

Grundkurs für Excel – VBA – Part II

Nico Ludwig

Themen

- Schleifen
- Exkurs: Arrays
- Der Direktbereich
- Mit Zeichenketten arbeiten
- Zelleigenschaften
- Prozeduren und Funktionen
- Call by Value und Call by Reference

Einführung Schleifen – Teil I

- Neben Sequenzen und Verzweigungen kennt VBA auch die Programmierung von Schleifen.
- Mit Schleifen können wir Programmteile in VBA wiederholt ausführen.
- Beispiel: der Benutzer soll eine Zahl eingeben, von der unser Programm die Quadratwurzel ziehen soll.
 - (1) Die Eingabe einer Dezimalzahl (reellen Zahl) vom VBA-Typ Double erfolgt über einen Eingabedialog.
 - (2) Wir dürfen nur positive Double als Eingabe akzeptieren!
 - Quadratwurzeln können nur von positiven reellen Zahlen berechnet werden!
 - (3) Wenn der Benutzer eine negative Zahl eingibt, soll das Programm nicht rechnen, sondern eine neue Eingabe verlangen.
 - (4) Wir vereinfachen die Benutzereingabe, indem wir solange Eingaben entgegennehmen, bis eine positive Zahl eingegeben wird.
- Naja, aber wie programmieren wir das?

Einführung Schleifen – Teil II

- Das entscheidende Wort bei der Lösungsbeschreibung war "solange".
- Wir wollen einen Vorgang/Programmteil solange eine Bedingung erfüllt ist wiederholen.
- ... und das ist genau die Beschreibung einer Schleife in menschlicher Sprache.
- In VBA programmieren wir Schleifen mit **While...Wend**-Anweisungen (kurz: **While**-Anweisung).

While-Schleifen

- Die Prozedur *Wurzel()* kann man so programmieren:

```
Sub Wurzel()  
    Dim eingabeWert As String  
    Dim zahl As Double  
    eingabeWert = InputBox("Aus welcher Zahl soll die Quadratwurzel gezogen werden?")  
    zahl = CDb1(eingabeWert)  
    While zahl < 0 ' Die Schleife beginnt hier ...  
        eingabeWert = InputBox("Aus welcher Zahl soll die Quadratwurzel gezogen werden?")  
        zahl = CDb1(eingabeWert)  
    Wend ' ... und endet hier.  
    [A1] = Sqr(quadratZahl)  
End Sub
```

- Im Zentrum steht natürlich die **While**-Schleife, aber wir müssen hier noch mehr klären.
 - Diese **While**-Schleife blockiert den Ablauf solange, bis die eingegebene Zahl positiv ist.

Wurzel() – Strings, Doubles und CDbI()

- Der erste Teil von *Wurzel()*:

```
Sub Wurzel()  
    Dim eingabeWert As String ' (1)  
    Dim zahl As Double       ' (2)  
    eingabeWert = InputBox("Aus welcher Zahl soll die Quadratwurzel gezogen werden?")  
    zahl = CDbI(eingabeWert)  
    ...
```

- In (1) und (2) definieren wir die Variablen *eingabeWert* und *zahl* vom Typ *String* und *Double*.
- Der Aufruf von *InputBox()* liefert zwar eine Zahl, aber diese Zahl steht in einem Text!
- Daher müssen wir das Ergebnis von *InputBox()* erstmal in der *String*-Variablen *eingabeWert* ablegen.
 - Merke: In Variablen vom Typ *String* können wir Text speichern.
- Um mit der Zahl "im Text" *eingabeWert* zu rechnen, müssen wir die Zahl aus dem Text "extrahieren".
- Die Extraktion machen wir mit der neuen Funktion *CDbI()* (engl. convert double).
- *CDbI()* übergeben wir einen *String* als Argument und erhalten den extrahierten *Double*-Wert zurück.
- Das Ergebnis von *CDbI()* speichern wir direkt in der *Double*-Variablen *zahl*.

While-Bedingung und Wiederholung

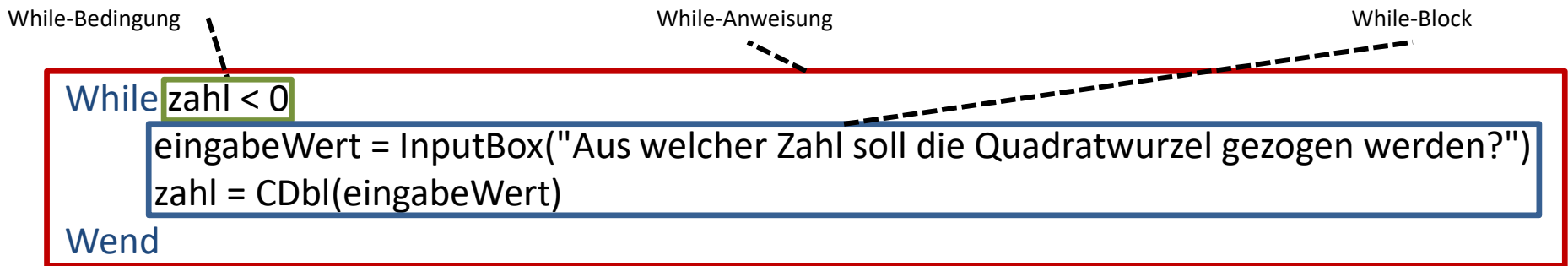
- Der zweite Teil von *Wurzel()*:

```
' ...  
While zahl < 0 ' (3) Die Schleife beginnt hier ...  
    eingabeWert = InputBox("Aus welcher Zahl soll die Quadratwurzel gezogen werden?")  
    zahl = CDBl(eingabeWert)  
Wend ' (4) ... und endet hier.  
[A1] = Sqr(quadratZahl) ' (5)  
End Sub
```

- In (3) wird die Schleifenbedingung geprüft, also ist die eingegebene Zahl (*zahl*) kleiner als 0?
 - Trifft die Bedingung zu (*zahl* ist negativ), wird der Schleifenblock betreten und der Benutzer erneut zur Eingabe von *zahl* aufgefordert!
 - Also es muss wieder ein *String* entgegengenommen und daraus ein *Double* extrahiert werden.
 - Das ist ein ganz wichtiger Aspekt: die Variablen *eingabeWert* und *zahl* werden im *While*-Block geändert!
 - Danach wird die Schleifenbedingung für *zahl* erneut geprüft, d.h. es gibt einen Sprung von (4) nach (3).
 - Ist *zahl* immer noch kleiner als 0, wird der Schleifenblock erneut betreten.
 - Das Spiel wiederholt sich solange, bis die Eingabe des Benutzers (*zahl*) nicht mehr negativ ist.
- Ist *zahl* aber positiv (die Schleifenbedingung trifft nicht zu), wird der Schleifenblock nicht betreten und das Programm in (5) fortgesetzt.
- In (5) wird mit der Funktion *Sqr()* (engl. Square root) die Quadratwurzel von *zahl* berechnet und der Zelle A1 zugewiesen.

Die Syntax der While-Schleife

- Die **While**-Schleife ist einfach aufgebaut:



- Die **While**-Bedingung kann genauso formuliert werden wie eine **If**-Bedingung.
 - Die Bedingung ist einfach ein Ausdruck, der einen **Boolean**-Wert (**True/False**) zurückgibt.
 - Neben Vergleichen können wir auch Prädikate wie `IsEmpty()` und **Or/And/Not**-Verknüpfungen verwenden.
- Im Unterschied zu **If** wird der **While**-Block eben solange die Bedingung zutrifft wiederholt.
- Die **While**-Bedingung wird geprüft, bevor der Block ausgeführt wird.
 - D.h. es ist möglich, dass der **While**-Block niemals ausgeführt wird.
 - Der Block wird z.B. niemals ausgeführt, wenn der Benutzer bei der ersten Eingabe schon eine positive Zahl angibt.
- Wir können **While**-Schleifen und **If**-Anweisungen problemlos verschachteln.

Schleifentypen

- Wie eben besprochen, wird die **While**-Bedingung geprüft, bevor der Block ausgeführt wird.
- Man sagt, die **While**-Schleife ist eine kopfgesteuerte oder abweisende Schleife.
- Manchmal ist eine andere Art von Schleife besser geeignet als die **While**-Schleife.
 - Z.B. dann, wenn wir die Schleifenbedingung erst nach dem Durchlaufen des Schleifenblocks prüfen wollen.
- Eine Schleife, der die Blockausführung vor der Bedingungsprüfung vornimmt heißt fußgesteuerte Schleife.
 - Bzw. spricht man auch von einer nicht-abweisenden Schleife.
- VBA kennt die **Do...Loop**-Schleife (kurz **Do**-Schleife) als fußgesteuerte Schleife.

Do-Schleifen

- Die fußgesteuerte **Do**-Schleife kann unsere *Wurzel()*-Prozedur stark vereinfachen:

```
Sub Wurzel()  
    Dim eingabeWert As String  
    Dim zahl As Double  
    Do  
        eingabeWert = InputBox("Aus welcher Zahl soll die Quadratwurzel gezogen werden?")  
        zahl = CDbI(eingabeWert)  
    Loop While zahl < 0 ' Die Bedingung wird erst hier überprüft!  
    [A1] = Sqr(zahl)  
End Sub
```

- Auch diese **Do**-Schleife blockiert den Ablauf solange, bis die eingegebene Zahl positiv ist.
 - Durch die Fusssteuerung müssen aber die Funktionen *InputBox()* und *CDbl()* nur einmal im Programm stehen!
 - Der Effekt: Die Prozedur kann so viel kompakter ausfallen und ist leichter verständlich.

