Inhalt

1. Einführung in die Informatik

2. Algorithmen

3. Imperative Programmierung

- Strukturierte / Prozedurale Programmierung
- Objektorientierte Programmierung
- Ein- und Ausgabe über Benutzerschnittstellen (Konsole, GUI)

Imperative Programmierung

… in der Sprache Visual Basic .NET

Inhalt des Abschnitts

- Varianten der Programmiersprache Basic
- Programmaufbau in Visual Basic
- Basisdatentypen, Variablen und einfache Anweisungen
- Ein- und Ausgabe über Benutzerschnittstellen (Konsole, GUI)
- Steuerfluss-Konstrukte
- Bedingungen und Aussagenlogik
- Felder und Strukturen
- Unterprogramme: Prozeduren und Funktionen
- Objektorientierte Programmierung

Basic-Varianten (1)

Basic wurde 1964 als einfache Programmiersprache entwickelt.

Erste Systeme als BASIC Interpreter, d.h. der BASIC-Code wurde zur Laufzeit des Programms im Maschinencode umgewandelt.

Später entstanden viele verschiedene BASIC-Dialekte:

- IBMPC-Basic Interpreter (GWBASIC)
- Quick-BASIC
- VBA – Makrosprache für Microsoft Programme (z.B. Excel)
- Visual Basic .NET innerhalb Visual Studio

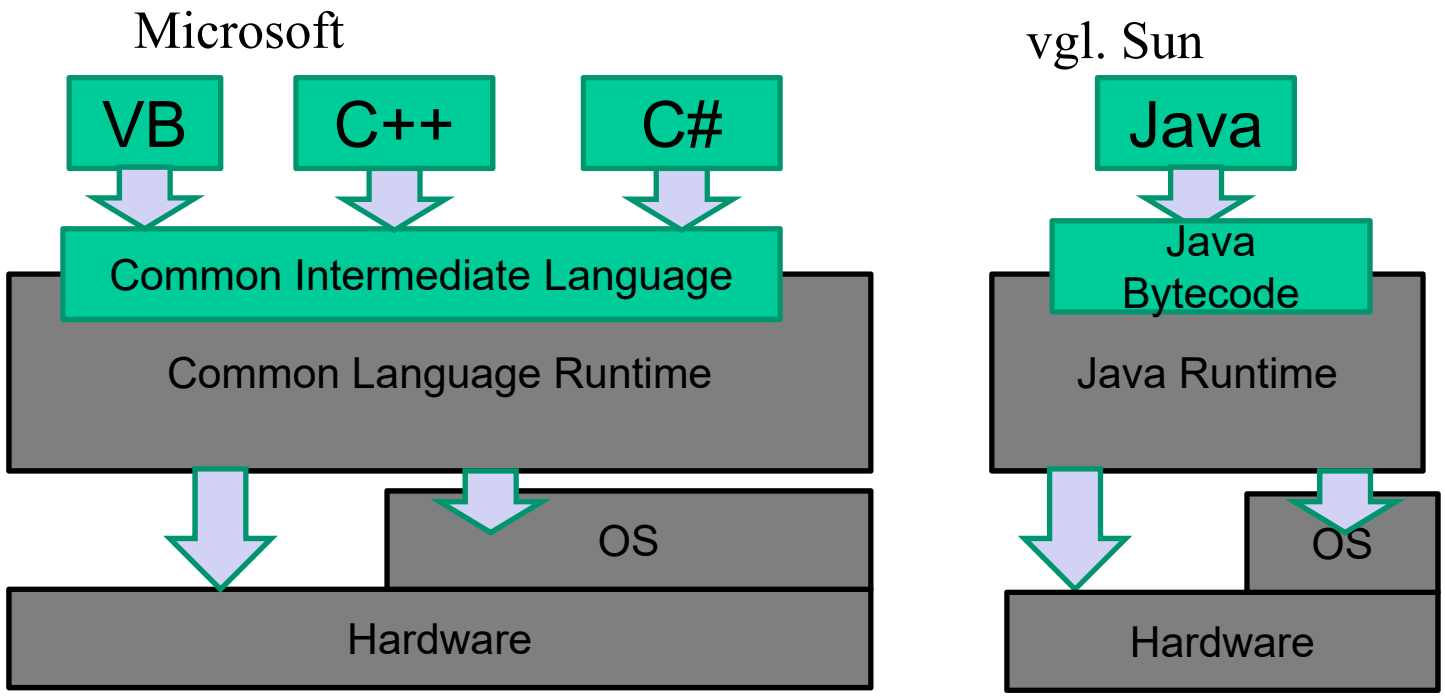
…

Dialekte unterscheiden sich z.B. darin, ob Unterprogramme unterstützt werden und welche Ein- und Ausgabebefehle existieren.

Basic-Varianten (2)

VBA-Skripte innerhalb MS-Excel sind als interpretierte Makros zu verstehen.

Visual Basic (VB) innerhalb Visual Studio ist eine Sprache, die auf die .NET Plattform kompiliert wird. Dort wird z.B. die Zusammenarbeit mit Modulen gewährleistet, die in anderen Programmiersprachen geschrieben wurden.



Basic-Varianten (3)

Programme, die in den verschiedenen BASIC-Systemen und Dialekten geschrieben sind, sind i.d.R. nicht kompatibel.

Aber:

Systemunabhängige Programmteile, wie z.B. Unterprogramme ohne Ein- und Ausgabe können wiederverwendet werden.

Merkmale der Sprache Visual Basic

Merkmale der Sprache:

- ***Keine Zeilennummerierung*** - Codezeilen ohne Nummerierung, Benutzung von Zeilenmarken möglich, um Sprünge zu programmieren
- ***imperative, prozedurale Sprache***
- ***Funktionen und Prozeduren*** – Modulares Programmieren durch selbstdefinierte Unterprogramme
- ***Objekte*** – Objekte mit Eigenschaften und Methoden als Erweiterung
- ***Grafische Benutzeroberfläche*** – Anwendungen können ausgehend von einer Oberfläche (Fenster) gestaltet werden. Bedien- und Steuerelemente können grafisch platziert werden. Ereignisse werden mit programmierten Aktionen verbunden

Programmaufbau (1)

Ein (unabhängig startfähiges) Programm hat immer eine Hauptroutine:

```
REM Visual Basic Programm
Module Module1
        Sub Main()
        REM  hier die BASIC Anweisungen
        End Sub
End Module
```

Programmaufbau (2)

Ein anderer Programmaufbau für BASIC-Interpretersysteme:

```
REM Visual Basic Programm
100 PI=4*ATN(1)
110 INPUT "Radius";R
120 U=2*PI*R
130 F=PI*R*R
140 PRINT "RADIUS=";R
150 PRINT "UMFANG=";U
160 PRINT "FLÄCHE=";F
170 END
```

Die Hauptroutine beginnt hier implizit mit der ersten Anweisung

Kommentare: REM oder ' (Apostroph)

Programmaufbau (3)

Deklaration der Variablen - global:

```
Module Module1
    Dim eingabefehler As Boolean
    Dim a, b, c As Integer
    Dim Nachricht As String
    REM Variablen global sichtbar, auch für andere Subroutinen
    Sub Main()
    REM Hier beginnt der Programmcode

    End Sub
End Module
```

