Teil I

Konzepte imperativer Programmierung

Abschnitt 2: Imperative Programmierung

2. Imperative Programmierung
2.1 Einleitung
2.2 Grunddatentypen und Ausdrücke
2.3 Imperative Variablenbehandlung
2.4 Anweisungen, Blöcke und Gültigkeitsbereiche
2.5 Klassenvariablen
2.6 Reihungen
2.7 (Statische) Methoden
2.8 Kontrollstrukturen
2.9 ... putting the pieces together ...

Überblick

2. Imperative Programmierung
2.1 Einleitung
2.2 Grunddatentypen und Ausdrücke
2.3 Imperative Variablenbehandlung
2.4 Anweisungen, Blöcke und Gültigkeitsbereiche
2.5 Klassenvariablen
2.6 Reihungen
2.7 (Statische) Methoden
2.8 Kontrollstrukturen
2.9 ... putting the pieces together ...

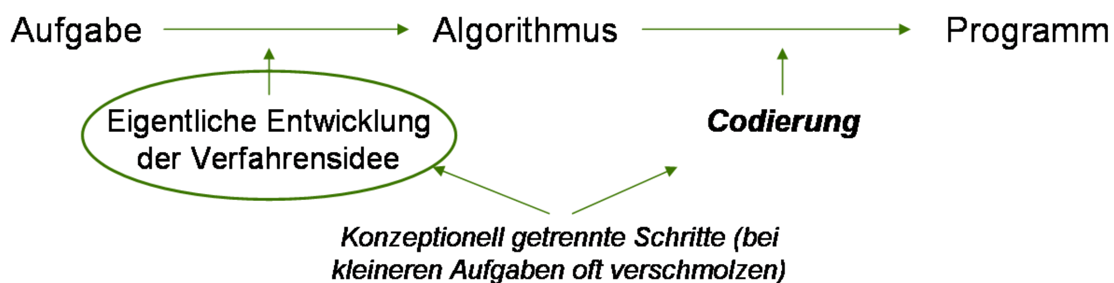
Was macht eigentlich ein Informatiker?

Informatik: Wissenschaft und Technik der Verarbeitung von Informationen mit Hilfe elektronischer Rechenanlagen (Computer)

Algorithmus: Menge von Verarbeitungsvorschriften, die von einem Computer ausgeführt werden können

Zentrale Aufgabe des Informatikers: Entwicklung von Algorithmen (und oft auch deren Realisierung auf dem Rechner als Programm)

Programm: Formale Darstellung eines Algorithmus (oder mehrerer) in einer Programmiersprache



Beispiel: Algorithmus *Strecke*

- **Aufgabe:** Ein frei beweglicher Körper mit der Masse m (> 0) werde aus der Ruhelage eine Zeit t lang mit einer auf ihn einwirkenden konstanten Kraft k bewegt. Gesucht ist ein Algorithmus, der (in Abhängigkeit von m, t und k) die Strecke s bestimmt, um die der Körper aus seiner ursprünglichen Lage fortbewegt wird.
- **Lösungsidee:** Es gelten folgende einfache physikalische Gesetze
  - die auf den Körper einwirkende Kraft erzeugt eine konstante Beschleunigung
    $$b = \frac{k}{m}.$$
  - der in der Zeit t zurückgelegte Weg bei einer Bewegung mit konstanter Beschleunigung b ist
    $$s = \frac{1}{2} \cdot b \cdot t^2.$$
  - durch Einsetzen erhält man

