

# Algorithmen und Datenstrukturen

## 3. Grundlagen der Programmierung (Teil 2): Ablaufstrukturen, Methoden

Prof. Dr.-Ing. Marc Stamminger



# Worum geht es in dieser Lehreinheit?

- In der vergangenen Lehreinheit haben wir uns damit auseinander gesetzt, wie Informationen, die wir für einen Algorithmus benötigen, in Variablen unterschiedlicher Typen von Daten gespeichert werden können.
- In dieser Lehreinheit geht es nun um die Steuerung des Ablaufs, in der ein Algorithmus seine Schritte z. B. nacheinander oder – abhängig von der Auswertung von Bedingungen – alternativ oder wiederholt ausführt. Die algorithmischen Mittel dafür sind sog. Ablauf- oder Kontrollstrukturen.
- Einen weiteren Schwerpunkt bilden Methoden (sog. Unterprogramme.) Damit können Programmteile aus dem Hauptprogramm herausgenommen, mit einem Namen versehen und zur mehrfachen Verwendung durch Aufruf der Methode bereitgestellt werden. Die Verwendung solcher Unterprogramme macht Programme i. d. R. sehr viel übersichtlicher.

# Lernziele: Was sollen Sie am Ende dieser Lehreinheit können?

- den Wert von Variablen bei und nach Ausführung eines Programms zu gegebenen Eingaben bestimmen können
- ein gegebenes Java-Programm gemäß einer Aufgabenstellung ändern können
- zu einer gegebenen Problemstellung einen Algorithmus entwickeln können
- einen Algorithmenentwurf in ein Java-Programm unter Verwendung primitiver Datentypen, Arrays, Strings, der Ablaufstrukturen Sequenz, Alternative, Mehrfachauswahl, **while**-Schleife, **do-while**-Schleife, **for**-Schleife sowie Unterprogrammen unter Berücksichtigung von Regeln guten Programmierstils umsetzen können
- für ein gegebenes Java-Programm die Gültigkeitsbereiche von Variablen angeben können

# Gliederung der Lehreinheit

- 3.1 Ablaufstrukturen
- 3.2 Methoden

# Anweisungen und Ablaufstrukturen

- Anweisungen und Ablaufstrukturen dienen dazu festzulegen, in welcher Reihenfolge ein Programm abgearbeitet wird.
- z. B. festlegbar, dass bestimmte Programmteile nur ausgeführt werden sollen, wenn bestimmte Bedingung erfüllt ist, oder dass bestimmte Teile wiederholt ausgeführt werden sollen.
- Solche den Ablauf beeinflussenden Befehle heißen auch ***Kontrollstrukturen***.

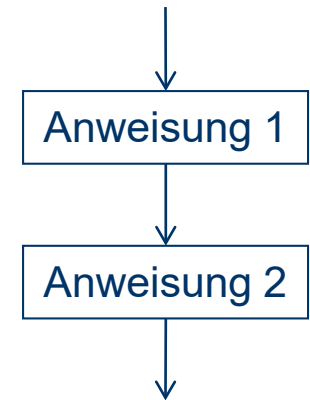
# Anweisung

- einzelne Vorschrift, die im Rahmen der Abarbeitung eines Programms auszuführen ist und den Programmzustand (z. B. Variablenwerte) ändert
- Beispiele
  - einfache Anweisung mit Semikolon (sog. **Terminator**) am Ende
    - Wertzuweisung  
`telKosten = (295 + anzSMS * 4 + anzMin * 5 ) / 100.0;`
  - **Block**, der Anweisungsfolge mit { und } klammert

# Anweisungsfolge, Sequenz

- **Sequenz:** Folge von nacheinander auszuführenden Anweisungen  $A_1, A_2, \dots, A_n$
- wird verwendet, wenn im Algorithmus Schritte nacheinander ausgeführt werden sollen
- durch eine Sequenz wird ein Programmanfangszustand  $z_A$  in einen Programmendzustand  $z_E$  überführt
- Java-Schreibweise:  $\mathbf{A_1; A_2; \dots A_n;}$
- zur besseren Lesbarkeit schreibt man im Programm jede Anweisung in eine neue Zeile
- Im Ablaufdiagramm (auch: Flussdiagramm) verwendet man durch Pfeile verbundene Rechtecke

Anweisungsfolge



# Java-Programm MinSuche

```
class MinSuche {  
    public static void main(String[] args) {  
        int[] a = {15, 23, 22, 21, 22, 18, 19, 17, 14, 16};  
        int merker = a[0];  
        int i = 1;  
        int n = a.length;  
        while (i < n) {  
            if (a[i] < merker) {  
                merker = a[i];  
            }  
            i = i + 1;  
        }  
        System.out.println(merker);  
    }  
}
```

Anweisungs-  
folgen

Anweisung



# Hinweis zur Programmzeilenformatierung

- Einrücktiefe stets 4 Leerzeichen, meist per Tabulator.
- Keine Zeile sollte mehr als 80 Zeichen enthalten, Kommentarzeilen nicht mehr als 70 Zeichen.
- Wenn eine Zeile zu lang wird, möglichst an folgenden Stellen umbrechen:
  - nach einem Komma      `int a, | b`
  - vor einem Operator      `47 | + 11`
  - möglichst gemäß der Bindungsstärke
- Folgezeile wird so weit eingerückt, wie der Ausdruck „auf gleicher Schachtelungsebene“ in der vorhergehenden Zeile.

