

252-0027

Einführung in die Programmierung

2.0 Einfache Java Programme

Thomas R. Gross

**Department Informatik
ETH Zürich**

Übersicht

- 2.0 Einfache Java Programme
- 2.1 Methoden
 - Struktur
- **2.2 Typen und Variable**
 - 2.2.1 Einführung
 - 2.2.2 Basistypen: Einfache (eingebaute) Typen
Operationen (mit Werten desselben und verschiedener Typen)
 - 2.2.3 Deklaration von Variablen

2.2.2 Primitive Types (eingebaute, Basis-Typen)

- Deklaration und Definition

`int x = expression; // Ausdruck`

- Was sind die Regeln für Ausdrücke?

- Zuerst für `int` (und `long`)
- EBNF Regel *expr*

EBNF Beschreibung Ausdruck (Expression)

number \Leftarrow *integer* | *integer* . { *digit* } | *sign* . *digit* { *digit* }

op \Leftarrow + | - | * | / | %

atom \Leftarrow *number* | *identifier*

term \Leftarrow (*expr*) | *atom*

expr \Leftarrow *term* { *op term* }

1. Nicht die vollständige Beschreibung für Java Ausdrücke

- Keine Überraschungen ... wie in der Mathematik/Schule
- *number* beschreibt Literals (nicht vollständig)

2. Beschreibt nur die Syntax

Arithmetische Operatoren

- **Operator: Verknüpft Werte (Literals) oder Ausdrücke.**

- + Addition
- Subtraktion (oder Negation)
- * Multiplikation
- / Division
- % Modulus (Rest)

- **Müssen festlegen was «a \otimes b» bedeutet (\otimes Operator; a,b int)**

- Was ist das Ergebnis?
- Was für einen Typ hat das Ergebnis?

Arithmetische Operatoren

- **Operator: Verknüpft Werte oder Ausdrücke.**

- + Addition

- / Division

-

- **Ergebnis hat einen [festgelegten] Typ**

- EBNF beschreibt nur Form (int + int *ergibt* ? **int**)

- Operator \otimes (+, -, /, *, %) :

- $\text{Typ_A} \otimes \text{Typ_A}$ *ergibt* Typ_A (für arithmetische Operatoren)

- $\text{Typ_A} \otimes \text{Typ_B}$ *ergibt* ??? (hängt von \otimes , Typ_A , Typ_B ab – später)

int Division mit /

- Wenn wir ganze Zahlen dividieren ist der Quotient auch wieder eine ganze Zahl.
- $14 / 4$ ergibt 3, nicht 3.5

$$\begin{array}{r} \underline{3} \\ 4 \) \ 14 \\ \underline{12} \\ 2 \end{array}$$

$$\begin{array}{r} \underline{4} \\ 10 \) \ 45 \\ \underline{40} \\ 5 \end{array}$$

$$\begin{array}{r} \underline{52} \\ 27 \) \ 1425 \\ \underline{135} \\ 75 \\ \underline{54} \\ 21 \end{array}$$

int Division mit /

- **Weitere Beispiele:**

- 32 / 5 ergibt 6
- 84 / 10 ergibt 8
- 156 / 100 ergibt 1
- -4 / 3 ergibt -1
- -101 / 9 ergibt -11

- **Division durch 0 führt zu einem Fehler während der Ausführung des Programmes**

int Rest mit %

- Der % Operator liefert den Rest der Division ganzer Zahlen
 - $14 \% 4$ ergibt 2
 - $218 \% 5$ ergibt 3

$$\begin{array}{r} 3 \\ \hline 4 \) \ 14 \\ \underline{12} \\ 2 \end{array}$$

$$\begin{array}{r} 43 \\ \hline 5 \) \ 218 \\ \underline{20} \\ 18 \\ \underline{15} \\ 3 \end{array}$$

int Rest mit %

- **Einsatz des % Operators:**

- Finde letzte Ziffer einer ganzen Zahl : $230857 \% 10$ ist 7
- Finde letzte 4 Ziffern: $658236489 \% 10000$ ist 6489
- Entscheide ob Zahl gerade ist: $7 \% 2$ ergibt 1,
 $42 \% 2$ ergibt 0

Was ist das Ergebnis von

1. $10 / 4$

2. $45 \% 6$

3. $-23 / 11$

4. $11 \% 0$

5. $2 \% 2$

6. $43 / 3$

7. $8 \% 20$

Was ist das Ergebnis von

1. $10 / 4$ 2

2. $45 \% 6$ 3

3. $-23 / 11$ -2

4. $11 \% 0$ Exception java.lang.ArithmeticException: / by zero

5. $2 \% 2$ 0

6. $43 / 3$ 14

7. $8 \% 20$ 8

Ausdrücke mit mehreren Operanden

- Hat ein Ausdruck mehrere Operanden X, Y, Z (mit Operator \odot) so müssen wir festlegen, was

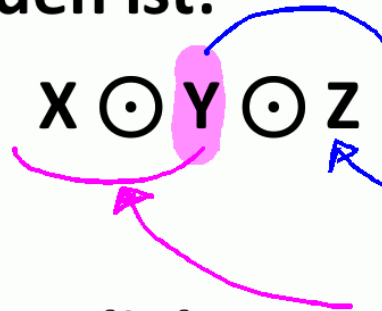
$$X \odot Y \odot Z$$

bedeutet.

- $7 + 5 + 3 \quad \rightarrow (7 + 5) + 3$
- $64 / 8 / 2 \quad \rightarrow (64 / 8) / 2 \quad \text{und nicht } 64 / (8 / 2)$
- Wird durch die *Assoziativität* der Operatoren bestimmt

Assoziativität ("Associativity") -- Bindung

- Die Assoziativität eines Operators \odot hält fest wie ein Operand zu verwenden ist:



- Y ist mit dem *linken* Operator verknüpft ("left—associative", "left—to—right associative")

$$X \odot Y \odot Z = (X \odot Y) \odot Z$$

- Y ist mit dem *rechten* Operator verknüpft ("right—associative", "right—to—left associative") $X \odot Y \odot Z = X \odot (Y \odot Z)$

Assoziativität

- **Links-assoziativ:** \odot ist mit dem *linken* Operator verknüpft («left-associative», «left-to-right associative»)

$$X \odot Y \odot Z = (X \odot Y) \odot Z$$

Viele der uns bekannten Operatoren: $+$, $*$,

- **rechts-assoziativ:** \odot ist mit dem *rechten* Operator verknüpft («right-associative», «right-to-left associative»)

Später werden wir Beispiele sehen (es gibt einige!)

