

Kontrollstrukturen

Softwareentwicklung I (IB)

Prof. Dr. Oliver Braun

Letzte Änderung: 28.01.2020 17:32

Algorithmen

- **Einfache Anweisungen** können nicht in kleinere Anweisungen zerlegt werde
- Beispiele: Variablendefinitionen, Wertzuweisungen, Ausgabeanweisungen
- **Zusammengesetzte Anweisungen** enthalten als Bausteine vollständige, untergeordnete Anweisungen
- Zusammengesetzte Anweisungen = **Kontrollstrukturen**
- Einfachste Kontrollstruktur: **Sequenz** = Anweisungsfolge = Aneinanderreihung von Anweisungen

- Beschreibungsformen für Algorithmen:

Umgangssprache Problematisch: Mißverständnisse,
Interpretationsmöglichkeiten, Sprachkenntnisse

Quelltext Nur mit Kenntnis einer konkreten Programmiersprache
lesbar

Neutrale, abstrakte Form Brauchbarer Kompromiss

- **Struktogramme** (= „Nassi-Shneiderman-Diagramme“)
- Halbgraphische Darstellungen, Einzelheiten fehlen
- Ziel: Reduktion auf die Idee, die wesentlichen Strukturen
- **Aktivitätsdiagramme** aus der UML (Unified Modeling Language)
 - Achtung: wirre Konstruktionen möglich

Elementarbausteine von Aktivitätsdiagrammen

- Umgangssprachlich:
Definiere n als ganze Zahl
Gib n den Wert 4
Zähle n um 1 hoch
Gib n aus

- Pseudocode: Kurze Codefragmente, ähnlich zu üblichen Programmiersprachen

```
int n  
n = 4  
n = n + 1  
print n
```



Alternativen (**if**)

- **if-Anweisung** (= „Alternative“, „bedingte Anweisung“, „Verzweigung“) besteht aus
 1. Bedingung (engl. condition) und
 2. untergeordneter Anweisung
- Untergeordnete Anweisung wird nur dann ausgeführt, wenn die Bedingung zutrifft, andernfalls übergangen
- Syntax

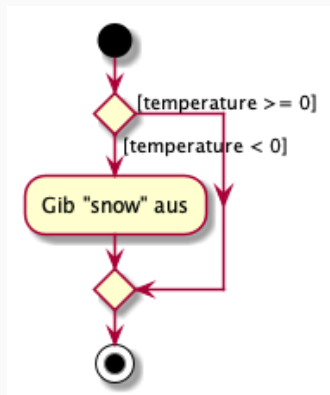
```
if (condition)  
    statement
```

Beispiel

- Text „snow“ ausgeben, wenn `temperature` einen negativen Wert hat
- andernfalls nichts ausgeben.

```
if (temperature < 0)
    System.out.println("snow");
```

- Zuerst die Bedingung `temperature < 0` prüfen ...
 - trifft zu? Ausgabeanweisung ausführen
 - trifft nicht zu? Ausgabeanweisung übergehen
- **Achtung:** UML Activity Diagrams spiegeln nicht direkt die Syntax der bedingten Anweisung wider.



- Bedingung = Ausdruck mit ja/nein-Ergebnis = **boolean**-Ausdruck
- Neue Art von Ausdruck: „trifft zu“ oder „trifft nicht zu“, keine dritte Möglichkeit
- Ergebnis der „Berechnung“ ist keine Zahl, sondern ein **Wahrheitswert**

- Bedingung einer **if**-Anweisung: **Vergleich** von zwei (numerischen) Ausdrücken
- Vergleich mit **relationalem Operator** (= „Vergleichsoperator“)
- Relationale Operatoren:
 - $<$ kleiner
 - \leq kleiner oder gleich
 - $>$ größer
 - \geq größer oder gleich
 - $=$ gleich
 - \neq nicht gleich

Vergleichsoperatoren (2)

- Häufiger Fehler:
 - Gleichheitsrelation „`==`“ in Java entspricht „`=`“ in der Mathematik
 - Wertzuweisung „`=`“ in Java hat keine mathematische Entsprechung
- Neue Gruppe von Operatoren:
 - Operanden sind Zahlen, Ergebnis ist Wahrheitswert
 - Dagegen arithmetische Operatoren: Operanden sind Zahlen, Ergebnis ist Zahl
- Syntax:

`expression relop expression`

- **Priorität** niedriger als arithmetische Operatoren. Beispiel:

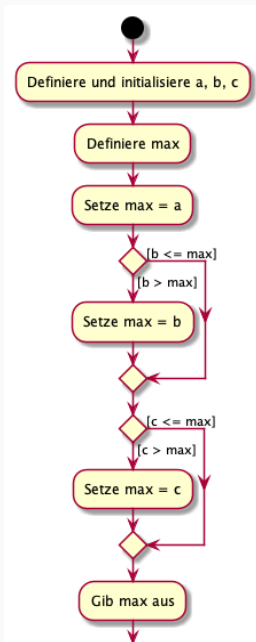
`2 + 3 < 2 * 3`

`5 < 2 * 3`

`5 < 6`

- Variablen **a**, **b** und **c** haben beliebige Werte. Der größte der drei soll in die Variable **max** kopiert und **max** dann ausgegeben werden.
- Algorithmus umgangssprachlich:
 - Annahme: **a** ist größter Wert, (vorläufig) an **max** zuweisen
 - Ist **b** größer als **max**? Dann **max** durch **b** ersetzen
 - Ist **c** noch größer als **max**? Dann **max** durch **c** ersetzen
 - **max** ausgeben
- Aufgabe: Erstellen Sie das entsprechende Aktivitätsdiagramm.