# Java-Programm MinSuche

```
class MinSuche {  
    public static void main(String[] args) {  
        int[] a = {15, 23, 22, 21, 22, 18, 19, 17, 14, 16};  
        int merker = a[0];  
        int i = 1;  
        int n = a.length;  
        while (i < n) {  
            if (a[i] < merker) {  
                merker = a[i];  
            }  
            i = i + 1;  
        }  
        System.out.println(merker);  
    }  
}
```

*Einrücktiefe 8*

*Einrücktiefe 4*

- Block:

- Zusammenfassung einer Folge von Anweisungen
- an allen Stellen innerhalb eines Programms verwendbar, an denen auch eine einzelne Anweisung stehen kann

- Java-Schreibweise:

```
... .. {  
    Anweisung 1;  
    ...  
    Anweisung n;  
}
```

- Beispiel:

```
... .. {  
    if (a[i] < merker)  
        merker = a[i];  
    i = i + 1;  
}
```

*Einrücktiefe +4*



# Bedingte Anweisung, Fallunterscheidung, Alternative (I)

- Erlaubt es, eine Anweisung/einen Anweisungsblock nur dann ausführen zu lassen, wenn eine zugehörige Bedingung erfüllt ist.

- Bedingte Anweisung in Java: ***if-else-Anweisung***

- Allgemeine Form der ***if-else***-Anweisung:

```
if (<Bedingung>) {  
    <Anweisungen1>  
} else {  
    <Anweisungen2>  
}
```

boolescher Ausdruck

„then-Block“, „then-Zweig“

„else-Block“, „else-Zweig“

- Auswertung:

- Zuerst wird Bedingung ausgewertet, die als Ergebnis einen Wert vom Datentyp **boolean** liefern muss.
- Falls Bedingung erfüllt (**true**) ist, werden die Anweisungen im then-Block ausgeführt, ansonsten die Anweisung in else-Block.

- Beispiel: **`if (i > j) {max = i;} else {max = j;}`**

# Bedingte Anweisung, Fallunterscheidung, Alternative (II)

- **else**-Fall kann auch weggelassen werden, wodurch sich die **if-Anweisung** (auch: einseitige bedingte Anweisung) ergibt
- Allgemeine Form der **if**-Anweisung:

```
if (<Bedingung>) {  
    <Anweisungen>  
}
```

```
if (<Bedingung>)  
    <eine Anweisung>
```

- Beispiel (Tausche Werte von **i** und **j**):

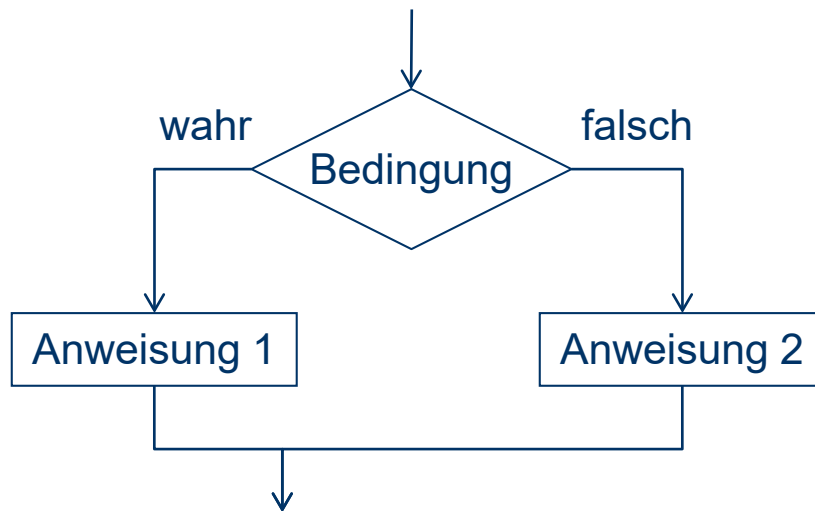
```
if (i > j) {  
    int h;  
    h = i;  
    i = j;  
    j = h;  
}
```

Kurzform für genau  
eine Anweisung  
→ besser vermeiden

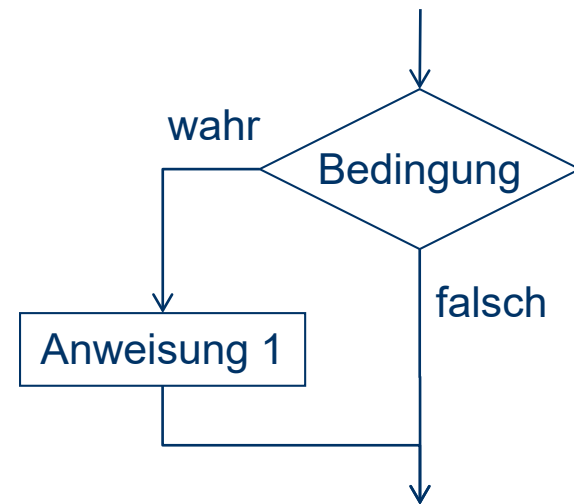
- Wenn die Bedingung nicht erfüllt ist, wird mit der ersten Anweisung nach der **if**-Anweisung fortgesetzt.

