Software-Entwicklung 1

V05: Basistypen und Operationen

Prof. Maalej & Team - @maalejw

Status der 4. Übungswoche

Zeit	Montag	Dienstag	Mittwoch	Donnerstag	Freitag
Vormittag	Gruppe 1 **Erfüllt: 79%**	Gruppe 3 **Erfüllt: 81%**	Gruppe 5 **Erfüllt: 86%**	Gruppe 6 **Erfüllt: 80%**	Gruppe 8 **Erfüllt: 64%**
Nachmittag	Gruppe 2 **Erfüllt: 79%**	Gruppe 4 **Erfüllt: 74%**	Vorlesung	Gruppe 7 **Erfüllt: 61%**	

Überblick

1. **Basistypen**
2. Operationen und Ausdrücke
3. Boolesche Algebra

Konzepte und Phänomenen

- **Phänomen**
  - Ein Objekt in der Welt wie wir es wahrnehmen
    - Beispiel: Diese Vorlesung um 14:55, meine schwarze Uhr
- **Konzepte**
  - Beschreiben die gemeinsamen Eigenschaften von Phänomenen
    - Beispiel: Alle Softwareentwicklung- Vorlesungen
    - Beispiel: Alle schwarze Uhren
- **Ein Konzept ist ein Tripel:**
  - **Name:** unterscheidet das Konzept von anderen Konzepten
  - **Zweck:** Eigenschaften die entscheiden ob ein Phänomen ein Teil eines Konzeptes ist
  - **Elemente:** Die Menge der Phänomenen, die Teil des Konzept sind