$$\mathit{strecke}(m, t, k) = \frac{k \cdot t \cdot t}{2 \cdot m}$$

Beispiel: Algorithmus *Wechselgeld*

- **Aufgabe:** Ein Kunde kauft Waren für $1 \leq r \leq 100$ EUR und bezahlt mit einem 100 EUR Schein (r sei ein voller EUR Betrag ohne Cent-Anteil). Gesucht ist ein Algorithmus, der zum Rechnungsbetrag r das Wechselgeld w bestimmt. Zur Vereinfachung nehmen wir an, dass w nur aus 1 EUR oder 2 EUR Münzen oder 5 EUR Scheinen bestehen soll. Es sollen möglichst wenige Münzen/Scheine ausgegeben werden (also ein 5 EUR Schein statt fünf 1 EUR Münzen).
- **Lösungsidee:** Die meisten "menschlichen Wechsler" führen die folgenden Schritte der Reihe nach aus:
  - Setze $w = \emptyset$.
  - **Falls** die letzte Ziffer von r eine 2,4,7 oder 9 ist,
    dann erhöhe r um 1 und nimm 1 zu w hinzu.
  - **Falls** die letzte Ziffer von r eine 1 oder 6 ist,
    dann erhöhe r um 2 und nimm 2 zu w hinzu.
  - **Falls** die letzte Ziffer von r eine 3 oder 8 ist,
    dann erhöhe r um 2 und nimm 2 zu w hinzu.
  - **Solange** $r < 100$: erhöhe r um 5 und nimm 5 zu w hinzu.

Grundidee der funktionalen Programmierung

- Algorithmen sind Abbildungen (*Funktionen*) zwischen Mengen von Objekten
- Ergebnis eines funktionalen Algorithmus wird durch *Anwendung* dieser Abbildung auf die Eingabeobjekte erzeugt
- Beispiel: Der Algorithmus *Strecke* wird dargestellt durch die Funktion $\mathit{strecke} : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ mit

$$\mathit{strecke}(m,t,k) = \frac{k \cdot t \cdot t}{2 \cdot m}$$

- Umsetzung in einer (funktionalen) Programmiersprache (z.B. SML):

```
fun strecke(m,t,k) = (k * t * t) / (2.0 * m)
```


Grundidee der funktionalen Programmierung (cont.)

- Beispiel: Der Algorithmus *Wechselgeld* wird dargestellt durch die (rekursiv definierte) Funktion $wg : \mathbb{N} \rightarrow SET(\mathbb{N})$ mit

$$wg(r) = \begin{cases} \emptyset & \text{falls } r = 100 \\ \{5\} \cup wg(r+5) & \text{falls } 100 - r \geq 5 \\ \{2\} \cup wg(r+2) & \text{falls } 5 > 100 - r \geq 2 \\ \{1\} \cup wg(r+1) & \text{falls } 2 > 100 - r \geq 1 \end{cases}$$

- Umsetzung in einer (funktionalen) Programmiersprache (z.B. SML):

```
fun wg(r) = if r = 100 then []
       else if 100 - r >= 5 then 5 :: (wg(r + 5))
       else if 5 > 100 - r andalso 100 - r >= 2
                                then 2 :: (wg(r + 2))
       else (* 2 > 100 - r andalso 100 - r >= 1 *)
            1 :: (wg(r + 1));
```

Grundidee der imperativen Programmierung

- Algorithmen bestehen aus zeitlicher Abfolge von Berechnungsschritten.
- Das Ergebnis eines imperativen Algorithmus wird durch Abarbeiten dieser Verarbeitungsschritte erreicht.
- Beispiel: Algorithmus *Strecke* wird dargestellt durch folgende Berechnungsschritte

  Schritt 1 $b = \frac{k}{m}$, d.h. berechne $\frac{k}{m}$ (aus den Eingabewerten k und m) und bezeichne das Ergebnis mit b;

  Schritt 2 Der Ergebniswert für Eingabewerte m, t, k ist $\frac{1}{2} \cdot b \cdot t^2$;

Funktionale und imperative Algorithmen

Grundidee der imperativen Programmierung (cont.)

