Grundlagen der Informatik für Ingenieure I

4. Java Sprachkonstrukte

4.1 Java-Zeichensatz

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

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

4.2 Quellcode-Layout

- **Quellcode-Layout:**
  - formatfreies Quellprogramm
  - **Trennzeichen** sind <blanks>, <tabs> oder <newline> (neue Zeile).
  - Ein Semicolon schließt eine **Anweisung** *(Statement)* ab.
  - **Blöcke** werden in geschweiften Klammern eingeschlossen.
  - Reservierte Worte (**Schlüsselworte** (*keywords)*) sind geschützt und dürfen nicht als Variablennamen verwendet werden.
  - Die Blockstruktur des Programmcodes sollte durch Einrücken hervorgehoben werden (wird von ***xemacs*** unterstützt).

4.2 Quellcode-Layout

- **Kommentare:**

  Es ist ausgesprochen hilfreich - insbesondere zum späteren Verständnis - die Funktionalität einer **Klasse** und seiner **Methoden** knapp und präzise im “Kopf” der Klasse bzw. Methode zu beschreiben.

  - Beispiele:

```
// der Rest der Zeile ist ein Kommentar

/* Beliebige Texte ueber mehrere Zeilen */

/** Dieser Code wird von “javadoc” für Dokumentationszwecke
    herausgezogen. */
```

4.3 Konstante und Variable

- **Konstante:**
  - ◆ Konstanten sind Größen, die nicht verändert werden können (im Gegensatz zu Variablen).
  - ◆ Eine Konstante hat einen **Typ** und einen **Wert**.
  - ◆ Es gibt zwei Arten von Konstanten:
    - **Literale Konstante** *(literals = wörtlich):*
      - der Typ geht implizit aus der Schreibweise hervor.
      - Beispiele:

```
Typ float: 3.141592;
Typ int:   40000;
Typ char:  'S';
```


4.3 Konstante und Variable

- **Konstante** (cont.)**:**
  - **Namenskonstante:**
    - die Konstante erhält einen Namen
    - **Syntax**:

```
final type CONSTNAME = value;
```

    - Beispiele:

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

    - Für Namen von Namenskonstanten werden nur Großbuchstaben verwendet.

4.3 Konstante und Variable

Variable

- Einer Variablen ist ein Speicherplatz zugeordnet.
- Der Name der Variablen ist die symbolische Bezeichnung (Adresse) dieses Speicherplatzes im **Arbeitsspeicher**
- Eine Variable hat
  - einen **Namen**
  - einen **Typ** (und damit einen Wertebereich),
  - einen aktuellen **Wert**,
  - einen **Gültigkeitsbereich** (von wo aus kann auf die Variable zugegriffen werden)
  - und eine **Lebensdauer**
- Eine Variable muß **vereinbart** (declared) werden.
- Mit **Anweisungen** kann man Variablen Werte zuweisen.
- Vor der Verwendung müssen den Variablen Werte zugewiesen werden.

4.3 Konstante und Variable

- Java unterscheidet drei **Arten** von Variablen
  - **Objekt-Variable** (Instanz-Variable)
    - beschreiben den Zustand eines Objekts;
    - ihr Geltungsbereich ist “objektglobal”.
  - **Klassen-Variable**
    - haben Ähnlichkeit mit Objektvariablen,
    - jedoch sind diese Variablen allen Objekten dieser Klasse gemeinsam.
  - **Lokale Variable**
    - werden innerhalb einer Methode oder eines Blocks vereinbart;
    - ihr Geltungsbereich ergibt sich aus dem Ort der **Deklaration** (Vereinbarung)

4.3 Konstante und Variable

◆ In Java können Variablen von folgendem **Typ** sein:

- einer der acht **primitiven Typen**
- **Objekt** (Instanz)
- String, Feld ***(array)*** - in Java, Objekte “besondere Art”

◆ **Variablennamen**

- dürfen außer aus Sonderzeichen ***(special characters)*** aus allen Zeichen des Zeichensatzes bestehen,
- dürfen nicht mit einer Ziffer beginnen.
- dürfen nicht identisch mit einen ***keyword*** (Schlüsselwort) sein.