Beispiel: Größter von drei Werten: Aktivitätsdiagramm



```
class Max3 {  
    public static void main(String[] args) {  
        // Definiere und initialisiere a, b, c  
        final int a = Integer.parseInt(args[0]);  
        final int b = Integer.parseInt(args[1]);  
        final int c = Integer.parseInt(args[2]);  
  
        int max = a; // Definiere max; Setze max = a  
  
        if (b > max) // Ist b > max?  
            max = b; // Ja - Setze max = b  
        if (c > max) // Ist c > max?  
            max = c; // Ja - Setze max = c  
  
        System.out.println(max);  
    }  
}
```

Zweiseitige **if**-Anweisungen

- Zweiseitige **if**-Anweisungen: Erweiterung der einfachen, einseitigen **if**-Anweisung
- Enthält eine Bedingung & zwei untergeordnete Anweisungen
- Wenn die Bedingung zutrifft, wird die erste Anweisung ausgeführt, andernfalls die zweite
- Syntax:

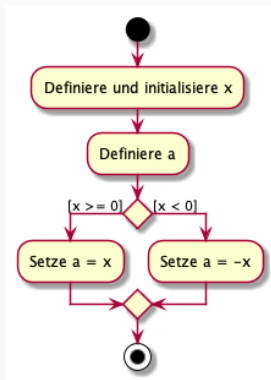
```
if (condition)
    then-statement
else
    else-statement
```

Zweiseitige **if**-Anweisungen (2)

- Beispiel: x enthält einen beliebigen Wert; in a soll dessen Absolutwert („Betrag“) berechnet werden:

```
double x = ...;
double a;
if (x  $\geq$  0)
    a = x;
else // x < 0
    a = -x;
```

- Es wird immer
 - *genau eine* der beiden Anweisungen ausgeführt, aber
 - *niemals beide*,
 - *niemals keine*.



Vergleich von Floatingpoint-Werten

- Relationale Operatoren sind polymorph, können ganze Zahlen und Floatingpoint-Werte vergleichen
- Gemischte Operanden: implizite Typkonversion
- Beispiel:

```
double a = 1.0/7.0;  
double b = a + 1.0;  
double c = b - 1.0;  
if (a == c)  
    System.out.println("gleich");  
else  
    System.out.println("verschieden");
```

- Was würden Sie erwarten?

- Ausgabe: „**verschieden**“, weil das Zwischenergebnis **b** eine zusätzliche gültige Stelle vor dem Komma braucht und damit am Ende eine Stelle „verliert“:

a = 0.14285714285714285

b = 1.1428571428571428

c = 0.1428571428571428

- Vergleich von exakten Floatingpoint-Werten mit „=“ und „≠“ heikel
- Floatingpoint-Werte in Bereichen prüfen, nicht auf Einzelwerte
- Im obigen Beispiel:

```
Math.abs(a - c) < 1e-10
```

statt

```
a == c
```

Ausgabe: `gleich`

Geschachtelte **if**-Anweisungen

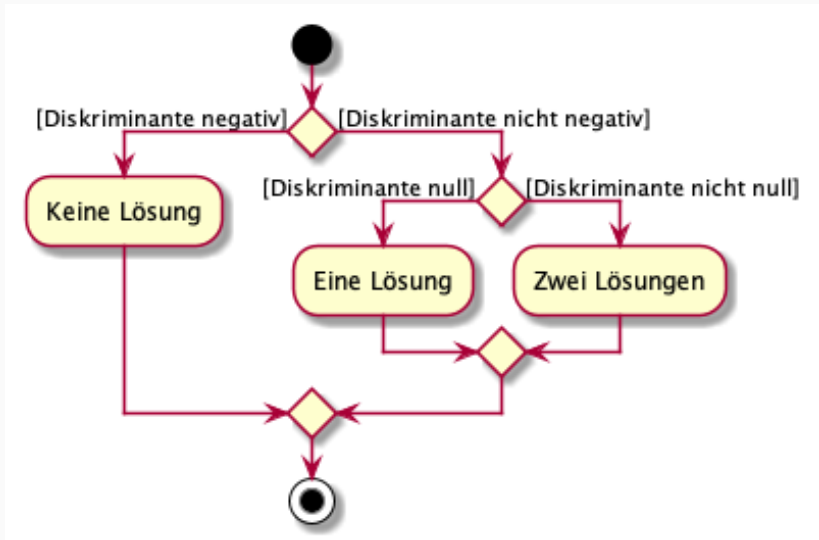
- **if**-Anweisung ist selbst eine Anweisung
- Eine **if**-Anweisung kann einer anderen untergeordnet werden:
„geschachtelte **if**-Anweisungen“
- Beispiel: Lösungen der quadratischen Gleichung $ax^2 + bx + c = 0$
abhängig vom Vorzeichen der Diskriminante $d = b^2 - 4ac$

$$d > 0: \quad \text{Zwei Lösungen } x_{1,2} = \frac{-b \pm \sqrt{d}}{2a}$$

$$d = 0: \quad \text{Eine Lösung } -\frac{b}{2a}$$

$$d < 0: \quad \text{Keine Lösung.}$$

Geschachtelte **if**-Anweisungen: Aktivitätsdiagramm



Geschachtelte `if`-Anweisungen: Programm

```
class SolveSqrt {  
    public static void main(String[] args) {  
        final double a = Double.parseDouble(args[0]);  
        final double b = Double.parseDouble(args[1]);  
        final double c = Double.parseDouble(args[2]);  
        final double d = b*b - 4*a*c;  
        if (d < 0)  
            System.out.printf("no solution%n");  
        else  
            if (d == 0)  
                System.out.printf("1 solution: %g%n",  
                    -b/(2*a));  
            else  
                System.out.printf("2 solutions: %g, %g%n",  
                    (-b + Math.sqrt(d))/(2*a),  
                    (-b - Math.sqrt(d))/(2*a));  
    }  
}
```

Mehrere `ifs` statt geschachtelte `ifs`

- Gleiches Ergebnis ohne geschachtelte `if`-Anweisungen:

...

```
if (d < 0)
```

...

```
if (d == 0)
```

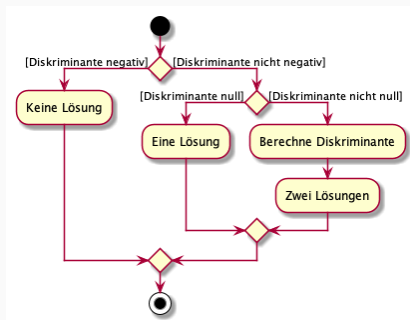
...

```
if (d > 0)
```

...