- Beispiel: Algorithmus *Wechselgeld* wird dargestellt durch folgende Berechnungsschritte

  Schritt 1 Setze $w = \emptyset$;

  Schritt 2 **Falls** die letzte Ziffer von r eine 2,4,7 oder 9 ist **dann** erhöhe r um 1 und nimm 1 zu w hinzu;

  Schritt 3 **Falls** die letzte Ziffer von r eine 1 oder 6 ist **dann** erhöhe r um 2 und nimm 2 zu w hinzu;

  Schritt 4 **Falls** die letzte Ziffer von r eine 3 oder 8 ist **dann** erhöhe r um 2 und nimm 2 zu w hinzu;

  Schritt 5 **Solange** $r < 100$: erhöhe r um 5 und nimm 5 zu w hinzu;

  Schritt 6 Gib w als Ergebnismenge aus;

Funktionale und imperative Algorithmen

Beobachtungen:

- Es gibt Algorithmen, deren Intuition einer funktionalen Auffassung des Problems entsprechen, d.h. das Ergebnis wird durch Anwenden einer Funktion auf die Eingabeobjekte erzeugt.
  Beispiel: Algorithmus *Strecke*
- Es gibt Algorithmen, deren Intuition einer imperativen Auffassung des Problems entsprechen, d.h. das Ergebnis wird durch sukzessives Abarbeiten einzelner Verarbeitungsschritte erzeugt.
  Beispiel: Algorithmus *Wechselgeld*

Bemerkungen:

- Je nach Lösungsansatz (Problem-Auffassung) empfiehlt sich, einen funktionalen oder imperativen Algorithmus zu entwerfen.
- Die meisten Programmiersprachen (insbesondere auch SML und Java) bieten Konzepte für beide Programmierparadigmen an.

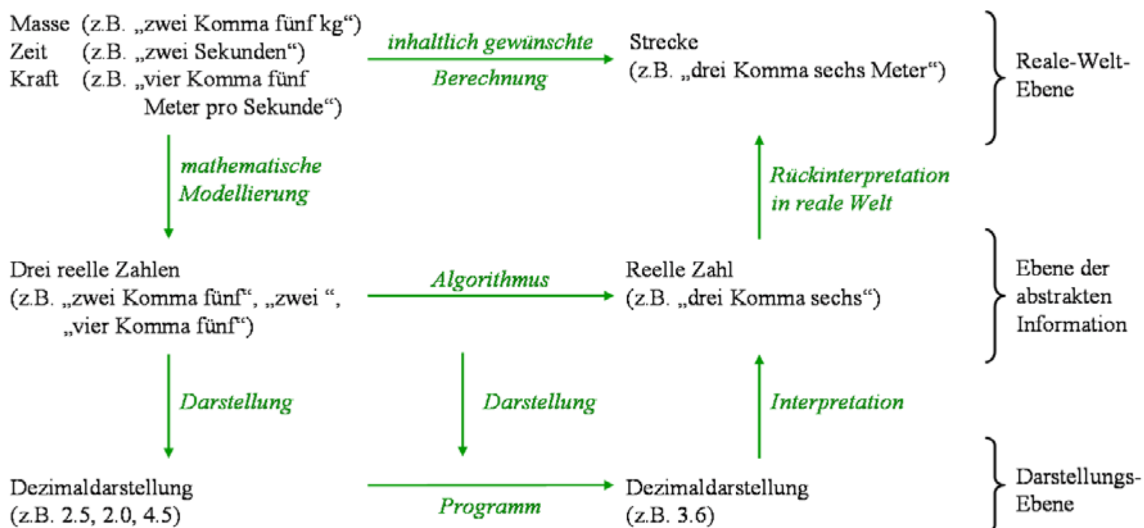
Überblick

2. Imperative Programmierung
2.1 Einleitung
2.2 Grunddatentypen und Ausdrücke
2.3 Imperative Variablenbehandlung
2.4 Anweisungen, Blöcke und Gültigkeitsbereiche
2.5 Klassenvariablen
2.6 Reihungen
2.7 (Statische) Methoden
2.8 Kontrollstrukturen
2.9 ... putting the pieces together ...

