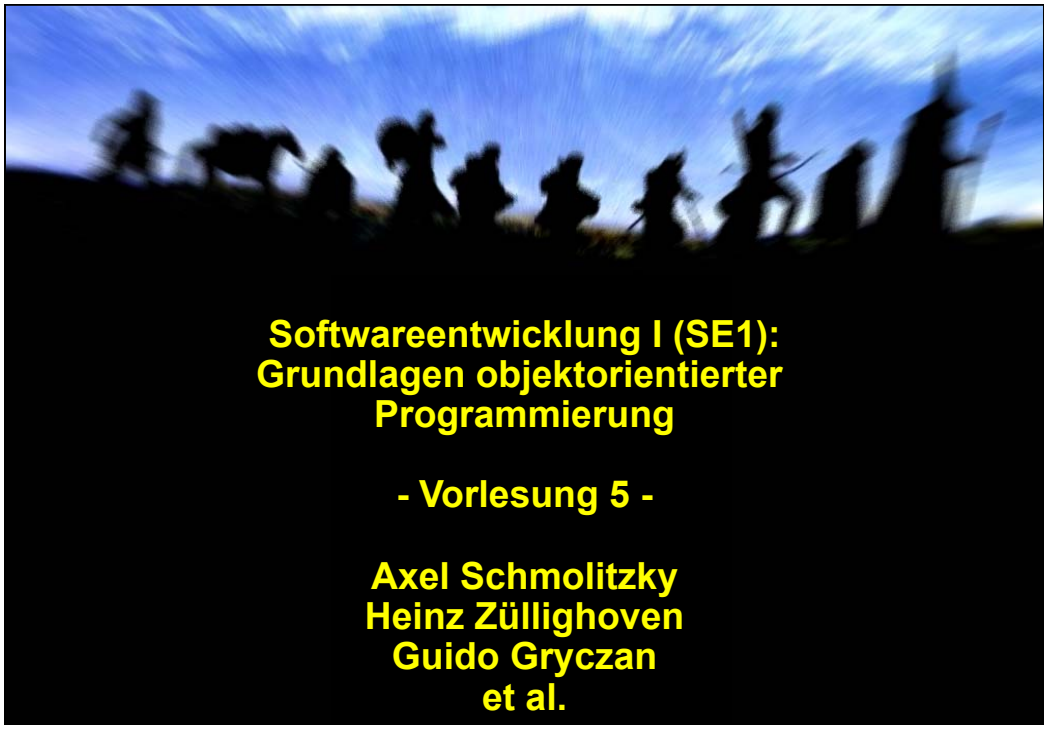
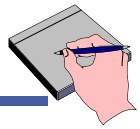


Softwareentwicklung I (SE1): Grundlagen objektorientierter Programmierung

- Vorlesung 5 -

Axel Schmolitzky
Heinz Züllighoven
Guido Gryczan
et al.

Zuweisung (1)



- Bei der **Zuweisung** wird ein **Ausdruck** ausgewertet und sein **Ergebnis** einer **Variablen** zugewiesen.
- **Syntax-Schema der Zuweisung**:

  `Linke-Seite Zuweisungsoperator Rechte-Seite`

  `Rechte-Seite`:
  - imperativ: meist arithmetische und boolesche Ausdrücke, Vergleiche und Zeichen oder Zeichenketten

  `Zuweisungsoperator`:
  - in Java (wie in C/C++) : `'='`
  - auch üblich (Pascal etc): `':='`

  `Linke-Seite`: Bezeichner einer Variablen

Typkompatibilität:

Der Typ der linken Seite muss zum Typ des Zuweisungsausdrucks passen, d.h. zunächst, die Typen müssen gleich sein.