...

- **Aber:** Weniger effizient: Immer drei Vergleiche, statt einem oder zwei



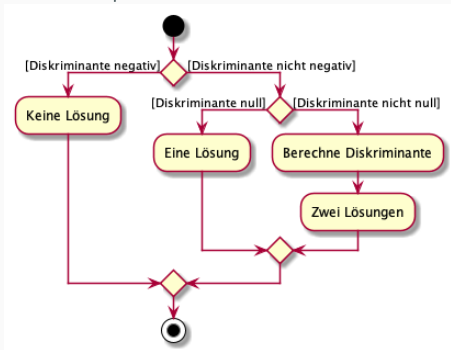
Blöcke als Anweisungsgruppe

- `if`-Anweisung kontrolliert eine oder zwei einzelne untergeordnete Anweisungen
- Oft gebraucht: mehrere untergeordnete Anweisungen
- Gruppieren einer Sequenz zu einem **Block** mit geschweiften Klammern
- Syntax

```
... {  
    statement  
    statement  
    statement  
    ...  
}
```


Blöcke als Anweisungsgruppe: Beispiel

Wurzel der Diskriminante im vorhergehenden Beispielprogramm nur einmal berechnen und zwischenspeichern:



```
...
else {
    a = 2*a;
    if (d == 0)
        System.out.printf(
            "1 solution: %g%n", -b/a);
    else {
        d = Math.sqrt(d);
        System.out.printf(
            "2 solutions: %g, %g%n",
            (-b + d)/a, (-b - d)/a);
    }
}
```

Leerer Block

- Block kann beliebig viele Anweisungen enthalten, einschließlich überhaupt keine: **leerer Block**: `{ }`
- Leerer Block äquivalent zur **leeren Anweisung**: `;`
- Äquivalent:

```
if ( ... )  
    ...  
else  
    { }
```

```
if ( ... )  
    ...  
else  
    ;
```

```
if ( ... )  
    ...  
if ( ... )  
    ...  
else  
    { { { ; ; ; } } }
```

- Zuordnungsproblem bei zwei `if` mit einem `else` (dangling else):

```
if (condition1)  
if (condition2) statement1  
else statement2
```

- `else` könnte jedem der beiden `ifs` zugeordnet werden.

Dangling Else: Mögliche Interpretationen

else → erstes if:

```
if (condition1)
    if (condition2)
        statement1
else
    statement2
```

else → zweites if:

```
if (condition1)
    if (condition2)
        statement1
    else
        statement2
```

- Compiler ignoriert Einrückung, orientiert sich am Programmtext: In beiden Fällen der Gleiche!
- **Regel:** `else` gehört zum textuell letzten freien `if` im selben Block („freies“ `if` = noch keinem `else` zugeordnet)
- Oben: Untere Interpretation auf der rechten Seite „gilt“

- Explizites Klammern schafft Klarheit, vermeidet Fehler

```
if (condition1) {  
    if (condition2)  
        statement1  
}
```

```
else  
    statement2
```

```
if (condition1) {  
    if (condition2)  
        statement1  
    else  
        statement2  
}
```

Verwenden Sie Klammern!

- verwenden Sie **immer** geschweifte Klammern!

```
if (d < 0) {  
    ...  
} else {  
    if (d == 0) {  
        ...  
    } else {  
        ...  
    }  
}
```

- das ist guter Stil und vermeidet manchen Fehler
- auf den Folien werden die Klammern um ein einzelnes statement aus Platzgründen oft weg gelassen

- Tiefe Verschachtelung von **if**-Anweisungen = **if-Kaskade**
- Beispiel: Vergleich eines Wertes mit einer Liste von Möglichkeiten
- Beispiel: Monatszahl (1 = Januar bis 12 = Dezember) in **month** gegeben, daraus in **days** Anzahl Tage berechnen

```
if (month == 1)
    days = 31;
else
    if (month == 2)
        days = 28;
    else
        if (month == 3)
            days = 31;
        ...
    else
```


if-Kaskade (2)

- Konsequente Einrückung aufgeben, um Einrückungstiefe zu begrenzen:

```
if (month == 1)
    days = 31;
else if (month == 2)
    days = 28;
else if (month == 3)
    days = 31;
...
else if (month == 12)
    days = 31;
```

- Bei regelmäßigen if-Kaskaden dieser Art besser `switch` verwenden

Wahrheitswerte (**boolean**)

Datentyp **boolean**

- Bedingung in einer **if**-Anweisung: Ausdruck mit „Wahrheitswert“ als Ergebnis
- Eigenständiger Typ **boolean**
- **boolean** hat nur zwei Werte für „wahr“ und „falsch“
- Allgemeine Konstanten = **Literale**
- **boolean**-Literale:

true = wahr, ja, zutreffend

false = falsch, nein, unzutreffend

- **boolean** kein numerischer Typ, nicht kompatibel zu **int** oder **double**
- Typecast kann nicht erzwungen werden, wird nicht übersetzt:

```
int i = (int) true; // error: incompatible types
```

- Relationale Operatoren erwarten numerische Operanden, liefern **boolean**-Ergebnis
- Auswertung nach demselben Schema wie arithmetische Ausdrücke gemäß Priorität und Assoziativität
- Relationale Operatoren „binden schwach“, kommen erst nach arithmetischen Operatoren zum Zug
- Beispiel:

`2 + 3 < 2 * 3`

`5 < 2 * 3`

`5 < 6`

`true`

- **Logische Operatoren** verknüpfen Wahrheitswerte
- Operanden sind Wahrheitswerte, Ergebnis ist Wahrheitswert
- Logische Operatoren:

