

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 und können diese korrekt anwenden
- Sie verstehen, wo eine Variable sichtbar ist, und können den *Gültigkeitsbereich* für eine Variable aufzeigen

6. Operatoren

Tabellarische Übersicht aller relevanten Operatoren

174

175

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
Stringkonkationation	+	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

176

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

177

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?

178

179

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 Präzision und Wertebereich.

180

181

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:

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

182

Fliesskommazahlensysteme

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

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

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

In Basis- β -Darstellung:

$$\pm d_0 \bullet d_1 \dots d_{p-1} \times \beta^e,$$

183

Fliesskommazahlensysteme

Beispiel

- $\beta = 10$

Darstellungen der Dezimalzahl 0.1

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

184

Definition: *Normalisierte Darstellung*

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

185

Normalisierte Darstellung

Normalisierte Zahl:

$$\pm d_0.d_1 \dots d_{p-1} \times \beta^e, \quad d_0 \neq 0$$

Bemerkung 1

Die normalisierte Darstellung ist eindeutig und deshalb zu bevorzugen.

Bemerkung 2

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

186

Menge der normalisierten Zahlen

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

187

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



188

Binäre und dezimale Systeme

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

189

Umrechnung dezimal → binär

Angenommen, $0 < x < 2$.

- Also: $x' = b_{-1}b_{-2}b_{-3}b_{-4} \dots = 2 \cdot (x - b_0)$
- Schritt 1 (für x): Berechnen von b_0 :

$$b_0 = \begin{cases} 1, & \text{falls } x \geq 1 \\ 0, & \text{sonst} \end{cases}$$

- Schritt 2 (für x): Berechnen von b_{-1}, b_{-2}, \dots :
Gehe zu Schritt 1 (für $x' = 2 \cdot (x - b_0)$)

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

⇒ $1.000\overline{11}$, periodisch, nicht endlich

191

192

Binärdarstellungen von 1.1 und 0.1

- sind nicht endlich, also gibt es Fehler bei der Konversion in ein (endliches) binäres Fließkommazahlensystem.
- 1.1f und 0.1f sind nicht 1.1 und 0.1, sondern geringfügig fehlerhafte Approximationen dieser Zahlen.

$$\begin{aligned} 1.1 &= \underline{1.100000000000000000}888178\dots \\ 1.1f &= \underline{1.1000000}238418\dots \end{aligned}$$

193

Rechnen mit Fließkommazahlen

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

$$\begin{array}{r} 1.111 \cdot 2^{-2} \\ + 1.011 \cdot 2^{-1} \\ \hline = 1.001 \cdot 2^0 \end{array}$$

1. Exponenten anpassen durch Denormalisieren einer Zahl 2. Binäre Addition der Signifikanden 3. Renormalisierung 4. Runden auf p signifikante Stellen, falls nötig

194

Der IEEE Standard 754

Legt Fließkommazahlensysteme und deren Rundungsverhalten fest

- Single precision (`float`) Zahlen:

$$F^*(2, 24, -126, 127) \quad \text{plus } 0, \infty, \dots$$

- Double precision (`double`) Zahlen:

$$F^*(2, 53, -1022, 1023) \quad \text{plus } 0, \infty, \dots$$

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

195

32-bit Darstellung einer Fließkommazahl



± Exponent

Mantisse

$$\pm 2^{-126}, \dots, 2^{127}$$

$$0, \infty, \dots$$

$$1.000000000000000000000000$$

...

$$1.111111111111111111111111$$

196

Der IEEE Standard 754

Warum

$$F^*(2, 24, -126, 127)?$$

- 1 Bit für das Vorzeichen
- 23 Bit für den Signifikanden (führendes Bit ist 1 und wird nicht gespeichert)
- 8 Bit für den Exponenten (256 mögliche Werte)(254 mögliche Exponenten, 2 Spezialwerte: 0, ∞ ,...)

⇒ insgesamt 32 Bit.

197

Der IEEE Standard 754

Warum

$$F^*(2, 53, -1022, 1023)?$$

- 1 Bit für das Vorzeichen
- 52 Bit für den Signifikanden (führendes Bit ist 1 und wird nicht gespeichert)
- 11 Bit für den Exponenten (2048 mögliche Exponenten, 2 Spezialwerte: 0, ∞ ,...)

⇒ insgesamt 64 Bit.

