

III. Programmierung im Kleinen – Namen und Dinge

- 1. Variable und einfache Typen**
- 2. Primitive Datentypen**
- 3. Datentypen und Programmstruktur**

2. Variablen und einfache Typen

- Bezeichner
- Variablen in Programmen
- Daten – Typ und Deklaration
- Literale und Konstanten

Bezeichner

Namen (*identifizier*) für Datenobjekte

- Frei gewählte (sinnvolle!) **Bezeichnungen** für „Elemente“ oder „Dinge“, also **Variablennamen**, Namen für **Konstante**, **Klassennamen**, **Methodennamen**, ...
- Der **Zugriff auf die Elemente** erfolgt über ihren Namen
- Java-Bezeichner bestehen aus
 - **Buchstaben**, **Ziffern** und den **Sonderzeichen** ‘_’ und ‘\$’,
 - wobei das **erste Zeichen keine Ziffer** sein darf,
 - außerdem gibt es einige **reservierte Wörter**, die nicht als Bezeichner verwendet werden dürfen

Bsp. (und Gegenbeispiele):

Reservierte Wörter

Die folgenden **Schlüsselwörter** dürfen nicht als Bezeichner verwendet werden; sie sind fester Bestandteil der Sprache Java

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Variable

Daten und Variable

- **Variablen** sind **Behälter** für Werte, die im Verlauf eines Programms auftreten; dieser Behälter hat stets einen **Namen**
- Behälter zur Ablage von **Daten** (eines bestimmten **Typs**)

Verwendung – Zuweisung von Daten

- **(Programm-) Variablen** besitzen stets einen **Wert**
- **Zuweisung** (Details später):

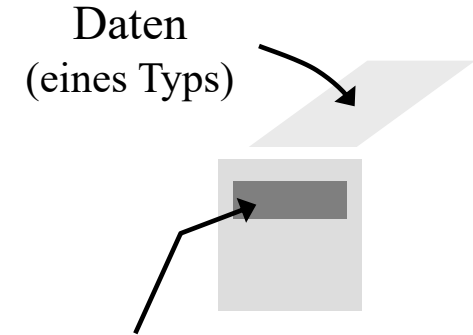
`<variable> = <Ausdruck>;`

- Ein **Gleichheitszeichen** bedeutet nicht wie in der Mathematik, dass die beiden Terme gleich sein müssen, sondern der Wert des Ausdrucks auf der rechten Seite wird der Variable auf der linken Seite zugewiesen

```
int i;

i = 5;
i = 12;
i = i + 7;
```

i:



Name = Variable

Deklaration von Variablen:

```
int      index;
String   name;
boolean  isGreen;
double   price;
```

Wertzuweisung:

```
index    = 5;
name     = "Ben";
isGreen  = false;
price    = 3.5;
```

Typen

- **Datentypen** legen die möglichen Werte einer Variable fest
- **Variablen** müssen **deklariert** werden, dabei werden
 - **Name** und
 - **Datentyp**

der Variable festgelegt:

```
<Datentyp> <Variablen-Name>;
```

Hinweise und Besonderheiten:

- Durch den **<Datentyp>** wird der **Wertebereich** einer Variable eingeschränkt sowie die möglichen **Operationen** festgelegt
- **<Variablen-Namen>** bestehen aus einer Folge von Buchstaben, Ziffern und '_'; Schlüsselwörter sind ausgenommen (vgl. Java-Bezeichner auf S.13)

Deklaration von Variablen:

```
int      index;
String   name;
boolean  isGreen;
double   price;
```

Wertzuweisung:

```
index    = 5;
name     = "Ben";
isGreen  = false;
price    = 3.5;
```

Daten – Typ und Deklaration

Typisierung

- Eine **Variable** in Java kann nur Daten **eines bestimmten Typs** speichern – man spricht von einer **strengen Typisierung** einer Sprache
- Java kennt so genannte **einfache Typen** (*primitive types*)

Es gibt:

<code>byte, short, int, long</code> :	für ganze Zahlen (Integer-Werte)
<code>float, double</code> :	für Gleitkommazahlen (reelle Zahlen in der verfügbaren Rechnergenauigkeit)
<code>char</code> :	Zeichen aus einem Zeichensatz (Unicode)
<code>boolean</code> :	logische Werte (<code>true</code> , <code>false</code>)

Verwendung: Die **unterschiedlichen Datentypen** legen die Operationen fest, die auf den Daten ausgeführt werden dürfen; die verschiedenen (**Unter-**) **Typen** für die Darstellung ganzer bzw. Gleitkomma-Zahlen legen fest, wieviel Speicher (in Bytes) für eine Variable reserviert wird

Die Details werden nachfolgend vertieft!

Zur Deklaration von Variablen

- Es können einzelne oder auch **mehrere Variablen** vom selben Typ auf einmal deklariert werden

```
<Datentyp> variable1;  
<Datentyp> variable2;
```

oder

```
<Datentyp> variable1, variable2;
```

Dabei kann sich die Deklaration über **mehrere Zeilen** erstrecken und auch **Kommentare** enthalten:

```
<Datentyp> variable1, // Funktion der Variable1  
            variable2; // Funktion der Variable2
```

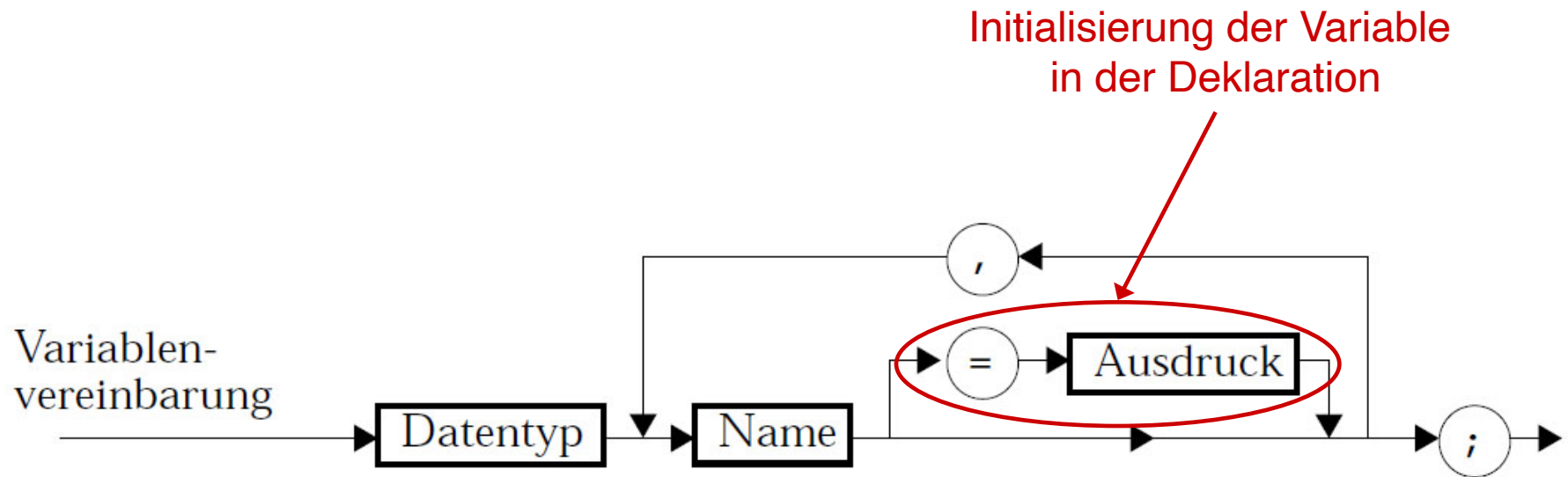
- Variablen müssen vor der Verwendung **initialisiert** werden, d.h. einen Anfangswert (aus dem Wertebereich ihres Types) zugewiesen bekommen

