Lernziele

Lernziele

- Sie wissen, wo Sie die Tabelle mit allen Operatoren finden

Lernziele

- Sie wissen, wo Sie die Tabelle mit allen Operatoren finden
- Sie verstehen den Aufbau eines *Fliesskommazahlensystems*

Lernziele

- Sie wissen, wo Sie die Tabelle mit allen Operatoren finden
- Sie verstehen den Aufbau eines *Fliesskommazahlensystems*
- Sie können die *Binärdarstellung* von Fliesskommazahlen berechnen

Lernziele

- Sie wissen, wo Sie die Tabelle mit allen Operatoren finden
- Sie verstehen den Aufbau eines *Fliesskommazahlensystems*
- Sie können die *Binärdarstellung* von Fliesskommazahlen berechnen
- Sie kennen die wichtigsten Kontrollstrukturen un können diese korrekt andwenden

Lernziele

- Sie wissen, wo Sie die Tabelle mit allen Operatoren finden
- Sie verstehen den Aufbau eines *Fliesskommazahlensystems*
- Sie können die *Binärdarstellung* von Fliesskommazahlen berechnen
- Sie kennen die wichtigsten Kontrollstrukturen un können diese korrekt andwenden
- Sie verstehen, wo eine Variable sichtbar ist, und können den *Gültigkeitsbereich* für eine Variable aufzeigen

6. Operatoren

Tabellarische Übersicht aller relevanten Operatoren

Operatoren: Tabelle

Beschreibung	Operator	Stelligkeit	Präzedenz	Assoziativität
Objekt-Member Zugriff	`.`	2	16	links
Array Zugriff	`[ ]`	2	16	links
Methodenaufruf	`( )`	2	16	links
Postfix Inkrement/Dekrement	`++` `--`	1	15	links
Präfix Inkrement/Dekrement	`++` `--`	1	14	rechts
Plus, Minus, Logisches Nicht	`+` `-` `!`	1	14	rechts
Typcast	`( )`	1	13	rechts
Objekterstellung	`new`	1	13	rechts
Multiplikativ	`*` `/` `%`	2	12	links
Additiv	`+` `-`	2	11	links
Stringkonkatination	`+`	2	11	links
Vergleiche	`<` `<=` `>` `>=`	2	9	links
Typvergleich	`instanceof`	2	9	links
(Nicht-)Gleichheit	`==` `!=`	2	8	links
Logisches Und	`&&`	2	4	links
Logisches Oder	`||`	2	3	links
Konditional	`? :`	3	2	rechts
Zuweisungen	`=` `+=` `-=` `*=` `/=` `%=`	2	1	rechts

Operatoren: Tabelle - Erklärungen

- Die Stelligkeit gibt die Anzahl der Operanden an
- Eine höhere Präzedenz bedeutet stärkere Bindung
- Bei gleicher Präzedenz wird gemäss der Assoziativität ausgewertet

7. Fliesskommazahlen

Fliesskommazahlensysteme;IEEE Standard;

Wir erinnern uns an letztes mal

```
public class Main {
    public static void main(String[] args) {
    Out.print("First number =? ");
    float n1 = In.readFloat();

    Out.print("Second number =? ");
    float n2 = In.readFloat();

    Out.print("Their difference =? ");
    float d = In.readFloat();

    Out.print("computed difference − input difference = ");
    Out.println(n1−n2−d);
    }
}
```

Eingabe 1.1

Eingabe 1.0

Eingabe 0.1

Ausgabe 2.2351742E-8

Ja was ist denn hier los?

Warum passiert das?

- Nicht alle reellen Zahlen können dargestellt werden
- Rundungsfehler können sich propagieren und verstärken im Verlauf der Programmausführung

Wir wollen verstehen, warum dies der Fall ist!

Definition: *Fliesskommazahlensysteme*

Ein Fliesskommazahlensystem beschreibt eine Teilmenge der reellen Zahlen durch Einschränkung von Präzission und Wertebereich.

Fliesskommazahlensysteme

Ein Fliesskommazahlensystem ist durch vier natürliche Zahlen definiert:

- $\beta \geq 2$, die Basis,

Fliesskommazahlensysteme

Ein Fliesskommazahlensystem ist durch vier natürliche Zahlen definiert:

- $\beta \geq 2$, die Basis,
- $p \geq 1$, die Präzision (Stellenzahl),

Fliesskommazahlensysteme

Ein Fliesskommazahlensystem ist durch vier natürliche Zahlen definiert:

- $\beta \geq 2$, die Basis,
- $p \geq 1$, die Präzision (Stellenzahl),
- $e_{\min}$, der kleinste Exponent,

Fliesskommazahlensysteme

Ein Fliesskommazahlensystem ist durch vier natürliche Zahlen definiert:

- $\beta \geq 2$, die Basis,
- $p \geq 1$, die Präzision (Stellenzahl),
- $e_{\min}$, der kleinste Exponent,
- $e_{\max}$, der grösste Exponent.

Fliesskommazahlensysteme

Ein Fliesskommazahlensystem ist durch vier natürliche Zahlen definiert:

- $\beta \geq 2$, die Basis,
- $p \geq 1$, die Präzision (Stellenzahl),
- $e_{\min}$, der kleinste Exponent,
- $e_{\max}$, der grösste Exponent.

Bezeichnung:

$$F(\beta, p, e_{\min}, e_{\max})$$

Fliesskommazahlensysteme

$F(\beta, p, e_{\min}, e_{\max})$ enthält die Zahlen

$$\pm \sum_{i=0}^{p-1} d_i \beta^{-i} \cdot \beta^e,$$

$$d_i \in \{0, \ldots, \beta - 1\}, \quad e \in \{e_{\min}, \ldots, e_{\max}\}.$$

Fliesskommazahlensysteme

$F(\beta, p, e_{\min}, e_{\max})$ enthält die Zahlen

$$\pm \sum_{i=0}^{p-1} d_i \beta^{-i} \cdot \beta^e,$$

$$d_i \in \{0, \ldots, \beta - 1\}, \quad e \in \{e_{\min}, \ldots, e_{\max}\}.$$

In Basis-β-Darstellung:

$$\pm\, d_{0\bullet} d_1 \ldots d_{p-1} \times \beta^e,$$

Fliesskommazahlensysteme

Beispiel

- $\beta = 10$

Darstellungen der Dezimalzahl 0.1

$$1.0 \cdot 10^{-1}, \quad 0.1 \cdot 10^{0}, \quad 0.01 \cdot 10^{1}, \quad \ldots$$

Definition: ***Normalisierte Darstellung***

Ein Darstellung ist normalisiert, genau dann wenn vor dem Komma eine einzelne Ziffer ungleich 0 steht.

Normalisierte Darstellung

Normalisierte Zahl:

$$\pm\, d_0{\bullet}d_1 \ldots d_{p-1} \times \beta^e, \qquad d_0 \neq 0$$

Bemerkung 1

Die normalisierte Darstellung ist eindeutig und deshalb zu bevorzugen.

Normalisierte Darstellung

Normalisierte Zahl:

$$\pm\, d_0{\bullet}d_1 \ldots d_{p-1} \times \beta^e, \qquad d_0 \neq 0$$

Bemerkung 1

Die normalisierte Darstellung ist eindeutig und deshalb zu bevorzugen.