Zuweisung (2)

- Bei der **Zuweisung** sprechen wir oft von der **rechten** und der **linken Seite** einer Zuweisung *(*engl.: right-hand side - RHS, left-hand side - LHS*)* oder von dem **L-Wert** und **R-Wert** *(*engl.: lvalue, rvalue*)*.
- **Bedeutung**:
  - L-Wert:
    Ist ein Bezeichner einer **Variablen**, der ein Speicherplatz zugeordnet ist. Dort wird der neu berechnete Wert gespeichert.
  - R-Wert:
    Ist ein **Ausdruck**, der einen Wert liefert. Ein R-Wert kann nur rechts vom Zuweisungsoperator stehen.
  - Im folgenden Beispiel haben die beiden Auftreten des Bezeichners `a` unterschiedliche Bedeutung:

    ```
    a = a + (3*i);
    ```

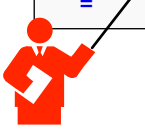
    Auf der linken Seite ist das `a` das Ziel, in dem etwas gespeichert werden soll; auf der rechten Seite ist es die Quelle eines Wertes, der mit anderen Werten in eine Berechnung einfließt.

Merke: Die Zuweisung ist komplizierter, als man auf den ersten Blick vermutet.

Zuweisung in Java

```
antwort = 40
antwort += 2
korrekt = (antwort == 42)
```

Operator	Funktion
`==`	Gleichheit
`!=`	Ungleichheit
`=`	Zuweisung

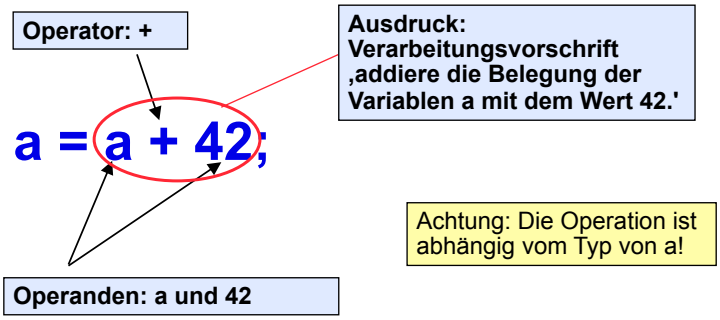


- Der Gleichheitstest wird häufig mit der Zuweisung verwechselt:

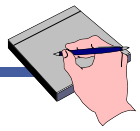
```
saldo =  0   // Zuweisung
saldo == 0   // Gleichheit
saldo != 0   // Ungleichheit
```

Ausdrücke und Operatoren

- In der imperativen Programmierung hat die Zuweisung einen zentralen Stellenwert. In vielen Fällen wird einer Variablen ein Wert dadurch zugewiesen, dass ein **Ausdruck** aus **Operanden** und **Operatoren** ausgewertet wird.
- Üblich sind arithmetische und boolesche Operatoren; in einigen „maschinennahen“ Sprachen kommen Operatoren zur Manipulation von Zeigern und Bit-Repräsentationen hinzu.

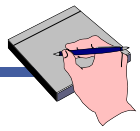


Ausdruck - nach Informatik-Duden



- **Ausdruck** (engl.: expression)
  - Synonym: Term
  - **Verarbeitungsvorschrift**, deren Ausführung einen Wert liefert. Ausdrücke entstehen, indem **Operanden** mit **Operatoren** verknüpft werden. In Programmiersprachen verwendet man häufig arithmetische und logische Ausdrücke.
  - Beispiel:
    Die Symbolfolge `5 * x + 3` ist ein arithmetischer Ausdruck, sofern `x` eine Zahl darstellt.

Operatoren



Als **Operator** bezeichnet man umgangssprachlich sowohl das **Operatorzeichen** (z.B. "+") als auch die damit verbundene **Operation** (z.B. "addieren"). Wir betrachten im folgenden vor allem die Operatorzeichen und ihre Verwendung in (Programm-) Texten.

Die Operatorenschreibweise ist im Zusammenhang mit Programmiersprachen allgemein gebräuchlich (es gibt andere Schreibweisen, z.B. als Funktion).

\+ − * /

SE1 – Level 1 7

Vereinbarungen über Operatoren

Die Operatorenschreibweise ist für uns deshalb einfach lesbar, weil wir bestimmte Vereinbarungen (implizit) kennen, die für die arithmetischen Operatoren gelten.

Bei der Einführung neuer Operatoren müssen diese Vereinbarungen explizit gemacht werden.

Vereinbarungen über Operatoren sind:

- Position,
- Stelligkeit,
- Präzedenz (Vorrangregel),
- Assoziationsreihenfolge.