Die Aktivität des Abstrahierens…

… ist die Klassifikation von Phänomenen in Konzepte

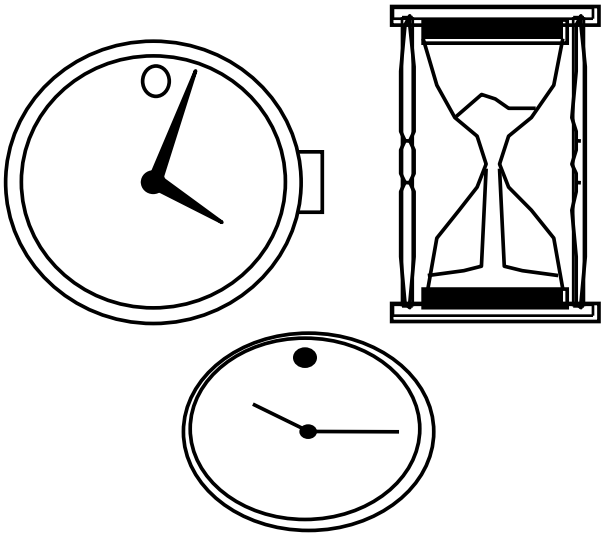
Name

Uhr

Zweck

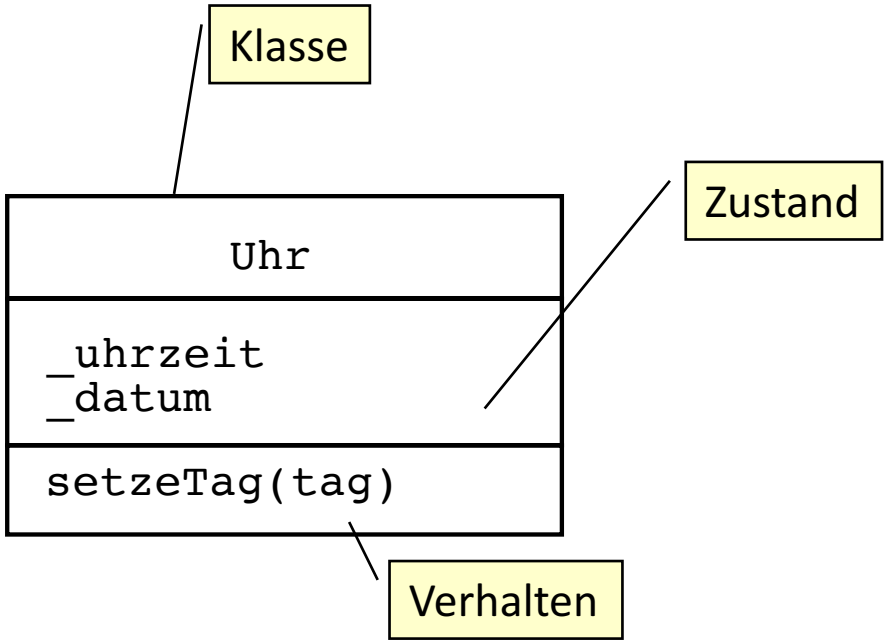
Ein Gerät um die Zeit zu messen

Elemente



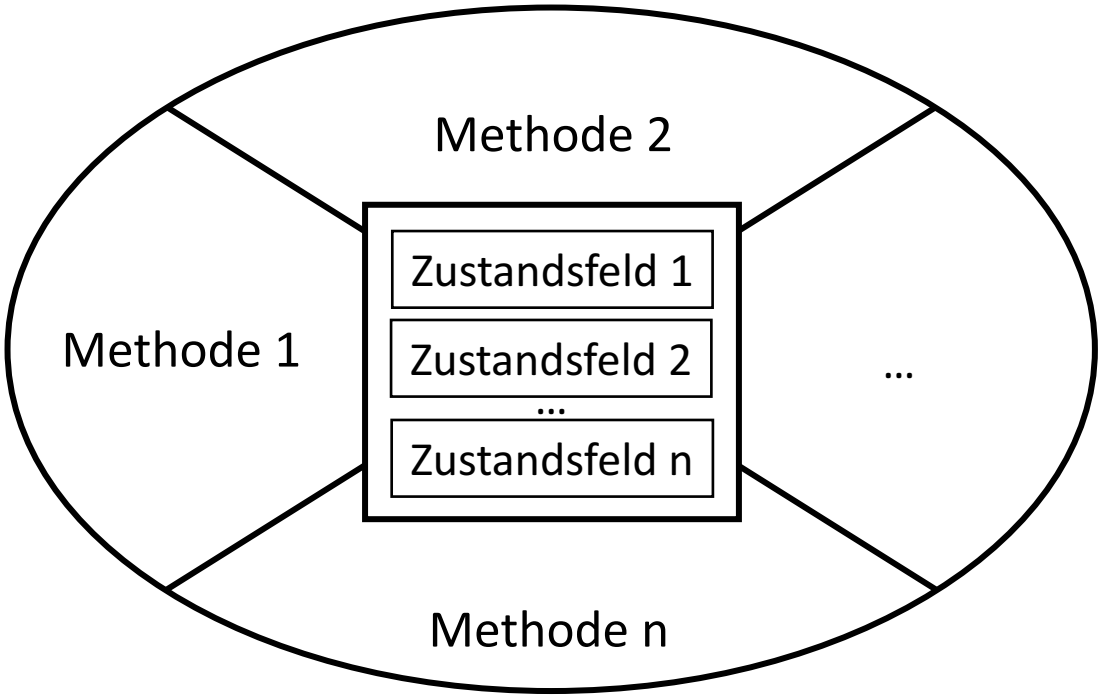
Klassen

- Sind Abstraktionen in Objekt-Orientierte Paradigma
  - Eine Klasse kapselt den Zustand und das Verhalten
  - Beispiel: Uhr



Bisherige Sicht auf Objekte

- Objekte sind **Exemplare** von **Klassen**
- Klassendefinition beschreibt das Verhalten und Zustände der Exemplare
  - **Verhalten** wird definiert über die **Methoden**
  - **Zustände** werden definiert über die **Zustandsfelder**



Typen und Objekte

- **Typ:**
  - Ein Konzept in Programmiersprachen
    - Name: int
    - Zweck: Die ganzen Zahlen
    - Elemente: `0,-1,1,2,-2,…`
    - Name: Boolean
    - Zweck: Wahrheitswerte
    - Elemente: {true, false}
- **Objekte (Exemplare):**
  - Elemente eines spezifischen Typs
- Der Typ einer Variable legt alle möglichen Belegungen (Elemente) der Variable fest
- Diese Abhängigkeiten sind synonym:
  - Typ <–> Belegung einer Variable
  - Konzept <–> Phänomen
  - Klasse <-> Objekt

Typbegriff

- Der Typ legt fest:
  - Mögliche Elemente der **Wertemenge**
  - Zulässigen **Operationen**

Beispiel

Typ: int
Wertemenge: { -2.147.483.648 ... 2.147.483.647 }
Operationen:
- ganzzahlig Addieren
- ganzzahlig Subtrahieren
- ganzzahlig Multiplizieren
- …

Elementare Typen (Basistypen)