Bei der **Deklaration** einer Variablen kann dieser **gleichzeitig** ein **initialer Wert** zugewiesen werden

```
<Datentyp> variable = initialerWert;
```


Struktur einer Variablendeklaration in Java

Syntaxdiagramm



(K. Echte, M. Goedicke. Lehrbuch der Programmierung mit Java, 1. Auflage. dpunkt, Heidelberg, 2000)

Gültigkeitsbereiche von Variablen

- Eine Variable kann in Java ab dem Zeitpunkt ihrer Deklaration bis zum Ende der Methode **verwendet** werden

```
int i;  
  
i = 3;      OK  
j = 4;      Fehler (bei Übersetzung)!  
int j;  
j = 6;      OK
```

- Variablen können in einer Klasse **auch außerhalb von Methoden deklariert** werden; dann muss der Deklaration (vorerst) ein **static** vorangestellt werden (Klassen-Variablen)

```
public class Prog {  
    static int k = 5;  
  
    public static void main(String[] args) {  
        k = 7;  
        ...  
    }  
}
```

Literale und Konstanten

Literale

- Literale sind fest definierte konstante Werte, die innerhalb von Ausdrücken verwendet werden können

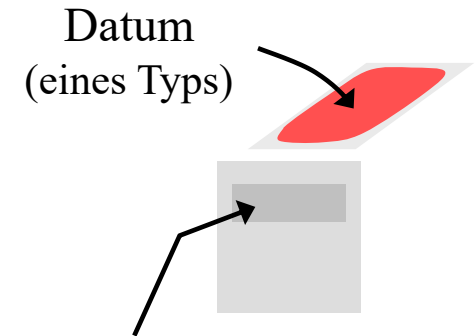
Bsp.: 674, 3.14, **true**, **false**, **null**

Konstanten

- Häufig soll eine Größe festgelegt werden, die einen festen unveränderlichen, d.h. während der Programmlaufzeit **konstanten**, Wert besitzt
- Mit dem reservierten Java-Schlüsselwort **final** kann eine Konstante festgelegt werden

Bsp.: `final int MAX_COUNT = 10;`

Konvention: Konstanten werden in **Groß-Buchstaben** geschrieben, bei **Zusammensetzungen** wird der Bezeichner **mit Unterstrich** zusammengefügt



Name = **Konstante**

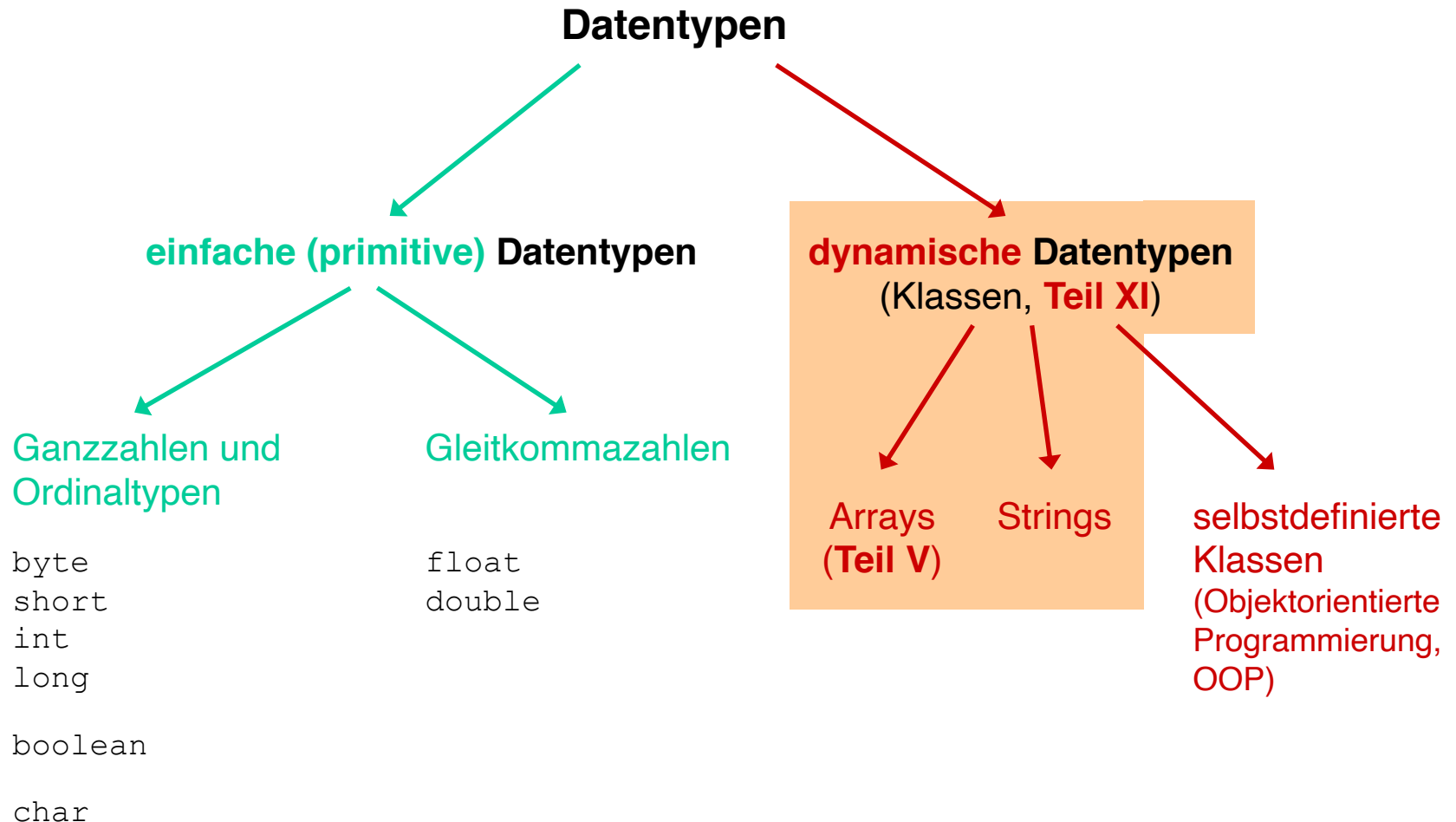
Containermodell: Die Größe hat einen **Typ** und erhält einen **Namen** (wie bei Variablen); jedoch wird der Container nach Zuweisung des Wertes verschlossen

3. Primitive Datentypen

- Taxonomie der Datentypen und Eigenschaften
- Ganze Zahlen (**integer**)
- Negative Ganzzahlen im Binärsystem
- Gleitkomma-Zahlen (**float**, **double**)
- Wahrheitswerte (**boolean**)
- Präzedenzregeln für Operatoren
- Zeichen (**char**)
- Zeichenketten (**String**)

Taxonomie der Datentypen und Eigenschaften

Taxonomie der Datentypen



Eigenschaften primitiver Datentypen

Primitive Datentypen

... besitzen einen **endlichen Wertebereich**

... besitzen eine **festgelegte Anzahl von Werten**

Konsequenzen

- Damit besitzen sie eine **definierte Unter- und Obergrenze** (bei Zahlenbereichen)

