252-0027
Einführung in die Programmierung

2.0 Einfache Java Programme

Thomas R. Gross

Department Informatik
ETH Zürich

Übersicht

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

  Vorwärts/Rückwärts schliessen

  - 2.3.1 Pre- und Postconditions
  - 2.3.2 Hoare-Tripel für Anweisungen
- **2.4 Verzweigungen**

2.3.2 Hoare Tripel (oder 3-Tupel)

Hoare Tripel (oder 3-Tupel)

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

```
{P} S {Q}
```

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

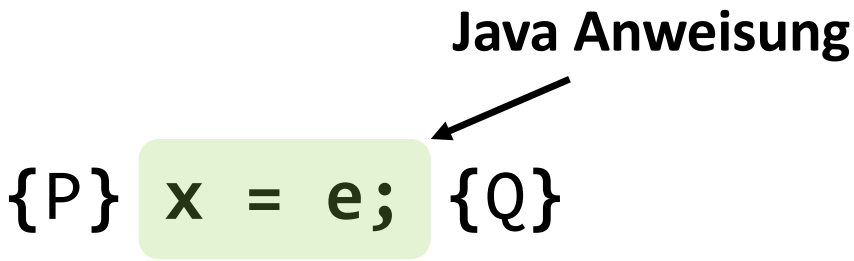
Hoare Tripel (oder 3-Tupel)

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

- **Für jedes Java Statement gibt es genaue Regeln die eine Precondition und eine Postcondition in Beziehung setzen**
  - Regel für Zuweisungen
  - Regel für zwei aufeinander folgende Anweisungen
  - Regel für «if»-Statements
  - [später:] Regel für Schleifen

2.3.2.1 Eine Zuweisung

Zuweisungen



- **Bilden wir Q’ in dem wir in Q die Variable x durch e ersetzen**
- **Das Tripel ist gültig wenn:**

  Für alle Zustände des Programms ist Q’ wahr wenn P wahr ist

  - D.h., aus P folgt Q’, geschrieben P ⇒ Q’

Review ⇒

A slightly more involved combination of two statements S and T is *implication*, where in mathematics one usually writes

$$S \implies T.$$

It stands for *"If S is true, then T is true"*. One also says *"S implies T"*. The statement $S \implies T$ is false if S is true and T is false, and in all other three cases it is true. In other words, the first three statements below are true while the last one is false.

- 4 is an even number $\implies$ 71 is a prime number.
- 5 is an even number $\implies$ 71 is a prime number.
- 5 is an even number $\implies$ 70 is a prime number.
- 4 is an even number $\implies$ 70 is a prime number.

We point out that $S \implies T$ does *not* express any kind of *causality* like *"because S is true, T is also true"*.

U. Maurer: Diskrete Mathematik (Script) ETH Zürich, 2020

We introduce a new, logical operator, *implication*, denoted as $A \to B$ and defined by the function table

A	B	$A \to B$
0	0	1
0	1	1
1	0	0
1	1	1

A → B can be understood as an alternative notation for ¬ A ∨ B, which has the same function table.

2.2.2 Two Meanings of the Symbol $\Longrightarrow$

Within a proof as described, the symbol $\Longrightarrow$ stands for the application of a simple proof step. It is important to note that the symbol $\Longrightarrow$ used in such a derivation has a different (though related) meaning compared to the discussion of the symbol $\Longrightarrow$ in Section 2.1.2. If $S \Longrightarrow T$ is written in the described sense as a proof step within a proof, then, assuming of course that the proof step is correct, the statement $S \Longrightarrow T$ is true. But, conversely, if $S \Longrightarrow T$ is true for some statements S and T, there may not exist a proof step demonstrating this. We hence emphasize again that the two uses of $\Longrightarrow$ are different.[4]

[4] It would be justified to use two different symbols for the two meanings

Beispiel

```
{z > 34}
  y = z+1;
{y > 1}
```

Q’ ist {z+1 > 1}

Gilt P ⇒ Q’ ?

Also (z>34) ⇒ (z+1) > 1 **?**

✓

Beispiel

```
{z ≠ 1}
  y = z*z;
{y ≠ z}
```

Q’ ist {z*z ≠ z}

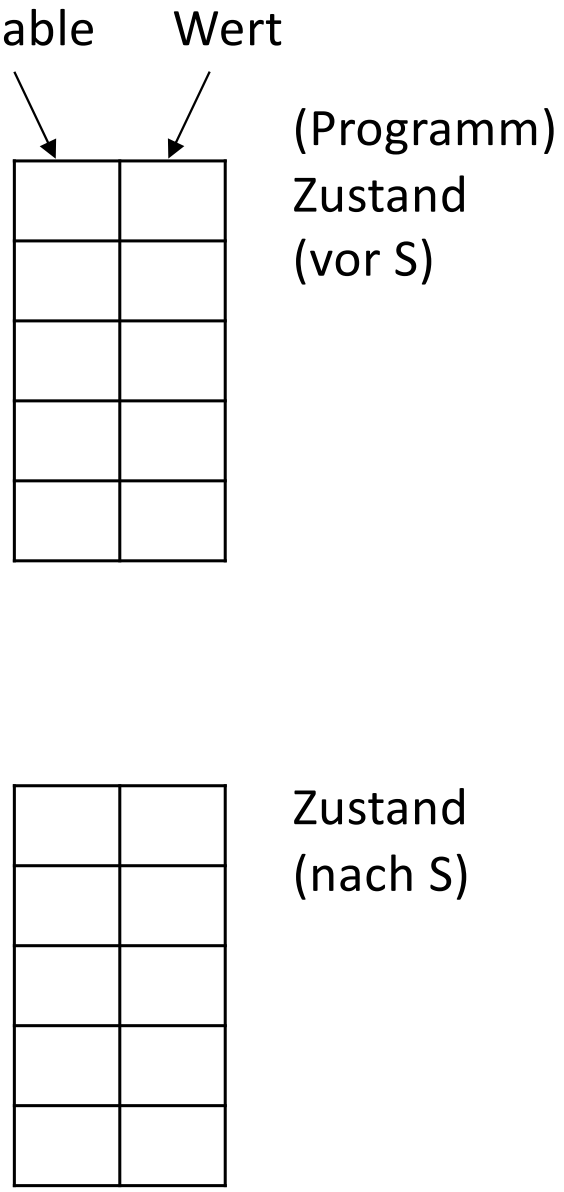
Gilt P ⇒ Q’ ?

Also (z ≠ 1) ⇒ (z*z) ≠ z **?**

✗ (z==0)

Hoare Tripel: {P} S {Q}

Aussagen über <u>alle</u> Programmaus-führungen



Hoare Tripel: {P} S {Q}

```
{??}
x = y + 1
{x > 5}
```

Zustand (vor S)

x	0
y	8

Zustand (nach S)

x	9
y	8

Hoare Tripel: {P} S {Q}

{y > 6}

x = y + 1

{x > 5}

Tripel ist gültig da aus y>6 folgt (y+1)>5 (das ist Q')

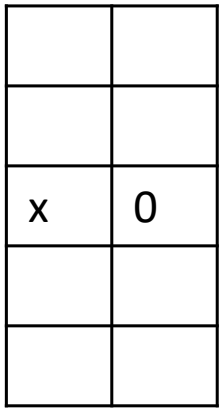
Zustand (vor S)

x	0
y	8

Zustand (nach S)

x	9
y	8

Hoare Tripel: {P} S {Q}

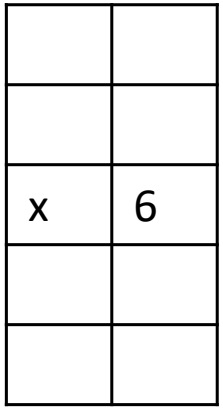


Zustand
(vor S)

{P}

x = 6

{Q}



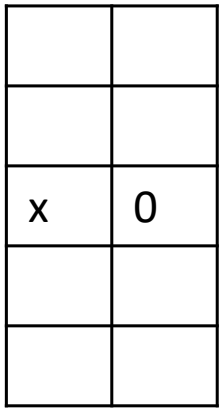
Zustand
(nach S)

Hoare Tripel: {P} S {Q}

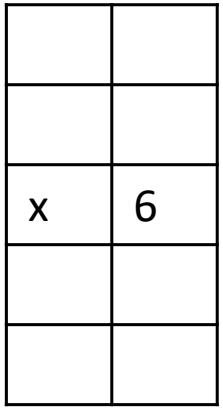
Aussagen über alle Programmaus-führungen

Wie drücken wir aus dass Q immer gilt?

```
{true}
  x = 6
{x > 5}
```



Zustand (vor S)



Zustand (nach S)

Beispiele

Poll

Gültig oder ungültig?

