252-0027
Einführung in die Programmierung

3. Kontrollstrukturen

Manuela Fischer, Malte Schwerhoff

Departement Informatik
ETH Zürich

3. Kontrollstrukturen

3.1 Verzweigungen

3.2 Schleifen I

3.3 Methoden II

3.4 Schleifen II

Programmstruktur bisher

```
public class name {
    public static void main(String[] args) {
        statement;
        statement;
        ...
        statement;
    }
}
```

Klasse = ein Programm mit Namen *name*

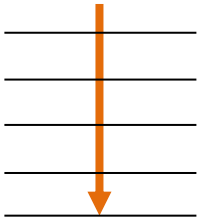
Methode `main` = Beginn der Ausführung des Programms

Folge von Anweisungen = Funktionalität des Programm

Programmstruktur

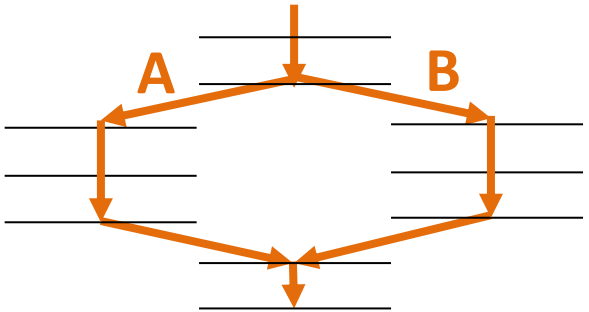
Bisher:

- linear, von oben nach unten
- eine Anweisung nach der anderen



Ziel: Verzweigungen

- Bedingte Ausführung von Code
- Entweder A oder B
- Basierend auf einer Entscheidung

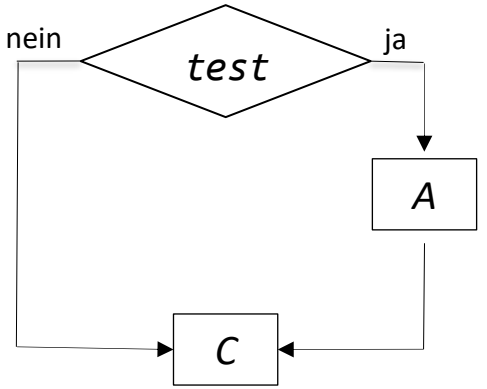


If-Anweisung («if statement»)

- Führe eine Gruppe *A* von Anweisungen nur dann aus, wenn eine Bedingung *test* wahr ist (*bedingte Ausführung, «conditional execution»*)

```
if (test) {
    A
}
C
```

if-Block



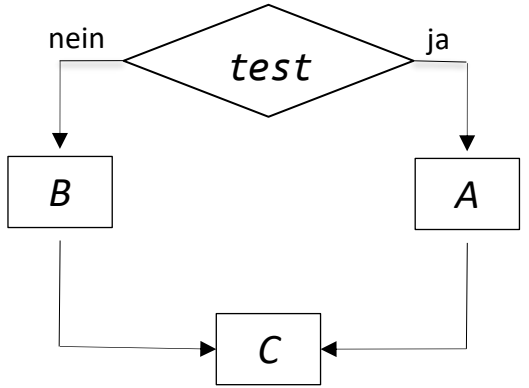
If-else-Anweisung («if-else statement»)

- Führe entweder eine Gruppe *A* von Anweisungen oder eine Gruppe *B* aus, abhängig von Bedingung *test*

```
if (test) {
    A
} else {
    B
}
C
```

if-Block

else-Block

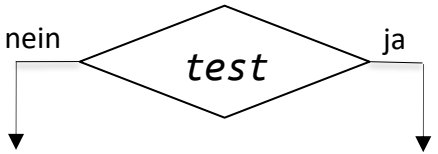


Beispiel

```
double punkte = console.nextDouble();

if (punkte >= 50.0) {
  System.out.println("Prüfung bestanden");
} else { // punkte < 50.0
  System.out.println("Prüfung nicht bestanden");
}
```

Bedingung in if-Statements

```
if (test) {
    A
}
```



- Als Bedingung (*test*) in if-Statements verwenden wir Boolesche Ausdrücke
- **Boolescher Ausdruck**: ein Ausdruck, der entweder wahr oder falsch ist
  - vom Typ `boolean` (primitiver Typ)
  - Hat den Wert `true` oder `false` (Wahrheitswerte)
  - Beispiele:
    - Literale (Konstanten): `true`, `false`
    - Vergleiche von Werten, z.B. `x >= 0`

Vergleichsoperatoren (Relationale Operatoren)

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

- Nicht alle Operatoren können für alle Typen (sinnvoll) angewendet werden
  - `1 < true` ergibt keinen Sinn → Compiler-Fehler
- Boolesche Operatoren haben tiefere Präzedenz als arithmetische
  - `1 + 1 == 2` entspricht `(1 + 1) == 2` und nicht `1 + (1 == 2)`

Beispiele

- Auswertungsreihenfolge (arithmetisch vor boolesch)

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

- Verkettung (nicht erlaubt!)

```
2 <= x <= 10
true   <= 10 // ERROR: operator <= is undefined for boolean, int
```

```
a <= b <= c
bedeutet
(a <= b) <= c
```

= versus == in Java

- == ist der Vergleichsoperator («comparison operator»)
  - Wahrheitswert, ob linke und rechte Seite gleich sind

- = ist der Zuweisungsoperator («assignment operator»)
  - Zuweisung `x = 4` hat den Typ `int` und den Wert `4`
  - kein Wahrheitswert!

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

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

Beispiel

```
boolean b = false;
if (b = true) {
  System.out.println("This will never be printed.");
} else {
  System.out.println("This will always be printed.");
}
```

Zuweisung hat Wert `true`

```
This will never be printed.
```

Logische (oder boolesche) Operatoren

- Boolesche Ausdrücke können durch **Logische Operatoren** zu neuen Booleschen Ausdrücken verknüpft werden

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

Präzedenz: `&&` vor `||`

Links-assoziativ

- Wahrheitstabelle für diese Operatoren für Boolesche Ausdrücke 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

Fragen: Wozu evaluieren diese Ausdrücke?

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

Verkettung: so geht es!

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)`

Beispiel

- Boolescher Ausdruck, der angibt, ob `jahr` ein Schaltjahr ist.
  - durch 4 teilbar (ohne Rest) und nicht durch 100 teilbar (ohne Rest),
  - ausser wenn durch 400 teilbar (ohne Rest)

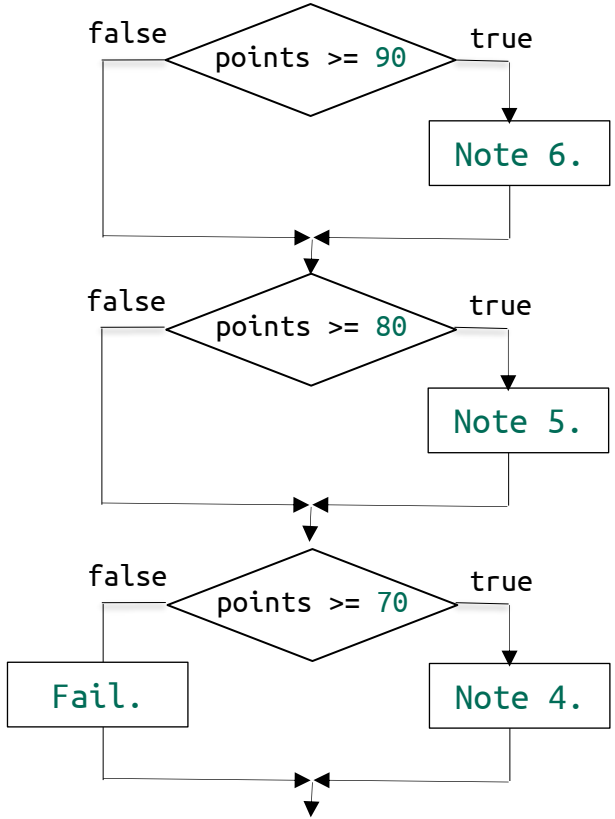
```
jahr % 4 == 0 && jahr % 100 ! = 0 || jahr % 400 == 0
```

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

Mit Klammern:
leserfreundlicher

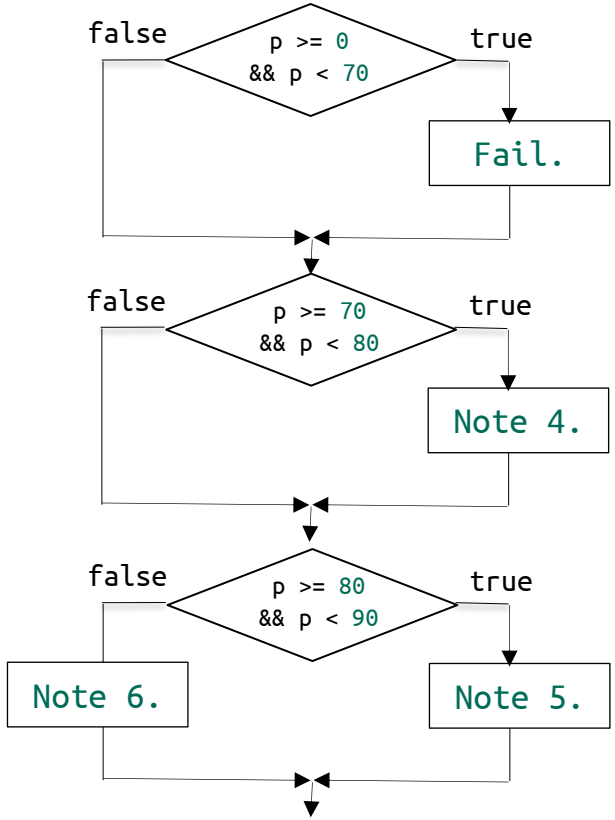
Was ist die Ausgabe?

```
int points = 80;
if (points >= 90) {
  System.out.println("Note 6.");
}
if (points >= 80) {
  System.out.println("Note 5.");
}
if (points >= 70) {
  System.out.println("Note 4.");
} else {
  System.out.println("Fail.");
}
```



Besser?

```
int points = ...;
if (points >= 0 && points < 70) {
  System.out.println("Fail.");
}
if (points >= 70 && points < 80) {
  System.out.println("Note 4.");
}
if (points >= 80 && points < 90) {
  System.out.println("Note 5.");
} else {
  System.out.println("Note 6.");
}
```



Schlechtes Beispiel

Im Internet gefundener Code …
(auf r/programminghorror)

```
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 "🔵🔵🔵🔵🔵🔵🔵🔵🔵🔵";
}
```

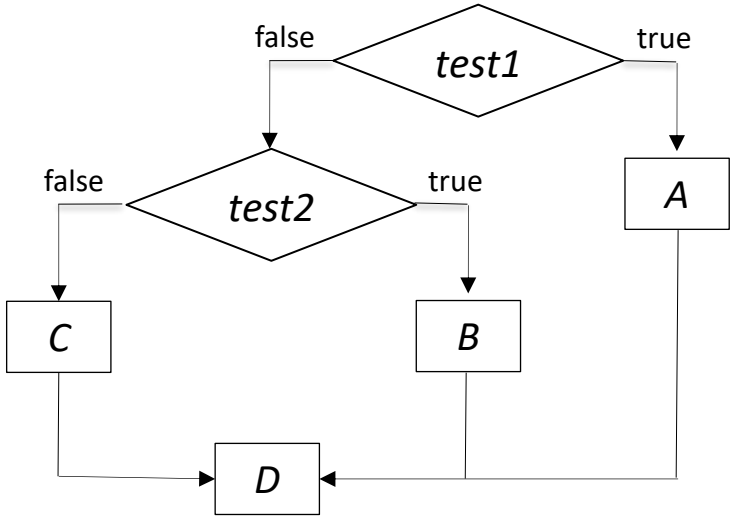
If-else-if-Anweisung («if-else-if statement»)

```
if (test1) {
    A
} else if (test2) {
    B
} else {
    C
}
D
```

if-Block

else-if-Block

else-Block



Auswahl bestimmt durch mehrere Tests!