# Ablauf- und Syntaxdiagramm zur bedingten Anweisung

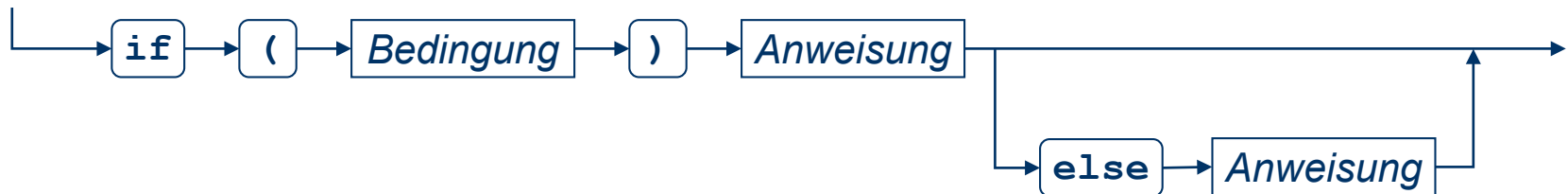
**if-else-Anweisung**



**if-Anweisung**



**if-else-Anweisung**



# Mögliche Fehlerquellen (I)

- Betrachten wir folgendes, syntaktisch korrektes Programm:

```
class FehlerDemo1 {  
    public static void main(String[] args) {  
        int i = 10;  
        int j = 12;  
        if ((i > j) || (i > 12)) { }  
        System.out.println(i);  
    }  
}
```

vermutlich ist der  
leere Rumpf der  
**if**-Anweisung  
nicht beabsichtigt


Problemlösung:  
schließende Klammer }  
verschieben

Wirkung:

- Die Anweisung **System.out.println(i);** wird in jedem Fall ausgeführt.
- Alleine das Einrücken der Anweisung reicht nicht.

# Mögliche Fehlerquellen (II)

```
class FehlerDemo2 {  
    public static void main(String[] args) {  
        int i = 10;  
        int j = 12;  
        if (i = j) {  
            System.out.println(i);  
        }  
    }  
}
```




- Vermutlich ist hier der Vergleich von **i** und **j** gemeint, korrekt wäre also **i == j**
- **i = j** stellt eine Zuweisung dar, die als Wert den Wert der rechten Seite (also: 12) liefert
- Die **if**-Anweisung erwartet aber einen booleschen Wert (also: **true** oder **false**).



# Mögliche Fehlerquellen (III)

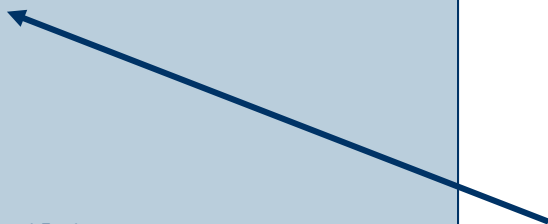
```
int k;  
int n = 5;  
if (n > 2) {  
    k = 3;  
}  
System.out.println(k);
```



- Vermutung:  
Weil `n` zur Laufzeit den Wert 5 hat, wird `k` bei jeder Programmausführung initialisiert sein.

- Übersetzer meldet aber:  
*Exception in thread "main"  
java.lang.Error: Unresolved  
compilation problem: The  
local variable k may not  
have been initialized*

```
// Deklaration und Initiali-  
// sierung von n bereits erfolgt  
int k = 0;  
if (n > 2) {  
    k = 3;  
}  
System.out.println(k);
```



so ok

# Mögliche Fehlerquellen (IV)

- Die gelegentlich in verschiedenen Online-Materialien zu findende Bezeichnung „*if-Schleife*“ ist *sachlich FALSCH!*
- Google-Recherche „if-schleife“ führte zu sehr vielen Fundstellen.
- In der if-Anweisung bzw. der if-else-Anweisung (Fallunterscheidungen) wird aufgrund der Auswertung einer Bedingung eine oder eine alternative Anweisung ausgeführt.
- In einer *Schleife* (Details gleich) wird aufgrund der Auswertung einer Bedingung eine Anweisung *wiederholt* (unter Umständen aber auch gar nicht) ausgeführt.

# Hinweis zur Programmzeilenformatierung

- Lange Bedingungen sollten geeignet eingerückt werden, um die Lesbarkeit des Programms zu verbessern.

- statt:

```
if ((vorname == "Peter") && (nachname == "Mueller"))  
    || ((vorname == "Peter") && (nachname == "Meier"))  
    || ((vorname == "Klaus") && (nachname == "Meier"))) {  
    System.out.println("ganz schoen seltener Name!");  
}
```

- besser:

```
if ((vorname == "Peter") && (nachname == "Mueller"))  
    || ((vorname == "Peter") && (nachname == "Meier"))  
    || ((vorname == "Klaus") && (nachname == "Meier"))) {  
    System.out.println("ganz schoen seltener Name!");  
}
```

weil dadurch der Rumpf gegenüber Bedingung hervorgehoben wird.

# Bedingter Ausdruck (auch: bedingter bzw. ?-Operator)

- Zweck:

- spezielle Form der Fallunterscheidung, die als Ergebnis einen Ausdruck liefert
- **if** ist kein Operator und hat kein Ergebnis

- Allgemeine Form:

**<Bedingung> ? <Ausdruck1> : <Ausdruck2>**

Wenn **<Bedingung>** wahr ist, dann ist **<Ausdruck1>** das Ergebnis des ?-Operators, sonst ist **<Ausdruck2>** das Ergebnis.

- Beispiel: statt

```
int max;
```

```
if (i > j) {max = i;} else {max = j;}
```

könnte man auch schreiben:

```
int max = (i > j) ? i : j;
```

# Hinweis zur Programmzeilenformatierung

- Bedingung sollte aus Lesbarkeitsgründen immer in Klammern stehen (außer bei Variante 3, s.u.), Leerzeichen um '?' und um ':'
- Formatierungsvarianten (je nach Geschmack und Platz)
- **Variante 1:** alles in einer Zeile  
`int max = (i > j) ? i : j;`
- **Variante 2:** zweizeilig; dann ? und : untereinander  
`String aktLaune = (kontoStand >= 0) ? "alles ok"  
: "frag nicht";`
- **Variante 3:** dreizeilig, dann =, ? und : untereinander  
`String aktLaune = kontoStand > 1000000  
? "bin dann mal weg"  
: "kommt auf genauen Wert an";`



# Wahr oder falsch?

- Was wird ausgegeben?

```
public class WahrOderFalsch {  
    public static void main(String[] args) {  
        System.out.println(    true?false:true  
                               == true?false:true);  
    }  
}
```

- Vorschläge:

- ☐ **true**
- ☐ **false**
- ☐ Etwas anderes

Bindungsstärke:  
? weniger stark als ==

```
true ? false  
    : ((true == true) ? false : true)
```


→ Im Zweifel immer Klammern verwenden und Bindungsstärke klar machen!