- **Es gibt Operatoren die sind *assoziativ* (in der Mathematik)**
 - Rechts—assoziativ *und* links—assoziativ: $(X \odot Y) \odot Z = X \odot (Y \odot Z)$

Aber es gibt noch mehr zu bedenken ...

- Was für einen Wert erhalten wir für

$$2 + 6 * 5$$

30

32

Rang Ordnung

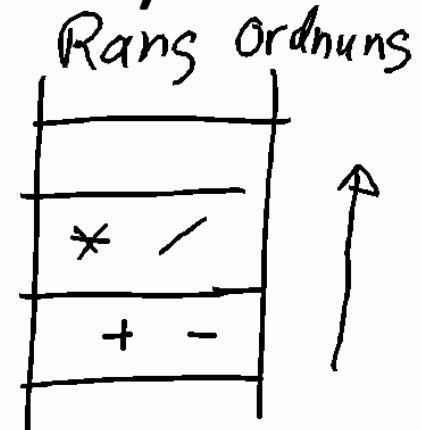
- Hat ein Ausdruck mehrere Operanden X, Y, Z (mit Operatoren \odot und \otimes) so müssen wir festlegen, was

$$X \odot Y \otimes Z$$

bedeutet. (\odot und \otimes sind unterschiedliche Operatoren)

Operand Y kann zuerst mit \odot oder \otimes verknüpft werden

- Die *Rang Ordnung* von \odot und \otimes entscheidet.



Rang Ordnung («Precedence»)

- Der Operand (Y) in $X \odot Y \otimes Z$ wird mit dem Operator verknüpft, der den höheren Rang hat.

$$1 + 3 * 4$$

$$1 + \mathbf{3 * 4}$$

$$1 + 12$$

$$\mathbf{1 + 12}$$

$$13$$

Rang Ordnung

$$6 + 8 / 2 * 3$$

links anfangen

6 ist Literal

8 ist Literal

/ hat höheren Rang als +

links anfangen: / zuerst

/ und * haben selben Rang

int Division ergibt 4

* hat höheren Rang als +

nur noch ein Operator

Resultat:

$$6 + 4 * 3$$

$$6 + 12$$

$$18$$

Operanden und Operatoren

- Operand wird vom Operator mit höherer Rang Ordnung («precedence», Präzedenz) verwendet
- Wenn zwei Operatoren die selbe Rang Ordnung haben, dann entscheidet die Assoziativität
- Wenn zwei Operatoren die selbe Rang Ordnung und Assoziativität haben, dann werden die (Teil)Ausdrücke von links nach rechts ausgewertet.
- Wenn etwas anderes gewünscht wird: Klammern verwenden!

Operanden und Operatoren

- Klammern bestimmen abweichende die Gruppierung:

$(1 + 3) * 4$ ergibt 16

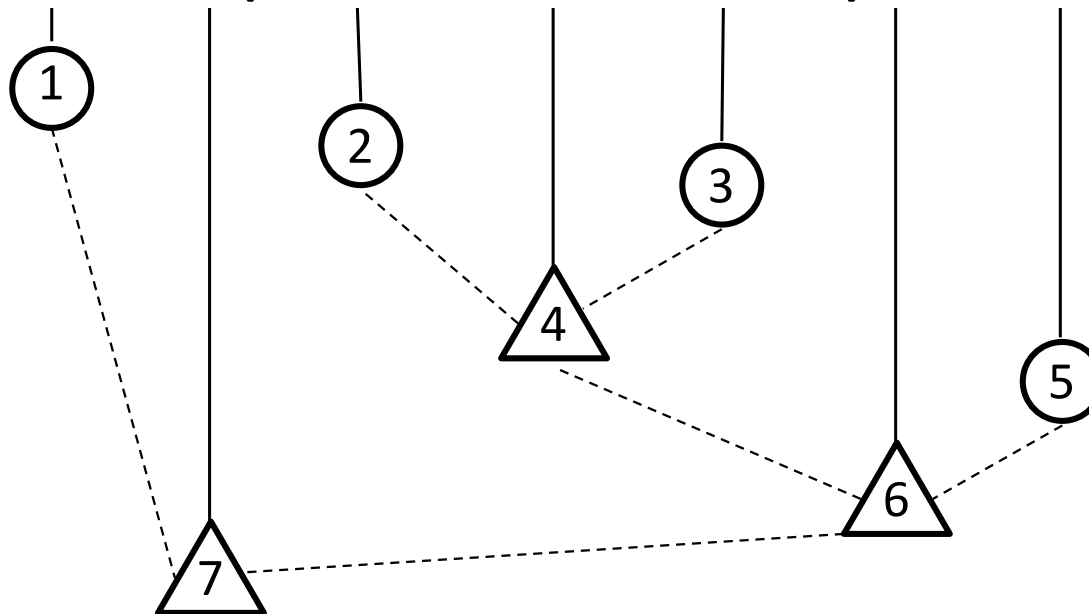
- Leerzeichen (oder Kommentare) haben keinen Einfluss auf die Reihenfolge der Auswertung

$1+3 * 4-2$ ergibt 11 (trotzdem keine gute Idee!)

