Praktische Informatik für Wirtschaftsmathematiker, Ingenieure und Naturwissenschaftler I (PIWIN I, 3 V + 1 Ü) WS 2002/03

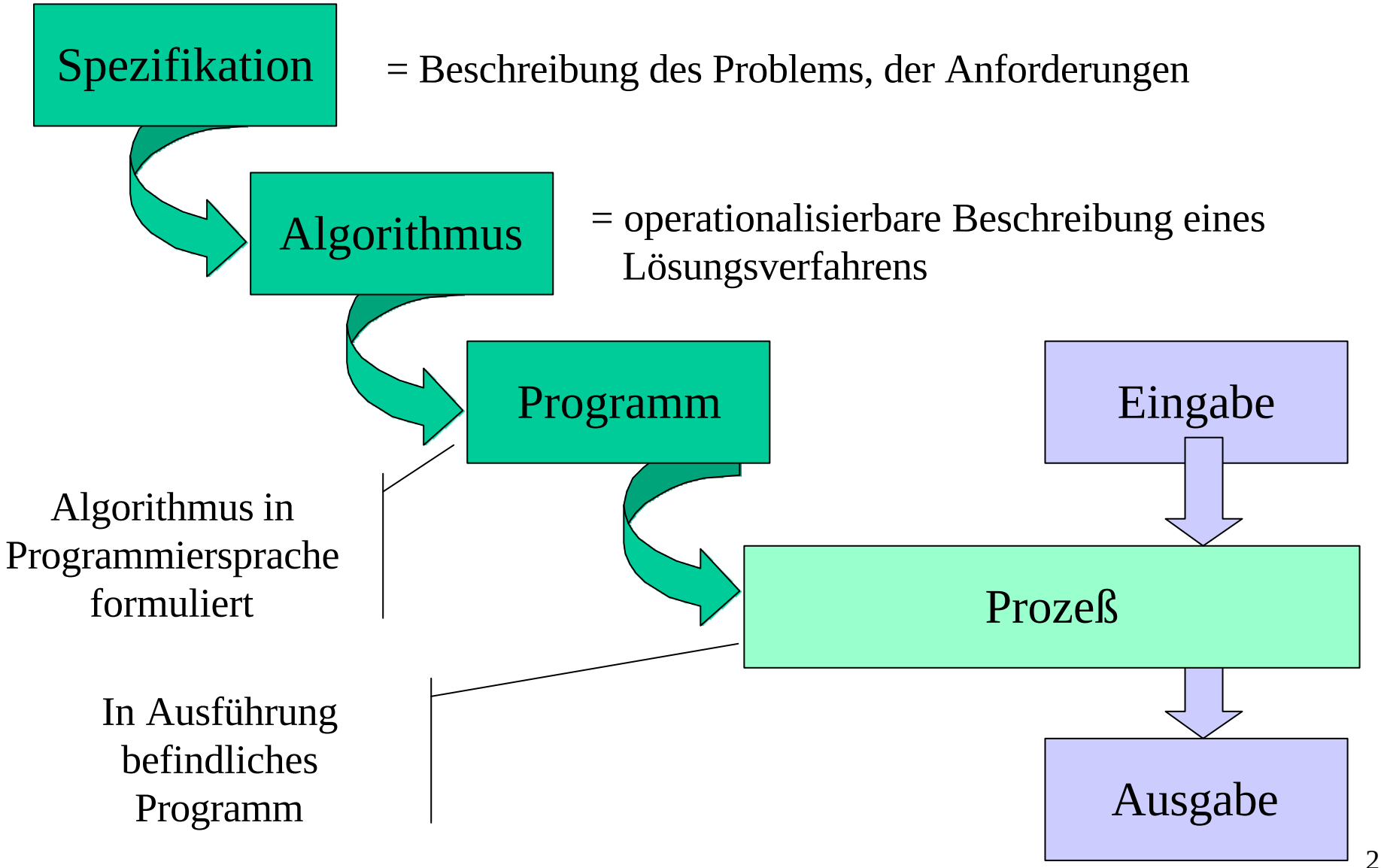
3. Vorlesungswoche

Basiskonstrukte imperativer und objektorientierter Programmiersprachen

Unterlagen: Gumm/Sommer, Kapitel 2

Echtle/Goedicke, Einführung in die Programmierung mit Java, dpunkt Verlag

Stationen im Entwurf von Algorithmen und Programmen



Übersicht

- Begriffe
  - Spezifikationen, Algorithmen, formale Sprachen, Grammatik
- Programmiersprachenkonzepte
  - Syntax und Semantik
  - imperative, objektorientierte, funktionale und logische Programmierung
  - formale Sprachen und Grammatik
- Grundlagen der Programmierung
  - imperative Programmierung:
    - Verfeinerung, elementare Operationen, Sequenz, Selektion, Iteration, funktionale Algorithmen und Rekursion, Variablen und Wertzuweisungen, Prozeduren, Funktionen und Modularität, Zuweisung, Sequenz
  - objektorientierte Programmierung
- Algorithmen und Datenstrukturen
- Berechenbarkeit und Entscheidbarkeit von Problemen
- Effizienz und Komplexität von Algorithmen
- Programmentwurf, Softwareentwurf

Kern imperativer Sprachen

- Zuweisungen
  - Variable speichern / tragen / beinhalten Werte
  - Zuweisungen müssen typverträglich sein
    - Variable haben einen Typ, daher zunächst (einfache) Datentypen und Operationen
- Kontrollstrukturen
  - Sequentielle Komposition, Sequenz
  - Alternative, Fallunterscheidung
  - Schleife, Wiederholung, Iteration
- Verfeinerung
  - Unterprogramme, Prozeduren, Funktionen
  - Blockstrukturierung
- Rekursion

Variable : eine mathematische Sichtweise

- Variable als Stellvertreter für einen unbekannten Wert

  z.B. Lösung linearer Gleichungssysteme

  a = b + 3, b = 2 * c, c = 13
- Variable in Verbindung mit Gleichungen erlauben Ersetzung, Einsetzen gleichwertiger Beschreibung für eine Variable

  z.B. a = (2 * 13) + 3 = 29
- Gleichungen erlauben Operationen, die die Lösung unverändert lassen

  z.B. a = b + 3 gdw a - 3 = b gdw a - b = 3

Achtung: in Programmiersprachen werden Variable und „=“ ganz anders verstanden und behandelt !!!

Variable: die Sichtweise in der Informatik

- In der Informatik steht eine Variable für einen Speicherplatz, der eine bestimmte Art von Datum aufnehmen kann.
- Deklaration: Programmiersprachen verlangen i.d.R. dass die Art des Datums festgelegt wird
  - Eine Variable hat einen (Daten-)Typ
  - Einfache Datentypen werden in jeder Sprache bereitgestellt
    - Zeichen: char
    - ganze Zahlen: byte, short, int, long
    - Fließkommazahlen: float, double
  - Deklaration erfolgt textuell vor der Verwendung einer Variable und durch Angabe des Typs, des Namens/Bezeichners und ggfs eines initialen Wertes
  - z.B.: int i = 5 ; deklariert einen Speicherplatz zur Aufnahme eines ganzahligen Wertes „int“, dieser Speicherplatz wird im folgenden mit „i“ bezeichnet und der Wert „5“ initial belegt.
- daher: Deklaration, Initialisierung vor Verwendung einer Variable

Syntax Diagramme für die Deklaration von Variablen und Anweisungen

Variablen können direkt in der Deklaration mit einem Wert initialisiert werden

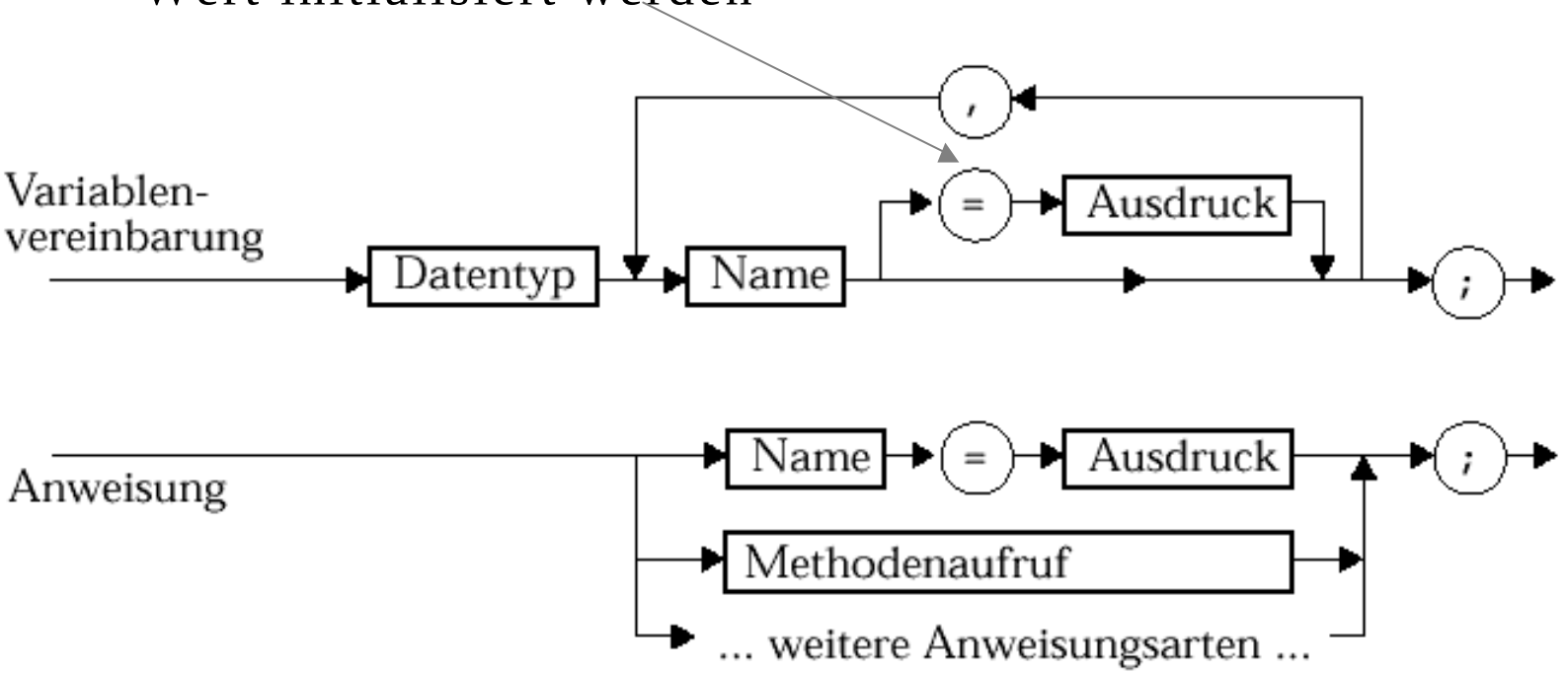


Abb. 2–2 *Syntaxdiagramme*

Lehrbuch der Programmierung mit Java, Echtle Goedicke, Heidelberg, © dpunkt 2000

(©M. Goedicke, UGH Essen)

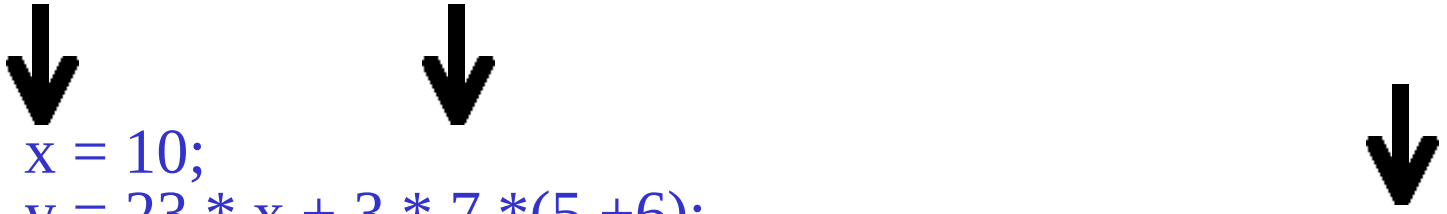
Einfache Datentypen

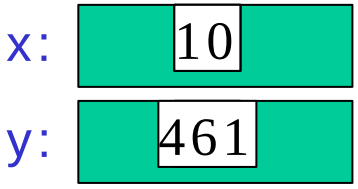
- Eine Programmiersprache stellt einen Vorrat an einfachen Datentypen bereit:
- z.B. integer
  - Wertebereich (4 Byte): -2^{31} ... 0 ... $2^{31}-1$
  - Operationen: +, -, *, /, %
  - Vergleiche: ==, !=, >, >=, <, <=
  - vordefinierte Methoden, z.B. : Math.min, Math.max, Math.abs
  - Konstante: z.B. 123, aber auch Integer.MAX_VALUE, MIN_VALUE

- analog: byte, short, long
- des weiteren:
  - float, double für Gleitpunktzahlen
  - ...