Dazu kommt die Definition der mit dem Operator verbundenen Operation.

SE1 – Level 1 8

Position von Operatoren

Position, d.h. die Anordnung von Operator und Operanden:

- **Infix**, die häufigste Schreibweise, bei der arithmetische Operatoren zwischen ihren beiden Operanden stehen:
  z.B. : `3 * 4`
- **Präfix**: der Operator steht vor seinen Operanden. Gebräuchlich bei arithmetischen Operationen mit einem Operanden. Diese Form wird auch *Funktionsschreibweise* genannt.
  z.B.: `-2`
- **Postfix**: der Operator steht nach seinen Operanden. Gebräuchlich bei arithmetischen Operationen mit einem Operanden.
  z.B.: `3!` (im Sinne von "Fakultät von 3")

SE1 – Level 1 9

Stelligkeit von Operatoren

Stelligkeit, d.h., Anzahl der Operanden (auch Argumente oder Parameter) eines Operators:

- **einstellig**, oft: unär (engl.: unary)
  z.B. : `3!`
- **zweistellig**, oft: binär (engl.: binary)
  z.B. : `3 * 4`
- **dreistellig**, ternär (engl.: ternary), besser: triadisch.
  In Programmiersprachen kommt meist nur vor
  `if` Operand1 `then` Operand2 `else` Operand3

SE1 – Level 1 10

Präzedenz von Operatoren

Präzedenz (Vorrangregel): bezeichnet die Stärke, mit der ein Operator seine Operanden „bindet“.

- Der Wert eines Ausdrucks hängt oft von der Reihenfolge ab, in der Operatoren eines Ausdrucks angewendet werden.
- Die Präzedenz ist für die arithmetischen Operationen bekannt („Punkt vor Strich“).
- Wenn die Präzedenzen nicht passen, muss geklammert werden:
- Beispiele:

```
3 + 5 * 7 - 3
(3 + 5) * (7 - 3)
```

SE1 - Level 1 11

Assoziativität von Operatoren

Die **Assoziativität** regelt die implizite Klammerung von Ausdrücken bei Operatoren gleicher Präzedenz.

- Beispiel:

```
5 - 4 - 3
```

ist gleichbedeutend mit

```
(5 - 4) - 3
```

Sprechweise:

Der Operator – ist **linksassoziativ**
(er assoziiert von links nach rechts).

- Die Assoziationsreihenfolge ist uninteressant, wenn die Reihenfolge nichts am Wert des Ausdrucks verändert; z.B.:

```
(3 + 4) + 5
```

ist synomym zu

```
3 + (4 + 5)
```

SE1 - Level 1 12

Zentrale Operatoren in Java

Operator	Funktion, arithmetisch
*	Multiplikation
/	Division
%	Modulo
+	Addition
-	Subtraktion
++	Inkrement
--	Dekrement

Sowohl prä- als auch postfix verwendbar

Operator	Funktion, boolesche
!	logisches NICHT
&&	logisches UND
||	logisches ODER
<	„kleiner als“
<=	„kleiner gleich“
>	„größer als“
>=	„größer gleich“
==	Gleichheit
!=	Ungleichheit

Operator	Funktion
=	Zuweisung

SE1 – Level 1 13

Alle Operatoren in Java: Präzedenz, Assoziativität et al.

Postfix-Operatoren:

1 **`()`** **`.`** `[]` `++` `--` links nach rechts

Unäre Operatoren (präfix):

2 `++` `--`

`!` **`-`** `~` `+` *rechts nach links*

3 **`new`** **`(`***Typname***`)`** *rechts nach links*

Binäre* Operatoren (infix):

4 **`*`** **`/`** **`%`** links nach rechts

5 **`+`** **`-`** links nach rechts

6 `<<` `>>` links nach rechts

7 **`<`** **`<=`** **`>`** **`>=`** links nach rechts

8 **`==`** **`!=`** links nach rechts

9 `&` links nach rechts

10 `^` links nach rechts

11 `|` links nach rechts

12 **`&&`** links nach rechts

13 **`||`** links nach rechts

14 `?` `:` *rechts nach links*

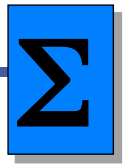
15 **`=`** `+=` `-=` `*=` `/=`
`%=` `&=` `^=` `|=`
`<<=` `>>=` *rechts nach links*

*Ausnahme: ternärer Op.

fett blau hervorgehoben: Relevante Operatoren für SE I

SE1 – Level 1 14

Zwischenergebnis: Zuweisungen et al.