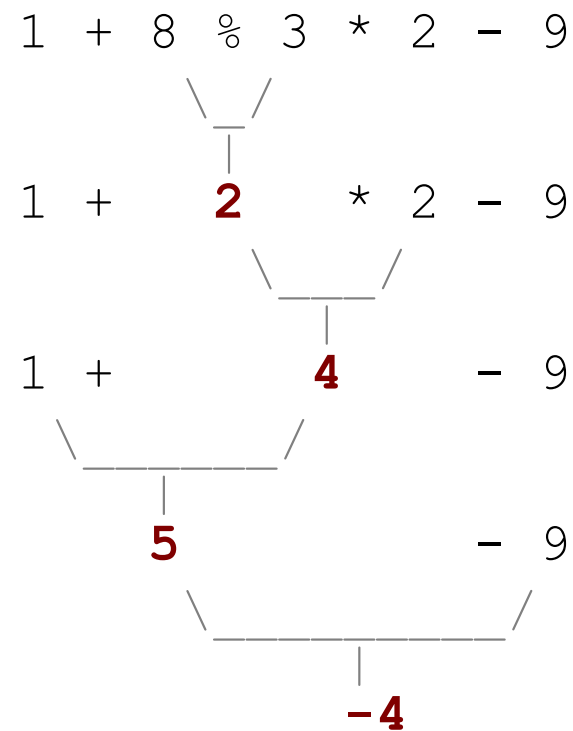
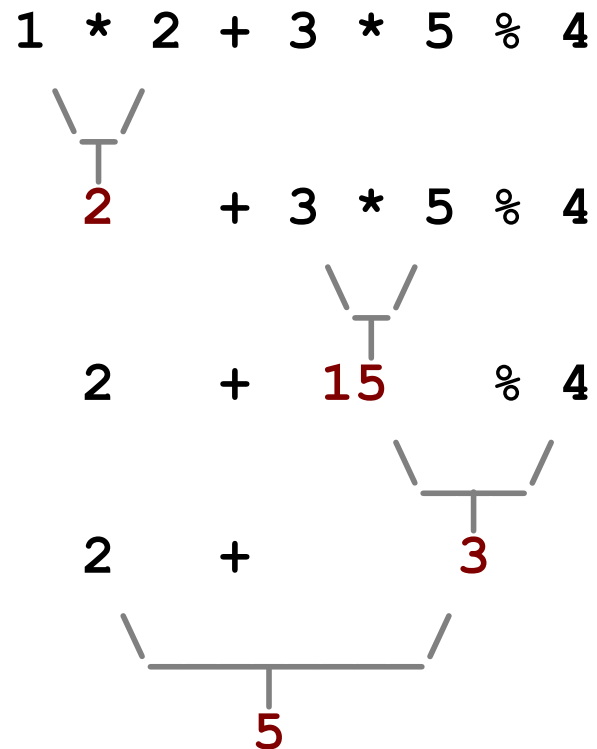
Operanden und Operatoren

- Beispiel Rang Ordnung und Assoziativität:

Ausdruck1 + (Ausdruck2 + Ausdruck3) * Ausdruck4



Rang Ordnung Beispiele



Rang Ordnung Beispiele

Welche Werte ergeben die Auswertung dieser Ausdrücke?

- $9 / 5$
- $695 \% 20$
- $7 + 6 * 5$
- $7 * 6 + 5$
- $248 \% 100 / 5$
- $6 * 3 - 9 / 4$
- $(5 - 7) * 4$
- $6 + (18 \% (17 - 12))$

$$9 / 5 ==> 1$$

$$695 \% 20 ==> 15$$

$$7 + 6 * 5 ==> 37$$

$$7 * 6 + 5 ==> 47$$

$$248 \% 100 / 5 ==> 9$$

$$6 * 3 - 9 / 4 ==> 16$$

$$(5 - 7) * 4 ==> -8$$


$$6 + (18 \% (17 - 12)) ==> 9$$


Reelle Zahlen (Typ `double`)

- Beispiele: `6.022` `-42.0` `2.143e17`
 - Hinzufügen von `.0` (oder `.`) an eine ganze Zahl macht diese zu `double`.
- Die Operatoren `+` `-` `*` `/` `%` und `()` sind auch für `double` definiert.
 - `/` berechnet ein genaues Resultat: `15.0 / 2.0` ergibt `7.5`
 - Rang Ordnung (der Auswertung) ist die selbe: `()` vor `*` `/` `%` vor `+` `-`

Beispiel mit reellen Zahlen

$$2.0 * 2.4 + 2.25 * 4.0 / 2.0$$


$$4.8 + 2.25 * 4.0 / 2.0$$


$$4.8 + 9.0 / 2.0$$


$$4.8 + 4.5$$


$$9.3$$

Arithmetische Operatoren

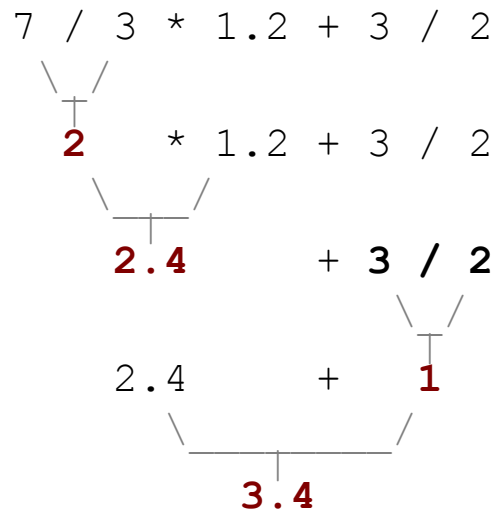
- **Evaluation eines Ausdrucks ergibt Wert eines Typs**
 - EBNF beschreibt nur Form
 - $\text{int} + \text{int}$ *ergibt ?* , $\text{int} + \text{long}$ *ergibt ?* – keine Antwort von EBNF
- **Operanden haben den selben Typ: Ergebnis hat selben Typ**
 - Operator \otimes : $\text{Typ_A} \otimes \text{Typ_A}$ *ergibt* Typ_A (für arithmetische Operatoren)
 - (Nehmen an dass das Ergebnis dargestellt werden kann – fixer Wertebereich für alle Typen!)
- **Operanden haben unterschiedliche Typen**
 - $\text{Typ_A} \otimes \text{Typ_B}$ *ergibt* $????$ (jetzt)

Kombinationen von Typen

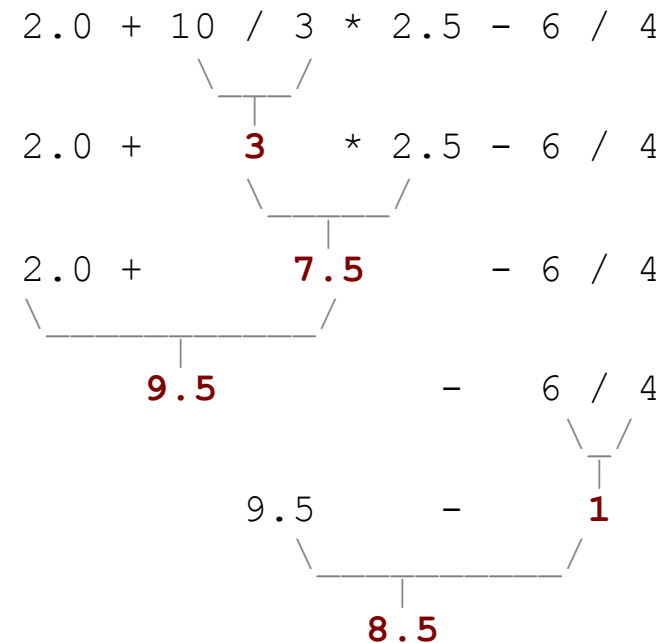
- Wenn `int` oder `long` und `double` kombiniert werden dann ist das Ergebnis `double`.
 - `4.2 * 3` ergibt `12.6`
- Auch dann wenn das Result als `int` darstellbar wäre
 - `4.5 * 2` ergibt `9.0`
- **Umwandlung («conversion») von einem Typ in einen anderen**
 - Wird für jeden Operator separat gemacht und beeinflusst nur dessen Operanden
 - *Automatisch* («implizit») – werden später sehen wie wir Umwandlungen erzwingen können

Ausdrücke mit verschiedenen Typen

- Umwandlung geschieht für jeden Operator separat ...



