

Zustandsübergänge

- ▶ Wir schreiben $S \xrightarrow{a} \hat{S}$, um auszudrücken, dass wir $\hat{S}$ aus S durch die Anwendung von a erhalten haben.
- ▶ Um uns möglicherweise Schreibarbeit zu ersparen, können wir auch Zustandsübergänge von mehreren sequentiellen Anweisungen zusammenfassen:
- ▶ Ist A eine Folge von Anweisungen (Deklaration/Initialisierung, Wertzuweisung) $a_1; \dots a_k$; und es gilt $S = S_0 \xrightarrow{a_1} \dots \xrightarrow{a_k} S_k = \hat{S}$, so heißt $\hat{S}$ Nachfolgezustand von S bzgl. A und wir schreiben $S \xrightarrow{A} \hat{S}$.

Zustandsübergänge

- Beispiel: hier nochmal der Algorithmus

```
procedure fBerechnen1( $x : \mathbb{R}$ )  $\rightarrow \mathbb{R}$   
  output  Berechnung der Funktion  $f$   
  pre     $x \neq 1$   
  body  
    var  $y_1, y_2, y_3 : \mathbb{R};$   
     $y_1 \leftarrow x + 1;$   
     $y_2 \leftarrow y_1 + 1/y_1;$   
     $y_3 \leftarrow y_2 \cdot y_2;$   
    return  $y_3;$ 
```

von oben.

- Wie wird der Algorithmus für einen konkreten Wert (z.B. 3.0) ausgeführt, d.h. was passiert beim Aufruf von *fBerechnen*(3.0)?

Zustandsübergänge

Ausführung des Aufrufs $fBerechnen(3.0)$:

Anweisung	x	y_1	y_2	y_3	Zustand
Aufruf $fBerechnen1(3.0)$	3.0	■	■	■	$\{(x, 3.0)\}$
var $y_1, y_2, y_3 : \mathbb{R};$	3.0	ω	ω	ω	$\{(x, 3.0), (y_1, \omega), (y_2, \omega), (y_3, \omega)\}$
$y_1 \leftarrow x + 1;$	3.0	$W(x + 1)$	ω	ω	$\{(x, 3.0), (y_1, W(x + 1)), (y_2, \omega), (y_3, \omega)\}$
bzw.	3.0	4.0	ω	ω	$\{(x, 3.0), (y_1, 4.0), (y_2, \omega), (y_3, \omega)\}$
...					

Legende: ■ = Zettel existiert noch nicht.



Zustandsübergänge

Fortsetzung:

Anweisung	x	y_1	y_2	y_3	Zustand
...					
$y_2 \leftarrow y_1 + 1/y_1;$	3.0	4.0	$W(y_1 + 1/y_1)$	ω	$\{(x, 3.0), (y_1, 4.0), (y_2, W(y_1 + 1/y_1)), (y_3, \omega)\}$
bzw.	3.0	4.0	4.25	ω	$\{(x, 3.0), (y_1, 4.0), (y_2, 4.25), (y_3, \omega)\}$
$y_3 \leftarrow y_2 \cdot y_2;$	3.0	4.0	4.25	$W(y_2 \cdot y_2)$	$\{(x, 3.0), (y_1, 4.0), (y_2, W(y_1 + 1/y_1)), (y_3, W(y_2 \cdot y_2))\}$
bzw.	3.0	4.0	4.25	18.0625	$\{(x, 3.0), (y_1, 4.0), (y_2, 4.25), (y_3, 18.0625)\}$

Legende: ■ = Zettel existiert noch nicht.

Prozeduraufrufe in Prozeduren

- ▶ Die Erweiterung des Modulbegriffs um Prozeduren ermöglicht uns nun natürlich auch, diese Prozeduren in anderen Algorithmen zu verwenden.
- ▶ Das Modul *BewegungI* auf der nächsten Folie ist eine imperative Variante des Moduls *Bewegung* aus dem vorherigen Kapitel .
- ▶ Die Prozedur *streckeI* wird in der Prozedur *arbeitI* **aufrufen**.
- ▶ Solch ein Aufruf ist syntaktisch ein Ausdruck, kann also überall stehen, wo ein Ausdruck stehen darf (die formale Erweiterung der induktiven Definition von Ausdrücken ist trivial, und sei dem Leser überlassen).