4.3 Konstante und Variable

◆ **Variablennamen** (cont.)**:**

Konvention:

- 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
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;
```


4.3 Konstante und Variable

◆ **Variablendeklarationen** können

- grundsätzlich an beliebiger Stelle stehen
- in Verbindung mit einer Zuweisung auftreten.
- **Regel**:

  In einem wohlstrukturierten Code stehen die **Deklarationen** jeweils am Anfang von Klasse bzw. Methode zusammengefaßt **vor** dem jeweiligen **Codeteil**.

4.4 Primitive Datentypen

◆ Ein primitiver Datentyp ist durch die Wertemenge und Operationen definiert.

- Wertemenge: Werte die der Datentyp annehmen kann
- Operationen, die auf dem Datentyp möglich sind

◆ Primitive Datentypen sind “integraler” Bestandteil der Sprache.

- **numerische** Datentypen:
  - **ganzzahlig**: byte, short, int, long
  - **reell**: float, double
- **nichtnumerische** Datentypen:
  - **logisch**: boolean
  - **Zeichen**: char

1 Ganzzahliger Datentyp

◆ **byte, short, int, long:**

- Operationen: `+ - * / % (modulo)`
- 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

1 Ganzzahliger Datentyp

- 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
```


2 Reeller und doppeltgenauer Datentyp

◆ **float, double:**

- Operationen: + - * / %
- Wertebereich:
  - 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
    $0 < m < 2^{52}$
    $e = [-1075, +970]$

2 Reeller und doppeltgenauer Datentyp

- 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
  - Beispiele:

```
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*

3 Zeichendatentyp

◆ **char:**

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

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

3 Zeichendatentyp (cont.)

- Ersatzdarstellung für nicht druckbare Zeichen:

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 Logischer Datentyp

◆ **boolean:**

- Operationen:
  - UND: &&
  - ODER: ||
  - exclusive ODER: ^
  - NOT: !
- Wertebereich: true, false
- Schreibweise der literalen Konstanten:

```
true  false
```

4.5 Zeichenkette (String)

- Zeichenketten in Java sind **Objekte** (Instanzen) der *class String.*
  - Da Zeichenketten Objekte sind, handelt es nicht einfach um eine Folge von Elementen 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 erzeugen (ohne ***new***)

```
“Hallo, ich bin eine Zeichenkette\n”
“A string with a \t tab in it”
“Mercedes\u2122 ist ein gesch\u00FCtztes Markenzeichen”
```

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

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

4.6 Ausdrücke und Zuweisungen

- **Ausdrücke** (Expressions)***:***
  - Formeln zur Berechnung (oder Bildung) eines Wertes.
  - bestehen aus Operanden, Operatoren und/oder runden Klammern.
  - Beispiele:

```
-a + b/z;                    (Arithmetischer Ausdruck)

(a / b) + (a*b);             (Arithmetischer Ausdruck)

-a+b+c;                      (Arithmetischer Ausdruck)

17 + 4 <= cardDeck           (Vergleichsausdruck)

x && y                       (logischer Ausdruck)
```

4.6 Ausdrücke und Zuweisungen

- **Ausdrücke** (cont.):
  - werden bei gleicher Wertigkeit der Operatoren *(precedence)* von links nach rechts ausgewertet.
  - Durch Setzen 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 Intention des Programmierers aus.

4.6 Ausdrücke und Zuweisungen

◆ *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 Zuweisungen

◆ *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 Zuweisungen

■ **Zuweisung** (Assignment):

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

Beispiel:

```
a = 25.5 * 3.1E9
```

◆ Die rechte Seite eines Ausdrucks wird zuerst ausgewertet.

◆ Dann wird das Ergebnis der Variablen auf der linken Seite zugewiesen.

◆ Deshalb sind Anweisungen der Form

```
x = x + y
```

unproblematisch.

◆ Eine Zuweisung ist keine math. Gleichung!

4.6 Ausdrücke und Zuweisungen

◆ Für diesen Typ der Zuweisungen wurde ein Satz spezieller Zuweisungsoperatoren 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
```