Normalisierte Darstellung

Normalisierte Zahl:

$$\pm\, d_0{\bullet}d_1 \ldots d_{p-1} \times \beta^e, \qquad d_0 \neq 0$$

Bemerkung 2

Die Zahl 0 (und alle Zahlen kleiner als $\beta^{e_{\min}}$) haben keine normalisierte Darstellung (werden wir später beheben)!

Menge der normalisierten Zahlen

$$F^*(\beta, p, e_{\min}, e_{\max})$$

Normalisierte Darstellung

Beispiel $F^*(2, 3, -2, 2)$ (nur positive Zahlen)

$d_0{\bullet}d_1d_2$	$e = -2$	$e = -1$	$e = 0$	$e = 1$	$e = 2$
1.00_2	0.25	0.5	1	2	4
1.01_2	0.3125	0.625	1.25	2.5	5
1.10_2	0.375	0.75	1.5	3	6
1.11_2	0.4375	0.875	1.75	3.5	7

Normalisierte Darstellung

Beispiel $F^*(2, \mathbf{3}, -2, 2)$ (nur positive Zahlen)

$d_0{\bullet}d_1d_2$	$e=-2$	$e=-1$	$e=0$	$e=1$	$e=2$
$\mathbf{1.00_2}$	0.25	0.5	1	2	4
$\mathbf{1.01_2}$	0.3125	0.625	1.25	2.5	5
$\mathbf{1.10_2}$	0.375	0.75	1.5	3	6
$\mathbf{1.11_2}$	0.4375	0.875	1.75	3.5	7

Normalisierte Darstellung

Beispiel $F^*(2, 3, -2, 2)$ (nur positive Zahlen)

$d_0 {\bullet} d_1 d_2$	$\mathbf{e = -2}$	$e = -1$	$e = 0$	$e = 1$	$e = 2$
1.00_2	0.25	0.5	1	2	4
1.01_2	0.3125	0.625	1.25	2.5	5
1.10_2	0.375	0.75	1.5	3	6
1.11_2	0.4375	0.875	1.75	3.5	7

$1.00 \cdot 2^{-2} = \frac{1}{4}$ $\quad$ $1.11 \cdot 2^{2} = 7$

Normalisierte Darstellung

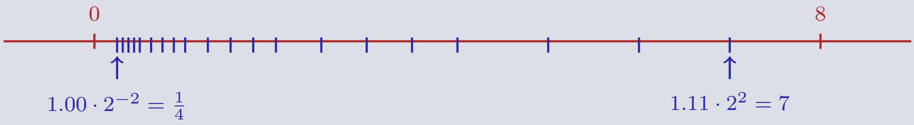
Beispiel $F^*(2, 3, -2, \mathbf{2})$ (nur positive Zahlen)

$d_0{\bullet}d_1d_2$	$e=-2$	$e=-1$	$e=0$	$e=1$	$\mathbf{e=2}$
1.00_2	0.25	0.5	1	2	4
1.01_2	0.3125	0.625	1.25	2.5	5
1.10_2	0.375	0.75	1.5	3	6
1.11_2	0.4375	0.875	1.75	3.5	7

Normalisierte Darstellung

Beispiel $F^*(2,3,-2,2)$ (nur positive Zahlen)

$d_0.d_1d_2$	$e=-2$	$e=-1$	$e=0$	$e=1$	$e=2$
1.00_2	0.25	0.5	1	2	4
1.01_2	0.3125	0.625	1.25	2.5	5
1.10_2	0.375	0.75	1.5	3	6
1.11_2	0.4375	0.875	1.75	3.5	7



Binäre und dezimale Systeme

- Intern rechnet der Computer mit $\beta = 2$ (binäres System)
- Literale und Eingaben haben $\beta = 10$ (dezimales System)

Binäre und dezimale Systeme

- Intern rechnet der Computer mit $\beta = 2$ (binäres System)
- Literale und Eingaben haben $\beta = 10$ (dezimales System)

Umrechnung $(0 < x < 2)$

Berechnung der *Binärdarstellung*:

$$x = \sum_{i=-\infty}^{0} b_i 2^i$$

Umrechnung $(0 < x < 2)$

Berechnung der *Binärdarstellung*:

$$x = b_0 \bullet b_{-1} b_{-2} b_{-3} \ldots$$

Umrechnung $(0 < x < 2)$

Berechnung der *Binärdarstellung*:

$$\begin{aligned} x &= b_0{\bullet}b_{-1}b_{-2}b_{-3}\dots \\ &= b_0 + 0{\bullet}b_{-1}b_{-2}b_{-3}\dots \end{aligned}$$

Umrechnung $(0 < x < 2)$

Berechnung der *Binärdarstellung*:

$$\begin{aligned} x &= b_0 \bullet b_{-1} b_{-2} b_{-3} \ldots \\ &= b_0 + 0 \bullet b_{-1} b_{-2} b_{-3} \ldots \\ &\Longrightarrow \end{aligned}$$

Umrechnung $(0 < x < 2)$

Berechnung der *Binärdarstellung*:

$$\begin{aligned}
x &= b_0{\bullet}b_{-1}b_{-2}b_{-3}\ldots \\
&= b_0 + 0{\bullet}b_{-1}b_{-2}b_{-3}\ldots \\
&\Longrightarrow \\
(x - b_0) &= 0{\bullet}b_{-1}b_{-2}b_{-3}b_{-4}\ldots
\end{aligned}$$

Umrechnung $(0 < x < 2)$

Berechnung der *Binärdarstellung*:

$$
\begin{aligned}
x &= b_0{\bullet}b_{-1}b_{-2}b_{-3}\dots \\
&= b_0 + 0{\bullet}b_{-1}b_{-2}b_{-3}\dots \\
&\Longrightarrow \\
2\cdot(x-b_0) &= b_{-1}{\bullet}b_{-2}b_{-3}b_{-4}\dots
\end{aligned}
$$

Umrechnung $(0 < x < 2)$

Berechnung der *Binärdarstellung*:

$$\begin{aligned} x &= b_0 \bullet b_{-1} b_{-2} b_{-3} \ldots \\ &= b_0 + 0 \bullet b_{-1} b_{-2} b_{-3} \ldots \\ &\Longrightarrow \\ 2 \cdot (x - b_0) &= b_{-1} \bullet b_{-2} b_{-3} b_{-4} \ldots \end{aligned}$$

Umrechnung $(0 < x < 2)$

Berechnung der *Binärdarstellung*:

$$\begin{aligned} x &= b_0 \bullet b_{-1} b_{-2} b_{-3} \dots \\ &= b_0 + 0 \bullet b_{-1} b_{-2} b_{-3} \dots \\ &\Longrightarrow \\ 2 \cdot (x - b_0) &= b_{-1} \bullet b_{-2} b_{-3} b_{-4} \dots \end{aligned}$$

Binärdarstellung von 1.1

x		b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 =$	$\mathbf{1}$		

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_{-1} = \mathbf{0}$		

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_{-1} = \mathbf{0}$	0.2	0.4

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_{-1} = \mathbf{0}$	0.2	0.4
0.4	$b_{-2} = \mathbf{0}$		

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_{-1} = \mathbf{0}$	0.2	0.4
0.4	$b_{-2} = \mathbf{0}$	0.4	0.8