Mathematische Abstraktion zur Datenverarbeitung

- Wie werden die zu verarbeitenden Daten eigentlich dargestellt/modelliert?
- Beispiel: Algorithmus *Strecke*
  - Eingabe: Masse m, Zeit t, Kraft k
    z.B. “zwei Komma fünf Kilogramm”, “zwei Sekunden”, “vier Komma fünf Meter pro Sekunde”
  - Mathematische Abstraktion: drei reelle Zahlen $m, t, k \in \mathbb{R}$
    z.B. “zwei Komma fünf”, “zwei Komma null” und “vier Komma fünf”
  - Darstellung: Dezimaldarstellung
    z.B. 2.5, 2.0, 4.5
  - Ergebnis: Strecke s
    z.B. 3.6 (Dezimaldarstellung)
    bzw. die reellen Zahl “drei Komma sechs” (mathematische Abstraktion)
    bzw. “drei Komma sechs Meter” (Rückinterpretation)
  - interne Darstellung (im Rechner): Binärdarstellung

Mathematische Abstraktion zur Datenverarbeitung



- Allgemein: Es werden gewisse *Grunddatentypen* (z.B. für reelle Zahlen, natürliche Zahlen, Zeichen, etc.) mit entsprechenden *Grundoperationen* (*Grundfunktionen*) angenommen.

Typische Grunddatentypen (mathematische Abstraktion)

- Boolesche (Wahrheits-)Werte
  - Wertebereich: $\mathbb{B} = \{TRUE, FALSE\}$
  - typische Operationen:

$$\begin{array}{lcl} \wedge & : & \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\ \vee & : & \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\ \neg & : & \mathbb{B} \to \mathbb{B} \end{array}$$

    eigentlich auch

$$\begin{array}{lcl} TRUE & : & \to \mathbb{B} \\ FALSE & : & \to \mathbb{B} \end{array}$$

- Natürliche Zahlen
  - Wertebereich: $\mathbb{N} = \{0, 1, 2, \ldots\}$
  - typische Operationen:

$$\begin{array}{rcl} +_{(\mathbb{N})} & : & \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ -_{(\mathbb{N})} & : & \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ \cdot_{(\mathbb{N})} & : & \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ :_{(\mathbb{N})} & : & \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ =_{(\mathbb{N})} & : & \mathbb{N} \times \mathbb{N} \to \mathbb{B} \\ \neq_{(\mathbb{N})} & : & \mathbb{N} \times \mathbb{N} \to \mathbb{B} \\ >_{(\mathbb{N})} & : & \mathbb{N} \times \mathbb{N} \to \mathbb{B} \\ <_{(\mathbb{N})} & : & \mathbb{N} \times \mathbb{N} \to \mathbb{B} \end{array}$$

    usw.
    eigentlich auch

$$\begin{array}{rcl} 0_{(\mathbb{N})} & : & \to \mathbb{N} \\ 1_{(\mathbb{N})} & : & \to \mathbb{N} \end{array}$$

    usw.

Typische Grunddatentypen (mathematische Abstraktion)

- Ganze Zahlen
  - Wertebereich: $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
  - typische Operationen: $+_{(\mathbb{Z})}, -_{(\mathbb{Z})}, \cdot_{(\mathbb{Z})}, :_{(\mathbb{Z})}, =_{(\mathbb{Z})}, \neq_{(\mathbb{Z})}, <_{(\mathbb{Z})}, >_{(\mathbb{Z})}, \ldots$
- Reelle Zahlen
  - Wertebereich: $\mathbb{R}$ (die reellen Zahlen)
  - typische Operationen: $+_{(\mathbb{R})}, -_{(\mathbb{R})}, \cdot_{(\mathbb{R})}, :_{(\mathbb{R})}, =_{(\mathbb{R})}, \neq_{(\mathbb{R})}, <_{(\mathbb{R})}, >_{(\mathbb{R})}, \ldots$
- Zeichen (Charakter)
  - Wertebereich: *CHAR* = {“A”, “B”, . . ., “a”, “b”, . . ., “1”, “2”, . . ., “!”, . . .} (z.B. alle druckbaren ASCII-Zeichen)
  - Operationen: $=_{(CHAR)}, \neq_{(CHAR)}, <_{(CHAR)}, >_{(CHAR)}, \ldots$

Bemerkungen:

- Operationen auf den Grunddatentypen werden meist als gegeben vorrausgesetzt (siehe unsere Algorithmen *Strecke* und *Wechselgeld*).
- Tatsächlich verbirgt sich hinter jeder Operation wieder ein Algorithmus zu deren Berechnung.