◆ 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.
```


4.6 Ausdrücke und Zuweisungen

◆ 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 = 1/8 ist 0!
y = -11/3  ist -3
```

◆ **Vergleichsoperatoren** (Relationen):

- Ergebnis einer Vergleichsoperation ist vom Typ *boolean*: *true, 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)

- Ein Programm besteht aus einzelnen Anweisungen *(Statements)*:
  - Zuweisungen
  - Funktionsaufrufe
  - Deklarationen
- Die Anweisungen eines Programms werden Zeile für Zeile oder besser *Statement* für *Statement* ausgeführt.
- Mit Programmsteueranweisung kann man den Kontrollfluß
  - ändern,
  - unterbrechen oder
  - beenden.

4.7 Programmsteuerungsanweisungen

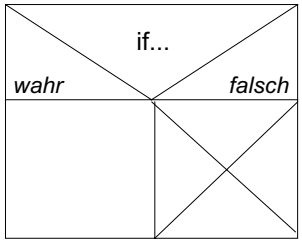
- ***Statement***
  - Ein *Statement* ist eine einzelne Anweisung.
  - Das Trennzeichen für *Statements* ist das Semicolon.
- ***Block Statement***
  - Ein *Blockstatement* ist die Zusammenfassung einzelner *Statements*, geklammert durch geschweifte Klammern.
  - Innerhalb von Blöcken vereinbarte lokale Variable sind nur dort gültig (Gültigkeitsbereich).
  - Ein Block kann überall dort stehen, wo auch ein einzelnes *Statement* stehen kann.

1 Bedingte Anweisungen

- ***If - Statement*** :

Nassi-Shneiderman-Diagramm

Syntax: `if ( logical_expr ) Statement`



Beipiele:

```
if ( flag == true )
    index = 0;

if ( x-y <= schranke )
    wert = x-y;
```

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

Beispiel: Vertauschen der Werte zweier Variablen

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

(man benötigt eine Hilfsvariable)

1 Bedingte Anweisungen

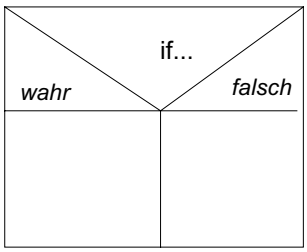
- ***If - Else - Statement***:

Syntax:
```
if ( logical_expr )
       Statement1
   else
       Statement2
```

Beispiel:

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

Symbol:

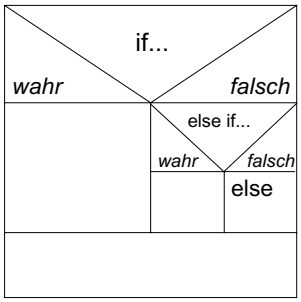


1 Bedingte Anweisungen

- ***If - Else -If - Statement***:

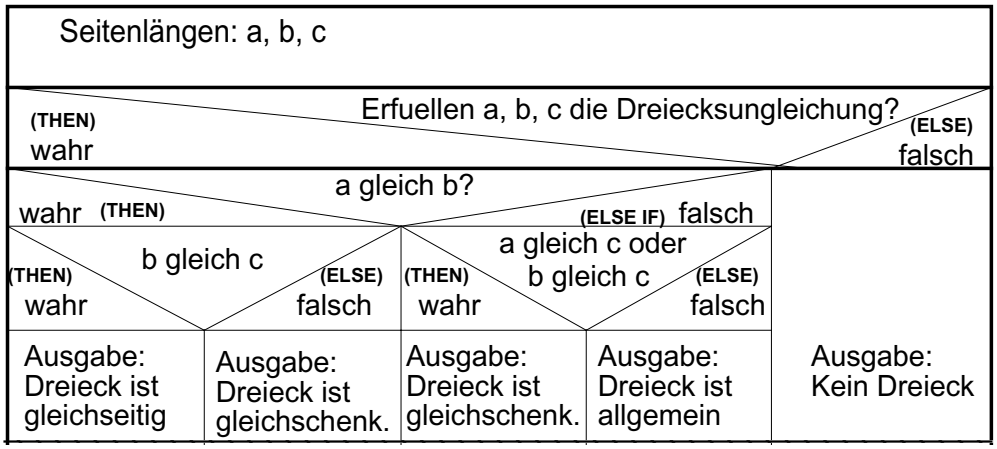
Syntax:
```
if ( logical_expr )
    Statement1
else if( another_logical_expr )
    Statement2
else
    Statement3
```