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_{-1} = \mathbf{0}$	0.2	0.4
0.4	$b_{-2} = \mathbf{0}$	0.4	0.8
0.8	$b_{-3} = \mathbf{0}$		

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_{-1} = \mathbf{0}$	0.2	0.4
0.4	$b_{-2} = \mathbf{0}$	0.4	0.8
0.8	$b_{-3} = \mathbf{0}$	0.8	1.6

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_{-1} = \mathbf{0}$	0.2	0.4
0.4	$b_{-2} = \mathbf{0}$	0.4	0.8
0.8	$b_{-3} = \mathbf{0}$	0.8	1.6
1.6	$b_{-4} = \mathbf{1}$		

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_{-1} = \mathbf{0}$	0.2	0.4
0.4	$b_{-2} = \mathbf{0}$	0.4	0.8
0.8	$b_{-3} = \mathbf{0}$	0.8	1.6
1.6	$b_{-4} = \mathbf{1}$	0.6	1.2

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_{-1} = 0$	0.2	0.4
0.4	$b_{-2} = 0$	0.4	0.8
0.8	$b_{-3} = 0$	0.8	1.6
1.6	$b_{-4} = 1$	0.6	1.2
1.2	$b_{-5} = 1$		

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_{-1} = 0$	0.2	0.4
0.4	$b_{-2} = 0$	0.4	0.8
0.8	$b_{-3} = 0$	0.8	1.6
1.6	$b_{-4} = 1$	0.6	1.2
1.2	$b_{-5} = 1$	0.2	0.4

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_{-1} = 0$	0.2	0.4
0.4	$b_{-2} = 0$	0.4	0.8
0.8	$b_{-3} = 0$	0.8	1.6
1.6	$b_{-4} = 1$	0.6	1.2
1.2	$b_{-5} = 1$	0.2	0.4

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_{-1} = 0$	0.2	0.4
0.4	$b_{-2} = 0$	0.4	0.8
0.8	$b_{-3} = 0$	0.8	1.6
1.6	$b_{-4} = 1$	0.6	1.2
1.2	$b_{-5} = 1$	0.2	0.4

$\Rightarrow$ $1.0\overline{0011}$, periodisch, *nicht* endlich

Binärdarstellungen von 1.1 und 0.1

- sind nicht endlich $\Rightarrow$ Fehler bei der Konversion
- `1.1f` und `0.1f`: *Approximationen* von 1.1 und 0.1

$$\begin{aligned} \texttt{1.1} &= \underline{1.1000000000000000}888178\ldots \\ \texttt{1.1f} &= \underline{1.1000000}238418\ldots \end{aligned}$$

Binärdarstellungen von 1.1 und 0.1

- sind nicht endlich $\Rightarrow$ Fehler bei der Konversion
- `1.1f` und `0.1f`: *Approximationen* von 1.1 und 0.1

$$\begin{aligned} \texttt{1.1} &= \underline{1.1000000000000000}888178\ldots \\ \texttt{1.1f} &= \underline{1.1000000}238418\ldots \end{aligned}$$

Binärdarstellungen von 1.1 und 0.1

- sind nicht endlich $\Rightarrow$ Fehler bei der Konversion
- `1.1f` und `0.1f`: *Approximationen* von 1.1 und 0.1

$$
\begin{array}{rcl}
\texttt{1.1} & = & \underline{1.10000000000000000}888178\ldots \\
\texttt{1.1f} & = & \underline{1.1000000}238418\ldots
\end{array}
$$

Rechnen mit Fliesskommazahlen

ist fast so einfach wie mit ganzen Zahlen.

Rechnen mit Fliesskommazahlen

Beispiel ($\beta = 2$, $p = 4$):

$$\begin{array}{rr} & 1.111 \cdot 2^{-2} \\ + & 1.011 \cdot 2^{-1} \end{array}$$

1. Exponenten anpassen durch Denormalisieren einer Zahl

Rechnen mit Fliesskommazahlen

Beispiel ($\beta = 2$, $p = 4$):

$$
\begin{array}{rr}
 & 1.111 \cdot 2^{-2} \\
+ & 10.110 \cdot 2^{-2} \checkmark
\end{array}
$$

1. Exponenten anpassen durch Denormalisieren einer Zahl

Rechnen mit Fliesskommazahlen

Beispiel ($\beta = 2$, $p = 4$):

$$\begin{array}{rr} & 1.111 \cdot 2^{-2} \\ + & 10.110 \cdot 2^{-2} \\ \hline \end{array}$$

2. Binäre Addition der Signifikanden

Rechnen mit Fliesskommazahlen

Beispiel ($\beta = 2$, $p = 4$):

$$
\begin{array}{rr}
 & 1.111 \cdot 2^{-2} \\
+ & 10.110 \cdot 2^{-2} \\
\hline
 & = 100.101 \cdot 2^{-2} \checkmark
\end{array}
$$

2. Binäre Addition der Signifikanden

Rechnen mit Fliesskommazahlen

Beispiel ($\beta = 2$, $p = 4$):

$$
\begin{array}{rr}
 & 1.111 \cdot 2^{-2} \\
+ & 10.110 \cdot 2^{-2} \\
\hline
= & 100.101 \cdot 2^{-2}
\end{array}
$$

3. Renormalisierung

Rechnen mit Fliesskommazahlen

Beispiel ($\beta = 2$, $p = 4$):

$$
\begin{array}{rr}
 & 1.111 \cdot 2^{-2} \\
+ & 10.110 \cdot 2^{-2} \\
\hline
 & = 1.00101 \cdot 2^{0} \checkmark
\end{array}
$$

3. Renormalisierung

Rechnen mit Fliesskommazahlen

Beispiel ($\beta = 2$, $p = 4$):

$$
\begin{array}{rr}
 & 1.111 \cdot 2^{-2} \\
+ & 10.110 \cdot 2^{-2} \\
\hline
 & = 1.00101 \cdot 2^{0}
\end{array}
$$

4. Runden auf p signifikante Stellen, falls nötig

Rechnen mit Fliesskommazahlen

Beispiel ($\beta = 2$, $p = 4$):

$$\begin{array}{rr} & 1.111 \cdot 2^{-2} \\ + & 10.110 \cdot 2^{-2} \\ \hline & = 1.001 \cdot 2^{0} \checkmark \end{array}$$

4. Runden auf p signifikante Stellen, falls nötig

Der IEEE Standard 754

Legt Fliesskommazahlensysteme und deren Rundungsverhalten fest

Der IEEE Standard 754

Legt Fliesskommazahlensysteme und deren Rundungsverhalten fest

- Single precision (`float`) Zahlen:

  $F^*(2, 24, -126, 127)$ plus 0, ∞, $\ldots$

Der IEEE Standard 754

Legt Fliesskommazahlensysteme und deren Rundungsverhalten fest

- Single precision (`float`) Zahlen:

  $F^*(2, 24, -126, 127)$ plus 0, ∞, $\ldots$

- Double precision (`double`) Zahlen:

  $F^*(2, 53, -1022, 1023)$ plus 0, ∞, $\ldots$

Der IEEE Standard 754

Legt Fliesskommazahlensysteme und deren Rundungsverhalten fest

- Single precision (`float`) Zahlen:

  $F^*(2, 24, -126, 127)$ plus 0, ∞, $\ldots$

- Double precision (`double`) Zahlen:

  $F^*(2, 53, -1022, 1023)$ plus 0, ∞, $\ldots$

- Alle arithmetischen Operationen runden das *exakte* Ergebnis auf die nächste darstellbare Zahl

32-bit Darstellung einer Fliesskommazahl

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

$\pm$ Exponent Mantisse

$$\pm \begin{array}{c} 2^{-126}, \ldots, 2^{127} \\ 0, \infty, \ldots \end{array}$$

$$\begin{array}{c} 1.00000000000000000000000 \\ \ldots \\ 1.11111111111111111111111 \end{array}$$

8. Kontrollanweisungen

Auswahlanweisungen, Iterationsanweisungen, Terminierung, Blöcke, Sichtbarkeit, Lokale Variablen, Switch-Answeisung

Anweisungen (Statements)

Eine Anweisung ist ...