Programmaufbau (4)

Deklaration der Variablen – lokal für Subroutinen:

```
Module Module1
    Sub Main()
        Dim eingabefehler As Boolean
        Dim a, b, c As Integer
        Dim Nachricht As String
        REM diese Variablen sind nur in der Main-Routine sichtbar

        REM Hier beginnt der Programmcode
    End Sub
End Module
```

Programmaufbau (5)

Programmanweisungen folgen nach der Variablendeklaration

```
Module Module1
    Sub Main()
      Dim eingabefehler As Boolean
      Dim a, b, c As Integer
      REM Hier beginnt der Programmcode
      Do
        eingabefehler = False
        a = InputBox("Bitte a eingeben:")
        If (Not IsNumeric(a) And a < 1) Then
              MsgBox("es wurde kein gültiger Wert eingegeben")
              eingabefehler = True
         End If
       Loop While (eingabefehler )
       …
     End Sub
   End Module
```

Programmaufbau (6)

Programmanweisungen sind:

- Einfache Anweisungen in BASIC-Syntax
- Steuerfluss-Konstrukte in BASIC-Syntax, z.B. für Programmschleifen
- Prozeduraufrufe (Sub – Subroutine)
- Aufrufe von Bibliotheksfunktionen, die üblicherweise als Objekte innerhalb .NET referenziert werden müssen

Basisdatentypen und Variablen (1)

- Variablen dienen zur Aufnahme von Werte während der Programmverarbeitung.
- Variablen werden durch Bezeichner referenziert und müssen entsprechend deklariert werden
- Jede Variable besitzt einen Typ, der wiederspiegelt, welche Werte die Variable aufnehmen kann, z.B. ganzzahlige Werte, Wahrheitswerte, Textzeilen usw.

Beispiel:

```
Dim A as Integer
Dim Nachricht as STRING
A=-42
Nachricht = “Es ist kalt!“
Console.Write(“Die Außentemperatur beträgt “)
Console.Write(A)
Console.WriteLine(“ Grad.“)
Console.WriteLine(Nachricht)
```

Basisdatentypen und Variablen (2)

Standardtypen:

Typ	Erläuterung	Speicherplatz
Byte	Ganzzahl zwischen 0 und 255	1 Byte
Boolean	Wahrheitswert (True = -1, False = 0)	1 Byte
Char	einzelnes Zeichen, z.B. `A`, `B`, …	2 Byte
Short	kurze Ganzzahl zwischen -32768 und 32767	2 Byte
Integer	Ganzzahl zwischen -2 147 483 648 und 2 147 483 647	4 Byte
Long	Lange Ganzzahl zwischen 2^{63}-1 und -2^{63}	
Single	Einfachgenaue Gleitkommazahl mit 7-stelliger Genauigkeit zwischen 10E-38 und 10E+38	4 Byte
Double	Doppeltgenaue Gleitkommazahl mit 16-stelliger Genauigkeit zwischen 10E-308 und 10E+308	8 Byte

Basisdatentypen und Variablen (3)

Standardtypen (Fortsetzung):

Typ	Erläuterung	Speicherplatz
Currency	Währung, umfasst 15 Stellen vor und 4 Stellen nach dem Dezimalpunkt	8 Byte
Date	1. Januar 100 0:00:00 bis 31. Dezember 9999 23:59:59	8 Byte
String	Zeichenfolge mit einer maximalen Länge von ca. 2 000 000 000 Zeichen	1 Byte pro Zeichen plus 10 Bytes
Object	Verweis auf ein Objekt	4 Byte

Basisdatentypen und Variablen (4)

Variablendeklaration:

DIM *name* AS *typ*

DIM und AS sind Schlüsselworte. Schlüsselworte dürfen nicht als Variablennamen verwendet werden.

Beispiele:

DIM meinGuthaben AS Currency
DIM meinGewicht AS Single

Variablendeklaration und Wertzuweisung

Getrennte Deklaration und Wertzuweisung

DIM meinGewicht AS Single

…

meinGewicht=75

Zuweisung bei Deklaration

DIM meinGewicht AS Single = 75

…

Konstanten-Deklaration

CONST pi AS Double = 3.14159

Pi kann wie eine Variable benutzt werden, solange man den Wert unverändert lässt.

pi= 47.11 → error BC30074: Zuweisung zu einer Konstanten nicht zulässig.

Einfache Anweisungen

Zuweisungen konstanter und variabler Werte:

y=3.14159

x=y

nachricht = “Die Kreiskonstante PI beträgt “

Auswertung von Ausdrücken (rechte Seite)
und Zuweisung (auf linke Seite):

*u = 2*y*r*

nachricht = “Der Umfang beträgt “+ Format(u)

Referenzieren von Objekten:

Value = TextBox1.Text

*TextBox2.Text = Value * 5.0*

Genaueres zu Objekten folgt später

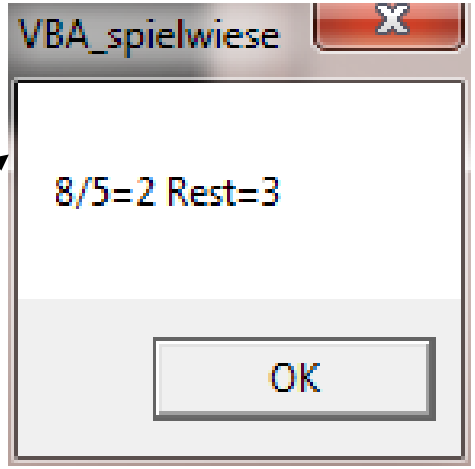
Operatoren in Anweisungen

Arithmetische Operatoren:

^	Potenzierung
* /	Multiplikation, Division
+ -	Addition, Subtraktion
\	Ganzzahl-Division ("mit Rest")
MOD	modulo ("Rest" bei der Division)
=	Zuweisung

Beispiel:

```
Dim a, b, c, d As Integer
a = 8
b = 5
c = a / b
d = a Mod b
MsgBox("8/5=" + Format(c) + " Rest=" + Format(d))
```



Benutzerschnittstelle: Eingabe (1)

Für Konsolen-Anwendungen:

DIM eingabe As String

REM Eingabe über die Konsole

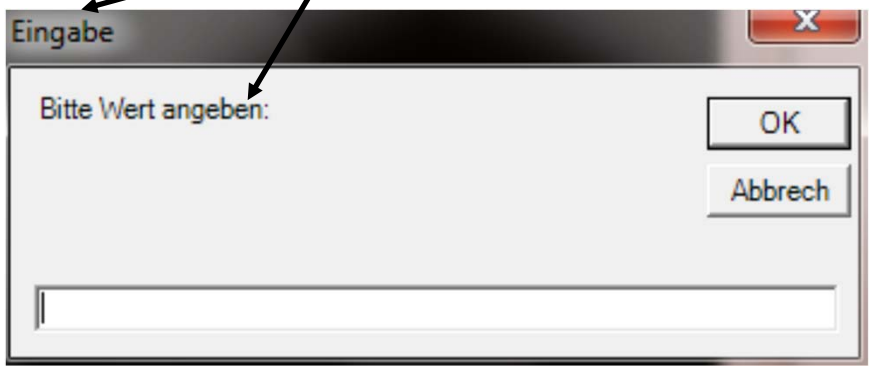
eingabe =Console.ReadLine()