Symbol:



1 Bedingte Anweisungen

Beispiel: Anwendung der Dreiecksungleichung



1 Bedingte Anweisungen

- **Beispiel**: Anwendung der Dreiecksungleichung

```
class Dreieck {

  private int a, b, 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 () {

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


1 Bedingte Anweisungen

- **Beispiel**: Anwendung der Dreiecksungleichung (cont.):

```
    if ( a + b > c && a + c > b && b + c > a ) {

      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" );
  }
}
```

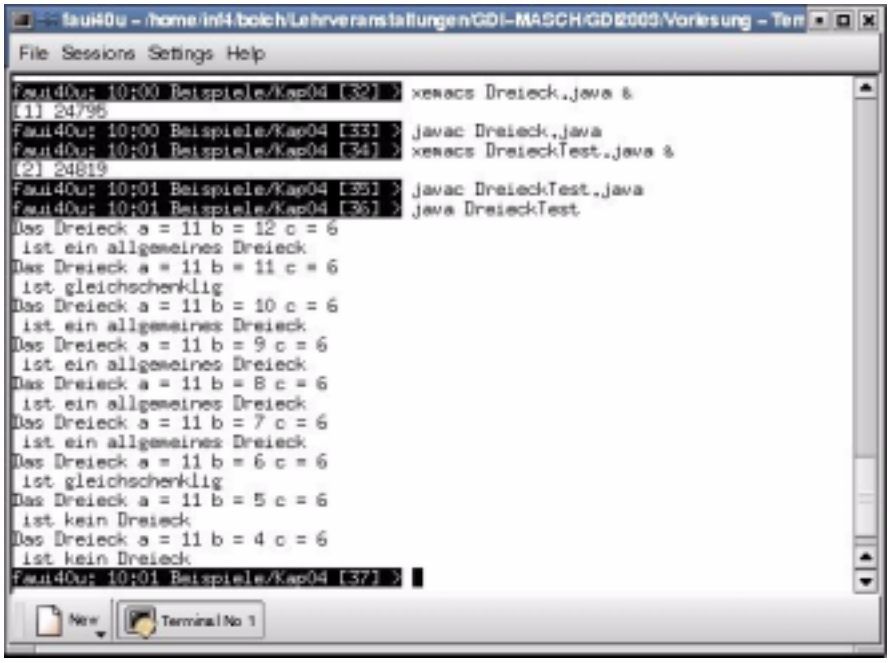
1 Bedingte Anweisungen

- **Beispiel**: Anwendung der Dreiecksungleichung (cont.)

```
/* 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();
    }
  }
}
```

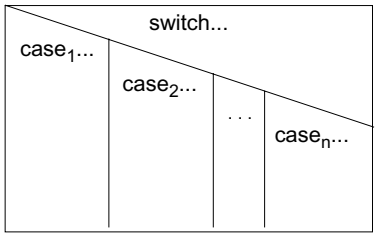

1 Bedingte Anweisungen

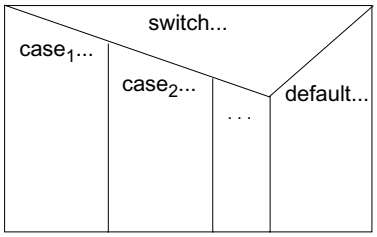
- **Beispiel**: Testausgabe



2 Mehrfachbedingungen

Syntax:
```
switch ( Variable | Expression ) {
        case Value1:
            Statement1
            break;
        case Value2:
            Statement2
            break;
        case Valuen:
            Statementn
            break;
        defaultopt:
            Statementopt
    }
```





2 Mehrfachbedingungen

- 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-Else-If-Statement* zurückgreifen.

2 Mehrfachbedingungen

- **Beispiel:**