- vergleichbar mit einem Satz in der natürlichen Sprache
- eine komplette Ausführungseinheit
- immer mit einem *Semikolon* abgeschlossen

Beispiel

```
f = 9f * celsius / 5 + 32 ;
```

Anweisungenarten

Gültige Anweisungen sind:

- Deklarationsanweisung
- Wertzuweisungen
- Inkrement / Dekrement Ausdrücke
- Methodenaufrufe
- Objekterzeugungs-Ausdrücke
- Nullanweisung

Anweisungenarten

Beispiele

```
float aValue;
aValue = 8933.234;
aValue++;
Out.println(aValue);
new Student();
;
```

Blöcke

Ein Block ist . . .

- eine Gruppe von Anweisungen
- überall erlaubt wo Anweisungen erlaubt sind
- durch geschweiften Klammern markiert

```
{
   statement1
   statement2
   ⋮
}
```

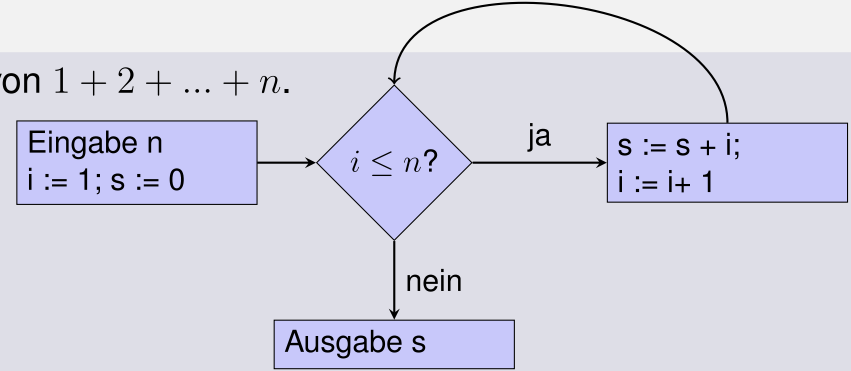
Kontrollfluss

- bisher *linear* (von oben nach unten)
- Für interessante Programme braucht man „Verzweigungen” und „Sprünge”.

Kontrollfluss

- bisher *linear* (von oben nach unten)
- Für interessante Programme braucht man „Verzweigungen” und „Sprünge”.

Berechung von $1 + 2 + ... + n$.



Auswahlanweisungen

realisieren Verzweigungen

- `if` Anweisung
- `if-else` Anweisung

`if`-Anweisung

`if` (*condition*)
    statement

if-Anweisung

`if` (*condition*)
    statement

```
int a = In.readInt();
if (a % 2 == 0) {
    Out.println("even");
}
```

`if`-Anweisung

`if` (*condition*)
    statement

Ist *condition* wahr, dann wird *statement* ausgeführt.

```
int a = In.readInt();
if (a % 2 == 0) {
    Out.println("even");
}
```

`if`-Anweisung

`if` (*condition*)
statement

```
int a = In.readInt();
if (a % 2 == 0) {
    Out.println("even");
}
```

- *statement*: beliebige Anweisung (*Rumpf* der `if`-Anweisung)
- *condition*: Ausdruck vom Typ `boolean`

`if-else`-Anweisung

`if` (*condition*)
    statement1
else
    statement2

if-else-Anweisung

```
if ( condition )
    statement1
else
    statement2
```

```
int a = In.readInt();
if (a % 2 == 0){
    Out.println("even");
} else {
    Out.println("odd");
}
```

`if-else`-Anweisung

```
if ( condition )
    statement1
else
    statement2
```

Ist *condition* wahr, so wird *statement1* ausgeführt, andernfalls wird *statement2* ausgeführt.

```java
int a = In.readInt();
if (a % 2 == 0){
    Out.println("even");
} else {
    Out.println("odd");
}
```

`if-else`-Anweisung

`if` (*condition*)
    statement1
else
    statement2

```
int a = In.readInt();
if (a % 2 == 0){
    Out.println("even");
} else {
    Out.println("odd");
}
```

- *condition*: Ausdruck vom Typ `boolean`
- *statement1*: *Rumpf* des `if`-Zweiges
- *statement2*: *Rumpf* des `else`-Zweiges

Layout!

```
int a = In.readInt();
if (a % 2 == 0){
    Out.println("even");
} else {
    Out.println("odd");
}
```

Layout!

```
int a = In.readInt();
if (a % 2 == 0){
    Out.println("even");
} else {
    Out.println("odd");
}
```

Einrückung

Einrückung

Iterationsanweisungen

realisieren „Schleifen”:

- `for`-Anweisung
- `while`-Anweisung
- `do`-Anweisung

Beispiel: Berechne $1 + 2 + \ldots + n$

```
// input
Out.print("Compute the sum 1+...+n for n=?");
int n = In.readInt();

// computation of sum_{i=1}^n i
int s = 0;
for (int i = 1; i <= n; ++i){
    s += i;
}

// output
Out.println("1+...+" + n + " = " + s);
```

Beispiel: Berechne $1 + 2 + ... + n$

```
// input
Out.print("Compute the sum 1+...+n for n=?");
int n = In.readInt();

// computation of sum_{i=1}^n i
int s = 0;
for (int i = 1; i <= n; ++i){
    s += i;
}

// output
Out.println("1+...+" + n + " = " + s);
```

`for`-Anweisung: Syntax

`for` (*init statement* *condition* ; *expression*)
    statement

`for`-Anweisung: Syntax

`for` (*init statement* *condition* ; *expression*)
    statement

Deklarationsanweisung

auch möglich: Ausdrucksanweisung, Nullanweisung

```
for ( int i=1;  i <= n ; ++i ) {
    s += i; // Rumpf
}
```

`for`-Anweisung: Syntax

`for` (*init statement* *condition* ; *expression*)
      statement

Ausdruck vom Typ `boolean`

```
for ( int i=1;  i <= n ; ++i ) {
    s += i; // Rumpf
}
```

`for`-Anweisung: Syntax

`for` (*init statement* *condition* ; *expression*)
statement

Ausdruck vom Typ `int`

```
for ( int i=1;  i <= n ; ++i ) {
    s += i; // Rumpf
}
```

`for`-Anweisung: Syntax

`for` (*init statement* *condition* ; *expression*)
    statement

Ausdrucksanweisung

```
for ( int i=1; i <= n ; ++i ) {
    s += i; // Rumpf
}
```

Beispiel: Harmonische Zahlen

- Die n-te Harmonische Zahl ist

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln n.$$

Beispiel: Harmonische Zahlen

- Die n-te Harmonische Zahl ist

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln n.$$

- Diese Summe kann vorwärts oder rückwärts berechnet werden, was mathematisch gesehen natürlich äquivalent ist.