Zuweisung: Variablen werden Werte zugewiesen

- Beispiel:
  Zuweisung einer Konstanten x = 10;
  Zuweisung des Resultates eines arithmetischen Ausdrucks
  y = 23 * x + 3 * 7 *(5 +6);
- Schrittnummern wie in den ersten Algorithmen werden nicht benötigt:
  ; bedeutet Hintereinanderausführung / Sequenz
- Die beiden Anweisungen oben sind Zuweisungen an Variablen, d.h. die durch sie repräsentierten Speicherplätze haben am Ende der Ausführung einen neuen Wert.
- Der alte Wert ist unwiederbringlich verloren!





(©M. Goedicke, UGH Essen)

Grundsätzlich wird durch eine Zuweisung ein errechneter Wert in einer Variablen abgelegt

- Der grundsätzliche Aufbau:
  Variablenname = Ausdruck
  ```
  b = 5*27;
  ```
- Die Verwendung des gespeicherten Wertes geschieht durch die Angabe des Variablennamens in einem Ausdruck
  ```
  a = b *8;
  ```
- Beachte: die Verwendung von Variablennamen auf der linken und der rechten Seite eines Ausdrucks hat daher unterschiedliche Bedeutung!

Der Ablauf der Zuweisung besteht aus insgesamt drei Schritten

```
linkeSeite = rechteSeite ;
```

1. Die linke Seite der Zuweisung wird ausgewertet ... hier der Variablenname ... kann aber komplexer sein
2. Die rechte Seite (Ausdruck) wird ausgewertet ... Regeln zu Operatorreihenfolgen etc. werden beachtet
3. Falls der Wert der rechten Seite typkompatibel zu der Variablen ist oder automatisch (d.h. implizit) angepasst werden kann erfolgt die Zuweisung

Die genannten drei Schritte laufen nur dann ungestört ab, wenn keine Ausnahmen registriert werden

```
linkeSeite = rechteSeite ;
```

1. Die genannte Variable könnte nicht vorhanden sein
2. Die Auswertung der rechten Seite liefert einen Fehler
3. Typ der Variablen und des Ergebnisses sind nicht typkompatibel

(©M. Goedicke, UGH Essen)

Programmiersprachen erlauben oft eine Reihe abstruser Spezialkonstrukte, die als „Komfort“ verstanden werden ...

linkeSeite2 = (linkeSeite1=rechteSeite1) op rechteSeite2 ;

- Nur zum Verständnis (!) anderer Programme:
  eine Zuweisung ist auch ein Ausdruck!
- Der zugewiesene Wert einer Zuweisung ist der „Wert“ einer Zuweisung als Ausdruck betrachtet

```
int  a = 1,  b = 2,  c,  d;

a = 3 + 2 * (a + b);

b = 7 - a/2;

c = 3 * (d = a + b + 1);
```

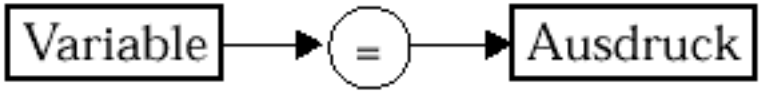
- Mehrere Zuweisungen in einem Ausdruck werden von rechts nach links abgearbeitet:

```
a=b=c=d=5;          a=(b=(c=(d=5)));
```

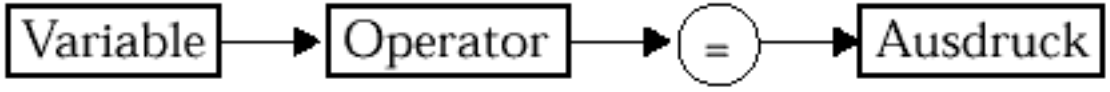
Und noch ein Beispiel:

- Es gibt einige Kurzversionen für häufig vorkommende Zuweisungen
- Z.B.: Erhöhung einer Variable um einen Wert
  a = a + 5; kann geschrieben werden a += 5;

Zuweisung



Spezielle Kurznotation einer Zuweisung



steht abkürzend für:

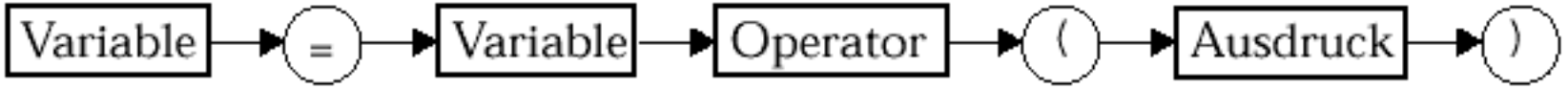


Abb. 2–6 *Syntax der Zuweisung*

Lehrbuch der Programmierung mit Java, Echtle Goedicke, Heidelberg, © dpunkt 2000

(©M. Goedicke, UGH Essen)

Die Zuweisungen verändern den Speicherinhalt.

Besonderheiten, Auswirkungen:

- Ringtausch: Vertausche den Wert der Variablen a und b

  Variante 1:

  ```
  a = b ;
  b = a ;
  ```

  liefert nicht das erwünschte Ergebnis! Warum ?

  Variante 2: verwendet Hilfsvariable c

  ```
  c = a ;
  a = b ;
  b = c ;
  ```

  funktioniert ! Warum ?

- Zuweisen impliziert Kopieren
  - Variable a beschreibt Kontostand des Kunden A, z.B. a == 1000
  - Variable b beschreibt Kontostand des Kunden B, z.B. b == 0
  - A überweist 1000 GE an B, durch b = a erhält B den Betrag ?

  Soll ein Datum von einer Variable zur anderen nicht kopiert, sondern verschoben werden, muß es an der Ausgangsvariable zerstört werden.

Was können Variable verwalten ?

- Primitive Datentypen
  - die von einer Programmiersprache bereits fest vorgegeben werden
- Konstruierte, komplexe Datentypen
  - die mit Hilfe einiger weiterer Konstrukte deklariert und genutzt werden können, (-> später mehr)
- Zur Erinnerung:
  - Variable beinhaltet: Namen, Speicherort, Typ, Inhalt
- Warum kann der Inhalt einer Variable nicht der Speicherort einer anderen Variable sein ?
  - Idee: „ich weiß es nicht, aber ich weiß wo es steht“
  - also: eine Variable speichert eine Speicheradresse, daher kann man mit dem Wert der Variable als Adresse im Speicher auf den Inhalt dort zugreifen
  - in C: Zeiger / Pointer, in Java: abgeschwächt als Referenz

  (-> später mehr)

Nochmals

- Großer Unterschied zum Variablen-Begriff in der Mathematik!
  - In der Mathematik: Variablen repräsentieren beliebigen aber festen Wert, der ggfs. auch noch unbekannt sein kann
  - In Programmen: Variablen repräsentieren Speicherplätze, die je nach Zuweisung ihren Wert ändern können und müssen den Variablen explizit zugewiesen werden (keine impliziten oder unbekannten Werte von Variablen!)
- Daher auch ein großer Unterschied bei der Bedeutung des Gleichheitszeichens (=)
  - In der Mathematik: bezeichnet die (algebraische) Gleichheit von linker und rechter Seite einer Gleichung ... algebraische Umformungen
  - In Programmen:Zuweisung (in anderen Programmiersprachen auch mit := bezeichnet) linke und rechte Seite haben unterschiedliche Bedeutung!!!

(©M. Goedicke, UGH Essen)

Weitere Bemerkungen zu Variablen in Programmen (2)

- Besonderheit von Java (und vergleichbaren Programmiersprachen)

  Die Bedeutung der Operationszeichen in Ausdrücken hängt von den Typen der beteiligten Variablen ab:

```
int x;
x = 10 + 5;
System.out.print("Der Wert von x:" + x);
```

Addition zweier int - Werte

Verkettung zweier Zeichenketten, wobei die zweite aus der Umwandlung eines int - Wertes (x) gewonnen wird

(©M. Goedicke, UGH Essen)

Datentypen

- Datentypen fassen die Wertemenge und die zulässigen Operationen auf den Werten zu *einem* Begriff zusammen
  z.B. macht es wohl wenig Sinn in dem Ausdruck

  `13 * X + X/2` für die Variable `X` den Wert `"Hallo "` einzusetzen!
- In Programmiersprachen wie Java muss daher alles zusammenpassen:
  - Typ einer Variable,
  - der Wert der dort gespeichert ist und
  - die Operationen, die auf diesem Wert angewendet werden

(©M. Goedicke, UGH Essen)

Ein sehr wichtiger Datentyp sind die Wahrheitswerte `true` und `false`

Beispiele für Variablen des Datentyps

```
boolean angemeldet, bezahlt, storniert;
angemeldet = true;
bezahlt = false;
```

Außerdem gibt es die Operationen <, >, <=, >=, ==, != die auf den numerischen Datentypen definiert sind und Werte aus dem Datentyp `boolean` liefern.

Operationen auf Werten des Datentyps `boolean` sind

log. oder	`||`,
log. und	`&&`
log. nicht	`!`

(©M. Goedicke, UGH Essen)

Beispiel ... (Prog 2-9)

```
public class booleanBeispiel {

  public static void main(String args[]) {
  float gemessenerDruck = 5.0f,
        Maximaldruck = 18.8f;
  int      ZuflussStufe = 2, AbflussStufe = 3;
  boolean  Ueberdruck, unkritisch;
    Ueberdruck = gemessenerDruck > Maximaldruck;
    unkritisch =    Ueberdruck
          && gemessenerDruck < 1.2f * Maximaldruck
          && ZuflussStufe <= AbflussStufe;


    System.out.println( "Überdruck: "+ Ueberdruck +
                  " unkritisch: "+ unkritisch );
            }
}
```

(©M. Goedicke, UGH Essen)

In dem Beispiel (Prog 2-9) werden boolesche und arithmetische Ausdrücke gemischt!

Dazu muss eine eindeutige Prioritätsregelung für die verschiedenen Operationen geschaffen werden!

1. Höchste Priorität genießen die unären Operatoren positives (+) und negatives (-) Vorzeichen sowie das boolsche Komplement (!).
2. Multiplikative Operationen (Punktrechnungen * / %)
3. Additive Operationen (Strichrechnungen + -)
4. Vergleiche (== != < > <= >=)
5. Und-Verknüpfung (&&)
6. Niedrigste Priorität besitzt schließlich die Oder-Verknüpfung (||).

Es hilft Klammern zu setzen!