- Wir nehmen an: alles `int` Variablen, ohne Overflow/Underflow

1. `{x ≠ 0} y = x*x; {y > 0}`
2. `{z ≠ 0} y = z*z; {y ≠ z}`
3. `{x >= 0} y = 2*x; {y > x}`

Beispiele

Poll

Gültig oder ungültig?

- Wir nehmen an: alles `int` Variablen, ohne Overflow/Underflow

1. `{x ≠ 0} y = x*x; {y > 0}` gültig
2. `{z ≠ 0} y = z*z; {y ≠ z}` ungültig
3. `{x >= 0} y = 2*x; {y > x}` ungültig

Welche dieser Aussagen trifft zu?

- {x ≠ 0} y = x*x; {y > 0} ist ein gueltiges Hoare Tripel (96% | 220)
- {z ≠ 0} y = z*z; {y ≠ z} ist ein gueltiges Hoare Tripel (6% | 15)
- {x >= 0} y = 2*x; {y > x} ist ein gueltiges Hoare Tripel (11% | 26)

2.3.2.2 Folgen von Anweisungen

- **Wir können auch Hoare Tripel für eine Folge von Anweisungen definieren**

```
{ Precondition }
Statement 1
Statement 2
{ Postcondition }
```

- **Auch hier ist Vorwärts- und Rückwärts-Schliessen möglich**

Folgen von Anweisungen

- **Einfachste Folge: zwei Statements**

$$\{P\}\ \mathbf{S1;S2}\ \{Q\}$$

- **Tripel ist gültig wenn (und nur wenn) es eine Aussage** `R` **gibt so dass**
  1. `{P} S1 {R}` ist gültig, *und*
  2. `{R} S2 {Q}` ist gültig.

Beispiel

- **Alle Variable sind `int`, kein Overflow/Underflow**

```
{z >= 1}
  y = z+1;
{y > 1}
  w = y*y;
{w > y}
```

Sei `R` die Aussage `{y > 1}`

1. **Wir zeigen dass `{z >= 1}` `y=z+1;` `{y > 1}` gültig ist.**

Regel für Zuweisungen: `z >= 1` impliziert `z+1 > 1`

2. **Wir zeigen dass `{y > 1}` `w=y*y;` `{w > y}` gültig ist.**

Regel für Zuweisungen: `y > 1` impliziert `y*y > y`

Beispiele

Poll

Gültig oder ungültig?

- Wir nehmen an alles `int` Variablen, ohne Overflow/Underflow

1. `{true}`
   `x = y;`
   `z = x;`
   `{y == z}`
2. `{x == 7 ∧ y == 5}`
   `tmp = x;`
   `x = tmp;`
   `y = x;`
   `{y == 7 ∧ x == 5}`

Beispiele

Poll

Gültig oder ungültig?

- Wir nehmen an alles `int` Variablen, ohne Overflow/Underflow

1. `{true}`

```
  x = y;
  z = x;
```

`{y == z}` gültig

2. `{x == 7 ∧ y == 5}`

```
  tmp = x;
  x = tmp;
  y = x;
```

`{y == 7 ∧ x == 5}` ungültig

Welche(s) Tripel sind (ist) gültig?

- // A {true} if (x > 7) { y = 4; } else { y = 3; } {y < 5} (96% | 214)
- // B {x > 10} if (x%4 == 0) { y = x/4; } {y ≥ 2} (21% | 48)

252-0027
Einführung in die Programmierung

2.4 Verzweigungen

Thomas R. Gross

Department Informatik
ETH Zürich

Übersicht

- **2.4 Verzweigungen**
  - 2.4.1 «if»-Anweisungen
    Vergleichsoperatoren
  - 2.4.2 Typ boolean
  - 2.4.3 Bedingte («short-circuit») Auswertung
  - 2.4.4 Pre- und Postconditions
  - 2.4.5 «Schwächste» Vorbedingung

2.4 Verzweigungen («if»-Anweisungen)

- **Wollen nur manche Anweisungen (Statements) ausführen**
- **Eine Anweisung die die Ausführung der anderen kontrolliert:**
  - if-Anweisung
  - Verschiedene Varianten

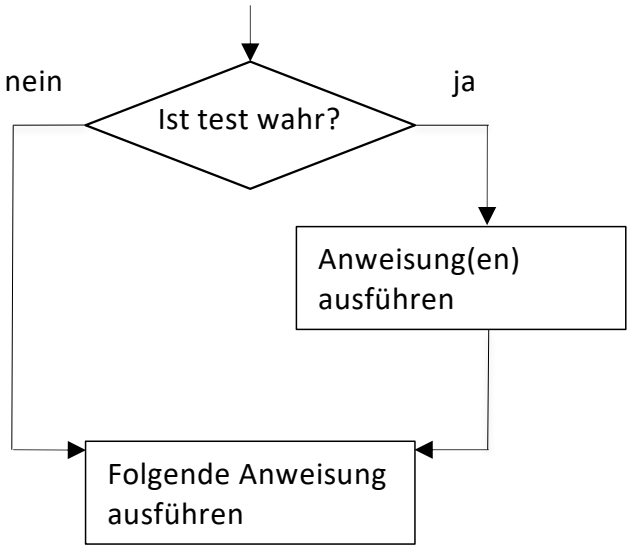
- **Manchmal spricht man auch von «bedingter Ausführung»**
  - Eine Bedingung muss erfüllt sein damit eine Anweisung ausgeführt wird

2.4.1 «if»-Anweisungen

«if»-Anweisung («if-Statement»)

- **Führt eine Anweisung (oder Anweisungen) nur aus wenn** test **den Wert wahr («true») ergibt.**

```
if (test) {
    statement(s);
    ...
}
// folgende Anweisung
```



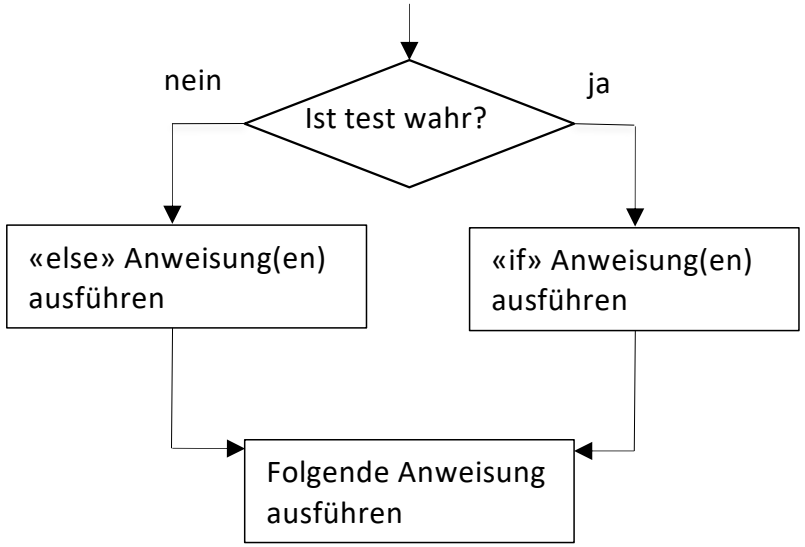
- **Beispiel:**

```
double punkte = console.nextDouble();
if (punkte >= 50.0) {
  System.out.println("Pruefung bestanden.");
}
```