Zeichen	Operanden	Bezeichnung	deutsch
&&	2	And	logisches Und
	2	Or	inklusives logisches Oder
^	2	Xor	exklusives logisches Oder
!	1	Not	logisches Nicht

„Wahrheitstabellen“ ordnen jeder Kombination ein Ergebnis zu, beschreiben Operatoren vollständig

• And

true	&&	true	→	true
true	&&	false	→	false
false	&&	true	→	false
false	&&	false	→	false

• Or

true		true	→	true
true		false	→	true
false		true	→	true
false		false	→	false

• Xor

true	^	true	→	false
true	^	false	→	true
false	^	true	→	true
false	^	false	→	false

- Andere Darstellung macht Unterschiede deutlich

Operanden	And	Or	Xor
true, true	true	true	false
true, false	false	true	true
false, true	false	true	true
false, false	false	false	false

Zusammengesetzte Bedingungen

- Logische Operatoren für zusammengesetzte Bedingungen
- Beispiel:
 - $-5 \leq x < 5$,
 - in Worten: x ist größer oder gleich -5 und x ist kleiner als 5
 - Als Java-Ausdruck: `(x >= -5) && (x < 5)`
- Beispiel: Berechnung der Tage eines Monats mit kürzerer `if`-Kaskade:

```
if (month == 2)
    days = 28;
else
    if ((month == 4) || (month == 6) ||
        (month == 9) || (month == 11))
        days = 30;
    else
        days = 31;
```


Zusammengesetzte Bedingungen (2)

- Priorität: binäre logische Operatoren ($\&\&$, $\|\|$, \wedge) binden schwächer als arithmetische und relationale Operatoren
- Not (!) bindet stärker als binäre Operatoren, wie alle unären Operatoren
- Beispiel: Die beiden folgenden Ausdrücke sind äquivalent:
 $x > 6 - 11 \ \&\& \ x + 1 < 2*3$
 $(x > (6 - 11)) \ \&\& \ ((x + 1) < (2*3))$

Mögliche Operatoren

- Allgemein: Mit zwei **boolean**-Operaden $2 \cdot 2 = 4$ mögliche Kombinationen:
 - beliebiger Operator \otimes

true \otimes **true** $\rightarrow ?$

true \otimes **false** $\rightarrow ?$

false \otimes **true** $\rightarrow ?$

false \otimes **false** $\rightarrow ?$

- Jedes ? kann **true** oder **false** sein $\Rightarrow 2 \cdot 2 \cdot 2 \cdot 2 = 2^4 = 16$ mögliche Wahrheitstabellen = mögliche binäre logische Operatoren
- And, Or, Xor sind drei der sechzehn möglichen Operatoren

- Operatoren fallen (bisher) in drei Gruppen:

Gruppe	Operatoren	Typen
Arithmetisch	+ - * / %	numerisch → numerisch
Relational	< > ≤ ⇒ = ≠	numerisch → boolean
Logisch	&& ^ !	boolean → boolean

- Zusätzlich: = und ≠ vergleichen auch **boolean**-Werte

```
if (x > 0 = y > 0)
    System.out.println("gleiches Vorzeichen");
```

Teilweise und vollständige Auswertung

- Einzelbedingungen in zusammengesetzten Bedingungen oft abhängig
- Beispiel: „Falls $b \neq 0$ und darüber hinaus $a/b > 0$...“:
`if (b \neq 0 && a/b > 0) ...`
- Für $b = 0$: Normale Auswertung von `&&` würde das Programm abbrechen wegen Division durch null
- Problem: **Vollständige Auswertung** der meisten binären Operatoren
- Lösung: **Teilweise Auswertung** von And
- Allgemein: Auswertung wird beendet, wenn das Ergebnis bereits nach der Berechnung des ersten Operanden feststeht Der verbleibende, zweite Operand wird nicht mehr berechnet!
- Teilweise Auswertung bei `&&` und `||`, aber nicht möglich bei `^`
- `&&`, `||` und der bedingte Operator (`?:`) werten teilweise aus, alle anderen vollständig

- Variablen mit Typ `boolean` zulässig. Beispiel:

```
boolean isOk;  
isOk = true;
```

Ebenso mit Initialisierung:

```
boolean isOk = true;
```

- Zuweisung von logischen Ausdrücken an `boolean`-Variablen:

```
boolean ice = temperature < 0;  
boolean steam = temperature > 100;  
boolean water = !ice && !steam;
```

- boolean-Variablen ohne Vergleich in Bedingungen:

```
if (water)
    System.out.println("Water");
else if (ice)
    System.out.println("Ice");
else if (steam)
    System.out.println("Steam");
else
    System.out.println("This cannot happen!");
```

Schleifen (**while**)

- **Schleife** = allgemeine Kontrollstruktur zum Wiederholen von Anweisungen
- Einfache Art von Schleifen: **while-Schleife**
- Syntax

```
while (condition) // Schleifenkopf  
    statement      // Schleifenrumpf
```


while-Schleife (Aktivitätsdiagramm)

```
while (condition)
```

```
statement
```

- `boolean`-Ausdruck `condition` steuert Ablauf:
 1. `condition` auswerten
 2. Falls `true`:
 - `statement` ausführen
 - zurück zu 1.sonst: Schleife beendet