(©M. Goedicke, UGH Essen)

Das folgende Syntaxdiagramm liefert die möglichen Ausdrücke

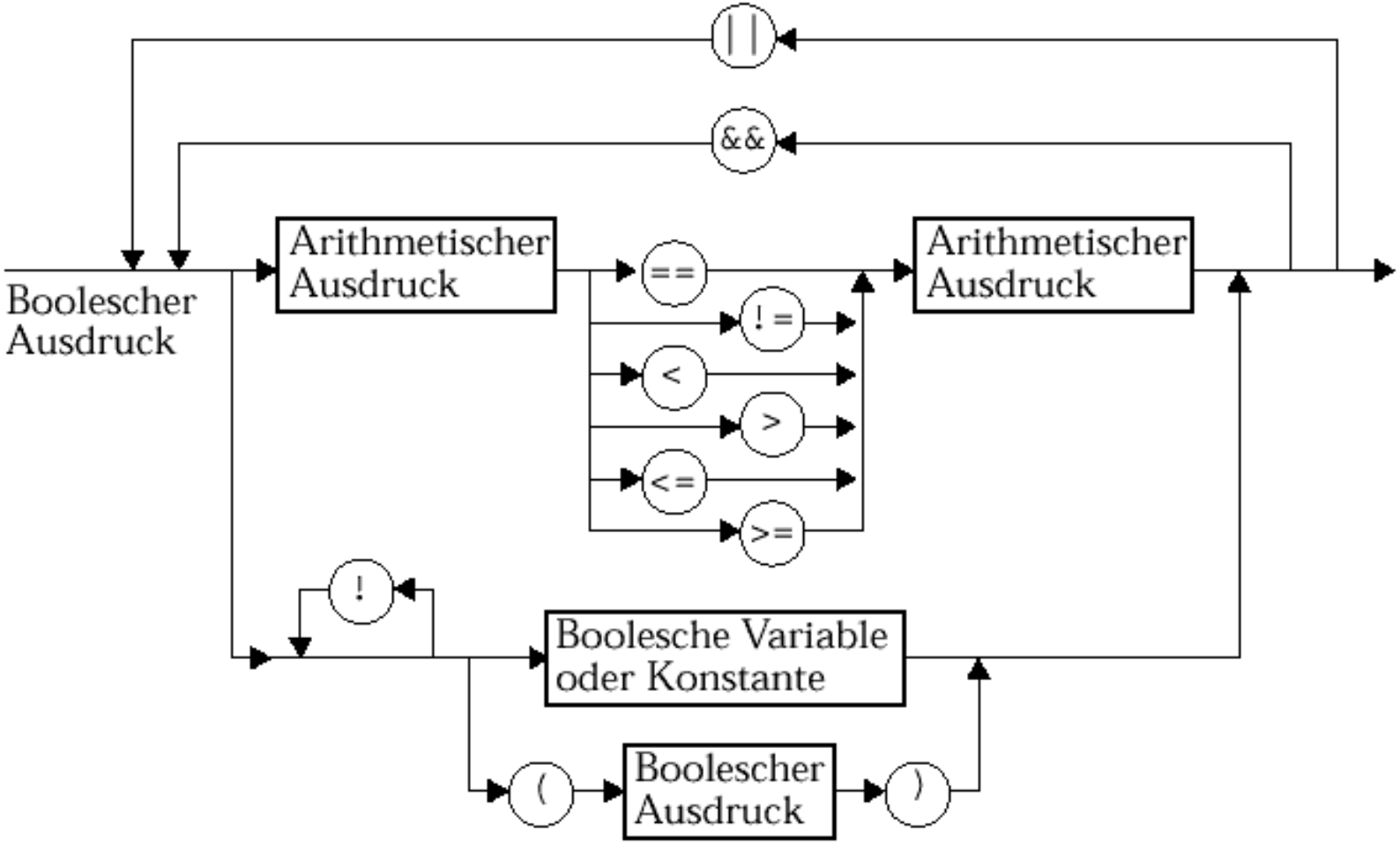


Abb. 2–4 *Syntaxdiagramm eines Booleschen Ausdrucks* (©M. Goedicke, UGH Essen)

Lehrbuch der Programmierung mit Java, Echtle Goedicke, Heidelberg, © dpunkt 2000

Die einzelnen Zeichen in Zeichenketten sind in dem Datentyp `char` definiert (1)

- Der Wertebereich umfasst die Groß- und Kleinbuchstaben, Sonderzeichen und Spezialzeichen (Steuerzeichen z.B. Leerzeichen)
- Konstanten werden in einfache Hochkomma ’ gesetzt: ‘a‘ ‘Ä‘ ‘?‘ ....
- Die Zeichen werden alle in einer Tabelle (Unicode) mit Nummern versehen ... die Bereich der Buchstaben und Ziffern sind zusammenhängend

(©M. Goedicke, UGH Essen)

Die einzelnen Zeichen in Zeichenketten sind in dem Datentyp `char` definiert (2)

- Steuerzeichen werden in einer Spezialnotation angegeben (sog. Escape-Sequenzen \ ):

  ‘`\n`‘ Zeilenvorschub, ‘\t‘ Tabulator
  ‘`\‘`‘ für ‘ , ‘`\“`’ für ” , ‘`\\`‘ für \

- Die Vergleichsoperationen sind für `char` auf der Basis der Unicode-Tabelle definiert

```
char rund = ‘(‘, eckig = ‘[‘;
boolean x = rund < eckig ;
```

(©M. Goedicke, UGH Essen)

Zeichenketten sind im Datentyp `String` definiert und ist *nicht* primitiv in Java

- In der Sprache Java spielen Werte (=Objekte) vom Typ `String` jedoch eine gewisse Sonderrolle
- Konstanten können direkt angegeben werden:
  `"Der Mond scheint blau \n "`
- Zeichenketten können mittels + verkettet werden
- Beliebige Datentypen können in Strings umgewandelt werden, wenn sie als Parameter von `println` auftauchen
- Als Vergleichsoperationen sind nur == und != zugelassen

(©M. Goedicke, UGH Essen)

Der Vergleich von Objekten vom Typ `String` ist allerdings mit Vorsicht zu genießen!

Beispiel Prog 2-11

```
public class StringVergleich {


    public static void main(String args[]) {
    String a = "merkwürdig", b = "merkwürdig",
    c = new String ("merkwürdig"),
    d = new String ("merkwürdig");
    System.out.println (a + " " + (a==b) + " " + c + " " + (c==d));
    }
}
```

(©M. Goedicke, UGH Essen)

In diesem Beispiel werden vier Objekte vom Typ `String` benutzt

Beispiel Prog 2-11

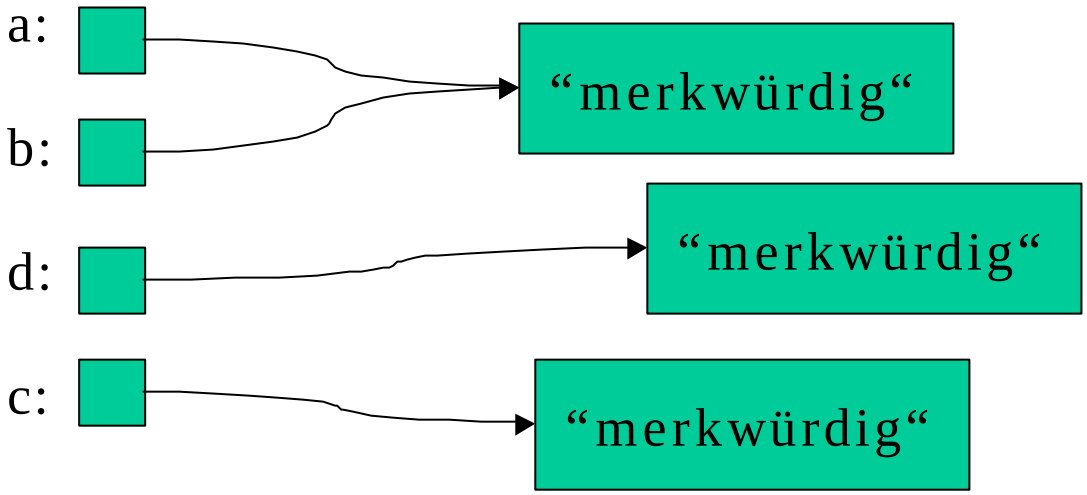
```
public class StringVergleich {

    public static void main(String args[]) {
    String a = "merkwürdig", b = "merkwürdig",
   c = new String ("merkwürdig"),
   d = new String ("merkwürdig");
   ...
```

Die letzten beiden `c, d` sind aber unterschiedliche Objekte .... Und daher … liefert

`==` zwischen unterschiedlichen Objekten hier false! (siehe Programmausführung!)

(©M. Goedicke, UGH Essen)

Die Situation zwischen den Variablen a,b,c und d läßt sich durch Kästen und Pfeile charakterisieren



new erzeugt neue separate Objekte und in diesem Fall mit dem gleichen Inhalt.
Während die Deklaration von a und b aus Ersparnisgründen auf ein und dasselbe Objekt verweisen.

(©M. Goedicke, UGH Essen)

Das gesamte System von Datentypen in Java wird als *streng* bezeichnet

- Jede Variable in Java muss mit einem Typ deklariert werden, der festlegt
  - welche Werte die Variable aufnehmen kann
  - welche Operationen auf diesen Werten anwendbar sind
- Hilft bei der Überprüfung vieler einfacher Fehler!
- Ist gelegentlich hinderlich, wenn man offensichtliche Gemeinsamkeiten von Datentypen ausnutzen möchte (z.B. Werte aus `short` und `int` verknüpfen)

(©M. Goedicke, UGH Essen)

Daher wird der Begriff der *Typkompatibilität* eingeführt

- Man wird also etwa `3.5*x` als
  - *typkompatibel* bezeichnen, wenn `3.5` (`float`) und `x double` ist
  - *nicht typkompatibel* bezeichnen wenn `x boolean` ist
- Desweiteren betrachte den Operator `+`
  - Bezeichnet Addition zwischen den numerischen Typen
  - String-Verkettung zwischen Strings

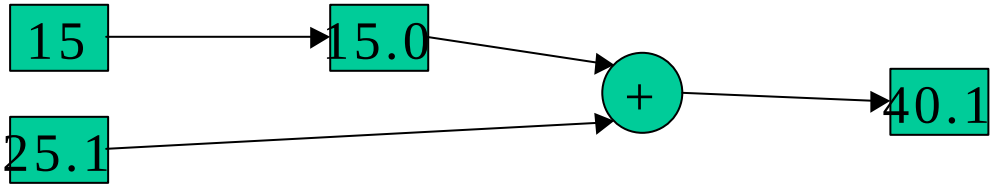
(©M. Goedicke, UGH Essen)

Das bedeutet, dass die Operanden von + auch die Ergebnistypen bestimmen

- Z.B. `int + int -> int` etc.
- Ausnahme:

  `byte + byte -> int`

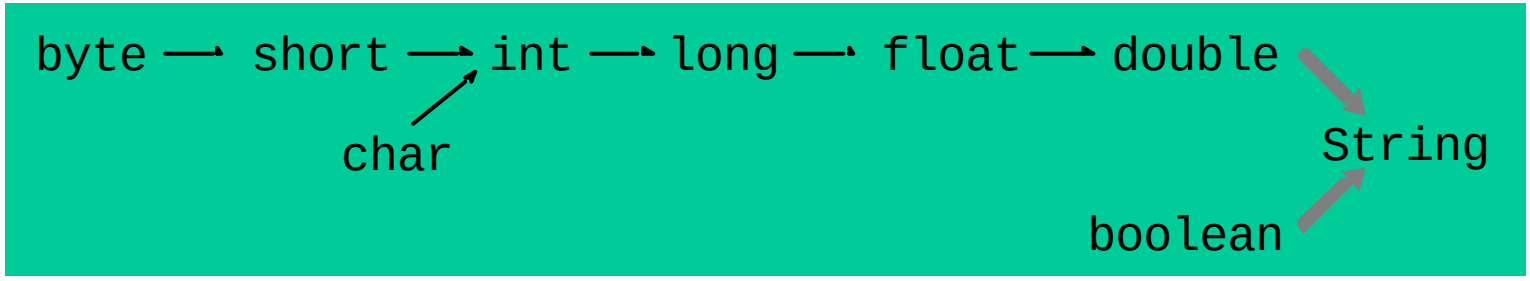
  `short + short -> int`
- Desweiteren 15+ 25.1 .... ist in Java erlaubt, aber es muss eine Regel her, die das Ergebnis bestimmt
- Es wird kein neuer Operator `int + float` eingeführt, sondern:



(©M. Goedicke, UGH Essen)

Diesen Vorgang nennt man *implizite Typanpassung*

- Ist (in Java) nicht zwischen beliebigen Typen möglich
- Die Umwandlung geschieht nur in Richtung des allgemeineren Typs hin
  > `int` und `boolean` bleiben nicht kompatibel
  > `int` -> `float` hingegen schon
- Insgesamt gilt folgende Regelung in Java(!):



(©M. Goedicke, UGH Essen)

Hier ist noch eine Besonderheit bzgl. des grauen Pfeils (-> `String`) zu beachten

- Bereits in den ersten Beispielprogrammen:
  `System.out.println("Wert:"+ a);`
- Hier wird keine in dem obigen Sinne implizite Datentyp-Umwandlung veranstaltet, sondern eine explizite mit Hilfe der Methode `toString`
- `toString` ist im Java-System definiert und kann vom Programmierer verändert werden
- Der Java-Compiler sorgt dafür, dass in Ausdrücken der Form `xxx + stringstring`
  für `xxx` zunächst das zugehörige toString aufgerufen wird

(©M. Goedicke, UGH Essen)

Da ist in Ausdrücken die Reihenfolge der Operatoren zu beachten

- Beispiel:

  `System.out.print(17 + " und " + 4)` liefert: `17 und 4`

  während

  `System.out.print(17 + 4 + " und")` liefert `21 und`

- Ansonsten arbeiten die Umwandlungsregeln intuitiv ...
- Gelegentlich sind auch Umwandlungen 5.4 -> 5 sinnvoll ...