Die Syntax der Do-Schleife

- Die Do-Schleife ist auch einfach aufgebaut:

```
Do
    eingabeWert = InputBox("Aus welcher Zahl soll die Quadratwurzel gezogen werden?")
    zahl = CDbI(eingabeWert)
Loop While zahl < 0
```

Do-Bedingung

Do-Anweisung

Do-Block

- Hier wird der Do-Block immer ein erstes Mal ausgeführt!
 - Da ist für unsere Aufgabe auch sinnvoll, denn der Benutzer muss ja erstmal eine Zahl zur Prüfung eingeben!
- Die Do-Bedingung kann genauso formuliert werden wie eine If- oder While-Bedingung.
- Wir können Do-Schleifen, While-Schleifen und If-Anweisungen problemlos verschachteln.

Berechnungen mit Schleifen Wiederholen – Teil I

- Das nächste Beispiel beleuchtet die Verarbeitung von vielen Daten mit Schleifen.
- Außer ein Programm bedingt zu blockieren, ist die Aufgabe von Schleifen Berechnungen zu wiederholen.
- Das passt auch zu der Idee hinter Excel, immerhin können sehr viele Daten in Tabellen gespeichert werden!
- Wir besprechen jetzt an einem einfachen Beispiel, wie die Datenverarbeitung mit Excels VBA funktioniert.
- Die Manipulation vieler Zellen ist eine klassische Aufgabe der Datenverarbeitung.
- Beispiel: Die Zahlen von 1 bis 10 sollen in die Zellen A1 bis A10 geschrieben werden.
 - Und das können wir mit einer Schleife formulieren!

Berechnungen mit Schleifen Wiederholen – Teil II

- Die Zahlen von 1 bis 10 kann man mit dieser Prozedur in die Zellen A1 bis A10 schreiben:

```
Sub Zählen()  
    Dim maxAnzahl, zähler As Integer  
    zähler = 1  
    maxAnzahl = 10  
    While zähler <= maxAnzahl  
        Cells(zähler, 1) = zähler  
        zähler = zähler + 1  
    Wend  
End Sub
```

	A
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10

- Ja, da müssen wir allerdings noch einige Aspekte in *Zählen()* erklären!
 - Aber der Code enthält auch bekannte Dinge wie die *While*-Schleife und Variablendefinitionen.

Berechnungen mit Schleifen Wiederholen – Teil III

- Das Prinzip: Die Adressen der Zellen in die wir schreiben, entwickeln sich mit den Werten, die wir schreiben wollen.
 - Die Idee dabei ist immer den nächst höheren Wert (ausgehend von 1) in die nächste Zelle (ausgehend von A1) zu schreiben.
 - Also wird müssen die Werte 1 – 10 in die Zellen A1 – A10 schreiben.
- Um das zu schaffen, müssen wir Zellkoordinaten programmatisch angeben können. – Warum?
- Bisher können wir nur Zellen mit der Syntax [A1] angeben, das genügt hier aber nicht!
 - [A1] ist nämlich ein konstanter Wert. D.h. wir können die Angabe "1" in dem Ausdruck [A1] gar nicht verändern!
- Mit der Funktion *Cells()* bietet VBA einen Weg, um Zellen mit variablen Koordinaten anzugeben:

[A1] = "Hallo Welt"



```
Dim koordZeile, koordSpalte As Integer
koordZeile = 1
koordSpalte = 1
Cells(koordZeile, koordSpalte) = "Hallo Welt"
```

- Die Verwendung von *Cells()* ist aufwendiger als die der []-Syntax, aber flexibler, da wir mit variablen Koordinaten arbeiten können.
- Wenn wir Variablen verwenden, um auf "Speicherstellen" wie Zellen zuzugreifen, spricht man von variabler Adressierung.

Berechnungen mit Schleifen Wiederholen – Teil IV

- Bisher haben Variablen nur einfach irgendwelche Eingabewerte (z.B. vom Eingabedialog) zwischengespeichert.
- *Zählen()* machen aber zum ersten mal eine Berechnung auf einer Variablen, hier *zähler*:

```
Sub Zählen()  
    Dim maxAnzahl, zähler As Integer  
    zähler = 1  
    maxAnzahl = 10  
    While zähler <= maxAnzahl  
        Cells(zähler, 1) = zähler  
        zähler = zähler + 1 ' ?  
    Wend  
End Sub
```

- Die Berechnung *zähler = zähler + 1* sieht merkwürdig aus. – Ein Wert kann doch nicht gleich demselben Wert plus 1 sein!
- Wir müssen diese Anweisung allerdings imperativ (lat. imperare, etwas befehlen) und nicht mathematisch lesen.
- Für VBA als imperative Programmiersprache steht da diese Anweisung: Der neue Wert von *zähler* ist der alte Wert von *zähler* plus 1.
- Wenn ein Wert (z.B. in Schleifen) erhöht/erniedrigt wird, nennt man das inkrementieren/dekrementieren (lat. incrementare, etwas vergrößern).

Berechnungen mit Schleifen Wiederholen – Teil V

- In der Schleife von *Zählen()* arbeiten die Inkrementierung von *zähler* und die variable Adressierung zusammen.

```
Sub Zählen()  
    Dim maxAnzahl, zähler As Integer  
    zähler = 1  
    maxAnzahl = 10  
    While zähler <= maxAnzahl  
        Cells(zähler, 1) = zähler ' (1)  
        zähler = zähler + 1 ' (2)  
    Wend ' (3)  
End Sub
```

- (1) Wir schreiben in die variabel adressierte Zelle *Cell(zähler, 1)* den Wert von zähler.
 - (2) Wir inkrementieren den *zähler* (um 1).
 - (3) Die Schleife geht Wiederholung und "rotiert" solange, bis *zähler* letztendlich den Wert *maxAnzahl* (10) erreicht.
- Es ist sehr wichtig, dass *zähler* in der Schleife verändert wird! – Sonst haben wir einen ernsten Programmierfehler!

Nicht-abbrechende Schleifen

- Ein typischer Fehler beim Programmieren mit Schleifen ist das falsche/fehlende Ändern der Prüfvariablen:

```
Sub Zählen() ' Hier ist ein Fehler drin!  
    Dim maxAnzahl, zähler As Integer  
    zähler = 1  
    maxAnzahl = 10  
    While zähler <= maxAnzahl  
        Cells(zähler, 1) = zähler  
    Wend  
End Sub
```

- Wenn wir diese Prozedur ausführen, "friert" Excel ein, d.h. es reagiert nicht mehr auf Benutzereingaben.
 - Im **While**-Block wird die Variable *zähler* nicht aktualisiert und gleichzeitig erwartet die Schleifenbedingung einen neuen Wert für *zähler*.
 - Da *zähler* immer den selben Wert hat (also immer 1) ist *zähler* immer \leq *maxAnzahl* und die Schleife endet nie!
 - Wir haben hier eine nicht-abbrechende Schleife (engl. infinity loop), die das Programm dauerhaft blockiert!
- Fehler können immer mal passieren: Excel beendet nicht-abbrechende Schleifen mit Druck auf **Strg-Pause**.

Zählschleifen

- Bei den bisherigen Schleifen wurden Anweisungen wiederholt solange eine Bedingung zutrifft.
- VBA kennt auch sogenannte Zählschleifen, dort ist die Zahl der Wiederholungen vorher bekannt.
 - Die Prozedur `Zählen()`, mit der Erzeugung einer Liste ganzer Zahlen von 1 – 10 ist so ein Beispiel, es sind 10 feste Wiederholungen!
- Zählschleifen werden mit der `For...Next`-Schleife (kurz `For`-Schleife) programmiert.
 - Formulieren wir also `Zählen()` mit einer `For`-Schleife neu (`ZählenFor()`).

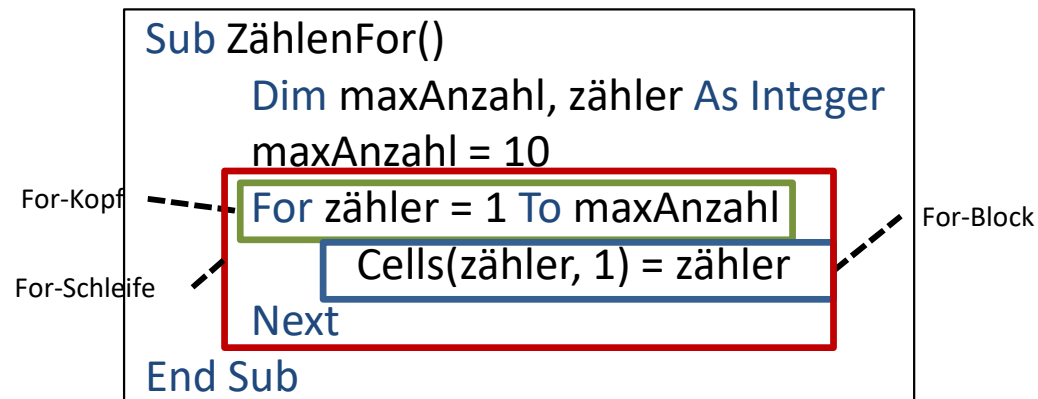
```
Sub Zählen()  
    Dim maxAnzahl, zähler As Integer  
    zähler = 1  
    maxAnzahl = 10  
    While zähler <= maxAnzahl  
        Cells(zähler, 1) = zähler  
        zähler = zähler + 1  
    Wend  
End Sub
```



```
Sub ZählenFor()  
    Dim maxAnzahl, zähler As Integer  
    maxAnzahl = 10  
    For zähler = 1 To maxAnzahl  
        Cells(zähler, 1) = zähler  
    Next  
End Sub
```

Die For-Schleife

- Die **For**-Schleife ist etwas komplexer aufgebaut als die anderen Schleifen:



Tipp: Wenn die Wiederholungsanzahl bekannt ist, sollte die **For**-Schleife bevorzugt werden! Ihre Verwendung führt zu gut lesbarem Code und Infinity Loops können vermieden werden.