Bsp.: `int`: $-2.147.483.648 \leq x \leq 2.147.483.647$

- (numerische) Variablen besitzen nur eine **beschränkte Genauigkeit**

→ später dazu mehr

Primitive Datentypen in Java:

```
void
boolean
char
byte
short
int
long
float
double
(String)
```

Bemerkungen

- Neben den primitiven Datentypen bietet Java auch die Möglichkeit, **eigene Datentypen** zu definieren
- In Wirklichkeit ist der **Datentyp `String`** kein primitiver Typ (Details dazu später)

Ganze Zahlen (integer)

Zahlen mit positivem und negativem Wertebereich

- Wertebereiche für ganze Zahlen
 - `byte` [-128 ... 127]
 - `short` [-32.768 ... 32.767]
 - `int` [-2.147.483.648 ... 2.147.483.647]
 - `long` [-2^{63} ... $2^{63}-1$]
- Fragen dazu ...
 - *Warum 127, 32767, ... und nicht 100, 10000, ... oder 99, 9999, ...?*
 - *Wofür braucht man überhaupt unterschiedliche Typen von ganzen Zahlen?*

Primitive Datentypen in Java:

`void`
`boolean`
`char`

`byte`
`short`
`int`
`long`

`float`
`double`
`(String)`

Rechnen mit ganzen Zahlen

■ **Arithmetische Operationen** für ganze Zahlen

- Addition: $+$
- Subtraktion: $-$
- Multiplikation: $*$
- Ganzzahlige Division: $/$
- Modulo: $\%$ (Division mit Rest)

Bsp. zur Rechnung mit Division / Modulo:

```
13 / 5;    liefert Ergebnis: 2
13 % 5;    liefert Ergebnis: 3
```

■ Präzedenz-Regeln:

Es gelten die üblichen arithmetischen **Regeln**

- Punkt vor Strich und
- ansonsten von links nach rechts,
d.h. $*$, $/$, $\%$ haben Vorrang vor $+$, $-$

Primitive Datentypen in Java:

```
void
boolean
char
byte
short
int
long
float
double
(String)
```


Arithmetische Operationen und Wertebereiche

- **Vorsicht:** Arithmetische Operationen dürfen den **Wertebereich** des jeweiligen Datentyps nicht überschreiten

Konsequenzen: Bei einer Überschreitung kommt es je nach Programmiersprache

- zum Programmabbruch (Überlauf) oder
- die Variable erhält einen anderen (**falschen**) **Wert** (Java)

```
int o = 10000000000 * 4;
int p = (10000000000 * 4) / 10;
int q = (10000000000 / 10) * 4;
```

Erzeugt Ausgabe: o = -294967296
p = -29496729
q = 400000000

Primitive Datentypen in Java:

void
boolean
char
byte
short
int
long
float
double
(String)

- **Folgerung:** Datentyp entsprechend wählen und Formel eventuell umformen (Zwischenergebnisse klein halten)!

Oops, was ist hier passiert ...?

Gleitkomma-Zahlen (float, double)

Format und Eigenschaften

- Näherungsweise Darstellung der rationalen Zahlen
- **Wertebereiche** für Gleitkomma-Zahlen
 - **float** [$-3.4028235 \times 10^{38}$; 3.4028235×10^{38}]
 - **double** [$-1.797693 \times 10^{308}$; 1.797693×10^{308}]
- Zahlen, die einen '.' enthalten, werden von Java automatisch als **double** interpretiert

Bsp.: -1.5 100.

Formatangaben:

- Eine Zahl kann als **float** gekennzeichnet werden, wenn ihr ein 'f' ('F') angehängt wird

Bsp.: 1f 1.e12f 3.14F

- Einer Zahl kann ein 'd' ('D') angehängt werden, um sie als **double** zu kennzeichnen

Bsp.: 5d 17.208E-3d 2.D

Primitive Datentypen in Java:

```
void
boolean
char
byte
short
int
long
float
double
(String)
```

Spezielle Werte

- Positiv / negativ Unendlich:

`Float.NEGATIVE_INFINITY`

`Float.POSITIVE_INFINITY`

- Not-a-Number

`Float.NaN`

Bsp.: Division durch Null

```
float fval1 = 10.0;
float fval2 = 0.0;
float fval3 = 0.0;
float fres1, fres2;
```

```
fres1 = fval1 / fval2;
fres2 = fval2 / fval3;
```

Ausgabe: Infinity

Ausgabe: NaN

Demo: [TestNumbersArithmetic.java](#)

Primitive Datentypen in Java:

void
boolean
char
byte
short
int
long
float
double
(String)

Vorsicht bei Vergleichen:
Es gilt immer NaN ungleich NaN

- Hinweis:** Verwende bei Vergleichen für *not-a-number* Resultate die Funktionen (Methoden) `Float.isNaN(...)` bzw. `Double.isNaN(...)`

Wahrheitswerte (boolean)

Boolesche Werte – Wahr oder Falsch

- Die meisten Programmiersprachen (z.B. Java) bieten einen speziellen Datentyp `boolean` zur Darstellung von Wahrheitswerten
- Wertebereich für Boolean
`true` und `false`
- Logische Operatoren
 - Konjunktion `& &`
 - Disjunktion `| |`
 - Negation `!`

Primitive Datentypen in Java:

```
void
boolean
char
byte
short
int
long
float
double
(String)
```

Hinweis: Mit logischen Operatoren werden boolesche Ausdrücke gebildet

Vergleichsoperationen

- **Vergleichsoperationen auf Zahlen** haben Wahrheitswerte zum Ergebnis

Bsp.:

```
int x = 5,
    y = 6;    // x, y haben bestimmte Werte

...(x < y)...  // in einer Anweisung koennen die Werte
               // der Variablen verglichen werden
```

- Menge von **Vergleichsoperatoren**

- == Symbol für Gleichheit
- != Symbol für Ungleichheit
- < kleiner
- > größer
- <= kleiner oder gleich
- >= größer oder gleich

```
float x = 5.0f,
      y = 18.0f;
int z = 7;
boolean b, c, d, e;

b = (x > y);
c = (x < z);
d = (x != y);
e = (x == y);
```

Hinweis: Vergleiche liefern Wahrheitswerte als Ergebnisse; sie können damit Teil eines booleschen Ausdrucks sein und auch Variablen vom Wahrheitstyp zugewiesen werden

Boolesche Ausdrücke

Struktur

- Logische Ausdrücke sind **Ausdrücke vom Typ `boolean`**
- Boolesche Ausdrücke werden gebildet mithilfe von
 - **Vergleichsoperationen** auf arithmetischen Ausdrücken
 - **Logischen Operationen** auf booleschen Ausdrücken
- Boolesche Ausdrücke können mithilfe bestimmter Regeln **umgeformt** bzw. **vereinfacht** werden (→ DeMorgan'sche Gesetze)

```
(a && b) || (a && c)  ⇔  a && (b || c)
!a || !b             ⇔  !(a && b)
```

```
float    x = 5.0f,
          y = 18.0f;
int      z = 7;
boolean  a, b;

a = true;
b = (x > y) &&
    (x < z) ||
    (x == y) ||
    a;
```

Einordnung: Boolesche Algebra
(→ **Formale Grundlagen der Informatik**)

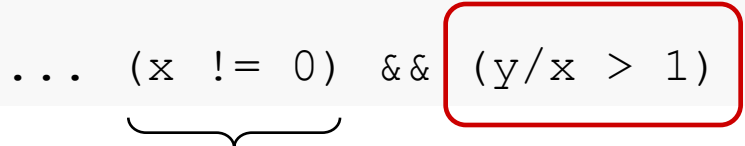
**b ist hier immer `true`,
da `a = true`**

Auswertung boolescher Ausdrücke

- Boolesche Ausdrücke liefern Wahrheitswerte als Ergebnis
- Die Ausdrücke können sich aus mehreren Operanden zusammen setzen
 - Die Auswertung der Operanden erfolgt sequenziell von links nach rechts
 - Die Auswertung wird ggf. abgebrochen, wenn das Ergebnis des logischen Ausdrucks fest steht und sich somit nicht mehr ändern kann (die logischen Operatoren `&&` und `||` heißen *short-circuited*)

Bsp.: `int x = 0;`

```
... (x != 0) && (y/x > 1)
```



Das Resultat ist **false**

Daraus folgt: `(x != 0) && irgendwas` ergibt immer **false** (!) und somit wird der Operand `(y/x > 1)` nicht mehr ausgewertet

Vorteil: Division durch Null (`y/0`) wird verhindert!