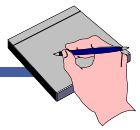
- Die **Anweisungen** in den Rümpfen von Methoden folgen den Prinzipien der **imperativen Programmierung**:
  - Sie werden sequenziell nach der textuellen Reihenfolge im Quelltext ausgeführt.
  - Sie verändern üblicherweise die Belegungen von Variablen.
- Variablen werden durch **Zuweisungen** verändert; wir unterscheiden **Exemplarvariablen**, **formale Parameter** und **lokale Variablen**.
- Auf der **linken Seite** des **Zuweisungsoperators** steht immer eine Variable, auf der **rechten Seite** immer ein **Ausdruck**.
- Arithmetische und boolesche Ausdrücke setzen sich aus **Operanden** und **Operatoren** zusammen.

Erster Kontakt mit dem Typbegriff

- Der Typbegriff spielt eine sehr wichtige Rolle in der Programmierung.
- Ein **Typ** in einer Programmiersprache legt fest
  - eine **Wertemenge** (z.b. bei `int` nur eine Untermenge der ganzen Zahlen) und
  - die **zulässigen Operationen** (z.B. Addieren, Subtrahieren) auf den Werten der Wertemenge.

Typ: `int`
Wertemenge: $\{ -2^{31} \ldots 2^{31}-1 \}$
Operationen: `ganzzahlig Addieren`, `ganzzahlig Subtrahieren`, `ganzzahlig Dividieren`, …

Elementare Typen und Literale



- Imperative und objektorientierte Programmiersprachen bieten i.d.R. einen Satz **elementarer Typen** (engl.: *basic* or *primitive data types*) an:
  - für **ganze Zahlen** → Typ `Integer` o.ä.
  - für **reelle Zahlen** → Typ `Float` oder `Real` (Gleitkommazahlen)
  - für **Zeichen** → Typ `Char` o.ä. (Werte eines bestimmten **Zeichensatzes**)
  - für **Wahrheitswerte** → Typ `Boolean`.
- Die **Werte** dieser elementaren Typen können explizit im Quelltext hingeschrieben werden; jede Programmiersprache bietet zu diesem Zweck sog. **Literale** an. Ein Literal ist eine Zeichenfolge (wie `13` oder `"gelb"`) im Quelltext, die einen Wert eindeutig repräsentiert und deren Struktur dem Compiler bekannt ist.

Überblick: Elementare Datentypen in Java

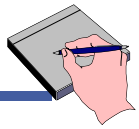
- Auch Java besitzt den üblichen Satz an elementaren Datentypen, die **primitive types** genannt werden. Ungewöhnlich ist die Festlegung der Wortlängen bei den numerischen Typen (jeweils in Klammern in Bit angegeben).
  - **Eine ganze Familie für ganze Zahlen**:
    - » `byte` (8), `short` (16), `int` (32), `long` (64)
  - **Zwei für Gleitkommazahlen**:
    - » `float` (32), `double` (64)
  - **Ein boolescher Datentyp**:
    - » `boolean`
  - **Ein Datentyp für Zeichen**:
    - » `char` (16), **0** bis **65535**

Datentyp	Bit	kleinster Wert	größter Wert
`long`	64	-2^{63}	$2^{63}-1$
`int`	32	-2^{31}	$2^{31}-1$
`short`	16	-32768 (-2^{15})	32767 ($2^{15}-1$)
`byte`	8	-128 (-2^{7})	127 ($2^{7}-1$)

Typ	Standardwert
byte, short int, long (ganze Zahlen)	0 bzw. 0L
boolean (Wahrheitswerte)	false
double, float (Gleitkommazahlen)	0.0 bzw. 0.0f
char (Zeichen)	'\u0000'

Wir geben auf den folgenden Folien nur einen Überblick.
Es ist empfehlenswert, in einem **Java-Referenzbuch** bei Bedarf weitere Details nachzulesen!