Beispiel: Harmonische Zahlen

```
Out.print("Compute H_n for n =? ");
int n = In.readInt();

float fs = 0;
for (int i = 1; i <= n; ++i){
    fs += 1.0f / i;
}
Out.println("Forward sum = " + fs);

float bs = 0;
for (int i = n; i >= 1; --i){
    bs += 1.0f / i;
}
Out.println("Backward sum = " + bs);
```

Beispiel: Harmonische Zahlen

```
Out.print("Compute H_n for n =? ");
int n = In.readInt();

float fs = 0;
for (int i = 1; i <= n; ++i){
    fs += 1.0f / i;
}
Out.println("Forward sum = " + fs);

float bs = 0;
for (int i = n; i >= 1; --i){
    bs += 1.0f / i;
}
Out.println("Backward sum = " + bs);
```

Eingabe: **10’000’000**

Vorwärts: **15.4037**

Rückwärts: **16.686**

Beispiel: Harmonische Zahlen

```
Out.print("Compute H_n for n =? ");
int n = In.readInt();

float fs = 0;
for (int i = 1; i <= n; ++i){
    fs += 1.0f / i;
}
Out.println("Forward sum = " + fs);

float bs = 0;
for (int i = n; i >= 1; --i){
    bs += 1.0f / i;
}
Out.println("Backward sum = " + bs);
```

Eingabe: **100’000’000**

Vorwärts: **15.4037**

Rückwärts: **18.8079**

Beispiel: Harmonische Zahlen

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist “richtig” falsch.
- Die Rückwärtssumme approximiert H_n gut.

Beispiel: Harmonische Zahlen

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist "richtig" falsch.
- Die Rückwärtssumme approximiert H_n gut.

Beispiel: Harmonische Zahlen

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist “richtig” falsch.
- Die Rückwärtssumme approximiert H_n gut.

Erklärung:

- Bei $1 + 1/2 + 1/3 + \cdots$ sind späte Terme zu klein, um noch beizutragen.
- *Fliesskomma Regel 2*

Beispiel: Harmonische Zahlen

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist "richtig" falsch.
- Die Rückwärtssumme approximiert H_n gut.

Erklärung:

- Bei $1 + 1/2 + 1/3 + \cdots$ sind späte Terme zu klein, um noch beizutragen.
- *Fliesskomma Regel 2*

Beispiel: Harmonische Zahlen

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist “richtig” falsch.
- Die Rückwärtssumme approximiert H_n gut.

Erklärung:

- Bei $1 + 1/2 + 1/3 + \cdots$ sind späte Terme zu klein, um noch beizutragen.
- *Fliesskomma Regel 2*

Beispiel: Primzahltest

Def.: Eine natürliche Zahl $n \geq 2$ ist eine Primzahl, wenn kein $d \in \{2, \ldots, n-1\}$ ein Teiler von n ist.

Beispiel: Primzahltest

Def.: Eine natürliche Zahl $n \geq 2$ ist eine Primzahl, wenn kein $d \in \{2, \ldots, n-1\}$ ein Teiler von n ist.

Eine Schleife, die das testet:

```
int d;
for (d=2; n%d != 0; ++d);
```

Beispiel: Primzahltest

Def.: Eine natürliche Zahl $n \geq 2$ ist eine Primzahl, wenn kein $d \in \{2, \ldots, n-1\}$ ein Teiler von n ist.

Eine Schleife, die das testet:

```
int d;
for (d=2; n%d != 0; ++d);
```

(Rumpf ist die Null-Anweisung)

Primzahltest: Terminierung

```
int d;
for (d=2; n%d != 0; ++d);
```

- Fortschritt: Startwert `d=2`, dann in jeder Iteration plus 1 (`++d`)

Primzahltest: Terminierung

```
int d;
for (d=2; n%d != 0; ++d);
```

- Fortschritt: Startwert `d=2`, dann in jeder Iteration plus 1 (`++d`)
- Abbruch: `n%d != 0` evaluiert zu `true` sobald ein Teiler erreicht wurde — spätestes, wenn `d == n`

Primzahltest: Terminierung

```
int d;
for (d=2; n%d != 0; ++d);
```

- Fortschritt: Startwert `d=2`, dann in jeder Iteration plus 1 (`++d`)
- Abbruch: `n%d != 0` evaluiert zu `true` sobald ein Teiler erreicht wurde — spätestes, wenn `d == n`
- Fortschritt garantiert, dass Abbruchbedingung erreicht wird

Primzahltest: Korrektheit

```
int d;
for (d=2; n%d != 0; ++d); // for n >= 2
```

Jeder mögliche Teiler `2 <= d <= n` wird ausprobiert. Falls die Schleife mit `d == n` terminiert, dann und genau dann ist `n` prim.

Endlosschleifen

- Endlosschleifen sind leicht zu produzieren:

```
for ( ; ; ) ;
```

  - Die *leere condition* ist wahr.
  - Die *leere expression* hat keinen Effekt.
  - Die *Nullanweisung* hat keinen Effekt.

Endlosschleifen

- Endlosschleifen sind leicht zu produzieren:

```
for ( ; ; ) ;
```

  - Die *leere condition* ist wahr.
  - Die *leere expression* hat keinen Effekt.
  - Die *Nullanweisung* hat keinen Effekt.

- ... aber nicht automatisch zu erkennen.

```
for ( e; v; e) r;
```

Halteproblem

Unentscheidbarkeit des Halteproblems

Es gibt kein Java Programm, das für jedes Java- Programm P und jede Eingabe I korrekt feststellen kann, ob das Programm P bei Eingabe von I terminiert.

[4] Alan Turing, 1936. Theoretische Fragestellungen dieser Art waren für Alan Turing die Hauptmotivation für die Konstruktion seiner Rechenmaschine.

Halteproblem

Unentscheidbarkeit des Halteproblems

Es gibt kein Java Programm, das für jedes Java- Programm P und jede Eingabe I korrekt feststellen kann, ob das Programm P bei Eingabe von I terminiert.

Das heisst, die Korrektheit von Programmen kann *nicht* automatisch überprüft werden.[4]

[4] Alan Turing, 1936. Theoretische Fragestellungen dieser Art waren für Alan Turing die Hauptmotivation für die Konstruktion seiner Rechenmaschine.