# Mehrfachauswahl: Schachtelung einfacher Alternativen (I)

- Prinzip: Anweisung des **else**-Zweigs enthält weitere bedingte Anweisung
- Beispiel: Vorzeichenberechnung

```
// a sei deklariert u. initialisiert
int vorzeichen; // vorz.: -1, 0, 1
if (a > 0) {
    vorzeichen = 1;
} else {
    if (a < 0) {
        vorzeichen = -1;
    } else {
        vorzeichen = 0;
    }
}
```

geeignete  
Klammerung und  
Einrückung  
machen den  
Programmtext  
besser lesbar



syntaktisch in Ordnung,  
aber ***schlechter Stil!***



```
int vorzeichen; // vorzeichen: -1, 0, 1
if (a > 0) { vorzeichen = 1; }
else if (a < 0) { vorzeichen = -1; } else { vorzeichen = 0; }
```

# Mehrfachauswahl: Schachtelung einfacher Alternativen (II)

- Formatierung gemäß Konvention:

```
// a sei deklariert u. initialisiert
int vorzeichen; // vorz.: -1, 0, 1
if (a > 0) {
    vorzeichen = 1;
} else if (a < 0) {
    vorzeichen = -1;
} else {
    vorzeichen = 0;
}
```

Geeignete  
Klammerung und  
Einrückung machen  
den Programmtext  
besser lesbar.

Alle **else** stehen untereinander!

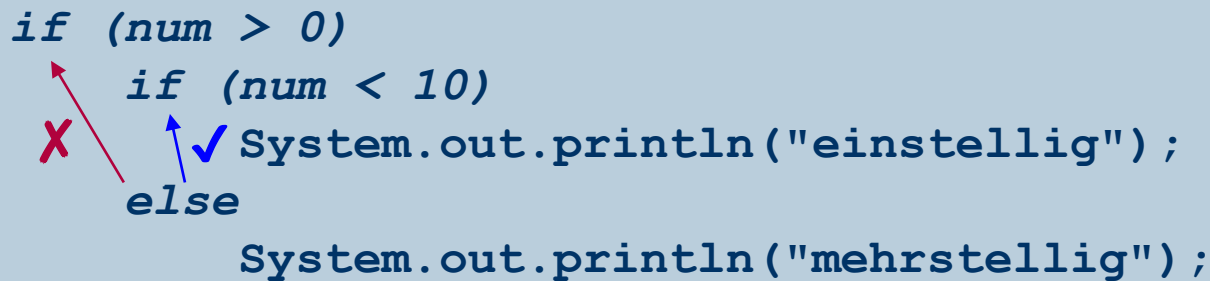
Das **else** umfasst nur eine  
Anweisung (die **if**-Anweisung), die  
nur im Fall einer Mehrfachauswahl  
ohne { . . . } formatiert wird.



# Zuordnung des `else`-Zweiges („baumelndes“, „dangling“ `else`)

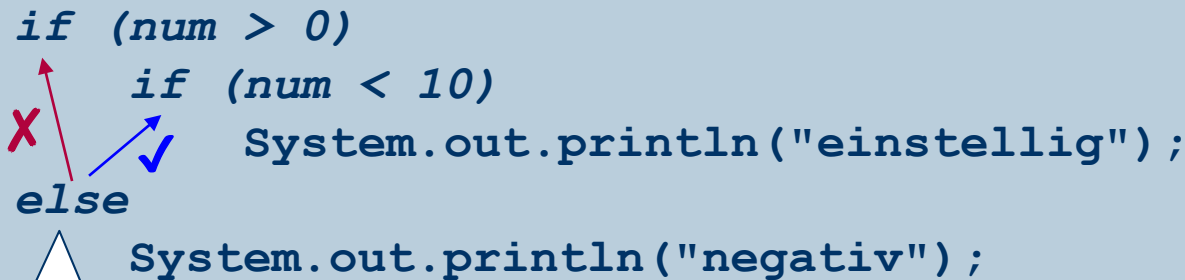
- Frage: Zu welchem `if` gehört das `else`?

```
if (num > 0)
  if (num < 10)
    System.out.println("einstellig");
  else
    System.out.println("mehrstellig");
```



Einrücktiefe ist irrelevant!  
In Java ist das innerste `if` gemeint, das selbst kein `else` hat.

```
if (num > 0)
  if (num < 10)
    System.out.println("einstellig");
else
  System.out.println("negativ");
```

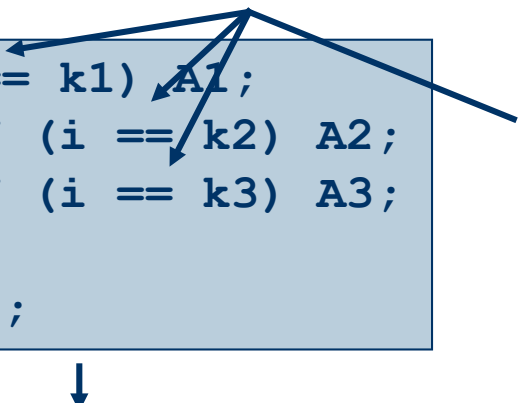


*Einrückung bewirkt nicht die Zuordnung zum ersten `if`*

# Mehrfachauswahl: **switch**-Anweisung

- Wenn Werte zur Auswahlsteuerung vom Typ **byte**, **short**, **int**, **char**, **String** oder vom Typ **enum** sind, ist komfortablere Variante der Mehrfachauswahl (sog. **switch-Anweisung**) verfügbar.


```
if (i == k1) A1;  
else if (i == k2) A2;  
else if (i == k3) A3;  
...  
else A0;
```