Zeichen und ihre Darstellung



- **Zeichen** werden im Rechner durch vordefinierte Werte (die sog. Codes) eines **Zeichensatzes** repräsentiert.
- In Java können wir einzelne Zeichen im Quelltext als Literale in Hochkommata notieren: `'*'`, `'0'`, `'A'`, `'z'`
- Merke: Das Literal `'4'` ist etwas anderes als das Literal `4`. Sie haben verschiedene Typen, im Fall von Java `char` und `int`.

```
char c = '4';
int i = 4;
```

Der ASCII-Zeichensatz

- Die meisten Programmiersprachen vor Java haben Zeichen durch die 128 vordefinierten Werte des sog. **ASCII-Zeichensatzes** dargestellt.
- **ASCII** (Akronym für **American Standard Code for Information Interchange**) ist laut Informatik-Duden:
  - Ein weit verbreiteter, besonders auf Heimcomputern üblicher 7-Bit-Code zur Darstellung von Ziffern, Buchstaben und Sonderzeichen.
  - Jeder ASCII-Codezahl zwischen 0 und 127 entspricht ein Zeichen. Beispiele:
    - ASCII 42 ⇒ *
    - ASCII 48 ⇒ 0
    - ASCII 65 ⇒ A
    - ASCII 122 ⇒ z

in gelb: die druckbaren ASCII-Zeichen

ASCII-Zeichensatz

+	0	1	2	3	4	5	6	7	8	9
30				!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~			

Java und der Unicode-Zeichensatz

- Java war die erste weit verbreitete Programmiersprache, die vollständig auf dem **Unicode Standard UTF-16** aufsetzte, der für jedes Zeichen **16 Bit** verwendet (und somit 65.536 verschiedene Zeichen ermöglicht). Damit lassen sich die Zeichen und Zahlen der meisten bekannten Kultursprachen darstellen.
- Die ersten 128 Zeichen entsprechen dem ASCII-Zeichensatz.
- Inzwischen erlaubt der Unicode-Standard eine Kodierung in bis zu **32 Bit**. Vier Milliarden Zeichen sollten dann für alle irdischen Zwecke ausreichen…
- Zwei Drittel des 16-Bit Unicode-Zeichensatzes werden für chinesische Schriftzeichen verwendet.
- Informationen zu Unicode finden sich im Web unter **http://unicode.org**

UTF – UCS Transformation Format

UCS – Universal Character Set

In Java kann ein Unicode-Zeichen mit einer speziellen Schreibweise notiert werden: `'\uXXXX'`.

`XXXX` ist dabei der vierstellige, hexadezimale Unicode des Zeichens, eventuell mit führenden Nullen.

`'a'` beispielsweise bezeichnet wie `'\u0061'` das a.

Literale für Zahlen in Java

- **Darstellung**:
  - » **Ganze Zahlen** (engl.: integer numbers) können wie gewohnt notiert werden, also etwa `542` oder `-1`; solche Literale werden dann als Dezimalzahlen vom Typ `int` aufgefasst, in diesem Fall mit den Werten **542** und **1** (das `-` ist ein Präfix-Operator, der die **1** hier negiert).
  - » **Gleitkommazahlen** (engl.: floating point numbers) werden in englischer Dezimalnotation mit einem Punkt notiert, z.B. `0.5`, oder mit einem expliziten Exponenten, z.B. `5e-1` (Wert in beiden Fällen **0,5**). Wenn kein `f` für `float` angehängt ist (wie beispielsweise bei `3.1415f`), wird für diese Literale der Typ `double` angenommen.

- Alternativ können ganze Zahlen auch **oktal** (beginnend mit einer **0**) oder **hexadezimal** (beginnend mit **0x**) angegeben werden.
  Bsp.: `29` u. `035` u. `0x1D` u. `0X1d` sind alternative Literale für die Dezimalzahl **29**.