3 / 2 ergibt 1 nicht 1.5.



Ausdrücke mit verschiedenen Typen

Bitte evaluieren Sie diese Ausdrücke

$$4 + 8 / 3.0 * 6 + 5$$

$$9.0 / (2.0 / 3) + 7$$

$$7 \% 3 * 2 + 4.0 * 3 / 2$$

$$9 / (2 / 3) + 7$$

$$20 \% 8 + 15 / 27 / (3 \% 6)$$

Ausdrücke mit verschiedenen Typen

Bitte evaluieren Sie diese Ausdrücke

$4 + 8 / 3.0 * 6 + 5$ **25.0**

$9.0 / (2.0 / 3) + 7$ **20.5**

$7 \% 3 * 2 + 4.0 * 3 / 2$ **8.0**

$9 / (2 / 3) + 7$ **Laufzeitfehler**

$20 \% 8 + 15 / 27 / (3 \% 6)$ **4**

Division durch 0 ist nicht erlaubt. $2/3$ ist 0, **9/0** resultiert in **Laufzeitfehler**

Typ Umwandlungen

- **Ausser impliziten Umwandlungen gibt es die Möglichkeit der expliziten Umwandlung**
 - implizit (ein Operand ist `double` – Umwandlung des anderen): `1.0 / 4`
 - explizite (erzwungene) Umwandlungen heissen «cast» oder «type cast»
- ***type cast*: Umwandlung von einem Typ in den anderen.**
 - Um `int` in `double` umzuwandeln (z.B. damit Division das gewünschte exakte Ergebnis liefert)
 - Um einen `double` Wert in einen `int` Wert zu verwandeln
 - Abschneiden («truncation») – nicht Rundung

Typ Umwandlungen

- **Syntax:**

(type) expression

- **Beispiele:**

<code>(double) 19 / 5;</code>	<code>// 3.8</code>
<code>(int) ((double) 19 / 5) ;</code>	<code>// 3</code>

Typ Umwandlungen

- **(type) ist ein Operator**
 - Der «cast operator»
 - Höherer Rang (Präzedenz) als arithmetische Operatoren
 - Rechts-assoziativ
 - D.h. wandelt nur den Operanden direkt rechts daneben um
- **Für alle Basistypen verfügbar**

Bemerkungen über explizite Umwandlungen

- Typ Umwandlung hat hohe Präzedenz und der Cast Operator bezieht sich nur auf den (Teil)Ausdruck direkt neben sich.
 - `(double) 1 + 1 / 2;` `// 1.0`
 (double) 1 *1* */ 2*
 - `1 + (double) 1 / 2;` `// 1.5`
 1 *1.0* *(double) 1* */ 2* *0.5*
- Mit Klammern kann man die Reihenfolge der Auswertung verändern.
 - `(double) (2 + 1 + 1) / 3;` `// 1.3333333333333333`

Rang Ordnung ...

- Die letzten Slides haben die Java Sicht gezeigt.
- Andere Programmiersprachen haben (eventuell) andere Regeln
 - Notwendigkeit von Typ Information
 - Rang Ordnung sollte festgelegt sein
- Nicht immer was man erwartet
 - Beispiel: Microsoft Excel

<div> <div> Excel </div> <div> File Edit View Insert Format Tools Data Window Help </div> </div>			
<div> <div> </div> <div> </div> </div>			
<div> <div>Home</div> <div>Insert</div> <div>Page Layout</div> <div>Formulas</div> <div>Data</div> <div>Review</div> <div>View</div> </div>			
SUM	<div> <div> </div> <div> fx - 2 ^ 2 </div> </div>		
	A	B	
1	- 2 ^ 2		
2			
3			
4			

Excel File Edit View Insert Format Tools Data Window Help			
Home Insert Page Layout Formulas Data Review View			
A2	fx		
	A	B	
1	4		
2			
3			
4			

Excel File Edit View Insert Format Tools Data Window Help

Home Insert Page Layout Formulas Data Review View

Paste Cut Copy Format

Calibri (Body) 12 A A

B I U



Wrap Text Merge & Center


SUM \times \checkmark fx $= 0 - 2 ^ 2$

	A	B
1	$= 0 - 2 ^ 2$	
2		
3		
4		

Excel File Edit View Insert Format Tools Data Window Help				
Home Insert Page Layout Formulas Data Review View				
<div> <div> <div>Paste</div> <div>Cut Copy Format</div> </div> <div> <div>Calibri (Body) 12 A A</div> <div>B I U</div> </div> <div> <div>= = =</div> <div>≡ ≡ ≡</div> </div> <div> <div>Wrap Text</div> <div>Merge & Center</div> </div> </div>				
A2	fx			
	A	B		
1		-4		
2				
3				
4				

Go back one page
Pull down to show history

  <https://support.microsoft.com/en-us/search?query=q132%2F6%2F86.asp>

 Microsoft Office Windows Surface Xbox Deals Support More

Microsoft Support

Contact us

[Search all of Microsoft](#)

Search results for "q132/6/86.asp." Found 1 results.

support.microsoft.com

<https://support.microsoft.com/support/kb/articles/q132/6/86.asp>

We would like to show you a description here but the site won't allow us.

Operator precedence

If you combine several operators in a single formula, Excel performs the operations in the order shown in the following table. If a formula contains operators with the same precedence — for example, if a formula contains both a multiplication and division operator — Excel evaluates the operators from left to right.

