Grundlagen der Informatik für Ingenieure I

Java Sprachkonstrukte

4.1 Java-Zeichensatz
4.2 Quellcode-Layout
4.3 Konstanten und Variable
4.4 Primitive Datentypen
4.5 Zeichenketten
4.6 Ausdrücke und Zuweisungen
4.7 Programmfluß-Steuerung

4.7.1 If-(Then)-Statement
4.7.2 If-(Then)-Else-Statement
4.7.3 If-(Then)-Else-If-Statement
4.7.4 Vergleichs-Operator
4.7.5 Switch-Statement
4.7.6 Schleifen-Konstrukte, Felder
4.7.6.1 Felder (arrays)
4.7.6.2 For-Statement
4.7.6.3 While-Statement
4.7.6.4 Break, Continue-Statement
4.7.7 Labeled Loops

4.8 Standard-E/A
4.9 Quellcode-Layout
4.10 Zusammenfassung

4.1 Java-Zeichensatz

- Zeichensatz
  - 52 Klein-/Großbuchstaben des englischen Alphabets,
  - $,
  - Unterstreichungsstrich,
  - 10 Ziffern (0-9),
  - Zeichen mit besonderer Bedeutung (Sonderzeichen, *special character*):

<b>, <tab> blank/tab	: colon	= equals sign	? questionmark
+ plus sign	- minus sign	* asterisk	/ slash
({[left parenthesis	)}] right parenthesis	, comma	. decimal point
’ apostrophe	! exclamation mark	" quotation mark	% percent
& ampersand	; semicolon	< less than	> greater than

4.2 Quellcode-Layout

- Quellcode-Layout
  - spaltenunabhängiges, formatfreies Quellprogramm
  - die Blockstruktur des Programmcodes sollte durch Einrücken hervorgehoben werden
  - Trennzeichen sind <blanks> oder <tabs>, ein Semicolon schließt eine **Anweisung** *(Statement)* ab, **Blöcke** werden durch geschweifte Klammern eingeschlossen.
  - Reservierte Worte (**Schlüsselworte** (*keywords*)) sind geschützt.
- Kommentare

  Es ist ausgesprochen hilfreich - insbesondere zum späteren Verständnis - die Funktionalität einer **Klasse** und den Effekt (die Auswirkung) einer **Methode** knapp und präzise im “Kopf” der Klasse bzw. Methode zu beschreiben.

```
/* Beliebige Texte ueber mehrere Zeilen */
// der Rest der Zeile ist ein Kommentar
/** Dieser Code wird von “javadoc” für Dokumentationszwecke
    herausgezogen. */
```


4.3 Konstante und Variable

- Konstante
  - Konstante sind Größen, die zur Laufzeit unveränderbar sind; deshalb stehen sie nie auf der linken Seite einer Zuweisung.
  - Eine Konstante hat einen Typ und einen Wert.
    - **literale Konstante** *(literals):*
      auch konstruierte Konstante; der Typ geht implizit aus der Schreibweise hervor. (Siehe primitive Datentypen Kap. 4.4)
    - **Namenskonstante:**
      sind Konstante die über einen Namen referiert werden.

`Syntax:final type CONSTNAME = value;`

Beispiele:

```
final float PI = 3.141592;
final int MAXSIZE = 40000;
final PersonalDB DIAPERS = new PersonalDB();
```

Regel: Für Namen von Namenskonstanten werden ausschließlich Großbuchstaben verwendet.

4.3 Konstante und Variable

Variable

- Einer Variablen ist ein Speicherplatz zugeordnet. Der Name der Variablen ist die symbolische Bezeichnung einer Adresse im **Arbeitsspeicher** bzw. im **virtuellen Adreßraum**.
  - Jede Variable hat
    - einen **Typ** (und damit einen Wertebereich),
    - einen **Namen**,
    - einen **Wert**,
    - einen **Gültigkeitsbereich** (von wo aus kann auf die Variable zugegriffen werden)
    - und eine **Lebensdauer**.
- Vor der Benutzung einer Variablen muß sie **vereinbart** (*declared*) werden.
- Mit **Anweisungen** kann man Variablen Werte zuweisen.
- Verwendet man Variable als Operanden, müssen den Variablen vorher Werte zugewiesen worden sein. Dies gilt auch für potentiell nicht erreichbaren Code.

4.3 Konstante und Variable

- Java unterscheidet drei Arten von Variablen
  - **Instanz-Variable**
    beschreiben den Zustandsraum eines Objekts; sie enthalten die Attribute eines Objekts ihr Geltungsbereich ist “objektglobal”.
  - **Klassen-Variable**
    haben Ähnlichkeit mit Instanzvariablen, jedoch sind diese Variablen allen Instanzen dieser Klasse gemeinsam.
  - **Lokale Variable**
    werden innerhalb einer Methode oder eines Blocks vereinbart; ihr Geltungsbereich ergibt sich aus dem Ort des **Deklarationsstatements.**
  - **Es gibt keine applikations- oder appletglobalen Variablen!**

4.3 Konstante und Variable

- In Java können Variablen von folgendem Typ sein:
  - einer der acht **primitiven Typen**
  - **Instanz(Objekt)** oder **Interface**
  - String, Feld ***(array)*** - in Java, Objekte “besondere Art”

  Variablennamen dürfen außer aus Sonderzeichen ***(special characters)*** aus allen Zeichen des Zeichensatzes bestehen, sie dürfen jedoch nicht mit einer Ziffer beginnen. Außerdem dürfen hierfür keine ***keywords*** (Schlüsselwörter) verwendet werden.