Binäre Operatoren für ganze Zahlen in Java

Arithmetische Operatoren mit Ergebnistyp `int`

- Java bietet die vier Grundrechenarten über die Infix-Operatoren `+`, `-`, `*` und `/` an. Dabei ist zu beachten, dass der Divisionsoperator eine **ganzzahlige Division** durchführt:

  `20/6` ⇒ **3**

  Das Ergebnis dieser Operation ist also wieder ein int-Wert.
- Zusätzlich gibt es den Operator `%`, der bei einer ganzzahligen Division den Rest liefert:

  `20%6` ⇒ **2**
- Die Präzedenz dieser fünf Operatoren entspricht unseren Erwartungen aus der Mathematik („Punktrechnung vor Strichrechnung“):

  `3 + 2 * 2 + 5` ⇒ **12**

Binäre Operatoren für ganze Zahlen in Java (II)

Vergleichsoperatoren mit Ergebnistyp `boolean`

- Für Vergleiche ganzer Zahlen stehen Infix-Operatoren für die Operationen *Größer* (`>`), *Größer-gleich* (`>=`), *Kleiner* (`<`) und *Kleiner-gleich* (`<=`) zur Verfügung.

  `2 > 1 + 1` ⇒ **falsch**
  `2 >= 3 - 1` ⇒ **wahr**
  `2 * 2 < 1 * 1` ⇒ **falsch**
  `2 <= 2 / 1` ⇒ **wahr**

- Zwei weitere Infix-Operatoren erlauben die Abfrage, ob zwei `int`-Ausdrücke gleich (`==`) oder ungleich (`!=`) sind:

  `3 == 2 + 1` ⇒ **wahr**
  `4 != 2 * 2` ⇒ **falsch**
- Vergleichoperatoren haben gegenüber den arithmetischen Operatoren eine niedrigere Präzedenz; sie werden also in einem Ausdruck zuletzt ausgewertet.

Boolesche Literale und Operatoren

- **Boolesche Werte** oder **Wahrheitswerte** sind in Java vom primitiven Typ `boolean`.
- Die **Literale** für die booleschen Werte **wahr** und **falsch** werden als `true` und `false` notiert.
- Die Standardoperatoren der **booleschen Algebra** werden in Java folgendermaßen notiert:
  - logisches Und: `&&`
  - logisches Oder: `||`
  - logische Verneinung: `!`

- Die Java-Operatoren für Gleichheit (`==`) und Ungleichheit (`!=`) sind auch auf boolesche Werte anwendbar. Beispiele:

```
true == false   ⇒   falsch
true != false   ⇒   wahr
```


Die Boolesche Algebra

- Der heutzutage übliche Begriff **Boolesche Algebra** (engl.: boolean algebra) geht auf den britischen Mathematiker **George Boole** zurück, der im 19. Jahrhundert lebte.
- In der Programmierung verwenden wir die boolesche Algebra vor allem für **logische Ausdrücke**, die nach der klassischen **Aussagenlogik** systematisch mit den Wahrheitswerten **wahr** und **falsch** umgehen.
- Ein Beispiel für eine einfache Aussage:
  - **Ich habe Hunger.**
- Diese Aussage kann wahr oder falsch sein.
- Eine Aussage kann **negiert** werden:
  - **Ich habe keinen Hunger.**
- Auch diese Aussage kann wahr oder falsch sein.
- Wenn wir annehmen, dass die erste Aussage falsch war, dann ist die zweite Aussage automatisch wahr, weil sie eine logische **Negation** der ersten darstellt.

Die Boolesche Algebra (II)

- Eine zweite Aussage, die ebenfalls wahr oder falsch sein kann:
  - **Ich habe Durst.**
- Wir können zwei Aussagen logisch miteinander verknüpfen:
  - **Ich habe Hunger und ich habe Durst.**
- Eine solche **Konjunktion** mit **und** ist für sich genommen wenig spannend; interessanter wird es, wenn wir die Verknüpfung zu einer **Bedingung** für eine Tätigkeit machen:
  - **Wenn ich Hunger und Durst habe, dann nehme ich etwas zu mir.**
- Ist das eine sinnvolle Aussage? Vermutlich nehmen wir auch etwas zu uns, wenn wir nur hungrig oder nur durstig sind… Also:
  - **Wenn ich Hunger oder Durst habe, dann nehme ich etwas zu mir.**
- Eine solche Verknüpfung mit **oder** wird auch **Disjunktion** genannt.