(©M. Goedicke, UGH Essen)

Umwandlungen dieser Art können mit Ungenauigkeiten und Verlust an Information verbunden sein

- Ist möglich, aber nur auf explizite Anweisung hin
- ... Das geschieht durch Voranstellen des gewünschten Ziel-Typs in Klammern:

```
float x = 82.2f;
int n = (int) x;
```

- Hier wird der Nachkomma-Teil abgeschnitten
- siehe Beispiel (Prog 2-12)
- Bitte solche Programme möglichst NIE schreiben!!!

(©M. Goedicke, UGH Essen)

Die Umwandlungsregeln beziehen sich auf sinnvolle Beziehungen zwischen den verschiedenen Datentypen

- Zahlen und Zeichen (auf der Basis der Unicode-Tabelle)
- Nicht alles was sinnvoll wäre geht auch in Java:

  `(int) "345"` geht nicht
  muss durch
  `Integer.parseInt("345")` erledigt werden

- Falls eine Konversion nicht existiert-> Exception

(©M. Goedicke, UGH Essen)

Informationsverlust droht u.a. bei Wertebereichsüberschreitungen

- Leider entdeckt Java dieses Problem nicht automatisch:

  `(short) 100000` liefert `-31072`

(©M. Goedicke, UGH Essen)

Zwischenstand

- Zuweisung
- Variable
- Datentypen, Typumwandlung

- Zuweisungen können aneinandergereiht werden: Sequenz
  - Trennzeichen zwischen Anweisungen: **;**
  - z.B. a = 3 ; b = a + 4

- Kontrollstrukturen
  - Verzweigung, Alternative, Fallunterscheidung
  - Schleife
  - Funktion, Prozedur, Methode
  - Rekursion

Verzweigungen steuern den Programmablauf abhängig von Bedingungen

- Zuweisungen werden von Bedingungen abhängig sein (Konto, Gewinne, ...)
- Bedingungen werden am häufigsten in Form der `If`-Anweisungen formuliert:

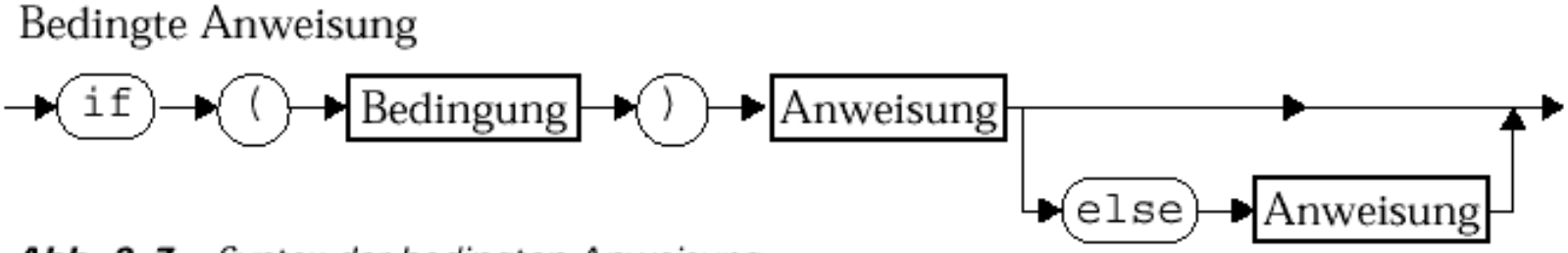


Abb. 2–7 *Syntax der bedingten Anweisung*

Lehrbuch der Programmierung mit Java, Echtle Goedicke, Heidelberg, © dpunkt 2000

(©M. Goedicke, UGH Essen)

Eine Bedingung ist durch einen booleschen Ausdruck gegeben

- Einfache oder komplexe boolesche Ausdrücke

```
a<=b      a> 3 && v || b != c && !w
```

  beachte die Typisierung der Variablen!
- Die Bedingung wird stets in runde Klammern eingeschlossen `( a<=b)` ....
- Bedeutung: Falls die Auswertung der Bedingung falsch ergibt, wird die erste Anweisung nicht ausgeführt, sondern – falls vorhanden – die Anweisung nach dem Schlüsselwort `else`

(©M. Goedicke, UGH Essen)

Zwei Beispiele ...

```
int g,k = 0;
g = Eingabe();
if (g == 1)
    k = k + 100;
else if (g == 2)
    k = k + 1000;
System.out.println("k = " + k);
```

- Frage: Was wird ausgegeben für die Eingabe
  1 ?
  2 ?
  3 ?

(©M. Goedicke, UGH Essen)

Block: zusammengehörende Anweisungsfolge

- wird durch {} als zusammengehörend gekennzeichnet
- sinnvoll z.B. bei Verzweigungen um mehr als eine Anweisung je Situation angeben zu können
- Blöcke erlauben in vielen Sprachen die Deklaration von Variablen, die nur innerhalb des Blockes zur Verfügung stehen
- Begrenzungssymbole können je nach Sprache unterschiedlich sein (begin, end), die Idee ist jedoch stets gleich.
- zur Vermeidung von Syntaxfehler empfiehlt es sich, die Begrenzungssymbole immer gleich paarweise einzugeben, und erst danach die dazwischenliegende Anweisungsfolge einzugeben.

(©M. Goedicke, UGH Essen)

Beispiel

```
float Winkel = Eingabe ();
if (Winkel > 90.0f && Winkel < 180.0f)
  { System.out.println ("stumpfer "  +
                        "Winkel");
    Winkel = 180.0f - Winkel;
  }
```

(©M. Goedicke, UGH Essen)

IF-Anweisungen können auch verschachtelt sein ...

```
if (...)
   if (...)
   else if ( ... )
```

Hier tritt das Problem auf, wie das else gebunden wird, wenn es im Prinzip zu mehreren ifs gehören könnte

→ In Java (und auch in vielen anderen Programmiersprachen) gilt:

Bindung immer an das innere `if` , es sei denn, es werden {} gesetzt

(©M. Goedicke, UGH Essen)

Beispiele zu verschachtelten Ifs ...

```
int  i = Eingabe (),  j = Eingabe ();
if (i == 5)
  if (j == 5)
    System.out.println ("i und j sind 5");
  else
    System.out.println ("nur i ist 5");
else
  if (j == 5)
    System.out.println ("nur j ist 5");
```

(©M. Goedicke, UGH Essen)

Beispiele zu verschachtelten Ifs ...

```
int  i = Eingabe (),  j = Eingabe ();
```

```
if (i == 5)
{ if (j == 5)
    System.out.println ("i und j sind 5");
}
else
  System.out.println ("i ist nicht 5");
```

```
if (j == 5)
    System.out.println ("j ist 5");
```

(©M. Goedicke, UGH Essen)

Für die Lesbarkeit lohnt es sich meist Klammern {} zu setzen!

```
float Winkel = Eingabe ();
if (Winkel <   90.0f)
  System.out.println ("spitzer Winkel");
else if (Winkel ==  90.0f)
  System.out.println ("rechter Winkel");
else if (Winkel <  180.0f)
  System.out.println ("Stumpfer Winkel");
else if (Winkel == 180.0f)
  System.out.println ("gestreckter" +"Winkel");
```

(©M. Goedicke, UGH Essen)

Zusätzliches, ähnliches Konstrukt für Fallunterscheidungen mit mehren Fällen: das `switch` - Statement

Auswahl aus einer gegebenen Menge von Alternativen mittels eines `int`-Wertes

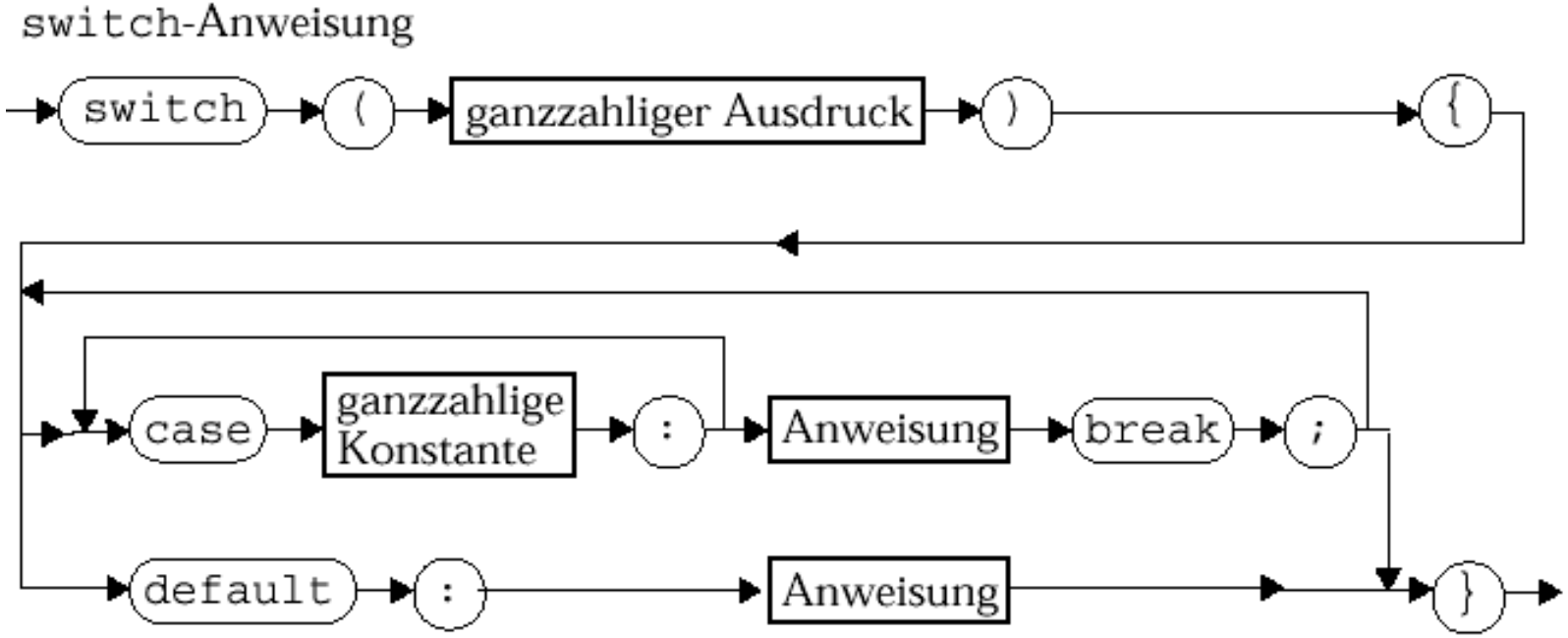


Abb. 2–8 *Syntax der switch-Anweisung*

Lehrbuch der Programmierung mit Java, Echtle Goedicke, Heidelberg, © dpunkt 2000

(©M. Goedicke, UGH Essen)