Regel: **Variablennamen beginnen immer mit einem Kleinbuchstaben; zur Strukturierung längerer Namen werden Großbuchstaben verwendet.**

Beispiele:

gültige Namen	**ungültige Namen**
vorName	1_vorName
ganze_Zahl	ganze Zahl *underscore* ist "normales" Zeichen)
index1	<variable>

4.3 Konstante und Variable

- Vereinbarung (Deklaration) von Variablen

```
Syntax:[modifier] typ varname [= value][, varname
[=value]] [...];
[ ]: optional        [...]: beliebig oft
```

Beispiele:

```
private int alter; (modifier siehe Kap. 5.3)
int a = 10, b = 12, c = 1;
String familienName;
boolean flag = true;
String uwe = “uwe”;
float currentTemperatureInNewYork;
```

- Variablendeklarationen können
  - grundsätzlich an beliebiger Stelle (vor ihrer Benutzung) stehen
  - in Verbindung einer Zuweisung auftreten

Regel: **In einem wohlstrukturierten Code stehen insbesondere die Deklarationen der primitiven Typen zusammengefaßt vor dem jeweiligen Codeteil. Ausnahmen: Schleifenindizes**

4.4 Primitive Datentypen

◆ Ein primitiver Datentyp ist durch das Tupel (Wertemenge; Operationen) definiert.
Sie sind - letztendlich aus Effizienzgründen - keine Objekte, sondern “integraler” Bestandteil der Sprache.

- numerische Datentypen
  - ganzzahlig: Wertebereich: -2^n bis $2^n -1$
    - byte: 8 bits; -128 bis 127
    - short: 16 bits; -32.768 bis 32.767
    - int: 32 bits; -2.147.483.648 bis 2.147.483.647
    - long: 64 bits; -9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807

4.4 Primitive Datentypen

- numerische Datentypen(cont)
  - reell (IEEE 754):
    - float

      $s * m * 2^e$
      mit $s = +1$ oder -1
      $0 < m < 2^{24}$,
      $e = [-149, +104]$
    - double (“doppelte” Genauigkeit)

      $s * m * 2^e$
      mit $s = +1$ oder -1
      $0 < m < 2^{52}$
      $e = [-1075, +970]$

◆ nichtnumerische Datentypen

- logisch: boolean
- Zeichen: char : 16 bit Unicode (im Gegensatz zu “C”!)

4.4.1 Ganzzahliger Datentyp

- byte, short, int, long
  - Operationen: `+ - * / % (modulo)`
  - Schreibweise der literalen Konstanten:

```
decimal:                    1
                            0
                            -199
                            32612
                            6L oder 6l Konstante v. Typ long

ist die Zahl groß genug, wird automatisch “long” gewählt

octal (base 8):             07631

hexadecimal (base 16):      0x10BC
```


4.4.2 Reeller und doppeltgenauer Datentyp

- float, double
  - Operationen: `+ - * / %`
  - IEEE-Format
  - Schreibweise der literalen Konstanten (*default* vom Typ double):
    Zahlendarstellung:
    Koeffizient: Dezimalzahl der Form d., d.d oder .d; d Ziffernfolge
    Exponent: ganzahliger Exponent Basis 10

```
7.5     35.E1   (=350.)
-5.34   .0016E4 (= 16.)
+1832.  2443.E2 (= 244300)
.3      50.E-2  (= .5)
4.23456f (oder F): literale Kontante ist vom Typ float
```

4.4.3 Zeichendatentyp

- char
  - Operationen: keine
  - Wertebereich: Zeichen aus der Menge der darstellbaren Zeichen; Unicode-Tabellen
  - werden als 16-bit *Unicode-Characters* gespeichert.
  - Schreibweise der literalen Konstanten:

```
'S'; 's'; ; '%'; '\n'
```


4.4.3 Zeichendatentyp

- Die Ersatzdarstellung für nicht druckbare Zeichen zeigt die folgende Tabelle:

Escape	Meaning
\n	Newline
\t	Tab
\b	Backspace
\r	Carriage return
\f	Formfeed
\\	Backslash
\'	Single quotation mark
\"	Double quotation mark
\ddd	Octal
\xdd	Hexadecimal
\udddd	Unicode character

4.4.4 Logischer Datentyp

- boolean
  - Operationen:
    - UND: &&
    - ODER: ||
    - exclusive ODER: ^
    - NOT: !
  - Wertebereich: true, false

    Schreibweise der literalen Konstanten:

    ```
    true, false
    ```

    Nicht zu verwechseln mit den "bitweisen" Operatoren auf ganzzahlige Typen (|; &; <<; >>; ...)!!!

4.5 Zeichenketten (Strings)

- Zeichenketten in Java sind Instanzen der *class String*
  - Da Zeichenketten also Objekte sind, handelt es nicht einfach um Felder mit Elemente des Typs *char,* sondern es sind auch Methoden (Operationen) definiert mit denen man Zeichenketten z. B.
    - verknüpfen,
    - testen und
    - vergleichen kann.
  - *String-Literale*

    String ist die einzige Klasse, die es erlaubt, auf diese weise Objekte zu instantiieren.

    ```
    “Hallo, ich bin eine Zeichenkette\n”
    “A string with a \t tab in it”
    “Mercedes\u2122 ist ein gesch\00FCtztes Markenzeichen”
    ```

    Die vollständige (Latin-)Unicode-Tabelle können Sie einsehen in:

    http://unicode.org (**Vorsicht!** Das ist nicht wenig!)