Wichtiger Hinweis: Das ist nicht bei allen Programmiersprachen so!

Beispiel – Zinsberechnung (Demo: [Interest.java](#))

```

1  class Interest {
2      /**
3       * Das Programm führt eine einfache Zinsberechnung durch und gibt den neuen Stand
4       * des Kapitals (Grundkapital + Zins) aus. Es werden vorgestellt:
5       * - Variablendeklarationen für Standardtypen
6       * - Variablen vom Typ String
7       * - Zuweisungen (mit Überschreibung eines alten Wertes)
8       * Autor: David J. Eck (mit Erweiterungen hn, Aug.2010)
9       */
10     public static void main(String[] args) {
11         double principal; // Wert der Einlage (Grundkapital)
12         double rate;      // jaehrliche Zinsrate
13         double interest;  // Zinsbetrag pro Jahr
14         String outText1, outText2;

15
16         outText1 = "Der Zinswert (bei 4%) ist (in EUR) ";
17         outText2 = "Der Wert der Einlage nach 1 Jahr ist (in EUR) ";
18
19         principal = 17000;
20         rate      = 0.04;
21         interest  = principal * rate; // berechne den Zins
22
23         principal = principal + interest; // berechne den neuen Wert des Kapitals nach
24                                         // 1 Jahr, mit dem hinzugerechneten Zins
25                                         // Hinweis: neuer Wert ersetzt alten Wert von principal!
26
27         /**
28          * -- Ausgabe ...
29          */
30         //System.out.print("Der Zinswert (bei 4%) ist (in EUR) ");
31         System.out.print(outText1);
32         System.out.println(interest);
33         //System.out.print("Der Wert des Kapitals nach 1 Jahr ist (in EUR) ");
34         System.out.print(outText2);
35         System.out.println(principal);
36     } // end main
37 } // end class Interest

```


Präzedenzregeln für Operatoren

Übersicht

- **Präzedenz-Regeln** legen die **Reihenfolge** fest, in der die **Operatoren eines Ausdrucks ausgewertet** werden
- In Java haben die Operatoren Vorrang in absteigender Reihenfolge:
 1. Einstellige Operatoren: positives (+) / negatives (–) Vorzeichen, Negation (!)
 2. Multiplikative Operatoren: *, /, %
 3. Additive Operatoren: +, –
 4. Relationale Operatoren: <, >, <=, >=
 5. Gleichheitsoperatoren: ==, !=
 6. Konjunktion: & &
 7. Disjunktion: | |
- Innerhalb einer Gruppe werden die Operationen von **links nach rechts** ausgewertet

Beispiel – Auswertung eines Ausdrucks

```
System.out.println(-1 + 20 / 4 < 7 && 9 > 6);
```

Zeichen (char)

Der Datentyp

- Der Datentyp `char` speichert jeweils ein **einzelnes Zeichen**
- Der **Wertebereich** von `char` umfasst
 - Groß- und Kleinbuchstaben,
 - Ziffern,
 - Satzzeichen sowie
 - Sonder- und Steuerzeichen

Hinweis: Java verwendet zur Repräsentation von Zeichen den **Unicode**, welcher 65536 Zeichen umfasst und es damit erlaubt, alle Sprachen der Welt zu schreiben

(→ Unicode Konsortium zur Entwicklung eines Standards, <http://www.unicode.org>)

Primitive Datentypen in Java:

```
void
boolean
char
byte
short
int
long
float
double
(String)
```

Exkurs – Zeichen-Codierung

Interne Codierung von Zeichen

... vergleichbar dem früher viel verwendeten Morse-Code • • • _ _ _ • • •

- Computer kennen eigentlich nur Zahlen

Zeichen werden durch Zahlen kodiert, d.h. **jedem Zeichen** wird eine **eindeutige Zahl zugeordnet**, die durch einen bestimmten Code festgelegt ist

- Der **Unicode** ordnet jedem Zeichen einen bestimmten Wert einer **16 Bit Zahl** zu ($2^{16} = 65536$)

Bsp.:

'0'	=	48;	'1'	=	49;	'2'	=	50;	...	;	'9'	=	57
'A'	=	65;	'B'	=	66;	'C'	=	67;	...	;	'Z'	=	90
'a'	=	97;	'b'	=	98;	'c'	=	99;	...	;	'z'	=	122
'@'	=	64;	'%'	=	37;	'Σ'	=	8721;					

- Bis heute wird auch noch häufig der **ASCII Code** verwendet, der die lateinischen Zeichen sowie einige Sonderzeichen durch eine 7 Bit Zahl codiert

Hinweis: Für im ASCII Code darstellbare Zeichen ist deren Codierung identisch zur Codierung im Unicode (UTF8)

Notation von Zeichen

- Zeichen werden in einfachen **Apostrophen** (**Hochkomma**) geschrieben

Bsp.: 'a', 'D', '!', '7', '+'

- Steuerzeichen** werden durch einen \ eingeleitet

Bsp.: '\n' (Zeilenumbruch),
'\t' (Tabulator)

- „**Hilfszeichen**“ werden ebenfalls durch einen \ eingeleitet bzw. ausgedrückt

Bsp.: '\ ' (Apostroph),
'\"' (Hochkomma),
'\\' (*back-slash*)

Primitive Datentypen in Java:

```
void
boolean
char
byte
short
int
long
float
double
(String)
```

Vergleich zwischen Zeichen

- Jedes Zeichen hat einen **eindeutigen Code** (siehe Unicode), der als **Ordnungszahl** interpretiert werden kann
- **Vergleichsoperatoren** für Zeichen:

==	gleich
!=	ungleich
<	kleiner
>	größer
<=	kleiner oder gleich
>=	größer oder gleich
- Zeichen besitzen eine **(alphabetische) Reihenfolge**, die durch die **zugrunde liegende Repräsentation** (Code, Ordnungszahl) bestimmt wird

Meist gilt:

'0'	<	'1'	<	...	<	'9'	
<	'A'	<	'B'	<	...	<	'Z'
<	'a'	<	'b'	<	...	<	'z'

Primitive Datentypen in Java:

```
void
boolean
char
byte
short
int
long
float
double
(String)
```

Zeichenketten

Klassen und Operationen

- Zur **Speicherung von Zeichenketten** stellt Java eine **Klasse `String`** zur Verfügung;

daher ist `String` kein primitiver Datentyp!