Beispiel

```
if (x > 0) {
  System.out.println("positive");
} else if (x < 0) {
  System.out.println("negative");
} else { // x == 0
  System.out.println("zero");
}
```

Beispiel

```
if (place == 1) {
  System.out.println("Gold!");
} else if (place == 2) {
  System.out.println("Silver!");
} else if (place == 3) {
  System.out.println("Bronze!");
} else {
  System.out.println("No luck!");
}
```

Beliebig viele else-if-Blöcke möglich

Unterschiedliche if-Konstrukte

```
if (test1) {
    A
} else if (test2) {
    B
} else {
    C
}
```

if/else
gegenseitiger Ausschluss
genau eines von *A, B, C*

```
if (test1) {
    A
} else if (test2) {
    B
} else if (test3) {
    C
}
```

if/else-if
gegenseitiger Ausschluss
höchstens eines von *A, B, C*

```
if (test1) {
    A
}
if (test2) {
    B
}
if (test3) {
    C
}
```

if/if
kein gegenseitiger Ausschluss
beliebig viele von *A, B, C*

Verbesserungsvorschläge?

```
boolean isPrime = ...;
if (isPrime == true) {
  System.out.println("Prime number!");
} else {
  System.out.println("No prime number!");
}
```

Schlechter Stil

```
boolean test = ...;
if (test == true) {
    …
}
```

```
boolean test = ...;
if (test == false) {
    …
}
```

Redundant

```
boolean test = ...;
if (test) {
    …
}
```

```
boolean test = ...;
if (!test) {
    …
}
```

Elegant(er)

Ternärer Operator («ternary operator»)

```
test ? valueTrue : valueFalse
```

- `test` ist Boolescher Ausdruck
- `valueTrue` und `valueFalse` sind Ausdrücke (typischerweise) vom gleichen Typ T
- Ergebnis des ternären Operators hat Typ T
  - Falls `test` wahr:
    Ergebnis ist `valueTrue`
  - Falls `test` falsch:
    Ergebnis ist `valueFalse`

```
if (test) {
  System.out.println("A");
} else {
  System.out.println("B");
}
```

```
System.out.println(test ? "A" : "B");
```

Weglassen der geschweiften Klammern

```
if (test) {
     statementTrue;
} else {
    statementFalse;
}
```

```
if (test) statementTrue;
else statementFalse;
```

- Falls Block nur aus einer Anweisung besteht, können geschweifte Klammern weggelassen werden
- Empfehlung: Klammern immer nutzen, um Fehler bei der Weiterentwicklung zu vermeiden

Suchen Sie einmal nach «apple goto fail»

Kurzschlussauswertung («short-circuit evaluation»)

Bei Berechnung von `p && q` und `p || q`

- Java wertet zuerst den linken Operanden `(p)` und dann den rechten `(q)` aus
- Die Auswertung wird beendet, sobald das Ergebnis feststeht
  - Bei `p && q`: Falls `p == false`, dann ist `p && q == false`
  - Bei `p || q`: Falls `p == true`, dann ist `p || q == true`

p	q	p && q
true	true	true
true	false	false
false	true	false
false	false	false

p	q	p || q
true	true	true
true	false	true
false	true	true
false	false	false

In den orange hinterlegten Fällen erfolgt keine Auswertung von q.

Kurzschlussauswertung

Bedingte Auswertung von `q` bei `p && q` und `p || q`

- bei `p && q` nur falls `p == true`
- bei `p || q` nur falls `p == false`

Konsequenzen

- nicht-kommutativ
  - `p && q` ist nicht zwingend äquivalent zu `q && p`
  - `p || q` ist nicht zwingend äquivalent zu `q || p`
- Bedingte Ausführung möglicher Seiteneffekte («side effects») von `q`

Werden später diskutiert

Gründe

- Sicherheit:
  `q` darf nur unter gewissen Bedingungen (`p`) ausgewertet werden (`p && q`)
- Effizienz:
  `q` muss nicht ausgewertet werden, falls `p` bereits erfolgreich war (`p || q`)

Kurzschlussauswertung: Beispiel

```
int a = …;
int b = …;
if (a / b > 2) { … }
```

Was wenn b == 0?

```
…
if (b != 0 && a / b > 2) { … }
```

bei b == 0 wird die rechte Seite nicht evaluiert

```
   b != 0 && (a / b < 2)
➜  false  && (a / b < 2)
➜  false  && (a / b < 2)
```

Wichtig: Reihenfolge der Operanden ist relevant

```
…
if (a / b > 2 && b != 0 ) { … }
```

Division durch 0 möglich, dann Laufzeitfehler

De Morgan’s Rules

- Distributionsregeln für die Negation boolescher Ausdrücke

Ursprünglicher Ausdruck	Negierter Ausdruck	Negiert (de Morgan)
`a && b`	`!(a && b)`	`!a || !b`
`a || b`	`!(a || b)`	`!a && !b`

- Beispiel:

Original	Negiert (de Morgan)

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

3. Kontrollstrukturen

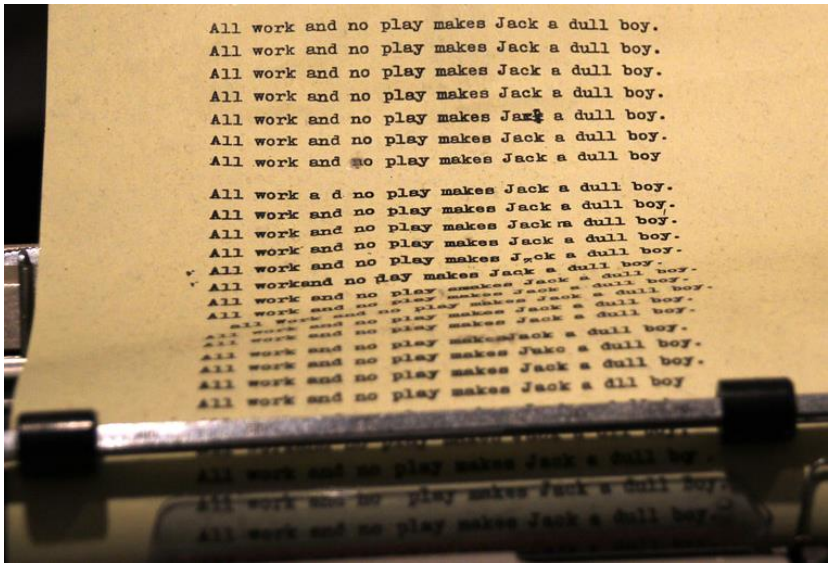
3.2 Schleifen

3.2.1 (Einfache) for-Schleife

3.2.2 Verschachtelte for-Schleifen

3.2.3 while-Schleife

Beispiel 1: Wiederholter Text

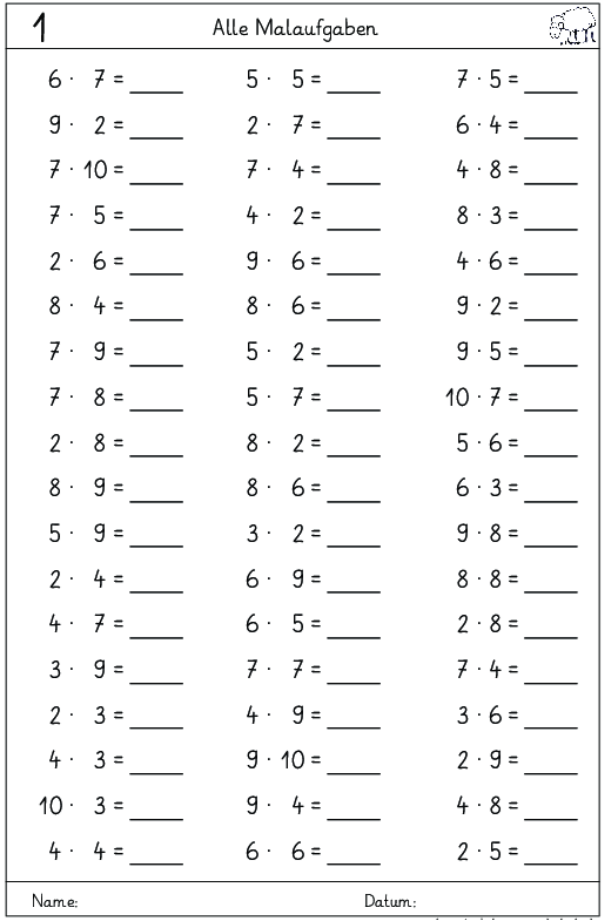


Beispiel 1: Wiederholter Text mit Java

```
System.out.println("I WILL NOT WASTE CHALK.");
System.out.println("I WILL NOT WASTE CHALK.");
System.out.println("I WILL NOT WASTE CHALK.");
System.out.println("I WILL NOT WASTE CHALK.");
System.out.println("I WILL NOT WASTE CHALK.");
System.out.println("I WILL NOT WASTE CHALK.");
System.out.println("I WILL NOT WASTE CHALK.");
System.out.println("I WILL NOT WASTE CHALK.");
System.out.println("I WILL NOT WASTE CHALK.");
System.out.println("I WILL NOT WASTE CHALK.");
System.out.println("I WILL NOT WASTE CHALK.");
System.out.println("I WILL NOT WASTE CHALK.");
System.out.println("I WILL NOT WASTE CHALK.");
```

Beispiel 2: Einmaleins-Aufgaben

- Zufällige Einmaleins-Aufgaben für Übungen generieren



1 Alle Malaufgaben

6 · 7 = ____	5 · 5 = ____	7 · 5 = ____
9 · 2 = ____	2 · 7 = ____	6 · 4 = ____
7 · 10 = ____	7 · 4 = ____	4 · 8 = ____
7 · 5 = ____	4 · 2 = ____	8 · 3 = ____
2 · 6 = ____	9 · 6 = ____	4 · 6 = ____
8 · 4 = ____	8 · 6 = ____	9 · 2 = ____
7 · 9 = ____	5 · 2 = ____	9 · 5 = ____
7 · 8 = ____	5 · 7 = ____	10 · 7 = ____
2 · 8 = ____	8 · 2 = ____	5 · 6 = ____
8 · 9 = ____	8 · 6 = ____	6 · 3 = ____
5 · 9 = ____	3 · 2 = ____	9 · 8 = ____
2 · 4 = ____	6 · 9 = ____	8 · 8 = ____
4 · 7 = ____	6 · 5 = ____	2 · 8 = ____
3 · 9 = ____	7 · 7 = ____	7 · 4 = ____
2 · 3 = ____	4 · 9 = ____	3 · 6 = ____
4 · 3 = ____	9 · 10 = ____	2 · 9 = ____
10 · 3 = ____	9 · 4 = ____	4 · 8 = ____
4 · 4 = ____	6 · 6 = ____	2 · 5 = ____

Name: Datum:

www.lernstuebchen-grundschule.de

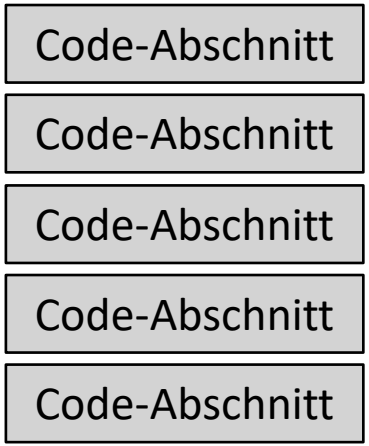
Beispiel 2: Zufällige Einmaleins-Aufgaben mit Java

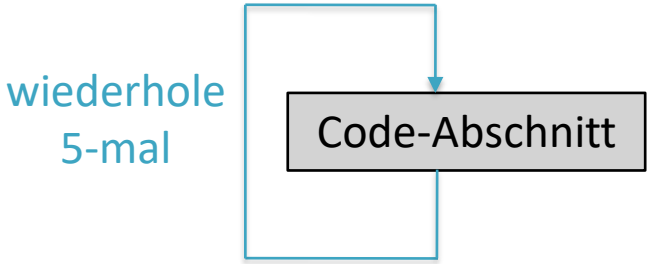
```
int mistakes = 0, first, second;

first = 1 + rand.nextInt(10);
second = 1 + rand.nextInt(10);
System.out.print(first + " * " + second + " = ");
if (console.nextInt() != (first * second)) {
    mistakes = mistakes + 1;
}

first = 1 + rand.nextInt(10);
second = 1 + rand.nextInt(10);
System.out.print(first + " * " + second + " = ");
if (console.nextInt() != (first * second)) {
    mistakes = mistakes + 1;
}

// … and eight more copies …

System.out.println((10 - mistakes) + " out of 10 correct!");
```

// Kopie

// Kopie

Schleife («Loop»)





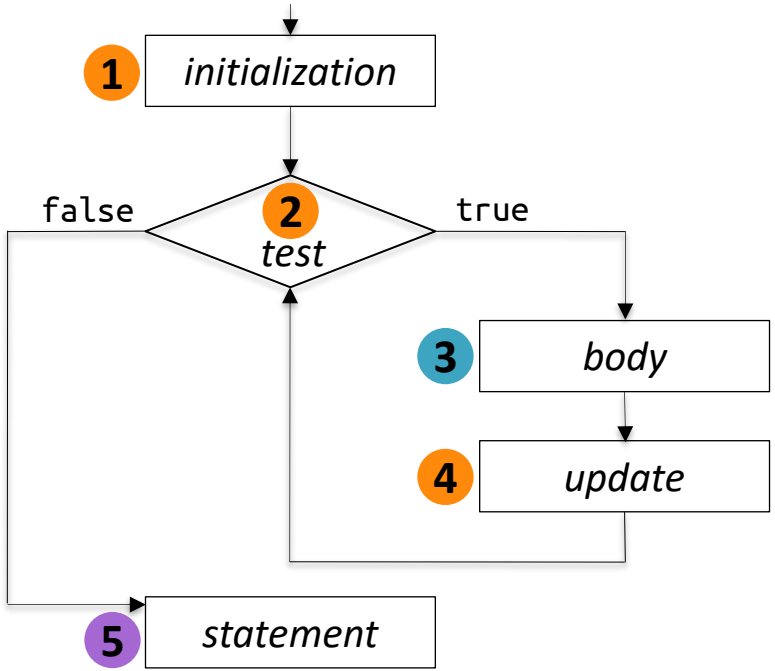
- Erlaubt uns, Code-Abschnitte beliebig oft auszuführen
  - Jede Wiederholung des Code-Abschnitts nennen wir eine **Iteration**

for-Schleife («for loop»): Kontrollfluss

```
for (initialization; test; update) {
    body
}
statement;
```

1 2 4 3 5

```
for (int i = 1; i <= 5; i = i + 1) {
 System.out.println(i); // 1 2 3 4 5
}
```



for-Schleife: Syntax, Semantik

```
for (initialization; test; update) {
    statement1;
    statement2;
    statement3;
    …
}
statement;
```

Kopf («head»)

Rumpf/Körper («body»)

Schleifen-/Laufvariable («loop variable»)

```
for (int i = 1; i <= 5; i = i + 1) {
 System.out.println(i); // 1 2 3 4 5
}
```

Syntax (vereinfacht)

- ***initialization***: Variablendeklaration oder Variablenzuweisung
- ***test***: Boolescher Ausdruck
- ***update***: Ausdruck mit Seiteneffekt

Semantik (siehe auch vorherige Folie)

1. Einmalige Ausführung von ***initialization***
2. Wiederhole bis Abbruch:
   1. Prüfe, ob ***test*** zu `true` evaluiert.
      Falls nein, Abbruch: springe zu ***statement***.
   - Führe Rumpf aus
   - Führe ***update*** aus

for-Schleife: Typische Nutzung

```
for (initialization; test; update) {
    body
}
```

```
for (int i = 1; i <= 5; i = i + 1) {
 System.out.println(i); // 1 2 3 4 5
}
```

- ***initialization*** initialisiert Schleifenvariable
  - Variable nur innerhalb der for-Schleife gültig (benötigt)
- ***test*** überprüft, ob Schleifenvariable einen Wert erreicht hat
  - Typischerweise mit `<` `<=` `>` `>=` `==` `!=`
- ***update*** verändert die Schleifenvariable
  - bringt die Schleifenvariable näher zum Abbruchswert

Beabsichtigtes Verhalten — Compiler überprüft das nicht!