«if-else»-Anweisung

- **Führt eine Gruppe von Anweisungen aus wenn** test **den Wert wahr («true») ergibt, sonst eine andere Gruppe**

```
if (test) {
    statement(s);
} else {
    statement(s);
}
// folgende Anweisung
```



- **Beispiel:**

```
double punkte = console.nextDouble();
if (punkte >= 50.0) {
   System.out.println("Pruefung bestanden.");
} else {
   System.out.println("Pruefung nicht bestanden.");
}
```

Boolesche Ausdrücke

- **Was für Tests können wir in einem if-Statement (oder if-else—Statement) verwenden?**

- **Fall 1: Variable und Werte eines Basistyps**
  - Bisher kennen wir nur `int`, `long` und `double`
  - Was wir vorstellen gilt für alle Basistypen

Boolesche Ausdrücke

- **if-Anweisungen und if-else-Anweisungen verwenden beide boolesche Ausdrücke**

```
if (i > 0) { ... }
if (i > 10) { ... } else { ... }
```

  - Oft Vergleiche oder Kombinationen von Vergleichen
  - Diese Ausdrücke werden ausgewertet --- Ergebnis entweder «true» oder «false»
  - `true` und `false` sind Konstanten (für Wahrheitswerte)
- **Boolesche Ausdrücke verwenden Vergleichs*operatoren***

Vergleichsoperatoren

Operator	Bedeutung	Beispiel	Wert
`==`	gleich	`1 + 1 == 2`	`true`
`!=`	ungleich	`3 != 2`	`true`
`<`	weniger als	`10 < 5`	`false`
`>`	grösser als	`10 > 5`	`true`
`<=`	weniger als oder gleich	`126 <= 100`	`false`
`>=`	grösser als oder gleich	`5 >= 5`	`true`

- **Vorsicht: nicht alle Vergleichsoperatoren können für alle Typen (sinnvoll) angewendet werden**
- **Vergleichsoperatoren haben tiefen Rang**
  - `1+1==2` soll `(1+1)==2` ergeben: `1+false` macht keinen Sinn

Vergleichsoperatoren

- **Vergleichsoperatoren haben eine tiefere Präzedenz als arithmetische Operatoren.**

```
5 * 7 >= 8 + 4 * (7 - 1)
5 * 7 >= 8 + 4 * 6
35    >= 8 + 24
35    >= 32
true
```

- **Vergleichsoperatoren können nicht eine «Kette» bilden wie in Mathematik**

```
2 <= x <= 10
true   <= 10
error!
```

(Annahme: x ist 15)

Boolesche Ausdrücke im if-Statement

- **Der boolesche Ausdruck steht im if-Statement in Klammern**
  - Aussagen sind boolesche Ausdrücke
- **Aussagen können mit && (UND) oder || (ODER) kombiniert werden**

```
2 <= x && x <= 10
True   && False
False
```

(Annahme: x ist 15)

- **Verwenden Sie Klammern um Klarheit zu schaffen**

```
(2 <= x) && (x <= 10)
```

Boolesche Operatoren

- **Ausdrücke mit Vergleichsoperatoren können durch *boolesche Operatoren* verknüpft werden**

Operator	Bedeutung	Beispiel	Wert
&&	and	`(2 == 3) && (-1 < 5)`	`false`
||	or	`(2 == 3) || (-1 < 5)`	`true`
!	not	`!(2 == 3)`	`true`

- **«Wahrheitstabelle» für diese Operatoren, für Aussagen *p* und *q*:**

p	q	p && q	p || q
`true`	`true`	`true`	`true`
`true`	`false`	`false`	`true`
`false`	`true`	`false`	`true`
`false`	`false`	`false`	`false`

p	!p
`true`	`false`
`false`	`true`

Boolesche Ausdrücke

Poll

- **Was ist das Ergebnis für die folgenden Ausdrücke?**

```
int x = 42;
int y = 17;
int z = 25;
```

1. `y < x && y <= z`
2. `x % 2 == y % 2 || x % 2 == z % 2`
3. `x <= y + z && x >= y + z`
4. `!(x < y && x < z)`
5. `(x + y) % 2 == 0 || !((z - y) % 2 == 0)`

Boolesche Ausdrücke

Poll

- **Was ist das Ergebnis für die folgenden Ausdrücke?**

```
int x = 42;
int y = 17;
int z = 25;
```

1. `y < x && y <= z` → `true`
2. `x % 2 == y % 2 || x % 2 == z % 2` → `false`
3. `x <= y + z && x >= y + z` → `true`
4. `!(x < y && x < z)` → `true`
5. `(x + y) % 2 == 0 || !((z - y) % 2 == 0)` → `false`

= und == in Java

- **= ist der Zuweisungsoperator (assignment operator)**

```
int k = 4;  // k hat nach diesem Statement den Wert 4
```

  - Ergebnis ist der Wert 4 -- kein Wahrheitswert

```
if (k = 4) { .... }
```

```
HW.java:10: error: incompatible types
if (k = 4) {
      ^
  required: boolean
  found:    int
```

  - Entsprechende Fehlermeldung
  - Aber manchmal ist die Fehlermeldung verwirrend

= und == in Java

- **= ist der Zuweisungsoperator («assignment operator»)**

```
int k = 4;   // k hat nach diesem Statement den Wert 4
```

  - Ergebnis ist der Wert 4 -- kein Wahrheitswert

```
if (k = 4) { .... }
```

```
HW.java:10: error: incompatible types
if (k = 4) {
      ^
  required: boolean
  found:    int
```

- **== ist der Vergleichsoperator (prüft Gleichheit)**
  - Kann in if-Statement verwendet werden

```
if (k == 4) {
```

Beispiel

- **Welcher boolesche Ausdruck ergibt `true` wenn ein Jahr `jahr` ein Schaltjahr ist?**
  - `jahr` ist Schaltjahr wenn `jahr` durch 4 teilbar ist (ohne Rest), `jahr` aber nicht durch 100 ohne Rest teilbar ist, es sei denn dass `jahr` ohne Rest durch 400 teilbar sei.
  - `int jahr; // aktuelles Jahr`

- `jahr % 4 == 0 && jahr % 100 != 0 || jahr % 400 == 0`
- Besser mit Klammern:

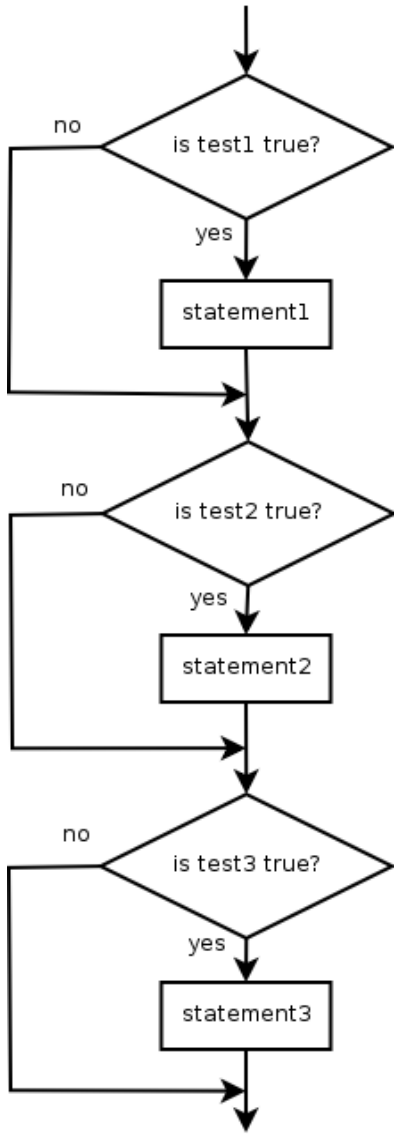
```
((jahr % 4 == 0) && (jahr % 100 != 0)) || (jahr % 400 == 0)
```