Datentyp	Zweck	Größe	Wertemenge	Beispiel
byte	ganze Zahlen	1 byte	-128 bis +127	`byte b = 65;`
short		2 bytes	-32.768 bis +32.767	`short s = 65;`
int		4 bytes	-2^{31} bis $2^{31}-1$	`int i = 65;`
long		8 bytes	-2^{63} bis $2^{63}-1$	`long i = 65L;`
float	Gleitkommazahlen	4 bytes	±1.4E-45 bis ±3.4E38	`float f = 65f;`
double		8 bytes	±4.9E-324 bis ±1.7E308	`double d = 65.55;`
char	Zeichen	2 bytes	16-bit Unicode Zeichen	`char c = 'A';` `char c = 65;`
boolean	Wahrheitswerte	1 bit	true oder false	`boolean b = true;`

Elementare Datentypen in Java

Auch Java besitzt den üblichen Satz an elementaren Datentypen („**primitive types**“). 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'

Es ist empfehlenswert, in einem Java-Referenzbuch bei Bedarf weitere Details nachzulesen!

Literale

- **Werte** der **elementaren Typen** können direkt im Quelltext stehen
- Zeichenfolge im Quelltext, die einen Wert eindeutig repräsentiert und deren Struktur dem Compiler bekannt ist
- Beispiel:
  - `13`
  - `14.21`
  - `‘a‘`
  - `"gelb"`
  - `true`

Zeichen und ihre Darstellung

- Zeichen werden im Speicher durch vordefinierte Werte eines Zeichensatzes repräsentiert (Codes)

- In Java werden einzelne Zeichen im Quelltext als Literale in Hochkommata notiert:

```
'*', '0', 'A', 'z'
```

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

Rückblick: 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**

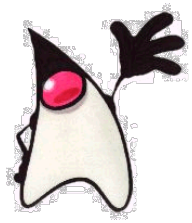
in gelb: die druckbaren ASCII-Zeichen

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**



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

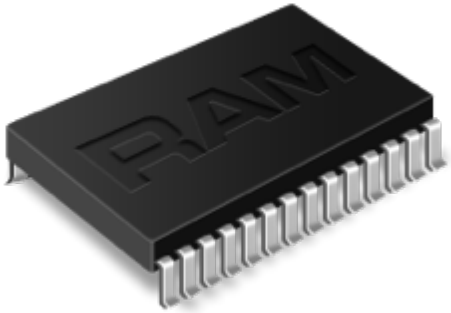
Literaltyp	Beispiel	Kommentar
int	542 oder -1	Ganze Zahlen
int	035	Entspricht 29, führende Null für Oktalschreibweise
int	0x1D	Entspricht 29, „0x“ für Hexadezimalschreibweise
double	0.5 oder 5e-1	Double-Gleitkommazahl
float	0.5f oder 5e-1f	Float-Gleitkommazahl

Die Hexziffern in der Übersicht

Dezimal	Binär	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

Dezimal	Binär	Hex
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Fallstricke mit Gleitkommazahlen



- Zahlen werden begrenzt im Speicher dargestellt
- Beispiel:
  - Größte und eine kleinste darstellbare ganze **int** Zahl (Maxint und Minint)
  - Gleitkommazahlen haben eine begrenzte Genauigkeit.

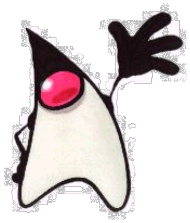
- Operationen auf ganzen Zahlen werden im **Wertebereich** exakt berechnet

- Dies gilt nicht für Gleitkommaarithmetik
  - Beispiel: (x+y)-y kann unter Umständen nicht exakt den Wert x haben
  - Beispiel: 0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1
- Fundamentales Problem der digitalen Datenverarbeitung

Gleitkommazahlen in Java (I)

- Java definiert die primitiven Typen float und double für Gleitkommazahlen nach dem IEEE Standard 754. Dieser Standard legt einfache Genauigkeit mit 32 Bit fest (engl.: single precision, in Java durch float umgesetzt) und doppelte Genauigkeit mit 64 Bit (engl.: double precision, in Java durch double umgesetzt).

- IEEE 754 definiert Summen von Zweierpotenzen und nicht, wie wir es aus dem Alltag gewohnt sind, Summen von Zehnerpotenzen. Durch diesen Bruch zum Dezimalsystem können auch solche Werte nicht exakt dargestellt werden, von denen wir es intuitiv erwarten würden, z.B. der relativ glatte Dezimalbruch 0,1. Als Dualbruch dargestellt sieht er so aus:

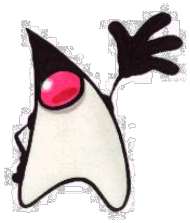


$$0{,}110 = 0{,}0001100110011001100\ldots_2 = 0{,}000112$$

- Diese unendliche Periode lässt sich in einer begrenzten Anzahl von Bits nicht erfassen.