Operator	Description
: (colon) (single space) , (comma)	Reference operators
–	Negation (as in –1)
^	Exponentiation
* and /	Multiplication and division
+ and –	Addition and subtraction

Operationen und Typen

- **Basistypen («primitive types»)**

- Beispiele: `int` und `long`
 - Haben Werte (4, 1, 0, -218), Operationen und Variable gesehen
- Beispiel: `double`
 - Haben Werte (0.0, -5.1, 4.3) und Operationen gesehen

- **Typen aus der Java Bibliothek**

- Beispiel: `String`
 - Haben Werte gesehen ("`hello`", "`2 + 3`")
 - Operationen durch Methoden ("`hello`".`toUpperCase()`)
 - Operationen mit Operator (*jetzt*)

- **Selbst entwickelte Typen (später)**

String Operationen

- **String Verkettung («concatenation»):** Der Operator `+` erlaubt es, zwei (oder mehr) Strings zu verketten (verknüpfen)

`"hello" + "world"` ergibt `"helloworld"`

`"a" + "b" + "c"` ergibt `"abc"`

- **Ausdrücke können Operanden verschiedener Typen kombinieren**
 - Auch Strings mit anderen Werten/Variablen/Ausdrücken

String Operationen

- **Java Regel: Für jeden Wert gibt es eine Darstellung als String**
 - Gilt für alle Typen (Basistypen, Bibliothek, selbst entwickelte Typen)
 - Ob die *default* Darstellung (D: voreingestellte *oder* standardmässig eingestellte Darstellung) sinnvoll/verständlich ist – eine andere Frage
 - Trotzdem sehr praktisch: Kann jeden Wert `W` mittels `print(W)` ausgeben

Darstellung als String

- Für Basistypen ist die default Darstellung die erwartete Darstellung
- Für `int` oder `long` Variable und Konstanten
 - `int Literal(Konstante) 42` Default Darstellung: `"42"`
 - `int j = 99` Default Darstellung für `j` : `"99"`
- Für `double` Variable genauso
- Das selbe gilt für `String`
 - `"Hello"` Default Darstellung: `"Hello"`

String Operationen

- **Java Regel: Für jeden Wert gibt es eine Darstellung als String**
 - Gilt für alle Typen (Basistypen, Bibliothek, selbst entwickelte Typen)
 - Ob die *default* Darstellung (D: voreingestellte *oder* standardmässig eingestellte Darstellung) sinnvoll/verständlich ist – eine andere Frage
 - Trotzdem sehr praktisch: Kann jeden Wert `W` mittels `print(W)` ausgeben
- **Der Operator `+` mit einem String und einem anderen Wert als Operanden ergibt einen längeren String**
 - Verwendet die default Darstellung

Operator + mit String Operand

- **String + anderer Wert als Operand: verknüpfe String mit default Darstellung des Operanden zu neuem String**

`"hello" + 42` ergibt `"hello42"`

`"hello" + j` ergibt `"hello99"` // `j` ist 99

- **Nur der Operator + ist erlaubt (kein *, /, oder -)**

Die üblichen Regeln ...

- **Von links nach rechts rechnen**

- $12 - 3 + 5$ ergibt ?

- $2 + 3 + \text{"Hello"}$ ergibt ?

- $\text{"Hello " } + 2 + 3$ ergibt ?

Die üblichen Regeln ...

- **Von links nach rechts rechnen**

- $12 - 3 + 5$ ergibt 14 (nicht 4)

- $2 + 3 + \text{"Hello"}$

- $\text{"Hello " } + 2 + 3$

Die üblichen Regeln ...

- **Von links nach rechts rechnen**

- $12 - 3 + 5$ ergibt 14 (nicht 4)
- $2 + 3 + \text{"Hello"}$ ergibt "5 Hello"
- $\text{"Hello " } + 2 + 3$

Die üblichen Regeln ...

- **Von links nach rechts rechnen**

- $12 - 3 + 5$ ergibt 14 (nicht 4)
- $2 + 3 + \text{"Hello"}$ ergibt "5 Hello"
- $\text{"Hello "} + 2 + 3$ ergibt "Hello 23"

- **Punkt vor Strich**

- $\text{"Hello "} + 2 * 3$

Die üblichen Regeln ...

- **Von links nach rechts rechnen**

- `12 - 3 + 5` ergibt 14 (nicht 4)
- `2 + 3 + " Hello"` ergibt `"5 Hello"`
- `"Hello " + 2 + 3` ergibt `"Hello 23"`

- **Punkt vor Strich**

- `"Hello " + 2 * 3` ergibt `"Hello 6"`

String Operationen Quiz

Poll

1 + 2 + "abc"

"abc" + 9 * 3

"1" + 1

4 - 1 + "abc"

Weitere String Operationen

`1 + 2 + "abc"` ergibt `"3abc"`

`"abc" + 9 * 3` ergibt `"abc27"`

`"1" + 1` ergibt `"11"`

`4 - 1 + "abc"` ergibt `"3abc"`

String Operationen

- Können + verwenden um einen String und den Wert eines Ausdrucks auszugeben.

```
System.out.println("Note: " + (4.8 + 5.4) / 2);
```

Output:
Note: 5.1

- Warum brauchen wir (und)?

2.2.3 Variablen

Übersicht

- 2.0 Einfache Java Programme
- 2.1 Methoden
 - Struktur
- **2.2 Typen und Variable**
 - 2.2.1 Einführung
 - 2.2.2 Basistypen: Einfache (eingebaute) Typen
 - 2.2.3 Deklaration von Variablen
Zuweisungen

Variable

- **Variable («variable»):** Name der es erlaubt, auf einen gespeicherten Wert zuzugreifen

Variable

- **Variable («variable»):** Name der es erlaubt, auf einen gespeicherten Wert zuzugreifen
- **Wert muss (irgendwo) vom Programm gespeichert werden**
 - In Speicherzelle(n) im Computer
 - Ähnlich Schnellwahl (Telefon)



Variable

- **Variable («variable»):** Name der es erlaubt, auf einen gespeicherten Wert zuzugreifen
- **Wert muss (irgendwo) vom Programm gespeichert werden**
 - In Speicherzelle(n) im Computer
 - Ähnlich Schnellwahl (Telefon)