Gebrauch von `if`

Was fällt Ihnen in diesem Code Beispiel auf?

```
int percent =  ... // Prozent Punkte

if (percent >= 90) {
    System.out.println("Ihre Note ist 6.0.");
}
if (percent >= 80) {
    System.out.println("Ihre Note ist 5.0.");
}
if (percent >= 70) {
    System.out.println("Ihre Note ist 4.0.");
}
if (percent >= 60) {
    System.out.println("Ihre Note ist 3.5.");
}
if (percent < 60) {
    System.out.println("Ihre Note ist 3.0.");
}
...
```

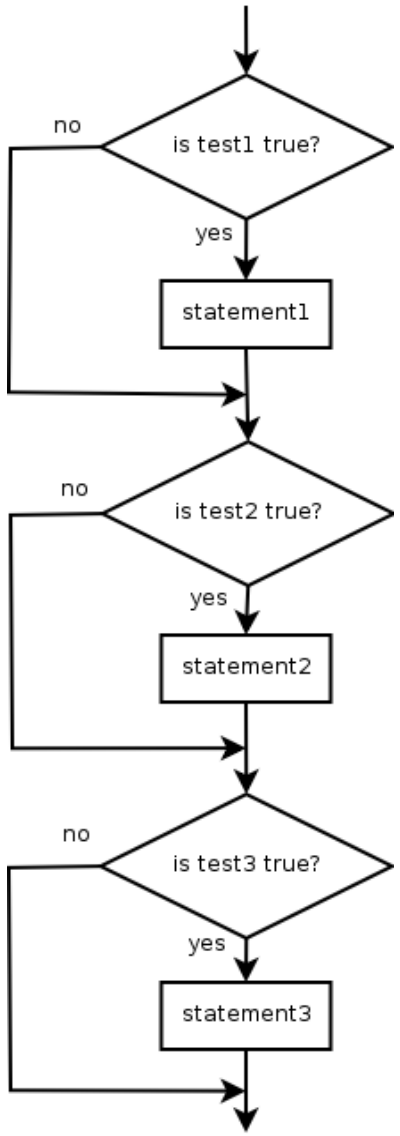


Gebrauch von `if`

Was fällt Ihnen in diesem Code Beispiel auf?

```
int percent =  ... // Prozent Punkte

if (percent >= 0 && percent <= 30) {
    System.out.println("Ihre Note ist 2.0.");
}
if (percent > 30 && percent <= 40) {
    System.out.println("Ihre Note ist 3.0.");
}
if (percent > 40 && percent <= 50) {
    System.out.println("Ihre Note ist 4.0.");
}
if (percent > 50 && percent <= 70) {
    System.out.println("Ihre Note ist 4.5.");
}
if (percent > 70 && percent <= 90) {
    System.out.println("Ihre Note ist 5.0.");
}
System.out.println("Ihre Note ist 6.0.");
```



Gebrauch von `if`

Code (?) um Prozentsatz (aufgerundet) zu visualisieren

```
if (percentage > 0.2 && percentage <= 0.3)
    return "●●●○○○○○○○";
```

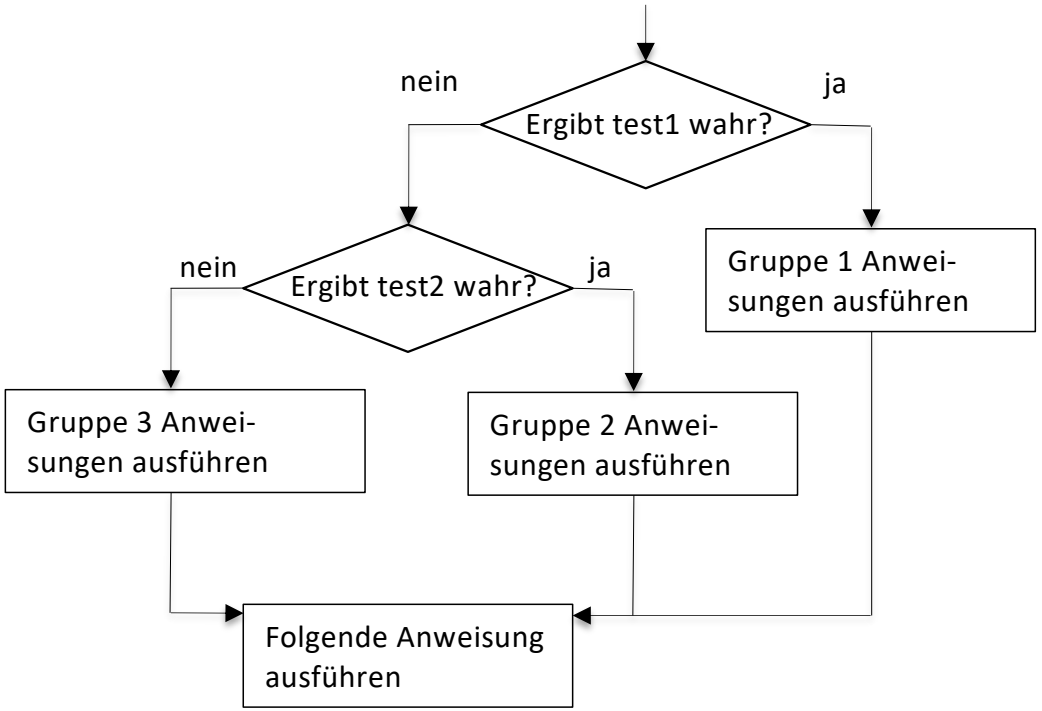
```
private static string GetPercentageRounds(double percentage)
{
    if (percentage == 0)
        return "○○○○○○○○○○";
    if (percentage > 0.0 && percentage <= 0.1)
        return "●○○○○○○○○○";
    if (percentage > 0.1 && percentage <= 0.2)
        return "●●○○○○○○○○";
    if (percentage > 0.2 && percentage <= 0.3)
        return "●●●○○○○○○○";
    if (percentage > 0.3 && percentage <= 0.4)
        return "●●●●○○○○○○";
    if (percentage > 0.4 && percentage <= 0.5)
        return "●●●●●○○○○○";
    if (percentage > 0.5 && percentage <= 0.6)
        return "●●●●●●○○○○";
    if (percentage > 0.6 && percentage <= 0.7)
        return "●●●●●●●○○○";
    if (percentage > 0.7 && percentage <= 0.8)
        return "●●●●●●●●○○";
    if (percentage > 0.8 && percentage <= 0.9)
        return "●●●●●●●●●○";

    return "●●●●●●●●●●";
}
```

Verschachtelte if-else-Anweisungen

Auswahl bestimmt durch mehrere Tests

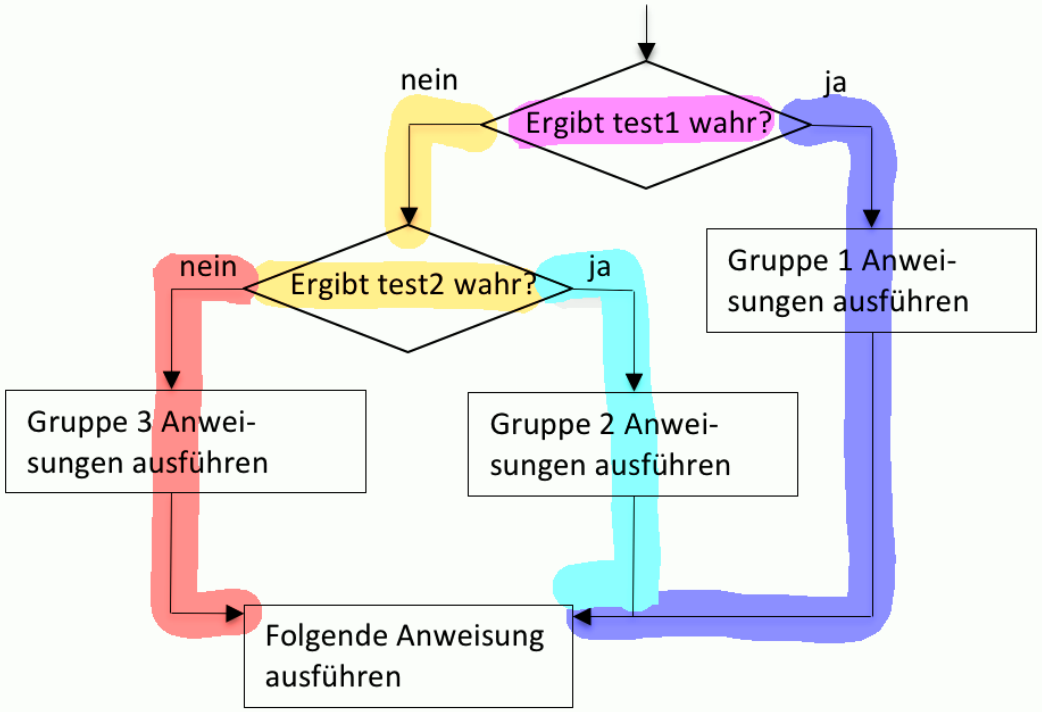
```
if (test1) {
    statement(s);
} else if (test2) {
    statement(s);
} else {
    statement(s);
}
// folgende Anweisung
```