Prozeduraufrufe in Prozeduren

module *BewegungI*

operations

procedure *streckeI*($m : \mathbb{R}, t : \mathbb{R}, k : \mathbb{R}$) $\rightarrow \mathbb{R}$

pre $m > 0, t \geq 0$

body

var $b : \mathbb{R};$

$b \leftarrow k \cdot t^2;$

$b \leftarrow b / (2 \cdot m);$

return $b;$

procedure *endgeschwindigkeit*($m : \mathbb{R}, t : \mathbb{R}, k : \mathbb{R}$) $\rightarrow \mathbb{R}$

...

procedure *arbeiteI*($m : \mathbb{R}, k : \mathbb{R}, t : \mathbb{R}$) $\rightarrow \mathbb{R}$

pre $m > 0, t \geq 0$

body

var $s : \mathbb{R};$

$s \leftarrow \textit{strecke}(m, t, k);$

return $k \cdot s;$

Prozeduren ohne Rückgabe

- ▶ Bei einer Funktion ist der Bildbereich eine wichtige Information: $f : D \rightarrow B$.
- ▶ Bei einer Prozedur, die keine Funktion ist, wählt man als Bildbereich oft die leere Menge: $p : D \rightarrow \emptyset$.
- ▶ Dies signalisiert, dass die Seiteneffekte der Prozedur zur eigentlichen Umsetzung eines Algorithmus gehören, dagegen aber kein (bestimmtes) Element aus dem Bildbereich einer Abbildung als Ergebnis des Algorithmus angesehen werden kann.
- ▶ Bei der **procedure**-Vereinbarung fehlt dann einfach der Bildbereich.

Prozeduren ohne Rückgabe

- Beispiel:

```
procedure vertausche( $x : \mathbb{N}, y : \mathbb{N}$ )  
  output  Vertausche die Werte von  $x$  und  $y$   
  body  
    var    $a : \mathbb{N};$   
     $a \leftarrow x;$   
     $x \leftarrow y;$   
     $y \leftarrow a;$ 
```

- In dieser Prozedur fehlt die finale **return**-Anweisung, da nichts zurück gegeben wird.

Prozeduren ohne Rückgabe

- ▶ Wir lassen den Aufruf einer solchen Prozedur ohne Wertzuweisung an eine Variable (was genau genommen nur ein Ausdruck ist) als Anweisung zu.

- ▶ Beispiel:

```
procedure groesserOderKleiner( $x : \mathbb{N}, y : \mathbb{N}$ )  $\rightarrow \mathbb{B}$   
  body  
    vertausche( $x, y$ );  
    return  $x < y$ ;
```

- ▶ Diese Anweisung (hier *vertausche*(n, m);) heißt Ausdrucksanweisung.
- ▶ Und jetzt stellt sich natürlich die Frage: vertauscht *vertausche*(x, y) in *groesserOderKleiner* tatsächlich die Werte von x und y ?
- ▶ Oder konkret: was für ein Ergebnis liefert z.B. *groesserOderKleiner*(1, 2)?

Prozeduraufrufe in Prozeduren

- Das bringt uns zurück zu der Frage: was passiert ganz allgemein, wenn eine Prozedur mit Signatur

$p_a(e_1 : T_1, \dots, e_n : T_n) \quad [\rightarrow T] \quad (\text{d.h. } \rightarrow T \text{ ist optional})$

innerhalb des Rumpfes einer anderen Prozedur p_u mit Variablen

$x_1 \in T_1, \dots, x_n \in T_n$, die in p_u bekannt sind, als Argument aufgerufen wird, d.h. im Rumpf von p_u steht die Anweisung

$p_a(x_1, \dots, x_n);$

Wir sagen hier auch, die Variablen $x_1, \dots, x_n$ werden an p_a **übergeben**.

(Im Beispiel: p_a ist *vertausche* und p_u ist *groesserOderKleiner*)

- Variablen können grundsätzlich auf zwei Arten übergeben werden:

Parameterübergabe bei Prozeduraufrufen

► Möglichkeit 1: *Call-by-value*

- Für jede (formale) Eingabe-Variable $e_1, \dots, e_n$ (der Signatur) wird im Methoden-Block eine neue Variable (ein Extra-Zettel) angelegt.
- Diese Extra-Variablen erhalten die *Werte* der übergebenen Variablen $x_1, \dots, x_n$ (bzgl. des Zustands zum Zeitpunkt des Aufrufs von p_a in p_u).
- Die übergebenen Variablen $x_1, \dots, x_n$ sind im Rumpf von p_a nicht *sichtbar*, d.h. nicht in der Menge $N(\mathcal{S}_0)$ (wobei $\mathcal{S}_0$ der Anfangszustand der Prozedur p_a ist) enthalten (später können natürlich Variablen/Konstanten gleichen Namens innerhalb von p_a deklariert werden, dann sind diese aber neue Zettel).
- Auf diese Weise bleibt der Wert der ursprüngliche Variable von Anweisungen innerhalb der Methode unberührt.