Variable

- **Wie man diese Wahlknöpfe benutzt:**

- Einrichten (in der Fabrik)
- Konfiguration
- Einsatz



- **Ähnlicher Ablauf für Variablen in einem Programm**

- *Deklaration* - gibt Namen und Typ an
- *Initialisierung* - speichert einen Wert in der Variablen
- *Gebrauch* - in einem Ausdruck oder println Anweisung

Deklaration

- **Deklaration einer Variablen: Reserviert Speicher für Werte**
 - Variable müssen deklariert sein bevor sie verwendet werden können
 - Fürs erste: in einer Methode

- **Syntax:** ***type name;***

name ist ein *Bezeichner* («*identifizier*»)

type ist der Typ

```
int x;
```

```
double meineNote;
```



- **Auch mit Initialisierung (Definition)**

```
int z = 99;
```


Arbeiten mit einer Variablen

- **Variable muss deklariert sein bevor sie im Programm gebraucht wird**

- Als Operand einer Zuweisung («assignment»)

```
int z;
```

```
z = 1;
```

- **Variable muss Wert haben bevor sie als Operand anderer Operatoren gebraucht werden kann**

- Wert kann von «Deklaration mit Initialisierung» oder Zuweisung kommen

Compiler Fehler Meldungen

- Eine Variable kann erst *nach* einer Zuweisung verwendet werden.

- `int x;`

- `System.out.println(x); // ERROR: x has no value`

- Keine Doppeldeklarationen.

- `int x;`

- `int x; // ERROR: x already exists`

- `int x = 3;`

- `int x = 5; // ERROR: x already exists`

Compiler Fehler Meldungen

- Keine Doppeltdекларationen.

- `int x;`
`long x;` // ERROR: x already exists

- `int x = 3;`
`double x = 3.0;` // ERROR: x already exists

Zuweisungen («Assignment»)

- **Zuweisung: Anweisung die Wert in einer Variable speichert.**

- Wert kann ein Ausdruck sein, die Zuweisung speichert das Ergebnis
- Fürs erste: Zuweisungen in einer Methode

- **Syntax: *name = expression;***

- `int x;`

- `x = 3;`

- `double meineNote;`

- `meineNote = 3.0 + 2.25;`

x	3
---	---

meineNote	5.25
-----------	------

Gebrauch von Variablen

- Wiederholte Zuweisungen sind erlaubt:

```
int x;  
x = 3;  
System.out.println(x + " here");    // 3 here  
x = 4 + 7;  
System.out.println("now x is " + x); // now x is 11
```

x	11
---	----

Assignment

- Java verwendet `=` um eine Zuweisung auszudrücken
 - «Assignment operator»

name = expression;

EBNF Description *assignment*

variableidentifizier \Leftarrow *bezeichner*

assignment \Leftarrow *variableidentifizier* `=` *expression* ;

Die EBNF Beschreibung für *expression* hatten wir schon gesehen.

Deklaration mit Initialisierung

EBNF Description *variabledeclaration*

typeidentifier \Leftarrow *bezeichner*

variableidentifier \Leftarrow *bezeichner*

variablelist \Leftarrow *variableidentifier* { , *variableidentifier* }

variableinitialization \Leftarrow *variableidentifier* = *expr*

variablespecification \Leftarrow *variableinitialization* | *variablelist*

variabledeclaration \Leftarrow *typeidentifier* *variablespecification* ;

Zuweisung (Programm) und Algebra (Mathematik)

- Zuweisung verwendet = , aber eine Zuweisung ist keine algebraische Gleichung!

= bedeutet: «speichere den Wert der RHS in der Variable der LHS»

Die rechte Seite des Ausdrucks wird zuerst ausgewertet,
dann wird das Ergebnis in der Variable auf der linken Seite gespeichert

- Was passiert hier?

```
int x = 3;
```

```
x = x + 2;    // ???
```

x	5
---	---

Was passiert hier ?

Poll

Sehen Sie sich diese Anweisungen an. Welchen Wert hat **x** am Ende?

```
int x;  
int y;  
x = 0;    // (S1)  
y = 1;    // (S2)  
x = y;    // (S3)  
y = 2;    // (S4)
```

x hat den Wert ???

Was passiert hier ?

Poll

Sehen Sie sich diese Anweisungen an. Welchen Wert hat **x** am Ende?

```
int x;  
int y;  
x = 0;    // (S1)  
y = 1;    // (S2)  
x = y;    // (S3)  
y = 2;    // (S4)
```

x hat den Wert ???

1

Was passiert hier ?

Poll

Sehen Sie sich diese Anweisungen an. Welchen Wert hat **x** am Ende?

```
int x;  
int y;  
x = 0;    // (S1)  
y = 1;    // (S2)  
x = y;    // (S3)  
y = 2;    // (S4)  
x hat den Wert ???
```

Nach Var	S1	S2	S3	S4
x	0	0	1	1
y	?	1	1	2

1

aus dem Archiv

Welchen Wert hat x am Ende?

• -1	(0% 0)
• 0	(1% 3)
• 1	(90% 239)
• 2	(7% 21)
• 3	(0% 0)
• Keinen dieser Werte bzw Wert undefiniert	(0% 0)

Übersicht

- **2.3 Aussagen über Programm(segment)e**

 - Vorwärts/Rückwärts schliessen

 - 2.3.1 Pre- und Postconditions

 - 2.3.2 Hoare-Tripel für Anweisungen

- **2.4 Verzweigungen**

2.3 Aussagen über Programm(segment)e

- **Beispiel Programm(segment)**

```
public static void main(String[] args) {  
    int laenge = 70;  
    int hoehe = 40;  
    int tiefe = 27;  
    int flaeche = 2*(laenge*tiefe + tiefe*hoehe + laenge*hoehe);  
    ...  
}
```

- **Wir wollen eine Aussage über die Berechnung machen**

- Muss Variable flaeche involvieren

Aussagen über Programm(segment)e

- Aussage über Variable `flaeche` muss an Ort (Stelle im Programm) gekoppelt sein

The diagram illustrates the placement of an assertion regarding the variable `flaeche` within a Java `main` method. It shows four lines of code: `int laenge = 70;`, `int hoehe = 40;`, `int tiefe = 27;`, and `int flaeche = 2*(laenge*tiefe + tiefe*hoehe + laenge*hoehe);`. To the left of the first three lines, there are callouts that say "Keine Aussage möglich" (No assertion possible). To the left of the fourth line, there is a callout that says "Aussage möglich" (Assertion possible). Below the code, there are three more callouts: "Keine Aussage möglich", "Aussage möglich", and "Aussage möglich".

```
public static void main(String[] args) {  
    int laenge = 70;  
    int hoehe = 40;  
    int tiefe = 27;  
    int flaeche = 2*(laenge*tiefe + tiefe*hoehe +  
                    laenge*hoehe);  
    ...  
}
```

Keine Aussage möglich

Keine Aussage möglich

Keine Aussage möglich

Keine Aussage möglich

Aussage möglich

...