- Java stellt jedoch speziell für **Strings** viele Operationen und Vereinfachungen zur Verfügung, wodurch **Strings** häufig **ähnlich wie primitive Datentypen behandelt** werden können (siehe Liste)

Bsp.: Erzeugen von **Strings** durch Zeichenketten in Anführungszeichen (z.B. "Hallo Welt")

```
String s      = "Hallo ",
      name = "Paul";
```

Konkatenation, d.h. Aneinanderreihung, von Zeichenketten durch **+**

```
String hello = "Hallo " + name;
```

Primitive Datentypen in Java:

```
void
boolean
char
byte
short
int
long
float
double
(String)
```

Operationen auf Strings

Format

- Die Klasse `String` stellt **Funktionen** (Methoden) zur Verfügung, mit denen Operationen auf Variablen vom Typ `String` ausgeführt werden können
- Aufruf** von Funktionen auf Objekte vom Typ `String`:

```
<string-Objekt>.<Funktions-Name>(<Parameter>)
```

Erläuterung: `'.'`-Operator (**Selektor**) ermöglicht den Zugriff (Aufruf) der Methoden für das Objekt

Einige Funktionen

geg.: Deklaration `String s1, s2;`

- `s1.equals(s2)`: boolesche Funktion, liefert `true` wenn `s1` aus exakt der gleichen Zeichenfolge wie `s2` besteht, sonst `false`
- `s1.equalsIgnoreCase(s2)`: boolesche Funktion, die überprüft ob `s1` die gleiche Zeichenkette wie `s2` ist – Groß- und Kleinbuchstaben werden als gleichwertig betrachtet
- `s1.length()`: liefert Ergebnis vom Typ `int` mit der Anzahl der Zeichen in `s1`

Primitive Datentypen in Java:

```
void
boolean
char
byte
short
int
long
float
double
(String)
```


- **`s1.charAt(k)`**: liefert Ergebnis vom Typ **`char`**, das dem Zeichen an der *k*-ten Position von *s1* entspricht (*k* vom Typ **`int`**)
- **`s1.substring(j, k)`**: liefert Ergebnis (*substring*) vom Typ **`String`** mit den Zeichen an den Positionen *j*, *j*+1, *j*+2, ..., *k*-1 (*j*, *k* vom Typ **`int`**); Hinweis: Zeichen an Position *k* wird nicht gelesen
- **`s1.indexOf(s2)`**: Wenn *s2* ein *Substring* von *s1* ist, dann liefert die Funktion die Position (Index, vom Typ **`int`**) mit der Startposition des *Substrings* in *s1*, sonst -1
 - `s1.indexOf(ch)`: Suche nach Zeichen *ch* in *s1*
 - `s1.indexOf(ch, k)`: Suche nach 1. Auftreten von *ch* ab Position *k* in *s1*
- **`s1.toUpperCase()`**,
`s1.toLowerCase()`: liefert Ergebnis vom Typ **`String`** wie *s1* in dem Zeichen konvertiert wurden, die nicht der Zeichenform des Ziels entsprechen
- **`s1.trim()`**: liefert Ergebnis vom Typ **`String`** mit einer neuen Zeichenkette, in der aus *s1* alle führenden und abschließenden leer-druckenden Zeichen, z.B. Leerzeichen, Tabulatoren, entfernt wurden

Hinweis: Durch die Funktionen `s1.toUpperCase()`, `s1.toLowerCase()`, `s1.trim()` wird der **`String`** *s1* nicht verändert, sondern ein neuer **`String`** generiert und zurück gegeben; die Funktionen sollten daher immer als Argument in einem Ausdruck, z.B. Zuweisung, verwendet werden

- **+** (Plus-Operator): Konkatenation von Zeichenketten (siehe oben)

Bsp.: **`String`** name = "Frank Kargl";

System.out.println("Hallo " + name +

". Schoen, dass Sie wieder da sind!")

Besonderheiten

- **Vorsicht** beim **Vergleichen von Strings**

Für den Vergleich von Strings sollte immer die Methode `String.equals(String)` verwendet werden!

Strings sollten niemals mit `==` oder `!=` verglichen werden

- Beispiele:

```
String a = "text",
      b = "text";
String c = new String("text"); // generiert
                               // ein neues Objekt vom Typ String

System.out.println(a == b);
                               liefert Ausgabe: true
System.out.println(a.equals(b));
                               liefert Ausgabe: true
System.out.println(a == c);
                               liefert Ausgabe: false
System.out.println(a.equals(c));
                               liefert Ausgabe: true
```

Primitive Datentypen in Java:

```
void
boolean
char
byte
short
int
long
float
double
(String)
```

Erklärungen zu `System.out.println()` später!

4. Datentypen und Programmstruktur

- Datentyp `void` und strenge Datentypisierung
- Aufzählungstypen (Enumerationen)
- Programmanweisungen, Blöcke und Zuweisungen

Datentyp `void` und strenge Datentypisierung

Verwendung

- `void` („das Nichts“) ist ein spezieller Datentyp, der **keinen Wertebereich** besitzt
- `void` wird **ausschließlich bei der Deklaration von Funktionen (Methoden)** verwendet, um festzulegen, dass die Funktion (Methode) **keinen Rückgabewert** besitzt

Bsp.:

```
public static void main(String[] args)
```

Statische und von allen Programmteilen sichtbare Funktion/Methode (Modifizierer, siehe **Teil X**, Objektorientierung)

Die (spezielle) Funktion/Methode `main` liefern **keinen Ergebniswert**

Funktion/Methode `main` mit ihren spezifischen Parametern (in Klammern)

Primitive Datentypen in Java:

```
void
boolean
char
byte
short
int
long
float
double
(String)
```

Strenges Typsystem

- In Java besitzt **jede Variable** einen **eindeutigen Datentyp**
 - Festlegung, welche Operationen auf die Variable anwendbar sind
 - Operatoren dürfen nur **typkompatible** (zueinander passende) Operanden miteinander verknüpfen (wird schon zu Übersetzungszeit (Compilation) überprüft)

Bsp.:

```
double x;
```

```
x = 3.5d;
```

OK

```
x = true;
```

ergibt Fehler (bei Compilation)!

```
x = x * 1.5d;
```

OK

```
x = x * true;
```

ergibt Fehler (bei Compilation)!

- **Bemerkungen:**

- Strenge Typisierung hilft **Programmierfehler** zu **vermeiden**!
- Datentypen lassen sich in andere **Typen umwandeln** (siehe nachfolgende Seiten)

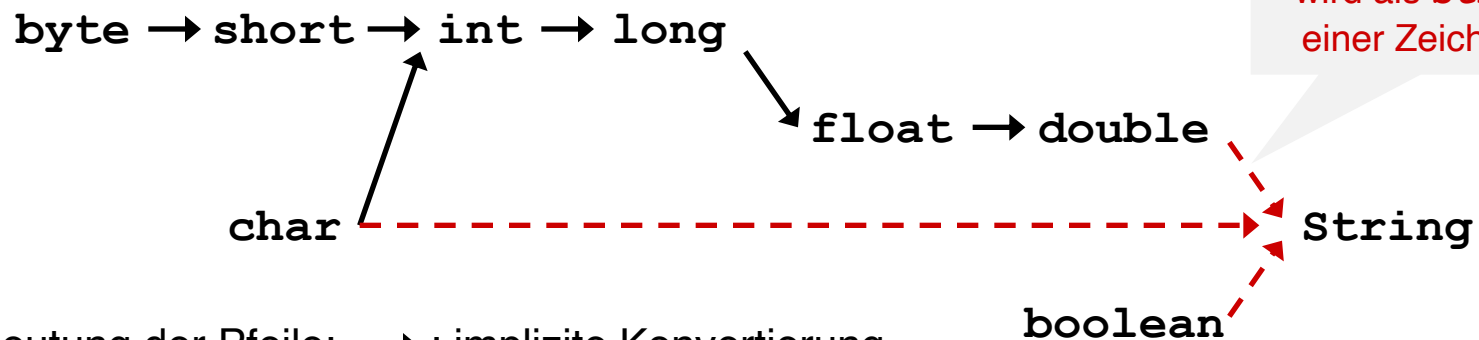
(Implizite) Typkonvertierung

Grundlegendes

- Operatoren lassen sich nicht nur auf Operanden des selben Datentyps anwenden
 - Werden typkompatible Operanden durch einen Operator verknüpft, so findet implizit eine Datentyp-Anpassung der Operanden statt
 - Typkonversionen erfolgen immer in Richtung auf einen allgemeineren Datentyp hin

Wichtig: Im Allgemeinen geht bei der Anpassung keine Information verloren!