Parameterübergabe: Call-by-value

- ▶ Java wertet Parameter call-by-value aus und wir spezifizieren das in unserer Semantik ebenso.
- ▶ Dies macht insbesondere deshalb Sinn, weil in Java (und bei uns) nicht nur Variablen sondern auch ganze Ausdrücke an Methoden übergeben werden dürfen (so hatten wir übrigens auch die Syntax von Ausdrücken definiert).
- ▶ Wir erweitern dazu die Formalisierung der Zustandsübergänge, die eine Anweisung a bewirkt, um folgende Fälle:

Parameterübergabe: Call-by-value

Fall 1:

- ▶ Beim Aufruf einer Prozedur mit der Signatur $p(e_1 : T_1, \dots, e_n : T_n)$ mit konkreten Eingaben (Ausdrücken) $w_1, \dots, w_n$ (d.h. Anweisung a hat die Form $p(w_1, \dots, w_n);$) im Zustand S wird zunächst der Zustand

$$\mathcal{S}_{init_p} = \{(e_1, W_S(w_1)), \dots, (e_n, W_S(w_n))\}$$

erreicht, d.h., $\mathcal{S}_{init_p}$ enthält nur die Eingabe-Variablen mit den Werten der übergebenen Ausdrücke bzgl. S .

- ▶ Der Rumpf r von p überführt $\mathcal{S}_{init_p}$ in $\mathcal{S}_{end_p}$, d.h. $\mathcal{S}_{init_p} \xrightarrow{r} \mathcal{S}_{end_p}$.
- ▶ Da p keinen Rückgabewert hat, ist der Nachfolgezustand von $\hat{S}$ bzgl. a gleich S , d.h. $S \xrightarrow{a} S$ (die Seiteneffekte haben also keinen Einfluss auf $\hat{S}$ bzw. S).

Parameterübergabe: Call-by-value

Fall 2:

- ▶ Beim Aufruf einer Prozedur der Signatur $p(e_1 : T_1, \dots, e_n : T_n) \rightarrow T$ mit konkreten Eingaben (Ausdrücke) $w_1, \dots, w_n$
(d.h. Anweisung a hat die Form $x = p(w_1, \dots, w_n);$)
im Zustand $S = \{\dots (x, w), \dots\}$ (wobei $w = \omega$ sein kann) wird zunächst der Zustand

$$S_{init_p} = \{(e_1, W_S(w_1)), \dots, (e_n, W_S(w_n))\}$$

erreicht, d.h., S_{init_p} enthält nur die Eingabe-Variablen mit den Werten der übergebenen Ausdrücke bzgl. S (wie oben).

... (Fortsetzung nächste Seite)

Parameterübergabe: Call-by-value

... (Fortsetzung Fall 2):

- ▶ Der Rumpf r von p überführt $\mathcal{S}_{init_p}$ in $\mathcal{S}_{end_p}$, d.h. $\mathcal{S}_{init_p} \xrightarrow{r} \mathcal{S}_{end_p}$.
- ▶ Die finale **return**-Anweisung in p enthält den Ausdruck t vom Typ T mit Wert $W_{\mathcal{S}_{end_p}}(u)$ im Zustand $\mathcal{S}_{end_p}$.
- ▶ Der Nachfolgezustand ist damit gegeben durch $\hat{\mathcal{S}} = \mathcal{S} \setminus \{(x, w)\} \cup \{(x, W_{\mathcal{S}_{end_p}}(u))\}$ und es gilt wieder $\mathcal{S} \xrightarrow{a} \hat{\mathcal{S}}$.