Verschachtelte if-else-Anweisungen

Auswahl bestimmt durch mehrere Tests

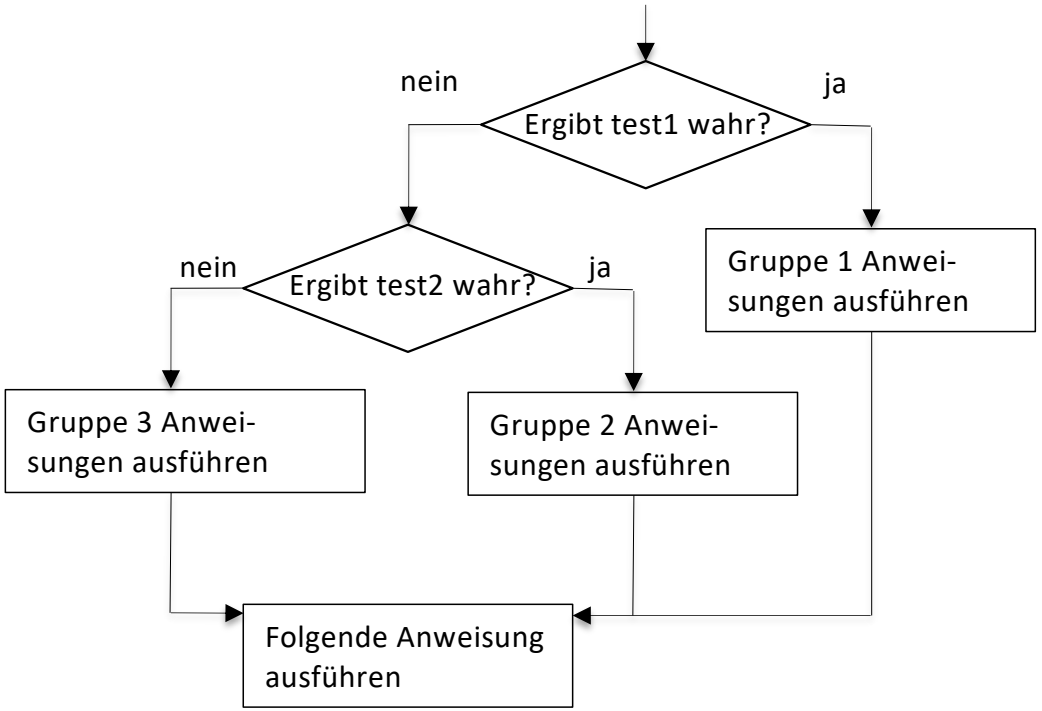
```
if (test1) {
    statement(s);
} else if (test2) {
    statement(s);
} else {
    statement(s);
}
// folgende Anweisung
```



Verschachtelte if-else-Anweisungen

Beispiel:

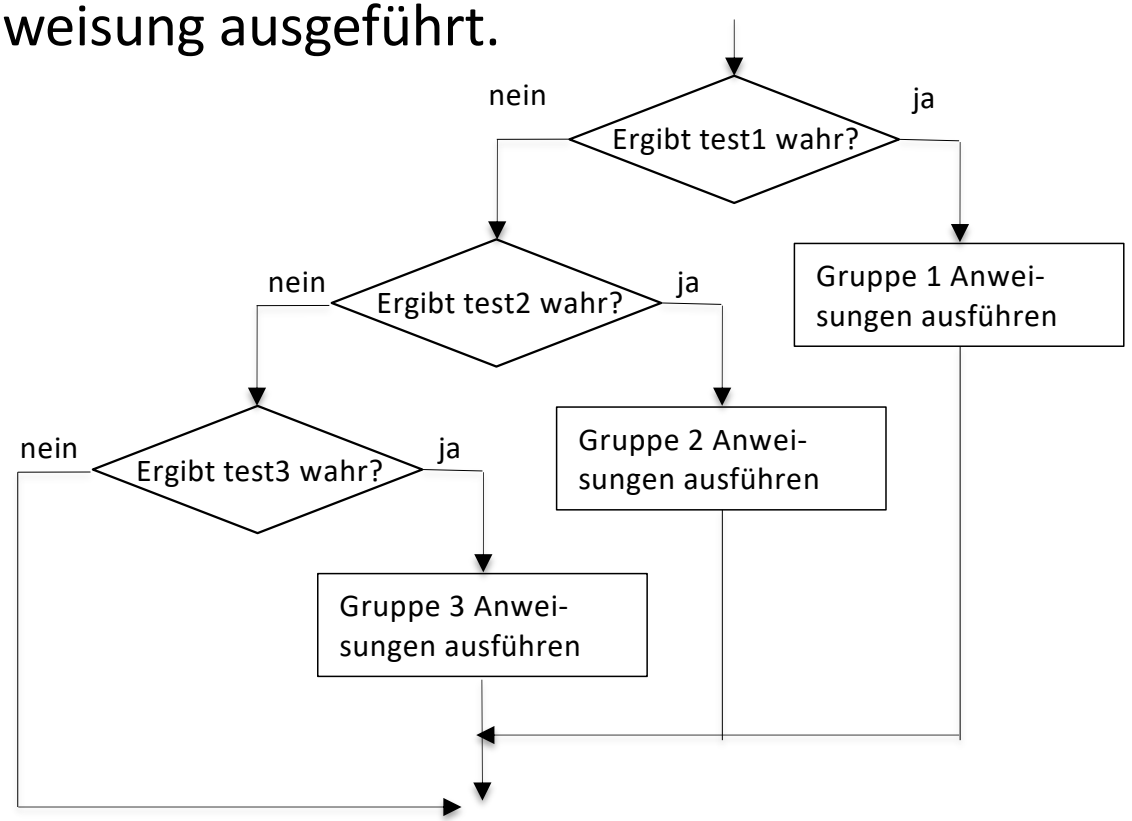
```
if (x > 0) {
  System.out.println("Positiv");
} else if (x < 0) {
  System.out.println("Negativ");
} else {
  System.out.println("Null");
}
```