…

REM Eingabe per Fenster

eingabe = InputBox("Bitte Wert angeben:","Eingabe","")

Standardwert



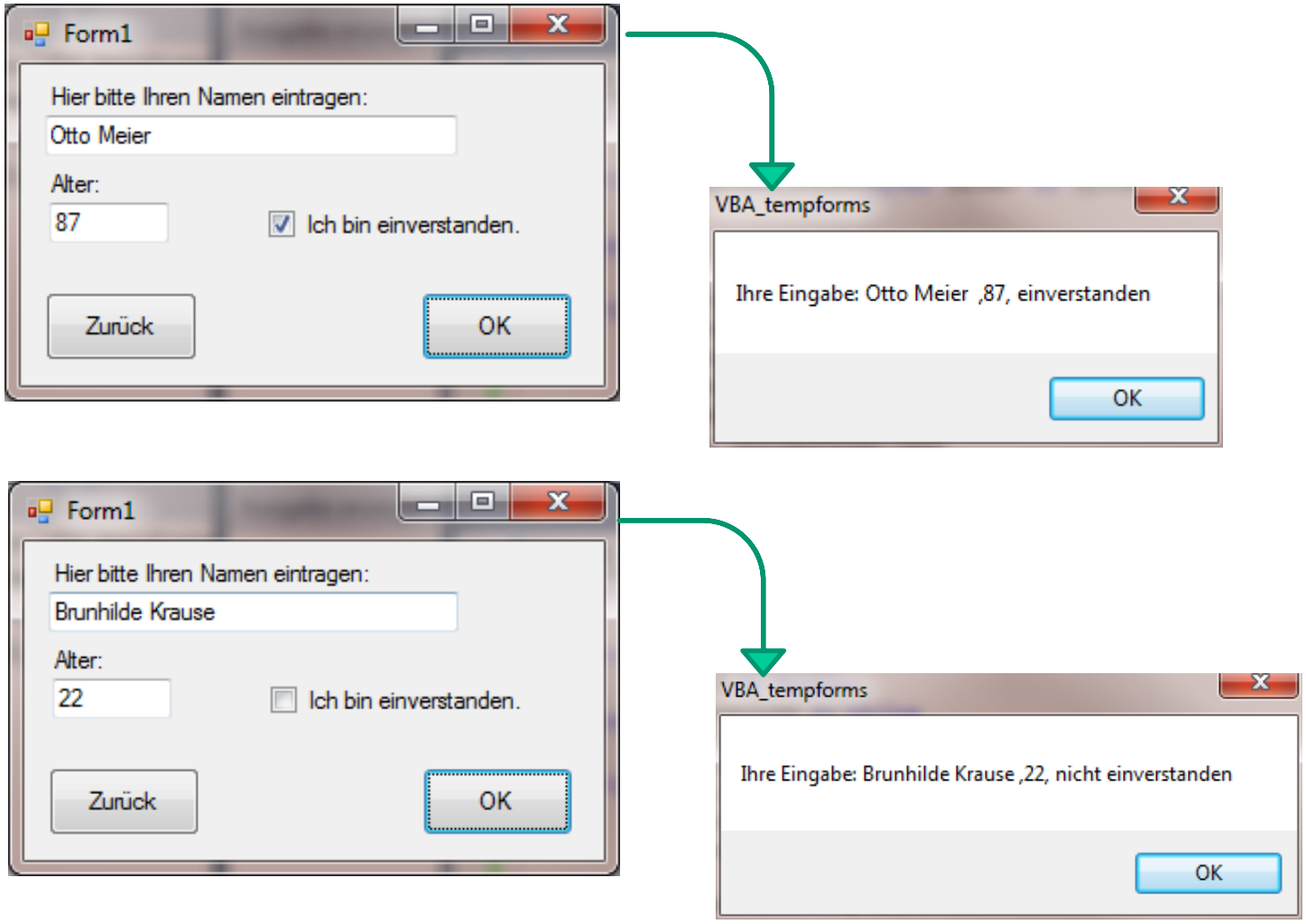
Benutzerschnittstelle: Eingabe (2)

Für Forms-Anwendungen:

```
Dim eingabe1 As String
Dim eingabe2 As Integer
Dim eingabe3 As Boolean
Dim checkNachricht As String
eingabe1 = ZeichenFeld1.Text
eingabe2 = NummerBox.Text
eingabe3 = CheckBox1.Checked
If (eingabe3) Then
   checkNachricht = ", einverstanden"
Else
   checkNachricht = ", nicht einverstanden"
End If
MsgBox("Ihre Eingabe: " + eingabe1 + " ," + Format(eingabe2) +
            checkNachricht)
```

Benutzerschnittstelle: Eingabe (3)

Für Forms-Anwendungen:



Benutzerschnittstelle: Ausgabe (1)

Für Konsolen-Anwendungen:

DIM ausgabe As String =“Hallo“

REM Ausgabe über die Konsole
Console.Write(ausgabe)
Console.WriteLine(ausgabe)
…
REM Ausgabe per Fenster
MsgBox(ausgabe)

Benutzerschnittstelle: Ausgabe (2)

Die Format()-Funktion

Man kann Variable verschiedener Typen in Text umwandeln und die Formatierung beeinflussen.

Benutzung:

Ziel = Format(Ausdruck [,Formatierung])

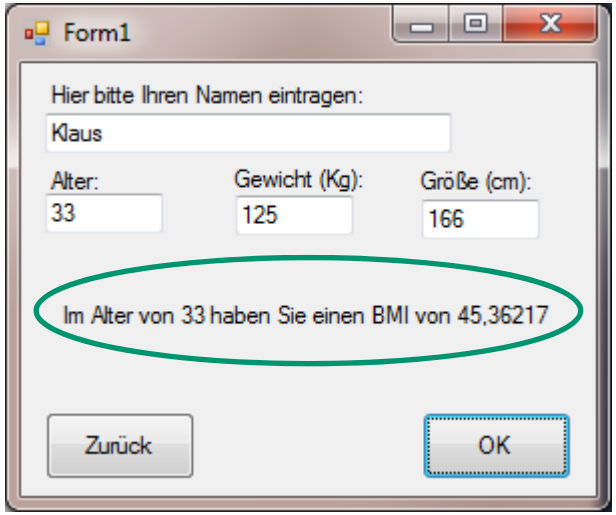
Beispiel:

DIM Temperatur As Float = 13.8

MsgBox(“ Die Raumtemperatur beträgt “+ Format(Temperatur, “ + 00.00 Grad Celsius “))

Benutzerschnittstelle: Ausgabe (3)

Innerhalb Forms-Anwendungen:

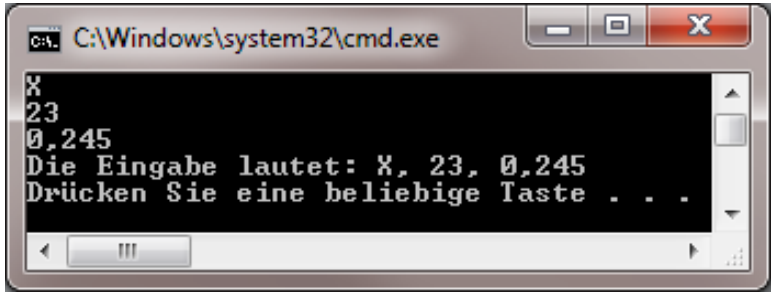



```
Dim Alter As Integer = NummerBox.Text
Dim Gewicht As Single = GewichtBox.Text
Dim Groesse As Single = GroesseBox.Text
Dim BMI As Single
Name = ZeichenFeld1.Text

BMI = Gewicht / (Groesse / 100.0 * Groesse / 100.0)

Label4.Text = "Im Alter von " + Format(Alter) + " haben Sie einen
BMI von " + Format(BMI)
```

Eingabe typisierter Variablen von der Konsole

Console.ReadLine() gibt den eingetippten Inhalt als String zurück.
Der Variableninhalt wird durch die Methode Parse() des jeweiligen Typ-Objekts ermittelt.

```
Dim zeichen As Char
Dim ganzezahl As Integer
Dim gebrochenezahl As Single
Dim eingabestring As String

REM Annahme: Reihenfolge Zeichen, ganze Zahl, gebrochene Zahl
eingabestring = Console.ReadLine()
zeichen = Char.Parse(eingabestring)
eingabestring = Console.ReadLine()
ganzezahl = Integer.Parse(eingabestring)
eingabestring = Console.ReadLine()
gebrochenezahl = Single.Parse(eingabestring)
```



Steuerfluss-Konstrukte

Der **Steuerfluss** beschreibt die Reihenfolge, in der die einzelnen Anweisungen ausgeführt werden.

Steuerfluss kann auch als Fluss oder Weg durch den Programmcode interpretiert werden.

Visual Basic ist eine imperative Programmiersprache – damit werden die Anweisungen von oben nach unten ausgeführt, wenn nichts anderes angegeben ist (Sequenz).

Steuerfluss-Konstrukte ermöglichen:

- Alternative(n)
- Zyklen
- Unterprogrammaufrufe

Realisierung durch besondere Schlüsselworte, Bedingungsausdrücke und Parameter

if-Anweisung (1)

Die if-Anweisung ist das Ausdrucksmittel für die Alternative

Variante als „einseitige“ Alternative

Bedingung	
ja	nein
Anweisung	/

if Bedingung Then Anweisung

oder

```
If Bedingung Then
        Anweisung1
        Anweisung2
        …
End if
```

Einzeilige Variante kommt ohne „End if“ aus

if-Anweisung (2)

Variante als zweiseitige Alternative:

Bedingung	
ja	nein
Anw1	Anw2

if Bedingung Then
Anw1
else
Anw2
End if

Auch mit mehreren Anweisungen: möglich

if Bedingung Then
Anw1
Anw2
else
Anw3
Anw4
End if

if-Anweisung (3)

Variante als Mehrfachauswahl

```
if  Bedingung1  Then
        Anw1
elseif Bedingung2
        Anw2
elseif Bedingung3
        Anw3
End if
```

Auch mit mehreren Anweisungen und/oder else-Zweig für „sonst“

```
if  Bedingung1  Then
        Anw1
        Anw2
elseif  Bedingung2
        Anw3
        Anw4
else
        Anw5
End if
```

Select-case-Konstrukt (1)

Zur Selektion unter mehreren alternativen Zweigen

Fallausdruck				
Wert1	Wert2	...		sonst
Anw_1	Anw_2	...	Anw_n	Anw_0

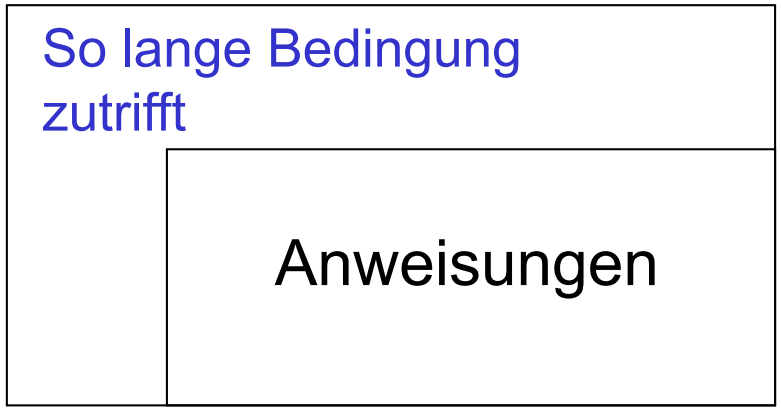
```
Select Case variable
  case Wert_1
          Anw_1
  case Wert_2
          Anw_2
  ...
   case Wert_n
          Anw_n

   case else
          Anw_0
End Select
```

Select-case-Konstrukt (2)

Weitere Variante mit Intervallen

```
Select Case variable
  case Anfang1 to Ende1
            Anw_1
  case Anfang2 to Ende2
            Anw_2
  …
   case Anfang_n to Ende_n
            Anw_n

   case else
            Anw_0;
End Select
```

do-while-Schleife

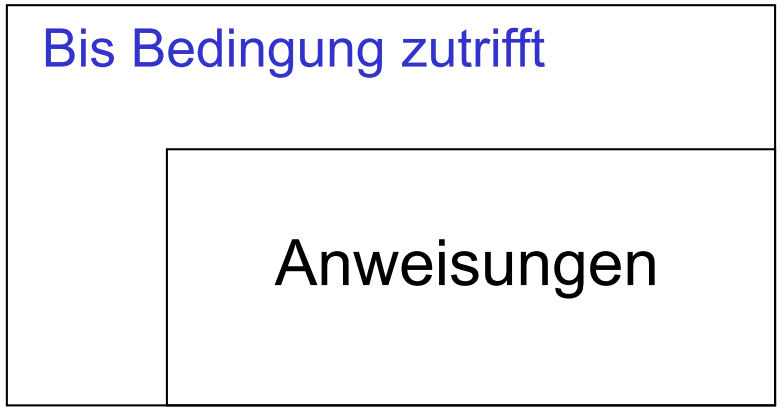


Allgemeine Form:

Do while Bedingung
anweisung1
anweisung2
Loop

- While: Die Anweisung oder Verbundanweisung wird solange wiederholt ausgeführt, wie die Bedingung zutrifft.
- Durch Änderungen der Variablenwerte wird die Bedingung i.d.R. nach endlich vielen Durchläufen irgendwann nicht mehr zutreffen und die Wiederholung endet.
- Trifft die Bedingung bei Eintritt in die Schleife nicht zu, wird die Anweisung nicht (auch nicht ein einziges mal) ausgeführt.