Beispiel für `switch`

```
char Ziffer;   int Wert;
Ziffer = ...
switch (Ziffer)
{ case '0': case '1': case '2': case '3': case '4':
  case '5': case '6': case '7': case '8': case '9':
    Wert = Ziffer - '0';                    break; ....
```

(©M. Goedicke, UGH Essen)

Beispiel für `switch` etwas vollständiger

```
char Ziffer;   int Wert;
Ziffer = ...
switch (Ziffer)
{ case '0': case '1': case '2': case '3': case '4':
  case '5': case '6': case '7': case '8': case '9':
    Wert = Ziffer - '0';                    break;
  case 'a': case 'b': case 'c': case 'd': case 'e':
  case 'f':
    Wert = Ziffer - 'a' + 10;               break;
  case 'A': case 'B': case 'C': case 'D': case 'E':
  case 'F':
    Wert = Ziffer - 'A' + 10;               break;
  default: System.out.println (Ziffer + " ist " +
                               "ungültig");
}
```

(©M. Goedicke, UGH Essen)

Bemerkungen zu `switch`

- Im Beispiel oben: `char` ist typkompatibel zu `int`

- Es gibt weitere Formen des `switch`-Statements, die auch ohne `break` auskommen ... aber unübersichtlich sind
- Auch hier gilt: Klammern erhöhen in der Regel die Übersicht und Lesbarkeit

(©M. Goedicke, UGH Essen)

Zwischenstand

- Variablen
  - Bezeichner, Datentyp, Speicherort, Wert
- Zuweisung
  - linke Seite: Bezeichung eines Speicherortes in den der Wert der rechten Seite geschrieben wird
  - rechte Seite: Wert, Ausdruck dessen Resultat zum Typ der linken Seite passen muss
- Alternative, Fallunterscheidung
  - if (Bedingung) then { Befehlsfolge1 } else { Befehlsfolge2 }
  - switch (Variable) { case 1 : ... break ; case 2 : ... break ; default : ... }
- es fehlen noch
  - Schleifen (while, do-while, for, ...)
  - Unterprogramme, (Prozedur, Funktion, Methode, ...)

Oft sind Berechnungen zu wiederholen

- Bisher sind die besprochenen Programme einmal durchgelaufen ... jede Anweisung wurde höchstens einmal ausgeführt
- Beispiele für Wiederholungen:
  - Mathematische Folgen und Reihen $a_{i+1} = f(i,a_i)$
  - Verarbeitung wiederkehrender Vorgänge (Buchungen...)
  - Primzahltest: n prim? -> teste ob 2,3,4, $\sqrt{n}$ Teiler von n sind.

(©M. Goedicke, UGH Essen)

Imperative Programmiersprachen (und Java) bieten für solche Aufgaben das Sprachkonstrukt der Schleife an

- Drei Arten
  - `while (Bedingung) { Anweisungsfolge }`
  - `do { Anweisungsfolge} while (Bedingung)`
  - `for (Initialisierung, Bedingung, Fortschritt) { Anweisungsf.}`
- Diese Vielfalt ist „nur“ durch Komfort begründet
- Die allgemeinste Form ist die `while`-Schleife

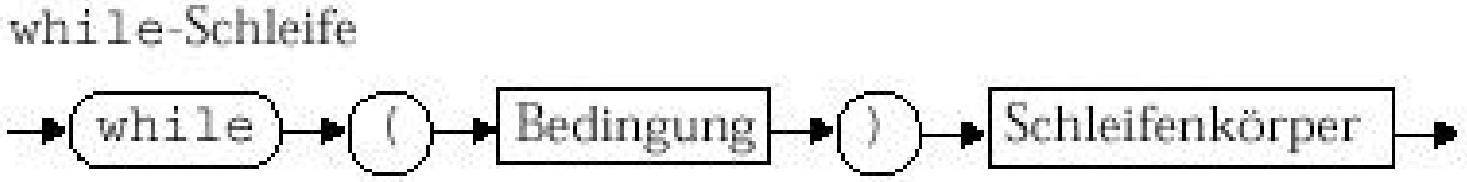


Abb. 2–9 *Syntax der while-Schleife*

Lehrbuch der Programmierung mit Java, Echtle Goedicke, Heidelberg, © dpunkt 2000

(©M. Goedicke, UGH Essen)

Grundsätzlich gilt, dass der Schleifenkörper solange wiederholt wird, wie die Bedingung angibt

- `while ( bedingung ) { ... }`

- Die Bedeutung kann auch durch ein Diagramm dargestellt werden (Kontrollflussgraph)

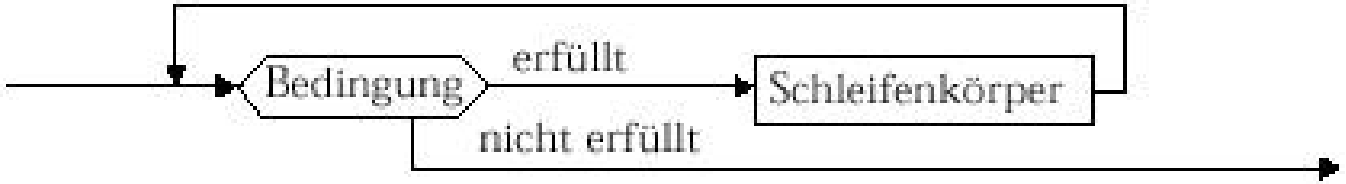


Abb. 2–10 *Semantik der while-Schleife*

Lehrbuch der Programmierung mit Java, Echtle Goedicke, Heidelberg, © dpunkt 2000

Beachte: Die while-Schleife kann auch 0-mal ausgeführt werden

(©M. Goedicke, UGH Essen)

Beispiel ... Reihenberechnung

```
int i = 1, a = 3;
while (i++ < 1000000)
        a = (4*a + 5*a*a)%13579;
```

- In 2-3 Zeilen werden 10^6 Ausführungen von Zeilen beschrieben
- Kleine Fehler haben große Auswirkungen:
  z.B.: `i` statt `i++`
- Die häufigsten Fehler in Schleifen
  - Bedingung verändert sich nicht oder ist falsch
  - Bedingung signalisiert falsches Ende
  - Falsche Initialisierung

(©M. Goedicke, UGH Essen)

Beispiel ... Primzahl-Test (1)

- Algorithmus-Idee: teste ob 2,3,4, √n Teiler von n sind (kann natürlich optimiert werden!)
- Umsetzung:
  Solange *kein Teiler gefunden* und *die Grenze nicht erreicht*: erhöhe den Teiler um eins
- Bedingung kein Teiler gefunden wird in boolescher Variablen `istPrimzahl` gespeichert

```
while (Teiler <= Wurzel && istPrimzahl)
    if (n % Teiler == 0)
       { istPrimzahl = false; }
     else { Teiler++; }
```

(©M. Goedicke, UGH Essen)

Beispiel ... Primzahl-Test (2)

```
int  n = Eingabe (),  Wurzel,  Teiler = 2;
boolean  istPrimzahl = true;

Wurzel = (int) java.lang.Math.sqrt
                        ((float) n);
```

```
while (Teiler <= Wurzel && istPrimzahl)
  if (n % Teiler == 0)
        istPrimzahl = false;
    else  Teiler++;
```

```
System.out.println (n + " prim: " + istPrimzahl);
```

(©M. Goedicke, UGH Essen)

Generell stellen Schleifen ein schwieriges Programmkonstrukt dar

- Einfaches aber effizientes Verfahren ist die Darstellung der Werteverläufe über Tabellen:

```
 n   Wurzel  Teiler  istPrimzahl
21    4      2      true
21    4      3      false
  (2 Iterationen)
```

```
 n   Wurzel  Teiler  istPrimzahl
37    6      2      true
37    6      3      true
37    6      4      true
37    6      5      true
37    6      6      true
  (5 Iterationen)
```

- Auch hier: ggfs. Klammern und Einrückung zur Erhöhung der Lesbarkeit
- Schleifen sind schwierig, aber ohne sie kommt man nicht aus

  Warum schwierig ? Warum nötig ?

(©M. Goedicke, UGH Essen)

Für Schleifen, die mindestens einmal ausgeführt werden gibt es `do-while`

- Syntax und Semantik durch Diagramme

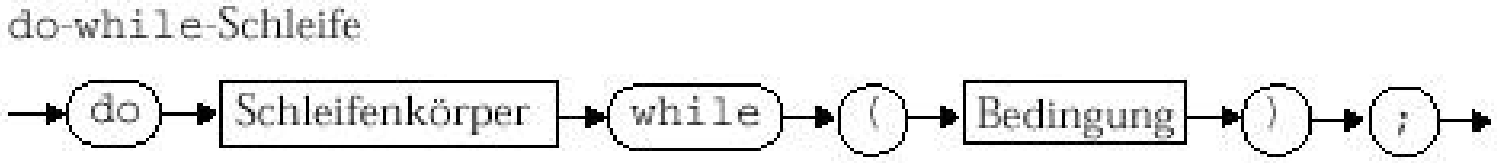


Abb. 2–12 *Syntax der do-while-Schleife*

Lehrbuch der Programmierung mit Java, Echtle Goedicke, Heidelberg, © dpunkt 2000

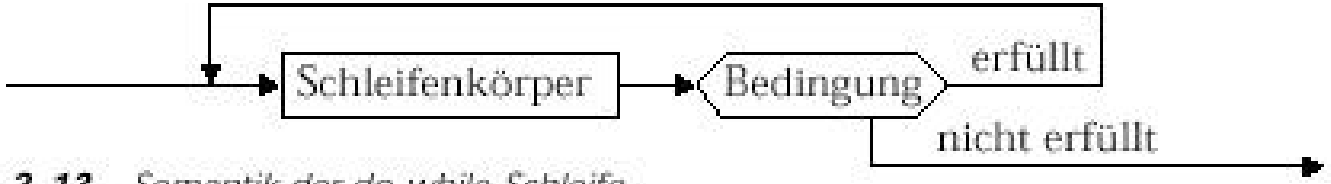


Abb. 2–13 *Semantik der do-while-Schleife*

Lehrbuch der Programmierung mit Java, Echtle Goedicke, Heidelberg, © dpunkt 2000

(©M. Goedicke, UGH Essen)