Beispiel: Die Collatz-Folge $(n \in \mathbb{N})$

- $n_0 = n$
- $n_i = \begin{cases} \dfrac{n_{i-1}}{2} & \text{, falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & \text{, falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1.$

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & \text{, falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & \text{, falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1.$

n=5: 5

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & \text{, falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & \text{, falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1.$

n=5: 5, 16

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & \text{, falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & \text{, falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1.$

n=5: 5, 16, 8

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & \text{, falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & \text{, falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1.$

n=5: 5, 16, 8, 4

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & \text{, falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & \text{, falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1.$

n=5: 5, 16, 8, 4, 2

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & \text{, falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & \text{, falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1.$

n=5: 5, 16, 8, 4, 2, 1

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & \text{, falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & \text{, falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1.$

n=5: 5, 16, 8, 4, 2, 1, 4

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & \text{, falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & \text{, falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1.$

n=5: 5, 16, 8, 4, 2, 1, 4, 2

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & \text{, falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & \text{, falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1.$

n=5: 5, 16, 8, 4, 2, 1, 4, 2, 1

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & \text{, falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & \text{, falls } n_{i-1} \text{ ungerade} \end{cases}, i \geq 1.$

n=5: 5, 16, 8, 4, 2, 1, 4, 2, 1, ... (Repetition bei 1)

Die Collatz-Folge in Java

```
// Input
Out.println("Compute Collatz sequence, n =? ");
int n = In.readInt();

// Iteration
while (n > 1) {        // stop when 1 reached
    if (n % 2 == 0) { // n is even
        n = n / 2;
    } else {             // n is odd
        n = 3 * n + 1;
    }
    Out.print(n + " ");
}
```

Die Collatz-Folge in Java

```
n = 27:
82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242,
121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233,
700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336,
668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276,
638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429,
7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232,
4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488,
244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20,
10, 5, 16, 8, 4, 2, 1
```

`while`-Anweisung: Warum?

- Bei `for`-Anweisung ist oft expression allein für den Fortschritt zuständig („Zählschleife”)

```
for (int i = 1; i <= n; ++i){
    s += i;
}
```

`while`-Anweisung: Warum?

- Bei `for`-Anweisung ist oft expression allein für den Fortschritt zuständig („Zählschleife”)

```
for (int i = 1; i <= n; ++i){
    s += i;
}
```

- Falls der Fortschritt nicht so einfach ist, kann `while` besser lesbar sein.

`while` Anweisung

`while` (*condition*)
  statement

`while` Anweisung

`while` (*condition*)
  statement

ist äquivalent zu

`for` (; *condition* ;)
  statement

Beispiel: Mini-Taschenrechner

```
int a;    // next input value
int s = 0; // sum of values so far
do {
    Out.print("next number =? ");
    a = In.readInt();
    s += a;
    Out.println("sum = " + s);
} while (a != 0);
```

`do` Anweisung

```
do
  statement
while ( expression );
```

`do` Anweisung

```
do
  statement
while ( expression );
```

ist äquivalent zu

```
statement
while ( expression )
  statement
```

Blöcke

- Beispiel: Rumpf der main Funktion

```
public static void main(String[] args) {
    ...
}
```

Blöcke

- Beispiel: Schleifenrumpf

```
for (int i = 1; i <= n; ++i) {
    s += i;
    Out.println("partial sum is " + s);
}
```

Blöcke

- Beispiel: if / else

```
if (d < n) { // d is a divisor of n in {2,...,n−1}
    Out.println(n + " = " + d + " * " + n / d);
} else {
    assert (d == n);
    Out.println(n + " is prime");
}
```

Sichtbarkeit

Deklaration in einem Block ist ausserhalb des Blocks nicht „sichtbar".

```
public static void main(String[] args)
{
    {
        int i = 2;
    }
    Out.println(i); // Fehler: undeklarierter Name
}
```

Sichtbarkeit

Deklaration in einem Block ist ausserhalb des Blocks nicht „sichtbar”.

```
public static void main(String[] args)
{
    {
        int i = 2;
    }
    Out.println(i); // Fehler: undeklarierter Name
}
```

main block

block

„Blickrichtung”

Kontrollanweisung definiert Block

Kontrollanweisungen verhalten sich in diesem Zusammenhang wie Blöcke.

```
public static void main(String[] args) {
{
    for (int i = 0; i < 10; ++i){
        s += i;
    }
    Out.println(i); // Fehler: undeklarierter Name
}
```

Kontrollanweisung definiert Block

Kontrollanweisungen verhalten sich in diesem Zusammenhang wie Blöcke.

```
public static void main(String[] args) {
{
    for (int i = 0; i < 10; ++i){
        s += i;
    }
    Out.println(i); // Fehler: undeklarierter Name
}
```

block

Gültigkeitsbereich

Im Block

```
{
    int i = 2;
    ...
}
```

Im Funktionsrumpf

```
void main(String[] args) {
    int i = 2;
    ...
}
```

In Kontrollanweisung

```
for ( int i = 0; i < 10; ++i) {s += i; ... }
```

Gültigkeitsbereich

Im Block

scope

```
{
    int i = 2;
    ...
}
```

Im Funktionsrumpf

scope

```
void main(String[] args) {
    int i = 2;
    ...
}
```

In Kontrollanweisung

```
for ( int i = 0; i < 10; ++i) {s += i; ... }
```

scope

Lokale Variablen

```
public static void main(String[] args) {
    int i = 5;
    for (int j = 0; j < 5; ++j) {
        Out.println(++i); // outputs
        int k = 2;
        Out.println(--k); // outputs
    }
}
```

Lokale Variablen

```
public static void main(String[] args) {
    int i = 5;
    for (int j = 0; j < 5; ++j) {
        Out.println(++i); // outputs 6, 7, 8, 9, 10
        int k = 2;
        Out.println(--k); // outputs 1, 1, 1, 1, 1
    }
}
```

Lokale Variablen

```
public static void main(String[] args) {
    int i = 5;
    for (int j = 0; j < 5; ++j) {
        Out.println(++i); // outputs 6, 7, 8, 9, 10
        int k = 2;
        Out.println(--k); // outputs 1, 1, 1, 1, 1
    }
}
```

Lokale Variablen (Deklaration in einem Block) haben *automatische Speicherdauer*.

Die „richtige” Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

Die „richtige” Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

- Wenige Anweisungen

Die „richtige” Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

- Wenige Anweisungen
- Wenige Zeilen Code

Die „richtige” Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

- Wenige Anweisungen
- Wenige Zeilen Code
- Einfacher Kontrollfluss

Die „richtige” Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

- Wenige Anweisungen
- Wenige Zeilen Code
- Einfacher Kontrollfluss
- Einfache Ausdrücke

Die „richtige” Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

- Wenige Anweisungen
- Wenige Zeilen Code
- Einfacher Kontrollfluss
- Einfache Ausdrücke

Ziele sind oft nicht gleichzeitig erreichbar.

Ungerade Zahlen in $\{0, \ldots, 100\}$

Erster (korrekter) Versuch:

```
for (int i = 0; i < 100; ++i) {
    if (i % 2 == 0){
        continue;
    }
    Out.println(i);
}
```

Ungerade Zahlen in $\{0, \ldots, 100\}$

Weniger Anweisungen, *weniger* Zeilen:

```
for (int i = 0; i < 100; ++i) {
    if (i % 2 != 0){
        Out.println(i);
    }
}
```

Ungerade Zahlen in $\{0, \ldots, 100\}$

Weniger Anweisungen, *einfacherer* Kontrollfluss:

```
for (int i = 1; i < 100; i += 2) {
    Out.println(i);
}
```

Ungerade Zahlen in $\{0, \ldots, 100\}$

Weniger Anweisungen, *einfacherer* Kontrollfluss:

```
for (int i = 1; i < 100; i += 2) {
    Out.println(i);
}
```

Das ist hier die “richtige” Iterationsanweisung!