198

8. Kontrollanweisungen

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

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 ;
```

199

200

Anweisungsarten

Gültige Anweisungen sind:

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

201

Anweisungsarten

Beispiele

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

202

Blöcke

Ein Block ist ...

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

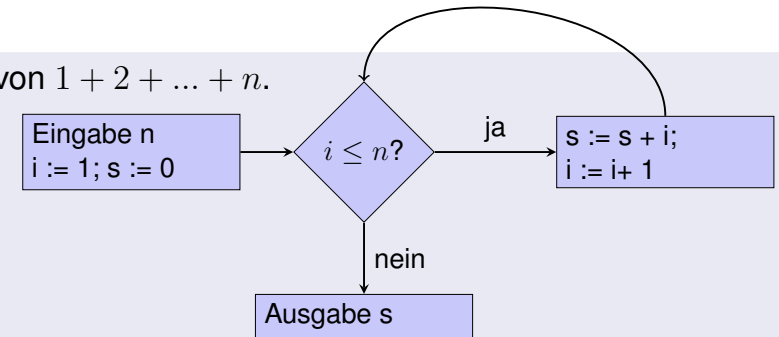
```
{  
  statement1  
  statement2  
  :  
}
```

203

Kontrollfluss

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

Berechnung von $1 + 2 + \dots + n$.



204

Auswahlweisungen

realisieren Verzweigungen

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

`if`-Anweisung

```
if ( condition )  
  statement
```

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

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

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

205

206

if-else-Anweisung

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

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

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

- *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"); ← Einrückung
} else {
    Out.println("odd"); ← Einrückung
}
```

207

208

Iterationsanweisungen

realisieren „Schleifen“:

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

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

```
// 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);
```

209

210

for-Anweisung: Syntax

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

- *init-statement*: Ausdrucksanweisung, Deklarationsanweisung, Nullanweisung
- *condition*: Ausdruck vom Typ `boolean`
- *expression*: beliebiger Ausdruck
- *statement*: beliebige Anweisung (*Rumpf* der for-Anweisung)

211

for-Anweisung: Semantik

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

- *init-statement* wird ausgeführt
- *condition* wird ausgewertet
 - `true`: Iteration beginnt
statement wird ausgeführt
expression wird ausgeführt
 - `falsch`: for-Anweisung wird beendet.

212

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.

213

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);
```

214

Beispiel: Harmonische Zahlen

Ergebnisse:

- Compute H_n for $n = ?$ 10000000
Forward sum = 15.4037
Backward sum = 16.686
- Compute H_n for $n = ?$ 100000000
Forward sum = 15.4037
Backward sum = 18.8079

215

Beispiel: Harmonische Zahlen

Beobachtung:

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

Erklärung:

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

216

Beispiel: Primzahltest

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

Eine Schleife, die das testet:

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

217

Primzahltest: Terminierung

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

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

218

Primzahltest: Korrektheit

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

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

219

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;
```

220

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

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

221

Beispiel: Die Collatz-Folge

$(n \in \mathbb{N})$

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

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

222

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 + " ");
}
```

223

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
```

224

Die Collatz-Folge

Erscheint die 1 für jedes n ?

- Man vermutet es, aber niemand kann es beweisen!
- Falls nicht, so ist die `while`-Anweisung zur Berechnung der Collatz-Folge für einige n theoretisch eine Endlosschleife!.

225

`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.

226

while-Anweisung: Semantik

```
while ( condition )  
    statement
```

- *condition* wird ausgewertet
 - **true**: Iteration beginnt
statement wird ausgeführt
 - **false**: *while*-Anweisung wird beendet.
- 

227

while Anweisung

```
while ( condition )  
    statement
```

- *statement*: beliebige Anweisung, Rumpf der *while* Anweisung.
- *condition*: Ausdruck vom Typ `boolean`.

228

while Anweisung

```
while ( condition )  
    statement
```

ist äquivalent zu

```
for ( ; condition ; )  
    statement
```

229

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);
```

230

do Anweisung

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

- *statement*: beliebige Anweisung, Rumpf der do Anweisung.
- *expression*: Ausdruck vom Typ `boolean`.

231

do Anweisung

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

ist äquivalent zu

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

232

do-Anweisung: Semantik

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

- Iteration beginnt ←
 - *statement* wird ausgeführt.
- *expression* wird ausgewertet
 - `true`: Iteration beginnt
 - `false`: do-Anweisung wird beendet.

233

Blöcke

- Beispiel: Rumpf der main Funktion

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

- Beispiel: Schleifenrumpf

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

234

Sichtbarkeit

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

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

235

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: undeklariertes Name
  }
}
```

236

Gültigkeitsbereich einer Deklaration

Gültigkeitsbereich: Ab Deklaration bis Ende des Programmteils, der die Deklaration enthält.

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; ... }
```

237

Automatische Speicherdauer

Lokale Variablen (Deklaration in Block)

- werden bei jedem Erreichen ihrer Deklaration neu „angelegt“, d.h.
 - Speicher / Adresse wird zugewiesen
 - evtl. Initialisierung wird ausgeführt
- werden am Ende ihrer deklarativen Region „abgebaut“ (Speicher wird freigegeben, Adresse wird ungültig)

238

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*.

239

Zusammenfassung

- Auswahl (bedingte *Verzweigungen*)
 - if und if-else-Anweisung
- Iteration (bedingte *Sprünge*)
 - for-Anweisung
 - while-Anweisung
 - do-Anweisung
- Blöcke und Gültigkeit von Deklarationen

240

Äquivalenz von Iterationsanweisungen

Wir haben gesehen:

- `while` und `do` können mit Hilfe von `for` simuliert werden

Es gilt aber sogar:

- Alle drei Iterationsanweisungen haben die gleiche „Ausdrucksstärke“ (Skript).

241

Die „richtige“ Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

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

Ziele sind oft nicht gleichzeitig erreichbar.

242

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

Erster (korrekter) Versuch:

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

243

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

Weniger Anweisungen, *weniger* Zeilen:

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

244

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

Weniger Anweisungen, *einfacherer* Kontrollfluss:

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

Das ist hier die "richtige" Iterationsanweisung!

245

Die switch-Anweisung

switch (*condition*)
statement

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