Boolesche Algebra in der Programmierung (mit Java)

- Wir können Aussagen, die wahr oder falsch sein können, unmittelbar mit **booleschen Variablen** in unseren Programmen beschreiben:

```
boolean hungrig = true; // boolesche Variable für die Aussage: Ich habe Hunger.
hungrig = false;        // Die Aussage kann mal wahr, mal falsch sein.
hungrig = !hungrig;     // Wir können eine Aussage negieren .
boolean durstig = false;
if (hungrig && durstig) // Wenn ich hungrig und durstig bin…
{
    ...
}
if (hungrig || durstig) // Wenn ich hungrig oder durstig bin…
```

- Boolesche Variablen sollten die **Aussage**, für die sie stehen, **klar ausdrücken**.
  - Im Beispiel ist `hungrig` gut gewählt: Wenn diese Variable mit `true` belegt ist, dann ist die Person hungrig; das ist gut verständlich.

  - Schlecht benannt wäre hingegen eine boolesche Variable `ausrichtung` für die Händigkeit einer Person – soll `true` für Rechts- oder Linkshändigkeit stehen?

Boolesche Ausdrücke: primär für Bedingungen

- Boolesche Ausdrücke (also Ausdrücke, die zur Laufzeit entweder den Wert **true** oder **false** liefern) werden überwiegend für **Bedingungen** eingesetzt, also in Situationen, in denen etwas entweder getan oder nicht getan werden soll.
- Wir kennen für solche Zwecke bereits die Fallunterscheidung:

```
if (x > 100) { … } else { … }
```

- Demnächst lernen wir zusätzlich **Schleifen** kennen, in denen Anweisungen wiederholt ausgeführt werden, solange eine Bedingung zutrifft:

```
while (x < 10)
{ …
}
```

- Bedingungen sind aber nicht auf solche **Vergleiche** beschränkt; eine Bedingung kann auch direkt mit einem **booleschen Literal** (also `true` oder `false`) formuliert werden, mit einer einfachen **booleschen Variablen**, mit dem Aufruf einer **booleschen Methode** oder einer beliebigen booleschen Verknüpfung all dieser Elemente.

Beispiele für Bedingungen

- Mit Literal (hier zum „Auskommentieren“ von Blöcken):

```
if (false) { Diese; Anweisungen; werden; niemals; ausgeführt; }
```

- Mit boolescher Variable:

```
if (versichert) { schreibe_versicherung_an(); }
```

- Mit boolescher Methode:

```
if (adminrechte_vorhanden(benutzer)) { installiere_update(); }
```

- Mit booleschen Verknüpfungen:

```
if ((kollision && !unverwundbar) || (restzeit() == 0))
{
    game_over();
}
```

Boolesche Operationen: Wahrheitstafeln

- Die Negation, die Konjunktion mit **und** (engl.: and) und die Disjunktion mit **oder** (engl.: or) sind die wichtigsten Operationen der booleschen Algebra.
- Diese Operationen bekommen einen oder zwei Wahrheitswerte und liefern jeweils einen Wahrheitswert (siehe unten). Diese Eigenschaften sind **universell** (unabhängig von einer Programmiersprache).

Negation:

not	
false	true
true	false

Konjunktion:

and	false	true
false	false	false
true	false	true

Disjunktion:

or	false	true
false	false	true
true	true	true

Beispiele mit Literalen (kein Java!):

not not not true	not (27 < 12)
not (false or true)	3 < 6 = 7 > 5
(true and false) and true	false or (false = true) or 5 > 7

Boolesche Operationen: Einige Rechenregeln

Seien **P**, **Q** und **R** logische Variable, dann gelten die folgenden Identitäten:

Kommutativgesetze:
P or Q ≡ Q or P
P and Q ≡ Q and P

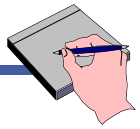
Distributivgesetze:
(P and Q) or R ≡ (P or R) and (Q or R)
(P or Q) and R ≡ (P and R) or (Q and R)

Assoziativgesetze:
(P or Q) or R ≡ P or (Q or R)
(P and Q) and R ≡ P and (Q and R)

De Morgans Gesetze:
not (P or Q) ≡ not P and not Q
not (P and Q) ≡ not P or not Q

Grundannahme: Der Operator **not** bindet stärker als die Operatoren **and** und **or**.