## **Achtung Fehlerquelle:**

- *in Bedingungen: doppeltes == zum Test auf Gleichheit*
- *in Wertzuweisungen: einfaches =*

```
// i muss byte, short, int, char, String oder enum sein  
switch (i) {  
case k1: A1; break; // ki sind Literale vom Typ byte, short  
case k2: A2; break; // int, char, String, enum;  
case k3: A3; break; // i muss vom gleichen Typ sein.  
...  
default: A0;  
}
```




- **break** beendet gesamte **switch**-Anweisung
- fehlt **break** in einem Fall, wird mit Anweisungen des direkten Falles fortgesetzt

# Beispiel: **switch**-Anweisung mit **break** (I)

Tage eines Monats:


```
int tage;  
switch (monat) {  
case 1:  
case 3:  
case 5:  
case 7:  
case 8:  
case 10:  
case 12:  
    tage = 31;  
    break;  
case 4:  
case 6:  
case 9:  
case 11:  
    tage = 30;  
    break;  
}
```



Kein **break** bei 1, 3, ... 10: bewirkt in diesem Fall die Ausführung des Programmtextes bei 12.

Letztes **break** eigentlich unnötig

```
case 2:  
    //nächste Folie  
    break;  
default:  
    tage = -1;  
    break;  
}  
if (tage < 0) ...  
else System.out.println(tage);
```



**default**-Teil hier für konservatives Programmieren, damit auch bei ungültigem Wert in **monat** ein geordnetes Verhalten realisiert ist.

# Beispiel: **switch**-Anweisung mit **break** (II)

## Mehrfachauswahl

### Schaltjahr-Regeln:

- Jahr ist kein Schaltjahr, wenn Jahreszahl nicht restlos durch 4 teilbar ist.
- Jahr ist kein Schaltjahr, wenn es durch 4 und 100 restlos teilbar ist.
- Jahr ist Schaltjahr, wenn es sowohl durch 4, durch 100 als auch durch 400 teilbar ist.

```
case 2:
    if (jahr % 4 != 0) {
        tage = 28;
    } else if (jahr % 100 != 0) {
        tage = 29;
    } else if (jahr % 400 != 0) {
        tage = 28;
    } else {
        tage = 29;
    }
    break;
```

$a \% b$ : Rest der Division  $a / b$   
(modulo-Operator)

# switch-Anweisung und Aufzählungstyp

- oft: Nutzung von Aufzählungstypen im Kontext der **switch**-Anweisung
- Beispiel:

```
Orientation direction = Orientation.NORTH;
// um 90 Grad im Uhrzeigersinn drehen
switch (direction) {
case NORTH:
    direction = Orientation.EAST;
    break;

case EAST:
    direction = Orientation.SOUTH;
    break;

...

default:
}
```

beachte: hier *nicht* **Orientation.NORTH**,  
sondern nur **NORTH**

# Hinweise zum Programmierstil

```
int numDays;  
switch (month) {  
case 4:  // Bearbeitung  
case 6:  // erfolgt im  
case 9:  // Fall 11  
case 11:  
    numDays = 30;  
    break;  
  
case 2:  
    // Schaltjahr-Fall  
    // hier bearbeiten  
    break;  
  
default:  
    numDays = 31;  
    break;  
}
```

- Kommentar, wenn „Durchreichen“ in den nächsten Fall wirklich beabsichtigt ist
- dann keine Leerzeile

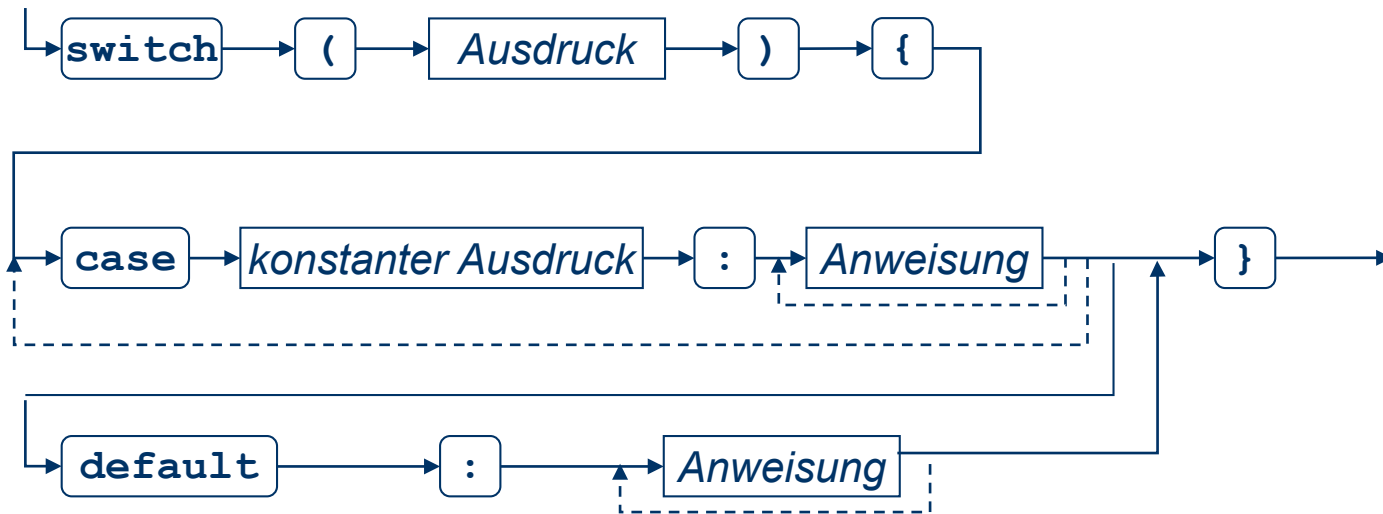
Leerzeile zwischen zwei Fällen

**default**-Fall immer vorsehen.  
Und immer am Ende vom **switch**.

eigentlich überflüssiges **break** am  
Ende schützt bei nachträglich  
darunter ergänzten weiteren Fällen