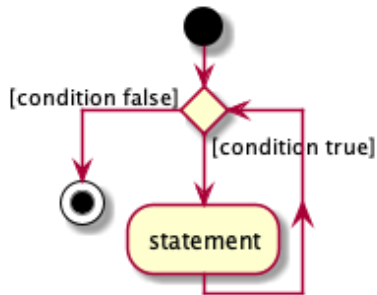


Abbildung: While-Schleife

Beispiel: Größter gemeinsamer Teiler

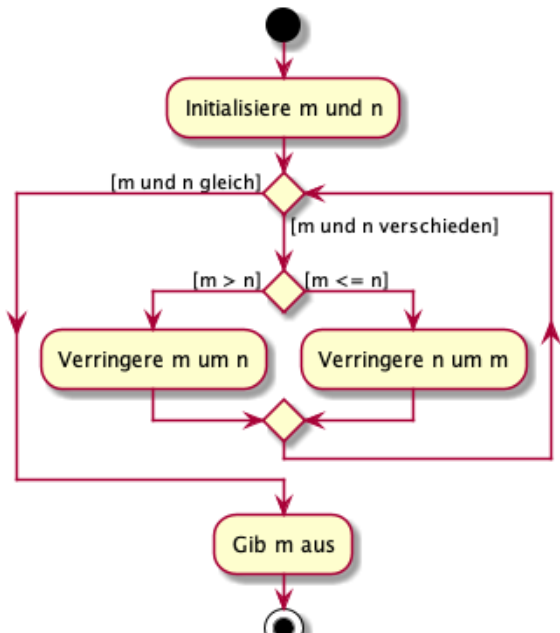
- Berechnung des **größten gemeinsamen Teilers** (engl. greatest common divider = GCD) von zwei natürlichen Zahlen m und n
- Verschiedene Algorithmen:
 - Differenzalgorithmus
 - Euklids Algorithmus
- Effizienz: Differenzalgorithmus langsamer als Euklids Algorithmus
- Beispiel: $m = 1000$, $n = 1$:

Differenzalgorithmus: 1000 Schleifendurchgänge

Euklids Algorithmus: 0 Schleifendurchgänge

- Umgangssprachlich:
 1. Initialisiere m und n
 2. Wiederhole solange $m \neq n$...
 - Verringere die größere Zahl um die kleinere
 3. m ist der ggT

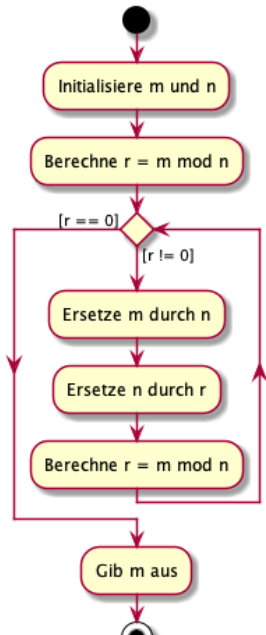
Differenzalgorithmus (Aktivitätsdiagramm)



```
class DifferenceGcd {  
    public static void main(String[] args) {  
        int m = Integer.parseInt(args[0]);  
        int n = Integer.parseInt(args[1]);  
        while (m  $\neq$  n) {  
            if (m > n) {  
                m = m - n;  
            } else {  
                n = n - m;  
            }  
        }  
        System.out.println(m);  
    }  
}
```

- Euklids Algorithmus zur Berechnung des ggT
- Umgangssprachlich:
 1. Initialisiere m und n
 2. Wiederhole bis die Division m/n aufgeht ...
 - 2.1 Berechne $r = m \bmod n$
 - 2.2 Ersetze m durch n
 - 2.3 Ersetze n durch r
 3. n ist der ggT

Euklids Algorithmus (Aktivitätsdiagramm)



```
class EuclidGcd {  
    public static void main(String[] args) {  
        int m = Integer.parseInt(args[0]);  
        int n = Integer.parseInt(args[1]);  
        int r = m % n;  
        while (r != 0) {  
            m = n;  
            n = r;  
            r = m % n;  
        }  
        System.out.println(n);  
    }  
}
```


- **Offene Schleifen**

- Anzahl Schleifendurchgänge vorher nicht bekannt
- Beispiele: GCD
- Gefahr einer Endlosschleife

- **Zählschleifen**

- Anzahl Schleifendurchgänge liegt fest
- Kontrolle mit einem „Schleifenzähler“
- Beispiel: 1×1-Tabelle
- Syntaktische Alternative: **for**-Schleife

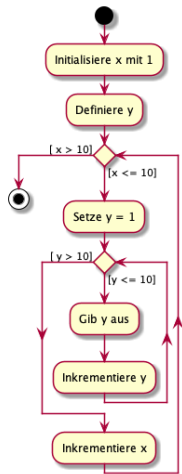
1×1-Tabelle (Ausgabe)

```
$ java MultiplicationTable
```

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

1x1-Tabelle (Implementierung)

```
class MultiplicationTable {  
    public static void main(String[] args) {  
        int x = 1;  
        int y;  
        while (x ≤ 10) { // äußere Schleife  
            y = 1;  
            while (y ≤ 10) { // innere Schleife  
                System.out.printf("%4d", x * y);  
                y = y + 1;  
            }  
            x = x + 1;  
            System.out.println();  
        }  
    }  
}
```



- Alternativen zur Formulierung von Zählschleifen:
 - Start mit 0 oder 1
 - Test des Endwertes mit $<$ oder \leq
- Allgemein üblich: Start mit 0, Test mit $<$
- Endwert = Anzahl Schleifendurchgänge **counter** durchläuft Werte 0, 1, ..., 9

```
int counter = 0;
while (counter < 10) {
    counter = counter + 1;
}
```

one-off errors (2)

- Gelegentlich notwendig: Start mit 1, Test mit \leq
- Beispiel: Verarbeitung von Kalendermonaten, `month` durchläuft Werte 1, 2, ..., 12

```
int month = 1;
while (month ≤ 12) {
    ...
    month = month + 1;
}
```

- Andere Kombination kaum jemals sinnvoll
- häufige Fehlerquelle one-off errors

Was wird ausgegeben?

```
int counter = 0;
while (counter < 10) {
    counter = counter + 1;
}
System.out.println(counter);
```

Operatorzuweisungen

- Häufig gebraucht: Wertzuweisung der Form
`variable = variable operator expression;` wobei...