4.6 Ausdrücke und Zuweisung

Ausdrücke*(Expressions)*

- Ein Ausdruck ist im allgemeinen eine Formel zur Berechnung (oder Bildung) eines Wertes. Er besteht aus Operanden, Operatoren und/oder runden Klammern.
  - Beispiele:

```
-a + b/z; (a / b) + (a*b); -a+b+c; (Arithmetischer Ausdruck)
17 + 4 <= cardDeck                  (Vergleichsausdruck)
x && y                              (logischer Ausdruck)
```

  - Ausdrücke werden bei gleicher Wertigkeit der Operatoren *(precedence)* von links nach rechts ausgewertet. Durch Setzten von Klammern kann man die Reihenfolge steuern.
    Zur Wertigkeit der Operatoren: siehe *"Precedence Table"*.
    - Beispiel: **`-a+b+c`** wird wie folgt ausgewertet: **`((-a)+b)+c`**
    - Das sinnvolle Setzen von Klammern bei komplexeren Ausdrücken erhöht die Lesbarkeit eines Programms erheblich und drückt die Intension des Programmierers aus.

4.6 Ausdrücke und Zuweisung

- *Precedence Table*

Operator	Notes
.[] ()	Parentheses (()) are used to group expressions; dot(.) is used for access to methods and variables within objects and classes (discussed later); square brackets ([]) are used for arrays.
++ -- ! ~ instanceof	The instanceof operator returns true or false based on whether the object is an instance of the named class or any of that class's subclasses.
new (type) expression	The **new** operator is used for creating new instances of classes; () in this case is for casting a value to another type.
* / %	Multiplication, division, modulus
+ -	Addition, substraction
<< >> >>>	Bitwise left and right shift

4.6 Ausdrücke und Zuweisung

◆ *Precedence Table (cont.)*

Operator	Notes
< > <= >=	Relational comparison tests
== !=	Equality
&	AND
^	XOR
|	OR
&&	Logical AND
||	Logical OR
? :	Shorthand for if...then...else
= += -= *= /= %= ^=	Various assignments

4.6 Ausdrücke und Zuweisung

■ Zuweisung (Assignment)

```
Syntax:variable = expr
(Lies: "=" als "ergibt sich aus")
```

Beispiel: a = 25.5 * 3.1E9

Die rechte Seite eines Ausdrucks wird zunächst ausgewertet, bevor das Ergebnis der auf der linken Seite stehenden Variablen zugewiesen wird. Deshalb sind Anweisungen der Form

```
x = x + y
```

unproblematisch. Hier wird deutlich, dass eine Zuweisung etwas anderes ist, als eine math. Gleichung!

Für diesen Typ der Zuweisungen wurde ein Satz spezieller Zuweisungs-operatoren kreiert um Schreibarbeit zu sparen:

```
x += y entspricht x = x + y
x -= y entspricht x = x - y
x *= y entspricht x = x * y
x /= y entspricht x = x / y
```

4.6 Ausdrücke und Zuweisung

- Incrementing; Decrementing

```
x++; entspricht x = x + 1;

y--; entspricht y = y - 1;
```

- Prefix-/Postfix-Schreibweise

```
y = x++;   erst wird y der Wert x zugewiesen, danach wird
           x inkrementiert
y = ++x;   erst wird wird x inkrementiert, danach wird
           y der neue Wert von x zugewiesen.
```

- Arithmetik mit Daten vom Typ *ganzzahlig*

  Bei der Integerdivision wird der Bruchanteil immer abgeschnitten
  Beispiele:

```
y = 9 / 3 ist 3          y = 2 * 2 * 2 ist 8
y = 11 / 3 ist 3         y = 1 / (2 * 2 * 2) ist 0!
y = -11 / 3 ist -3
```


4.6 Ausdrücke und Zuweisung

- **Vergleichsoperatoren(Relationen)**
  - Das Ergebnis einer Vergleichsoperation ist immer vom Typ *boolean*; also *true* oder *false.*

Operator	Bedeutung	Beispiel
==	gleich	x == 3;
!=	ungleich	x != 3;
<	kleiner als	x < 3;
>	größer als	x > 3;
<=	kleiner gleich	x <= 3;
>=	größer gleich	x >= 3;

4.7 Programmsteuerungsanweisungen

(Control Statements)

- Zunächst einmal werden die Anweisungen eines Programms Zeile für Zeile oder besser *Statement* für *Statement* ausgeführt.
  Mit Programmsteueranweisung kann man den Kontrollfluß
  - ändern,
  - unterbrechen oder
  - beenden.

- ***Statement***
  Ein *Statement* ist eine einzelne Anweisung. Das Trennzeichen für *Statements* ist das Semicolon.

- ***Block Statement***
  Ein *Blockstatement* ist die Zusammenfassung einzelner *Statements*, geklammert durch geschweifte Klammern. Innerhalb von Blöcken vereinbarte lokale Variable sind nur dort gültig (Gültigkeitsbereich). Ein Block kann im Quellcode überall dort stehen, wo auch ein einzelnes *Statement* stehen kann.