```
switch ( option ) {

        case 'A': System.out.println( "Gewählte Option Abbrechen" );
        break;

        case 'S': System.out.println( "Gewählte Option Speichern" );
        break;

        case 'L': System.out.println( "Gewählte Option Laden" );
        break;

        default: System.out.println( "Ungültige Option" );
        break;
}
```


3 Felder (*arrays*)

- Bei vielen Problemen 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 vom gleichen Typ enthalten.

3 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 (akkumulativ) zu den Dimensionen, die bei den Variablen stehen:

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

ist äquivalent zu

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


3 Felder (*arrays*)

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, z. B.:

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

- Die Klammern sind leer, da bisher noch keine Aussage über die Länge bzw. Größe der Felder getroffen wurden.
- Bisher existieren lediglich Strukturinformationen --> Dimension des Feldes.
- Die Größe des Feldes wird erst festgelegt, wenn das Objekt erzeugt (instantiiert) wird.

3 Felder (*arrays*)

◆ *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 = new String[3];
long bigNumbers[] = new long[100];
String[] vornamen = { "Uwe", "Paul","Georg" };
```


3 Felder (*arrays*)

◆ Zugriffe auf *array*-Elemente

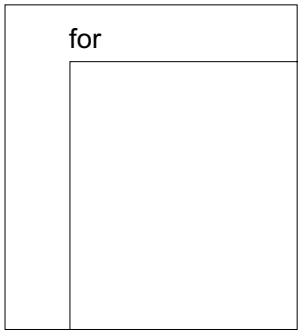
```
vornamen[1] = "Wilhelm";
bigNumbers[99] = bigNumbers[0] + bigNumbers[10];
if ( vornamen[2] == "Georg" )
    vornamen[0] = "Willi";
```

Wie in C oder C++ beginnt man auch in Java mit der Zählung der Indizes von Feldern bei "0" !!!

4 Schleifen

- ***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 Schleifen

Beispiele (cont.):

```
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.

4 Schleifen

- ***For-Statement*** (cont.):

Beispiel: Summation von 10 Feldelementen

```
...                                k

sum = 0;

for ( int i = 0; i < 10; i++ ) {

    sum = sum + feld[i];
}
...
```


4 Schleifen

- Schleifenkonstrukte dürfen geschachtelt sein.

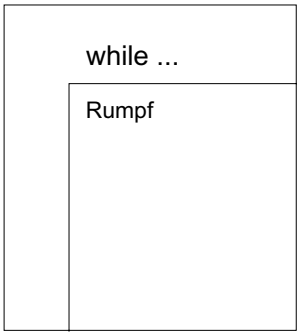
Beispiel:

```
// Matrixmultiplikation c_ij = Σ(a_ik * b_kj)

....
int i, j, k;
for ( i = 0; i < I; i++ ) {
    for ( j = 0; j < J; j++ ) {
        c[i,j] = 0.0;
        for ( k = 0; k < K; k++ ) {
            c[i,j] = c[i,j] + a[i,k] * b[k,j];
        }
    }
}
.....
```

4 Schleifen

While - Statement:

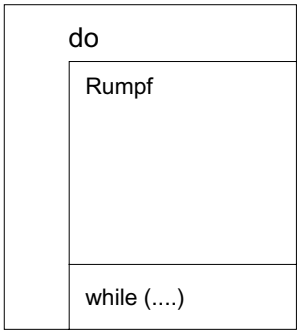
Syntax:
```
while ( logical Expr )
    Statement
```



4 Schleifen

Do - While - Statement:

Syntax:
```
do
    Statement
while ( logical Expr );
```



4 Schleifen

◆ *For-, While-* bzw. *Do-While*-Konstrukte sind weitgehend äquivalent:

- Ein *for*-Konstrukt ist dann sinnvoll, wenn man ohnehin bei der Berechnung mit **Indizes** hantieren muß und diese sich in einer **konformen Weise** verändern.
- Das *while*-Konstrukt ist dann sinnvoll, 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 ist dann sinnvoll, wenn der Schleifenrumpf unabhängig vom Zustand der Variablen zu Beginn der Schleife **mindestens einmal** durchlaufen werden soll.

4 Schleifen

Beispiel: Summation von 10 Feldelementen (siehe Seite 49)

....