<code>variable</code>	auf beiden Seiten die selbe Variable
<code>operator</code>	binärer Operator
<code>expression</code>	kompatibler Ausdruck

- Äquivalente Schreibweise als **Operatorzuweisung**:
`variable operator= expression;`

- Beispiel:

```
x = x + 2;
```

```
x += 2;           // äquivalent
```

- Syntaktische Kurzform ohne neue Funktionalität

- Schleifenvariablen oft in Einerschritten nach oben oder unten gezählt:

```
variable = variable + 1;
```

```
variable = variable - 1;
```

- Mit Operatorzuweisungen:

```
variable += 1;
```

```
variable -= 1;
```

- Noch kürzer mit **Inkrementoperator ++** (**Dekrementoperator --**):

```
variable++;
```

```
variable--;
```

- Folgende Anweisungen sind (fast) äquivalent:

```
variable = variable + 1;
```

```
variable += 1;
```

```
variable++;
```


Inkrement- und Dekrementoperator (Beispiele)

```
int counter = 0;
while (counter < 10)
    counter++;
```

```
class MultiplicationTable {
    public static void main(String[] args) {
        int x = 1;
        int y;
        while (x ≤ 10) { // äußere Schleife
            y = 1;
            while (y ≤ 10) { // innere Schleife
                System.out.printf("%4d", x*y);
                y++;
            }
            x++;
            System.out.println();
        }
    }
}
```

Inkrement und Dekrement als Ausdruck

- `variable++` ist Anweisung und Ausdruck
- Wert von `variable++` = Wert der Variablen vor dem Inkrementieren (entsprechend `variable--` vor dem Dekrementieren)

```
int a = 1;
System.out.println(a++); // gibt 1 aus
System.out.println(a++); // gibt 2 aus
System.out.println(a);   // gibt 3 aus
```

- `++`, `--` wahlweise als Präfix- oder Postfixoperatoren
 - **`++variable`** inkrementiert (dekrementiert) zuerst, liefert dann den Wert
 - **`variable++`** liefert zuerst den Wert, inkrementiert (dekrementiert) dann

Inkrement und Dekrement als Ausdruck (2)

- Beispiel (vergleiche mit obigem Beispiel)

```
int a = 1;
System.out.println(++a);    // gibt 2 aus
System.out.println(++a);    // gibt 3 aus
System.out.println(a);      // gibt 3 aus
```

- Ausdrücke mit ++ und -- schwer lesbar. Beispiel:

```
int a = 1;
int b = a+++ (a+++ ++a);    // b = ?
```

- **Tipp:** ++, -- nicht in zusammengesetzten Ausdrücken verwenden.

- Dreistelliger **bedingter Operator** (engl. conditional operator)
- Syntax mit Operatorzeichen `?` und `:`

```
condition ? yes-expression : no-expression
```

- Ablauf:
 1. condition auswerten
 2. Falls `true`: yes-expression auswerten
Falls `false`: no-expression auswerten

```
int a = ... ;  
int b = a == 0 ? 1 : 2;
```

Bedingter Operator (2)

- Typ der beiden expressions kompatibel

```
int a = ... ;  
// ok:  
double d1 = a == 0 ? 1 : 2.0;  
// Fehler:  
double d2 = a == 0 ? 1 : false;  
// ok, aber überflüssig:  
boolean b1 = a == 0 ? true : false;  
// äquivalent zu b1 (besser, da kürzer):  
boolean b2 = a == 0;
```

Bedingter Operator (3)

- Beziehung zu `if`:

```
variable = condition ? yes-expression  
                : no-expression;
```

äquivalent zu:

```
if (condition)  
    variable = yes-expression;  
else  
    variable = no-expression;
```

- Syntaktisch
 - `?:` = Ausdruck, liefert einen Wert, Teil einer Anweisung
 - `if` = Anweisung, ohne Wert

Bedingter Operator - Teilweise Auswertung

- Nur zwei der drei Operanden werden berechnet, der dritte wird nicht berechnet

```
int a = ... ;  
int b = a == 0 ? 0 : 1/a;      // ok
```

- Beispiel: Maximum

```
max = a > b ? a : b;
```

- Zwei oder mehr Anwendung übertrieben, `if`-Anweisung ist klarer

```
max = a > b ? (a > c ? a : c) : (b > c ? b : c);
```

Geschachtelte Schleifen

- **while**-Schleife ist selbst eine Anweisung, kann im Rumpf einer weiteren Schleife stehen: **geschachtelte Schleifen**
- Schematisch

```
while (condition) {           // äußere Schleife
    while (condition) {      // innere Schleife
        statement ...
    }
}
```

- Beispiel: 1x1-Tabelle
- **Hotspot** = häufig ausgeführte Anweisungen
- Volksmund:
90% der Zeit verbringt ein Programm in 10% des Codes.

- **do-Schleife** (auch „do/while-Schleife“) = alternative Schleifenkonstruktion
- Syntax

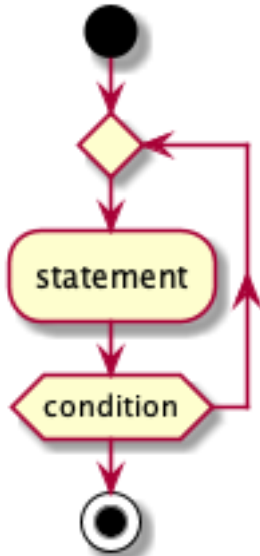
```
do
```

```
    statement
```

```
while (condition);
```

- **boolean**-Ausdruck `condition` steuert Ablauf:
 1. `statement` ausführen
 2. `condition` auswerten
 3. Falls `true`: Zurück zu 1.
andernfalls: Schleife beendet

do-Schleifen (Aktivitätsdiagramm)



do-Schleifen (Beispiel)

Augensumme beim Würfeln berechnen, bei einer 6 noch einmal würfeln:

```
int s = 0;
int w;
do {
    w = ...
    s += w;
} while (w == 6);
System.out.println(s);
```

Schleifenarten

- **for** (kommt später) und **while** arbeiten anders als **do**
- Bezeichnungen:
 - **Abweisende Schleife**: Test am Anfang, eventuell kein Durchlauf
 - **Annehmende Schleife**: Test am Ende, in jedem Fall wenigstens ein Durchlauf
- Merkmale

Schleifenart	Abweisend	Annehmend
Java	while, for	do
Durchläufe	eventuell keine	wenigstens einmal
Gebrauch	häufig	selten