Parameterübergabe: Call-by-value

Beispiel:

- Hier nochmal die beiden Prozeduren von vorher:

```
procedure vertausche( $x : \mathbb{N}, y : \mathbb{N}$ )  
  output   Vertausche die Werte von  $x$  und  $y$   
  body
```

```
1    var    $a : \mathbb{N}$ ;  
2     $a \leftarrow x$ ;  
3     $x \leftarrow y$ ;  
4     $y \leftarrow a$ ;
```

```
procedure groesserOderKleiner( $x : \mathbb{N}, y : \mathbb{N}$ )  $\rightarrow \mathbb{B}$   
  body
```

```
5    vertausche( $x, y$ );  
6    return   $x < y$ ;
```

- Was für ein Ergebnis liefert *groesserOderKleiner*(1,2)?

Parameterübergabe: Call-by-value

procedure *vertausche*($x : \mathbb{N}, y : \mathbb{N}$)

body

```

1  var  a :  $\mathbb{N}$ ;
2  a  $\leftarrow$  x;
3  x  $\leftarrow$  y;
4  y  $\leftarrow$  a;
```

procedure *groesserOderKleiner*($x : \mathbb{N}, y : \mathbb{N}$) $\rightarrow \mathbb{B}$

body

```

5  vertausche(x, y);
6  return  x < y;
```

Aufruf von *groesserOderKleiner*(1, 2):

- Wir unterscheiden die formalen Eingabeparameter von *groesserOderKleiner* und *vertausche*: x_a/y_a sind die von *groesserOderKleiner*, der äußeren Prozedur, x_i/y_i die von *vertausche*, der inneren Prozedur
- Der Zustand vor Zeile 5 ist $S_0 = \{(x_a, 1), (y_a, 2)\}$; dies ist der Anfangszustand von *groesserOderKleiner* für die konkreten Eingabewerte 1 und 2.
- Beim Aufruf von *vertausche*(x_a, y_a) in Zeile 5 werden neue Zettel mit Namen x_i und y_i für die Eingabeparameter von *vertausche* angelegt, die nur im Rumpf von *vertausche* gelten, und mit den Werten von den übergebenen Ausdrücken x_a und y_a belegt sind.
- Der Anfangszustand (Nachfolgezustand von S_0) von *vertausche* ist also $S_1 = \{(x_i, W_{S_0}(x_a), (y_i, W_{S_0}(y_a))\}$, d.h. $S_1 = \{(x_i, 1, (y_i, 2)\}$ (vor Zeile 1).

Parameterübergabe: Call-by-value

procedure *vertausche*($x : \mathbb{N}, y : \mathbb{N}$)

body

```
1  var  a :  $\mathbb{N}$ ;  
2  a  $\leftarrow$  x;  
3  x  $\leftarrow$  y;  
4  y  $\leftarrow$  a;
```

procedure *groesserOderKleiner*($x : \mathbb{N}, y : \mathbb{N}$) $\rightarrow \mathbb{B}$

body

```
5  vertausche(x, y);  
6  return  x < y;
```

Aufruf von *groesserOderKleiner*(1, 2) (Fortsetzung):

- ▶ Am Ende (nach Zeile 4) von *vertausche* ist offenbar der Zustand $\mathcal{S}_2 = \{(x_i, 2), (y_i, 1), (a, 1)\}$ erreicht (a ist die lokale Variable in *vertausche*).
- ▶ Nach Beendigung des Rumpfes von *vertausche* sind nun x_i und y_i aus *vertausche* in *groesserOderKleiner* nicht mehr sichtbar. Der Nachfolgezustand von $\mathcal{S}_2$ (nach Zeile 5) ist also $\mathcal{S}_3 = \{(x_a, 1), (y_a, 2)\}$.
- ▶ Rückgabe ist als $W(x_a < y_a)$, d.h. *TRUE*.

Parameterübergabe bei Prozeduraufrufen

- ▶ Der Vollständigkeit halber noch Möglichkeit 2: *Call-by-reference*
 - ▶ Für jede (formale) Eingabe-Variable $e_1, \dots, e_n$ (der Signatur) wird im Methoden-Block *keine* neue Variable angelegt.
 - ▶ Die Eingabe-Variablen $e_1, \dots, e_n$ erhalten stattdessen eine *Referenz* (*Verweis*) auf die übergebenen Variablen $x_1, \dots, x_n$.
 - ▶ Wenn innerhalb der Methode der Wert einer der Eingabe-Variablen $e_1, \dots, e_n$ verändert wird, wird also in Wirklichkeit eine der übergebenen Variablen $x_1, \dots, x_n$ verändert, obwohl diese im Rumpf von p_a gar nicht sichtbar sind.
 - ▶ Das hat dann offenbar auch Auswirkungen außerhalb der Methode!
- ▶ **Achtung: call-by-reference ist daher eine potentielle Quelle unbeabsichtigter Seiteneffekte!!!**