Aussage möglich

Ziel: Aussagen über ein Programm machen

- **Zuerst: welche Aussagen gelten für eine Methode**
 - Erlaubte Anweisungen in der Methode:
 - Zuweisungen
- **Also zuerst Programm(segment) ist Rumpf (Body) einer Methode**

(Logische) Aussagen

- **Aussage («assertion»): Eine Behauptung die entweder wahr oder falsch ist.**
 - Wir fragen dann oft «Ist die Aussage wahr»?
- **Beispiele:**
 - Zürich ist ein Kanton der Schweiz
 - Stockholm ist die Hauptstadt Norwegens
 - 11 ist eine Primzahl
 - 120 ist kleiner als 11
 - $x \geq 0$ (*hängt von x ab*)
 - x geteilt durch 2 ergibt 8 (*hängt von x ab*)

(Logische) Aussagen

- **Nicht alle Aussagen sind wahr**
 - Für einige Aussagen können wir vielleicht nicht sofort entscheiden ob sie wahr oder falsch sind
- **Wichtig ist dass es Sinn macht zu fragen, ob die Aussage wahr oder falsch ist**
- **Logisch heisst hier: im Sinne der klassischen Logik**

Ziel: Aussagen über ein Programm herleiten

- **Welche Aussagen gelten (an einer Stelle) im Programm?**
 - Ggf. was für *Annahmen* sind nötig, dass eine Aussage P wahr ist
- **Was heisst «an einer Stelle»?**

Arbeiten mit Aussagen

- **Wir stellen uns vor, Positionen (Punkte) im Code haben einen Namen**
 - Point A
 - Point B
 - ...
- **Alle Anweisungen, die davor (im Programm) erscheinen, sind ausgeführt wenn wir diesen Punkt (während der Ausführung) erreichen**
- **Keine Anweisung danach wurde ausgeführt**

Hoare Logik

- **Tony Hoare entwickelte in den (19)70gern einen Ansatz wie man über Programme logische Schlüsse ziehen kann.**
- **Schritt 1: Vorwärts und rückwärts schliessen**
- **Schritt 2: Von einer Anweisung zu mehreren Anweisungen und Blöcken**

Warum

- Ziel ist es, dass Sie für ein einfaches Programm genau argumentieren können.
- In der täglichen Programmentwicklung genügt es oft, weniger detailliert zu argumentieren als es die Hoare Logik erforderlich macht.
 - Für einfache Programme ist dieser Ansatz zu aufwändig.
 - Für realistische Programme wird der Ansatz schnell kompliziert.
 - Wir haben immer noch kein gutes Modell für Objekte und Parallelismus. Aliasing ist eine Herausforderung.
 - Aber manchmal hilft der Ansatz denn ...

- **Eine gute Schulung, systematisch zu programmieren**
 - Wir können Aussagen machen über Zustände (der Ausführung) eines Programms (und später auch eines Objekts)
 - Wir können den Effekt eines Programms beschreiben
 - Wir können definieren was es heisst dass eine Aussage «weaker» (schwächer) oder «stronger» (stärker) ist.
- **Wichtig für die Definition von Schnittstellen (zwischen Modulen) wenn wir entscheiden müssen welche Bedingungen erfüllt sein müssen (um eine Methode aufzurufen).**

Was für Aussagen brauchen wir?

- **Wie finden wir Aussagen für Stellen (Punkte) im Programm?**
 - Uns interessieren nicht irgendwelche Aussagen sondern solche, die das Verhalten der Ausführung beschreiben
- **Alle Programm(segment)e arbeiten mit int Variablen und Werten**
 - Wir nehmen an (oder wissen) dass die endliche Darstellung kein Problem ist
 - Alle Ergebnisse können korrekt dargestellt werden

Beispiel

- **Vorwärts schliessen**

- Vom Zustand *vor* der Ausführung eines Programm(segments)
- Nehmen wir an wir wissen (oder vermuten) $w > 0$

```
// w > 0
```

```
x = 17;
```

```
//
```

```
y = 42;
```

```
//
```

```
z = w + x + y;
```

```
//
```

Beispiel

■ Vorwärts schliessen

- Vom Zustand *vor* der Ausführung eines Programm(segments)
- Nehmen wir an wir wissen (oder vermuten) $w > 0$

$// w > 0$

$x = 17;$

$// w > 0 \wedge x \text{ hat den Wert } 17$

$y = 42;$

$// w > 0 \wedge x \text{ hat den Wert } 17 \wedge y \text{ hat den Wert } 42$

$z = w + x + y;$

$// w > 0 \wedge x \text{ hat den Wert } 17 \wedge y \text{ hat den Wert } 42 \wedge z > 59$

- Jetzt wissen wir einiges mehr über das Programm, u.a. $z > 59$

Was sehen wir?

- Folge von Statements: hier Zuweisungen
- Vor und nach jeder Zuweisung eine Aussage
 - Als Java Kommentar

```
// w > 0  
x = 17;
```

- Der Kommentar ist logische Aussage, *kein* Java Ausdruck oder Anweisung

Beispiel

- **Rückwärts schliessen:**

- Nehmen wir an wir wollen dass z nach Ausführung negativ ist

```
//  
x = 17;  
//  
y = 42;  
//  
z = w + x + y;  
// z < 0
```

Beispiel

- **Rückwärts schliessen:**

- Nehmen wir an wir wollen dass z nach Ausführung negativ ist

`// w + 17 + 42 < 0`

`x = 17;`

`// w + x + 42 < 0`

`y = 42;`

`// w + x + y < 0`

`z = w + x + y;`

`// z < 0`

Anderer Blickwinkel

- **Wie erreichen wir ein gewünschtes Resultat**
 - Den gewünschten Zustand nach der Ausführung von Anweisungen