- Wie die **While**-Schleife ist die **For**-Schleife kopfgesteuert, sie ist also eine abweisende Schleife.
- Der Kopf der **For**-Schleife beschreibt diesmal aber keine Bedingung, sondern eine Zählanweisung.
 - Die Zählanweisung beschreibt eine Zählvariable (*zähler*) und setzt einen Startwert (1), dann gibt sie nach dem **To**-Schlüsselwort den Endwert der Zählvariablen an.
 - Mit dieser Zählanweisung wird die Zählvariable sooft um eins inkrementiert und der Schleifenblock wiederholt, bis der Endwert erreicht ist.
 - In der Zählanweisung können wir Zusätzlich mit den **Step**-Schlüsselwort einen anderen Inkrementwert als eins oder auch einen Dekrementwert angeben.
- Der **For**-Block bietet alle Möglichkeiten der anderen Schleifen, zusätzlich können wir auf den aktuellen Wert der Zählvariable zugreifen.
- Der **For**-Block wird mit dem **Next**-Schlüsselwort beendet.
- Zählvariable, Endwert und Inkrementwert sind fast immer vom Typ **Integer**, andere Typen sind in begrenztem Umfang zulässig.
 - Wesentlich ist, dass der Typ eben "abzählbar" sein muss.

For-Schleifen verschachteln – Teil I

- [illegible]

[illegible]

For-Schleifen verschachteln – Teil II

- Kurzum, der Code um das zu erreichen sieht so aus:

```
Sub VerschachtelteForSchleifen()  
    Dim zeilenAnzahl, spaltenAnzahl As Integer  
    zeilenAnzahl = 10  
    spaltenAnzahl = 10 ' entspricht Spalte J  
    For i = 1 To zeilenAnzahl  
        For j = 1 To spaltenAnzahl  
            Cells(i, j) = 0  
        Next  
    Next  
End Sub
```

	A	B	C	D	E	F	G	H	I	J
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0

- Der Code sieht raffiniert aus, ist aber recht gut verständlich.

For-Schleifen verschachteln – Teil III

```
Dim zeilenAnzahl, spaltenAnzahl As Integer
zeilenAnzahl = 10
spaltenAnzahl = 10 ' entspricht Spalte J
For i = 1 To zeilenAnzahl ' äußere Schleife
    For j = 1 To spaltenAnzahl ' innere Schleife
        Cells(i, j) = 0
    Next
Next
```

- So arbeitet der Code:
 - Die äußere Schleife generiert die variable Koordinate i, für die Zeilennummer, maximal *zeilenAnzahl*-mal.
 - Die innere Schleife generiert die variable Koordinate j, für die Spaltennummer, maximal *spaltenAnzahl*-mal.
 - Die Schleifen erzeugen so in ihrer Verschachtelung die Koordinaten "i, j" für jede Zelle in A1:J10 und "füttern" damit die Funktion *Cells()*.
 - Dann wird der adressierten Zelle der Wert 0 einfach zugewiesen.
 - Die innere Schleife bearbeitet so die zehn Spalten einer Zeile.
 - Die äußere Schleife "schiebt" die innere Schleife auf die nächste Zeile und die innere Schleife wird wiederholt.
 - Wenn Zählschleifen verwendet werden, um Koordinaten zu erzeugen, haben die Zählvariablen oft kurze Namen, wie *i, j, k* usw.

For-Schleifen verschachteln – Teil IV

- Anschaulich arbeitet die Prozedur auf den Zellbereich so:

	A	B	C	D	E	F	G	H	I	J
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0

→ innere Schleife (j-Koordinate)
→ äußere Schleife (i-Koordinate)

Exkurs: Arrays – Teil I

- VBA kennt eine speziell Variante von Variablen, die mehr als einen Wert speichern können.
- Es handelt sich demnach um Variablen, die ein ganzes Feld an Werten speichern können.
- Solche Variablen werden daher als Feld, oder im Englischen als Array bezeichnet.
- In der Programmierpraxis ist die Bezeichnung "Array" aber gängiger!

Exkurs: Arrays – Teil II

- Beispiel: Der Benutzer soll 10 Zahlen eingeben, ihr Durchschnittswert soll in A1 gespeichert werden.
- Das kann man mit einem Array so programmieren:

```
Sub ArrayDurchschnitt()  
    Dim alleWerte(10) As Double ' Double-Array mit 10 Elementen  
  
    For i = 1 To UBound(alleWerte)  
        alleWerte(i) = CDbI(InputBox("Bitte Zahl eingeben"))  
    Next  
    For i = 1 To UBound(alleWerte)  
        summe = summe + alleWerte(i)  
    Next  
    [A1] = "Ergebnis: " & summe / UBound(alleWerte) ' Durchschnitt  
End Sub
```

- *ArrayDurchschnitt()* enthält wieder bekannte aber auch jede Menge neue Elemente.

Exkurs: Arrays – Teil III

```
Dim alleWerte(10) As Double
```

- Anfangs wird die Variable *alleWerte* als Double-Array definiert.
 - Die Definition unterscheidet sich von einer "einfachen" Variablen, da wir in Klammern die Anzahl der Elemente des Arrays angeben.

- Das Array *alleWerte* sieht im Speicher zunächst so aus:

0.0 ₁	0.0 ₂	0.0 ₃	0.0 ₄	0.0 ₅	0.0 ₆	0.0 ₇	0.0 ₈	0.0 ₉	0.0 ₁₀
<i>alleWerte</i>									

- *alleWert* hat also zehn Elemente (oder "Fächer"/"Slots"), die je einen Double als Wert aufnehmen können.
 - Jedes einzelne Element kann mit seinem Index, das ist seine "Platznummer" im Array angesprochen werden.
 - Die Indices in einem Array haben die Werte von 1 (erstes Element) bis Arraygröße (letztes Element).
 - Initial sind alle Elements mit dem Wert 0.0 belegt.
- Und dieses Array füllen wir jetzt mit Werten mit einer For-Schleife.

Exkurs: Arrays – Teil IV

- Die **For**-Schleife füllt nun das Array auf, in dem vom Benutzer 10 Zahlen abgefragt werden.

```
For i = 1 To UBound(alleWerte)
    alleWerte(i) = CDBl(InputBox("Bitte Zahl eingeben"))
Next
```

Tipp: Insbes. in Zählschleifen werden für Zählvariablen für Array-Indices eigentlich immer kurze Namen, wie i, j, k usw.

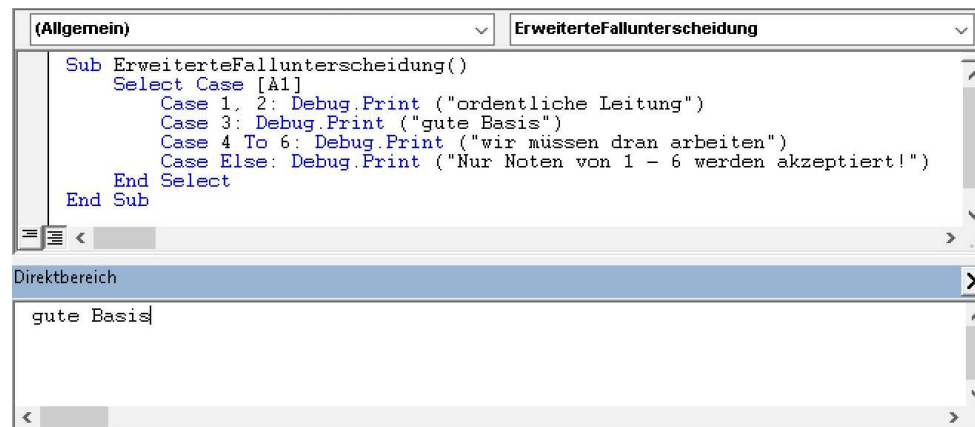
- Arraygröße und Zugriff:
 - Mit dem Schlüsselwort **UBound()** wird die Größe eines Arrays, also die Anzahl seiner Elemente ermittelt.
 - **UBound(alleWerte)** liefert in unserem Fall den Wert 10.
 - Der Ausdruck **alleWerte(i)** liefert uns den Wert des Elementes an der "Stelle i" im Array
 - => Mit **UBound()** erhalten wir die Größe des Arrays, also den Endwert der **For**-Schleife.
 - => Mit dem fortlaufenden Zähler *i* können wir auf jedes einzelne Element des Arrays zugreifen, bzw. schreiben.
 - => Bei jedem Schleifendurchlauf wird der Benutzer per Eingabedialog nach einer Zahl gefragt, die im jew. Array-Element gespeichert wird.
- Nach Eingabe der zehn Zahlen könnten in *alleWerte* etwa diese Zahlen gespeichert sein:

22 ₁	9.37 ₂	31.2 ₃	27 ₄	87.4 ₅	63 ₆	18.5 ₇	56 ₈	30 ₉	7.03 ₁₀
-----------------	-------------------	-------------------	-----------------	-------------------	-----------------	-------------------	-----------------	-----------------	--------------------

alleWerte

Der Direktbereich – Teil I

- Programmausgaben können auch in den sogenannten VBA-Direktbereich ausgegeben werden.
 - Die Idee ist dabei, dass diese Ausgabe nicht störend und auffällig auf Dialogen Zellen erfolgt, sondern im Hintergrund.
 - Der Direktbereich ist nämlich nur in der VBA-IDE sichtbar und nirgends in Excel selbst.
- Wir verwenden die Funktion `Debug.Print()`, um Informationen in den Direktbereich zu schreiben:



- Die Hintergrundausgabe von Zusatzinformationen durch ein Programm wird manchmal Logging (von engl. log "protokollieren") genannt.
- Logging ist eine sehr nützliche Funktion, da wir bequem Informationen ausgeben können, ohne immer Haltepunkte zu verwenden.
- Logging erlaubt es die "Entwicklung" von Daten während des Programmablaufes zu beobachten.

Der Direktbereich – Teil II

- Der Direktbereich kann noch mehr: wir können dort direkt VBA-Anweisungen und Ausdrücke ausführen.
 - Genau, deshalb heißt der Direktbereich Direktbereich.
 - Im Direktbereich können wir "mal eben" VBA-Code-Stücke direkt ausprobieren!