do-until-Schleife



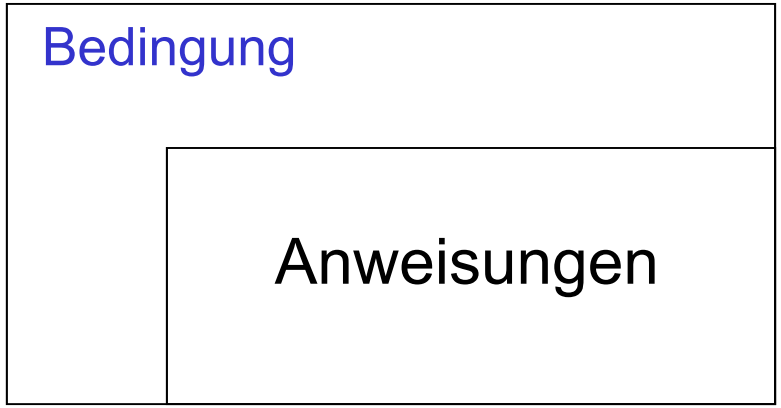
Allgemeine Form:

Do until Bedingung
anweisung1
anweisung2
Loop

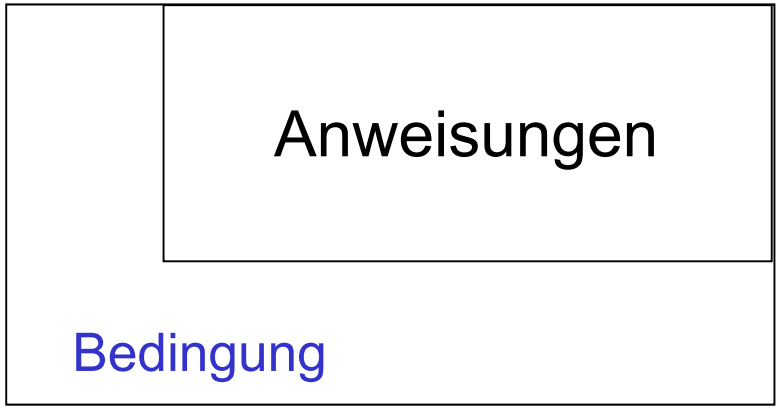
- Until: Die Anweisung oder Verbundanweisung wird solange wiederholt ausgeführt, bis die Bedingung zutrifft.
- Das Zutreffen der Bedingung ist das Abbruchkriterium

Schleifen – Kopf- vs. Fußgesteuert

Kopfgesteuert (abweisender Zyklus):



Fußgesteuert (nichtabweisender Zyklus):



Eine fußgesteuerte Schleife prüft am Ende eines Durchlaufs, ob der Anweisungsblock wiederholt wird. Der Anweisungsblock wird damit mindestens einmal abgearbeitet.

Varianten:

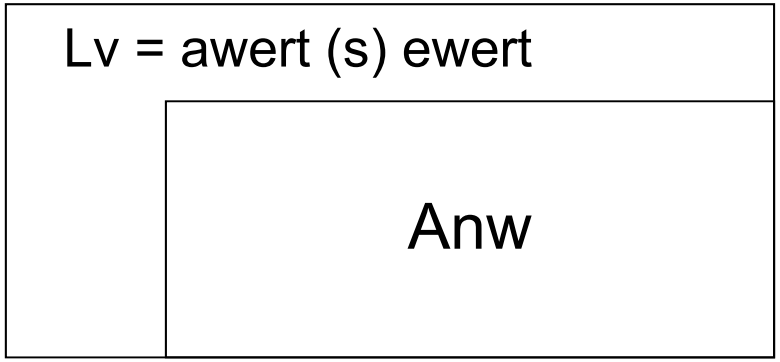
Do

…

Loop While/Until Bedignung

For-Schleife (1)

Zählzyklus:



Entsprechende Form:

FOR *Lv=awert* ***TO*** *ewert* ***STEP*** *s*
Anweisung1
…
NEXT *Lv*
Lv dient hier als Zählvariable

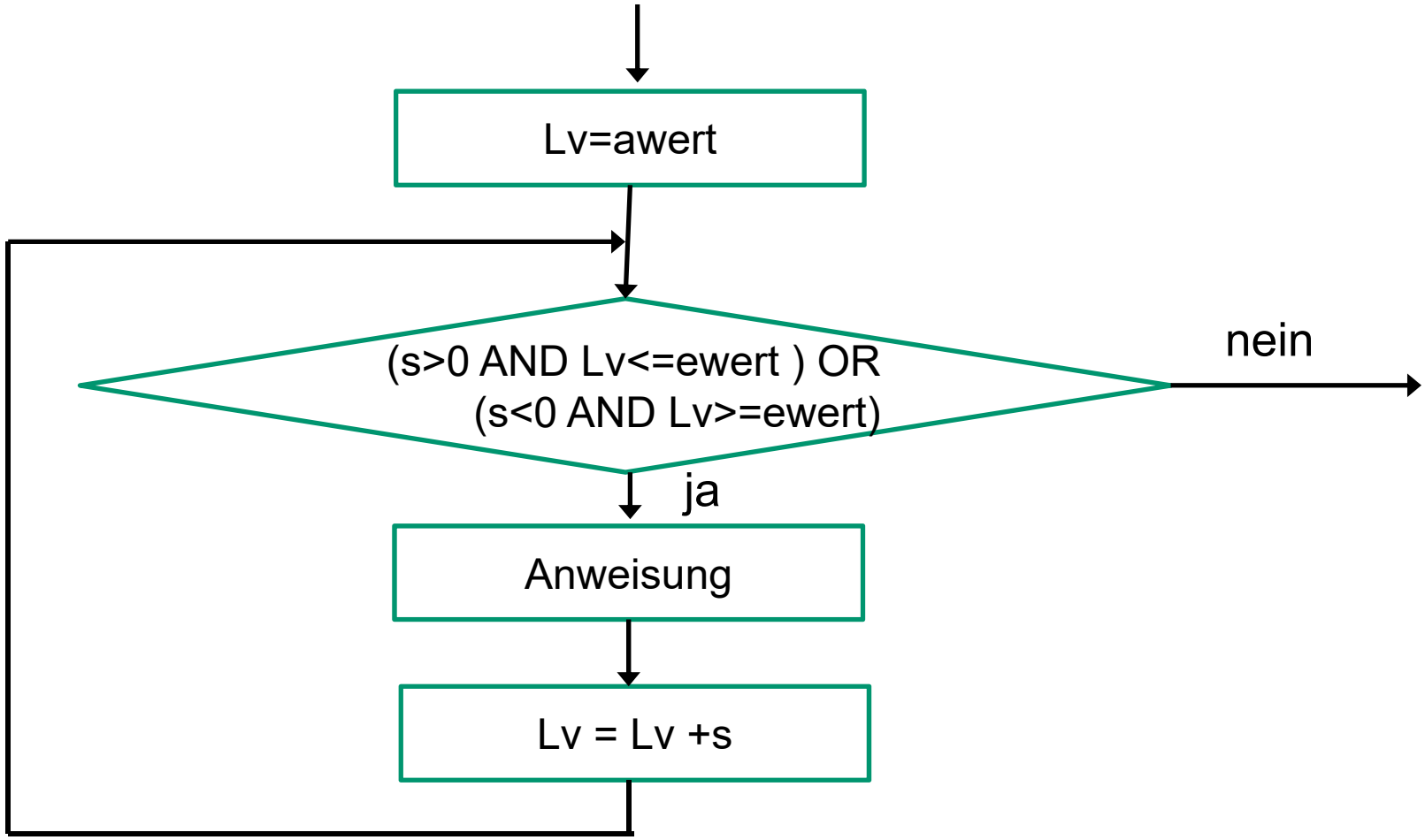
Solange die Zählvariable den *ewert* nicht über-/unterschreitet, wird die Anweisung wiederholt. STEP kann weggelassen werden, wenn die Schrittweite 1 ist.