Verschachtelte if-else-if

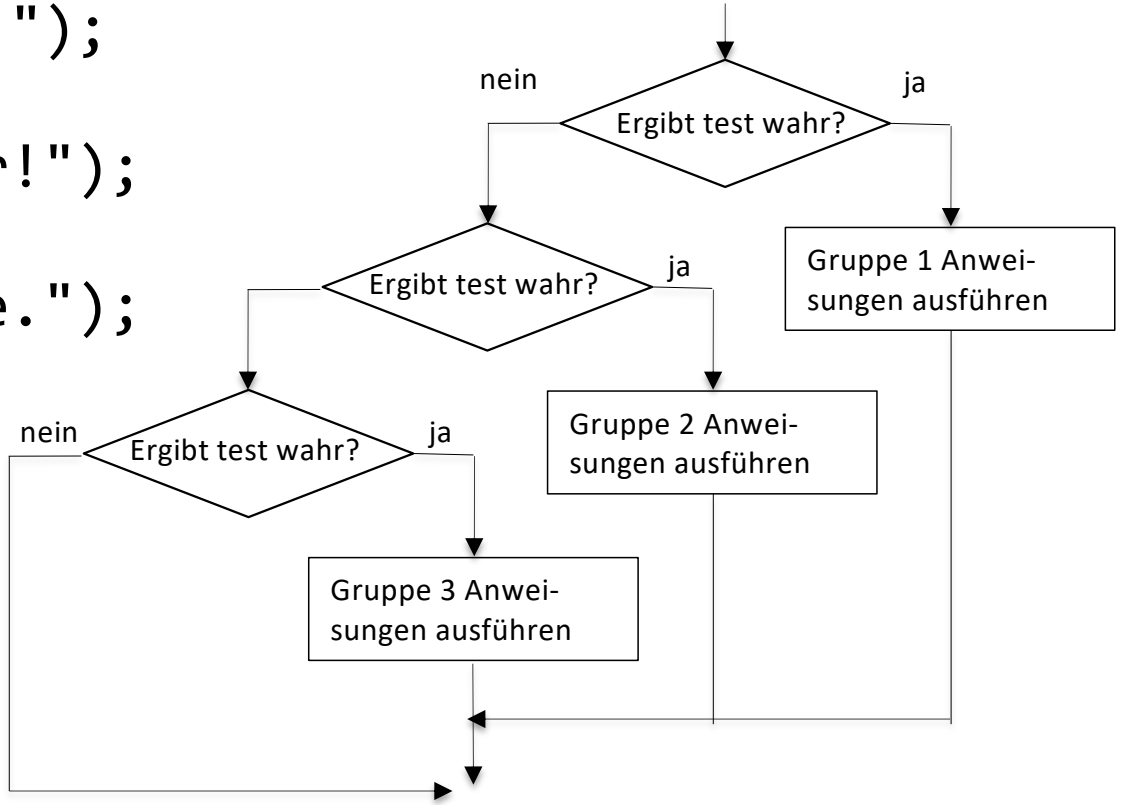
- Endet mit `else`: genau ein *Pfad* muss ausgeführt werden
- Endet mit `if`: Eventuell wird keine Anweisung ausgeführt.

```
if (test1) {
      statement(s);          //Gruppe 1
} else if (test2) {
      statement(s);          //Gruppe 2
} else if (test3) {
      statement(s);          //Gruppe 3
}
```



Verschachtelte if-else-if

```
if (place == 1) {
  System.out.println("Gold!!");
} else if (place == 2) {
  System.out.println("Silber!");
} else if (place == 3) {
  System.out.println("Bronze.");
}
```



Verschachtelte if-Konstrukte

- Genau ein 1 Pfad mit Anweisung(en) *(gegenseitiger Ausschluss)*

```
if (test) {
    statement(s);
} else if (test) {
    statement(s);
} else {
    statement(s);
}
```

verschachtelte if/else

- 0 oder 1 Pfad mit Anweisung(en) *(gegenseitiger Ausschluss)*

```
if (test) {
    statement(s);
} else if (test) {
    statement(s);
} else if (test) {
    statement(s);
}
```

verschachtelte if/else/if

- 0, 1, oder viele Pfade mit Anweisung(en) *(unabhängig, kein gegenseitiger Ausschluss)*

```
if (test) {
    statement(s);
}
if (test) {
    statement(s);
}
if (test) {
    statement(s);
}
```

aufeinanderfolgende if/if/if

Welche if/else-Kombination?

Poll

(1) verschachtelte if/else (2) verschachtelte if/else/if (3) Folge von if/if/if

1. Ob – abhängig von früheren Rennen – jemand in der ersten, zweiten, oder letzten Gruppe startet.

2. Ob es eine Medaille (Notendurschnitt ≥ 5.9) oder eine Urkunde (5.75 – 5.9) gibt.

3. Ob eine Zahl durch 2, 3, und/oder 5 teilbar ist.

4. Note (auf 0.25 gerundet) aufgrund der Punkte (Prozent) in der Prüfung.

Welche if/else-Kombination?

Poll

(1) verschachtelte if/else (2) verschachtelte if/else/if (3) Folge von if/if/if

- Ob – abhängig von früheren Rennen – jemand in der ersten, zweiten, oder letzten Gruppe startet.
  - **(1)** verschachtelte `if / else if / else`
- Ob es eine Medaille (Notendurschnitt ≥ 5.9) oder eine Urkunde (5.75 – 5.9) gibt.
  - **(2)** verschachtelte `if / else if`
- Ob eine Zahl durch 2, 3, und/oder 5 teilbar ist.
  - **(3)** Folge von `if / if / if`
- Note (auf 0.25 gerundet) aufgrund der Punkte (Prozent) in der Prüfung.
  - **(1)** verschachtelte `if / else if / else if / else if / else`

2.4.2 Typ `boolean`

- **Boolesche Werte können in Variablen des Typs `boolean` gespeichert werden.**
- **Der Typ `boolean` kennt nur zwei Werte: wahr (`true`) und falsch (`false`).**
  - Ein Vergleich («**test**») ist ein boolescher Ausdruck (ein Ausdruck der ein `boolean` Ergebnis hat).
  - Boolesche Ausdrücke können mit den booleschen Operatoren kombiniert werden.
  - `boolean quadrant1;` oder `boolean quadrant1 = true;` deklarieren boolesche Variable.

Gebrauch von `boolean`

- **Kann Ergebnis eines komplizierten Ausdrucks speichern und später wiederverwenden**
  - Vorausgesetzt die Komponenten ändern sich nicht …
- **Macht Programm lesbarer**

Beispiel – wer kann Antrag stellen?

kantonglarus

Departement Sicherheit und Justiz
Zivilstands- und Bürgerrechtsdienst

Wohnsitz

Bund:	10 Jahre, wovon 3 in den letzten 5 Jahren (die Zeit zwischen dem 8. und 18. Lebensjahr zählt doppelt; tatsächlicher Aufenthalt muss jedoch 6 Jahre betragen).
Kanton und Gemeinde:	5 Jahre im Kanton, wovon die letzten 3 Jahre ohne Unterbruch in der Gemeinde, in der das Bürgerrecht beantragt wird.

Für die Berechnung der 10 Jahre zählt der Aufenthalt mit einer

- C- oder B-Bewilligung (Aufenthalt) **ganz**,
- F-Bewilligung (vorläufig Aufgenommene) **halb**,
- L-Bewilligung (Kurzaufenthalt) oder N-Bewilligung (Asylsuchende) **nicht**

Beispiel (Keine Rechtsauskunft)

```
int jahreCH_CB;
int jahrCH_F;
int jahreCHletzte5;
int jahreKvor18;
int jahreKnach18;
int jahreGvor18;
int jahreGnach18;

int jahreCH = (jahreCH_F/2 +jahreCH_CB);
boolean bundJ10 = jahreCH >= 10;
boolean bund1 = bundJ10 && (jahreCHletzte5 >= 3);
boolean bund2 = (jahreCH>=6) && (jahreKvor18 +
                                 jahreGvor18)*2 >= 10;
boolean antragErlaubt = bund1 || bund2;
```

Hinweise

- **Manchmal sieht man solchen Code (test ob eine Variable den Wert `true` hat):**

```
boolean isPrime = … ;
if (isPrime == true) {    // schlecht
    ...
}
```

- **Das ist nicht nötig und redundant. Besser :**

```
if (isPrime) {            // gut
    ...
}
```

Hinweise

- **Auch nicht besser ist der Test für `false`:**

```
if (isPrime == false) {   // schlecht

if (!isPrime) {           // gut
```

2.4.3 Bedingte Auswertung

- **Für && und || müssen nicht immer beide Operanden ausgewertet werden, um das Ergebnis zu ermitteln**
- **Java *beendet* die Auswertung eines booleschen Ausdrucks sobald das Ergebnis fest steht.**
- **Dies nennen wir eine «short-circuit» Auswertung oder *bedingte Auswertung***
  - Folgende Teilausdrücke werden abhängig von zuerst ausgewerteten Ausdrücken (nicht) evaluiert
  - Regel der Programmiersprache