```
double sum = 0;
int i = 0;
while ( i < 10 ) {
    sum = sum + feld[i];
    i++;
}
```

....

4 Schleifen

Beispiel: Drucken der Potenzen von 2 mit dem **Do - While - Statement**

```
....
long i = 1;
do {
    i = i * 2;
    System.out.print( i + " " );
} while ( i < 300000000000 );
....
```


4 Schleifen

Break-, Continue - Statement:

- Es kommt vor, daß aufgrund von Bedingungen, die sich 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;`

4 Schleifen

- **Continue-Statement:**

Beispiel: Vermeidung einer Division durch 0,
durch Verlassen der aktuellen Iteration mit **`continue`**

```
....
float[] array1, array2;
int index;
....
for ( index = 0; index < array1.length; index++ ) {
    if ( array1[index] == 0 )
        continue;
    array2[index] = 1./array1[index];
}
....
```


4 Schleifen

- **Break-Statement:**

Beispiel: Verlassen einer Schleife mit **`break:`**

```
....
int i;
for ( i = 0; i < 1000; i++ ) {
    // wenn i den Wert 500 hat, die Schleife verlassen
    if ( i == 500 )
        break;
}
....
```

4 Schleifen

Labeled Loops*:*

- Schleifenkonstrukte können ineinander geschachtelt sein.
- Mit Labeled Loops kann man aus geschachtelten Schleifenkonstrukten herausspringen:

Syntax: `label:`

Beispiel:

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


4 Schleifen

Labeled Loops (cont.):

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

4.8 Standard-E/A

- Vom Betriebssystem werden 3 E/A-Kanäle automatisch geöffnet:
  - standardIn: *Keyboard*
  - standardOut: *Screen*
  - standardError: *Screen*

- In Java gibt es für das E/A-System geeignete Klassen und Methoden.
  Diesem Thema ist ein eigenes Kapitel gewidmet.
  Für die ersten Aufgaben und Beispiele führen wir hier ein Minimalset ein:

  - für die Ausgabe

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

  - für die Eingabe
    - *BufferedReader von standardIn* `// Zeile einlesen`
    - Von der Kommandozeile einlesen

4.8 Standard-E/A

- **Ausgabe**:

  - 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 Methode.

  - Der Einfluß des Programmierers auf das Ausgabe-Layout beschränkt sich auf das Auffüllen mit Leerzeichen und der Einstreuung von Zeilenvorschüben.

  - Beispiel:

```
....
anz = 100000;
System.out.println( "Erlangen hat " + anz + "Einwohner" );
....
```

4.8 Standard-E/A

- **Eingabe** vom Keyboard:
  - Vom *Keyboard* können nur einzelne Zeichen oder Zeichenfolgen (*characters* oder *Strings*) eingelesen werden.
  - **Eine Ziffernfolge ist kein Zahl sondern ein String.**
  - Wir müssen also ***Strings*** in primitive Typen umwandeln.
  - Hierfür gibt es geeignete Methoden für Objekte der primitiven Typen: *Integer, Float, Double,* etc. im allgemeinen in der Form:

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

4.8 Standard-E/A

- 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.*

4.8 Standard-E/A

- Für das Einlesen von Zeichenstrings von der Tastatur verwenden wir eine gepufferte Variante eines *InputStream*-Objekts.
- Dieses Objekt stellt uns u. a. eine Methode zur Verfügung, die ganze Zeilen liest: **`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 zusammenfassen:

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


4.8 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 Unterbrechungen, 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 aufgefangen, wird das Programm vom System mit einer Fehlermeldung abgebrochen.
- Dem *Exceptionhandling* ist ein eigenes Kapitel gewidmet, deshalb werden wir zunächst keine Fehlerbehandlung vornehmen und mit dem Abbruch des Programms im Fehlerfall leben müssen.

4.8 Standard-E/A

◆ Dreiecksbeispiel mit Eingabe von *StandardIn* :

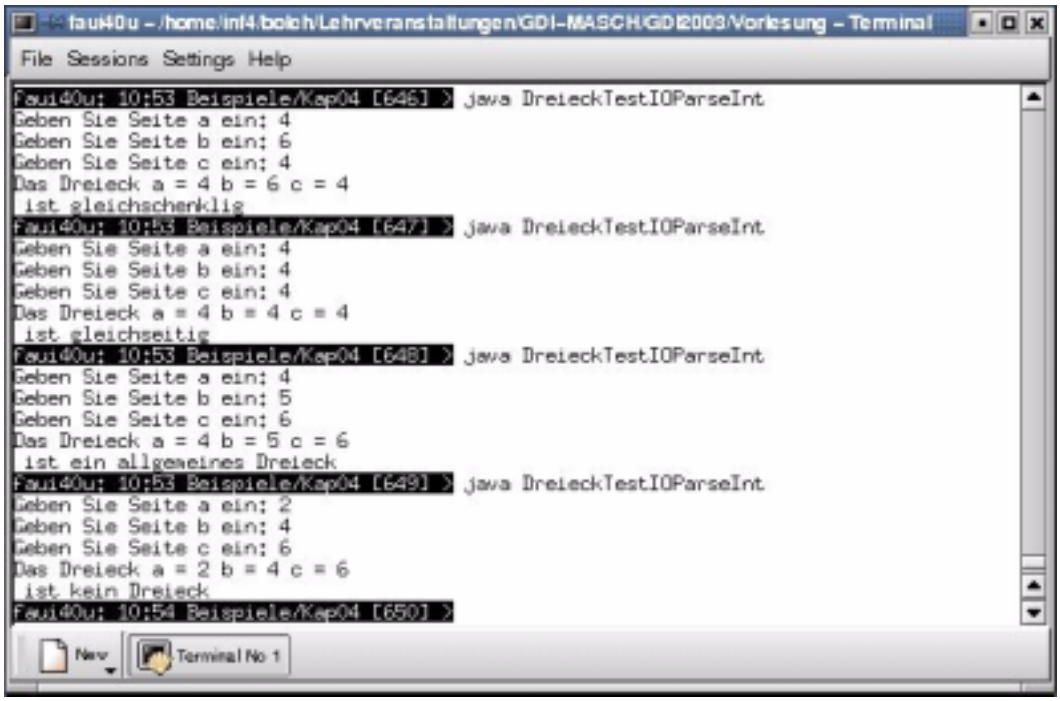
```
/* Testklasse der Klasse Dreieck */
/* Version 2: Mit Eingabe ueber StandardIN */
/* Nicht robust gegen fehlerhafte Eingabe */

import java.io.*;

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

    Dreieck triangel;
    int seiteA;
    int seiteB;
    int seiteC;