Beispiel do-while (1)

```
int Summe = 0, Anzahl = 0;
```

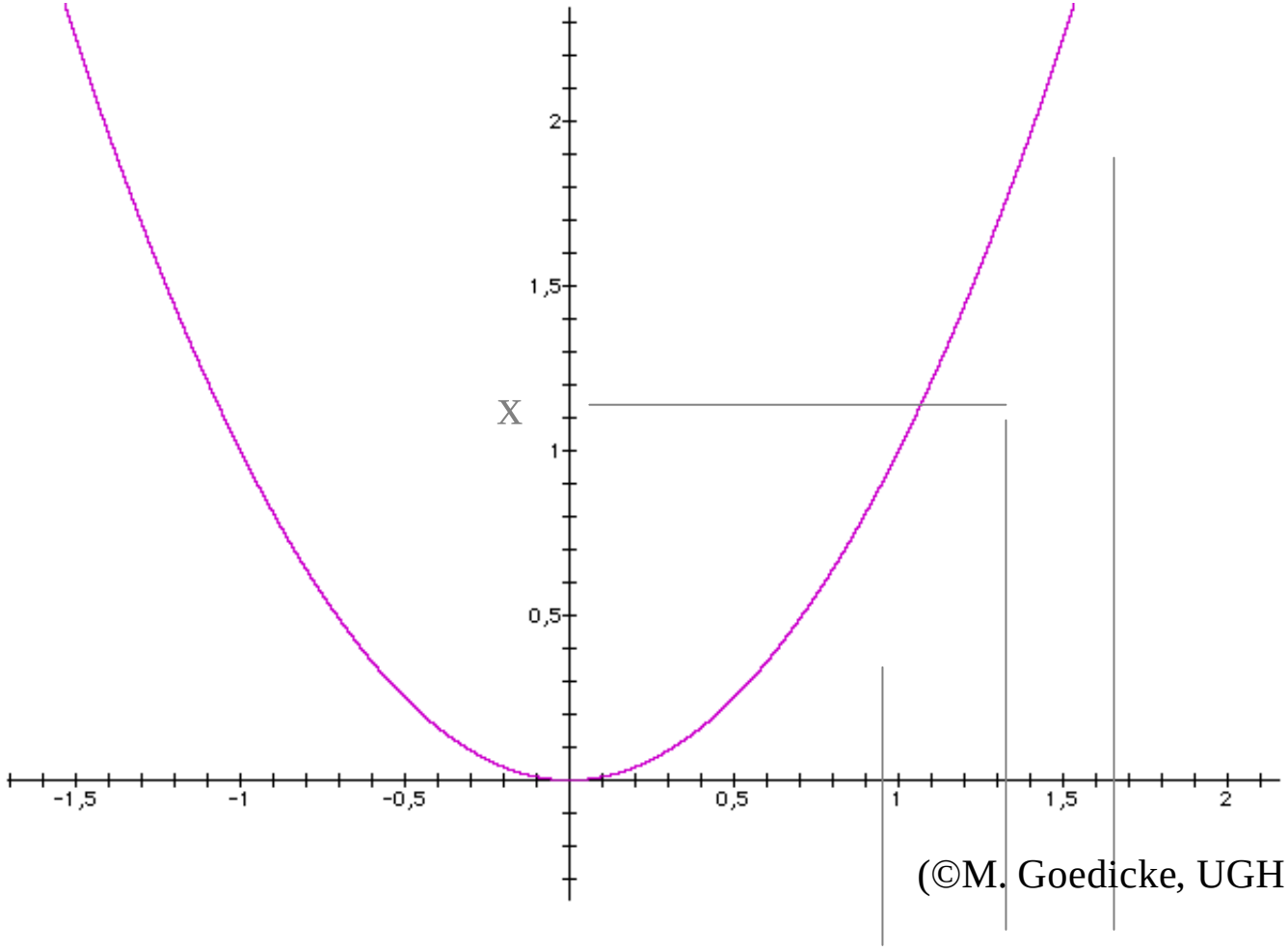
```
do { Summe = Summe + Eingabe ();
    Anzahl++;
  }
while (Summe <= 100);
```

```
System.out.println ("Sum. " + Summe +
                    ", Anz. " + Anzahl);
```

(©M. Goedicke, UGH Essen)

Beispiel do-while (2)

Quadrat-Wurzel Berechnung mittels Intervallschachtelung



(©M. Goedicke, UGH Essen)

m_2 m_i m_1

Beispiel do-while (2)

Quadrat-Wurzel Berechnung mittels Intervallschachtelung

Start: Intervall [0,x+1], Mitte m = 0,5*(uG+oG)

Algorithmus:

Berechne neue Mitte m = 0,5*(uG+oG)

Falls $m^2 > x$ oG = m

sonst uG = m

Abbruch: falls oG - uG < ε

(©M. Goedicke, UGH Essen)

Beispiel do-while (2)

```
float x = Eingabe (),
      uG = 0,  oG = x + 1,  m,
      epsilon = 0.001f;
do {  m = (uG + oG)/2;
      if (m*m > x)  oG = m;
      else          uG = m;
   }
while (oG - uG > epsilon);
System.out.println ( "Wurzel " + x
                   + " beträgt ungefähr "
                   + m);
```

(©M. Goedicke, UGH Essen)

Generell bestehen Schleifen immer aus drei Teilen

1. Vorbereitende Anweisungen
   - Deklaration von Variablen, Initialisierungen
2. Fortschaltenden Anweisungen
3. Abfrage der Schleifenbedingung

Daher sollte bei der Prüfung der Korrektheit eines (Teil-) Programms aus einer Schleife auch insbesondere diese Aspekte geprüft werden

1. werden die Variablen, die für die Schleifenbedingung gebraucht werden, deklariert und sinnvoll initialisiert ?

2. werden die Variablen, die für die Schleifenbedingung gebraucht werden, innerhalb des Schleifenkörpers verändert

3. ist eine Veränderung der Schleifenbedingung für den Abbruch der Schleife gesichert

Weiteres spezielles Schleifenkonstrukt:

-> for-Schleife

(©M. Goedicke, UGH Essen)

Die for-Schleife bietet eine direkte Syntax für diese drei Teile einer Schleife

- Vorbereitende Anweisungen
  - Variablenvereinbarungen, Initialisierungen

```
int i = Startwert
```

- Fortschaltenden Anweisungen

```
i++
```

- Abfrage der Schleifenbedingung

```
i<= Endwert
```

```
for (int i= Startwert; i<= Endwert; i++) { ... }
```

(©M. Goedicke, UGH Essen)

Die for-Schleife bietet noch einiges mehr … hier die Syntax-Diagramme ...

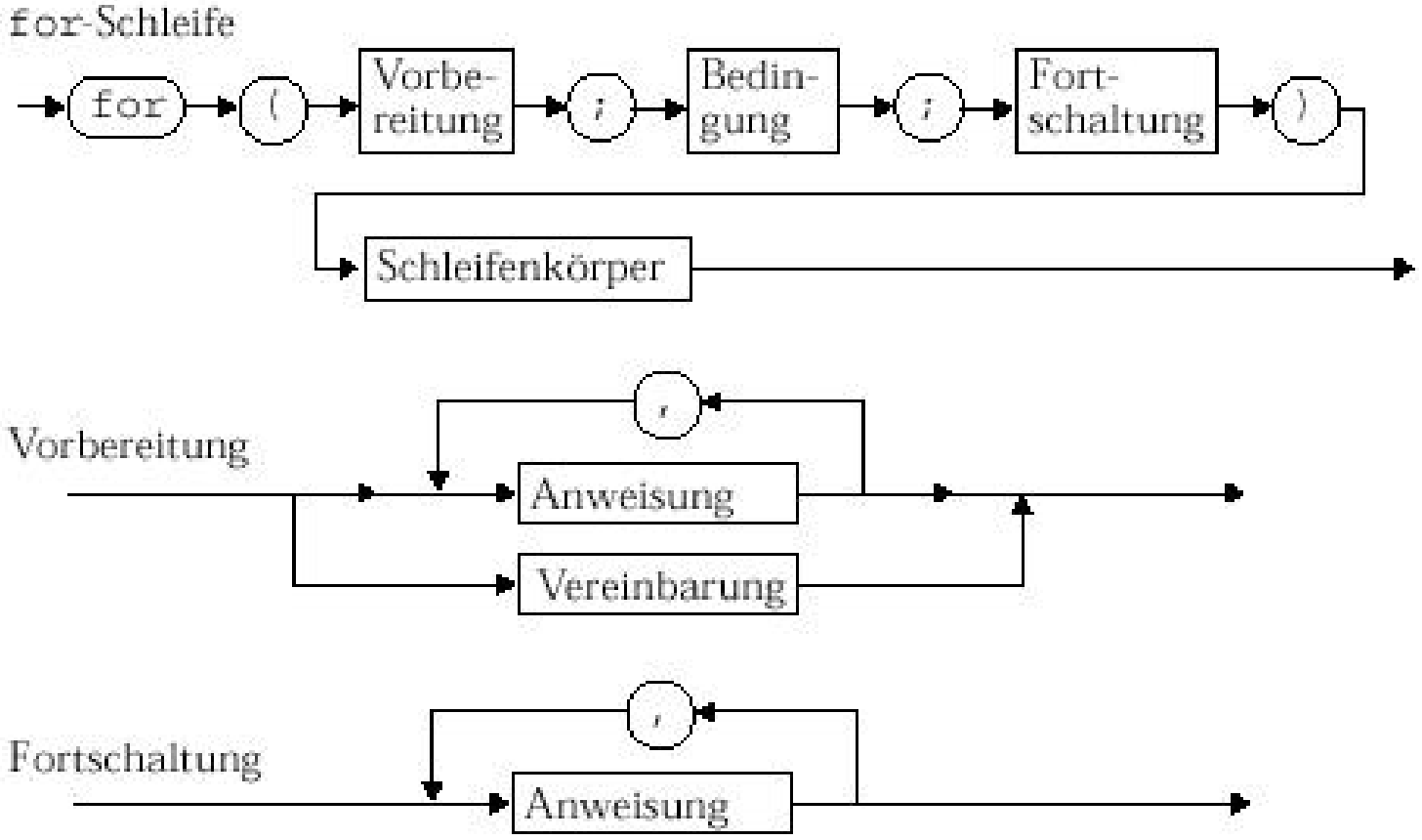


Abb. 2–14 *Syntax der for-Schleife*

Lehrbuch der Programmierung mit Java, Echtle Goedicke, Heidelberg, © dpunkt 2000

(©M. Goedicke, UGH Essen)

Damit sind auch komplexe for-Schleifen möglich

- Z.B.:

```
for (int  i=2,j=10 ; // Start
          i<=5 ;     // Ende
          i++ , j--) // Weiter
    { ... }          // Rumpf
```

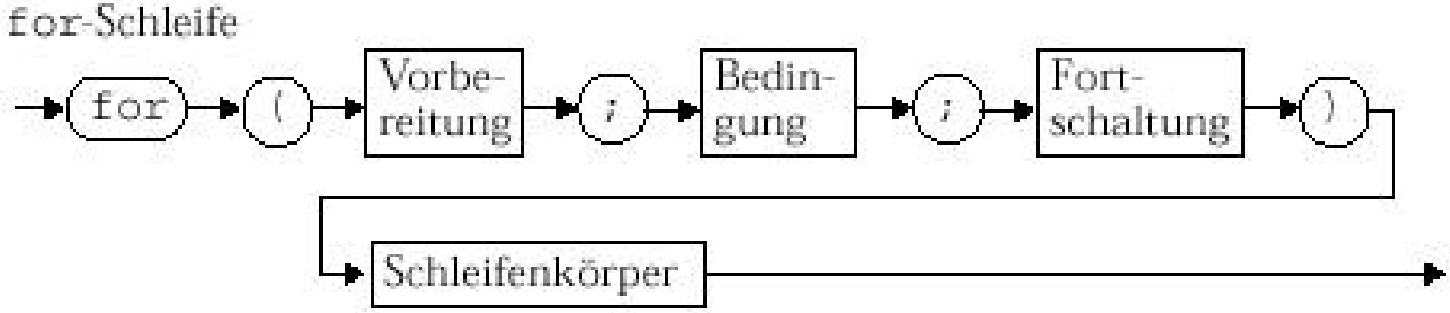
Bemerkungen:

- die schleifen-lokal vereinbarten Variablen sind nur innerhalb der Schleife gültig, dürfen aber auch außerhalb nicht noch einmal deklariert werden!
- Die Schleifenvariablen müssen nicht unbedingt lokal vereinbart sein, es erleichtert jedoch den Überblick über die Verwendung von Variablen zu behalten (Lokalität der Verwendung)
- Mehrfache Fortschaltungsanweisungen sind eher unüblich

(©M. Goedicke, UGH Essen)

Die Bedeutung der for-Schleife ...