Die `switch`-Anweisung

`switch` (*condition*)
statement

Die `switch`-Anweisung

`switch` (*condition*)
statement

```
int Note;
...
switch (Note) {
    case 6:
        Out.print("super!");
        break;
    case 5:
        Out.print("gut!");
        break;
    case 4:
        Out.print("ok!");
        break;
    default:
        Out.print("schade.");
}
```

Die `switch`-Anweisung

`switch` (*condition*)
statement

Die `switch`-Anweisung

`switch` (*condition*)
statement

- *condition*: Ausdruck, konvertierbar in einen integralem Typ
- *statement* : beliebige Anweisung, in welcher `case` und `default`-Marken erlaubt sind, `break` hat eine spezielle Bedeutung.

Kontrollfluss `switch` allgemein

Fehlt **`break`**, geht es mit dem nächsten Fall weiter.

7: Keine Note!
6: bestanden!
5: bestanden!
4: bestanden!
3: oops!
2: ooops!
1: oooops!
0: Keine Note!

```
switch (Note) {
    case 6:
    case 5:
    case 4:
        Out.print("bestanden!");
        break;
    case 1:
        Out.print("o");
    case 2:
        Out.print("o");
    case 3:
        Out.print("oops!");
        break;
    default:
        Out.print("Keine Note!");
}
```

Definition: *Kontrollfluss*

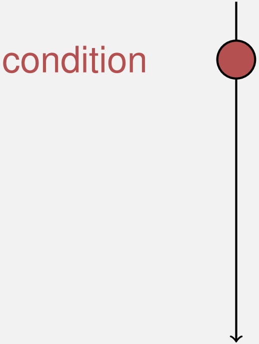
Reihenfolge der (wiederholten) Ausführung von Anweisungen

Kontrollfluss

- Grundsätzlich von oben nach unten. . .

Kontrollfluss

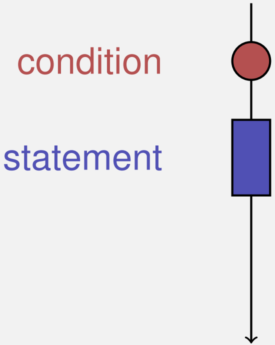
- Grundsätzlich von oben nach unten. . .
- . . . ausser in Auswahl- und Kontrollanweisungen



`if` (*condition*)
    statement

Kontrollfluss

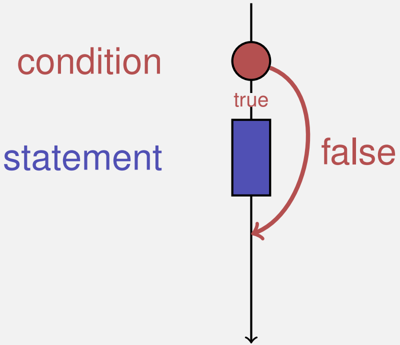
- Grundsätzlich von oben nach unten. . .
- . . . ausser in Auswahl- und Kontrollanweisungen



`if` (*condition*)
    statement

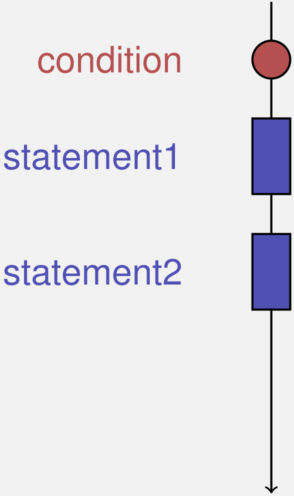
Kontrollfluss

- Grundsätzlich von oben nach unten. . .
- . . . ausser in Auswahl- und Kontrollanweisungen



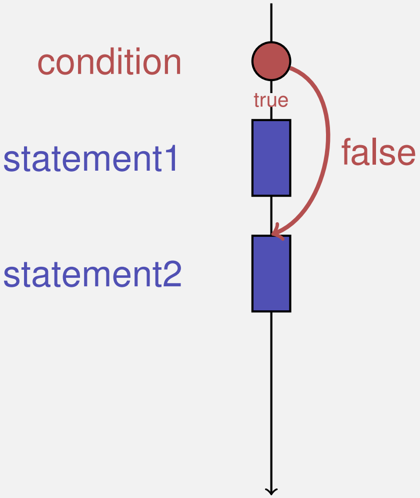
`if` (*condition*)
  statement

Kontrollfluss `if else`



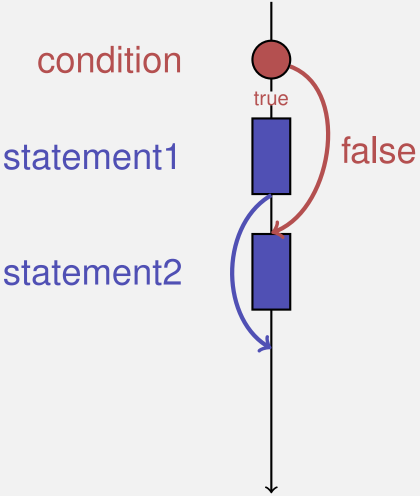
`if` (*condition*)
    statement1
else
    statement2