Bedingte («short-circuit») Auswertung

- **Für && und || müssen nicht immer beide Operanden ausgewertet werden, um das Ergebnis zu ermitteln**
- **Java *beendet* die Auswertung eines booleschen Ausdrucks sobald das Ergebnis fest steht.**
  - && stoppt sobald ein Teil(ausdruck) `false` ist
  - || stoppt sobald ein Teil(ausdruck) `true` ist

Boolesche Operatoren

- **Ausdrücke mit Vergleichsoperatoren können durch *boolesche Operatoren* verknüpft werden**
  - && und || sind links-assoziativ
  - Ausdrücke werden von links nach rechts, gemäss Präzedenz und Assoziativität ausgewertet
  - Klammern schaffen Klarheit

Auswertung eines Tests

- Gegeben Programm(segment) mit drei `int` Variablen a, b und x
- Wir wollen x zum Quotienten a/b setzen, aber nur wenn a/b grösser als 0 ist

```
int a;
int b;
int x;   // a, b werden irgendwie gesetzt

      // nur positive Werte sollten gespeichert werden
      //    integer division kann 0 fuer a,b !=0 ergeben
      // duerfen nicht durch 0 dividieren

      x = ...
```

Auswertung eines Tests

- Gegeben Programm(segment) mit drei `int` Variablen a, b und x
- Wir wollen x zum Quotienten a/b setzen, aber nur wenn a/b grösser als 0 ist

```
int a;
int b;
int x;   // a, b werden irgendwie gesetzt

      // nur positive Werte sollten gespeichert werden
      //    integer division kann 0 fuer a,b !=0 ergeben
      if (b != 0) {

         x = ...
```

Auswertung eines Tests

- **Wollen Quotienten a/b nur speichern wenn grösser als 0**

```
int a;
int b;
int x;   // a, b werden irgendwie gesetzt

      // nur positive Werte sollten gespeichert werden
      //    integer division kann 0 fuer a,b !=0 ergeben
      if (b != 0) {
         if (a/b > 0) {
            x = a/b;
         }
      }
```

Auswertung eines Tests

- **Viele if-Statements machen Programm unleserlich**

```
int a;
int b;
int x;   // a, b werden irgendwie gesetzt

      // nur positive Werte sollten gespeichert werden
      //    integer division kann 0 fuer a,b !=0 ergeben
      if (b != 0) {
         if (a/b > 0) {
            x = a/b;
         }
      }
```

Reihenfolge der Operanden ist wichtig

- **Dieser Code *führt* zu einer Fehlermeldung wenn b == 0:**

```
int a;
int b;
int x;   // a, b werden irgendwie gesetzt

      // nur positive Werte sollten gespeichert werden
      //    integer division kann 0 fuer a,b !=0 ergeben

      if ((a/b > 0) && (b != 0)) {
          x = a/b;
      }
```

Bedingte («short-circuit») Auswertung

- **Dieser Code führt zu *keiner* Fehlermeldung wenn b == 0:**

```
int a;
int b;
int x;   // a, b werden irgendwie gesetzt

      // nur positive Werte sollten gespeichert werden
      //    integer division kann 0 fuer a,b !=0 ergeben

      if ((b != 0) && (a/b > 0)) {
          x = a/b;
      }
```

De Morgan's Regeln

Regeln für die Negation boolescher Ausdrücke.

- Praktisch wenn man das Gegenteil eines Ausdrucks braucht.

Ursprünglicher Ausdruck	Negierter Ausdruck	Alternative
`a && b`	`!a || !b`	`!(a && b)`
`a || b`	`!a && !b`	`!(a || b)`

- Beispiel:

Original	Negiert

`if (x == 7 && y > 3) {`
`  …`
`}`
	`if (x != 7 || y <= 3) {`
	`  …`
	`}`

2.4.4 Pre/Postconditions für if-Anweisungen

If-Statement Muster

```
// ursprüngliche Annahmen
if (test) {
  // wissen das test true war
  if-Block

} else {
  // wissen das test false war
  else-Block

}
// können if-Block oder else-Block ausgeführt haben
```

Grundidee(n)

1. **Die Precondition für den if-Block und den else-Blocks (eines if-Statements) beinhaltet das Ergebnis des Tests.**

2. **Die Postcondition *nach* dem if-Statement ist die Disjunktion («oder») der Postconditions des if- und else-Blockes.**

If-Statement Muster

```
// ursprüngliche Annahmen
if (test) {
  // wissen das test true war
  if-Block
  // Q1
} else {
  // wissen das test false war
  else-Block
  // Q2
}
// Q1 ∨ Q2 (können if-Block oder else-Block ausgeführt haben)
```

If-Statements

$$\{P\}\ \texttt{if (b) S1 else S2}\ \{Q\}$$

- **Tripel ist gültig wenn (und nur wenn) es Aussagen $Q1$, $Q2$ gibt so dass**
  1. $\{P \wedge b\}$ S1 $\{Q1\}$ ist gültig *und*
  2. $\{P \wedge !b\}$ S2 $\{Q2\}$ ist gültig *und*
  3. Nach dem if-Statement gilt Q, d.h.
     a) Aus Q1 folgt Q
     b) Aus Q2 folgt Q

If-Statements

```
{P} if (b) S1 else S2 {Q}
```

- **Tripel ist gültig wenn (und nur wenn) es Aussagen** `Q1, Q2` **gibt so dass**
  1. `{P ∧ b} S1 {Q1}` ist gültig *und*
  2. `{P ∧ !b} S2 {Q2}` ist gültig *und*
  3. Nach dem if-Statement gilt Q, d.h.

     (aus Q1 folgt Q) und (aus Q2 folgt Q)

If-Statements

$$\{P\}\ \texttt{if (b) S1 else S2}\ \{Q\}$$

- **Tripel ist gültig wenn (und nur wenn) es Aussagen $Q1$, $Q2$ gibt so dass**
  1. $\{P \wedge b\}\ S1\ \{Q1\}$ ist gültig *und*
  2. $\{P \wedge !b\}\ S2\ \{Q2\}$ ist gültig *und*
  3. Nach dem if-Statement gilt Q, d.h.

     $(Q1 \Rightarrow Q) \wedge (Q2 \Rightarrow Q)$

If-Statements

$$\{P\}\ \texttt{if (b) S1 else S2}\ \{Q\}$$

- **Tripel ist gültig wenn (und nur wenn) es Aussagen** $Q1$, $Q2$ **gibt so dass**
  1. $\{P \wedge b\}\ S1\ \{Q1\}$ ist gültig *und*
  2. $\{P \wedge !b\}\ S2\ \{Q2\}$ ist gültig *und*
  3. $(Q1 \vee Q2) \Rightarrow Q$

If-Statements

$$\{P\}\ \texttt{if (b) S1 else S2}\ \{Q\}$$

- **Tripel ist gültig wenn (und nur wenn) es Aussagen $Q1$, $Q2$ gibt so dass**
  1. $\{P \wedge b\}\ S1\ \{Q1\}$ ist gültig *und*
  2. $\{P \wedge !b\}\ S2\ \{Q2\}$ ist gültig *und*
  3. Nach dem if-Statement gilt Q, d.h.
     a) Aus Q1 folgt Q
     b) Aus Q2 folgt Q

Beispiel

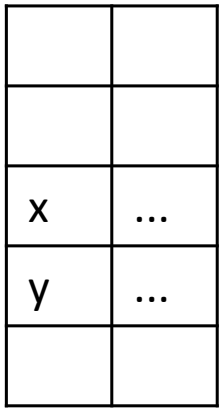
Alle Variable sind `int`, kein Overflow/Underflow

```
{true}
if (x > 7) { y = x; }
else { y = 20; }
{y > 5}
```