Beispiel

- **Rückwärts schliessen:**

- Nehmen wir an wir wollen dass z nach Ausführung negativ ist

`// w + 17 + 42 < 0`

`x = 17;`

`// w + x + 42 < 0`

`y = 42;`

`// w + x + y < 0`

`z = w + x + y;`

`// z < 0`

- Dann müssen wir wissen (oder vermuten) dass vor der Ausführung gilt: $w < -59$
 - Notwendig und hinreichend

Vorwärts vs Rückwärts, Teil 1

- **Vorwärts schliessen:**

- Bestimmt was sich aus den ursprünglichen Annahmen herleiten lässt.
- Sehr praktisch wenn eine *Invariante* gelten soll

- **Rückwärts schliessen:**

- Bestimmt hinreichende Bedingungen die ein bestimmtes Ergebnis garantieren
 - Wenn das Ergebniss erwünscht ist, dann folgt aus den Bedingungen die Korrektheit.
 - Ist das Ergebniss unerwünscht, dann reichen die Bedingungen um einen Bug (oder einen Sonderfall) zu generieren

Vorwärts vs Rückwärts, Teil 2

- **Vorwärts schliessen:**
 - Simuliert die Ausführung des Programms (für viele «Inputs» «gleichzeitig»)
 - Oft leicht zu verstehen, erscheint «natürlich»
 - Aber führt dazu dass (viele) Details festgehalten werden, die letztlich irrelevant sind.

Vorwärts vs Rückwärts, Teil 3

- **Rückwärts schliessen:**

- Oft von grossem praktischen Nutzen: Sie müssen verstehen (oder festhalten) was jede Anweisung zum Erreichen eines bestimmten Zustandes beiträgt.
- Ein Programm(segment) «rückwärts» zu lesen erfordert Übung aber führt zu einer neuen Sicht auf ein Programm.

2.3.1 Pre- und Postconditions

Pre- und Postconditions

- **Precondition: notwendige Vorbedingungen die erfüllt sein müssen (vor Auszuführung einer Anweisung)**
- **Postcondition: Ergebnis der Ausführung (wenn Precondition erfüllt)**

Pre- und Postconditions

- **Pre- und Postconditions für eine Anweisung**
 - Wenn vor der Ausführung der Anweisung die Precondition erfüllt ist, dann gilt nach der Ausführung die Postcondition, *oder*
 - Damit nach der Ausführung die Postcondition gilt, muss vor der Ausführung die Precondition erfüllt sein
 - Precondition, Anweisung und Postcondition hängen zusammen!
- **Werden wir für Folgen von Anweisungen erweitern**

Pre- und Postconditions

- **Pre- und Postconditions für eine Anweisung**
 - Wenn vor der Ausführung der Anweisung die Precondition erfüllt ist, dann gilt nach der Ausführung die Postcondition, *oder*
 - Damit nach der Ausführung die Postcondition gilt, muss vor der Ausführung die Precondition erfüllt sein
 - Precondition, Anweisung und Postcondition hängen zusammen!
- **Werden wir für Folgen von Anweisungen erweitern**

Terminologie

- Die Annahme (Aussage), die vor der Ausführung eines Statements gilt, ist die *Precondition*.
- Die Aussage, die nach der Ausführung gilt (unter der Annahme dass die Precondition gültig ist und das Statement ausgeführt wurde), ist die *Postcondition*.
 - Wenn wir diesen Punkt erreichen dann gilt die Postcondition
 - Wenn wir diesen Punkt *nicht* erreichen (z.B. wegen eines Laufzeitfehlers) *dann machen wir keine Aussage*

Pre/Postconditions

- Aussagen (Pre/Postconditions) sind logische (bool'sche) Ausdrücke die sich auf den Zustand (der Ausführung) eines Programms beziehen.
- Der Zustand eines Programms ist sichtbar durch die Namen der Variablen (und jede Variable liefert ihren Wert)
 - Die Variablennamen können in (logischen) Ausdrücken verwendet werden – solange die Ausdrücke *keine Nebenwirkungen* («*no side-effects*») haben
 - Nebenwirkungen wurden noch nicht diskutiert: das ist eine Warnung an Studierende mit Vorkenntnissen

Die übliche Notation

- Statt die Pre/Postconditions in Kommentaren (nach //) anzugeben verwenden viele Texte {...}

- Kein Java mehr ...

- Aber diese Schreibweise hat sich eingebürgert, lange vor Java

→ { $w < -59$ }
x = 17;

⇒ { $w + x < -42$ }

x < y ✓
y < z ✓
~~Java~~

- Zwischen { und } steht eine logische Aussage

- Kein Java Code (aber wir verwenden Java's Operatoren oder die aus der Mathematik bekannten)

assert-Statement in Java

- Eine Aussage («Assertion», Behauptung) in Java kann durch ein assert-Statement ausgedrückt werden.
- Syntax:

`assert expression;`
- *expression* ist ein logischer Ausdruck (nach den Java Regeln)
 - Also ohne «{» und «}»
 - Beispiel: `assert x>0;`

assert-Statement in Java

- Wenn zur Laufzeit *expression* nicht gültig ist, dann wird ein Laufzeitfehler generiert
- Da { und } (fast) überall in Texten verwendet werden verwenden wir diese Zeichen
 - ... und die Hoare Logik kann nicht nur für Java verwendet werden

2.3.2 Hoare Tripel (oder 3-Tupel)

Hoare Tripel (oder 3-Tupel)

- Ein *Hoare Tripel* besteht aus zwei Aussagen und einem Programmsegment:

$$\{P\} \ S \ \{Q\}$$

- P ist die Precondition
- S das Programmsegment (bzw Statement)
- Q ist die Postcondition

Hoare Tripel (oder 3-Tupel)

- Ein *Hoare Tripel* $\{P\} \ S \ \{Q\}$ ist *gültig* wenn (und nur wenn):
 - Für jeden Zustand, für den P gültig ist, ergibt die Ausführung von S immer einen Zustand für den Q gültig ist.
 - Informell: Wenn P wahr ist vor der Ausführung von S , dann muss Q nachher wahr sein.
- Andernfalls ist das Hoare Tripel *ungültig*.

Überblick

- **Wenn P wahr ist vor der Ausführung von S , dann muss Q nachher wahr sein.**
 - Bisher «informell»
- **Für jedes Java Statement gibt es genaue Regeln die eine Precondition und eine Postcondition in Beziehung setzen**
 - Regel für Zuweisungen
 - Regel für zwei aufeinander folgende Anweisungen
 - Regel für «if»-Statements
 - [später:] Regel für Schleifen