Gleitkommazahlen in Java (II)

- Beim Umgang mit Gleitkommazahlen muss aufgrund dieser Eigenschaften besonders auf Wertebereich und Genauigkeitsgrenzen geachtet werden.
- Wie bei den Typen für ganze Zahlen stehen in Java auch für float und double die vier Grundrechenarten über Infix-Operatoren zur Verfügung, ebenso wie die Vergleichsoperatoren.
- Zu beachten dabei: Die Operatoren sind die gleichen, aber die Operationen sind teilweise sehr unterschiedlich!



Gleitkommazahlen sind ein umfangreiches Thema.
Unter https://tams.informatik.uni-hamburg.de/lectures/2016ws/vorlesung/rs/ gibt es mehr Informationen zu IEEE 754, aber auch zum Zweierkomplement (int etc.).

Typumwandlungen

- Typprüfungen bewahren uns vor Fehlern. Die Zuweisung

```
int i = true;
```

→ Typfehler!

- Diese Zuweisung funktioniert, obwohl auf der rechten Seite ein int-Ausdruck steht und auf der linken Seite eine double-Variable:

```
double d = 5;
```

- Typumwandlungen (engl.: type conversion oder type cast)
  - implizit (automatisch)
  - explizit (durch den Programmierer)

- Zur Laufzeit eine Umwandlung einzelner Bits
- Art der Umwandlung hängt dabei von Ausgangstyp und Zieltyp

Automatische Typumwandlungen in Java

- Zieltyp hat höhere Genauigkeit als Ausgangstyp (engl. *widening conversion*)

```
double d = 5;
```

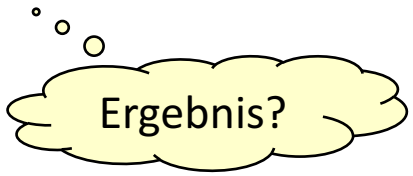
- ist zulässig, weil sich alle int-Werte auch als double-Werte darstellen lassen
- Bitmuster für den Wert 5 wird bei der Ausführung in die Gleitkomma-Darstellung umgewandelt.

Ein weiteres Beispiel:

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

- Automatische Umwandlungen können auch mehrfach innerhalb eines Ausdrucks auftreten:

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



Explizite Typumwandlungen

- Zieltyp hat niedrigere Genauigkeit als Ausgangstyp (engl.: narrowing conversion)
- 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