Grunddatentypen (primitive Typen) in Java

- Es gibt in Java Grunddatentypen (auch *atomare* oder *primitive Typen*) für $\mathbb{B}$, *CHAR*, eine Teilmenge von $\mathbb{Z}$ und für eine Teilmenge von $\mathbb{R}$.
- Es gibt in Java keinen eigenen Grunddatentyp für $\mathbb{N}$.
- Die Werte der einzelnen primitiven Typen werden intern binär dargestellt.
- Die Datentypen unterscheiden sich u.a. in der Anzahl der Bits, die für ihre Darstellung verwendet werden.
- Die Anzahl der Bits, die für die Darstellung der Werte eines primitiven Datentyps verwendet werden heißt *Länge* des Typs.
- Die Länge eines Datentyps hat Einfluss auf den Wertebereich des Typs.

Grunddatentypen (primitive Typen) in Java

Überblick

Typname	Länge	Wertebereich
`boolean`	1 Byte	Wahrheitswerte {`true`,`false`}
`char`	2 Byte	Alle Unicode-Zeichen
`byte`	1 Byte	Ganze Zahlen von -2^7 bis $2^7 - 1$
`short`	2 Byte	Ganze Zahlen von -2^{15} bis $2^{15} - 1$
`int`	4 Byte	Ganze Zahlen von -2^{31} bis $2^{31} - 1$
`long`	8 Byte	Ganze Zahlen von -2^{63} bis $2^{63} - 1$
`float`	4 Byte	Gleitkommazahlen (einfache Genauigkeit)
`double`	8 Byte	Gleitkommazahlen (doppelte Genauigkeit)

Wahrheitswerte

- Java hat einen eigenen Typ `boolean` für Wahrheitswerte.
- Die beiden einzigen Werte (*Literale*, *Konstanten*) sind `true` für “wahr” und `false` für “falsch”.

Boole’sche Operatoren

Operator	**Bezeichnung**	**Bedeutung**
!	logisches NICHT ($\neg$)	`!a` ergibt `false` wenn `a` wahr ist, sonst `true`.
&	logisches UND ($\wedge$)	`a & b` ergibt `true`, wenn sowohl `a` als auch `b` wahr sind. Beide Teilausdrücke (`a` und `b`) werden ausgewertet.
&&	sequentielles UND	`a && b` ergibt `true`, wenn sowohl `a` als auch `b` wahr sind. Ist `a` bereits falsch, wird `false` zurückgegeben und `b` nicht mehr ausgewertet.
|	logisches ODER ($\vee$)	`a | b` ergibt `true`, wenn mindestens eine der beiden Ausdrücke `a` oder `b` wahr sind. Beide Teilausdrücke (`a` und `b`) werden ausgewertet.
||	sequentielles ODER	`a || b` ergibt `true`, wenn mindestens eine der beiden Ausdrücke `a` oder `b` wahr sind. Ist bereits `a` wahr, so wird `true` zurückgegeben und `b` nicht mehr ausgewertet.
^	exklusives ODER (EXOR)	`a ^ b` ergibt `true`, wenn beide Ausdrücke `a` und `b` einen unterschiedlichen Wahrheitswert haben.