- Schreiben von Text in eine Zelle via Anweisung im Direktbereich:

Direktbereich
[A1] = "Hallo"

	A
1	Hallo

- Berechnung von Ausdrücken und Ausgabe im Direktbereich:

Direktbereich
? 12 + 9
21

Gut zu wissen:

Der ?-Operator, der Print-Operator, hat große Tradition in BASIC. Er gibt einfach das Ergebnis seines Arguments auf der Programmausgabe aus.

- Wird der ?-Operator vor den auszuführenden Ausdruck gesetzt, wird das Ergebnis des Ausdrucks selbst in den Direktbereich ausgegeben.
- Hinweis: Der Inhalt des Direktbereichs kann einfach mit Strg-A und Entf gelöscht werden.

Mit Zeichenketten arbeiten – Teil I

- Bisher haben wir nur mit numerischen Datentypen gearbeitet.
 - Das ist ja auch die herausragende Eigenschaft von Excel.
- Aber wir haben es auch oft mit der Verarbeitung von Textdaten zu tun.
 - Zeichenketten werden im Englischen eben "Strings" (engl. für Kette) genannt.
 - Wir wissen schon, dass Texte in Variablen vom Typ String gespeichert werden.
 - Außerdem wissen wir, dass Texte in doppelte Anführungszeichen gesetzt werden.

```
Sub Strings()  
    Dim text As String  
    text = "bar"  
    Debug.Print text  
    ' >bar  
End Sub
```

Mit Zeichenketten arbeiten – Teil II

- Eine sehr wichtige Funktion ist das Zusammensetzen von Strings.
 - Man spricht auch von der Verkettung (engl. concatenation, von lat. catena "Kette") von Strings.
- Die String-Verkettung wird mit dem &-Operator erledigt:

```
Sub StringVerkettung()  
    Dim text As String  
    text = "Hallo" & " " & "Welt!"  
    Debug.Print text  
    ' >Hallo Welt!  
End Sub
```

- Wir können Strings auch mit anderen Typen, z.B. Integer verketteten:

```
Sub StringIntegerVerkettung()  
    text =  
    Debug.Print "Zahl des Tages: " & 42  
    ' >Zahl des Tages: 42  
End Sub
```

Mit Zeichenketten arbeiten – Teil III

- Eine weitere String-Operation ist das bestimmen der String-Länge.

- Das wird mit der Funktion `Len()` gemacht:

```
Sub StringLänge()  
    Dim text As String  
    Dim textLänge As Integer  
    text = "bar"  
    textLänge = Len(text)  
    Debug.Print textLänge  
    ' >3  
End Sub
```

- Mit `Len()` erhalten wir die Anzahl der Buchstaben im übergebenen String. Das Ergebnis von `Len()` hat den Typ `Integer`.

Mit Zeichenketten arbeiten – Teil IV

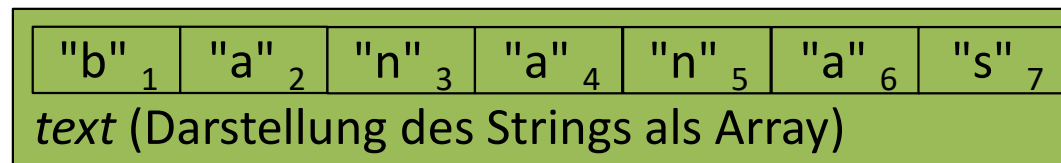
- In Zusammenhang mit der String-Länge steht auch der Zugriff auf einzelne Buchstaben.
 - Das macht man z.B. mit der Funktion *Mid()*:

```
Sub StringZugriff()  
    Dim text As String  
    text = "bananas"  
    For i = 1 To Len(text)  
        Debug.Print Mid(text, i, 1)  
    Next  
End Sub
```

Gut zu wissen:

Einzelne Buchstaben werden in der Programmierung oft als Character (kurz "Char") bezeichnet.

- *Mid()* werden der Zugriffs-String, die Position des Buchstabens im String und die Anzahl der "herauszuholenden" Buchstaben übergeben.
- Beachte: die Position eines Buchstabens ist 1-basiert, auch das ist eine Gemeinsamkeit mit Arrays.
- In der For-Schleife können wir bequem jedes Zeichen einzeln zugreifen: die Zählvariable für die Position und *Len(text)* für den Endwert.
- *Mid(text, i, 1)* holt in jedem Schleifendurchgang einen String mit der Länge 1 an der Position i.



Teilzeichenketten

- Statt einzelnen Buchstaben können wir auch ganze Teilzeichenketten herausholen.
 - Teilzeichenketten werden im Englischen Substrings genannt.
 - Die Extraktion von Substring kann ebenfalls mit *Mid()* gemacht werden:

```
Sub SubStrings()  
    Dim text, teilZeichenkette As String  
    text = "bananas"  
    teilZeichenkette = Mid(text, 3, 4)  
    Debug.Print teilZeichenkette  
    ' > nana  
End Sub
```

- Wir benutzen *Mid()* wie beim Buchstabenzugriff, aber extrahieren mehr als einen Buchstaben.
- Substrings können auch mit *Left()* und *Right()* extrahiert werden.
 - Anders als *Mid()* erhalten *Left()* und *Right()* keine Startposition, sie extrahieren Strings immer vom Anfang und Ende des Zugriffs-Strings.
 - Der Vorteil der Funktionen ist hierbei, dass sie weniger Argumente benötigen, was Fehler vermeiden kann.

Zeichenketten vergleichen

- Wenn wir Strings miteinander vergleichen wollen, können wir den =-Operator nicht benutzen!
- Stattdessen bietet VBA die Funktion *StrComp()* (StringCompare, engl. für String-Vergleich) an:

```
Sub StringVergleich()  
    Dim name1, name2 As String  
    name1 = "Frank"  
    name2 = "Frank"  
  
    If 0 = StrComp(name1, name2) Then  
        Debug.Print "Die Strings sind gleich!"  
    End If  
End Sub
```

- *StrComp()* werden die beiden zu vergleichenden Strings übergeben:
 - Das Ergebnis ist 0, wenn beide Strings genau gleich sind, also genau dieselbe Kette an Buchstaben darstellen.
 - Das Ergebnis ist -1, wenn der erste String lexikographisch kleiner ist als der zweite.
 - Das Ergebnis ist 1, wenn der erste String lexikographisch größer ist als der zweite.

In Zeichenketten suchen

- Eine weitere wichtige Operation ist die Suche von Substrings im Zugriffs-String.
 - Hierzu können wir die VBA-Funktion *InStr()* verwenden.
 - Die folgende Prozedur zählt wie oft der Substring "an" in "bananas" enthalten ist:

```
Sub SubStringZählen()  
    Dim aString As String  
    Dim letztePosition, anzahl_an As Integer  
  
    aString = "bananas"  
    letztePosition = InStr(1, aString, "an")  
    While letztePosition <> 0  
        anzahl_an = anzahl_an + 1  
        letztePosition = InStr(letztePosition + 2, aString, "an")  
    Wend  
    Debug.Print "Substring 'an' wurde " & anzahl_an & "-mal gefunden."  
    ' >Substring 'an' wurde 2-mal gefunden.  
End Sub
```

- *InStr()* werden die Startsuchposition, der Zugriffs-String und der zu suchende String übergeben.
 - Das Ergebnis ist die Position des Substrings im Zugriffs-String, oder 0, wenn der Substring nicht gefunden wurde.

Zelleigenschaften – Teil I

- Neben der Beeinflussung von Zellwerten können wir mit VBA auch Zellen formatieren.
- Es gibt sehr viele Formatierungsmöglichkeiten, aber wie greifen wir programmatisch darauf zu?
- VBA definiert hierzu für eine Zelle sogenannte Eigenschaften (engl. Properties).
- Wenn wir einen Wert eine Zelle setzten benutzen wir so eine (intuitive) Syntax:

```
Cells(1, 1) = "Hallo Welt"
```

- Eigentlich ist es nur eine abkürzende Syntax hierfür:

```
Cells(1, 1).Value = "Hallo Welt"
```

Zelleigenschaften – Teil II

- Das neue syntaktische Merkmal ist der Punkt-"Operator".
 - Der Punktoperator erlaubt uns auf die Propertys eines Objektes zuzugreifen.
- Was ist denn ein "Objekt"?
- Einfach gesagt, jedes "ansteuerbare" Element in Excel ist ein Objekt, z.B.:
 - Ein Bereich, eine Zelle
 - Ein Arbeitsblatt
 - Eine Zeichenkette
 - Ein Formular
- Objekte sind Grundlage der sog. objekt-orientierten Programmierung.
 - Das ist ein sehr mächtiges Konzept, aber nicht mehr Grundlage dieses Kurses.

Zelleigenschaften – Teil III

- Ein Objekt (etwa eine Zelle) kann in VBA durch eine Variable repräsentiert werden.