Typumwandlungen



- Typprüfungen bewahren uns vor Fehlern. Die Zuweisung

  `int i = true;` ⇒ **Typfehler!**

  beispielsweise führt bei der Übersetzung zu einer Fehlermeldung, weil der Ausdruck rechts vom Typ `boolean` ist, auf der linken Seite der Zuweisung aber eine `int`-Variable steht.
- Andererseits erwarten wir, dass die Zuweisung

  ```
  double d = 5;
  ```

  funktioniert, obwohl auf der rechten Seite ein `int`-Ausdruck steht und auf der linken Seite eine `double`-Variable.
- Die Lösung sind so genannte **Typumwandlungen** (engl.: type conversion oder type cast), die in Programmiersprachen **implizit** (automatisch) oder **explizit** (durch den Programmierer) durchgeführt werden.
- Eine Typumwandlung bewirkt zur Laufzeit eine Umwandlung einzelner Bits. Die Art der Umwandlung hängt dabei von Ausgangstyp und Zieltyp ab.

Automatische Typumwandlungen in Java

- **Zieltyp hat höhere Genauigkeit als Ausgangstyp**
  Umwandlung kann automatisch vorgenommen werden (engl.: coercion), weil keine Genauigkeit verloren gehen kann (engl. auch: widening conversion). Die Zuweisung

  ```
  double d = 5;
  ```

  ist deshalb in Java zulässig, weil sich alle `int`-Wert auch als `double`-Werte darstellen lassen. Das Bitmuster für den Wert **5** im Zweierkomplement wird bei der Ausführung in die Gleitkomma-Darstellung nach IEEE 754 umgewandelt.

  Ein weiteres Beispiel:

  ```
  int i = 'a';
  ```

  Automatische Umwandlungen können auch mehrfach innerhalb eines Ausdrucks auftreten:

  ```
  double d = 3 + '4' – 3.1415f;
  ```

Explizite Typumwandlungen in Java

- **Zieltyp hat niedrigere Genauigkeit als Ausgangstyp**
  Weil bei der Umwandlung Genauigkeit verloren gehen kann (engl.: narrowing conversion), muss der Programmierer die Umwandlung explizit erzwingen (und wissen, was er tut).
  Die Zuweisung

```
int i = 3.1415;
```
⇒ Fehlermeldung: possible loss of precision!

  ist in Java nicht zulässig, weil die Genauigkeit des Gleitkomma-Ausdrucks bei der Zuweisung an eine `int`-Variable verloren gehen kann.

  Wenn wir diesen Verlust bewusst in Kauf nehmen wollen, schreiben wir vor den Ausdruck **in runden Klammern** explizit den Zieltyp der Umwandlung:

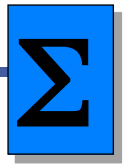
```
int i = (int)3.1415;
```

  Dies bewirkt eine Umwandlung in die ganze Zahl **3** vom Typ `int`.

  Ein häufig gemachter Fehler in Java in diesem Zusammenhang:

```
float f = 3.1415;
```
⇒ **???**

Zusammenfassung elementare Typen



- Programmiersprachen bieten üblicherweise einen Satz an **elementaren Typen**.
- Die Werte elementarer Typen werden im Quelltext mit **Literalen** benannt.
- **Java** verfügt über so genannte **primitive Typen** für
  - ganze Zahlen (realisiert als Zweierkomplement)
  - Wahrheitswerte
  - Zeichen (basierend auf dem **Unicode-Zeichensatz**)
  - Gleitkommazahlen (realisiert nach IEEE 754)
- Zwischen den primitiven Typen können explizite und implizite **Typumwandlungen** stattfinden.