- Konvertierungsschema



Bedeutung der Pfeile: \longrightarrow : implizite Konvertierung,
 $- \longrightarrow$: implizite Konvertierung – Ergebnis hat eine neue Bedeutung

Besonderheiten

- Hinweis: Eine Typkonversion zu **String** erzeugt eine **neue Codierung** für die primitiven Datentypen der Eingabe, auf denen die ursprünglichen Operationen dann so nicht mehr anwendbar sind (später dazu mehr)
- Die **Reihenfolge der Operatoren** muss auch hier beachtet werden

Bsp.:

```
System.out.println(17 + " und " + 4);
```

liefert Ausgabe: 17 und 4

Argumente für print-Kommando werden

- konvertiert (17 → **String**, 4 → **String**) und
- an einander gehängt

```
System.out.println(17 + 4 + " und");
```

liefert Ausgabe: 21 und

Argumente für print-Kommando werden

- berechnet (arithmetische Operation: 17 + 4),
- konvertiert (Ergebnis = 21 → **String**) und
- an einander gehängt

(Explizite) Typkonvertierung

- Manchmal sind auch **Typkonvertierungen** notwendig, die mit einem **Genauigkeitsverlust** oder einer **Bedeutungsänderung** einhergehen
 - Eine solche Umwandlung ist **(in Java)** i. a. **nicht implizit möglich**
 - Sie kann **vom Programmierer** jedoch **explizit** vorgenommen werden
 - Hierzu wird der gewünschte Datentyp in Klammern dem Ausdruck vorangestellt

Bsp.:

```
double a = 65.2d;  
int i = (int) (a / 3.0); liefert Ergebnis: 21  
char x = (char) (int) a; liefert Ergebnis: 'A'
```

Hinweis: Die explizite Typkonvertierung wird auch *casting* genannt

- **Wichtig:** Bei expliziten Typumwandlungen ist **Vorsicht** geboten; diese sollten nur sehr sparsam verwendet und wenn möglich ganz vermieden werden

Aufzählungstypen (Enumerationen)

Datentypen

- **Vorbemerkungen:** Eine grundlegende Eigenschaft von Java – und vielen anderen höheren Programmiersprachen – ist die Möglichkeit der **Definition neuer Datentypen** und Klassen; seit Java 5.0: Möglichkeit der Definition von **enums** (kurz für *enumerated types*)
- Eine Aufzählung **enum** ist ein Typ mit einer **festgelegten Liste möglicher Werte**

```
enum <enum-Typ-Name> {<Werte-Liste>}
```

Erläuterungen:

- **Wichtig:** **Definition** kann nicht innerhalb eines Unterprogramms erfolgen; sie **muss** (auch) außerhalb der `main()` Routine platziert werden
- **<enum-Typ-Name>** kann ein beliebiger **Bezeichner** sein; Listenelemente werden durch Komma `,` getrennt
- Jedes Element der **<Werte-Liste>** muss ein Bezeichner sein

Bsp.: **enum** Season {SPRING, SUMMER, FALL, WINTER}

Konvention: Werte der Aufzählung in **Groß-Buchstaben** geschrieben

Definition eines neuen Aufzählungs-Typs

- **enum** ist technisch gesehen eine **Klasse**, die **Werte einer enum-Variable** sind technisch gesehen **Objekte**
- Als Objekte können sie **Unterprogramme** oder **Funktionen** enthalten, die auf Objekten des Typs **enum** ausgeführt werden können (ähnlich wie bei **Strings**)
- Bei der **Übersetzung** mit

```
javac <Klassen-Name>.java
```

werden neben der **ausführbaren Datei**

```
<Klassen-Name>.class
```

für jeden deklarierten **enum-Typ** eigene **Objektdateien**

```
<Klassen-Name>$<enum-Typ-Name>.class
```

erzeugt!

Beispiel – Sternzeichen und Wochentage (Demo: [EnumDemo.java](#))

```

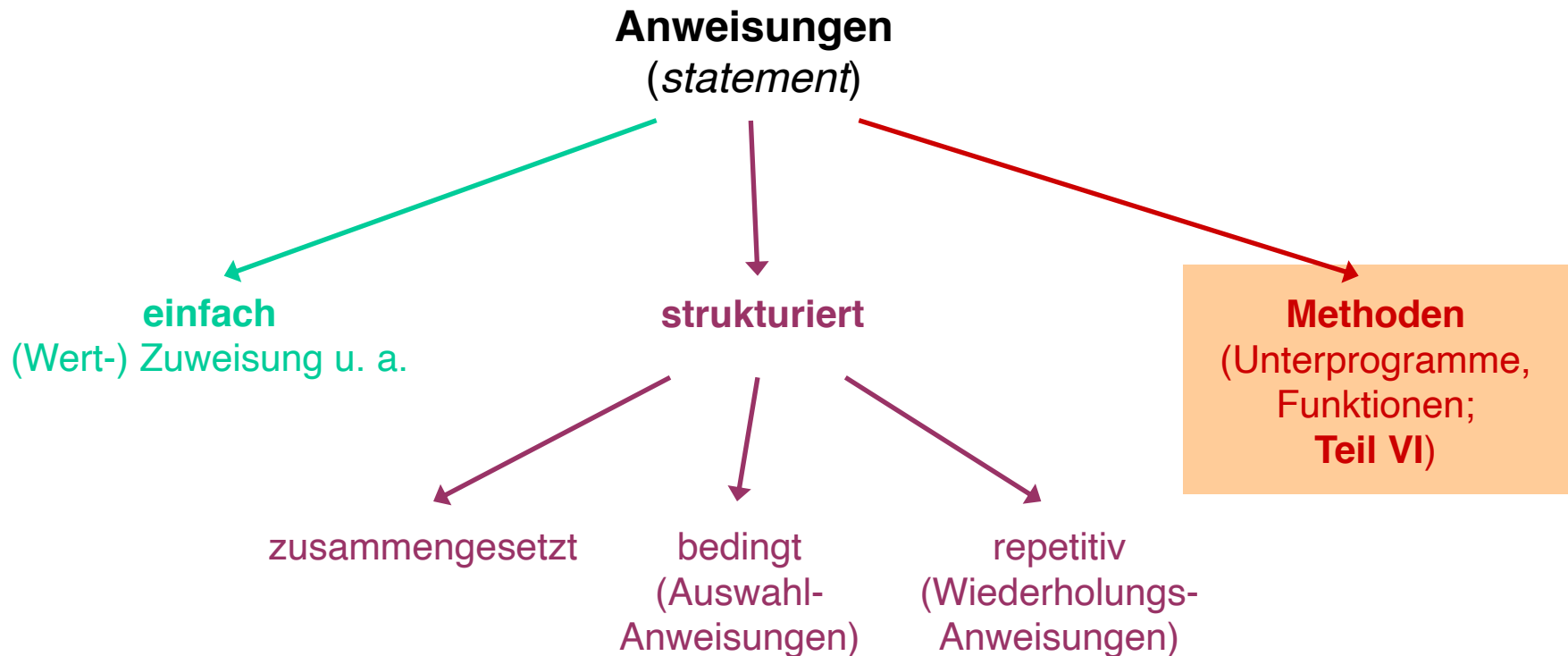
1 public class EnumDemo {
2     /**
3      * Definiere 2 enum types - die Definitionen sind ausserhalb von main() sichtbar.
4      * Es werden vorgestellt:
5      * - Definition neuer Typen (am Beispiel von Aufzählungen, enumerations)
6      *   (erzeugt zusätzlich neue Klassen!)
7      * - Deklaration von Variablen neuer Typen und Verwendung von subroutines (methods)
8      *   auf diesen Variablen
9      * Autor: David J. Eck (mit kleinen Erweiterungen hn, Aug.2010)
10    */
11    enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY }
12
13    enum Month { JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC }
14
15    public static void main(String[] args) {
16        Day day; // deklariere Variable vom Typ Day
17        Month libra; // deklariere Variable vom Typ Month
18
19        day = Day.FRIDAY;
20        libra = Month.OCT;
21
22        /*
23         * -- zeige die ausgesuchten Werte ...
24         */
25        System.out.println("Das Sternzeichen ist 'Libra', wenn jemand im Monat " +
26                           libra + " geboren ist!");
27        System.out.println("Dies ist der " + libra.ordinal() + "+1. Monat des Jahres " +
28                           "(wir zaehlen von 0 an ...)");
29
30        System.out.println("Prima, wenn wir zum " + day + " kommen - Wochenende ...");
31        System.out.println(day + " ist der " + day.ordinal() + ". Tag der Woche.");
32    } // end main
33 } // end public class EnumDemo