Parameterübergabe: Call-by-reference

procedure *vertausche*($x : \mathbb{N}, y : \mathbb{N}$)

body

```
1  var  a :  $\mathbb{N}$ ;  
2  a  $\leftarrow$  x;  
3  x  $\leftarrow$  y;  
4  y  $\leftarrow$  a;
```

procedure *groesserOderKleiner*($x : \mathbb{N}, y : \mathbb{N}$) $\rightarrow \mathbb{B}$

body

```
5  vertausche(x, y);  
6  return  x < y;
```

Aufruf von *groesserOderKleiner*(1, 2) (mit call-by-reference):

- ▶ Beim Aufruf von *vertausche*(x, y) mit ($x, 1$) und ($x, 2$) in Zeile 5 werden für die Eingabeparameter von *vertausche* Referenzen auf die ursprünglichen Zettel hergenommen.
- ▶ Mit unserer vorherigen Unterscheidung referenziert x_i nur x_a (und y_i entspr. x_i), d.h. x_i und x_a bezeichnen letztlich den selben Zettel (Speicherzelle).
- ▶ Im Rumpf von *vertausche* werden also jetzt tatsächlich die Werte der Zettel x und y vertauscht.
- ▶ Am Ende (Zeile 6) wird dann natürlich *FALSE* zurück gegeben.

Parameterübergabe: Call-by-value/-reference

- ▶ Tatsächlich ist call-by-reference in einigen Programmiersprachen möglich, z.B. in C durch die Verwendung sog. *Pointer* (vergessen Sie es schnell wieder, das ist häßliches Programmieren — warum gibt es diese Pointer dann überhaupt??? Und wieso gibt es call-by-reference?).
- ▶ Übrigens wird uns call-by-reference noch verfolgen:
- ▶ Java stellt neben den atomaren Datentypen auch Referenz-Typen (z.B. Arrays) bzw. Objekt-Typen (z.B. benutzereigene Datentypen durch Klassen) zur Verfügung, deren Variablen anders behandelt werden und dabei zu einem call-by-reference Effekt führen.
- ▶ Dazu aber später mehr.

Parameterübergabe: Call-by-value/-reference

- ▶ Aber Moment mal:

Mit call-by-value haben Prozeduren vielleicht Seiteneffekte. Aber diese Seiteneffekte sind außerhalb der Prozeduren nicht bemerkbar.

- ▶ D.h., wenn eine Prozedur keine Rückgabe liefert, dann ist sie für uns momentan eigentlich nutzlos¹⁴???

- ▶ Wie gesagt: momentan. Mit Referenztypen ändert sich das dann.

- ▶ Aber tatsächlich: wenn zwei Prozeduren (Algorithmen) gemeinsame Daten verändern sollen (was übrigens schlechter Stil ist), brauchen wir aktuell noch etwas anderes, nämlich sog. globale Variablen/Konstanten (siehe später).



¹⁴Abgesehen von externen Effekten wie Ausgabe, etc.

Vereinbarkeit von funktionalen und imperativen Konzepten

- Wie bereits erwähnt ergänzen sich funktionale und imperative Konzepte wie folgendes Beispiel zeigt:

module *BewegungFI*

operations

function *strecke*($m : \mathbb{R}, t : \mathbb{R}, k : \mathbb{R}$) $\rightarrow \mathbb{R}$

pre $m > 0, t \geq 0$

body $k \cdot t^2 / (2 \cdot m)$

function *endgeschwindigkeit*($m : \mathbb{R}, t : \mathbb{R}, k : \mathbb{R}$) $\rightarrow \mathbb{R}$

pre $m > 0, t \geq 0$

body $(k/m) \cdot t$

procedure *arbeit*($m : \mathbb{R}, k : \mathbb{R}, t : \mathbb{R}$) $\rightarrow \mathbb{R}$

pre $m > 0, t \geq 0$

body

var $s : \mathbb{R};$

$s \leftarrow \textit{strecke}(m, t, k);$

return $k \cdot s;$

Überblick

6. Grundlagen der imperativen Programmierung

6.1 Variablen, Anweisungen, Prozeduren

6.2 Prozeduraufrufe

6.3 Variablen, Anweisungen und Prozeduren in Java

6.4 Bedingte Anweisungen und Iteration

6.5 Reihungen (Arrays)

6.6 Zeichenketten (Strings)

6.7 Strukturierung von Programmen

6.8 Zusammenfassung und Beispiele

Variablen und Konstanten in Java

- ▶ Eine Variablendeklaration hat in Java die Gestalt

```
Typname variablenName;
```