- Syntax



- Semantik

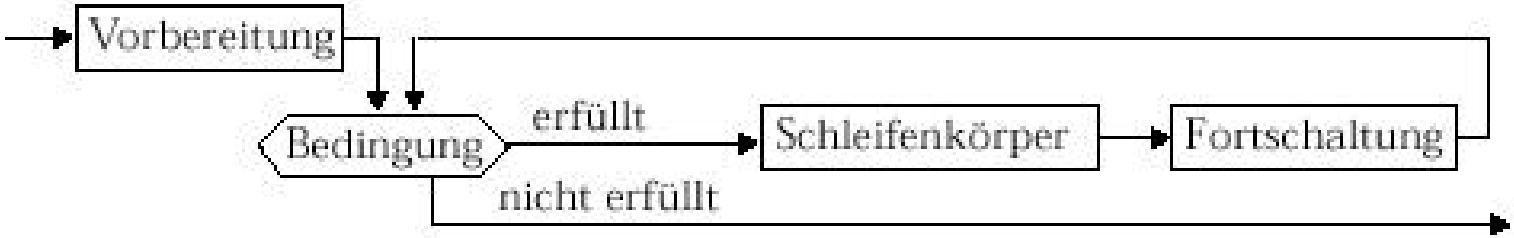


Abb. 2–15 *Semantik der for-Schleife*

Lehrbuch der Programmierung mit Java, Echtle Goedicke, Heidelberg, © dpunkt 2000

- Beachte: Auch die for-Schleife kann null-mal ausgeführt werden

(©M. Goedicke, UGH Essen)

... und ein paar Beispiele ... (1)

Das Beispiel aus Prog 2-23

Mit Hilfe von `while` (Prog 2-23):

```
int  i = 1,  a = 3;
while (i++ < 1000000)
    { a = (4*a + 5*a*a)%13579; }
```

Mit Hilfe von `for` (Prog2-27):

```
for (int i = 1, a = 3; i < 1000000; i++)
   { a = (4*a + 5*a*a)%13579; }
```

(©M. Goedicke, UGH Essen)

... und ein paar Beispiele ... (2)

Drucken einer Funktionstabelle ...

$f(x) = x^2 - 8$ für $x \in \{-5,-4,-3, \ldots , 10,11,12\}$

```
int  x ,  y;
for (x = -5; x <= 12 ; x++)
 { y = x*x -8;
   System.out.println("x= " + x +
                      " y= " + y);
 }
```

(©M. Goedicke, UGH Essen)

Schleifen dürfen verschachtelt sein

- Z.B. eine for - Schleife als Rumpf einer anderen for - Schleife:

```
for (int i = 10; i <= 30; i = i + 10)
    for (int j = 1; j <= 4; j++)
      System.out.print (i + " " + j + ",   ");
System.out.println ();
```

äußere Schleife

innere Schleife

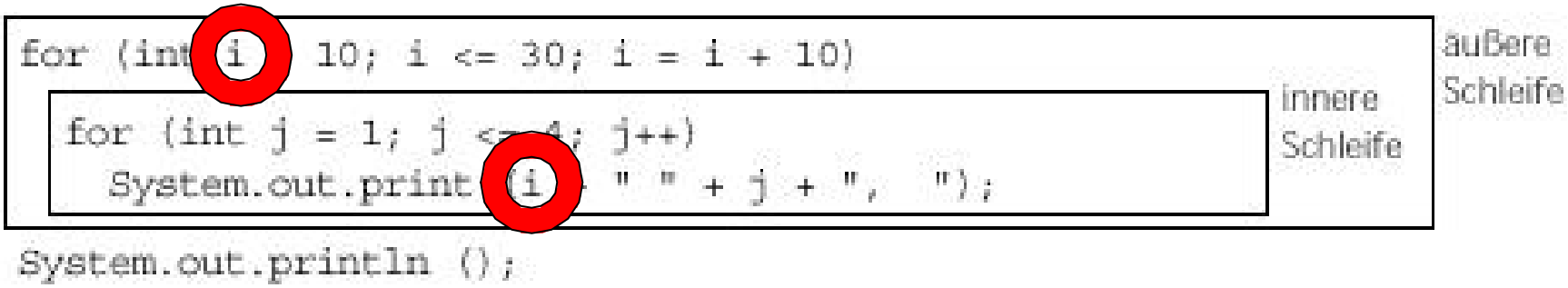
- Ein Durchlauf der äußeren Schleife ist eine vollständige Ausführung der inneren Schleife

```
10   1
     2
     3
     4        // innere Schleife einmal vollständig
20   1        // äußere Schleife in 2. Iteration
 ...
```

(©M. Goedicke, UGH Essen)

Zur Erinnerung: im Rumpf von for-Schleifen sind die Laufvariablen verfügbar

- Z.B. kann damit eine innere Schleife von dem Wert einer Laufvariablen der äußeren Schleife abhängig gemacht werden:



- Ausgabe:
  1 1,2 1, 2 2, 3 1, 3 2, 3 3,

(©M. Goedicke, UGH Essen)

Ein Beispiel für verschachtelte for-Schleifen

- Das Ziel ist einen Kalender der folgenden Art auszugeben:

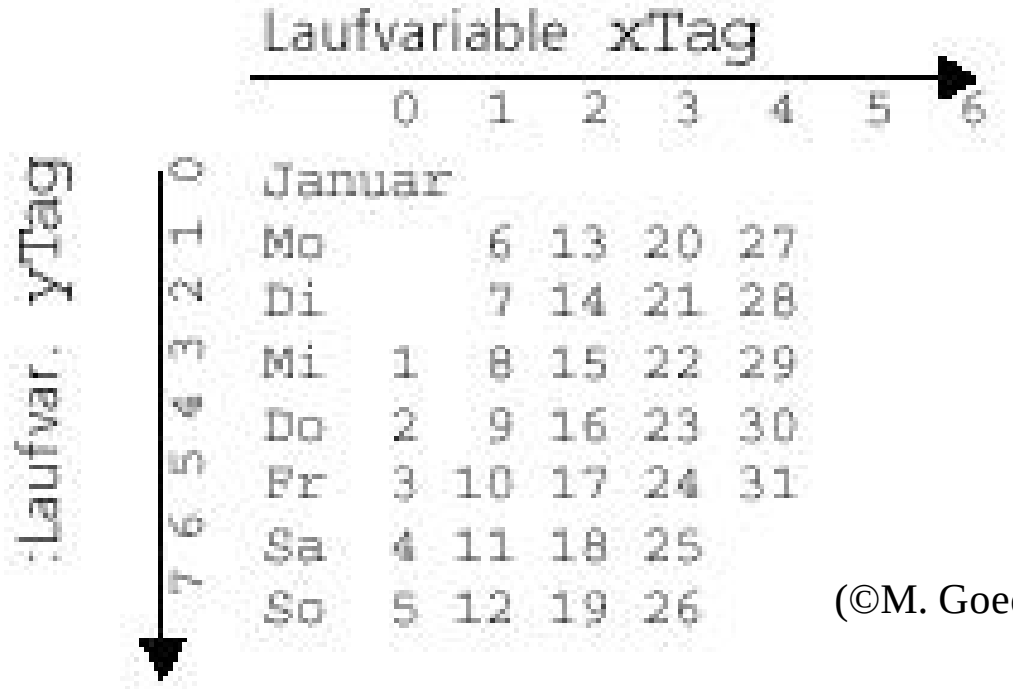
```
Kalender 2003

Januar               Februar              März
Mo    6 13 20 27     Mo    3 10 17 24     Mo    3
Di    7 14 21 28     Di    4 11 18 25     Di    4
Mi  1  8 15 22 29    Mi    5 12 19 26     Mi    3
Do  2  9 16 23 30    Do    6 13 20 27     Do    5
Fr  3 10 17 24 31    Fr    7 14 21 28     Fr    7
Sa  4 11 18 25       Sa  1  8 15 22       Sa  1  8
So  5 12 19 26       So  2  9 16 23       So  2  9
```

- Hier: Beschränkung auf einen Monat ≠ Prog 2-31

(©M. Goedicke, UGH Essen)

Die Idee beruht auf der zeilenweisen Behandlung der verschiedenen Spalten

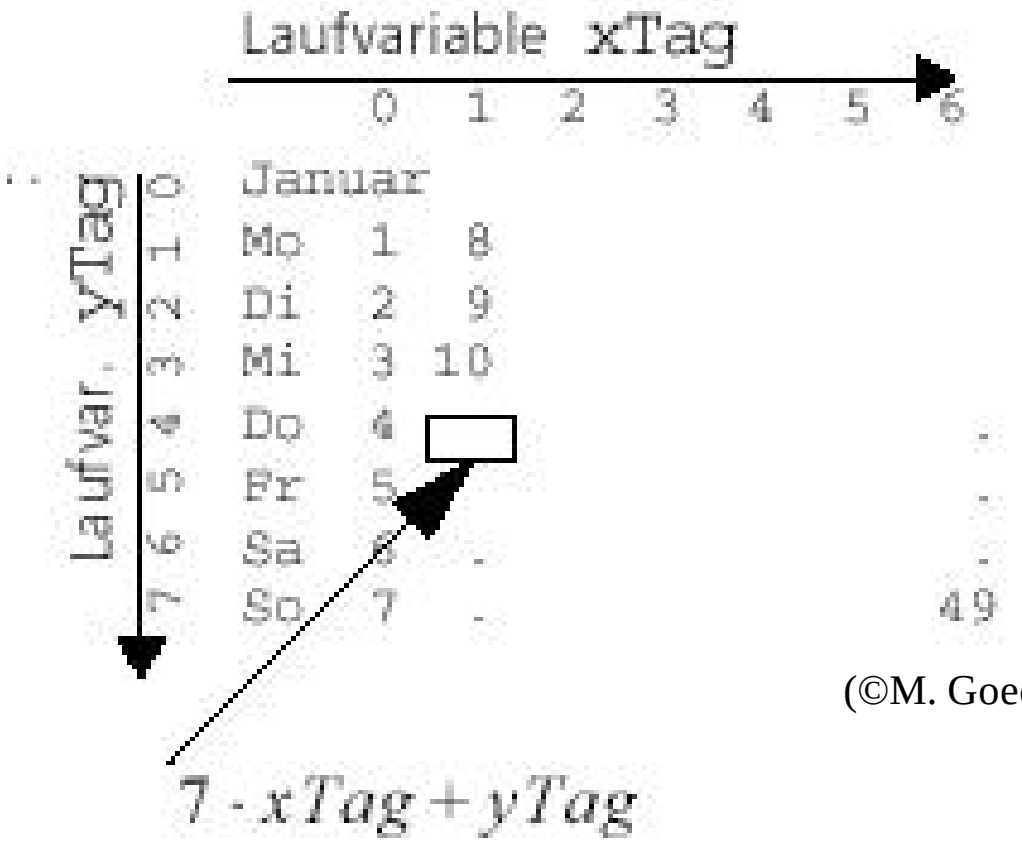
- Die Spalte im Monat -> xTag
- Die Zeile im Monat -> yTag



(©M. Goedicke, UGH Essen)

So ganz passt das noch nicht ...