Beispiel:

```
FOR i=1 TO 10
  Call Auswerten( datensatz(i) )
NEXT i
```

FOR-Schleife (2)

Wirkungsweise einer FOR-Schleife als PAP:



Nutzung der verschiedenen Schleifen

Die verschiedenen Varianten einer Schleife –

- Kopfgesteuertes while, do until,
- Fußgesteuertes do-while und do-until
- Zählschleife mit for

können oft alternativ verwendet werden.

Aspekte für eine sinnvolle Auswahl:

- Verwendung ***der kopfgesteuerten while/do until-Schleife***, wenn die Anzahl der Iterationen n unbekannt ist, mit n >= 0 (auch null Durchläufe möglich!)

- Verwendung der ***fußgesteuerten Schleife***, wenn im Gegensatz dazu n>=1 (mindestens ein Durchlauf!)

- Bevorzugung der ***for-Zählschleife*** dann, wenn die Anzahl der Iterationen schon bekannt ist. Insbesondere zum „Durchlaufen“ von Feldern geeignet.

Operatoren im Bedingungs-Ausdruck

Vergleichsoperatoren

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

Beispiel:

```
If (BMI <19) Then
        MsgBox("Sie sind untergewichtig!")
endif
```

Bedingungs-Ausdruck

Einzelne Bedingungen können mit logischen Operationen verbunden werden.

- AND – logisches UND,
- OR – logisches ODER,
- XOR – logisches Exklusives ODER (entweder-oder)
- NOT - Negation

Beispiele:

If ((NOT zahl < 0) AND (NOT zahl > 64)) Then ...

If ((zahl >= 0) AND (zahl <= 64)) Then …

if (zahl >= 0 AND zahl <= 64) Then …

Alle drei Beispiele bedeuten das gleiche – es wird geprüft, ob zahl im Intervall zwischen 0 und 64 liegt.

Mehr dazu im Abschnitt Aussagenlogik

Felder (1)

Felder (Arrays) speichern viele Datenelemente des gleichen Typs.
Auf einzelne Elemente kann über einen Index zugegriffen werden

Dim arrayname (dim1,dim2, …) As Typ

Bei eindimensionalen Feldern nur dim1, z.B.
Dim liste (10) As Integer … hier werden insg. 11 Elemente bereitgestellt, Indexbereich 0 bis 10

Eine Dimension kann auch als Indexintervall spezifiziert werden
Dim taegl_schlafstuden(1 TO 366) As Integer

Zuweisung auf Feldelement:
taegl_schlafstunden(1) = 10
taegl_schlafstunden(201) = 4

Referenz auf Feldelement
schlafheute = taegl_schlafstunden(111)

Felder (2)

Mehrdimensionale Felder durch Angabe mehrerer Dimensionen

Dim Matrix (0 TO 999, 0 TO 999) As Double

Oder

Dim Matrix (1000,1000) As Double

Zugriff:

Matrix(1,1)= 47.11 *pivotelem= Matrix(i,j)*

Ermittlung der Dimensionsbereiche zur Programmlaufzeit:

Funktionen *Lbound(feldname [,dimension])* und

Ubound(feldname [,dimension])

Liefern den kleinsten bzw. größten Index in der angegebenen *dimension* (kann weggelassen werden wenn 1-dim. Feld)

Felder (3)

Beispiel:

Initialisierung einer Einheitsmatrix

```
Dim matrix (100,100) as Double
…
For j=1 To 100
        For i=1 To 100
                if i=j Then
                        matrix(i,j) = 1.0
                Else
                        matrix(i,j) = 0.0
        Next i
Next j
```

Dynamische Felder

Bislang waren Felder statisch, d.h. sie hatten während der Programmlaufzeit immer eine feste Größe.

Problem: Oft ergibt sich die benötigte Feldgröße erst zur Laufzeit.

Dynamische Felder:

Dim feld() as Float … hier keine Angabe für die Dimension

…

Redim Feld (-10 TO 10) … ab jetzt kann Feld benutzt werden

Redim Feld (100) … Feld wird vergrößert, Daten gehen verloren

Redim Preserve Feld(105) … Feld wird vergrößert,
Daten bleiben erhalten

Strukturen (1)

- Strukturen gruppieren Variablen zu neuen benutzerdefinierten Typen.
- Solche Typen können dann für Variablendeklarationen genutzt werden, anstelle der Basisdatentypen (Int, Single, String usw.)
- Strukturen fassen Daten zusammen, die inhaltlich zusammen hängen. Zum Beispiel können Daten als Strukturen sortiert werden.

```
Structure strukturname
        Public element1 As typ1
        Public element2 As typ2
        …
End Structure
```

Ältere Visual Basic Systeme arbeiten mit dem Schlüsselwort TYPE statt STRUCTURE …

```
Type strukturname
        element1 As typ1
        element2 As typ2
        …
End Type
```

Benutzung und Zugriff:

```
Dim eintrag As strukturname
eintrag.element1 = x
y= eintrag.element
```

Strukturen (2)

Beispiel:

```
Structure messung
    Public datum As Date
    Public xpos As Integer
    Public ypos As Integer
    Public zpos As Integer
    Public wert As Single
End Structure
```

Benutzung und Zugriff:

```
Dim einzelmessung As messung
Dim allemessungen(0 TO 366) As  messung

einzelmessung.xpos=1
allemessungen(5).wert = 815.344
allemessungen(50) = einzelmessung
```

Enumerations-Datentyp (1)

Manchmal ist es nötig, nichtnumerische Information zu speichern, die durch eine relativ geringe Anzahl von Werten repräsentiert wird.

Aufzählungstyp (engl. Enumeration Type) als ein selbstdefinierter Datentyp

Syntax:

```
Public Enum Typbezeichner
    Wert1
    Wert2
    Wert3
    …
End Enum
```

Nach der Deklaration kann man den Typbezeichner wie einen Basisdatentyp (Int, Single usw.) benutzen.

Konstanten werden durch *Typbezeichner.Wert* angegeben.