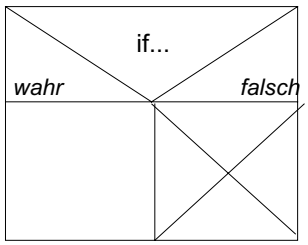
4.7.1 *If - (Then) - Statement*

- ***If - (Then) - Statement***

`Syntax:if(logical-expr) Statement`

Beipiel:

```
if (flag == true)
    index = 0;
if (x-y <= schranke)
    grenzwert = x-y;
```



Überall, wo ein *Statement* stehen kann, kann auch ein Block stehen:

Beispiel:

```
if (x < y) {
    temp = x;
    x = y;
    y = temp;
}
```

(will man die Werte zweier Variablen vertauschen, braucht man eine Hilfsvariable.)

4.7.2 If - (Then) - Else - Statement

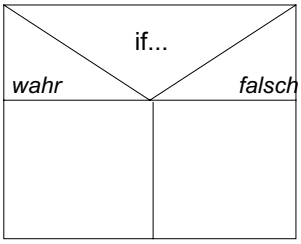
◆ ***If - (Then) - Else - Statement***

```
Syntax:if (logical-expr)
            StatementNoShortIf
       else
            Statement
```

Beispiel:

```
if (x < y)
    x = -x;
else
    y = -y;
```

Symbol:

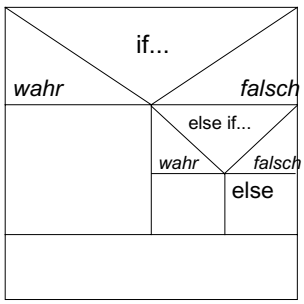


4.7.3 If - (Then) - Else - If - Statement

◆ ***If - (Then) - Else -If - Statement***

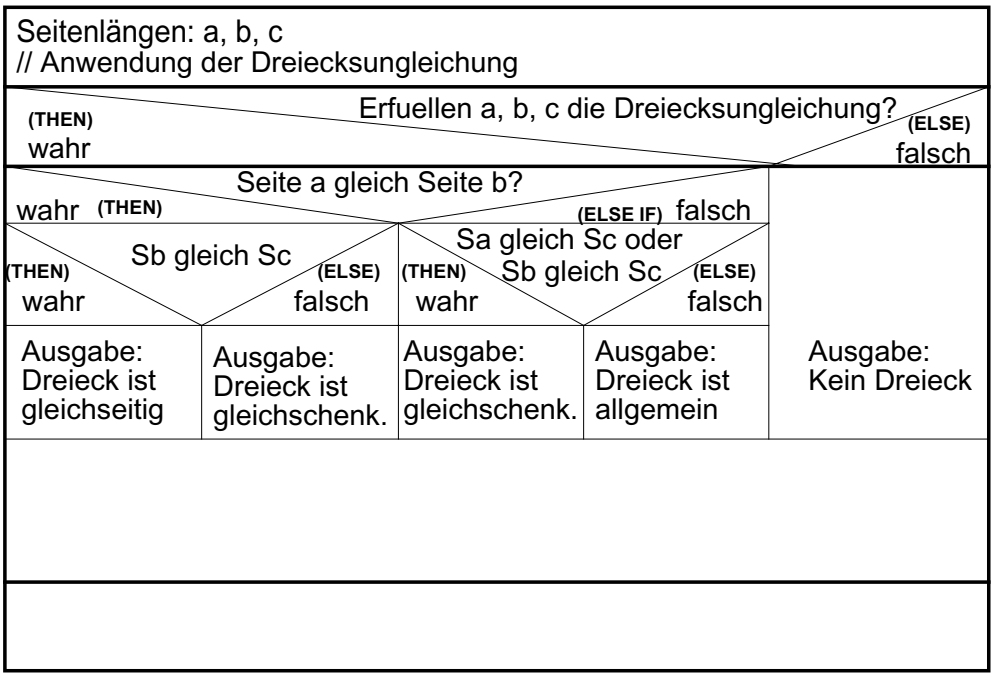
```
Syntax:if (logical-expr)
            StatementNoShortIf
       else if
            StatementNoShortIf
       else
            StatementNoShortIf
```

Symbol:



Für eine Klasse derartige Konstrukte ist u. U. das *Switch*-Konstrukt (Sprungverteiler) geeigneter.

4.7.4 Beispiel: If - (Then) - Else -If - Statement



4.7.4 Beispiel: If - (Then) - Else -If - Statement

```
class Dreieck {
   private int a;
   private int b;
   private int c;

   /* Constructor initialisiert die Instanzvariablen */

   public Dreieck (int sa, int sb, int sc) {

      a = sa; b = sb; c = sc;
   }

   /* Dreieckstypbestimmung mit Dreiecksungleichung */

   public void dreiecksTyp () {

      int h;

      System.out.println("Das Dreieck a = " + a + " b = " + b + " c = "  + c);

      h = a - b;

      h = Math.abs(h);
```

4.7.4.0 Beispiel: If - (Then) - Else -If - Statement

```
    if (h <= c && c <= a + b){
      if ( a == b) {
        if (b == c)
          System.out.println(" ist gleichseitig");

        else
          System.out.println(" ist gleichschenklig");
      }

      else {
        if (( a == c ) || ( b == c))
          System.out.println(" ist gleichschenklig");
        else
          System.out.println(" ist ein allgemeines Dreieck");
      }
    }
    else
      System.out.println(" ist kein Dreieck");

  }

}
```


4.7.4 Beispiel: If - (Then) - Else -If - Statement

```
/* Testklasse der Klasse Dreieck */
/* Version 1: Ohne Eingabe */

class DreieckTest {
  public static void main (String args[]) {

    Dreieck triangel;
    int seiteA, seiteB, seiteC;

    seiteA = 11;
    seiteB = 12;
    seiteC = 6;

    for ( seiteB = 12; seiteB >= 4; seiteB-- ){

      triangel = new Dreieck(seiteA, seiteB, seiteC);

      triangel.dreiecksTyp();
    }
  }
}
```