Zeichen (Character)

- Java hat einen eigenen Typ **`char`** für (unicode-) Zeichen.
- Werte (*Literale*, *Konstanten*) werden in einfachen Hochkommata gesetzt, z.B. `'A'` für das Zeichen “A”.
- Einige Sonderzeichen können mit Hilfe von *Standard-Escape-Sequenzen* dargestellt werden:

Sequenz	**Bedeutung**
`\b`	Backspace (Rückschritt)
`\t`	Tabulator (horizontal)
`\n`	Newline (Zeilenumbruch)
`\f`	Seitenumbruch (Formfeed)
`\r`	Wagenrücklauf (Carriage return)
`\"`	doppeltes Anführungszeichen
`\'`	einfaches Anführungszeichen
`\\`	Backslash

Ganze Zahlen

- Java hat vier Datentypen für ganze (vorzeichenbehaftete) Zahlen: **`byte`** (Länge: 8 Bit), **`short`** (Länge: 16 Bit), **`int`** (Länge: 32 Bit) und **`long`** (Länge: 64 Bit).
- Werte (*Literale*, *Konstanten*) können geschrieben werden in
  - Dezimalform: bestehen aus den Ziffern `0`, . . ., `9`
  - Oktalform: beginnen mit dem Präfix `0` und bestehen aus Ziffern `0`, . . ., `7`
  - Hexadezimalform: beginnen mit dem Präfix `0x` und bestehen aus Ziffern `0`, . . ., `9` und den Buchstaben `a`, . . ., `f` (bzw. `A`, . . ., `F`)
- Negative Zahlen erhalten ein vorangestelltes `-`.

Gleitkommazahlen

- Java hat zwei Datentypen für Fließkommazahlen: **`float`** (Länge: 32 Bit) und **`double`** (Länge: 64 Bit).
- Werte (*Literale*, *Konstanten*) werden immer in Dezimalnotation geschrieben und bestehen maximal aus
  - Vorkommateil (Präfix - möglich)
  - Dezimalpunkt (*)
  - Nachkommateil
  - Exponent `e` oder `E` (Präfix - möglich) (*)
  - Suffix `f` oder `F` (**`float`**) oder `d` oder `D` (**`double`**) (*)

  wobei mindestens einer der mit (*) gekennzeichneten Bestandteile vorhanden sein muss.
- Negative Zahlen erhalten ein vorangestelltes `-`.
- Beispiele:
  - double: `6.23`, `623E-2`, `62.3e-1`
  - float: `6.23f`, `623E-2F`, `62.2e-1f`

Arithmetische Operationen

Operator	**Bezeichnung**	**Bedeutung**
`+`	Positives Vorzeichen	`+n` ist gleichbedeutend mit `n`
`-`	Negatives Vorzeichen	`-n` kehrt das Vorzeichen von `n` um
`+`	Summe	`a+b` ergibt die Summe von `a` und `b`
`-`	Differenz	`a-b` ergibt die Differenz von `a` und `b`
`*`	Produkt	`a*b` ergibt das Produkt aus `a` und `b`
`/`	Quotient	`a/b` ergibt den Quotienten von `a` und `b`
`%`	Modulo	`a%b` ergibt den Rest der ganzzahligen Division von a durch b. Auch auf Fließkommazahlen anwendbar
`++`	Präinkrement	`++a` ergibt `a+1` und erhöht `a` um `1`
`++`	Postinkrement	`a++` ergibt `a` und erhöht `a` um `1`
`--`	Prädekrement	`--a` ergibt `a-1` und verringert `a` um `1`
`--`	Postdekrement	`a--` ergibt `a` und verringert `a` um `1`

Vergleichsoperationen

Operator	Bezeichnung	Bedeutung
`==`	Gleich	`a==b` ergibt **`true`**, wenn `a` gleich `b` ist
`!=`	Ungleich	`a!=b` ergibt **`true`**, wenn `a` ungleich `b` ist
`<`	Kleiner	`a<b` ergibt **`true`**, wenn `a` kleiner `b` ist
`<=`	Kleiner gleich	`a<=b` ergibt **`true`**, wenn `a` kleiner oder gleich `b` ist
`>`	Größer	`a>b` ergibt **`true`**, wenn `a` größer `b` ist
`>=`	Größer gleich	`a>=b` ergibt **`true`**, wenn `a` größer oder gleich `b` ist

Ausdrücke

Aus Werten/Literalen/Konstanten (z.B. den **`int`** Werten `6` und `8`) und Operatoren (z.B. `+`, `*`, `<`, `&&`) können Ausdrücke gebildet werden. Ein gültiger Ausdruck hat selbst wieder einen Wert:

- `6 + 8` *`//Wert: 14 vom Typ int`*
- `6 * 8` *`//Wert: 48 vom Typ int`*
- `6 < 8` *`//Wert: true vom Typ boolean`*
- `6 && 8` *`//ungültiger Ausdruck`*

  Warum?