```

Programmanweisungen, Blöcke und Zuweisungen

Imperative Programmierung

- Ein Programm manipuliert Daten mittels **Anweisungen** (*statements*)
- Die Manipulationen führen zu **Änderungen des Zustands** eines Prozesses
- **Taxonomie** der Operationsprinzipien:



Anweisungen und Anweisungsfolgen

- Programme sind Folgen von **Vereinbarungen** und **Anweisungen**

- Bei der Ausführung eines Programms werden diese Folgen **sequentiell abgearbeitet**
- Die Ausführung der Anweisungen führen zur schrittweisen Veränderung der (Programm-) Zustands; man spricht von **imperativer Programmierung**

- Jede Anweisung muss durch ein **Semikolon ';'** abgeschlossen werden

Ausnahme: Wenn die Anweisung mit einer geschweiften Klammer (}) endet, folgt kein ; (schreibt man dennoch ein ';' , so wird dies vom Compiler als **leere Anweisung** interpretiert!)

**Anweisungen in
Java:**

**Zuweisung
Bedingte Anweisung
Bewachte Anweisung
Schleife**

Gruppen von Anweisungen – Blöcke

- Anweisungen können mithilfe von `{ }` „gruppiert“ werden (zusammengesetzte Anweisung)
- Eine solche „Gruppe“ stellt dann selbst wieder eine **Anweisung** dar (mit `{ ... }` wird ein **Block** definiert)

Bsp.:

```
public static void main(String[] args) {
    double d = 5.0;
    float f = 3.7f;
    {
        int i = (int) (d*f);
        i = i + 7;
    }
}
```

Anweisungen in Java:

Zuweisung

Bedingte Anweisung

Bewachte Anweisung

Schleife

Lebenszeit und Sichtbarkeit

Hinweise:

Methode



- Variablen, die in einer solchen Gruppe deklariert werden, sind auch nur innerhalb dieser Gruppe gültig (im Bsp.: die Variable `i` (vom Typ `int`) ist nur im Abschnitt der inneren Klammern `{ }`, d.h. dem inneren **Block**, bekannt)
- Deklarierte Größen (Variable, Konstante) sind nur während der Ausführungszeit des entsprechenden Blocks gültig
- Innerhalb des neuen Blocks dürfen Größen der umschließenden Blöcke bis zur Ebene der Methoden (hier `main`) verwendet werden – man darf jedoch Größen nicht mit demselben Namen neu deklarieren (das gilt unabhängig vom Datentyp)

Zuweisungen und Ausdrücke

Struktur

- Eine Zuweisung besteht aus
 - dem Namen einer **Variable**,
 - einem **Gleichheitszeichen**,
 - einem **Ausdruck** und
 - einem abschließenden **Semikolon**

```
variable = <Ausdruck>;
```

Konsequenz: Die Zuweisung speichert den aus dem Ausdruck (rechte Seite) errechneten Wert in der Variablen (linke Seite)

- Es ist möglich, eine **Variablendeklaration mit einer Zuweisung** zu kombinieren

```
<Datentyp> variable = <Ausdruck>;
```