Eingangsbeispiele mittels for-Schleifen

```
for (int i = 1; i <= 12; i = i + 1) {
  System.out.println("I WILL NOT WASTE CHALK.");
}
```

```
int mistakes = 0, first, second;
for (int i = 1; i <= 10; i = i + 1) {
  first = 1 + rand.nextInt(10);
  second = 1 + rand.nextInt(10);
  System.out.print(first + " * " + second + " = ");
  if (console.nextInt() != (first * second)) {
    mistakes = mistakes + 1;
  }
}
```

Beispiel: Mehrere Anweisungen im Rumpf

```
System.out.println("+----+");
for (int i = 1; i <= 3; i = i + 1) {
  System.out.println("\\    /");
  System.out.println("/    \\");
}
System.out.println("+----+");
```

```
+----+
\    /
/    \
\    /
/    \
\    /
/    \
+----+
```

Beispiel: Countdown

Ziel: Countdown 10, 9, 8, …, 1

Runterzählen statt Raufzählen!

```
System.out.print("T minus ");
for (int i = 10; i >= 1; i = i - 1) {
  System.out.print(i + ", ");
}
System.out.println("blastoff!");
```

```
T minus 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, blastoff!
```

Endlosschleife («infinite loop»)

```
for (int i = 10; i % 2 == 0; i = i + 2) {
  System.out.print(i + " ");
}
```

```
10 12 14 16 18 20 22 24 26 28 30 …
```

- Schleife terminiert (= endet) nicht
- Eclipse: Programm muss vom Benutzer abgebrochen werden durch Klick auf 

3. Kontrollstrukturen

3.2 Schleifen

3.2.1 (Einfache) for-Schleife

3.2.2 Verschachtelte for-Schleifen

3.2.3 while-Schleife

Beispiel: Sterne

```
**********
**********
**********
**********
**********
```

```
for (int i = 1; i <= 5; i = i + 1) {
  System.out.println("**********");
}
```

```
for (int i = 1; i <= 5; i = i + 1) {
  System.out.print("*");
  …
  System.out.print("*");
  System.out.println(); // new line
}
```

10-mal;
for-Schleife?

Verschachtelte Schleifen («nested loops»)

```
for (int i = 1; i <= 5; i = i + 1) {
    for (int j = 1; j <= 10; j = j + 1) {
        …
    }
}
```

Innere Schleife

Äussere Schleife

- Schleife innerhalb (im Rumpf) einer anderen Schleife
  - (Typischerweise) mit einer anderen Schleifenvariable (hier `j`)
- Für jeden Schritt (Iteration) der äusseren Schleife wird die innere Schleife einmal vollständig ausgeführt
  - Für jede der 5 Iterationen der äusseren Schleife wird die innere Schleife 10-mal ausgeführt

Beispiel: Sterne

```
for (int i = 1; i <= 5; i = i + 1) {
  System.out.print("*");
  …
  System.out.print("*");
  System.out.println(); // new line
}
```

```
**********
**********
**********
**********
**********
```

```
for (int i = 1; i <= 5; i = i + 1) {
  for (int j = 1; j <= 10; j = j + 1) {
    System.out.print("*");
  }
  System.out.println(); // new line
}
```

Beispiel: Sterne 2.0

```
for (int i = 1; i <= 5; i = i + 1) {
  for (int j = 1; j <= i; j = j + 1) {
    System.out.print("*");
  }
  System.out.println();
}
```

```
*
**
***
****
*****
```

Beispiel: Zweimal Zahlen

```
for (int i = 1; i <= 5; i = i + 1) {
  for (int j = 1; j <= i; j = j + 1) {
    System.out.print(i);
  }
  System.out.println();
}
```

```
1
22
333
4444
55555
```

```
for (int i = 1; i <= 5; i = i + 1) {
  for (int j = 1; j <= i; j = j + 1) {
    System.out.print(j);
  }
  System.out.println();
}
```

```
1
12
123
1234
12345
```

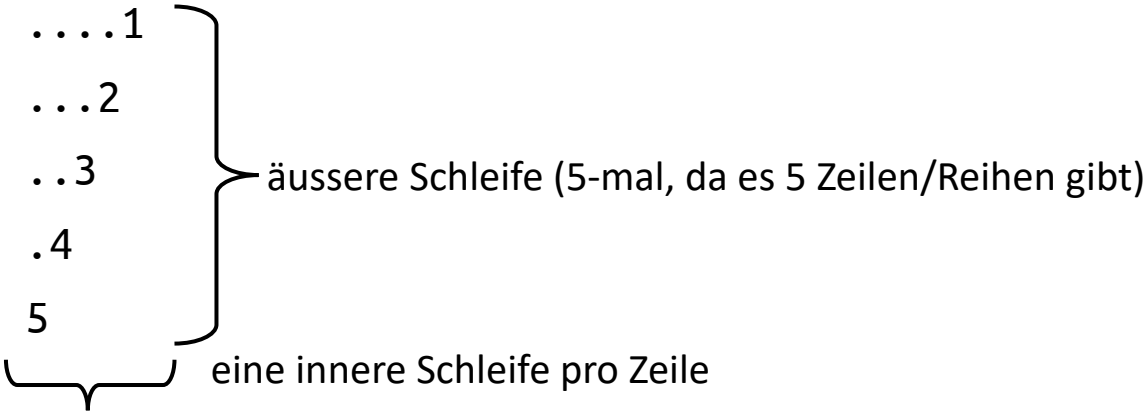
Beispiel: So viele Sterne …

```
for (int i = 1; i <= 5; i = i + 1) {
  for (int j = 1; j <= 10; i = i + 1) {
    System.out.print("*");
  }
  System.out.println();
}
```

Endlosschleifen

```
for (int i = 1; i <= 5; i = i + 1) {
  for (int j = 1; i <= 10; j = j + 1) {
    System.out.print("*");
  }
  System.out.println();
}
```

Beispiel



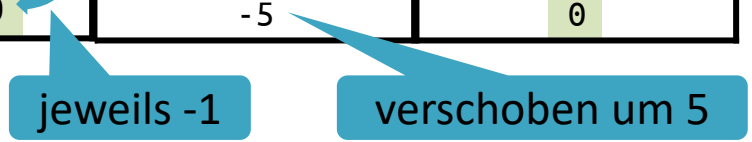
- Äussere Schleife: eine Iteration pro Zeile
  - 5 Iterationen: `row = 1, 2, 3, 4, 5`
- Innere Schleife: Muster der Zeile
  - Ein paar Punkte (abhängig von `row`), dann die Zahl `row`

Beispiel

- Innere Schleife: Muster der Zeile
  - Ein paar Punkte (abhängig von `row`), dann die Zahl `row`
- Wie viele Punkte?
  - Tabelle kann beim Erkennen des Musters helfen

```
....1
...2
..3
.4
5
```

row	# Punkte	(-1)*row	(-1)*row + 5
1	4	-1	4
2	3	-2	3
3	2	-3	2
4	1	-4	1
5	0	-5	0



Beispiel

```
for (int row = 1; row <= 5; row = row + 1) {
  for (int pt = 1; pt <= -row + 5; pt = pt + 1) {
    System.out.print(".");
  }
  System.out.print(row);
  System.out.println();
}
```

```
....1
...2
..3
.4
5
```

3. Kontrollstrukturen

3.2 Schleifen

3.2.1 (Einfache) for-Schleife

3.2.2 Verschachtelte for-Schleifen

3.2.3 while-Schleife

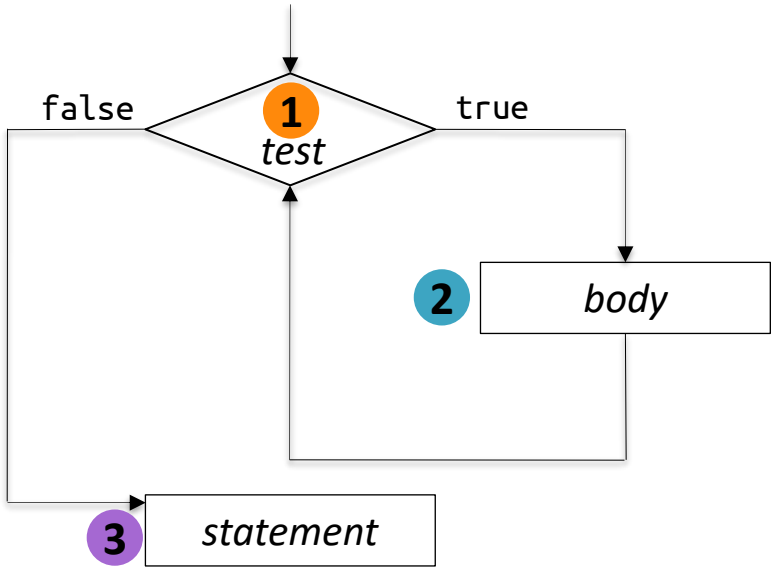
Typen von Schleifen

- **Bestimmte Schleife** («**definite loop**») oder **Zählschleife** («**counting loop**»)
  - Anzahl Iterationen ist vor Beginn der Ausführung der Schleife bekannt
    - Die ersten 10 Quadratzahlen ausgeben
    - Alle ungeraden Zahlen zwischen 7 und 91 ausgeben

- **Unbestimmte Schleife** («**indefinite loop**»)
  - Anzahl Iterationen nicht vor Beginn klar
    - Input von Konsole einlesen bis der Benutzer eine negative Zahl eintippt
    - Datei lesen, bis drei aufeinanderfolgende Sätze mit einem «!» enden
    - Einzahlungen entgegennehmen, bis ein bestimmter Wert überschritten wird

while-Schleife («while loop»)

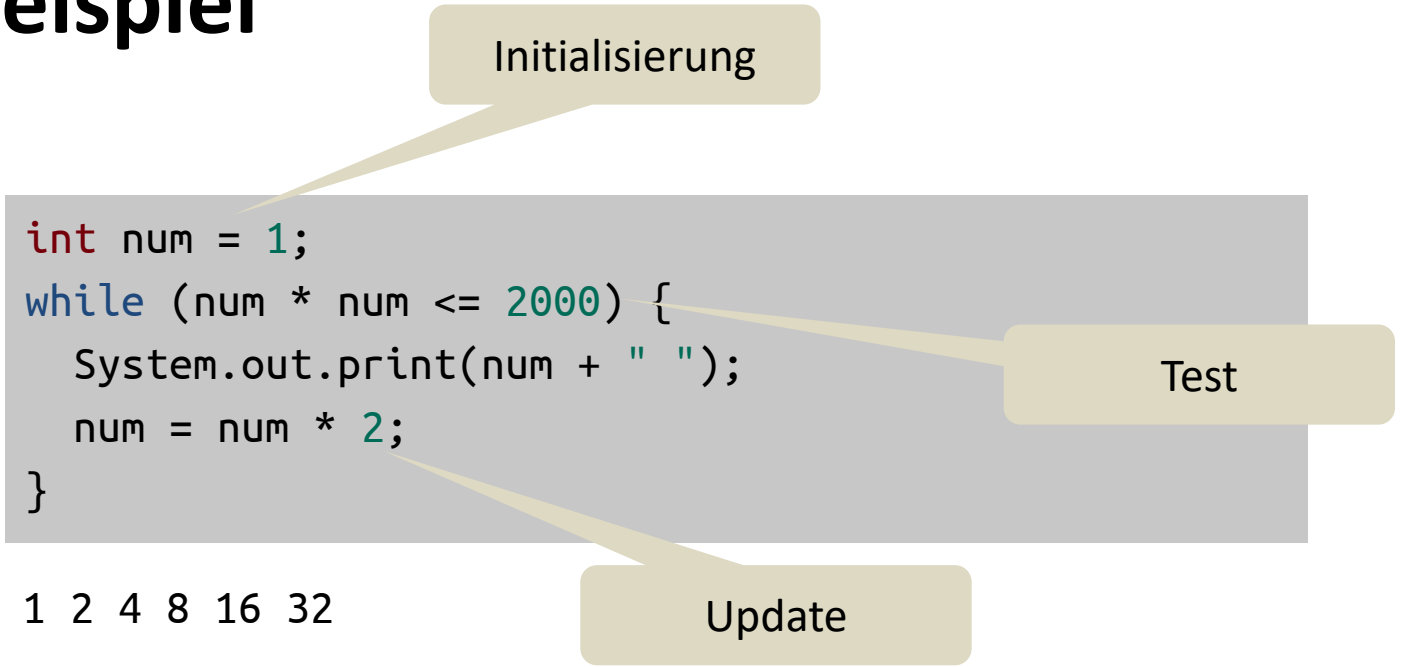
```
while (test) {
    body
}
statement;
```



while-Schleife führt Rumpf so lange aus, wie *test* den Wert `true` ergibt.