- Manchmal auch **kopfgesteuerte** und **fussgesteuerte** Schleifen

break und **continue**

- Anweisungen **break** und **continue** unterbrechen den normalen Ablauf von Schleifen
- Im Rumpf von Schleifen zulässig (außerdem **break** in **switch**).
- Zweck: Kürzere Formulierung von Schleifen
- Aber: Stehen der strukturierten Programmierung eigentlich entgegen
- Notwendig bei foreach-Schleifen

- Anweisung **break** beendet eine Schleife sofort, der Rest des Rumpfes wird übersprungen
- **break** = einfache Anweisung (wie Definitionen, Wertzuweisungen)
- Zweck: Entscheidung über Fortsetzung einer Schleife fällt mitten im Rumpf

- ohne break

```
boolean flag = true;
while (flag) {
    ... // Erster Teil des Rumpfes
    flag = ...;
    if (flag) {
        ... // Rest des Rumpfes
    }
}
```

- mit break

```
while (true) {
    ... // Erster Teil des Rumpfes
    if (!condition)
        break;
```


mit `break` ohne doppelte Berechnung des Divisionsrestes:

```
while (true) {  
    int r = m % n;  
    if (r == 0)  
        break;  
    m = n;  
    n = r;  
}
```

- Anweisung `continue` startet sofort den nächsten Schleifendurchlauf, der Rest des Rumpfes wird übersprungen
- Wie `break`: Nützlich zur Behandlung von Sonderfällen
- Zweck: Folge von Entscheidungen über Fortsetzung des Schleifendurchlaufes mitten im Rumpf

Schematisch

- ohne `continue`

```
while ( ... ) {  
    ...  
    if (condition1) {  
        ...  
        if (condition2) {  
            ...  
            if (condition3) {  
                ...  
            }  
        }  
    }  
}
```

- mit `continue`

```
while ( ... ) {  
    ...  
    if (!condition1)  
        continue;  
    ...  
    if (!condition2)  
        continue;  
    ...  
    if (!condition3)  
        continue;  
    ...  
}
```

- `break` und `continue` spalten Kontrollfluss: (wenn überhaupt) mit

Gültigkeitsbereiche

- Oben eingeführte Blöcke gruppieren Anweisungen
- Innerhalb eines Blocks alle Anweisungsarten erlaubt, auch Definitionen
- **Gültigkeitsbereich** (engl. scope) einer Variablen...
 - beginnt mit der Definition und
 - endet mit dem Block, in dem die Definition steht.
- Außerhalb des Blocks: Variable „gilt nicht“

- 1x1-Tabelle

```
1  class MultiplicationTable {
2      public static void main(String[] args) {
3          int x = 1;
4          while (x ≤ 10) { // äußere Schleife
5              int y = 1;
6              while (y ≤ 10) { // innere Schleife
7                  System.out.printf("%4d", x * y);
8                  y++;
9              }
10             x++;
11             System.out.println();
12         }
13     }
14 }
Kontrollstrukturen }
```

Beispiel – Gültigkeitsbereiche

- Gültigkeitsbereich von `x`
 - ab Zeile 3 **nach** `;`
 - bis Zeile 13 **vor** `}`
- Gültigkeitsbereich von `y`
 - ab Zeile 5 **nach** `;`
 - bis Zeile 12 **vor** `}`
- Gültigkeitsbereiche bezogen auf Quelltext, werden vom Compiler überprüft
- Zur Laufzeit irrelevant

Namenskollision

- Gültigkeitsbereich umfasst untergeordnete (geschachtelte) Blöcke
- Vorhergehendes Beispiel: `x` in beiden geschachtelten Blöcken verfügbar
- Namenskollision: Definition des gleichen Namens, wie in einem umfassenden Block

```
int x;  
...  
while (x ≤ 10) {  
    int x;    // Namenskollision!  
    ...  
}  
...
```

- Java: Doppelte Definition unzulässig

- Aber: Kein Problem in disjunkten Blöcken:

```
while (...) {  
    int y;  
    ...  
}  
while (...) {  
    int y;  
    ...  
}
```

Zwei isolierte, unabhängige Variablen, die (zufällig) beide „y“ heißen

- **Lebensdauer** = Zeitintervall, für das eine Variable zur Laufzeit existiert
- Die gleiche Variable wird möglicherweise vielfach geschaffen und wieder freigegeben (**Inkarnationen**)
- Aufeinander folgende Inkarnationen unabhängig 1×1-Programm:
y wird 10-mal geschaffen und 10-mal freigegeben
- Schaffen und Freigeben praktisch ohne Laufzeitkosten
- Vergleich:
 - Gültigkeitsbereich = Zeilen im Quelltext
 - Lebensdauer = Zeitspanne beim Ablauf

Zählschleifen (**for**)

- **for**-Schleifen = Sprachmittel für Zählschleifen
- Etwas barocke Konstruktion, Erbstück der Programmiersprache „C“
- kompakter als **while**-Schleifen
- Syntax

```
for (start; condition; next)  
    statement
```

- start und next sind Anweisungen

Aktivitätsdiagramm

- Aktivitätsdiagramme für `for`-Schleifen gibt es nicht, daher mit `while`-Schleifen simulieren

```
for (start; condition; next)  
  statement
```

```
start;
```

```
while (condition)  
  statement  
next
```

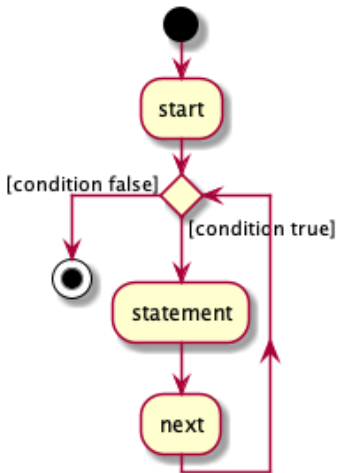


Abbildung: for-Schleife

- Werte 0 bis 9 ausgeben:

```
for (int i = 0; i < 10; i++)  
    System.out.println(i);
```

entspricht

```
int i = 0;  
while (i < 10) {  
    System.out.println(i);  
    i++;  
}
```