**Anweisungen in
Java:**

Zuweisung

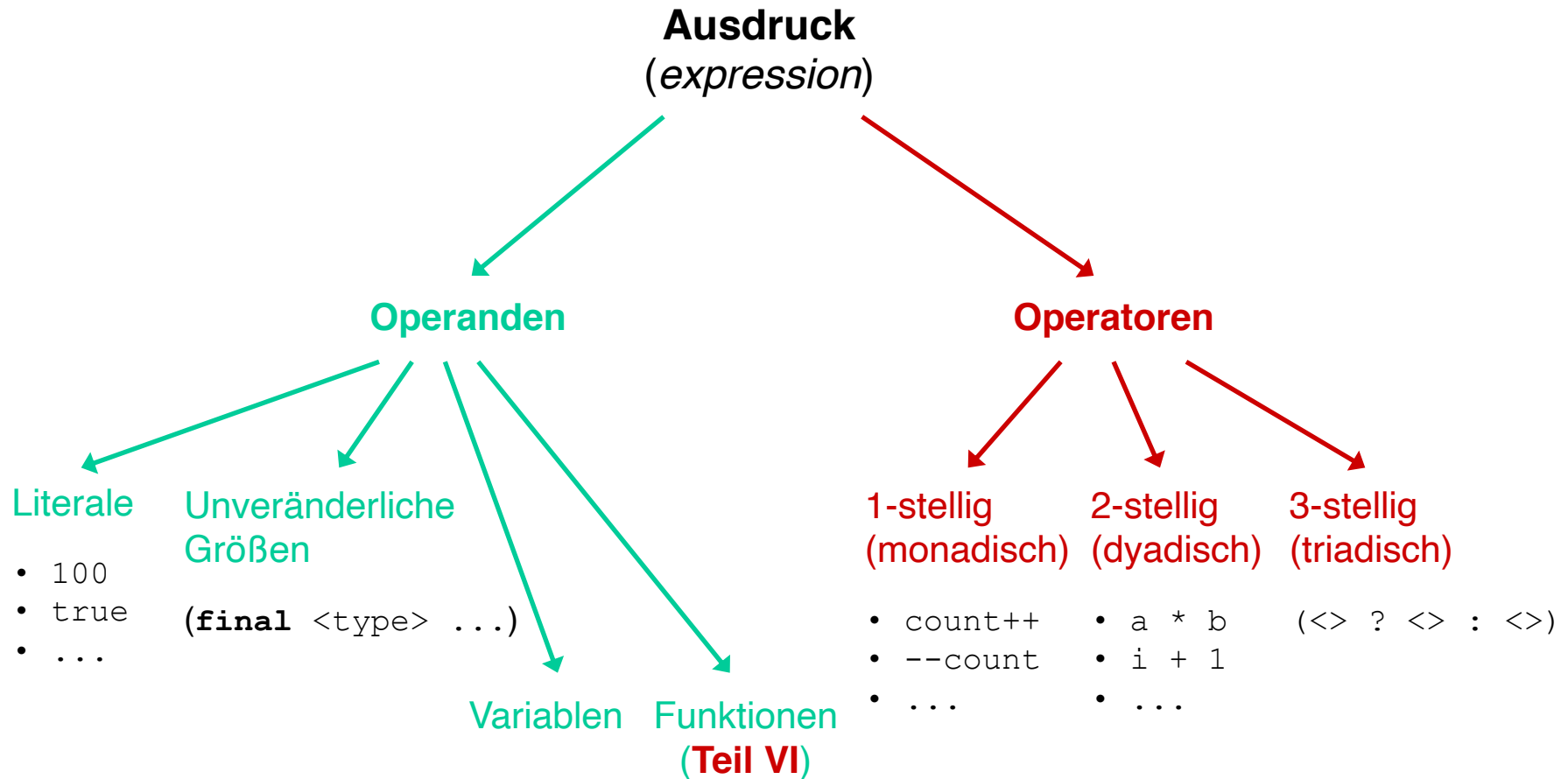
Bedingte Anweisung

Bewachte Anweisung

Schleife

Ausdrücke

- Ausdrücke sind Kompositionen von **Operanden** und **Operatoren**, entsprechend mathematischer Gleichungen
- **Taxonomie** gültiger Ausdrücke



- Innerhalb eines **Ausdrucks** kann der **Wert einer Variable** durch Angabe des Variablen-Namens verwendet werden

Achtung: Die Verwendung von Variablennamen auf der linken und der rechten Seite einer Zuweisung hat unterschiedliche Bedeutung

Bsp.: `int a, b;`

```
a = 5 * 3;
b = (a + 7) % 12;
```

Zuweisung (schreiben)

Zugriff auf Inhalt der Variablen (lesen)

- Wenn nötig und möglich, wird bei Zuweisungen eine **implizite Typkonvertierung** durchgeführt

Bsp.: `float x = 3;`
`int y = 'a';`

**Anweisungen in
Java:**

Zuweisung
Bedingte Anweisung
Bewachte Anweisung
Schleife

Zeitlicher Ablauf einer Zuweisung

1. Die **linke Seite** der Zuweisung wird ausgewertet (kann später auch komplexer werden)
2. Die **rechte Seite** (Ausdruck) wird
 - ausgewertet und
 - ihr Wert unter Beachtung der Präzedenzregeln ermittelt
3. Falls der Wert der rechten Seite typkompatibel zur Variable ist oder implizit angepasst werden kann, erfolgt die **Zuweisung** des Wertes an die Variable

Bsp.: `a = 5 * 3;`
`b = (a + 7) % 12;`

Hinweis: Aufgrund der sequenziellen Analyse und Auswertung kann die **Variable** selbst auch auf der rechten Seite auftreten (Verwendung ihres alten Wertes), bevor der neue Wert wieder an **dieselbe Variable** zugewiesen wird

Bsp.: `a = 5 * 3;` **Inhalt von a:** 15
`a = (a + 7) % 12;` **Inhalt von a:** $(15+7) \% 12 = 10$

**Anweisungen in
Java:**

Zuweisung
Bedingte Anweisung
Bewachte Anweisung
Schleife

Zuweisungen und Ausdrücke

- Eine Zuweisung ist selbst wieder ein Ausdruck

Bsp.: `x = (y = 7);` **Wirkung: `x` und `y` erhalten beide den Wert 7**

- **Vorsicht bei Vergleichen:**

- `=` (Zuweisung) und
- `==` (logischer Vergleich)

werden leicht verwechselt

Bsp.: `boolean a = false,`
`b = true;`

`boolean c = (a == b);` **liefert Ergebnis: `false`**
`boolean d = (a = b);` **liefert Ergebnis: `true`**

**Anweisungen in
Java:**

Zuweisung

Bedingte Anweisung

Bewachte Anweisung

Schleife

Kurzformen

- Für einige häufig vorkommende Zuweisungen gibt es **abgekürzte Schreibweisen**

`i += a;` entspricht
`i = i + a;`

- Weiterhin gelten die folgenden **Kurzschreibweisen**

- `i -= a;` entspricht: `i = i - a`
- `i *= a;` entspricht: `i = i * a`
- `i /= a;` entspricht: `i = i / a`
- `i %= a;` entspricht: `i = i % a`

Wirkung: Der Wert der Variablen (linke Seite) wird vor der Zuweisung mit dem Argument auf der rechten Seite verknüpft (Subtraktion, Multiplikation, etc.), das Ergebnis wird dann anschließend der Variablen wieder zugewiesen

**Anweisungen in
Java:**

Zuweisung

Bedingte Anweisung

**Bewachte Anweisung
Schleife**

Numerische Operatoren für Inkrement und Dekrement

- In Programmen werden häufig **Zählvariablen** verwendet

Hierzu können

- Inkrement- (`i++`) und
- Dekrement- (`i--`) Operatoren

angewendet werden

- Auswertung von Ausdrücken:

- **Inkrement** des Werts der Variable `i`:
 - `i++` nach Verwendung des Wertes von `i`
 - `++i` Inkrement vor Verwendung des Wertes von `i` (entspricht `i += 1`)
- **Dekrement** des Werts der Variable `i`:
 - `i--` nach Verwendung des Wertes von `i`
 - `--i` Dekrement vor Verwendung des Wertes von `i` (entspricht `i -= 1`)

**Anweisungen in
Java:**

Zuweisung

Bedingte Anweisung

**Bewachte Anweisung
Schleife**

Bsp.: `int a = 6, b, c;`

`b = a++ * 3;` **Inhalte der Variablen: `b = 18` und `a = 7`**
`c = ++a / 2;` **Inhalte der Variablen: `a = 8` und `c = 4`**

Hinweis: Die übertriebene Verwendung von `++` und `--` führt leicht zu unübersichtlichen Ausdrücken – und damit zu Fehlern;

Tipp: Verwende `++` / `--` -Operatoren nur in Einzelanweisungen, nicht in Ausdrücken

Präzedenzregeln ergänzt

- Die **Präzedenz-Regeln** der Anwendung von Operatoren bei der Auswertung von Ausdrücken können jetzt noch ergänzt werden:

1. Einstellige (unäre) Operatoren:
 ++, -- (Inkrement, Dekrement),
 positives (+) / negatives (-) Vorzeichen,
 ! (Negation), (<type>) (Typ casting)
2. Multiplikative Operatoren: *, /, %
3. Additive Operatoren: +, -
4. Relationale Operatoren: <, >, <=, >=
5. Gleichheitsoperatoren: ==, !=
6. Konjunktion: &&
7. Disjunktion: ||
8. Zuweisungsoperatoren: =, +=, -=, *=, /=, %=

- Empfehlung:** Verwende Klammern (. . .), um die Ausdrücke und ihre Auswertung zu strukturieren und somit lesbarer zu machen