4.7.4 Beispiel:If - (Then) - Else -If - Statement

Testausgabe:

```
Das Dreieck a = 11 b = 12 c = 6
 ist ein allgemeines Dreieck
Das Dreieck a = 11 b = 11 c = 6
 ist gleichschenklig
Das Dreieck a = 11 b = 10 c = 6
 ist ein allgemeines Dreieck
Das Dreieck a = 11 b = 9 c = 6
 ist ein allgemeines Dreieck
Das Dreieck a = 11 b = 8 c = 6
 ist ein allgemeines Dreieck
Das Dreieck a = 11 b = 7 c = 6
 ist ein allgemeines Dreieck
Das Dreieck a = 11 b = 6 c = 6
 ist gleichschenklig
Das Dreieck a = 11 b = 5 c = 6
 ist ein allgemeines Dreieck
Das Dreieck a = 11 b = 4 c = 6
 ist kein Dreieck
```


4.8 Vergleichsoperator

- Vergleichsoperator

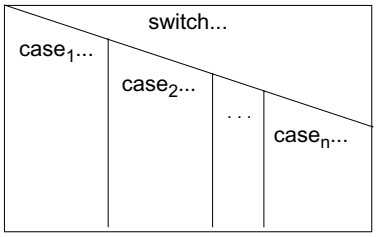
```
Syntax:logischer Ausdruck ? trueerg : falseerg
```

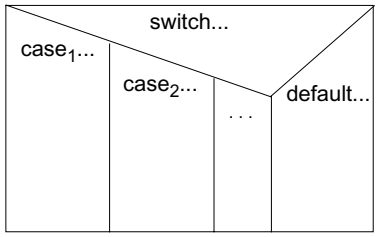
Beispiel:

```
maximum = x >= y ? x : y
```

4.9 Switch-Statement

```
Syntax:switch (Variable_prim | Expression_prim)
               case Value_0:
                   Statement_0
                   break;
               case Value_1:
                   Statement_1
                   break;
               case Value_n:
                   Statement_n
                   break;
               default_opt:
                   Statement_opt
```





4.9 Switch-Statement

- Das Switch-Statement ist, im Gegensatz zu anderen Sprachen, in Java in seiner Mächtigkeit stark eingeschränkt:
  - Die Testvariable bzw. der Testausdruck kann nur vom Typ *byte, char, short* oder *int* sein.
  - Es wird nur ein Test auf Gleichheit ausgeführt.
- Für alle anderen Fälle muß man auf das *If(Then)Else-If-Statement* zurückgreifen.

4.10 Schleifen(*loops*) - Konstrukte, Felder (*arrays*)

- Viele Lösungen - nicht nur mathematischer - Probleme werden iterativ herbeigeführt. Typischerweise liegen die zu bearbeitenden Daten in Form von ein- oder mehrdimensionalen Feldern (*arrays*) vor.
- In Java sind *arrays* spezielle Objekte, die als Einzelelemente primitive Datentypen oder wiederum Objekte enthalten können.
- Ein *array* kann jeweils nur Elemente eines Typs enthalten.

4.10.1 Felder (*arrays*)

- Deklaration einer *array*-Variablen:

```
String vornamen[];
long bigNumbers[][];
char kleinBuchstaben[];
```

  Die Anzahl der [ ] definiert die Dimension des *arrays*.

- Die Klammern können auch bei der Typbezeichnung stehen. Dann gelten sie (akumulativ) zu den Dimensionen, die bei den Variablen stehen:

```
int [][] a, b[], c[][];
```

  ist äquivalent zu

```
int a[][], b[][][], c[][][][];
```

Regel: Um die Deklarationen übersichtlich zu gestalten, sollte man die Klammern nur dann beim Typ angeben, wenn alle Variablen des Statements die gleiche Dimension besitzen, also z. B.:

```
String [][] a, b, c;
```

4.10.1 Felder (*arrays*)

- Wie Sie sicherlich bemerkt haben, handelt es sich um leere Klammern; wir haben noch keine Aussage über die Länge bzw. Größe der Felder getroffen. Bisher existieren lediglich Strukturinformationen.
  Der Grund dafür ist, dass wir bisher nur eine Variable vereinbart haben, über die wir zur Laufzeit *arrays* “referieren” wollen. Die Größe des Feldes wird also erst zur Laufzeit festgelegt, dann, wenn wir das Objekt instantiieren.
- *array* - Objekt - Instantiierung
  Es gibt zwei Methoden Array-Objekte zu instantiieren
  - mit Hilfe des Operators ***new*** oder
  - durch Initialisierung bei der Deklaration

    Beispiele:

```
String[] vornamen = {“Uwe”, “Paul”,“Georg”};
String[] vornamen = new String[3];
long bigNumbers[] = new long[100];
```


4.10.1 Felder (*arrays*)

- Zugriffe auf *array*-Elemente

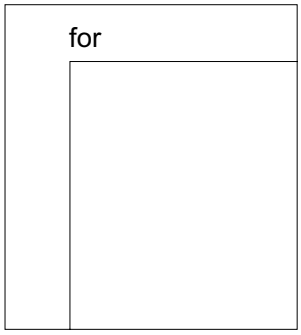
```
vornamen[1] = “Wilhelm”;
bigNumbers[99] = bigNumbers[0] + bigNumbers[10];
if (Vornamen[2] == “Georg”)
    Vornamen[0] = “Willi”;
```

Wie in C oder C++ werden auch in Java die Indizes von Feldern bei “0” zu zählen begonnen!!!