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


4.8 Standard-E/A

◆ Dreiecksbeispiel mit Eingabe von *StandardIn (*cont.):

```
      System.out.println( "Geben Sie Seite a ein: " );
      seiteA = Integer.parseInt( inStream.readLine() );

      System.out.println( "Geben Sie Seite b ein: " );
      seiteB = Integer.parseInt( inStream.readLine() );

      System.out.println( "Geben Sie Seite c ein: " );
      seiteC = Integer.parseInt( inStream.readLine() );

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

◆ Dreiecksbeispiel mit Eingabe von StandardIn (Ergebnisse):



4.8 Standard-E/A

Eingabe aus der Kommandozeile:

- Die Parameter der Methode ***main( ): "String args[]"*** dienen der Übernahme von Parametern aus der Kommandozeile.
- Übergeben wird ein Feld von *Strings*.
- In jedem Feld-Element befindet sich ein Parameter als *String.*
- Die Trennzeichen für Parameter in der Kommandozeile sind Leerzeichen (z.B.: **`java DreieckTest 3 5 7`**)
- Die ggf. notwendige Umwandlung von *Strings* in primitive numerische Typen geschieht wie bei der Eingabe über *StandardIn*. mit den entsprechenden Parse-Methoden (z.B. ***IntegerparsInt()***).

4.8 Standard-E/A

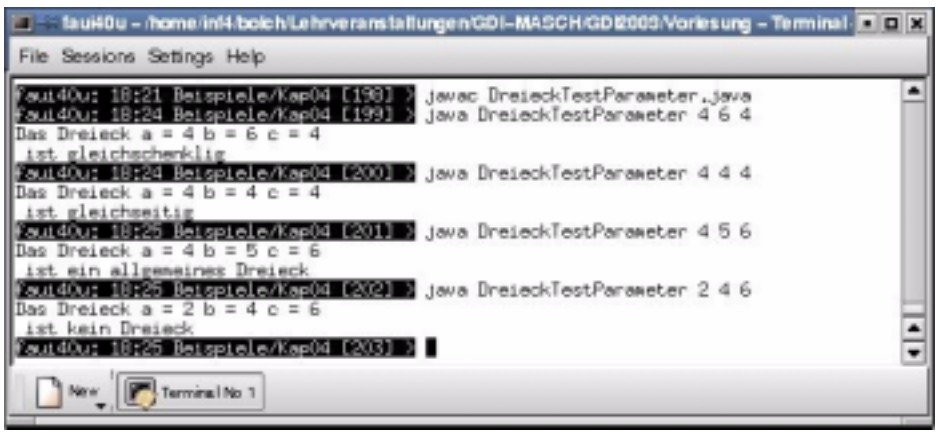
◆ Dreiecksbeispiel mit Eingabe aus der Kommandozeile:

```
/* 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();
  }
}
```


4.8 Standard-E/A

◆ Dreiecksbeispiel mit Eingabe aus der Kommandozeile:

• Ergebnisse:



4.9 Layout-Konventionen

- Klassennamen beginnen mit einem Großbuchstaben

  `MotorBike, Dreieck`

- Methodennamen beginnen mit einem Kleinbuchstaben

  `startEngine, dreiecksTyp`

- Variablennamen beginnen mit einem Kleinbuchstaben

  `a, b, i, anz, color, seiteA, engineOn, ....`

  - Die verwendeten Namen sollen “Aussagekraft” haben. Zur besseren Lesbarkeit verwendet man Großbuchstaben zur Gliederung des Namens:

    `familienName, currentTemperatureInNewYork`

  - Man soll mit der Länge auch nicht übertreiben!
  - Variablennamen für Indizes, Laufvariablen, ggf. Hilfsvariable bestehen aus einem oder zwei Kleinbuchstaben.
  - Objektvariablen sind grundsätzlich als “private” zu deklarieren.

4.9 Layout-Konventionen

- Namenskonstante *(final static)* werden in Großbuchstaben geschrieben:

  `MAXSIZE, DIAPERS`

- Die öffnende geschweifte Klammer wird hinter dem “öffnenden” Statement geschrieben - nicht auf eine eigene Zeile:

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

- Die schließende geschweifte Klammer steht auf einer eigenen Zeile:

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

        Dreieck triangel;
        int seiteA;
        ....

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

4.9 Layout-Konventionen

◆ **Statements** (Anweisungen):

- Jedes Statement sollte in einer neuen Zeile stehen.
- 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!
- Nicht zu viele Leerzeilen verwenden!

4.9 Layout-Konventionen

◆ 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 Quell-Datei** (*Source*-Datei) mit der *Extension **".java"*** gehalten:

- Es ist möglich (aber meist unerwünscht) 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.

4.10 Zusammenfassung

- Zeichensatz
- Konstante und Variable
- Primitive Datentypen *(byte, short, long, int, char, boolean, float, double)*
- Zeichenketten *(Strings)*
- Ausdrücke *(Expressions)* und Zuweisungen *(Assignments)*
- Programmsteueranweisungen *(Control Statements)*
  - Bedingte Anweisungen *(if, else)*
  - Mehrfachbedingungen *(switch)*
  - Felder *(arrays)*
  - Schleifen *(for, while, do - while, continue, break, label)*
- Quellcode-Layout
- Standard-E/A

Notizen