- Wir können das Setzen eines Zellwertes mit einem Zellenobjekt noch deutlicher hinschreiben:

```
Cells(1, 1).Value = "Hallo Welt"
```

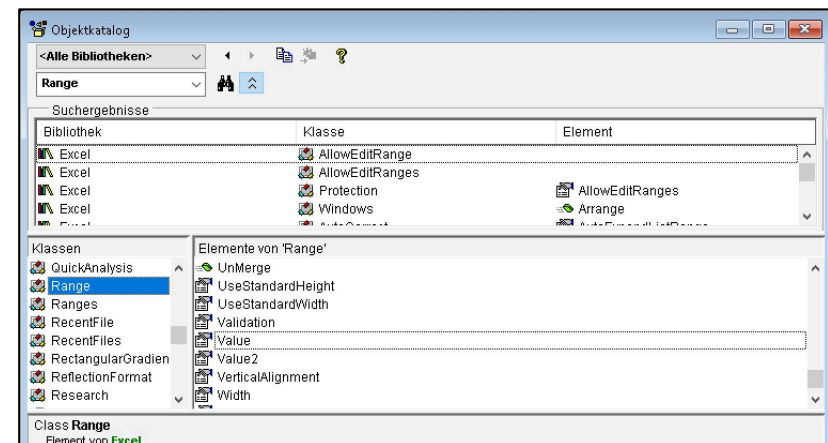
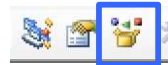


```
Dim zellObjekt As Range ' (1)  
Set zellObjekt = Cells(1, 1) ' (2)  
zellObjekt.Value = "Hallo Welt"
```

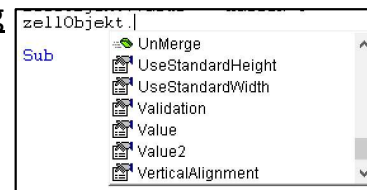
- Was passiert in dem neuen Code?
 - Zunächst definieren wir in (1) eine neue Variable *zellObjekt*, die den Typ Range hat.
 - Merke: In einer Range-Variablen kann man ein Zellobjekt speichern.
 - In (2) rufen wir die Funktion *Cells()* auf, die ein Zellobjekt vom Typ *Range* zurückgibt, dass die Zelle [A1] repräsentiert.
 - Das Ergebnis von *Cells()* ist zum Typ der Variablen *zellObjekt* passend (Fachwort: kompatibel).
 - Mit dem Set-Schlüsselwort können wir schließlich der Variable *zellObjekt* das *Range*-Objekt aus *Cells()* zuweisen.
 - Leider müssen wir in Excel beim Zuweisen von Objekten an eine Variable das Set-Schlüsselwort und nicht nur den "nackten" =-Operator verwenden.
 - Die Variable *zellObjekt* enthält (oder besser "referenziert") jetzt ein *Range*-Objekt.
 - Das *Range*-Objekt enthält viele Propertys, u.a. *Value*, die wir mit dem Punktoperator ansprechen können.

Übersicht der Propertys von Objekten

- Über die Propertys eines Objekten können wir viele Eigenschaften von Objekten ablesen und einstellen.
- Excel bietet uns Hilfen, um mit der Flut an Propertys von Objekten umzugehen.
 - Zunächst können wir den Objektkatalog verwenden:



- Alternativ dazu können wir die Auto-Vervollständigung benutzen, die nach Eingabe des Punktoperators im Editor aufklappt:



Tipp: Die Autovervollständigung kann mit Druck auf Strg+Leertaste nach dem eingegebenen Punkt explizit ausgelöst werden.

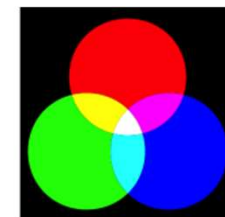
Die Farbe einer Zelle ändern

- Wir können in diesem Kurs nicht über alle Objekte und deren Propertys sprechen.
- Aber wir besprechen das Interior-Property (engl. Innenraum) des *Range*-Objekts beispielhaft.
 - *Interior* bietet uns Zugriff auf den Innenraum einer Zelle, also z.B. deren Hintergrundfarbe, Farbverlauf oder Muster.
 - Um z.B. die Hintergrundfarbe einer Zelle zu setzen, können wir das "Unter-Property" *Color* von *Interior* von *Range* zugreifen:

```
zellObjekt.Interior.Color = RGB(255, 0, 0)
```

	A	B
1		

- Neu für uns ist die Verwendung mehrerer Punktoperatoren, um Properties von Properties zuzugreifen.
 - Dennoch ist die Idee dahinter nachziehbar: wir können direkt auf Propertys zugreifen, ohne Zwischenobjekte in Variablen zu speichern.
- Für uns ist auch die Funktion *RGB()* neu.
 - *RGB()* bestimmt aus den drei Werten für rot, grün und blau eine Farbe und gibt das Ergebnis zurück.
 - Die drei Werte werden als *Integer*-Werte jeweils zwischen 0 und 255 (inkl.) angegeben und beschreiben eine Farbe mittels additiver Farbmischung.
 - *RGB(255, 0, 0)* -> rot, *RGB(0, 255, 0)* -> grün, *RGB(0, 0, 255)* -> blau, *RGB(255, 0, 255)* -> magenta, *RGB(0, 0, 0)* -> schwarz, *RGB(255, 255, 255)* -> weiß, *RGB(150, 150, 150)* -> grau



additive color mix

Strukturierung von Makros mit Prozeduren – Teil I

- Bisher haben wir "Makro" und "Prozedur" immer als eine Einheit betrachtet.
- Eigentlich kann ein Makro aus beliebig vielen Prozeduren zusammengesetzt sein.
 - Der Aufbau von Programmen mit Prozeduren ist Basis der prozeduralen Programmierung.
 - Klassisch kann man eine Prozedur auch als Unterprogramm (Sub-Programm) verstehen.
- Programme werden oft mit Unterprogrammen für Eingabe, Verarbeitung und Ausgabe aufgebaut.
 - Wir sprechen hierbei auch vom EVA-Prinzip (Eingabe, Verarbeitung, Ausgabe).
- Nun besprechen wir, wie wir Prozeduren in VBA einsetzen können.

Strukturierung von Makros mit Prozeduren – Teil II

- Den grundsätzlichen Aufbau einer einzelnen Prozedur kennen wir schon:

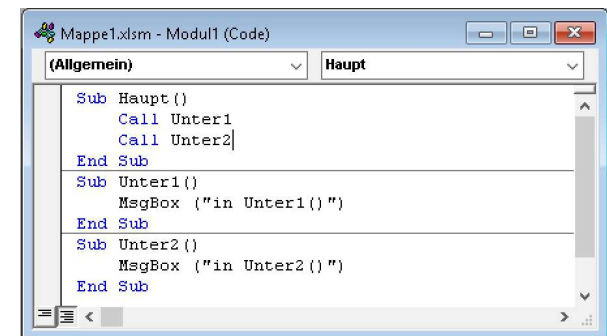
```
Sub HelloWorld()  
    MsgBox ("Hallo Welt")  
End Sub
```

- So bauen wir eine Haupt-Prozedur auf, die "ihre" Unterprozeduren aufruft:

```
Sub Haupt()  
    Call Unter1  
    Call Unter2  
End Sub
```

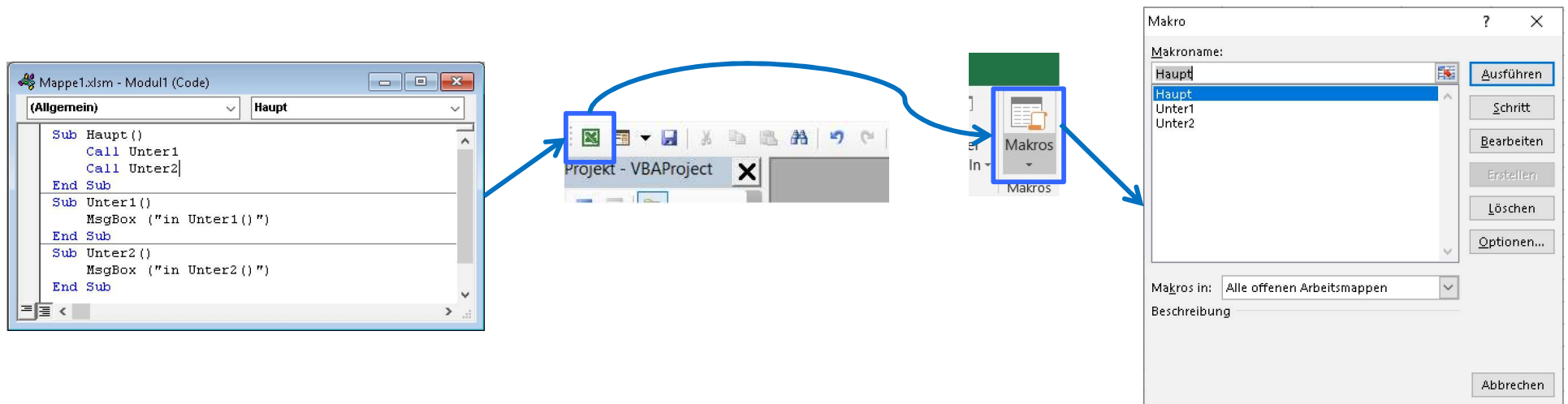
```
Sub Unter1()  
    Call MsgBox("in Unter1()")  
End Sub
```

```
Sub Unter2()  
    Call MsgBox("in Unter2()")  
End Sub
```

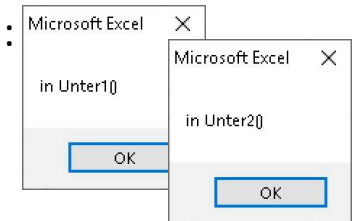


Strukturierung von Makros mit Prozeduren – Teil III

- Bei der Ausführung des Makros müssen wir uns jetzt für *Haupt()* entscheiden:

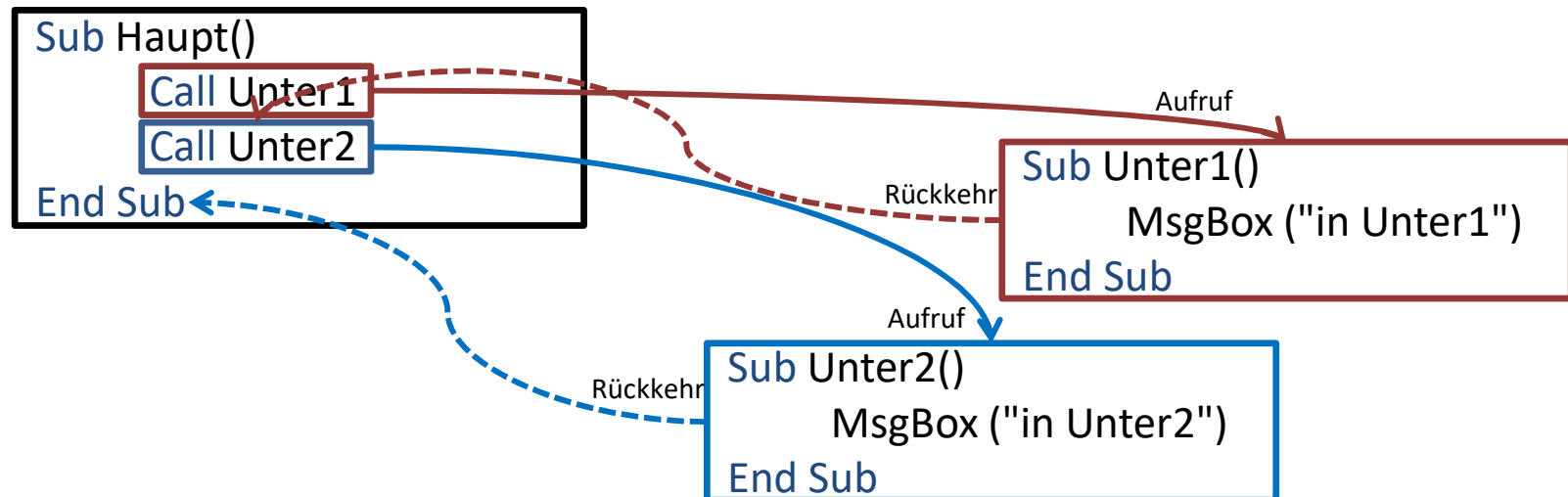


- Die Ausführung von *Haupt()* ruft die (Unter-)Prozeduren *Unter1()* und *Unter2()* aus.
 - Wir sehen das durch die sich öffnenden Dialoge, die wir wegklicken müssen:



Strukturierung von Makros mit Prozeduren – Teil IV

- Prozeduren werden mit der Call-Anweisung aufgerufen.
 - Wenn die aufgerufene Prozedur vollständig durchlaufen wurde, kehrt die Ausführung wieder an die aufrufende Prozedur zurück.



- Somit blockiert eine aufgerufene Prozedur die Ausführung der aufrufenden Prozedur.
 - Haupt()* ist die aufrufende Prozedur auch Aufrufer genannt, *Unter1()* und *Unter2()* sind die aufgerufenen Prozeduren.
 - Das erklärt, warum wir den Dialog aus *Unter1()* wegklicken müssen, bevor *Unter2()* ausgeführt wird und wir den 2. Dialog sehen.

EVA mit Prozeduren – Teil I

- Ein typische Anwendung für Prozeduren ist die Eingabeprüfung im "Eingabeschritt" beim EVA-Prinzip.
 - Wir arbeiten jetzt das Noten-Beispiel um, um Prozeduren einzusetzen.

```
Sub NotenBeispiel()  
    Call Eingabe()  
    Call Ausgabe()  
End Sub
```

```
Sub Eingabe()  
    [A1] = InputBox("Welche Note hat der Kandidat?")  
    if [A1] <= 0 Or [A1] > 6 Then ' Prüfung  
        MsgBox("Das ist ein ungültiger Wert!")  
    End If  
End Sub
```

```
Sub Ausgabe()  
    Select Case [A1]  
        Case 1: [B1] = "sehr gut"  
        Case 2: [B1] = "gut"  
        Case 3: [B1] = "befriedigend"  
        Case 4: [B1] = "ausreichend"  
        Case 5: [B1] = "mangelhaft"  
        Case 6: [B1] = "ungenügend"  
    End Select  
End Sub
```

EVA mit Prozeduren – Teil II

- Dieser erste Schritt, die prozedurale Dekomposition von *NotenBeispiel()*, führt schon mal zu mehr Struktur:

```
Sub NotenBeispiel()  
    Call Eingabe()  
    Call Ausgabe()  
End Sub
```

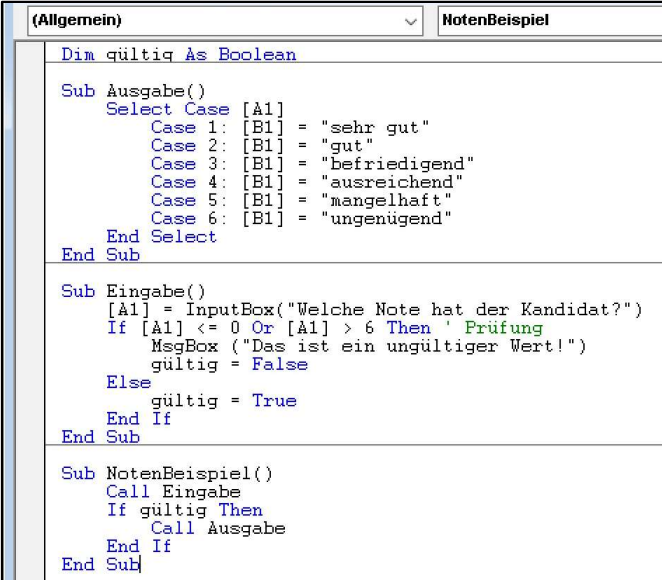
```
Sub Eingabe()  
    [A1] = InputBox("Welche Note hat der Kandidat?")  
    if [A1] <= 0 Or [A1] > 6 Then ' Prüfung  
        MsgBox("Das ist ein ungültiger Wert!")  
    End If  
End Sub
```

```
Sub Ausgabe()  
    Select Case [A1]  
        Case 1: [B1] = "sehr gut"  
        Case 2: [B1] = "gut"  
        Case 3: [B1] = "befriedigend"  
        Case 4: [B1] = "ausreichend"  
        Case 5: [B1] = "mangelhaft"  
        Case 6: [B1] = "ungenügend"  
    End Select  
End Sub
```

- VBA bietet uns viele Möglichkeiten, um die Dekomposition weiter zu verbessern, denn es gibt hier einige Probleme:
 - Der Ausgabeschritt, also die Prozedur *Ausgabe()* wird immer ausgeführt, egal ob die Eingabe gültig war oder nicht.
 - Das Beispiel funktioniert nur, wenn die Eingabe in der Zelle [A1] steht und die Ausgabe wird immer nach [B1] geschrieben!

Prozedurübergreifende Programmsteuerung – globale Variablen

- Dass *Ausgabe()* immer ausgeführt wird, egal ob die Eingabe gültig war, tritt auf, weil die Prozeduren *Eingabe()* und *Ausgabe()* unabhängig voneinander ausgeführt werden.
 - D.h. wir müssen erreichen, dass *Eingabe()* und *Ausgabe()* miteinander kommunizieren und eine Abhängigkeit entsteht.
- Eine einfache Lösung hierbei ist der Einsatz einer globalen Variablen.
 - Auf eine globale Variable können alle Prozeduren (eines Modules) zugreifen, sie also lesen und schreiben.
- Folgendes Beispiel zeigt so eine Lösung.
 - Die neue globale **Boolean**-Variable "*gültig*" existiert parallel zu *NotenBeispiel()*, *Eingabe()* und *Ausgabe()*.
 - *NotenBeispiel()*, *Eingabe()* und *Ausgabe()* sind nämlich auch globale Prozeduren, die sich und globale Variablen gegenseitig "sehen".
 - Bei einer ungültigen Eingabe, wird in *Eingabe()* die globale Variable "*gültig*" auf **False** gesetzt und die Warnung gezeigt.
 - Wenn die Eingabe aber gültig ist, wird in *Eingabe()* die globale Variable "*gültig*" auf **True** gesetzt.
 - In *NotenBeispiel()* wird *Ausgabe()* nur dann ausgeführt, wenn die globale Variable "*gültig*" auf **True** steht.



```
Dim gültig As Boolean

Sub Ausgabe()
    Select Case [A1]
        Case 1: [B1] = "sehr gut"
        Case 2: [B1] = "gut"
        Case 3: [B1] = "befriedigend"
        Case 4: [B1] = "ausreichend"
        Case 5: [B1] = "mangelhaft"
        Case 6: [B1] = "ungenügend"
    End Select
End Sub

Sub Eingabe()
    [A1] = InputBox("Welche Note hat der Kandidat?")
    If [A1] <= 0 Or [A1] > 6 Then ' Prüfung
        MsgBox ("Das ist ein ungültiger Wert!")
        gültig = False
    Else
        gültig = True
    End If
End Sub

Sub NotenBeispiel()
    Call Eingabe
    If gültig Then
        Call Ausgabe
    End If
End Sub
```


Prozedurübergreifende Programmsteuerung – Funktionen – Teil I

- Globale Variablen haben in VBA/in Basic allgemein eine gewisse Tradition, aber man kann leicht den Überblick verlieren.
 - Man muss sich nur vorstellen, wie ein Makro mit vielen globalen Variablen aussieht.
 - Wenn Prozeduren über globale Variablen kommunizieren, spricht man von seiteneffekt-orientierter Programmierung.
 - Globale Variablen arbeiten wie ein "Seitenkanal" zum "normalen" Prozedurablauf und können logische Abläufe "verdecken".
- Eine Alternative zum Einsatz von Seiteneffekten ist die Verwendung von Funktionen.
 - Funktionen kommunizieren nach ihrer Ausführung direkt mit der aufrufenden Prozedur über ihre Rückgabewerte.
- Formulieren wir also *Eingabe()* als Funktion um, so dass sie die Eingabegültigkeit direkt zurückgibt:

```
Dim gültig As Boolean ' globale gültig-Variable
Sub Eingabe() ' Eingabe als Prozedur
    [A1] = InputBox("Welche Note hat der Kandidat?")
    If [A1] <= 0 Or [A1] > 6 Then ' Prüfung
        MsgBox("Das ist ein ungültiger Wert!")
        gültig = False
    Else
        gültig = True
    End If
End Sub
```



```
Function Eingabe() ' Eingabe als Funktion
    [A1] = InputBox("Welche Note hat der Kandidat?")
    If [A1] <= 0 Or [A1] > 6 Then ' Prüfung
        MsgBox("Das ist ein ungültiger Wert!")
        Eingabe = False ' Rückgabe
    Else
        Eingabe = True ' Rückgabe
    End If
End Function
```