Enumerations-Datentyp (2)

Beispiel:

```
Public Enum Einheit
    Sekunde
    Kelvin
    Joule
    Newton
    Gramm
    Meter
    Pascal
    Mol
    Ampere
End Enum
```

```
Public Enum Groesse
    Nano
    Mikro
    Milli
    Kilo
    Mega
    Giga
End Enum

Structure messwert
    Public wert As Double
    Public gr As Groesse
    Public eh As Einheit
End Structure
```

```
Dim mw1, mw2 As messwert

mw1.wert = 20

mw1.gr = Groesse.Milli

mw1.eh = Einheit.Meter
```

Prozeduren (1)

- Prozeduren sind Unterprogramme, die einen Teilalgorithmus enthalten
- Prozeduren werden einmal programmiert und i.d.R. sehr oft im Programm aufgerufen
- Prozeduren dienen der Hierarchisierung des Programms
- Eingabeparameter und Ausgabeparameter können übergeben werden

```
Sub prozedurname( Parameter )
        Deklarationen
        Anweisung1
        Anweisung2
        …
End Sub
```

Aufruf:

```
…
prozedurname ( Argumente )
```

Prozeduren (2)

Parameter:

- Übergabe von Variablen, die von der Prozedur ausgewertet werden können (ByVal – Call by Value)
- Parameter können auch der Rückgabe von Ergebnissen dienen (ByRef – Call by Reference)
- Für jeden Parameter sollte ein Typ angegeben werden

```
Sub Berechnen(ByVal a As Double, ByVal b As Double, ByRef c As Double)
        c = Math.Sqrt(a*a+b*b)
End Sub
```

Bei **„Call by Value“** -Parametern werden Prozedur-lokale Kopien der übergebenen Werte angelegt. Im Beispiel kann man mit a und b rechnen, ohne sie innerhalb der Prozedur deklarieren zu müssen. Änderungen auf a und b sind nicht nach außen sichtbar, denn sie erfolgen auf den Kopien.

Prozeduren (3)

```
Sub Berechnen(ByVal a As Double, ByVal b As Double, ByRef c As Double)
        c = Math.Sqrt(a*a+b*b)
End Sub
```

„Call By Reference“ - Parameter übergeben die Speicheradressen der als Parameter angegebenen Werte. Wird innerhalb der Prozedur der Wert der Parametervariable geändert, so ist das nach außen sichtbar. Im Beispiel ist das der Parameter c.

Aufruf:

```
Dim s1, s2, s3 As Double
s1= 50
s2 = 70
…
Berechnen( s1,s2, s3)     REM  Ergebnis entsteht auf s3
…
```

Funktionen (1)

Funktionen und Prozeduren sind ähnlich

Funktionen geben einen Wert zurück und können deshalb in Zuweisungen oder als Parameter benutzt werden.

Function *Funktionsname* ***(*** *Parameter* ***) As*** *Rückgabetyp*

Deklarationen

Anweisung1

Anweisung2

…

Funktionsname = Rückgabewert

return *Rückgabewert ` alternative Variante*

…

End Function

Die Rückgabe eines Wertes wird u.a. durch eine Zuweisung auf den Funktionsnamen realisiert. Diese Zuweisung kann beliebig oft erfolgen. Die Rückgabe ergibt sich aus der letzten Zuweisung. Eine Rückgabe mittels ***return*** (wie in den meisten anderen Programmiersprachen) funktioniert auch.

Funktionen (2)

Funktionen können wie Prozeduren eine Parameterliste enthalten.

Die Unterscheidung ByVal und ByRef ist auch bei Funktionen möglich.

Beispiel:

```
Function Berechnungsfunktion(ByVal a As Single, ByVal b As
Single) As Single
    Dim c As Single
    c = Math.Sqrt(a * a + b * b)
    Berechnungsfunktion = c    REM alternativ: RETURN c
End Function
```

Aufruf:

```
s3 = Berechnungsfunktion(s1, s2)
```

Funktionen (3)

Parameter als Strukturtyp

```
Function Add(ByVal a As messwert, ByVal b As messwert) As messwert
    Dim c As messwert
    If a.gr = b.gr Then
        If a.eh = b.eh Then
            c.wert = a.wert + b.wert
            c.eh = a.eh
            c.gr = a.gr
        Else
            REM Größen-Vorsätze beachten
            …
        End If
    Else
        c.eh = Einheit.Ungueltig
        c.gr = Groesse.Ohne
        c.wert = 0
    End If
End Function
```

Aufruf:

```
Dim mw1, mw2, mw3 As messwert

mw1.wert = 20
mw1.gr = Groesse.Milli
mw1.eh = Einheit.Meter

mw2.wert = 0.4
mw2.gr = Groesse.Milli
mw1.eh = Einheit.Meter

mw3 = Add(mw1, mw2)
```

Funktionen (4)

Feld als Parameter

```
Function Mittelwert(werte() As Double, n As Integer) As Double
    Dim sum As Double = 0

    If n = 0 Then
        Return 0
    End If

    For i = 1 To n Step 1
        sum = sum + werte(i)
    Next

    Return sum / n
End Function
```

Aufruf:

```
Dim messreihe() As Double =
    {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0}
Dim mw

mw = Mittelwert(messreihe, 10)
Console.WriteLine(Format(mw))
```

Lokale und globale Variablen (1)

Es ergeben sich Unterschiede, je nach dem wo man Variablen deklariert

- innerhalb Funktionen oder Prozeduren
- außerhalb, im Modulkontext

Beispiel:

```
Module Module1
   Dim global1 As Integer
   Dim global2 As Integer
   …
   Sub Main
   …
   End Sub
End Module
```

```
Sub Unterfunktion1()
      Dim lokal1, lokal2 As Integer
      lokal1 = global2
      lokal2 = global1
            …
End Sub
Sub Unterfunktion2()
      Dim lokal1, lokal2 As Integer
      lokal1 = 2 * global1
      lokal2 = 2 * global2
            …
End Sub
```

Lokale und globale Variablen (2)

Reservierung des Speicherplatzes

- globale Variablen werden statisch angelegt (einmal für Programmablauf)
- lokale Variablen werden für jeden Methodenaufruf eigens angelegt.

	lokale Variablen	globale Variablen
angelegt	*dynamisch*: neu bei jedem Prozedur/Funktions-aufruf	*statisch*: einmalig zu Programmbeginn
freigegeben	jeweils am Ende der Prozedur/Funktion	am Programmende

globale Variablen bleiben also über Methodenaufrufe hinweg erhalten.

Lokale und globale Variablen (3)

Sichtbarkeitsbereich (Gültigkeitsbereich, *Scope*)

- Programmstück, in dem auf eine Variable zugegriffen werden kann.
- ab Deklaration bis zum Ende des Blocks, in dem Deklaration steht.
- außerhalb dieses Blocks ist der Bezeichner nicht sichtbar.

Im Beispiel rechts:

- lokales x verdeckt in Rechne() globales x.
- Zugriff auf x in Rechne() betrifft lokales x.
- globales x lebt noch, ist in Rechne() aber nicht sichtbar (*verschattet*).