```
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.");
}
```

246

Semantik der switch-Anweisung

```
switch (condition)
  statement
```

- `condition` wird ausgewertet.
- Beinhaltet `statement` eine `case`-Marke mit (konstantem) Wert von `condition`, wird dorthin gesprungen.
- Sonst wird, sofern vorhanden, an die `default`-Marke gesprungen. Wenn nicht vorhanden, wird `statement` übersprungen.
- Die `break`-Anweisung beendet die `switch`-Anweisung.

247

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!");
}
```

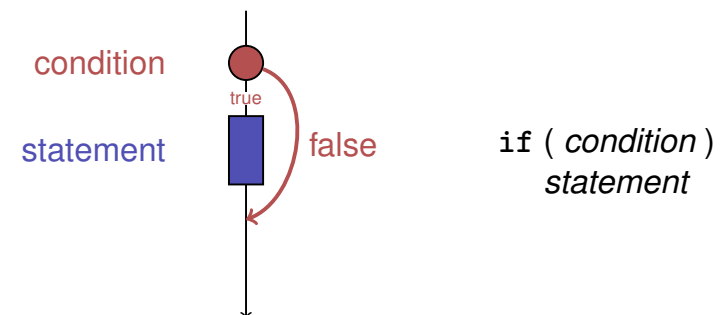
248

Definition: **Kontrollfluss**

Reihenfolge der (wiederholten) Ausführung von Anweisungen

Kontrollfluss

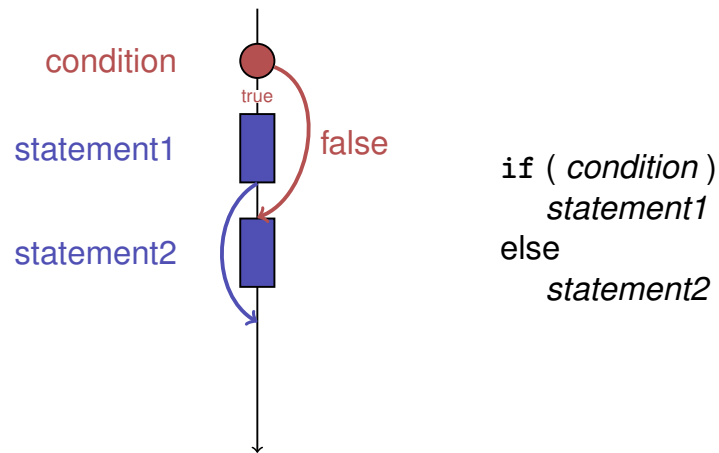
- Grundsätzlich von oben nach unten. . .
- . . . ausser in Auswahl- und Kontrollanweisungen



249

250

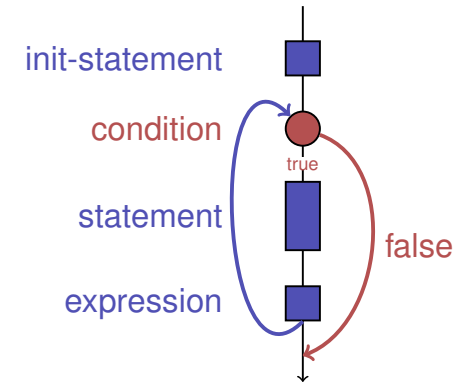
Kontrollfluss if else



251

Kontrollfluss for

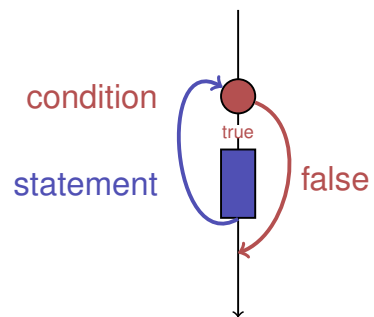
`for (init statement condition ; expression)
statement`



252

Kontrollfluss while

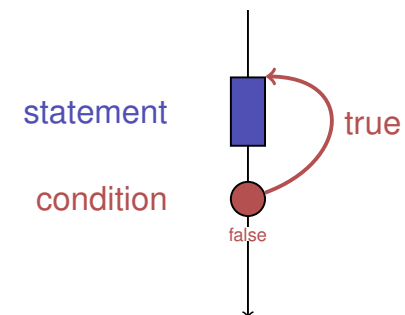
`while (condition)
statement`



253

Kontrollfluss do while

`do
statement
while (condition)`



254

Kontrollfluss switch