Prozedurübergreifende Programmsteuerung – Funktionen – Teil II

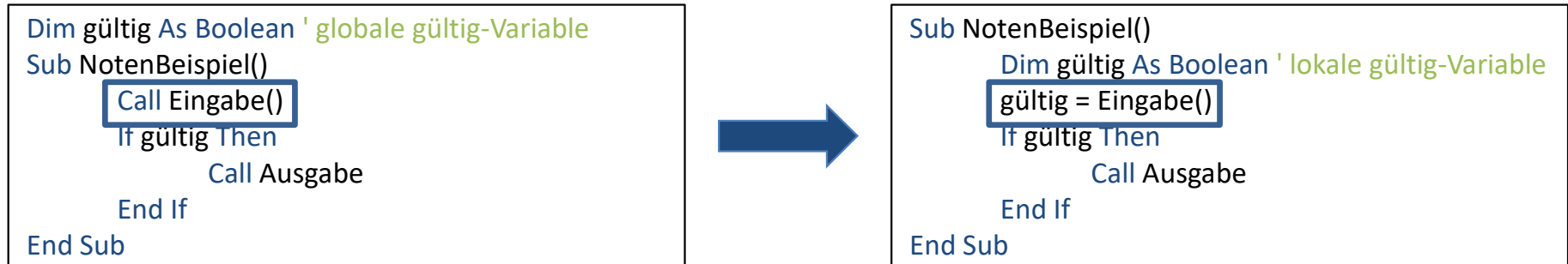
- Auf den zweiten Blick erkennen wir wie sich in *Eingabe()*, jetzt als Funktion definiert, der Code geändert hat.

```
Function Eingabe() ' Eingabe als Funktion
    [A1] = InputBox("Welche Note hat der Kandidat?")
    If [A1] <= 0 Or [A1] > 6 Then ' Prüfung
        MsgBox("Das ist ein ungültiger Wert!")
        Eingabe = False ' Rückgabe
    Else
        Eingabe = True ' Rückgabe
    End If
End Function
```

- Die Funktionsrückgabe geschieht, wenn wir in der Funktion *Eingabe()* der Funktion als lokale Variable einen Wert zuweisen.
- Die Funktion wird mit dem Schlüsselwort **Function** eingeleitet und mit **End Function** abgeschlossen.

Prozedurübergreifende Programmsteuerung – Funktionen – Teil III

- Auch der Code, um *Eingabe()* als Funktion aufzurufen hat sich geändert:



- Die Variable *gültig* ist nun keine globale Variable mehr, sondern eine lokale Variable.
 - D.h., dass die Definition von *gültig* nur in *NotenBeispiel()* vorgenommen wurde!
 - D.h. auch, dass *gültig* nur in *NotenBeispiel()* sichtbar ist, es steht für andere Prozeduren nicht zur Verfügung!
 - Auch für *Eingabe()* steht *gültig* als Variable, die nur in *NotenBeispiel()* lokal ist, nicht mehr als "Seitenkanal" zur Verfügung!
 - *Eingabe()* nutzt jetzt einen anderen "Kanal", es gibt den Gültigkeitswert einfach als Ergebnis zurück!
 - Also hat *Eingabe()* jetzt ein Ergebnis, dass wir der lokalen Variablen zuweisen können.
 - Es ist auch zu beachten, dass *Eingabe()* als Funktion ohne die `Call`-Anweisung aufgerufen wird.
- Beachte: *NotenBeispiel()* (der Aufrufer) ist weiterhin als Prozedur und nicht als Funktion definiert.

Prozedurübergreifende Programmsteuerung – Funktionen – Teil IV

- Leider haben wir immer noch einen Seiteneffekt: die Zelle A1!
 - In der Funktion *Eingabe()* wird die Benutzereingabe nach A1 geschrieben.

```
Function Eingabe() ' Eingabe als Funktion
    [A1] = InputBox("Welche Note hat der Kandidat?")
    If [A1] <= 0 Or [A1] > 6 Then ' Prüfung
        MsgBox("Das ist ein ungültiger Wert!")
        Eingabe = False
    Else
        Eingabe = True
    End If
End Function
```

- Wir können das verbessern:
 - (1) In *Eingabe()* speichern wir den Eingabewert (also den Notenwert von 1-6) in einer Variablen und nicht in einer Zelle.
 - (2) *Eingabe()* gibt den erhaltenen Eingabewert als Ergebnis selbst zurück.
 - Also *Eingabe()* gibt dann keinen **Boolean**-Wert mehr zurück, sondern einen **Integer**-Wert (nämlich den Notenwert).
 - (3) Da wir die Information über die Gültigkeit verlieren, müssen wir das anders an den Aufrufer mitteilen.
 - Wir machen das so, in dem wir eine unmögliche Note, nämlich 0, als ungültigen Wert zurückliefern.

Prozedurübergreifende Programmsteuerung – Funktionen – Teil V

- Die neue Variante von *Eingabe()* sieht dann so aus:

```
Function Eingabe()  
    Dim note As Integer ' lokale Zwischenvariable für die Eingabe  
    note = InputBox("Welche Note hat der Kandidat?")  
    If note <= 0 Or note > 6 Then  
        MsgBox("Das ist ein ungültiger Wert!")  
        Eingabe = 0 ' 0 als ungültiger Notenwert  
    Else  
        Eingabe = note  
    End If  
End Function
```

- NotenBeispiel()* nimmt nun die Note als *Integer* von *Eingabe()* entgegen und prüft sie gegen 0:

```
Sub NotenBeispiel()  
    Dim note As Integer ' lokaler Notenwert  
    note = Eingabe()  
    If note <> 0 Then ' Ausgabe nur dann, wenn der Notenwert gültig ist  
        Call Ausgabe  
    End If  
End Sub
```

Prozedurübergreifende Programmsteuerung – Parameter – Teil I

- Aber wir müssen jetzt auch *Ausgabe()* anpassen!
 - Denn *Ausgabe()* liest immer noch die Zelle A1, die die Eingabe ja gar nicht mehr "füllt":

```
Sub Ausgabe()  
    Select Case [A1] ' Hier wird die Zelle A1 gelesen  
        Case 1: [B1] = "sehr gut"  
        Case 2: [B1] = "gut"  
        Case 3: [B1] = "befriedigend"  
        Case 4: [B1] = "ausreichend"  
        Case 5: [B1] = "mangelhaft"  
        Case 6: [B1] = "ungenügend"  
    End Select  
End Sub
```

- Eine Lösung: wir entfernen den Seitenkanal über A1 und stattdessen *Ausgabe()* mit einem Parameter aus.
 - Parameter kennen wir ja auch schon von den Arbeitsmappen-Funktionen in Excel.
- Wie Rückgabewerte bei Funktionen erlauben Parameter als Eingabewerte Kommunikation zwischen Prozeduren.

Prozedurübergreifende Programmsteuerung – Parameter – Teil II

- Wir definieren nun in *Ausgabe()* einen Integer-Parameter, der den Notenwert der Eingabe entgegennimmt:

```
Sub Ausgabe(note As Integer) ' Ausgabe() hat jetzt einen Integer-Parameter
    Select Case note
        Case 1: [B1] = "sehr gut"
        Case 2: [B1] = "gut"
        Case 3: [B1] = "befriedigend"
        Case 4: [B1] = "ausreichend"
        Case 5: [B1] = "mangelhaft"
        Case 6: [B1] = "ungenügend"
    End Select
End Sub
```

- Dem Aufruf von *Ausgabe()* muss nun der Notenwert aus *Eingabe()* als Argument übergeben werden.
 - Genau wie bei Arbeitsblattfunktionen übergeben wir Argumente in den Klammern hinter dem Prozeduraufruf:

```
Sub NotenBeispiel()
    Dim note As Integer
    note = Eingabe()
    If note <> 0 Then
        Call Ausgabe(note) ' Übergabe an Ausgabe()
    End If
End Sub
```

Prozedurübergreifende Programmsteuerung – Parameter – Teil III

- Die Änderungen an der Syntax von *Ausgabe()* sind überschaubar:

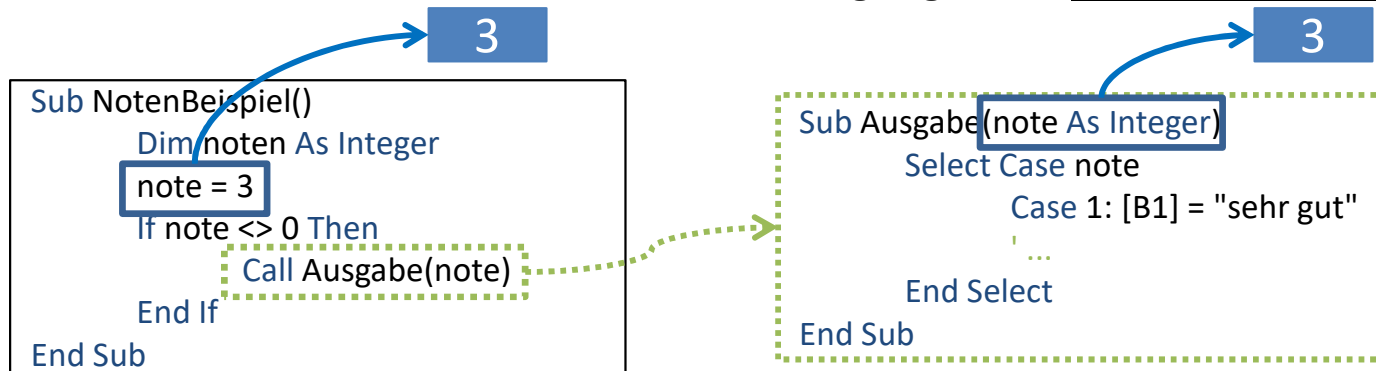
```
Sub Ausgabe(note As Integer) ' Ausgabe() hat jetzt einen Integer-Parameter
    Select Case note
        Case 1: [B1] = "sehr gut"
        ' ...
    End Select
End Sub
```