Typkonversion

- Motivation:
  - Bei unserem Algorithmus *Strecke* (Folie 38) haben wir Daten unterschiedlichen Typs miteinander kombiniert:

    $$strecke(m, t, k) = \frac{k \cdot t \cdot t}{2 \cdot m}$$

    wobei (strenggenommen) $m, t, k \in \mathbb{R}$ also z.B. vom Typ **`float`** und $2 \in \mathbb{N}$, also z.B. vom Typ **`int`**.
- Frage: Ist das erlaubt?
- Warum nicht? Weil in unserer mathematischen Abstraktion $\cdot$ eine Operation ist, die ausschließlich auf $\mathbb{N}$, $\mathbb{Z}$, oder $\mathbb{R}$ definiert ist, also eigentlich

$$\begin{array}{lll} \cdot^{(\mathbb{N})} & : & \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}, \\ \cdot^{(\mathbb{Z})} & : & \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}, \\ \cdot^{(\mathbb{R})} & : & \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}. \end{array}$$

- Offensichtlich benötigen wir eine Typanpassung, damit wir Daten unterschiedlicher Typen gemeinsam verarbeiten können.

Automatische Typkonversion in Java

“Kleiner Beziehung” zwischen Datentypen:

`byte < short < int < long < float < double`

Java konvertiert Ausdrücke automatisch in den allgemeineren Typ. Beispiele:

- `1 + 1.7` ist vom Typ **`double`**
- `1.0f + 2` ist vom Typ **`float`**
- `1.0f + 2.0` ist vom Typ **`double`**

Explizite Typkonversion: Type-Casting

Type Casting

Erzwingen der Typkonversion (zum spezielleren Typ `type`) durch Voranstellen von `(type)`: Der Ausdruck `(type) a` wandelt den Ausdruck `a` in einen Ausdruck vom Typ `type` um.

Beispiele:

- `(`**`byte`**`)3` ist vom Typ **`byte`**
- `(`**`int`**`)(2.0 + 5.0)` ist vom Typ **`int`**
- `(`**`float`**`)1.3e-7` ist vom Typ **`float`**

Bei der Typkonversion in einen spezielleren Typ kann Information verloren gehen.
Beispiele:

- `(`**`int`**`)5.2` ergibt `5`
- `(`**`int`**`)-5.2` ergibt `-5`

Type-Cast-Operator

Im Ausdruck `(type) a` ist `(type)` ein Operator. Type-Cast-Operatoren bilden also zusammen mit einem Ausdruck wieder einen Ausdruck.
Der Typ des Operators ist z.B.:

```
  (int)  :  charUbyteUshortUintUlongUfloatUdouble→int
(float)  :  charUbyteUshortUintUlongUfloatUdouble→float
```

Korrespondenz Java – SML

	Java	SML
Gleitpunktzahlen	**`float, double`**	`real`
unäres Minus	`-`	`~`
Division	`/`	`/`
Modulo	`%`	nicht vorhanden
Konversion nach ganze Zahl	`(int)`	`truncate`
Ganze Zahlen	**`int`**	**`int`**
Ganzzahldivision	`/`	`div`
Modulo	`%`	`mod`
Boole'sche Werte	**`boolean`**	`bool`
logisches NICHT	`!`	`not`
logisches UND	`&`	nicht vorhanden
sequentielles UND	`&&`	`andalso`
logisches ODER	`|`	nicht vorhanden
sequentielles ODER	`||`	`orelse`
exklusives ODER (EXOR)	`^`	nicht vorhanden