# Syntaxdiagramm zur **switch**-Anweisung

**switch-**  
Anweisung



Ausdruckstyp muss **byte**, **short**,  
**int**, **char**, **String** oder **enum** sein.



# Ohne Fleiß kein Preis (I)

- Was wird ausgegeben?

```
import java.util.*;
public class OhneFleissKeinPreis {
    private static Random rnd = new Random();
    public static void main(String[] args) {
        StringBuffer word = null;
        switch(rnd.nextInt(2)) {
            case 1: word = new StringBuffer('P');
            case 2: word = new StringBuffer('G');
            default: word = new StringBuffer('M');
        }
        word.append("ain");
        System.out.println(word);
    }
}
```

liegt in  $[0..2)$  → Doku lesen!

Da kein **break** vorhanden ist, wird immer der **default**-Fall ausgeführt. Daher wohl **Main**.

→ Doku lesen! **StringBuffer**-Konstruktor mit **char** legt Größe '**M**'=77 an.

Leider falsch!  
Ausgegeben wird *nur ain*.





# Ohne Fleiß kein Preis (II)

- Behebungsmöglichkeit:

```
import java.util.*;
public class OhneFleissKeinPreis {
    private static Random rnd = new Random();
    public static void main(String[] args) {
        StringBuffer word = null;
        switch(rnd.nextInt(3)) {
            case 1: word = new StringBuffer("P"); break;
            case 2: word = new StringBuffer("G"); break;
            default: word = new StringBuffer("M"); break;
        }
        word.append("ain");
        System.out.println(word);
    }
}
```

→ ein break in jedem Fall!

→ char oder String immer mit Bedacht wählen!

# while-Schleife (I)

- Grundgedanke der Steuerung: wiederholte Ausführung von Aktionen, bis eine Zielbedingung erfüllt ist.
- Entspricht der Schleife **solange ... führe aus ...** in der Pseudocode-Darstellung von Algorithmen

- Form:

```
while (<Bedingung>) {  
    <Anweisungen>  
}
```

Kurzform (vermeiden):

```
while (<Bedingung>)  
    <eine Anweisung>
```

- Auswertung

- Ausführung beginnt mit Auswertung der Bedingung (**Schleifentest**).
- Falls diese erfüllt (**true**) ist, werden die Anweisungen im **Schleifenrumpf** ausgeführt, an den Anfang der Schleife zurückgekehrt und auf dieselbe Weise verfahren.
- Anderenfalls wird Ausführung der Schleife abgeschlossen und Programm setzt mit erster Anweisung nach Schleife fort.

## while-Schleife (II)

- **<Anweisung>** sollte das Ergebnis der Bedingung beeinflussen (z. B. durch Veränderung von Variablenwerten, deren Wert Gegenstand der Schleifenbedingung ist), sonst sog. **Endlosschleife**, Algorithmus terminiert dann nicht.
- **while**-Schleifen heißen auch **abweisende** (oder **kopfgesteuerte**) **Schleifen**, weil ihr Rumpf u. U. nie durchlaufen wird (wenn die Schleifenbedingung beim ersten Test nicht erfüllt ist).
- Abbruch einer **while**-Schleife ist möglich mit der **break**-Anweisung (z. B. bei einer „endlos“ laufenden Schleife, die nur in einem bestimmten Fall unterbrochen werden soll).  
Gilt als *schlechter Programmierstil* und ist immer vermeidbar.

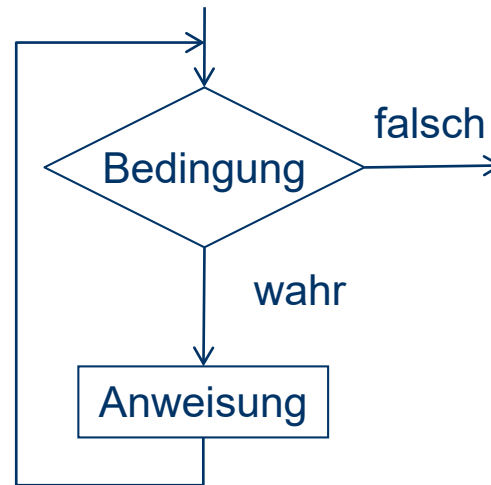
```
class MinSuche {  
    public static void main(String[] args) {  
        int[] a = {15, 23, 22, 21, 22, 18, 19, 17, 14, 16};  
        int merker = a[0];  
        int i = 1;  
        int n = a.length;  
        while (i < n) {  
            if (a[i] < merker) {  
                merker = a[i];  
            }  
            i = i + 1;  
        }  
        System.out.println(merker);  
    }  
}
```

Schleifenbedingung

Schleifenrumpf

# Ablauf- und Syntaxdiagramm zur `while`-Schleife

## `while`-Schleife



`while`-  
Schleife



# Beispiel: ganzzahliger 2er-Logarithmus

- Beispiel: ganzzahliger 2er-Logarithmus:  
„Wie oft muss man eine Zahl durch 2 teilen, bis sie 1 wird?“