- In den (sonst leeren) Klammern der Prozedurdefinition definieren wir jetzt eine Integer-Variable *note*, *note* ist jetzt ein Parameter.
 - Im Gegensatz zu lokalen Variablen werden Parameter nicht mit dem *Dim*-Schlüsselwort definiert.
 - *note* kann jetzt vom Aufrufer beim Aufruf mit einem Wert belegt werden, dem Argument.
 - *note* kann jetzt in *Ausgabe()* wie jede andere lokale Variable zugegriffen werden.
- In *NotenBeispiel()* wird jetzt einfach der Notenwert an *Ausgabe()* übergeben:

```
Sub NotenBeispiel()
    Dim note As Integer
    note = Eingabe()
    If note <> 0 Then
        Call Ausgabe(note) ' Übergabe an Ausgabe()
    End If
End Sub
```


Prozedurübergreifende Programmsteuerung – Call by Value – Teil I

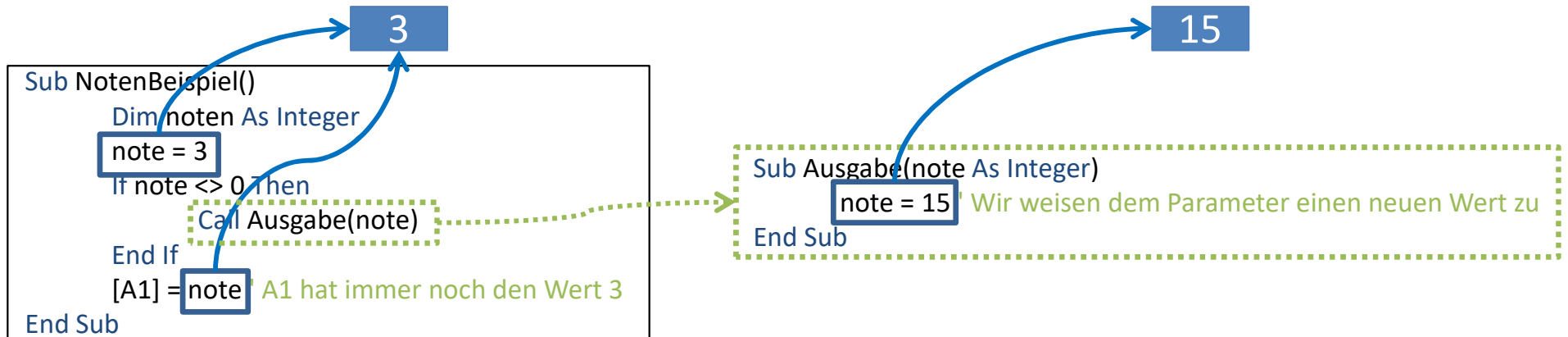
- Die Diskussion über Parameter schließt auch Überlegungen zur Speicherverwendung ein:



- Die lokale Variable *note* in *NotenBeispiel()* und der Parameter *note* in *Ausgabe()* liegen an versch. Stellen im Arbeitsspeicher.
 - Das bedeutet, dass der Parameter *note* eine Kopie des Wertes den der Aufrufer übergeben hat enthält.
 - !! Es spielt keine Rolle, dass die lokale Variable und der Parameter hier den gleichen Namen haben !!
- Wenn die aufgerufene Prozedur eine Kopie des Arguments erhält nennt man das "Call by Value".

Prozedurübergreifende Programmsteuerung – Call by Value – Teil II

- Den "Call by Value"-Effekt kann man einfach beweisen.
 - Wenn wir den Parameter *note* in *Ausgabe()* ändern, ändert sich der Wert der lokalen Variable *note* in *NotenBeispiel()* nicht!



- Was wir hier sehen ist der Kopiereffekt der Argumentübergabe mit "Call by Value".
 - *Ausgabe()* erhielt nur eine Kopie des Ausgangswerts 3, in *Ausgabe()* haben wir dann die Kopie auf den Wert 15 geändert.
 - Die "original" *note* in *NotenBeispiel()* bleibt von den Änderungen in *Ausgabe()* gänzlich unbeeindruckt.

Prozedurübergreifende Programmsteuerung – Exkurs – Call by Reference – Teil I

- Wir können Argumente auch so übergeben, dass sie in der aufgerufenen Prozedur geändert werden können.
 - Änderungen an den Parametern haben dann Auswirkungen oder "Rückwirkungen" auf die Argumente der aufrufenden Prozedur.
- Eine sinnvolle Anwendung hierfür wäre die Rückgabe des Notentexts:

```
Sub Ausgabe(note As Integer)
```

```
    Select Case note
```

```
        Case 1: [B1] = "sehr gut"
```

```
        Case 2: [B1] = "gut"
```

```
        Case 3: [B1] = "befriedigend"
```

```
        Case 4: [B1] = "ausreichend"
```

```
        Case 5: [B1] = "mangelhaft"
```

```
        Case 6: [B1] = "ungenügend"
```

```
    End Select
```

```
End Sub
```



```
Sub Ausgabe(note As Integer, notenText As String)
```

```
    Select Case note
```

```
        Case 1: notenText = "sehr gut"
```

```
        Case 2: notenText = "gut"
```

```
        Case 3: notenText = "befriedigend"
```

```
        Case 4: notenText = "ausreichend"
```

```
        Case 5: notenText = "mangelhaft"
```

```
        Case 6: notenText = "ungenügend"
```

```
    End Select
```

```
End Sub
```

- Wir haben somit den Seiteneffekt/Seitenkanal mit der Zelle B1 durch die Verwendung eines Parameters ersetzt.
- Allerdings verhält sich *Ausgabe()* eigentlich immer noch wie eine "Call by Value"-Prozedur!
 - Also wenn wir *Ausgabe()* "wie gehabt" aufrufen, würden immer noch Kopien der Arguments übergeben.

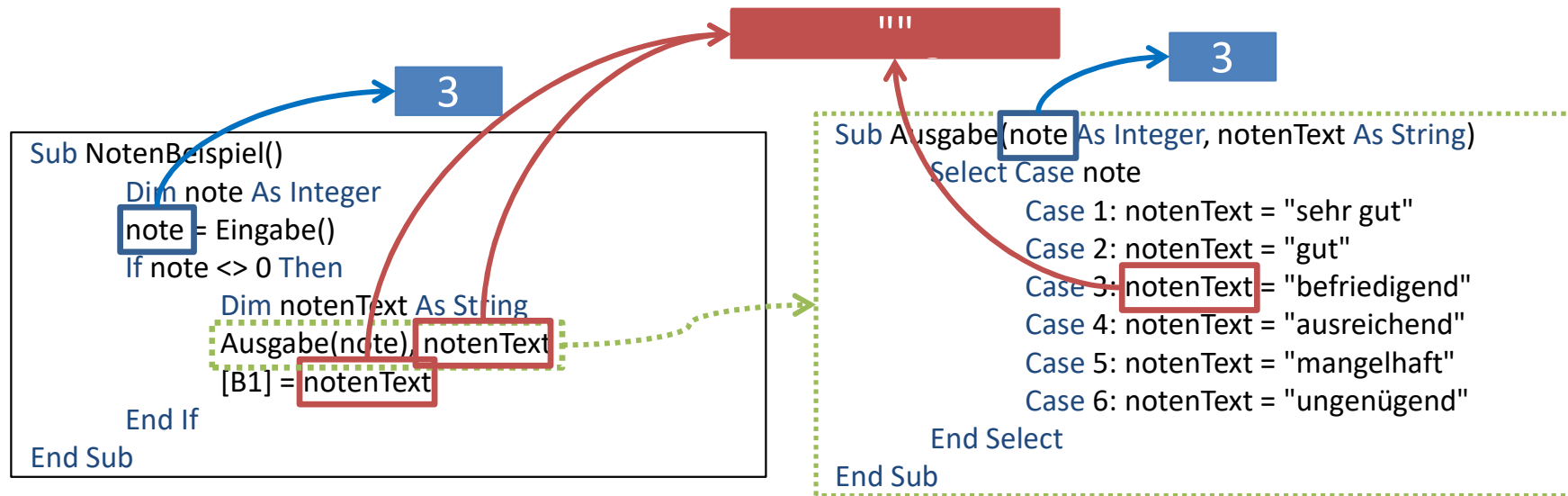
Prozedurübergreifende Programmsteuerung – Exkurs – Call by Reference – Teil II

- Der "Trick" liegt nun darin, dass wir *Ausgabe()* auf spezielle Art aufrufen müssen, um Kopien zu verhindern:

```
Sub NotenBeispiel()  
    Dim note As Integer  
    note = Eingabe()  
    If note <> 0 Then  
        Dim notenText As String  
        Ausgabe(note), notenText  
        [B1] = notenText  
    End If  
End Sub
```

- Die neue Syntax für den Aufruf von *Ausgabe()* sieht etwas merkwürdig aus.
 - (Zunächst mal müssen wir *notenText* als lokale Variable definieren, wir wollen ja die Änderung daran dann in B1 speichern.)
 - Beim Aufruf müssen die "Call by Value" Argumente, also *note*, wie zuvor in Klammern übergeben werden.
 - Dahinter wird ein Komma gesetzt und dann werden die Argumente geschrieben, die nicht kopiert werden sollen, also *notenText*.
- Schauen wir uns das gesamt Bild an.

Prozedurübergreifende Programmsteuerung – Exkurs – Call by Reference – Teil III



- In der Grafik werden die Unterschiede zwischen der Übergabe mit und ohne Kopie klar.
 - Die lokale Variable `notenText` beansprucht jetzt nur eine einzige Stelle im Arbeitsspeicher.
 - Diesen Speicher teilen sich jetzt die Prozeduren `NotenBeispiel()` und `Ausgabe()`, beiden "sehen" die gleiche Stelle für den `notenText`.
 - Der geteilte Speicher ist jetzt unser Seitenkanal für die Kommunikation zwischen den Prozeduren.
 - Wenn `notenText` in `Ausgabe()` geändert wird, ändert es sich auch im Aufrufer, diesen Effekt nennt man Aliasing.
- Werden bei der Argumentübergabe nur Verweise auf Speicherstellen übergeben nennt man das "Call by Reference".