In anderen Worten: die while-Schleife bricht ab, sobald *test* den Wert `false` ergibt.

Beispiel


Initialisierung
int num = 1;
while (num * num <= 2000) {
System.out.print(num + " ");
num = num * 2;
}
Test
Update
1 2 4 8 16 32

Beispiel: erster nicht-trivialer Teiler von 91

Anzahl Iterationen unklar

```
int n = 91;
int factor = 2;
while (n % factor != 0) {
  factor = factor + 1;
}
System.out.println("first divisor is " + factor);
```

```
first divisor is 7
```

3. Kontrollstrukturen

3.3 Methoden II

3.3.1 Parameter

3.3.2 Rückgabewerte

3.3.3 Namensräume

3.3.4 Methodenüberladung

Wiederholung: Methoden

- Methode: Benannte Sequenz von Anweisungen

```
1  public class HelloWorld {
2
3      public static void main(String[] args) {
4          System.out.println("Hello World!");
5          System.out.println("Hello World again!");
6      }
7  }
```

- Diese werden beim Methodenaufruf ausgeführt

Beispiel: Versprechen

```
public static void main(String[] args) {
  System.out.print("Manuela verspricht: ");
  System.out.println("\"Ich werde die Übungen machen.\"");
  System.out.print("Malte verspricht: ");
  System.out.println("\"Ich werde die Übungen machen.\"");
}
```

Redundanz

```
Manuela verspricht: "Ich werde die Übungen machen."
Malte verspricht: "Ich werde die Übungen machen."
```

Beispiel: Versprechen mit Methode v1

```
public static void main(String[] args) {
  System.out.print("Manuela verspricht: ");
  vorsatz();
  System.out.print("Malte verspricht: ");
  vorsatz();
}

public static void vorsatz() {
  System.out.println("\"Ich werde die Übungen machen.\"");
}
```

Auslagern und wiederverwenden!

Unschön: Texte, die zusammengehören, wurden separiert.

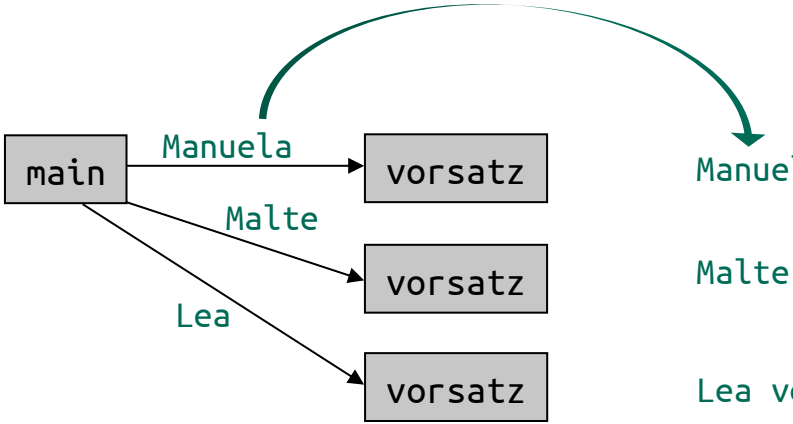
Beispiel: Versprechen mit Methoden v2

```
public static void main(String[] args) {
  vorsatzManuela();
  vorsatzMalte();
}
public static void vorsatzManuela() {
  System.out.print("Manuela verspricht: ");
  System.out.println("\"Ich werde die Übungen machen.\"");
}
public static void vorsatzMalte() {
  System.out.print("Manuela verspricht: ");
  System.out.println("\"Ich werde die Übungen machen.\"");
}
```

Redundanz

Unschön: eine Methode pro Person und viel Code-Duplikation …

Parametrisierung




```
Manuela verspricht: "Ich werde die Übungen machen."

Malte verspricht: "Ich werde die Übungen machen."

Lea verspricht: "Ich werde die Übungen machen."
```

Parametrisierung

- **Parametrisierung** einer Methode:
  Methode mit Parametern versehen
  - **Parameter**:
    ein Wert, den die aufgerufene Methode vom Aufrufer erhält
  - Bei der Deklaration:
    Wir geben an, dass die Methode einen Parameter braucht.
  - Beim Aufruf:
    Wir übergeben einen Wert für den Parameter.
  - Innerhalb der Methode:
    Parameter kann wie eine Variable verwendet werden.

Parameterdeklaration

```
public static void methodName(parameterType parameterName) {
  ...
}
```

- Bei der Deklaration der Methode wird der Parameter der Methode innerhalb der Klammern deklariert
  - *methodName*: Name der Methode
  - *parameterType*: Typ der Parameter-Variable
    - *Z.B.* `int`, `double`, `boolean`…
  - *parameterName*: Name der Parameter-Variable
  - Parameter-Variable kann in Methode wie Variable verwendet werden

Aufruf einer parametrisierten Methode

```
methodName(parameterValue);
```

- Beim Aufruf Methode wird der Wert des Parameters in den Klammern übergeben
  - *parameterValue*: Ausdruck mit Typ *parameterType*
- Wert wird in Parameter-Variable gespeichert
- Die Anweisungen der Methode werden ausgeführt mit diesem Wert für die Parameter-Variable

Beispiel: Versprechen mit Parametrisierung

Parameterdeklaration

Parameter wird wie Variable verwendet

```
public static void vorsatz(String name) {
  System.out.print(name + " verspricht: ");
  System.out.println("\"Ich werde die Übungen machen.\"");
}
```

```
public static void main(String[] args) {
  vorsatz("Manuela");
  vorsatz("Malte");
}
```

Wertübergabe bei Aufruf

```
Manuela verspricht: "Ich werde die Übungen machen."
Malte verspricht: "Ich werde die Übungen machen."
```

Formale versus tatsächliche Parameter

- **Formaler Parameter** («**formal parameter**»):
  die Parameter-Variable, die von der Methode deklariert wird
- **Tatsächlicher Parameter** («**actual parameter**»),
  auch **Argument** («**argument**»):
  der Wert, der bei Aufruf an die Methode übergeben wird
  - Kann für jeden Aufruf ein anderer Wert sein
- Beim Aufruf wird der formale Parameter mit dem Wert des tatsächlichen Parameters initialisiert

Beispiel: PIN-Wiedergabe

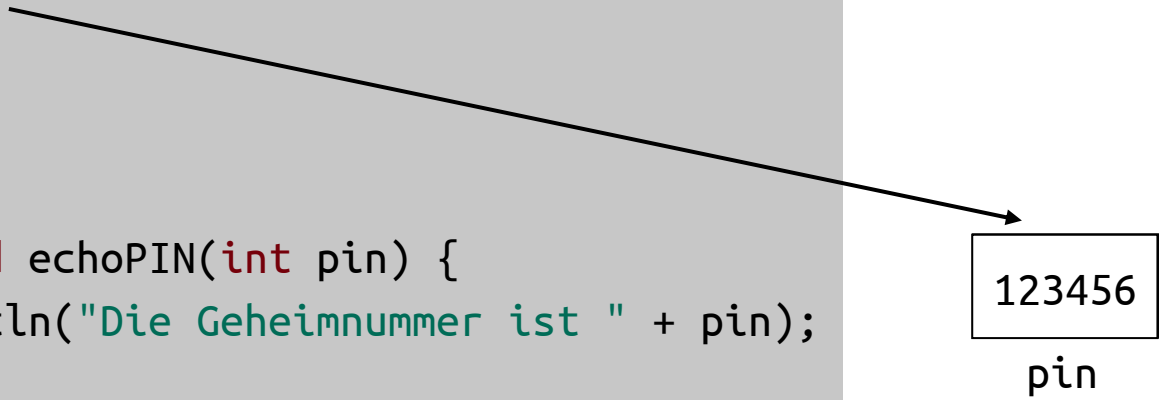
```
public static void main(String[] args) {
  echoPIN(123456);
  echoPIN(40 + 2);
}

public static void echoPIN(int pin) {
  System.out.println("Die Geheimnummer ist " + pin);
}
```

tatsächliche Parameter (Argumente)

formaler Parameter (Parameter-Variable)

Beispiel: PIN-Wiedergabe

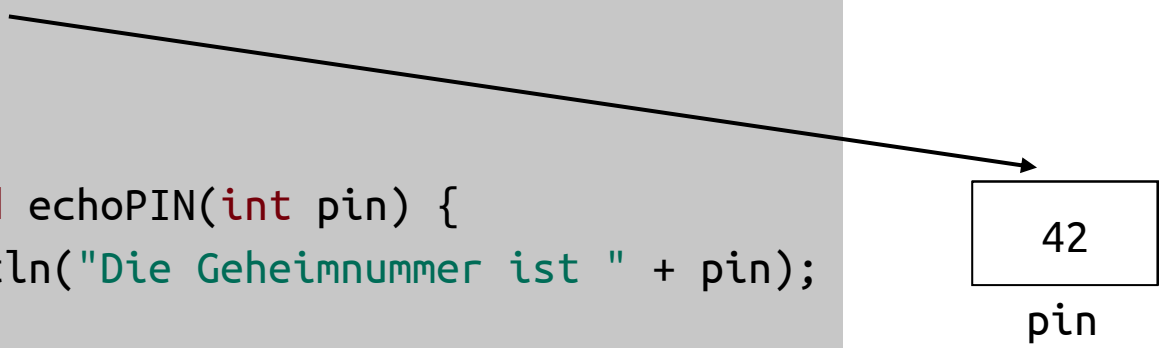
```
public static void main(String[] args) {
  echoPIN(123456);
  echoPIN(40 + 2);
}

public static void echoPIN(int pin) {
  System.out.println("Die Geheimnummer ist " + pin);
}
```




```
Die Geheimnummer ist 123456
```

Beispiel: PIN-Wiedergabe (weitergeführt)

```
public static void main(String[] args) {
  echoPIN(123456);
  echoPIN(40 + 2);
}

public static void echoPIN(int pin) {
  System.out.println("Die Geheimnummer ist " + pin);
}
```




```
...
Die Geheimnummer ist 42
```

Mögliche Fehler

- Vergessenes Argument bei Aufruf:

```
echoPIN(); // ERROR: parameter value required
```

- Falscher Typ des Arguments (Umwandlungsregeln wie bisher):

```
echoPIN(1.67); // ERROR: must be of type int
```

Mehrere Parameter

- Mehrere Parameter durch Komma getrennt
- Deklaration:

```
public static void name(type1 name1, type2 name2, …, typeN nameN) {
   ...
}
```

- Aufruf

```
name(value1, value2, …, valueN);
```

Beispiel: Summe der Zahlen zwischen Grenzen

- Berechne Summe aller Zahlen zwischen `from` und `to`

```
public static void sumFromTo(int from, int to) {
  int sum = 0;
  for (int i = from; i <= to; i = i + 1) {
    sum = sum + i;
  }
  System.out.println("the sum is " + sum);
}
```

```
sumFromTo(10, 11);
sumFromTo(1, 100);
```

```
the sum is 21
the sum is 5050
```

Was passiert hier?

```
public static void plusOne(int x) {
  x = x + 1;
}
```

```
int x = 6;
int y = 7;
plusOne(x);
System.out.println(x);
plusOne(y);
System.out.println(y);
```

6
7

Wertübergabe («Value Semantics»)

Allgemeinere Typen später

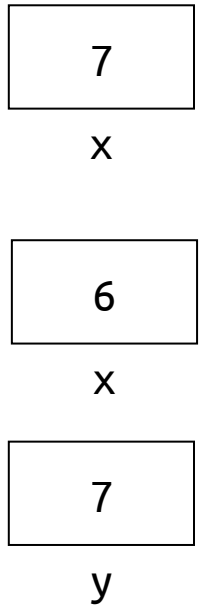
Für primitive Parametertypen (`int`, `boolean`, …):

- Aufrufer berechnet und übergibt Wert für Parameter
  - Ausdruck wird evaluiert und Ergebnis in Parameter-Variable **kopiert**
- Aufgerufene Methode erhält nur **Kopie** des Werts
  - Weiss nicht, woher der Wert kommt
  - Kennt die Variablen des Aufrufers nicht
  - Veränderungen der Parameter-Variable in der aufgerufenen Methode haben keine Auswirkungen auf die aufrufende Methode
  - Parameter-Variable ist neue Variable, auch wenn sie zufällig gleich heisst wie eine Variable in der aufrufenden Methode