Konvention: Variablennamen beginnen mit kleinen Buchstaben

Beispiel: **int** x;

- ▶ Eine Konstantendeklaration hat in Java die Gestalt

```
final Typname KONSTANTENNAME;
```

Konvention: Konstantennamen bestehen komplett aus großen Buchstaben

Beispiel: **final int** PI;

- ▶ Auch in Java hat jede Variable/Konstante damit einen Typ (eine Deklaration ist also das Bereitstellen eines Platzhalters des entsprechenden Typs).



Variablen und Konstanten in Java

- ▶ Wie bereits erwähnt, müssen Variablen/Konstanten in einem Java Programm nicht am Anfang einer Methode vereinbart werden, sondern können auch „zwischen durch“ eingeführt werden (überall dort, wo eine Anweisung stehen darf, da es sich dabei ja auch um eine Anweisung handelt).
- ▶ Intuitiv bedeutet das, dass die Menge $N(S)$ während einer Methode „wachsen“ kann.
- ▶ Der Compiler hält beim Übersetzen in Bytecode die aktuelle Liste, um die (syntaktische) Gültigkeit von Ausdrücken überprüfen zu können.
- ▶ Variablen/Konstanten, die erst später im Code deklariert werden, können also vorher nicht in Ausdrücken verwendet werden.

Variablen und Konstanten in Java

- ▶ Eine Wertzuweisung (z.B. Initialisierung) hat die Gestalt

```
variablenName = NeuerWert;
```

bzw.

```
KONSTANTENNAME = Wert; (nur als Initialisierung)
```

- ▶ Eine Variablen- bzw. Konstantendeklaration kann auch mit der Initialisierung verbunden sein, d.h. der ersten Wertzuweisung.

```
Typname variablenName = InitialerWert;
```

(Konstantendeklaration mit Initialisierung analog mit Zusatz `final`)

- ▶ Wie gesagt, Deklaration, Initialisierung und Wertzuweisung sind *Anweisungen*, die wie bei uns mit einem Semikolon beendet werden.

Variablen und Konstanten in Java

Beispiele:

- Konstanten: **final** <typ> <NAME> = <ausdruck>;

```
final double Y_1 = 1;  
final double Y_2 = Y_1 + 1 / Y_1;  
final double Y_3 = Y_2 * Y_2;  
final char NEWLINE = '\n';  
final double BESTEHENSGRENZE_PROZENT = 0.5;  
final int GESAMTPUNKTZAHL = 80;
```

- Variablen: <typ> <name> = <ausdruck>;

```
double y = x + 1; //Achtung: x muss vorher bekannt sein!  
int klausurPunkte = 42;  
boolean klausurBestanden =  
    ((double) klausurPunkte) /  
    GESAMTPUNKTZAHL >= BESTEHENSGRENZE_PROZENT;
```

Abkürzungen für Wertzuweisungen in Java

Für bestimmte einfache Operationen (Addition und Subtraktion mit 1 als zweitem Operanden) kennen wir schon Kurznotationen:

Operator	Bezeichnung	Bedeutung
++	Präinkrement	<code>++a</code> ergibt <code>a+1</code> und erhöht <code>a</code> um 1
++	Postinkrement	<code>a++</code> ergibt <code>a</code> und erhöht <code>a</code> um 1
--	Prädecrement	<code>--a</code> ergibt <code>a-1</code> und verringert <code>a</code> um 1
--	Postdecrement	<code>a--</code> ergibt <code>a</code> und verringert <code>a</code> um 1

Also steht z.B. `a++`; für `a = a + 1`; . Besonders ist, dass es sich dabei um Ausdrücke handelt, die einen Wert haben.