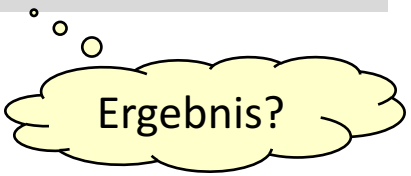
- Explizite Typumwandlung:

```
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:

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



Überblick

1 Basistypen

2 **Operationen und Ausdrücke**

3 Boolesche Algebra

Operatoren

- Als **Operator** bezeichnet man umgangssprachlich
  - das **Operatorzeichen** (z.B. "+")
  - die damit verbundene **Operation** (z.B. "addieren“)

- Operatorenschreibweise ist im Zusammenhang mit Programmiersprachen allgemein gebräuchlich

Vereinbarungen über Operatoren

- **Vereinbarungen über Operatoren** sind:
  - Position
  - Stelligkeit
  - Präzedenz (Vorrangregel)
  - Assoziationsreihenfolge
  - Definition der mit dem Operator verbundenen Operation

- Operatorenschreibweise ist intuitiv

- Bei Neueinführung von Operatoren müssen Vereinbarungen explizit gemacht werden

Position von Operatoren

Infix

- Häufigste Schreibweise
- Operatoren stehen zwischen den beiden Operanden

  z.B. : **3 * 4**

Präfix (Funktionsschreibweise)

- Operator steht vor seinen Operanden
- Oft benutzt bei Operationen mit einem Operanden

  z.B.: **-2**

Postfix

- Operator steht nach seinen Operanden
- Oft benutzt für arithmetische Operationen mit einem Operanden

  z.B.: **3!** (“3 Fakultät")

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

Präzedenz von Operatoren

- **Präzedenz** (Vorrangregel): bezeichnet die Stärke, mit der ein Operator seine Operanden „bindet“
- Wert eines Ausdrucks ist oft abhängig von der Reihenfolge der Auswertung ab
- „**Punkt vor Strich**“ - für Arithmetische Operationen
  - Mit Klammern können Präzedenzen explizit bestimmt werden
  - Beispiele:
    - `3 + 5 * 7 – 3`
    - `(3 + 5) * (7 – 3)`

Assoziativität von Operatoren

- Bestimmt die implizite Klammerung von Ausdrücken bei Operatoren gleicher Präzedenz
  - Beispiel:
    - 5 - 4 - 3 ist gleichbedeutend mit (5 - 4) - 3
    - Sprechweise: Operator ist **linksassoziativ** (assoziiert von links nach rechts)

- Assoziationsreihenfolge irrelevant, wenn die Auswertungsreihenfolge nichts am Wert ändert
  - Beispiel:
    - (3 + 4) + 5
    - 3 + (4 + 5)

Zweistellige Operatoren für ganze Zahlen mit Ergebnistyp int

- Vier Grundrechenarten in Java: + - * /
- Rest bei ganzzahliger Division mit dem % Operator
- Infixnotation
- Präzedenz ist „Punktrechnung vor Strichrechnung“

Operator	Bezeichnung	Ausdruck	Ergebnis (int)
+	Plus	14 + 3	17
-	Minus	24 - 13	11
*	Mal	7 * 4	28
/	Durch	20 / 6	3
%	Modulo	20 % 6	2

Zweistellige Operatoren für ganze Zahlen mit Ergebnistyp boolean

Operator	Bezeichnung	Ausdruck	Ergebnis (boolean)
>	Größer	2 > 1+1	false
>=	Größergleich	2 >= 3-1	true
<	Kleiner	2 * 2 < 1 * 1	false
<=	Kleinergleich	2 <= 2/1	true
==	Gleich	3 == 2+1	true
!=	Ungleich	4 != 2*2	false

- Vergleichoperatoren haben gegenüber den arithmetischen Operatoren eine niedrigere Präzedenz
- Werden im Ausdruck zuletzt ausgewertet

Boolesche Literale und Operatoren

- Boolesche Werte oder Wahrheitswerte sind in Java vom primitiven Typ boolean
- Literale für die booleschen Werte sind **true** und **false**

Operator	Bezeichnung	Ausdruck	Ergebnis (boolean)
&&	Logisches Und	true && false	false
||	Logisches Oder	true || false	true
!	Logische Verneinung	!true	false
==	Gleich	true == false	false
!=	Ungleich	true != false	true

Übersicht: Zentrale Operatoren in Java

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

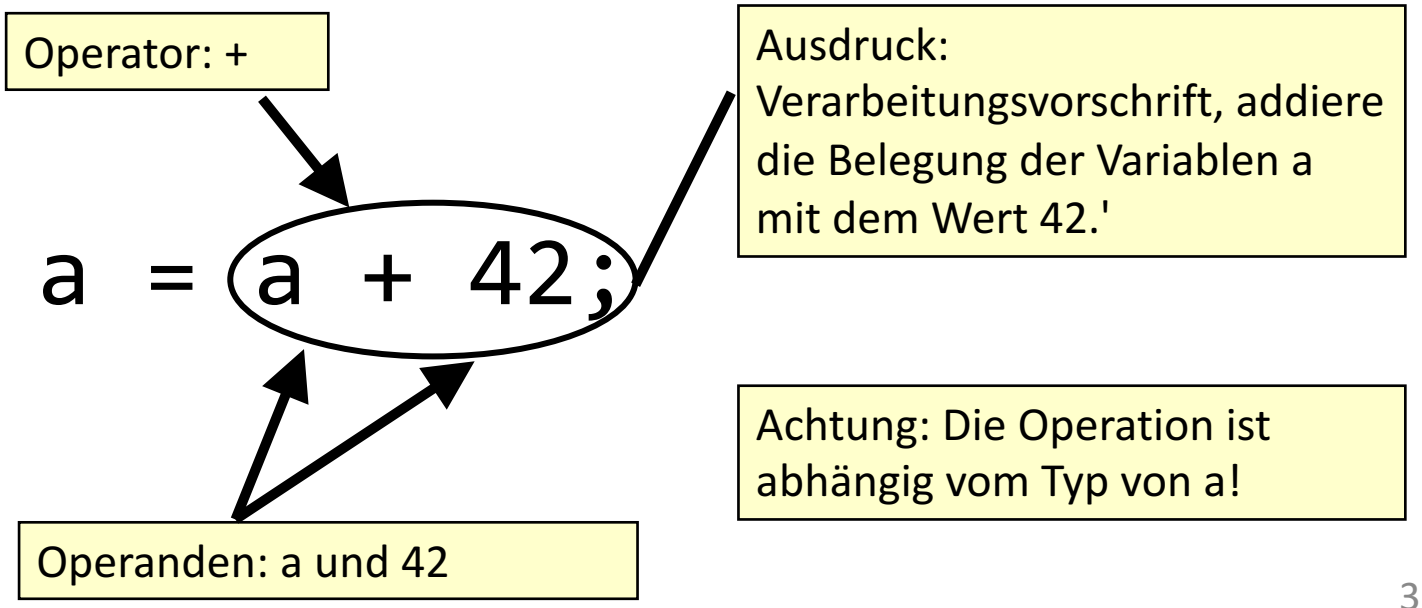
Sowohl prä- als auch postfix verwendbar

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

Operator	Funktion
=	Zuweisung

Ausdrücke und Operatoren

- Zuweisung hat einen zentralen Stellenwert in imperativer Programmierung
- Oft wird einer Variablen ein Wert zugewiesen indem ein **Ausdruck** aus **Operanden** und **Operatoren** ausgewertet wird
- **Arithmetische** und **boolesche** Operatoren sind die üblichen



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

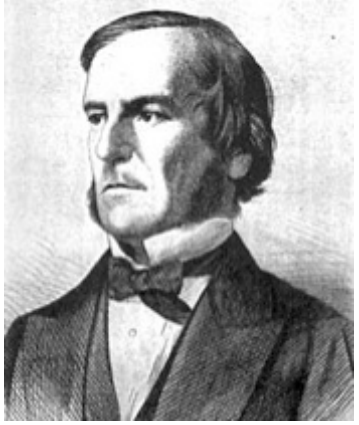
Überblick

1 Basistypen

2 Operationen und Ausdrücke

3 **Boolesche Algebra**

Die Boolesche Algebra



- Boolesche Algebra (EN.: boolean algebra) entstand durch den Mathematiker George Boole im 19. Jahrhundert
- 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.
- Beispiel für eine 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

- Aussagen können mit booleschen Variablen in unseren Programmen beschreiben werden