Beispiel: Was passiert hier?

```
public static void plusOne(int x) {
  x = x + 1;
}
```

```
int x = 6;
int y = 7;
plusOne(x);
System.out.println(x);
plusOne(y);
System.out.println(y);
```

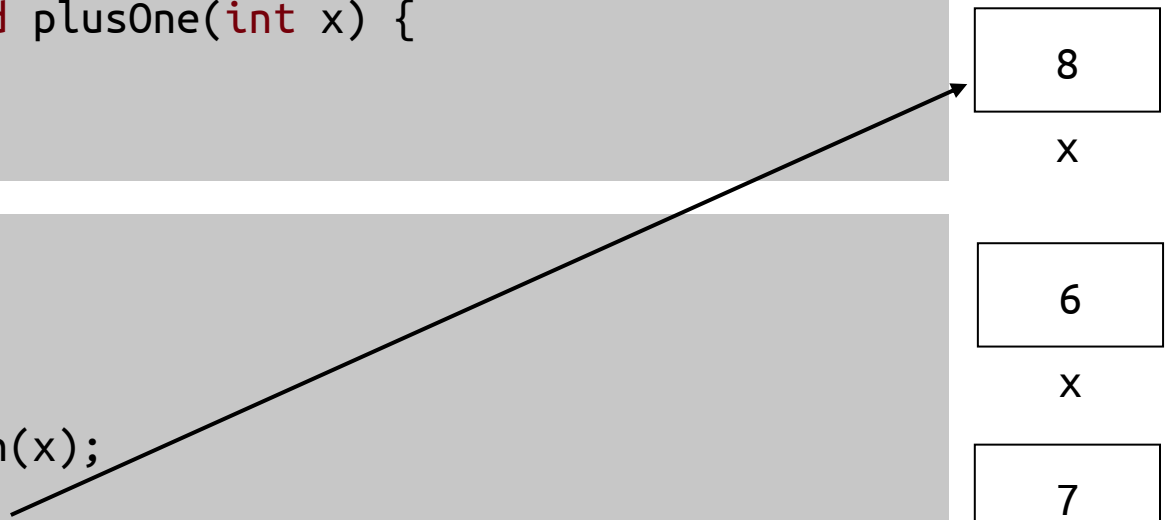


6
7

Beispiel: Was passiert hier?

```
public static void plusOne(int x) {
  x = x + 1;
}
```

```
int x = 6;
int y = 7;
plusOne(x);
System.out.println(x);
plusOne(y);
System.out.println(y);
```



6
7

Beispiel: Was passiert hier? Mit Extra-Output

```
public static void plusOne(int x) {
  x = x + 1;
  System.out.println(x);
}
```

```
int x = 6;
int y = 7;
plusOne(x);
System.out.println(x);
plusOne(y);
System.out.println(y);
```

7
6
8
7

3. Kontrollstrukturen

3.3 Methoden II

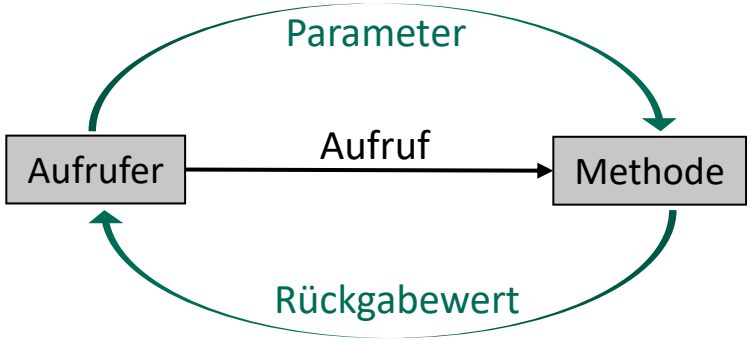
3.3.1 Parameter

3.3.2 Rückgabewerte

3.3.3 Namensräume

3.3.4 Methodenüberladung

Kommunikation zwischen Aufrufer und Methode



- Rückgabewert ist Gegenstück zu Parametern:
  - Parameter schicken Werte vom Aufrufer in die aufgerufene Methode
  - Rückgabewerte schicken Werte aus der aufgerufenen Methode zum Aufrufer

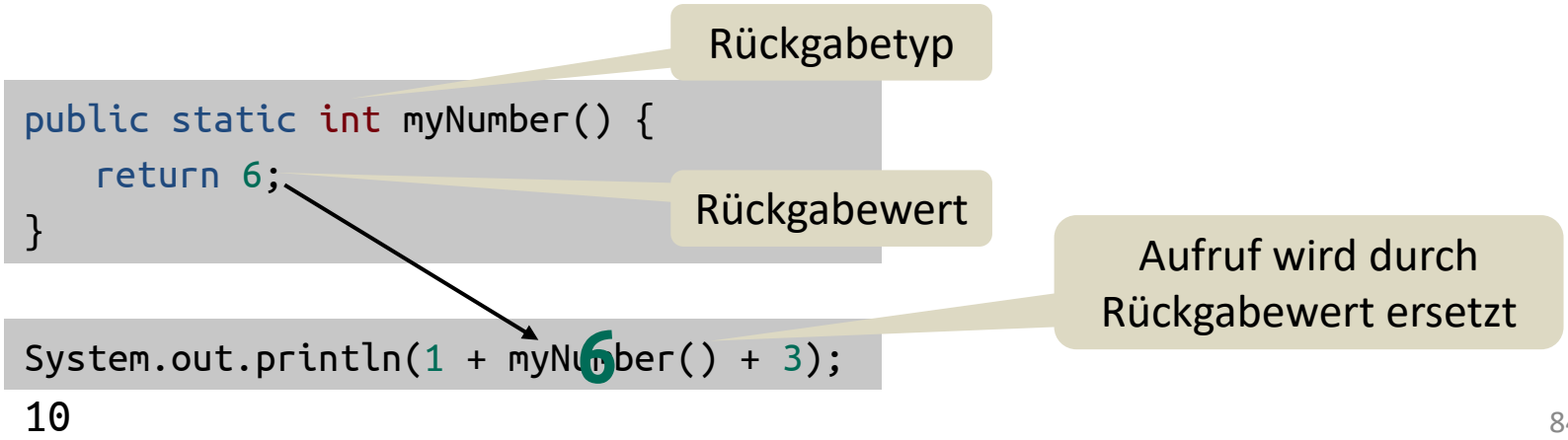
Rückgabewert («return value»)

- Kommunikation von der aufgerufenen Methode zum Aufrufer mit **Rückgabewert** («**return value**»)
  - Deklariert mit Typ vor dem Methodennamen
    - Typ `void` falls *kein* Rückgabewert
  - Wert wird zurückgegeben mittels `return`-Anweisung innerhalb der Methode

```
public static returnType name(…) {
    ...
    return expression;
}
```

Rückgabeanweisung («return statement»)

- **Rückgabeanweisung** («**return statement**») `return` *expression*`;`
  - Ausdruck *expression* wird ausgewertet
  - Wert wird an Aufrufer der Methode zurückgegeben (**Kopie** für primitive Typen)
  - Aufgerufene Methode wird beendet
  - Aufruf wird durch Rückgabewert «ersetzt»; Aufruf kann Teil eines Ausdrucks sein



Beispiel: Plus 1

Variante 1

```
public static int plusOne(int x) {
  x = x + 1;
  return x;
}
```

Variante 2

```
public static int plusOne(int x) {
  return (x + 1);
}
```

```
System.out.println(plusOne(6));
```

7

Beispiel: Fahrenheit zu Celsius

```java
public static double fahrenheitToCelsius(double fahrenheit) {
  return 5.0 / 9.0 * (fahrenheit - 32.0);
}
```

Variante 1

```java
public static double fahrenheitToCelsius(double fahrenheit) {
  double celsius = 5.0 / 9.0 * (fahrenheit - 32.0);
  return celsius;
}
```

Variante 2

Beobachtung: Methode kann beliebig viele Variablen verwenden; sie sind aber nicht sichtbar und belanglos für den Aufrufer, der nur eine Kopie des Rückgabewerts erhält.

Beispiel: Absolutwert

```
public static int abs(int x) {
  if (x < 0) {
    x = -x;
  }
  return x;
}
```

```
int x = -6;
System.out.println("absolute value of " + x + ": " + abs(x));
```

```
absolute value of -6: 6
```

Beispiel: Primzahltest

```
public static boolean isPrime(int n) {
  boolean foundDivisor = false;
  int d = 2;
  while (!foundDivisor) {
    if (n % d == 0) {
      foundDivisor = true;
    } else {
      d = d + 1;
    }
  } // d is smallest divisor of n that is >= 2
  return d == n; // prime if smallest divisor >= 2 is n itself
}
```

Nicht-verwendeter Rückgabewert

- Rückgabewerte können ignoriert werden
  - Keine Warnung oder Fehlermeldung vom Compiler!

```
public static int squareNumber(int x) {
  return x * x;
}
```

```
squareNumber(3); // return value 9 is computed but not used
```

- Wieso ist das erlaubt?
  - Im Moment nicht sinnvoll
  - Potentiell interessant später, wenn Methoden Seiteneffekte haben (können)

Was passiert hier?

```
public static void main(String[] args) {
  squareNumber(3);
  System.out.println("3^2 = " + result);
}

public static int squareNumber(int x) {
  int result = x * x;
  return result;
}
```

Lokale Variable in Methode nicht sichtbar von aussen!

```
// ERROR: result not defined
```

Rückgabeanweisung ohne Rückgabewert

- Methoden ohne Rückgabewert können trotzdem ein `return;` enthalten
  - Können, müssen aber nicht
  - Beendet den Aufruf und schickt *keinen* Wert zurück

```
public static void nonSense() {
  System.out.println("This is reached.");
  return;
  System.out.println("This is not reached.");
}
```

```
nonSense();
```

```
This is reached.
```

Mehrere Rückgabeanweisungen

- Methoden ohne Rückgabewert müssen kein `return` haben, dürfen aber
  - Vorzeitiges Verlassen der Methode
- Methoden mit Rückgabewert müssen mindestens ein `return` haben
  - Sonst gibt es einen Compiler-Fehler
- Mehr als ein `return` ist erlaubt und manchmal auch sinnvoll
  - insbesondere für Fallunterscheidungen (in Kombination mit `if else`)

Beispiel: Maximum

```java
public static int max(int a, int b) {
  int result;
  if (a >= b) {
    result = a;
  } else {
    result = b;
  }
  return result;
}
```

Variante 1

```java
public static int max(int a, int b) {
  if (a >= b) {
    return a;
  } else {
    return b;
  }
}
```

Variante 2

Was passiert hier? v1

```
public static void main(String[] args) {
  System.out.println("The maximum of 6 and 7 is " + max(6, 7));
}

public static int max(int a, int b) {
  if (a > b) {
    return a;
  } else if (b > a) {
    return b;
  }
}
```

Kein `return` wenn `a == b`

```
// ERROR: missing return statement
```

Alle Ausführungspfade durch eine Methode müssen ein `return` enthalten!

Was passiert hier? v2

```
public static void main(String[] args) {
  System.out.println("The maximum of 6 and 7 is " + max(6, 7));
}

public static int max(int a, int b) {
  if (a >= b) {
    return a;
  } else if (b > a) {
    return b;
  }
}
```

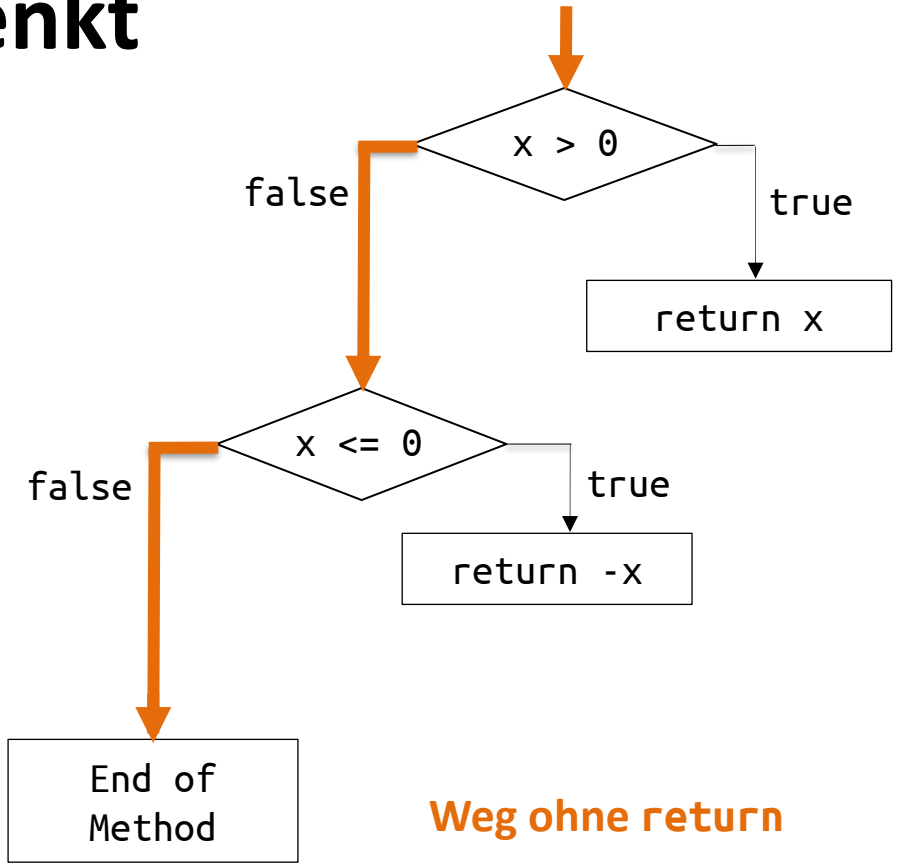
```
// ERROR: missing return statement
```

Der Compiler meint, dass es einen Weg gibt ohne `return` (das implizite `else`).