4.10.2 For - Statement

For-Statement

Syntax: for (ForInit$_{opt}$; Logical Expr$_{opt}$; ForUpdate$_{opt}$)
Statement



Beispiele:

```
for (int i = 0; i <= 1000; i++) {
    ...
}
for (;;) // ist ein gültiges For-Statement (forever).
```


4.10.2 For - Statement

Beispiele:

```
int[] intarr = new int[100];
int i;
for ( i = 0; i < 100; i++ )
    intarr[i] = i;
...
oder ...

for ( int i = 0; i < intarr.length; i++ ){
    intarr[i] = i;
    .....
}
```

Im 2. Beispiel benutzen wir die Instanzvariable “length” der Klasse *array (des Objekts intarr)*, um festzustellen, wann die Schleife terminieren soll. Mit dieser Vorgehensweise erreichen wir, dass dieses Codesstück robust gegen Änderungen der Länge des *arrays* ist; wir greifen also nicht auf “Implementierungswissen” zum Zeitpunkt des Programmentwurfs zurück.

4.10.2 For - Statement

Beispiel: Summation von 10 Feldelementen

```
...
sum = 0;   // Bevor eine Variable benutzt wird,
           //  muß ihr ein Wert zugewiesen werden

for (int i = 0, i < 10, i++) {
    sum = sum + feld[i];      // feld ist irgendwo init.
}
...
```


4.10.2 For - Statement

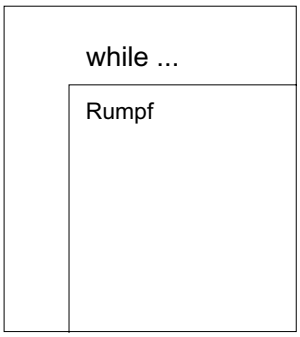
- Schleifenkonstrukte dürfen geschachtelt sein.

  Beispiel:

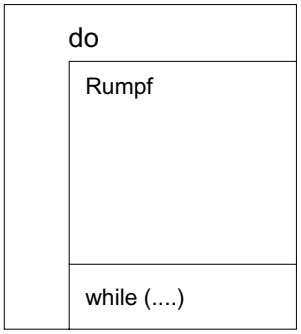
```
// Matrixmultiplikation
....
for (int i = 0; i < N; i++ {
    for (int j = 0; j < M; j++ {
        a[i,j] = 0.0
        for (int k = 0; k < O; k++ {
            a[i,j] = a[i,j] + b[i,k] * c[k,j];
    }
}
.....
```

4.10.3 While - Statement; Do - While - Statement

```
Syntax:while(logical Expr_opt)
          Statement
```




```
bzw.
Syntax:do
          Statement
       while(logical Expr_opt);
```



4.10.3 While - Statement; Do - While - Statement

- *For-, Do-* bzw. *Do-While*-Konstrukte sind äquivalent und ihre Benutzung ist eher eine Stilfrage.
  - Ein *for*-Konstrukt ist besonders geeignet, wenn man ohnehin bei der Berechnung mit Indizes hantieren muß und diese sich in einer konformen Weise verändern.
  - Das *while*-Konstrukt ist dann besonders geeignet, wenn es nur auf das Abprüfen einer Bedingung ankommt, wobei das Erreichen der Bedingung sich nicht an einer konformen Veränderung einer Variablen orientieren muß.
  - Das *do-while*-Konstrukt bietet sich an, wenn sichergestellt werden muß, dass der Schleifenrumpf unabhängig vom Zustand der Variablen zu Beginn der Schleifenberechnung, mindestens einmal durchlaufen werden soll.

4.10.3 While - Statement; Do - While - Statement

Beispiel:

```
...
short i, j;
...
    while(i > 0) {
        j = i;
        i = i + 1;
    }
System.out.println("Die grösste ganze Zahl ist: " + j)
System.out.println("Die kleinste ganze Zahl ist: " + i)
```

oder

```
...
    for (i=1; i>0; i++)
    j = i;
System.out.println("Die grösste ganze Zahl ist: " + j)
System.out.println("Die kleinste ganze Zahl ist: " + i)
    .....
```


4.11 Break-, Continue - Statement

- Es kommt vor, daß aufgrund von Bedingungen, die sich aus Berechnungen innerhalb des Rumpfes einer Schleife ergeben,
  - die Schleife terminieren soll:

`Syntax:break;`

  - oder mit der nächsten Iteration in der Berechnung fortgefahren werden soll:

`Syntax:continue`

Beispiel:

```
float[] array1, array2
int index = 0;
....
while (index < array1.length){
    if (array1[index] == 0)
        continue;
    array2[index] = 1./array1[index]
}
....
// Vermeidung einer Division durch 0
```

4.12 Labeled Loops

◆ Wie wir bereits wissen, können Schleifenkonstrukte ineinander geschachtelt sein. Um in der Lage zu sein aus geschachtelten Schleifenkonstrukten herauszuspringen, wurde die Möglichkeit eingeführt, *loops* zu *labeln:*

`Syntax:label:`

Beispiel:

```
....
out:
    for (int i = 0; i < 10; i++) {
        while (x < 50) {
            if (i * x > 400)
                break out;
            ....
        }
        ...
    }
```

- Labels sind für alle Control-Statements möglich; es gibt kein *goto*-Statement; es kann nur aus Schleifen **heraus**gesprungen werden.