Algorithmus:  
solange Zahl > 1, teile  
sie ganzzahlig durch 2;  
zähle die Anzahl der  
Teilungen in Variable  
**ergebnis**

```
public class Log2 {  
    public static void main(String[] args) {  
        int zahl = Integer.parseInt(args[0]);  
        int ergebnis = 0;  
        while (zahl > 1) {  
            zahl = zahl / 2;  
            ergebnis = ergebnis + 1;  
        }  
        System.out.println(ergebnis);  
    }  
}
```

- Aufrufbeispiel:  
**> java Log2 14**  
**3**



# Endlose Ungleichheit

- Wie wird daraus eine Endlosschleife?

```
while (i != i) {}
```

```
double i = 0.0 / 0.0;  // = Double.NaN;  
while (i != i) {}
```

→ Beim Rechnen mit Gleitkommatypen, gibt es Zahlen, die *Not-a-Number* sind. Rechnen mit NaN ergibt immer NaN, auch `NaN != NaN`

- Wie wird daraus eine Endlosschleife? (Diesmal ohne Gleitkomma!)

```
while (i != i + 0) {}
```

```
String i = "";  
while (i != i + 0) {}
```

+ hier String-Konkatenation,  
Objekt-Referenzvergleich

Lieber/wenn überhaupt explizit: + "0"  
→ Lesbar programmieren!

# do-while-Schleife (I)

- Schleife, bei der bekannt ist, dass der Schleifenrumpf mindestens einmal durchlaufen wird.
- Deshalb heißen **do-while**-Schleifen auch ***nicht-abweisende*** (auch: ***fußgesteuerte***) **Schleifen**

- Standardform:

```
do {  
    <Anweisungen>  
} while (<Bedingung>);
```

Kurzform (vermeiden):

```
do <Anweisung>  
while (<Bedingung>);
```

- Beispiel:

```
int i = 1;  
do {  
    System.out.println(i);  
    i = i + 1;  
} while ( i < 5 );
```



## do-while-Schleife (II)

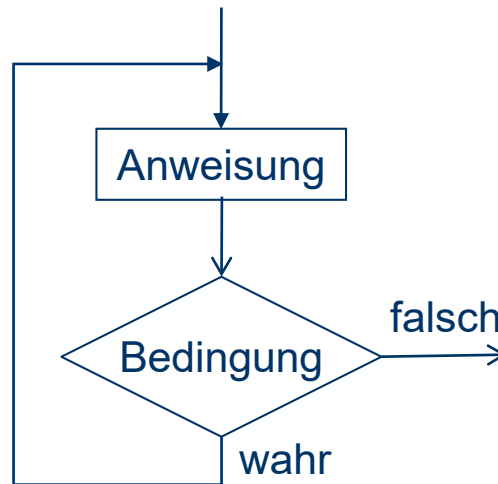
- Jede **do-while**-Schleife kann leicht in eine **while**-Schleife folgender Form überführt werden:

```
{  
    <Anweisungen>  
    while (<Bedingung>) {  
        <Anweisungen>  
    }  
}
```

- *Potenzielles Problem:*
  - *<Anweisungen> werden dupliziert.*
  - *Das ist potenziell kritisch im Hinblick auf Wartbarkeit.*
- Manchmal ist **do-while**-Schleife intuitiver, z. B. beim Warten auf eine Benutzerinteraktion.

# Ablauf- und Syntaxdiagramm zur `do-while`-Schleife

## `do-while`-Schleife



## `do-while`- Schleife



# for-Schleife (auch: Zählschleife) (I)

- kann verwendet werden, wenn bereits vor der Ausführung bekannt ist, wie oft der Schleifenrumpf durchlaufen werden soll
- Beispiel: mache etwas mit jedem Element eines Arrays  
(z. B. auf dem Bildschirm ausgeben)
- Standardform:  

```
for (<Initialausdruck>; <Bedingung>; <Inkrementausdruck>) {  
    <Anweisungen>  
}
```
- Auswertung:
  - Initialisierungsausdr. auswerten; ggf. dort deklarierte Variablen anlegen.
  - Schleifenbedingung auswerten; wenn nicht erfüllt: Schleifenende, also: **abweisende** (oder: **kopfgesteuerte**) **Schleife**
  - Ausführung der <Anweisungen>
  - Ausführung des <Inkrementausdrucks>, der typischerweise die Variablen verändert (hochzählt), die in der Initialisierung gesetzt und im Schleifentest mit Grenzwerten verglichen werden.

# for-Schleife (auch: Zählschleife) (II)

Initialisierungsausdruck

Inkrementierungsausdruck

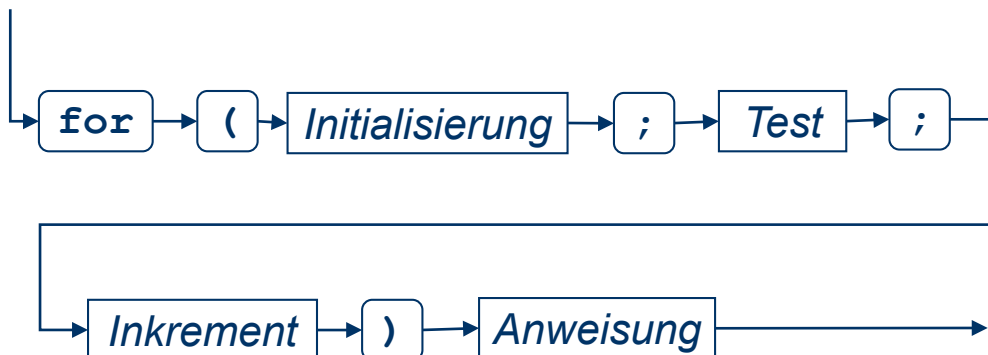
```
int summe = 0;  
for (int i = 1; i <= 5; i++) {  
    summe += i;  
}
```