Wie der Compiler denkt

```
public static int silly(int x) {
  if (x > 0) {
    return x;
  } else {
    if (x <= 0) {
      return -x;
    }
  }
}
```

```
// ERROR: missing return
```



Weg ohne return

3. Kontrollstrukturen

3.3 Methoden II

Sichtbarkeit von Namen

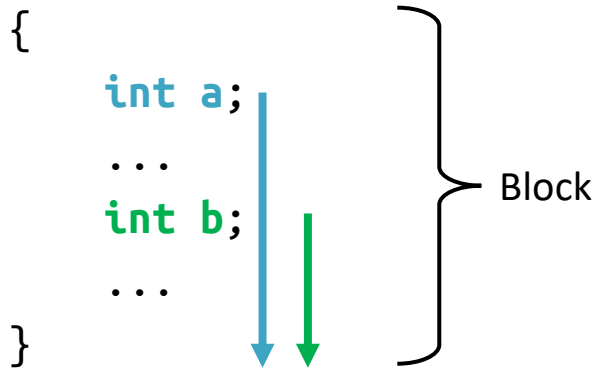
Scope (**Namensraum**):
Teil des Programms, in dem ein Name/Symbol sichtbar ist

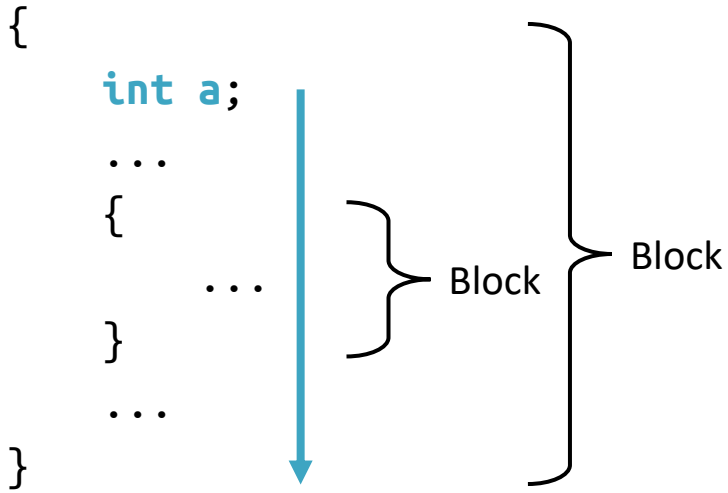
- Variable kann nur innerhalb ihres Scopes gelesen/modifiziert werden
- Methode kann nur innerhalb ihres Scopes aufgerufen werden
  - Später mehr zu Sichtbarkeit von Methoden
- Ein paar Regeln als Approximation hier; weitere Aspekte folgen später

Scope von Variablen

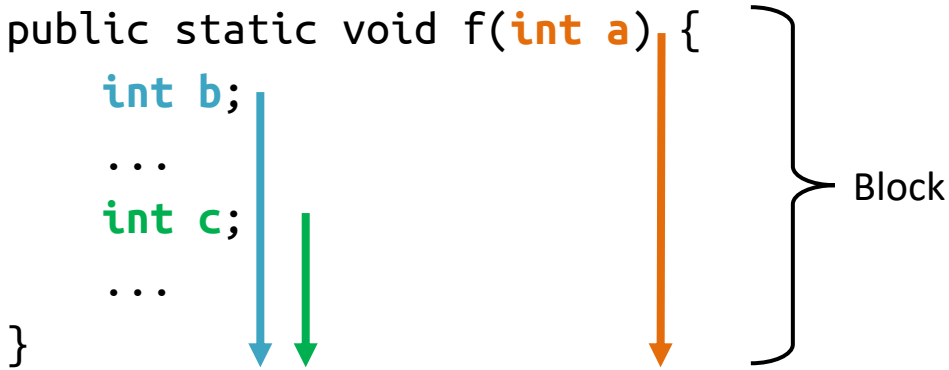
Von der Deklaration bis zum Ende des aktuellen Blocks

- Block durch { } begrenzt
- Inklusive aller eingeschlossenen Blöcke bei verschachtelten Blöcken



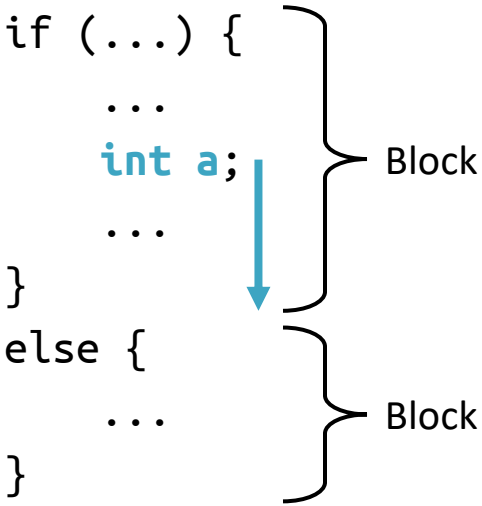


Scope von Variablen: In Methoden



Regel:
Eine Variable, die in einer Methode deklariert wurde,
existiert nur in der Methode. Das gilt auch für Parameter-Variablen.

Scope von Variablen: In Verzweigungen



Regel:
Eine Variable, die in einem `if/else`-Block deklariert wurde, existiert nur in diesem Block.

Scope von Variablen: In Schleifen-Kopf

```
for (int i = 0; ...; ...) {
    ...
}
```

Regel:
Eine Variable, die im Kopf einer for-Schleife deklariert wurde, existiert nur in dieser Schleife (Kopf und Rumpf).

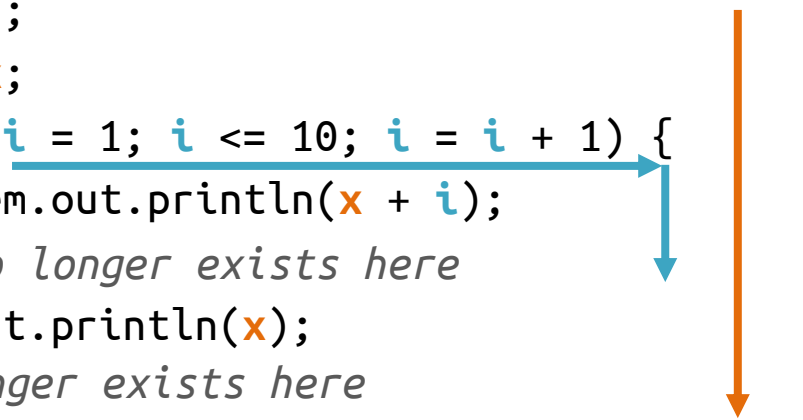
Scope von Variablen: In Schleifen-Rumpf

```
for (...; ...; ...) {
    ...
    int a;
    ...
}
```

Regel:
Eine Variable, die im Rumpf einer for-Schleife deklariert wurde,
existiert nur in einer Iteration des Schleifen-Rumpfs.
In der nächsten Iteration wird eine neue Variable mit gleichem Namen angelegt.

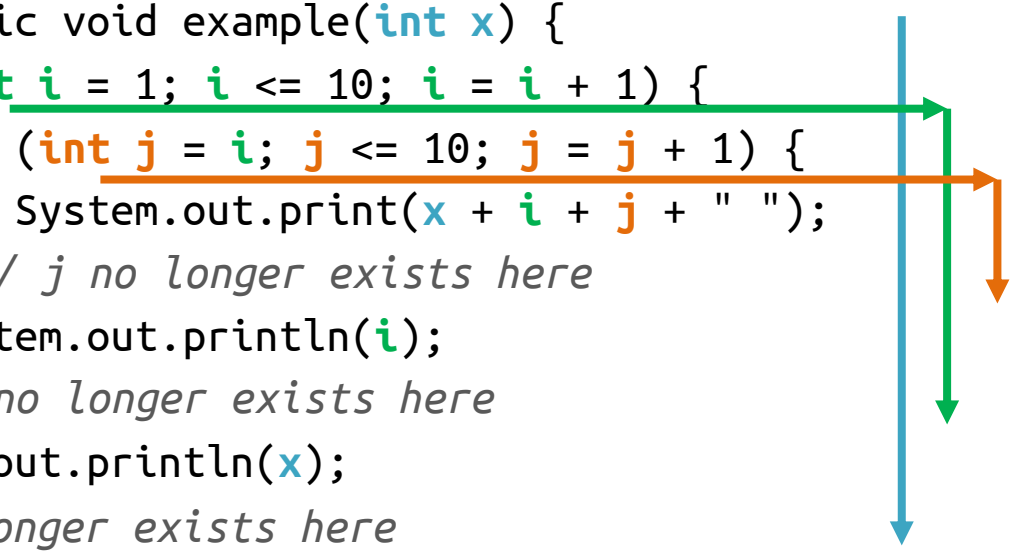
Beispiel 1

```
public static void example() {
    int x = 3;
    x = x * x;
    for (int i = 1; i <= 10; i = i + 1) {
        System.out.println(x + i);
    } // i no longer exists here
    System.out.println(x);
} // x no longer exists here
System.out.println(x); // ERROR: outside scope
```



Beispiel 2

```
public static void example(int x) {
    for (int i = 1; i <= 10; i = i + 1) {
        for (int j = i; j <= 10; j = j + 1) {
            System.out.print(x + i + j + " ");
        } // j no longer exists here
        System.out.println(i);
    } // i no longer exists here
    System.out.println(x);
} // x no longer exists here
```



Eindeutige Deklaration

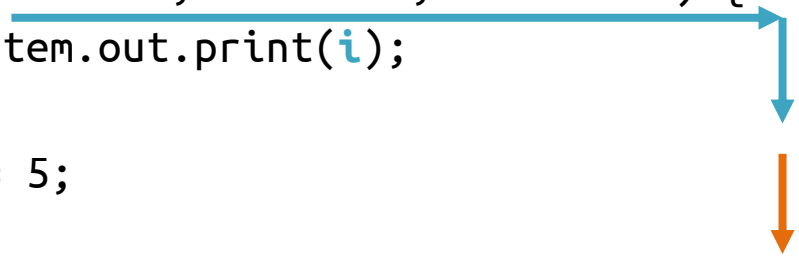
```
for (int i = 1; i <= 100; i = i + 1) {
  int i = 2; // ERROR
  System.out.print(i);
}
```

```
for (int i = 1; i <= 100; i = i + 1) {
  for (int i = 1; i <= 100; i = i + 1) { // ERROR
    System.out.print(i);
  }
}
```

Regel:
Scopes von Variablen mit dem gleichen Namen dürfen sich nicht überschneiden.

Mehrere Variablen mit gleichem Namen

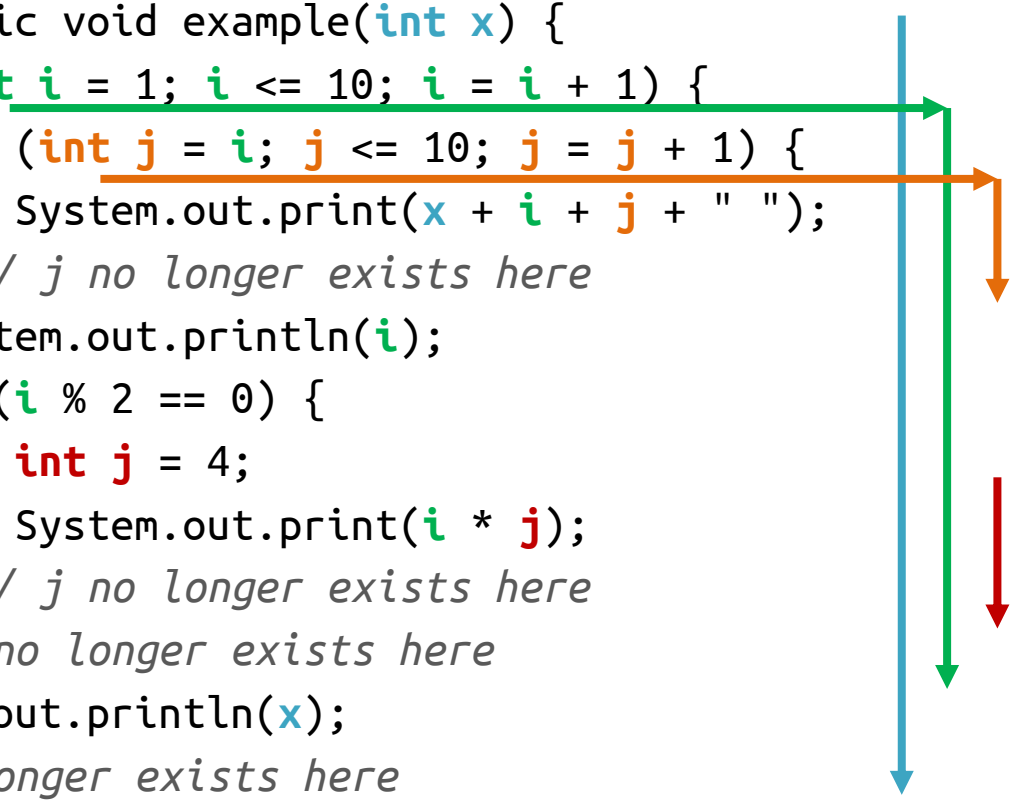
```
for (int i = 1; i <= 100; i = i + 1) {
    System.out.print(i);
}
int i = 5;
...
```



Variablen mit nicht-überlappenden Scopes dürfen den gleichen Namen haben (zwei verschiedene Variablen mit zufällig dem gleichen Namen).