4.13 Standard-E/A

■ Wie Sie bereits wissen, werden vom Betriebssystem 3 E/A-Kanäle automatisch geöffnet:

- standardIn: *Keyboard*
- standardOut: *Screen*
- standardError: *Screen*

■ Java stellt für das E/A-System geeignet Klassen und Methoden zur Verfügung. Diesem Thema ist ein eigenes Kapitel gewidmet. Für die ersten Aufgaben und Beispiele führen wir hier einen Minmalset ein und zwar:

- für die Ausgabe

```
System.out.println("text");      // Zeile ausdrucken
System.out.print("text");        // ohne \n
```

- für die Eingabe

```
BufferedReader von standardIn    // Zeile einlesen
```

4.13 Standard-E/A

- Die bereits von uns eingeführte Ausgabe ist relativ einfach in der Handhabung:
  In den auszugebenen *String* können mit dem “+ -Operator” numerische Werte integriert werden. Die Umwandlung von numerischen Werten in eine String-Zifferndarstellung auf dem *Screen* übernimmt die ausführende Instanz. Der Einfluß des Programmierers auf das Ausgabe-Layout beschränkt sich auf das Auffüllen mit Leerzeichen und der Einstreuung von Zeilenvorschüben.
- Etwas komplizierter ist das Einlesen von Daten, da wir vom *Keyboard* nur einzelne Zeichen oder Zeichenfolgen (*characters* oder *Strings*) einlesen können. **Eine Ziffernfolge ist kein Zahl sondern ein String.**

4.13 Standard-E/A

- Wir müssen also *Strings* in primitive Typen umwandeln.
  Hierfür gibt es geeignete Klassen!!-Methoden für Objekte!! der primitiven Typen, nämlich für Objekte des Typs *Integer, Float, Double,* etc. im allgemeinen in der Form (ab Java 1.2 bzw. Java 2):
  `Classname.parseClassname(string)`

  Beispiel:

```
int number;
float floatNumber;
.....
number = Integer.parseInt(ziffernString);
floatnumber = Float.parseFloat(ziffernString);
......
```

- Die Typschlüsselworte für primitive Datentypen werden klein geschrieben. Vereinbart man jedoch ein entsprechendes Objekt, dann wird der erste Buchstabe groß geschrieben, also: *float - Float; int - Integer; double - Double.*

4.13 Standard-E/A

- Für das Einlesen von Zeichenstrings vom *Screen* verwenden wir eine gepufferte Variante eines *InputStream*-Objekts, welches den *Inputkanal* vom *Screen* bereits geöffnet hat. Dieses Objekt stellt uns u. a. eine Methode zur Verfügung, die Zeilen vom *Screen* ließt: **`xxxx.readLine()`**

```
int number;
string ziffernString;
BufferedReader inStream;
....
instream = new BufferedReader(new InputStreamReader(System.in));
....
ziffernstring = inStream.readLine();
number = Integer.parseInt(ziffernString);
....
```

  Die letzten beiden Statements lassen sich zu einem zusammenfassen:

```
number = Integer.parseInt(instream.readLine());
```

  Dem Java-E/A-System ist ein eigenes Kapitel gewidmet, in dem die hier pragmatisch eingeführten Mechanismen genauer erläutert werden.

4.13 Standard-E/A

- Als letztes müssen wir noch das Problem lösen:
  "Was passiert bei falscher Eingabe?"
  - Das System merkt das und "wirft" eine *Exception*. *Exceptions* sind vom System generierte Unterbrechungssignale auf die der Benutzer geeignet reagieren sollte. Er "fängt" Sie auf und läßt ein Codestück abarbeiten, das den Fehler behandelt. In unserem Fall z. B. die Ausgabe einer Fehlermeldung mit der Bitte, die Eingabe zu wiederholen. Wird die *Exception* nicht bearbeitet, wird das Programm vom System terminiert.
  - Dem *Exceptionhandling* ist ebenfalls ein eigenes Kapitel gewidmet, deshalb werden wir zunächst keine Fehlerbehandlung vornehmen, also mit dem Abbruch des Programms im Fehlerfall leben müssen.

4.13 Standard-E/A

◆ Dreiecksbeispiel mit Eingabe von *StandardIn*

```
/* Testklasse der Klasse Dreieck */
/* Version 2: Mit Eingabe ueber StandardIN */
/* Nicht robust gegen fehlerhafte Eingabe */

import java.io.*;

class DreieckTestIOParseInt {
  public static void main (String args[])
    throws java.io.IOException {

    Dreieck triangel;
    int seiteA = 0; // Init. wegen "switch" !
    int seiteB = 0;
    int seiteC = 0;
    int number;
    int argCount = 0;

    System.out.println("Geben Sie 3 Seiten als Integer ein, Trennzeichen '\n'!");

    BufferedReader inStream = new BufferedReader(new
                                                 InputStreamReader(System.in));
```


4.13 Standard-E/A

```
    while (argcount < 3){
      argcount++;

      // get a line and transform to an integer

       number = Integer.parseInt(inStream.readLine());

      switch (argCount) {
      case 1:
        seiteA = number;
        System.out.println("Seite a: "+seiteA);
        break;

      case 2:
        seiteB = number;
        System.out.println("Seite b: "+seiteB);
        break;
      case 3:
        seiteC = number;
        System.out.println("Seite c: "+seiteC);
        break;
      }
    }
      triangel = new Dreieck(seiteA, seiteB, seiteC);
      triangel.dreiecksTyp();
  }
}
```

