variablen, z. B. Zahl1.

Lösung:

```
Sub ModuloRechnen()
  Dim nWert1, nWert2, nWert3, nErgebnis As Integer
  Dim sAusgabe As String
  nWert1 = Cells(1,1)
  nWert2 = Cells(2,1)
  nWert3 = Cells(3,1)
  nErgebnis = nWert1 Mod nWert2
  If nWert3 Mod 2 = 0 Then
     ' Fall 1: nWert3 ist gerade
    MsgBox nAusgabe
  Else
     ' Fall 2: nWert3 ist ungerade
    Cells(4,1) = nAusgabe
  End If
End Sub
```

Aufgabe 3 (Folie 18, 2008/12/11)

Schreiben Sie einen Makro, der den Inhalt der Zellen A1 bis A10 im Arbeitsblatt 2 in Arbeitsblatt 1 in den Zellen A1 bis J1 anzeigt, wenn diese durch 13 ohne Rest teilbar sind und größer als 8 sind. Sollte die Bedingung nicht erfüllt sein, so soll die jeweilige Zelle den Wert "Unpassender Wert" bekommen.
Tipp: Auf Zellen in anderem Arbeitsblatt zugreifen mit Worksheets(Nummer).Cells(...)

Lösung:

```
Sub Aufgabe3
  Dim nZaehler As Integer
  For nZaehler = 1 To 10
    nWert = Cells(1, nZaehler)  ' Wert auslesen
    If (nWert > 8) And (nWert Mod 13 = 0) Then
      ' Beide Bedingungen erfüllt, also Wert in die Zeile
      ' auf dem 2. Arbeitsblatt schreiben
      Worksheets(2).Cells(nZaehler, 1) = nWert
    Else
      Worksheets(2).Cells(nZaehler, 1) ="Unpassender Wert"
    End If
  Next
End Sub
```

Aufgabe 4 (Folie 6, 2008/12/16)

Schreiben Sie ein VBA-Makroprogramm, das zehn Zahlen aus der Tabelle (A1 bis A10) in ein 10-elementiges Feld liest, dieses Feld mit Bubblesort sortiert und das Ergebnis in die Spalte daneben (also in die Zellen B1 bis B10) schreibt.

Lösung:

```
Sub BubbleSort()
  Dim i, j As Integer
  Dim nFeld(10) As Integer

  ' Zahlen aus Zellen A1-A10 lesen
  For i = 1 To 10
    nFeld(i) = Cells(i,1) ' Zeile i, Spalte 1
  Next

  ' Jetzt sortieren
  For i = 1 To 9
    For j = 1 To 9
      If nFeld(j) > nFeld(j+1) Then
        nTemp = nFeld(j)
        nFeld(j) = nFeld(j+1)
        nFeld(j+1) = nTemp
      EndIf
    Next j
  Next i

  ' Sortierte Zahlen nach B1-B10 schreiben
  For i = 1 To 10
    Cells(i,2) = nFeld(i) ' Zeile i, Spalte 2
  Next
End Sub
```