Anweisungen

- ▶ Wie wir bereits festgestellt haben, steht eine Anweisung für einen einzelnen Abarbeitungsschritt in einem Algorithmus.
- ▶ Einige wichtige Arten von Anweisungen in Java sind:

- ▶ Die *leere* Anweisung, bestehend aus einem einzigen `;`;
- ▶ Vereinbarungen und Initialisierungen von Variablen/Konstanten
- ▶ *Ausdrucksanweisung*: `<ausdruck>;`;

Dabei spielt der Wert von `<ausdruck>` keine Rolle, die Anweisung ist daher nur sinnvoll (und in Java nur dann erlaubt), wenn `<ausdruck>` einen Nebeneffekt hat, z.B. Wertzuweisung sowie (Prä- / Post-)Inkrement und Dekrement von Variablen/Konstanten, Funktions-/Prozeduraufruf (werden wir später kennenlernen), Instanzerzeugung (werden wir später kennenlernen)

Der Anweisungsblock

- ▶ Ein Block von Anweisungen wird in Java gebildet von einer öffnenden geschweiften Klammer und einer schließenden geschweiften Klammer, die eine beliebige Menge von Anweisungen umschließen:

```
{  
    Anweisung1;  
    Anweisung2;  
    ...  
}
```

- ▶ Die Anweisungen im Block werden nacheinander ausgeführt.
- ▶ Der Block als Ganzes gilt als eine einzige Anweisung, kann also überall da stehen, wo syntaktisch eine einzige Anweisung verlangt ist.
- ▶ Blöcke können beliebig geschachtelt sein.

Lebensdauer, Gültigkeit, Sichtbarkeit

- ▶ Das Konzept des (Anweisungs-)Blocks ist wichtig im Zusammenhang mit Variablen und Konstanten.
- ▶ Die *Lebensdauer* einer Variablen ist die Zeitspanne, in der die virtuelle Maschine der Variablen einen *Speicherplatz* zu Verfügung stellt.
- ▶ Die *Gültigkeit* einer Variablen erstreckt sich auf alle Programmstellen, an denen der Name der Variablen dem Compiler durch eine Vereinbarung (*Deklaration*) bekannt ist.
- ▶ Die *Sichtbarkeit* einer Variablen erstreckt sich auf alle Programmstellen, an denen man über den Namen der Variablen auf ihren Wert zugreifen kann.

Gültigkeitsbereich von Variablen

- ▶ Eine in einem Block deklarierte (*lokale*) Variable ist ab ihrer Deklaration bis zum Ende des Blocks gültig und sichtbar.
- ▶ Mit Verlassen des Blocks, in dem eine Variable lokal deklariert wurde, endet auch ihre Gültigkeit und Sichtbarkeit.
- ▶ Damit oder kurz danach endet normalerweise auch die Lebensdauer der Variablen, da der Speicherplatz, auf den die Variable verwiesen hat, im Prinzip wieder freigegeben ist (dazu später mehr) und für neue Variablen verwendet werden kann.
- ▶ Solange eine Variable sichtbar ist, darf keine neue Variable gleichen Namens angelegt werden.
- ▶ Der Rumpf einer Methode definiert z.B. einen Block (siehe gleich).

Gültigkeitsbereich von Variablen

► Beispiel:

```
...  
int i = 0;  
{  
    int i = 1;    // nicht erlaubt, i ist schon bekannt  
    i = 1;       // erlaubt  
    int j = 0;  
}  
j = 1;           // nicht moeglich, j ist nicht bekannt  
...  
int j;           // erlaubt
```

- Bei verschachtelten Blöcken: Variablen des äußeren Blocks sind im inneren Block sichtbar.

Prozeduren in Java

- ▶ In Java werden Prozeduren (wie Funktionen) durch *Methoden* realisiert.
- ▶ Eine Methode wird definiert durch
 - ▶ den *Methodenkopf*:

public static <typ> <name>(<parameterliste>)

der den Namen und die *Signatur* der Methode spezifiziert:

<typ> ist der Bildbereich (Ergebnistyp)

<parameterliste> spezifiziert die Eingabeparameter und -typen und besteht aus endlich vielen (auch keinen) Paaren von Typen und Variablennamen (<Typ> <VName>) jeweils durch Komma getrennt

- ▶ den *Methodenrumpf*: einen Block von Anweisungen (in entspr. Klammern), der sich an den Methodenkopf anschließt.
- ▶ Beispiel: **public static int** mitte(**int** x, **int** y, **int** z) ...