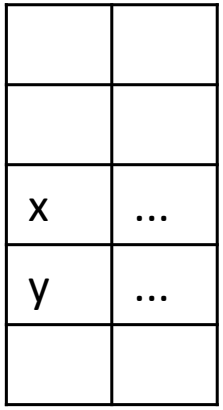
- Sei Q1 `{y > 7}` (andere Aussagen gehen evtl. auch)
- Sei Q2 `{y == 20}` (andere Aussagen gehen evtl. auch)

Hoare Tripel: {P} S {Q}

```
{true}
if (x > 7) { y = x; }
else { y = 20; }
{y > 5}
```



Zustand (vor S)



Zustand (nach S)

Beispiel

Alle Variable sind `int`, kein Overflow/Underflow

```
{true}
if (x > 7) { y = x; }
else { y = 20; }
{y > 5}
```

- Sei Q1 `{y > 7}` (andere Aussagen gehen evtl. auch)
- Sei Q2 `{y == 20}` (andere Aussagen gehen evtl. auch)

- **Mit der Regel für Zuweisungen können wir zeigen**

```
{true ∧ x > 7}
y = x;
{y > 7}
```

- **Mit der Regel für Zuweisungen**

```
{true ∧ x <= 7}
y = 20;
{y == 20}
```

- **Dann zeige dass**

```
(y>7) ∨ (y==20) ⇒ y>5
```

gültig

Beispiel

Alle Variable sind `int`, kein Overflow/Underflow

```
{true}
if (x > 7) { y = 4; }
else { y = 20; }
{y > 5}
```

- Sei Q1 `{y > 3}` (andere Aussagen gehen evtl. auch)
- Sei Q2 `{y == 20}` (andere Aussagen gehen evtl. auch)

- Mit der Regel für Zuweisungen können wir zeigen

```
{true ∧ x > 7}
y = 4;
{y > 3}
```

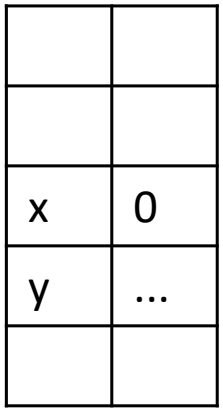
- Mit der Regel für Zuweisungen

```
{true ∧ x <= 7}
y = 20;
{y == 20}
```

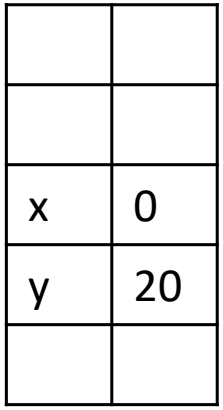
- **Dann zeige dass**
  a) Aus Q1 folgt Q
  b) Aus Q2 folgt Q

Hoare Tripel: {P} S {Q}

```
{true}
if (x > 7) { y = 4; }
else { y = 20; }
{y > 5}
```



Zustand (vor S)



Zustand (nach S)

Hoare Tripel: {P} S {Q}

```
{true}
if (x > 7) { y = 4; }
else { y = 20; }
{y > 5}
```

Zustand (vor S)

x	8
y	22

Zustand (nach S)

x	8
y	**4**

Beispiel

Alle Variable sind `int`, kein Overflow/Underflow

```
{true}
if (x > 7) { y = 4; }
else { y = 20; }
{y > 5}
```

- Sei Q1 `{y > 3}` (andere Aussagen gehen evtl. auch)
- Sei Q2 `{y == 20}` (andere Aussagen gehen evtl. auch)

- Mit der Regel für Zuweisungen können wir zeigen

```
{true ∧ x > 7}
y = 4;
{y > 3}
```

- Mit der Regel für Zuweisungen

```
{true ∧ x <= 7}
y = 20;
{y == 20}
```

- **Dann zeige dass**
  a) $y>3 \Rightarrow y>5$
  b) $y==20 \Rightarrow y>5$

nicht gültig

Beispiel

Alle Variable sind `int`, kein Overflow/Underflow

```
{true}
if (x > 7) { y = 4; }
else { y = 20; }
{y > 5}
```

- Sei Q1 `{y > 3}`
- Sei Q2 `{y == 20}` (andere Aussagen gehen evtl. auch)

- **Mit der Regel für Zuweisungen können wir zeigen**

```
{true ∧ x > 7}
y = 4;
{y > 3}
```

- **Mit der Regel für Zuweisungen**

```
{true ∧ x <= 7}
y = 20;
{y == 20}
```

- **Dann zeige dass**

$y>3 \lor y==20 \not\Rightarrow y>5$

Vorsicht

A	B	$A \rightarrow B$
0	0	1
0	1	1
1	0	0
1	1	1

- **Zur Party darf wer mindestens 18 oder ETH Student/in ist**
- **Informell:**

`(age>=18) ∨ (ETHstudent)`

- **Hoare Logik**

`y>3 ∨ y==20 ⇒ y>5`

- **Wenn gültig: Es kann nicht sein dass `y>3 ∨ y==20` wahr aber `y>5` falsch ist**
- **Nicht: Wenn `y>3 ∨ y==20` wahr ist dann ist `y>5` wahr**

«Darf zur Party»

We point out that $S \Longrightarrow T$ does *not* express any kind of *causality* like "*because* S is true, T is also true".

Beispiel

Alle Variable sind `int`, kein Overflow/Underflow

```
{x > 0}
if (a < 10) {y = 2*x;}
else {y = a*x;}
{y > 0}
```

- Sei Q1 {y ≥ 2} (andere Aussagen gehen evtl. auch)
- Sei Q2 {y ≥ 10} (andere Aussagen gehen evtl. auch)

- **Mit der Regel für Zuweisungen können wir zeigen**

```
{x>0 ∧ a<10}
y = 2*x;
{y ≥ 2}
```

- **Mit der Regel für Zuweisungen**

```
{x>0 ∧ a≥10}
y = a*x;
{y ≥ 10}
```

- **Dann zeige dass**

```
y≥2 ∨ y≥10 ⇒ y>0
```

Beispiele

Poll

Gültig oder ungültig?

- Wir nehmen an alles `int` Variablen, ohne Overflow/Underflow

(A)

```
{true}
if (x > 7) {
   y = 4;
} else {
   y = 3;
}
{y < 5}
```

(B)

```
{x > 10}
if (x%4 == 0) {
  y = x/4;
}
{y ≥ 2}
```

Beispiele

Poll

Gültig oder ungültig?

- Wir nehmen an alles `int` Variablen, ohne Overflow/Underflow

(A)

```
{true}
if (x > 7) {
   y = 4;
} else {
   y = 3;
}
{y < 5}
```

(B)

```
{x > 10}
if (x%4 == 0) {
  y = x/4;
} else {
  ;
}
{y ≥ 2}
```

Beispiele

Poll

Gültig oder ungültig?

- Wir nehmen an alles `int` Variablen, ohne Overflow/Underflow

```
{true}
if (x > 7) {
   y = 4;
} else {
   y = 3;
}
{y < 5}
```

gültig

```
{x > 10}
if (x%4 == 0) {
  y = x/4;
}
{y ≥ 2}
```

ungültig

```
{x > 10}
if (x%4 == 0) {
  y = x/4;
} else {
  ;
}
{y ≥ 2}
```

- Sei Q1 {y ≥ 2} (andere Aussagen gehen evtl. auch)
- Für S2 können wir keine genaueren Aussagen über den Zustand von y machen

Poll

- **Da {x > 10} ist x mindestens 12 und somit y ≥ 2**
- **Aber wir wissen nicht welchen Wert y vorher hatte und so kann man nicht zeigen dass y≥2**
- **Hätten wir als Precondition {x > 10 ∧ y > 10} gehabt (z.B.) so könnte Q2 {y > 10} sein und das Tripel wäre gültig.**

Welche(s) Tripel ist (sind) gültig?

- 1.) {true} x = y; z = x; {y == z} (98% | 270)
- 2). {x == 7 ∧ y == 5} tmp = x; x = tmp; y = x; {y == 7 ∧ x == 5} (1% | 5)