Kontrollfluss `if else`



`if` (*condition*)
    statement1
else
    statement2

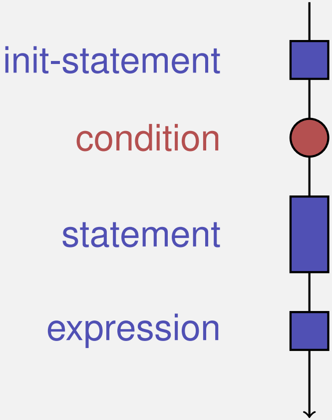
Kontrollfluss `if else`



`if` (*condition*)
    statement1
else
    statement2

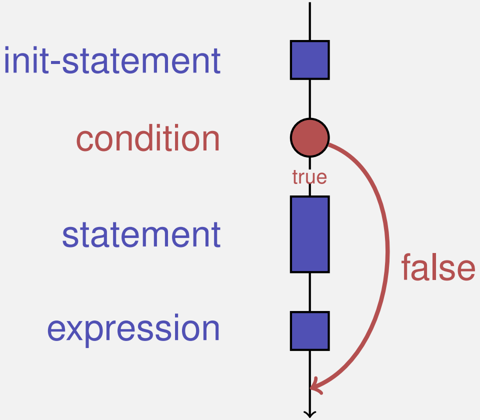
Kontrollfluss `for`

`for` (*init statement* *condition* ; *expression*)
    statement



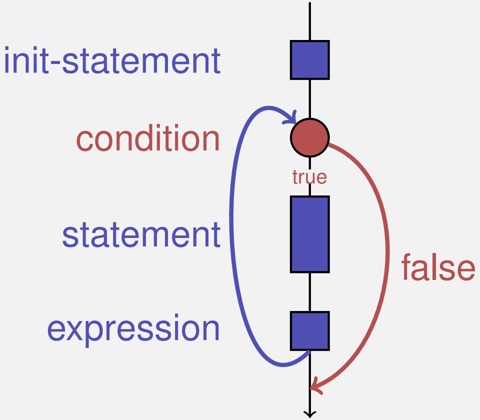
Kontrollfluss `for`

`for` (*init statement* *condition* ; *expression*)
    statement

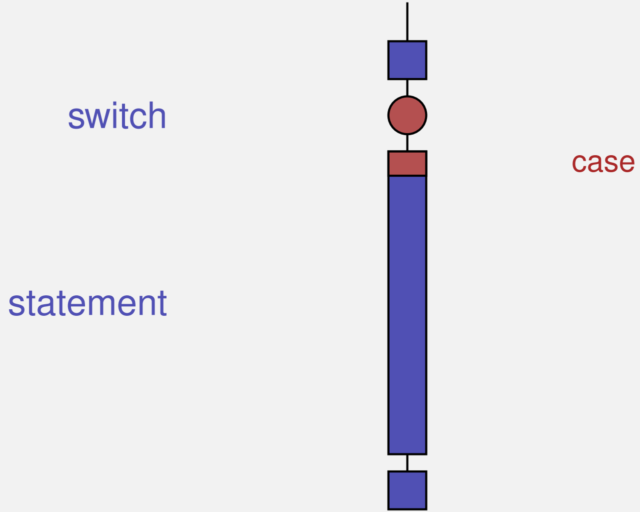


Kontrollfluss `for`

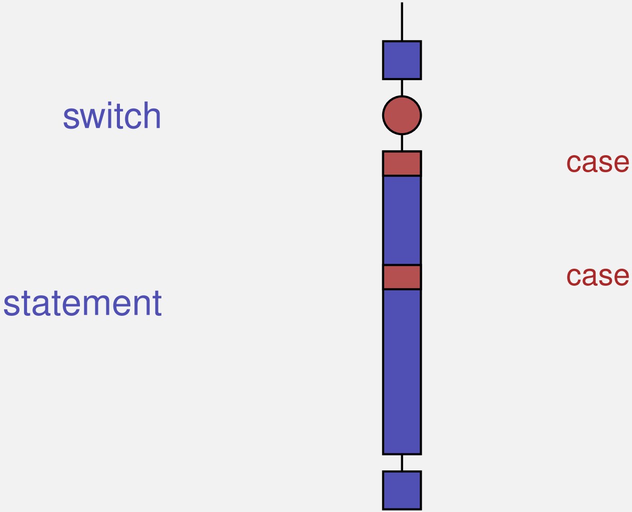
`for` (*init statement* *condition* ; *expression*)
    statement



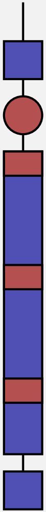
Kontrollfluss `switch`



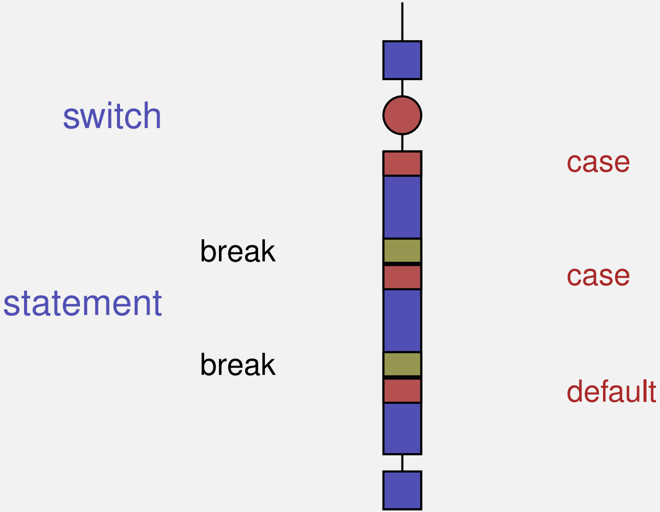
Kontrollfluss `switch`



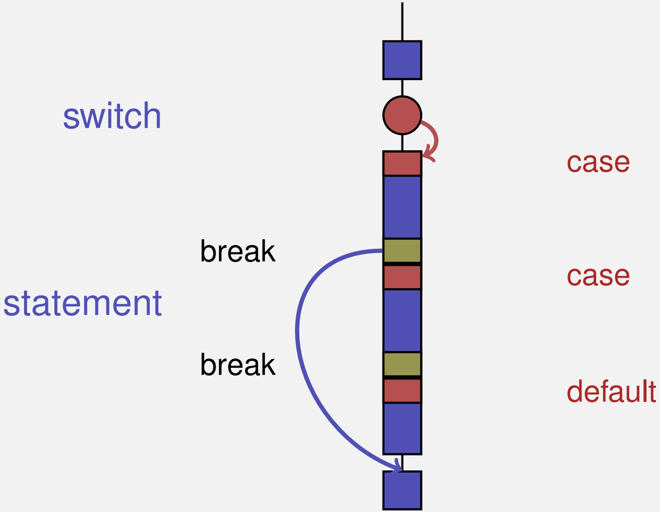
Kontrollfluss `switch`



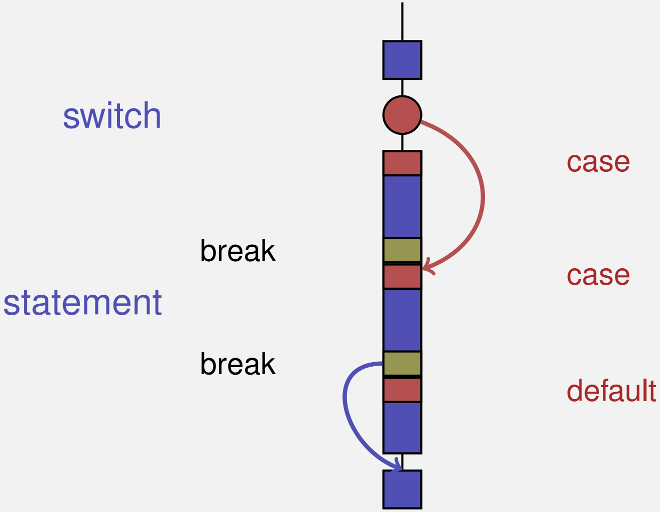
Kontrollfluss `switch`



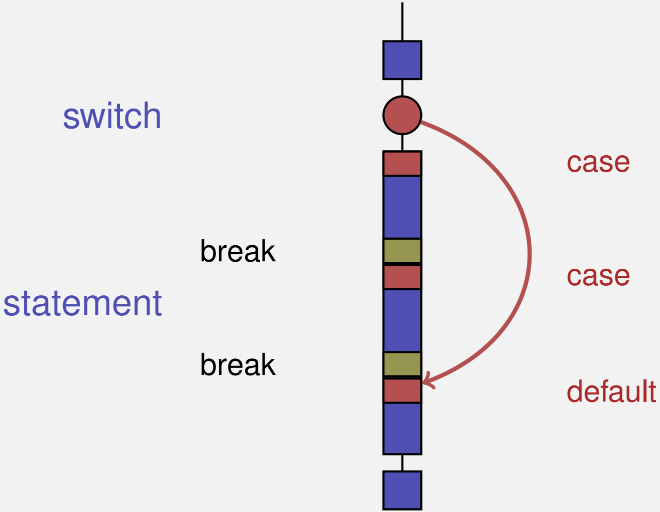
Kontrollfluss `switch`



Kontrollfluss `switch`



Kontrollfluss `switch`



Kontrollfluss `switch`