Prozeduren in Java

- ▶ Als besonderer Ergebnis-Typ einer Methode ist auch `void` möglich. Dieser Ergebnis-Typ steht für die leere Menge als Bildbereich.
- ▶ Eine Methode mit Ergebnistyp `void` gibt *kein* Ergebnis zurück; der Sinn einer solchen Methode liegt also ausschließlich in den Nebeneffekten.
- ▶ Das Ergebnis einer Methode ist der Ausdruck nach dem Schlüsselwort `return`; nach Auswertung dieses Ausdrucks endet die Ausführung der Methode.
 - ▶ Eine Methode mit Ergebnistyp `void` hat entweder keine oder eine leere `return`-Anweisung.
 - ▶ Eine Methode, die einen Ergebnistyp `<type> ≠ void` hat, muss mindestens eine `return`-Anweisung mit einem Ausdruck vom Typ `<type>` haben.

Prozeduren in Java

- ▶ Analog können wir nun auch das Modulkonzept erweitern, um benutzereigene Prozeduren bereitzustellen.
- ▶ Beispiel:

```
public class Bewegung1
{
    /**
     * ...
     */
    public static double streckel(double m, double t, double k)
    {
        double b = k / m;
        return 0.5 * b * (t * t);
    }

    ...
}
```

Prozeduren in Java

- ▶ Eine Methode mit Ergebnistyp `<typ>` $\neq$ `void` ist ein Ausdruck vom Typ `<typ>` und kann überall dort stehen, wo ein Ausdruck vom Typ `<typ>` verlangt ist (z.B. bei einer Wertzuweisung an eine Variable vom Typ `<typ>`).
- ▶ Beispiel: `double s = strecke(3.0, 4.2, 7.1);`
- ▶ Achtung: da Java nicht zwischen Funktion und Prozedur unterscheidet, könnte die Methode `strecke` Nebeneffekte haben!

Prozeduren in Java

- ▶ Eine Methode mit Ergebnistyp `void` ist ein Ausdruck vom Typ `void` und kann als Ausdrucksanweisung verwendet werden.
- ▶ Die Seiteneffekte der Methode ist das (hoffentlich) beabsichtigte Resultat der Anweisung.
- ▶ Beispiel: `System.out.println` ist eine Methode mit Ergebnistyp `void` und hat als Seiteneffekt: gib den Eingabeparameter (Typ `String` für Zeichenketten) auf dem Bildschirm aus, d.h.
`System.out.println(...);` ist eine Ausdrucksanweisung.
- ▶ Unser theoretisches Zustandsmodell gilt analog für Java Anweisungen und Prozeduraufrufe, d.h. Nebeneffekte bei Prozeduren sind mit unseren bisherigen Konzepten zunächst ohne Wirkung (call-by-value!)

Prozeduren in Java

- ▶ Der Rumpf einer Methode definiert einen Anweisungsblock (daher auch die Blockklammern).
- ▶ Es gelten die bereits bekannten Regeln zu Lebensdauer, Sichtbarkeit, etc. für Blöcke.
- ▶ Zusätzlich zu den lokalen Variablen, die innerhalb des Blocks vereinbart werden, sind noch die formalen Eingabe-Variablen der Methode sichtbar (so sind wir es ja auch gewohnt).

Prozeduren in Java

- ▶ Java setzt wie erwähnt call-by-value um:
- ▶ Beispiel:

```
public class Exchange
{
    public static void swap(int i, int j) {
        int c = i;
        i = j;
        j = c;
    }

    public static void main(String[] args) {
        int x = 1;
        int y = 2;
        swap(x,y);
        System.out.println(x); // Ausgabe?
        System.out.println(y); // Ausgabe?
    }
}
```

Prozeduren in Java

► Aufruf von `main`:

- Zunächst werden nacheinander $(x, 1)$, $(y, 2)$ angelegt (wir ignorieren `args` im Kopf von `main`).
- Beim Aufruf von `swap` werden neue Zettel angelegt: `i` als Kopie für `x` und `j` als Kopie für `y`:
 $(i, 1), (j, 2)$
- Dann
 $(i, 1), (j, 2), (c, 1)$
 $(i, 2), (j, 2), (c, 1)$
 $(i, 2), (j, 1), (c, 1)$
- Alles schön und gut, aber nach dem Aufruf von `swap` in `main` (beim Ausgabe mit `System.out`) hat `x` den Wert 1 und `y` den Wert 2, d.h. `swap` hat keinerlei Wirkung auf `x` und `y` in `main`

Und das ist genau das, was wir formalisiert haben!