Beispiele (1)

- Berechnung der Zahlensumme $1 + 2 + \dots + n$ mit **for**-Schleife:

```
final int upto = 100;
int sum = 0;
for (int counter = 1; counter ≤ upto; counter++)
    sum += counter;
System.out.println(sum);
```

- Euklid-Algorithmus:

```
for (int r = m%n; r ≠ 0; r = m%n) {
    m = n;
    n = r;
}
```

- 1×1-Tabelle:

```
for (int x = 1; x ≤ 10; x++) {  
    for (int y = 1; y ≤ 10; y++)  
        System.out.printf("%4d", x*y);  
    System.out.println();  
}
```


- **for**- und **while**-Schleifen gegenseitig ersetzbar

```
for (start; condition; next)
```

```
    statement
```

```
{
```

```
    start;
```

```
    while (condition) {
```

```
        statement
```

```
        next;
```

```
    }
```

```
}
```

Gültigkeit von Zählvariablen

- Gültigkeitsbereich der Zählvariablen von **for**-Schleifen: Kopf und Rumpf

```
for (int x = 0; x ≤ 10; x++) {  
    System.out.println(x);  
}
```

- Folge: Mehrere aufeinander folgende **for**-Schleifen mit gleich benannten Laufvariablen zulässig:

```
for (int x = 0; x ≤ 10; x++) {  
    System.out.println(x);  
}  
for (int x = 0; x ≤ 10; x++) {  
    System.out.println(x);  
}
```

Verteiler (**switch**)

- **switch**-Anweisungen („Verteiler“) ersetzen längere, unübersichtliche `if`-Kaskaden
- Syntax
- Semantik:

```
switch (expression) {  
  case label1:  
    statement ...  
    break;  
  case label2:  
    statement ...  
    break;  
  ...  
}
```

1. Der Wert der **expression** wird einmal berechnet.
2. Das Ergebnis wird nacheinander mit den **labels** verglichen, bis zum ersten gleichen Wert.
3. Die dem **label** nachfolgenden **statements** werden ausgeführt, bis zum **break**;

Berechnung der Anzahl Tage im Monat, hier mit `switch` statt `if`-Kaskade:

```
switch (month) {  
  case 1:  
    days = 31;  
    break;  
  case 2:  
    days = 28;  
    break;  
  case 3:  
    days = 31;  
    break;  
  ...  
  case 12:  
    days = 31;  
    break;
```

- `case`-Labels müssen eindeutig sein, doppelte Werte unzulässig
- und müssen konstant (vom Compiler berechenbar) sein
- Wenn kein `case`-Label passt, geschieht nichts (ganzes `switch` wirkt wie eine leere Anweisung)
- Mehrere (verschiedene) `case`-Labels vor einer Anweisungsfolge sind zulässig.

- Beispiel:

```
switch (month) {  
  case 1: case 3: case 5: case 7:  
  case 8: case 10: case 12:  
    days = 31;  
    break;  
  case 2:  
    days = 28;  
    break;  
  case 4: case 6: case 9: case 11:  
    days = 30;  
    break;  
}
```

Defaultfall

- `default` = spezielles `case`-Label, passt auf alle übrigen Werte
- `default` darf nur einmal und nur am Ende genannt werden

```
switch (month) {  
    case 2:  
        days = 28;  
        break;  
    case 4: case 6: case 9: case 11:  
        days = 30;  
        break;  
    default:    // alle übrigen Monate  
        days = 31;  
        break;  
}
```


Fall through

- **break** beendet **switch**
- Falls **break** fehlt, wird mit den Anweisungen des nächsten Zweiges fortgefahren (fall through)
- Schlechtes Beispiel:

```
int days = 31;
switch (month) {
    case 2:
        days--;
        days--; // kein break - fall through
    case 4: case 6: case 9: case 11:
        days--; // kein break, kein default - !?
}
```

- `switch`-Rumpf = Gültigkeitsbereich
- Definitionen im `switch`-Rumpf gelten immer, nicht aber Initialisierungen

```
switch ( ... ) {  
    case 0:  
        int y = 0;  
        System.out.println(y);  
        break;  
    case 1:  
        // Fehler: y definiert,  
        // aber nicht initialisiert  
        System.out.println(y);  
        break;  
}
```

```
switch ( ... ) {  
    case 0:  
        final int y;  
        y = 0; // ok, einzige Wertzuweisung  
        System.out.println(y);  
        break;  
    case 1:  
        y = 1; // ok, einzige Wertzuweisung  
        System.out.println(y);  
        break;  
}
```

- *Unübersichtlich: besser keine Definitionen im switch-Rumpf*

- **switch** ist selbst eine Anweisung
⇒ kann in einem übergeordneten **switch** stehen
- Nützlich um unregelmäßige Tabellen zu implementieren Funktion $f(x, y)$
definiert als:

	$y = 0$	$y = 1$	$y = 2$
$x = 0$	1	undefiniert	2
$x = 1$	2	0	undefiniert
$x = 2$	undefiniert	1	3

Programmfragment (2/2)

```
boolean undefined = false;
int value = 0;
switch (x) {
    case 0:
        switch (y) {
            case 0: value = 1;           break;
            case 1: undefined = true;   break;
            case 2: value = 2;           break;
        }
        break;
    case 1:
        switch (y) {
            case 0: value = 2;           break;
            case 1: value = 0;           break;
            ...
            ...
        }
    ...
}
```

- Typ `int` als `switch`-Ausdruck zulässig
- `switch` nicht zulässig mit Typen ...
 - `double`: Test von exakten Werten problematisch wegen Rundungsfehlern
 - `boolean`: nur zwei mögliche Werte, `if` völlig ausreichend
- Allgemein: ganzzahlige Typen, Aufzählungstypen (siehe später) und Strings (seit Java 7).