```
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

Boolesche Ausdrücke für Bedingungen

- Boolesche Ausdrücke werden überwiegend für Bedingungen eingesetzt
- Für Fallunterscheidung:

```
if (x > 100) { … } else { … }
```

- Für Wiederholungen:

```
while (x < 10)
{ …
}
```

- Bedingungen können kombiniert werden:

```
while (!nagelIstImBrett() && !istMittagsRuhe())
{
        schlagNagel();
}
```

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 Ergebnisprozedur:

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

- Mit booleschen Verknüpfungen:

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

Boolesche Operationen: Wahrheitstafeln

- Negation, Konjunktion und Disjunktion sind die wichtigsten Operationen der booleschen Algebra
- Diese Operationen bekommen einen oder zwei Wahrheitswerte und liefern jeweils einen Wahrheitswert

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 (false or true)

(true and false) and true

not (27 < 12)

3 < 6 = 7 > 5

false or (false = true) or 5 > 7

Boolesche Operationen: Einige Rechenregeln

Seien **P**, **Q** und **R** logische Variable, dann gilt folgendes:

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

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

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)

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**.

Achtung: Bei De Morgan

- Wir formulieren, dass wir arbeiten können, wenn es nicht der Fall ist, dass wir hungrig oder durstig sind: !(hungrig || durstig).
- Laut De Morgan ist dies äquivalent zu der Aussage: !hungrig && !durstig
- Die jeweils letzte Spalte in den beiden Tabellen zeigt uns: es kommen tatsächlich die gleichen Werte heraus.

hungrig	durstig	h || d	!(h || d)
false	false	false	true
false	true	true	false
true	false	true	false
true	true	true	false

hungrig	durstig	!hungrig	!durstig	!h && !d
false	false	true	true	true
false	true	true	false	false
true	false	false	true	false
true	true	false	false	false

Zusammenfassung

1. Programmiersprachen bieten einen Satz an elementaren Typen für ganze Zahlen, Wahrheitswerte, Zeichen, Gleitkommazahlen.

2. Die Werte elementarer Typen werden im Quelltext mit Literalen benannt.

3. Zwischen den primitiven Typen können explizite und implizite Typumwandlungen stattfinden.

4. Arithmetische und boolesche Ausdrücke setzen sich aus Operanden und Operatoren zusammen.