Visual Basic Quellcode:

```
Dim x, y As Integer
…
Function Rechne(ByVal a)
        Dim x As Integer
        x = 1
        Rechne = 0
        While (x <= 3)
            Rechne = Rechne + a * x
            x = x + 1
        End While
End Function
…
x = 42
y = Rechne(5)
Console.WriteLine("x=" + Format(x)
                    + " y=" + Format(y))
```

Ausgabe:

```
x=42 y=30
```

Klassen und Objekte (1)

Begriffe

Objekt – Objekte definieren sich über Eigenschaften und auf sie anwendbare Methoden. Für grafische Elemente einer Anwendung, stehen Objekte bereit. Das ermöglicht z.B. das Auslesen von Werten aus Eingabefeldern:

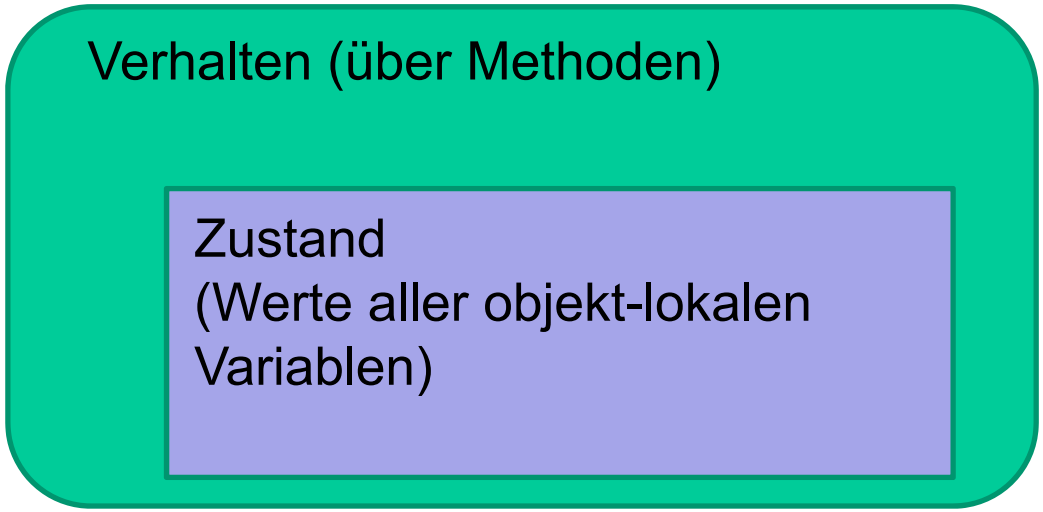
```
Dim Value As Integer
Value = Val(TextBox1.Text)
```

Eigenschaften – definieren Zustände eines Objekts. Eigenschaften können gelesen oder gesetzt werden. Alle Eigenschaften zusammen ergeben den Zustand des Objekts.

Methoden – Unterprogramme, die an einen Objekt-Typ gebunden sind und den Objektzustand ändern können

Klassen und Objekte (2)

Sicht auf ein Objekt:



Klasse:

- Softwarekonstruktion, die ein Objekt mit seinen Methoden und Variablen definiert.
- Wird oft auch als Objekttyp bezeichnet.
  Vergleichbar mit Struktur-Typ und daraus deklarierten Variablen
- Man erhält ein Objekt, indem man eine Klasse instanziiert, d.h. eine Instanz einer Klasse bildet

Klassen und Objekte (3)

Zugriff auf Eigenschaften und Methoden:

Name des Objekts, gefolgt von einem Punkt (.) und danach der Name der Eigenschaft oder der Methode

Beispiele (aus Forms-Projekten):

t1 = TagBox1.Text

DifferenzBox.Text = diff

Me.Close() REM das ist ein Methodenaufruf des Objekts Me

Me verweist immer auf das Formular, dessen Programmcode gerade ausgeführt wird.

Ziele der Objektorientierung (1)

Es sollen kurz die Hauptziele angegeben werden, die objektorientierte Sprachen charakterisieren:

Datenabstraktion: Definition und Verwendung anwendungs-bezogener Datentypen und der auf ihnen möglichen Operationen (Methoden und Operatoren).

Datenkapselung: Vereinbarung von Daten und zugeordneten Prozeduren (Methoden), die diese Daten verwalten, in einer Programmeinheit.
Zugriffe auf Daten und die Benutzung der Methoden können über Zugriffsspezifizierer einschränkt werden. Oft sind die Datenelemente eines Objektes vollständig gekapselt (private) und für den Nutzer nicht einsehbar. Nur über die öffentlichen Methoden (public) kann der Nutzer das Objekt manipulieren.

Ziele der Objektorientierung (2)

Trennung von Implementierung und Schnittstelle:
Durch die Datenkapselung wird es möglich, dass der Programmierer Freiheiten bei der internen Realsisierung der Klassen hat.
Er ist nur an die Schnittstelle und die äußere Wirkung der Methoden gebunden. Der Nutzer der Klasse benötigt in seinem Programm ausschließlich die Schnittstelle, die durch die Methoden bestimmt wird.

Damit ist ein Anwendungsprogramm vollkommen unabhängig von konkreten internen Realisierungen der Klassen.

Ziele der Objektorientierung (3)

Spezialisierung und Generalisierung (Vererbung):
Durch das Prinzip der Spezialisierung wird es möglich, aus einer Basisklasse eine weitere Klassen abzuleiten, die viele Gemeinsamkeiten mit ersterer haben und nur in einigen Details abweichen. Eine abgeleitete Klasse erbt die Datenstruktur und vererbbare Elementfunktionen von der Basisklasse. Es müssen folglich nur die Abweichungen (Spezialisierung) neu programmiert werden. Das vermindert den Kodierungsaufwand, eliminiert mehrfachen ähnlichen Code und senkt die mögliche Fehlerquote.

Polymorphismus: In Klassenhierarchien mit Vererbung kann dieselbe Methode für Objekte unterschiedlicher Typen unterschiedliche Aktionen auslösen.

Klassen und Objekte (1)

Klassen definieren:

```
Public Class Fahrzeug
   Dim v, vmax, leermasse, nutzmasse As Integer
   Dim Bezeichnung As String

   Sub New (ByVal bez as String, ByVal vm as Integer, ByVal lm As Integer)
     Bezeichnung = bez
     vmax=vm
     leermasse = lm
   End Sub

   Function ausgabe() As String
      ausgabe = Bezeichnung + “ mit Geschwindigkeit: " +   Format(v)
   End Function
   …
End Class
```

Beispiel angelehnt an:
http://openbook.galileocomputing.de/einstieg_vb_2010

Klassen und Objekte (2)

Klassen definieren (Fortsetzung):

```
Public Class Fahrzeug
  …
   Sub beschleunigen(ByVal wert As Integer)
      v = v + wert
      if v>xmax Then v=vmax
   End Sub
   Sub bremsen(ByVal wert As Integer)
      v = v - wert
      if v<0  Then v=0
   End Sub
End Class
```

Beispiel angelehnt an:
http://openbook.galileocomputing.de/einstieg_vb_2010

Benutzerdefinierte Objekte (3)

Benutzerdefinierte Objekte instanziieren:

Dim stadtrad,rennrad As fahrzeug

stadtrad = New fahrzeug("Stadtrad", 30, 15)

REM hier mal ein träges Fahrrad, mit vmax=30 Km/h und 15 Kg

rennrad = new fahrzeug("Wettkampfrad", 55, 8)

REM ein leichteres und schnelleres Rad

Dim kleinwagen As New fahrzeug("Fiat Topolino",80, 650)

REM Deklaration und Instanziierung in einem Schritt

Benutzerdefinierte Objekte (4)

Benutzerdefinierte Objekte benutzen:

stadtrad.beschleunigen(10)
rennrad.beschleunigen(25)
stadtrad.bremsen(10)
…
kleinwagen.beschleunigen(50)
rennrad.beschleunigen(20)
…
kleinwagen.bremsen(25)
…

Am Ende sind Objekte freizugeben

stadtrad = Nothing
rennrad = Nothing