4.13 Standard-E/A

- Die Eingabeparameter der Methode *main() "String args[]"* dienen der Übernahme von Parametern aus der Kommandozeile.
  Übergeben wird ein Feld von *Strings*. In jedem Feld befindet sich der *String* eines Parameters. Die Trennzeichen für Parameter in der Kommandozeile sind - wie wir bereits wissen - Leerzeichen.
  Im Gegensatz zu C oder C++ steht im ersten Element des Feldes der 1. Parameter und nicht der Kommandoname!
- Die ggf. notwendige Umwandlung von *Strings* in primitive numerische Typen geschieht wie bei der Eingabe über *StandardIn*.

4.13 Standard-E/A

- Dreiecksbeispiel mit Eingabe aus der Kommandozeile:

```
/* Testklasse der Klasse Dreieck */
/* Version 3: Mit Eingabe ueber Kommandozeile */
/* Nicht robust gegen fehlerhafte Eingabe */

import java.io.*;

class DreieckTestParameter {
  public static void main (String args[]) {

    Dreieck triangel;
    int seiteA = 0;
    int seiteB = 0;
    int seiteC = 0;

    if (args.length != 3) {
      System.out.println("3 Seiten hat ein Dreieck! Versuch's noch einmal!");
      System.exit(-1);
    }
    seiteA = Integer.parseInt(args[0]);
    seiteB = Integer.parseInt(args[1]);
    seiteC = Integer.parseInt(args[2]);

    triangel = new Dreieck(seiteA, seiteB, seiteC);
    triangel.dreiecksTyp();
  }
}
```

4.14 Quellcode-Layout

- Klassennamen beginnen mit einem Großbuchstaben
- Methodennamen beginnen mit einem Kleinbuchstaben
- Variablennamen beginnen mit einem Kleinbuchstaben
  - Variablennamen/Methodennamen sollen “Aussagekraft” haben. Zur besseren Lesbarkeit verwendet man Großbuchstaben zur Gliederung des Namens. Man soll mit der Länge auch nicht übertreiben!
  - Variablennamen für Indizes, Laufvariablen, ggf. Hilfsvariable bestehen aus einem oder zwei Kleinbuchstaben.
  - Instanzvariablen sind grundsätzlich als *“private”* zu deklarieren. Ausnahmen sind zu begründen - besser: *access - methods*.
- Namenskonstante *(final static)* werden in Großbuchstaben geschrieben.
- Die öffnende geschweifte Klammer wird hinter dem “öffnenden” Statement geschrieben - nicht auf eine eigene Zeile.
- Die schließende Klammer steht auf einer eigenen Zeile.

4.14 Quellcode-Layout

- Statements
  - Nur bei sehr einfachen Statements darf mehr als ein Statement in einer Zeile stehen.
  - Eine “kompakte” Programmierung darf nicht zu Lasten der besseren Lesbarkeit gehen.
  - Bei komplexeren Ausdrücken ist der besseren Lesbarkeit wegen das Setzen von Klammern der Anwendung der *Precedence-Regeln* der Vorzug zu geben. Das gilt insbesondere für nichtarithmetische Operatoren!
  - Bei Verwendung von Variablen verschiedenen Typs in einem Ausdruck, ist durch *Casting* die Intension des Programmierers deutlich zu machen, ausgenommen(Details später):
    - Byte --> Short --> Integer --> Long
    - Float --> Double
    - (Byte, Short, Integer) --> Float
    - Long --> Double

4.14 Quellcode-Layout

- Bei komplexeren Sachverhalten wird zu einer Klasse oder einer Methode ein Kommentar erwartet.
  - Der Kommentar zu einer Methode beschreibt den Effekt der Methode; also die Auswirkungen des Methodenaufrufs auf den Objektzustand.
  - Der Kommentar zu einer Klasse beschreibt den “Zweck” der Klasse.
- Jede Klasse wird in einer separaten *Source*-Datei mit der *Extension “.java”* gehalten.
  Es ist zwar möglich mehrere Klassen in einer *Source*-Datei zusammenzufassen, jedoch kann nur eine Klasse *“public”* sein. Der Compiler generiert in jedem Fall für jede Klasse eine separate “*.class”*-Datei.

4.15 Zusammenfassung

- Quellcode
  - Zeichensatz, Zeile, Statement, ’A’ ungleich ’a’,
  - formatfreies Quellprogramm; Kommentare
  - Statement, Separatoren (Trennzeichen), Sonderzeichen, Schlüsselworte
- primitive Datentypen, String, array
  - numerisch, zeichen, logisch
  - byte, short, int, long, float, double, char, logical
  - Namen, Konstante, Variable, Felder, Dimensionen (Rang)
  - Wertebereich, Genauigkeit, Gültigkeitsbereich, (Lebensdauer)

4.15 Zusammenfassung

- Programmflußstrukturen
  - Ausdrücke *(Expressions)*, Zuweisungen *(Assignments)*, *Statements*, *Blockstatements*, *Label,* Abarbeitungsreihenfolgen *(Precedence)*
  - If - (Then) - Statement; If- (Then) - Else - Statement; If- (Then) - Else - If - Statement; Vergleichsoperator Switch-Statement
  - For - Statement; While - Statement; Do - While - Statement;
  - Break - Statement; Continue - Statement
  - Label
- Standard - E/A