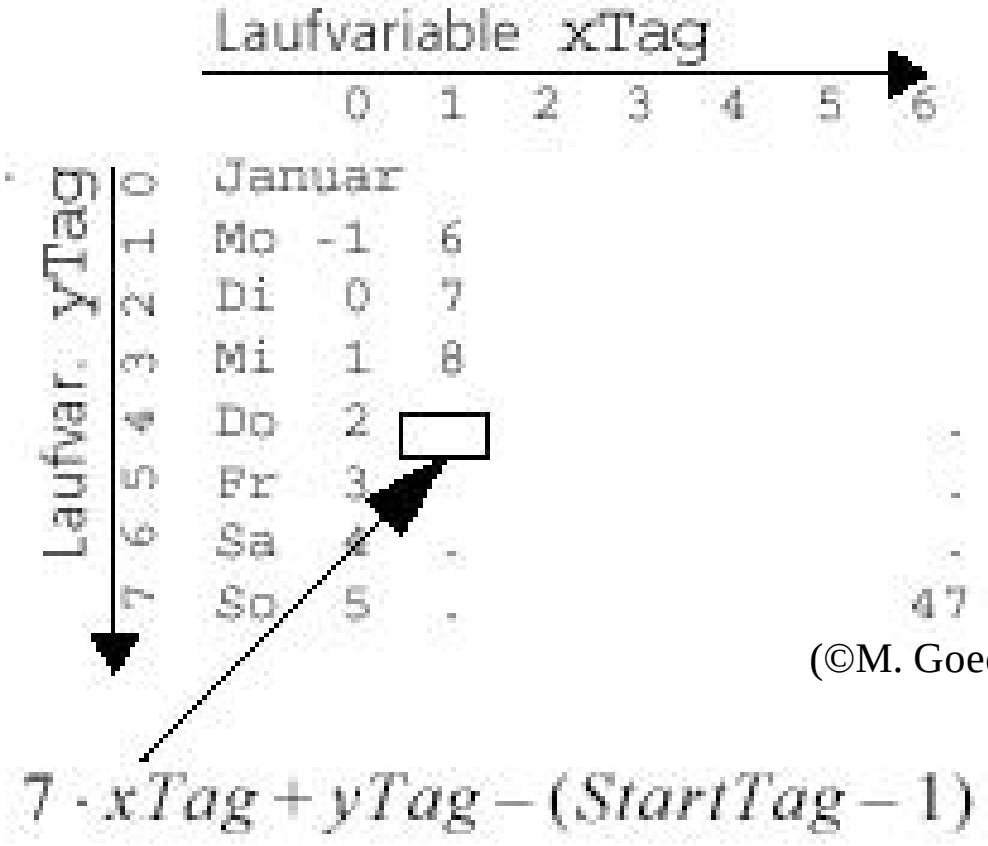
- Die Spalte im Monat -> xTag (0..6)
- Die Zeile im Monat -> yTag (0..7)



(©M. Goedicke, UGH Essen)

Eine kleine Anpassung bringt schon weiter

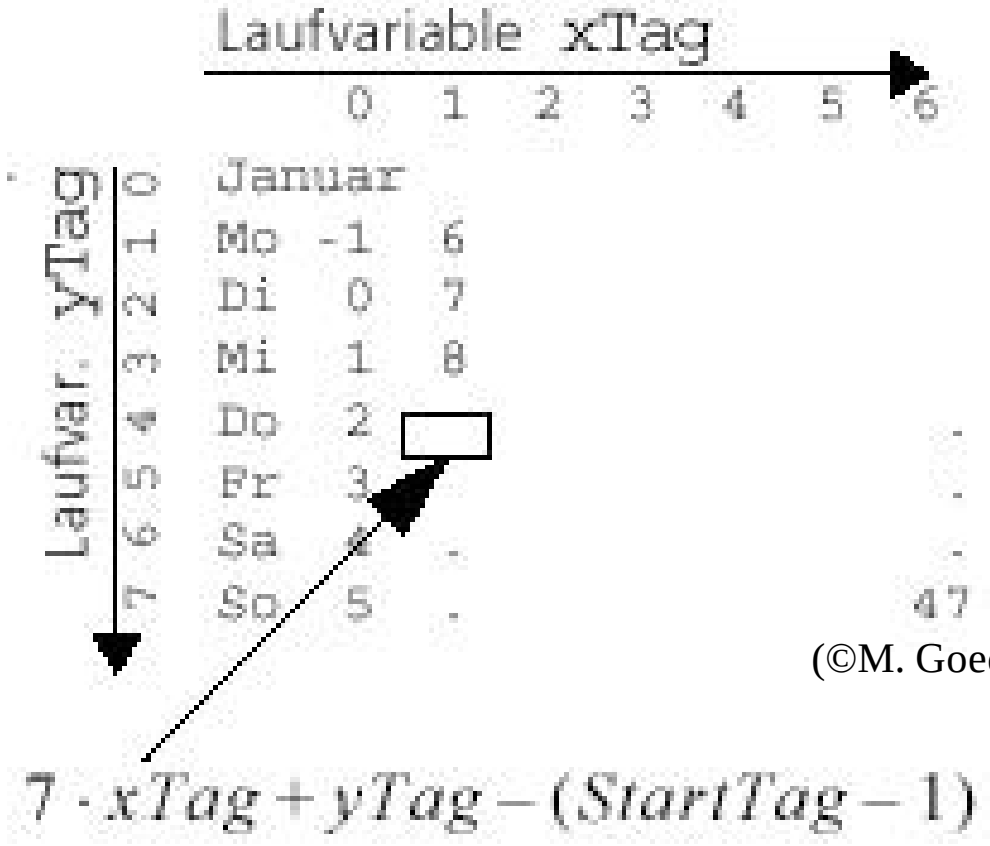
- Die Spalte im Monat -> xTag (0..6)
- Die Zeile im Monat -> yTag (0..7)



(©M. Goedicke, UGH Essen)

Auf dieser Basis kann dann die Programmkonstruktion erfolgen

- Die äußere Schleife über yTag (0..7)
- Die innere Schleife über xTag (0..6)



(©M. Goedicke, UGH Essen)

Innerhalb der geschachtelten Schleifen muss die Ausgabe der Information erfolgen

- Die äußere Schleife über yTag (0..7)
- Die innere Schleife über xTag (0..6)

```
For (yTag = 0; yTag <= 7 ; yTag++)
{
   ...
   for (xTag = 0; xTag <= 6 && 0!=yTag; xTag++)
   {  // die erste Zeile (yTag == 0) ist speziell !
      // Falls ein gültiges Datum (1..31) vorliegt
      // ausdrucken sonst Leerzeichen
   }
}
```

(©M. Goedicke, UGH Essen)

Die geschachtelten Schleifen (1)...

```
for (int yTag = 0; yTag <=7;yTag++)
{   switch (yTag)
            { case 0: System.out.print("   "+ NameM); break;
              case 1: System.out.print("Mo"); break;
              case 2: System.out.print("Di"); break;
              case 3: System.out.print("Mi"); break;
              case 4: System.out.print("Do"); break;
              case 5: System.out.print("Fr"); break;
              case 6: System.out.print("Sa"); break;
              case 7: System.out.print("So"); break;
            };
```

(©M. Goedicke, UGH Essen)

// siehe nächste Folie...

Die geschachtelten Schleifen (2)...

```
for (int xTag= 0; xTag <= 6 && 0 != yTag ; xTag++)
            {
             int Tag = 7*xTag + yTag -(StartTag-1);

             if (1<= Tag && Tag <=9)
               {System.out.print("  "+Tag);}
               else if (10 <= Tag && Tag <= TageImMonat)
                   {System.out.print(" "+Tag);}
               else {System.out.print("   ");}
            }

            System.out.println();
}
```

(©M. Goedicke, UGH Essen)

Noch zwei Punkte zu Schleifen: continue & break

Abbruchsmöglichkeiten für Schleifendurchläufe

- bisher nur bei Prüfung der Bedingung vor/nach Durchlauf des Schleifenkörpers
- zusätzliche Möglichkeiten durch spezielle Anweisungen:
  - Mit `continue` kann die Ausführung eines Schleifenrumpfs abgebrochen und mit der nächsten Iteration (nach Prüfung der Schleifenbedingung) fortgesetzt werden
  - Mit `break` kann die Ausführung einer Schleife beendet werden
- Besonderheit in Java: Anweisungen können benannt werden:

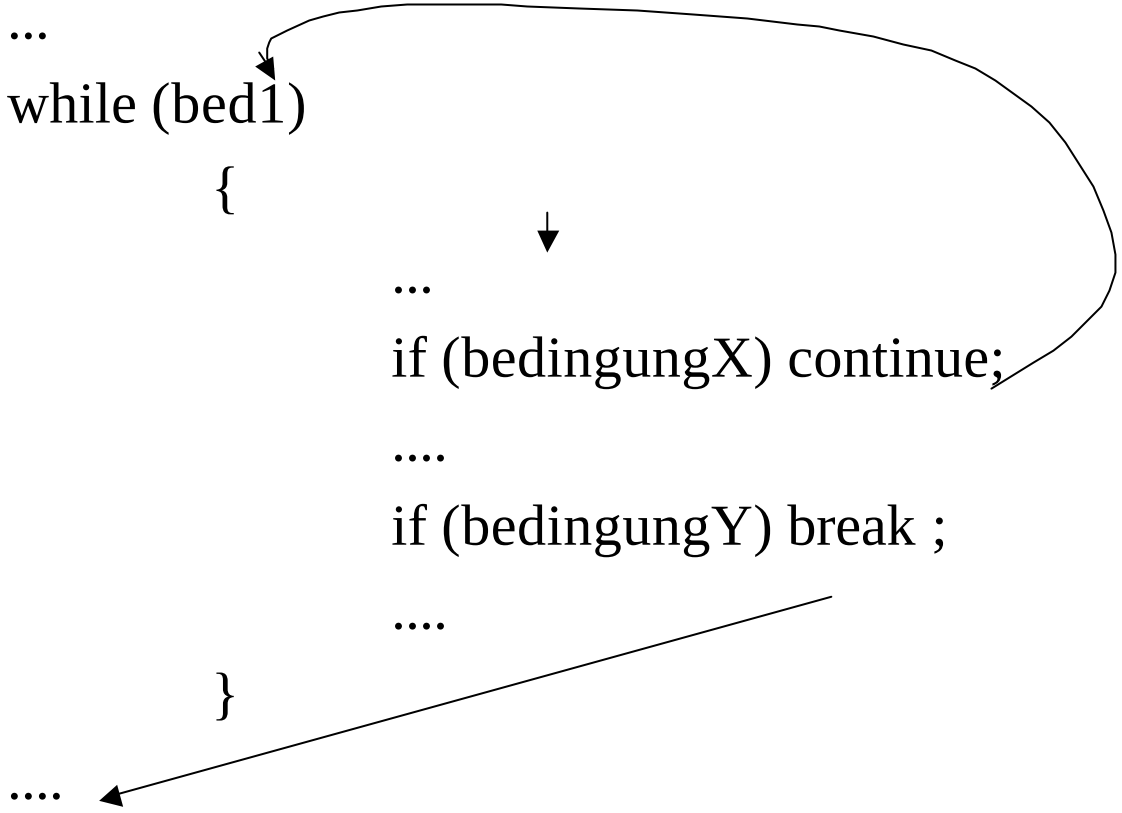
  ```
  Benennung: Anweisung;
  Startwert: istPrimzahl = true;
  ```

  - Um bei geschachtelten Schleifen eine äußere Schleife für `break` oder `continue` zu identifizieren, muss der jeweilige Schleifenkopf mit einem Namen versehen und in der break bzw. continue-Anweisung angegeben werden

(©M. Goedicke, UGH Essen)

Beispiel zu continue & break

einfache, normale Anwendung



(©M. Goedicke, UGH Essen)

Beispiel zu continue & break

Anmerkung: unüblich schwieriger Fall !!!

```
...
Aussen: while (bed1)
          {
            Innen: while (bed2)
            {
              if (bed3) continue Aussen;
              if (bed4) continue Innen;

              if (bed5) break Aussen;
            }
          }
....
```

(©M. Goedicke, UGH Essen)

Vorsicht mit `continue` & `break`

- Nur in übersichtlichen Fällen und sparsam verwenden!
- Programme werden durch die Verwendung dieser Sprachkonstrukte schnell unübersichtlich

(©M. Goedicke, UGH Essen)

Zwischenstand

- Variablen
  - Bezeichner, Datentyp, Speicherort, Wert
- Zuweisung
  - linke Seite: Bezeichung eines Speicherortes in den der Wert der rechten Seite geschrieben wird
  - rechte Seite: Wert, Ausdruck dessen Resultat zum Typ der linken Seite passen muss
- Alternative, Fallunterscheidung
  - if (Bedingung) then { Befehlsfolge1 } else { Befehlsfolge2 }
  - switch (Variable) { case 1 : ... break ; case 2 : ... break ; default : ... }
- Schleifen
  - while (Bedingung) { Befehlsfolge }
  - Varianten: do-while, for,
  - Steuerungsmöglichkeiten: continue, break
- Es fehlen noch
  - Unterprogramme, (Prozedur, Funktion, Methode, ...)
  - Hilfsmittel zur Konstruktion komplexer Datentypen
  - ...