Beispiel

```
public static void example(int x) {
    for (int i = 1; i <= 10; i = i + 1) {
        for (int j = i; j <= 10; j = j + 1) {
            System.out.print(x + i + j + " ");
        } // j no longer exists here
        System.out.println(i);
        if (i % 2 == 0) {
            int j = 4;
            System.out.print(i * j);
        } // j no longer exists here
    } // i no longer exists here
    System.out.println(x);
} // x no longer exists here
```



Warum Scopes?

- Lesbarkeit der Programme
  - Z.B. durch Wiederverwendung der gleichen Variablennamen an unterschiedlichen Stellen im Programm bzw. in unterschiedlichen Kontexten

- Vereinfachung der Speicherverwaltung
  - Nicht weiter relevant für EProg

3. Kontrollstrukturen

3.3 Methoden II

3.3.1 Parameter

3.3.2 Rückgabewerte

3.3.3 Namensräume

3.3.4 Methodenüberladung

Methodenüberladung

- **Signatur** einer Methode: Name und Liste der Parametertypen

```
public static returnType methodName(type1, type2, …, typeN)
```

- Mehrere Methoden mit der gleichen Signatur nicht erlaubt!
  - Compilerfehler: *`Method is already defined`*
- Mehrere Methoden mit dem gleichen Namen erlaubt, falls sie unterschiedliche Parametertypen (und/oder -anzahl) haben.
  - Parameternamen und Rückgabetyp sind irrelevant
  - Nennt sich **Methodenüberladung** («**Method Overloading**»)
    - «passendste» Methode wird ausgewählt bei Aufruf

Beispiel 1: Maximum

```
public static int max(int a, int b) {
  return a >= b ? a : b;
}
```

```
public static boolean max(boolean a, boolean b) {
  return a || b;
}
```

```
max(3, 2);          // returns 3, calling first method
max(true, false);   // returns true, calling second method
```

Beispiel 2: Summe

```java
public static int sum(int from, int to) {
  int sum = 0;
  for (int i = from; i <= to; i = i + 1) {
    sum = sum + i;
  }
  return sum;
}
```

```java
public static int sum(int to) {
  return sum(1, to);
}
```

```java
sum(2, 3);   // returns 5, calling first method
sum(3);      // returns 6, calling second method
```

Was nicht erlaubt ist…

- Methoden mit nur unterschiedlichen Parameternamen

```
public static int f(int a) {
  ...
}
```

```
public static int f(int b) {
  ...
}
```

- Methoden mit nur unterschiedlichen Rückgabetypen

```
public static int f(int a) {
  ...
}
```

```
public static void f(int a) {
  ...
}
```

- Oder Kombination davon

Was passiert hier?

```java
public static String foo(int a, double b) {
  return "int double";
}
```

```java
public static String foo(double a, int b) {
  return "double int";
}
```

```java
foo(1, 1.0);        // returns "int double"
foo(1.0, 1);        // returns "double int"
foo(1.0, 1.0);      // ERROR: no matching method
foo(1, 1);          // ERROR: both methods match (ambiguous)
```

Regel: Es braucht eine eindeutig beste Methode, sonst Compilerfehler!

3. Kontrollstrukturen

3.4 Schleifen II

3.4.1 Inkrement und Dekrement

3.4.2 Typische Fehler

3.4.3 Benutzergesteuerte Schleifen

3.4.4 do-while-Schleife

Häufiges Szenario bei for-Schleifen

- Schleifenvariable wird um eins vergrössert oder verkleinert

```
for (int i = start; i < bound; i = i + 1) {
  ...
}
```

```
for (int i = start; i > bound; i = i - 1) {
  ...
}
```

- Kurzform für Veränderung um 1
  - **Post-Inkrement** («**post-increment**»): ***variable++*** für ***variable = variable + 1***
  - **Post-Dekrement** («**post-decrement**»): ***variable--*** für ***variable = variable - 1***

Beispiele

```
for (int i = start; i < bound; i++) {
  ...
}
```

```
for (int i = start; i > bound; i--) {
  ...
}
```

```
int x = 2;
x++; // x has value 3 now
```

```
double y = 2.4;
y--; // y has value 1.4 now
```

Wieso Post-Inkrement und -Dekrement?

Variable wird (insbesondere in Ausdrücken)

- zuerst verwendet (gelesen) …
- … und erst danach verändert (inkrementiert/dekrementiert)!

```
int x = 2;
System.out.println(x++);
```

```
int x = 2;
int xOld = x; // read first
x = x + 1; // then increment
System.out.println(xOld);
```

Post-Inkrement in Ausdrücken

```
int x = 2;
System.out.println(x++); // output 2; x gets value 3
System.out.println(x++); // output 3; x gets value 4
```

```
int x = 2;
int y = x++;             // x gets value 3; y gets value 2
System.out.println(x);   // output 3
System.out.println(y);   // output 2
```

Zeile verändert zwei Variablen!

Erinnerung: Ablauf einer Zuweisung

1. Rechte Seite (ein Ausdruck) wird ausgewertet
2. Resultat (Wert) wird in Variable auf linker Seite (Name einer Variablen) gespeichert

Beispiele

```
int a = 1;
a = a++;
```

```
int a = 1;
int aOld = a;       // 1
a = a + 1;          // 2
a = aOld;           // 1
```

```
int b = 10;
int c = b-- - b--;
```

```
int b = 10;
int bOld1 = b;      // 10
b = b - 1;          // 9
int bOld2 = b;      // 9
b = b - 1;          // 8
c = bOld1 - bOld2;  // 1
```

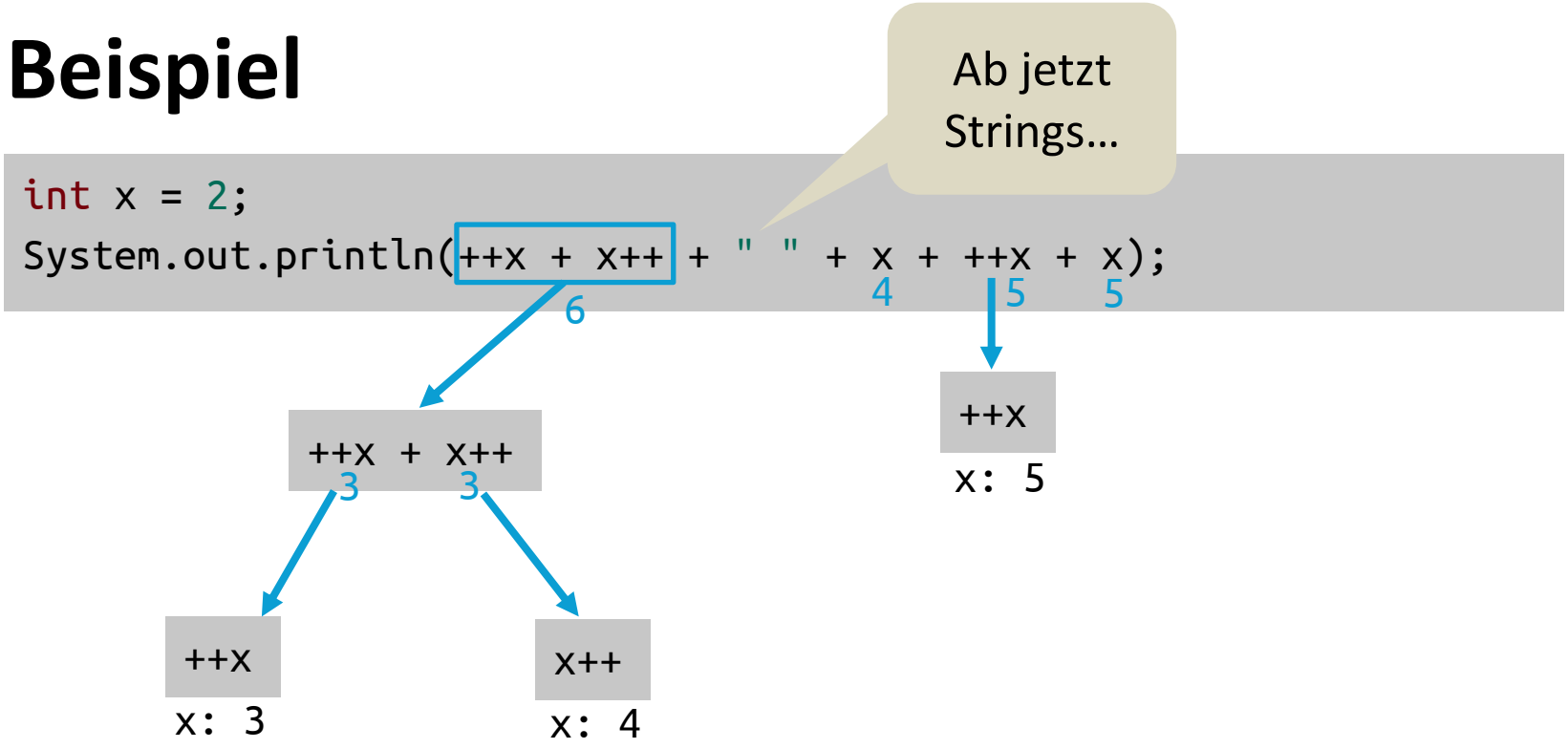
Weitere Kurzformen

- **Prä-Inkrement («pre-increment»)** und **Prä-Dekrement («pre-decrement»)**: zuerst ändern, dann verwenden
  - `++variable` für `variable = variable + 1`
  - `--variable` für `variable = variable - 1`
- Veränderung mit beliebigen Wert (statt nur ± 1)
  - `variable += value` für `variable = variable + value;`
  - `variable -= value` für `variable = variable - value;`
  - `variable *= value` für `variable = variable * value;`
  - `variable /= value` für `variable = variable / value;`
  - `variable %= value` für `variable = variable % value;`

Achtung:

```
x += 3
x =+ 3
```

Beispiel



Output: 6 455

Was ist der Wert von x?

```
boolean p = ...;
int x = 2;
boolean q = p && x++ < 3;
```

Kurzschlussauswertung führt zu bedingter Ausführung von Seiteneffekt x++

```
boolean p = true;
int x = 2;
boolean q = p && x++ < 3;
```

```
p    && x++ < 3
true && x++ < 3
true && 2   < 3   // x is 3
```

x ist 3

```
boolean p = false;
int x = 2;
boolean q = p && x++ < 3;
```

```
p     && x++ < 3
false && x++ < 3
false && x++ < 3   // x stays 2
```

x ist 2

Einschub: Seiteneffekte

Stark vereinfacht: Es gibt 2 Arten von Code-Komponenten

- Code-Komponenten, die einen Wert haben (Literale, Ausdrücke)
- Code-Komponenten, die einen Effekt haben (Anweisungen)
  - Effekt haben: den Zustand des Programms verändern

- Ausdrücke mit Seiteneffekten verändern den Zustand des Programms
  - Inkrement/Dekrement: `x++`, `x--`
  - Operationen, die Fehlermeldungen auslösen
- Es gibt auch Methoden mit Seiteneffekten
  - Lernen wir später kennen

Kurzformen: Warnung und Empfehlungen

- **Kurzformen sind nett, aber gefährlich**
  - Vorsicht insbesondere bei Kurzschlussauswertung
  - Gebrauch erlaubt, nicht erzwungen
- **Unser Ziel ist es, verständliche Programme zu schreiben**
  - Lesen des Programms soll einfach und kein Puzzle sein!
  - Trotzdem sollten Sie Inkrement und Dekrement erkennen

- **Unsere Empfehlung:**
  - Kurzform nicht in Ausdrücken verwenden, sondern nur eigenständig als Anweisung
  - Inkrement/Dekrement vor allem für Schleifen

3. Kontrollstrukturen

3.4 Schleifen II

Eine triviale (?) Aufgabe

Schreiben Sie eine Methode `printNumbers(int n)`, die alle Zahlen von `1` bis `n` (für den Parameter `n`) durch Kommas getrennt ausgibt.

Parameter: `n = 6`
Ausgabe: `1, 2, 3, 4, 5, 6`

Parameter: `n = 7`
Ausgabe: `1, 2, 3, 4, 5, 6, 7`

Welche Schleife(n) ist/sind korrekt?

```
for (int i = 1; i <= n; i++) {
  System.out.print(", " + i + ", ");
}
```

A

Output für n = 5:

```
, 1, , 2, , 3, , 4, , 5,
```

```
for (int i = 1; i <= n; i++) {
  System.out.print(i + ", ");
}
```

B

Output für n = 5:

```
1, 2, 3, 4, 5,
```

```
for (int i = 1; i <= n; i++) {
  System.out.print(", " + i);
}
```

C

Output für n = 5:

```
, 1, 2, 3, 4, 5
```

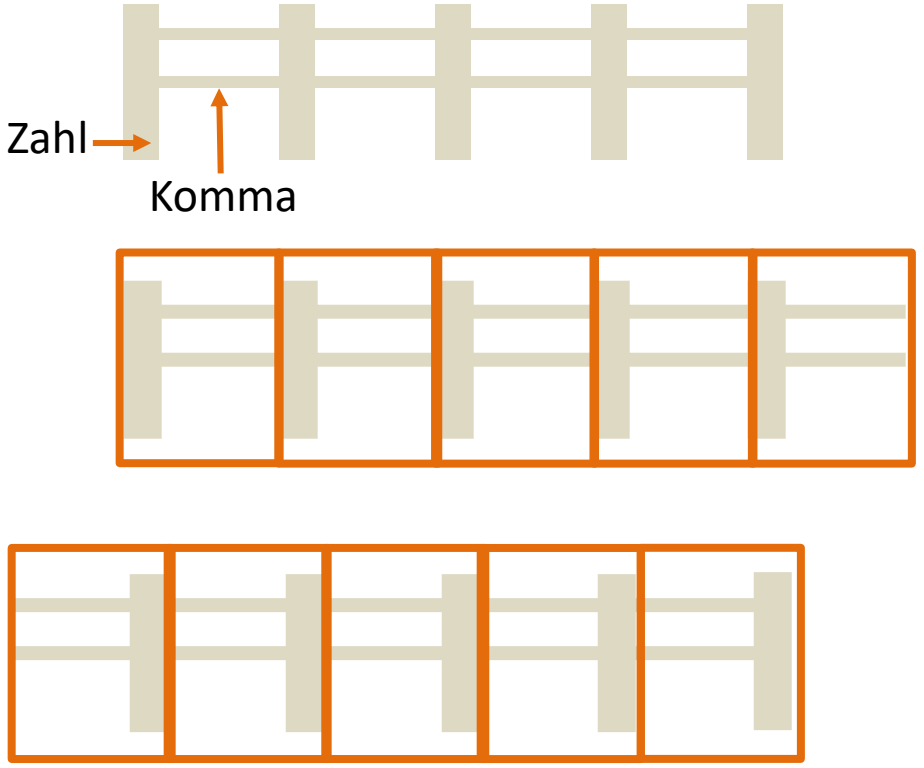
siehe Wikipedia

Zaunpfahlfehler: Analogie zu Gartenzaun

- n Pfosten (Pfähle)
- n-1 Latten (Querstreben)

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

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

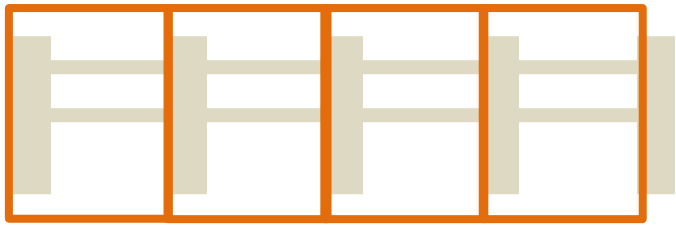


siehe Wikipedia

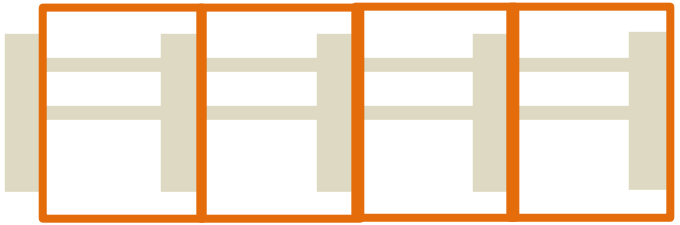
Lösung fürs Zaunpfahlproblem

Einen Pfosten (ersten oder letzten) ausserhalb der Schleife hinzufügen!

```
for (int i = 1; i <= n - 1; i++) {
  pfosten();
  latte();
}
pfosten();
```



```
pfosten();
for (int i = 1; i <= n - 1; i++) {
  latte();
  pfosten();
}
```



Triviale (?) Aufgabe: Lösungen

```java
public static void printNumbers(int n) {
  for (int i = 1; i <= n - 1; i++) {
    System.out.print(i + ", ");
  }
  System.out.print(n);
}
```

Für `n < 1` haben wir keine Iteration

```java
public static void printNumbers(int n) {
  System.out.print(1);
  for (int i = 2; i <= n; i++) {
    System.out.print(", " + i);
  }
}
```

Für `n < 2` haben wir keine Iteration

Typischer Fehler

- Off-by-one (um eins daneben)
  - Schleife wird einmal zu viel oder zu wenig oft durchlaufen
  - Indexierproblem: Schleifenvariable ist um eins daneben
  - Zaunpfahlfehler

Unser Tipp

- kleines Beispiel mit 2-3 Iterationen durchspielen
- Spezialfälle betrachten:
  - Was passiert bei `n = 1` oder `n = 0`?

Eine etwas (?) schwierigere Aufgabe

Schreiben Sie eine Methode `printPrimes(int n)`, die alle Primzahlen bis zur Obergrenze `n` (für den Parameter `n`) durch Komma getrennt ausgibt.

Parameter: `n = 20`
Ausgabe: `2, 3, 5, 7, 11, 13, 17, 19`

Lösung

```
public static void printPrimes(int n) {
  if (n < 2) {
    return;
  }
  System.out.print(2);
  for (int i = 3; i <= n; i++) {
    if (isPrime(i)) {
      System.out.print(", " + i);
    }
  }
}
```

Für n < 2 wollen wir nichts ausgeben!

```
public static boolean isPrime(int n) {
  int numberOfDivisors = 0;
  for (int d = 1; d <= n; d++) {
    if (n % d == 0) {
      numberOfDivisors++;
    }
  }
  return numberOfDivisors == 2;
}
```

Korrekt auch für n < 2

3. Kontrollstrukturen

3.4 Schleifen II

Eingabewerte zur Schleifenkontrolle

- Eingaben können nicht nur zur Berechnung verwendet werden, sondern können auch (die Terminierung von) Schleifen kontrollieren
- Interessantes Beispiel einer unbestimmten Schleife
  - Schleife wird ausgeführt, bis ein Sentinel gesehen wird
  - oft auch Sentinel-Schleife genannt

- **Sentinel**: ein Zeichen, das ein Hinweis auf etwas (z.B. das Ende) ist

Beispiel: Einfacher Taschenrechner

Schreiben Sie ein Programm, das Zahlen einliest, bis der Benutzer die Zahl 0 eintippt. Danach soll die Summe aller eingetippten Zahlen ausgegeben werden.

- 0 ist das Sentinel (Ende der Eingabesequenz)

```
Enter a number (0 to quit): 10
Enter a number (0 to quit): 20
Enter a number (0 to quit): 30
Enter a number (0 to quit): 0
The total is 60
```

Erster Versuch

```
Scanner console = new Scanner(System.in);
int sum = 0;
int number;

while (number != 0) {
  System.out.print("Enter a number (0 to quit): ");
  number = console.nextInt();
  sum = sum + number;
}

System.out.print("The total is " + sum);
```

Fehler: Variable nicht initialisiert!

Zweiter Versuch

```
Scanner console = new Scanner(System.in);
int sum = 0;
int number = -1; // dummy value; anything but 0

while (number != 0) {
  System.out.print("Enter a number (0 to quit): ");
  number = console.nextInt();
  sum = sum + number;
}

System.out.print("The total is " + sum);
```

Korrekt, aber schlechter Stil! Was wenn -1 zum Sentinel wird?

-1 als Sentinel: 1. Versuch

```
Scanner console = new Scanner(System.in);
int sum = 0;
int number = -2; // dummy value; anything but -1

while (number != -1) {
  System.out.print("Enter a number (-1 to quit): ");
  number = console.nextInt();
  sum = sum + number;
}

System.out.print("The total is " + sum);
```

-1 als Sentinel: möglicher Verlauf

```
Enter a number (-1 to quit): 10
Enter a number (-1 to quit): 20
Enter a number (-1 to quit): 30
Enter a number (-1 to quit): -1
The total is 59
```

Problem: Sentinel wird zum Total addiert

- Wenn Sentinel 0 ist, hat das keinen Effekt, nun aber schon
- ➢ Code sollte möglichst unabhängig von der Wahl des Sentinels funktionieren

-1 als Sentinel: 1. unschöne Lösung

```
Scanner console = new Scanner(System.in);
int sum = 1; // to compensate sentinel of -1
int number = -2; // dummy value; anything but -1

while (number != -1) {
  System.out.print("Enter a number (-1 to quit): ");
  number = console.nextInt();
  sum = sum + number;
}

System.out.print("The total is " + sum);
```

Sentinel kompensieren

-1 als Sentinel: 2. unschöne Lösung

```
Scanner console = new Scanner(System.in);
int sum = 0;
int number = -2; // dummy value; anything but -1

while (number != -1) {
  System.out.print("Enter a number (-1 to quit): ");
  number = console.nextInt();
  if (number != -1) {
    sum = sum + number;
  }
}
System.out.print("The total is " + sum);
```

-1 als Sentinel: besserer Ansatz

Wieder ein Zaunpfahlproblem

- Wir müssen `n` Zahlen lesen, aber nur `n - 1` Zahlen addieren
- Ein Einlesen aus der Schleife rausziehen (erster Pfosten), in Schleife Zahl addieren (Querstrebe), dann neue Zahl lesen (Pfosten)

```
sum = 0
print prompt and read input
while input is not sentinel:
  add input to sum
  print prompt and read input
print sum
```

-1 als Sentinel: Lösung

```
int sum = 0;
Scanner console = new Scanner(System.in);
System.out.print("Enter a number (-1 to quit): ");
int number = console.nextInt();
while (number != -1) {
  sum = sum + number;
  System.out.print("Enter a number (-1 to quit): ");
  number = console.nextInt();
}

System.out.print("The total is " + sum);
```

Redundanz

-1 als Sentinel: Lösung mit Methode

```
int sum = 0;
int number = readNumber();
while (number != -1) {
  sum = sum + number;
  number = readNumber();
}
System.out.print("The total is " + sum);
```

```
public static int readNumber() {
  Scanner console = new Scanner(System.in);
  System.out.print("Enter a number (-1 to quit): ");
  return console.nextInt();
}
```

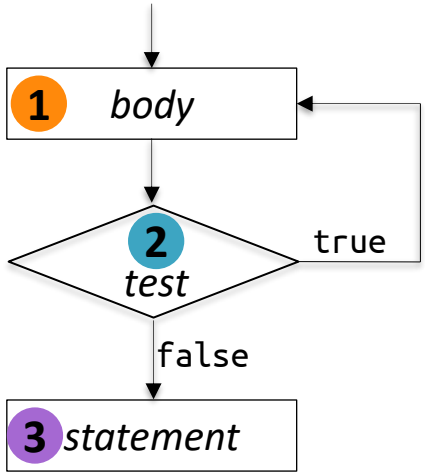
Könnte mit break-Anweisung elegant gelöst werden; diese wird hier aber nicht diskutiert.

3. Kontrollstrukturen

3.4 Schleifen II

do-while-Schleife («do-while loop»)

```
do {
    body 1
} while (test); 2
statement; 3
```



do-while-Schleife führt Test erst am Ende des Schleifenrumpfs aus, um zu entscheiden, ob eine weitere Iteration nötig ist.

Immer mindestens eine Iteration!

-1 als Sentinel: Lösung mit do-while

```
Scanner console = new Scanner(System.in);
int sum = 0;
int number;
do {
  readNumber();
  if (number != -1) {
    sum = sum + number;
  }
} while (number != -1);
System.out.print("The total is " + sum);
```

Könnte mit break-Anweisung elegant gelöst werden; diese wird hier aber nicht diskutiert.

Taschenrechner mit 3 Schleifen-Typen

Wir tippen n Zahlen, letzte davon ist `-1`:

- n-mal einlesen
- (n-1)-mal addieren

for

```
readNumber();
for ( ;number != -1; ) {
  sum = sum + number;
  readNumber();
}
```

n - 1 Iterationen,
n-mal einlesen

while

```
readNumber();
while (number != -1) {
  sum = sum + number;
  readNumber();
}
```

do-while

```
do {
  readNumber();
  if (number != -1) {
    sum = sum + number;
  }
} while (number != -1);
```

n Iterationen,
n-mal einlesen

Zaunpfahlproblem mit 3 Schleifen-Typen

- n Pfosten (Pfähle), n-1 Latten (Querstreben)

for

```
pfosten();
for (i=2; i<=n; i++) {
  latte();
  pfosten();
}
```

while

```
pfosten();
i = 2;
while (i <= n) {
  latte();
  pfosten();
  i++;
}
```

do-while

```
i = 1;
do {
  pfosten();
  latte();
  i++;
} while (i < n);
pfosten();
```

n - 1 Iterationen

do-while

```
i = 1;
do {
  pfosten();
  if (i < n) {
    latte();
  }
  i++;
} while (i <= n);
```

n Iterationen

Überblick: Schleifen

```
for (init; test; update) {
 body
}
```

for-Schleife

```
while (test) {
 body
}
```

while-Schleife

```
do {
 body
} while (test);
```

do-while-Schleife

- Die 3 Schleifen-Arten sind äquivalent (gleich ausdrucksstark)
  - Übung: Zeigen Sie, dass die 3 Schleifen äquivalent sind.
- Je nach Kontext ist eine Schleife besser (sinnvoller) als eine andere.