i: sog. Laufvariable

Schleifenbedingung

i	summe
vor Schleife: nicht existent	0
1	1
2	3
3	6
4	10
5	15
6	15
nach Schleife: nicht existent	15

for-Schleife





# Schau genau (II)

## ■ Was wird ausgegeben?

```
public class SchauGenau {  
    public static void main(String[] args) {  
        int count = 0;  
        for (int i = 0; i < 100; i++); {  
            count++;  
        }  
        System.out.println(count);  
    }  
}
```

Der Block der for-Schleife  
endet nach dem ;  
Dann folgt separater Block.

## ■ Vorschläge:

- ☐ 99
- ☐ 100
- ☐ 101
- ☐ Etwas anderes

1

# for- und while-Schleifen (I)

- Eine **for**-Schleife kann (im Wesentlichen) nach folgendem Muster auf eine semantisch äquivalente **while**-Schleife abgebildet werden:

```
for (<Initialausdruck>; <Bedingung>; <Inkrementausdruck>) {  
    <Anweisungen>  
}
```



```
{  
    <Initialausdruck>;  
    while (<Bedingung>) {  
        <Anweisungen>  
        <Inkrementausdruck>;  
    }  
}
```

# for- und while-Schleifen (II)

Aufsummieren der Zahlen  
von 1 bis n  
mit **for**-Schleife

```
int summe = 0;
for (int i = 1; i <= n; i++) {
    summe += i;
}
```

*Initialausdruck*

*Anweisung*

Aufsummieren der Zahlen  
von 1 bis n  
mit **while**-Schleife:

```
int summe = 0;
{
    int i = 1;
    while (i <= n) {
        summe += i;
        i++;
    }
}
```

*Bedingung*

*Inkrementausdruck*

# Geschachtelte Schleifen

- Beispiel: Ausgabe eines Rechtecks der Größe  $n \cdot m$  aus Sternen

```
public class Rechteck {  
    public static void main(String[] args) {  
        int n = Integer.parseInt(args[0]);  
        int m = Integer.parseInt(args[1]);  
        for (int i = 0; i < n; i = i + 1) {  
            for (int j = 0; j < m; j = j + 1) {  
                System.out.print('*');  
            }  
            System.out.println(); // beginne neue Zeile  
        }  
    }  
}
```

äußere Schleife { } innere Schleife

Der Rumpf der **äußeren Schleife** ist mit { } geklammert. Er besteht aus zwei Anweisungen (der inneren **for**-Schleife und dem **println()**).

Die **innere Schleife** bräuchte keine { }, da der Rumpf nur aus einer Anweisung besteht. Klammern sind aber möglich und gehören zum guten Stil!



# Mögliche Fehlerquellen (I)

- Achtung! In Java ist nach

```
for (int i = 0; i < n; i = i + 1) {  
    ...  
}  
int j = i;           // Fehler
```

die Variable `i` nicht mehr bekannt, d. h. implizit bildet die `for`-Schleife einen Block, in dem die Initialisierung stattfindet.


- Will man den Wert von `i` nach der `for`-Schleife weiterverwenden, so muss `i` außerhalb der Schleife deklariert werden:

```
int i;  
for (i = 0; i < n; i = i + 1) { //nicht int i = 0  
    ...  
}  
  
int j = i;           // ok
```

# Mögliche Fehlerquellen (II)

- Achtung! Folgendes Programmstück funktioniert nicht:

Lösung



```
for (int i = 1; i <= n; i++) {  
    int summe;  
    summe += i;  
}  
System.out.println(summe);  
// Uebersetzungsfehler: summe nicht bekannt
```

- Grund: Variable **summe** ist nur gültig nur im umschließenden Block. Nach Ausführung der Schleife/des Blocks ist die Variable unbekannt.
- Konzeptionell:
  - Die Variable **summe** würde bei jedem Durchlauf des Rumpfes neu angelegt.
  - Dann würde zu dieser noch un-initialisierten Variable der Wert von **i** addiert.
  - Vor dem nächsten Durchlauf würde die Variable **summe** wieder gelöscht.

# Mögliche Fehlerquellen (III)

- Beispiel mit typischen Fehlern:

*"die" main-Methode hat void als Ergebnistyp*

*Parametername fehlt*

```
public class Summe {  
    public static int main(String[]) {  
        int[] a = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
  
        int summe = 0;  
        int i;  
        for (int i = 1; i <= a.length; i+) {  
            summe += a[i];  
            System.out.println("Die Summe ist: ", summe)  
        }  
    }  
}
```

*lokale Variablen mit gleichem Namen verboten*

*Arraygrenzen 0...length-1*

*++*

*} fehlt*

*Konkatenation von Zeichenketten mit +*

*; fehlt*



# In der Tretmühle (I)

- Was wird ausgegeben?

```
public class RatRace {  
    static final int END = Integer.MAX_VALUE;  
    static final int START = END - 100;  
  
    public static void main(String[] args) {  
        int count = 0;  
        for (int i = START; i <= END; i++)  
            count++;  
        System.out.println(count);  
    }  
}
```

- Vorschläge:

- ☐ 100
- ☐ 101
- ☐ Etwas anderes

Nichts!  
Endlosschleife

Überlauf:  
 $\text{MAX\_VALUE} + 1 = \text{MIN\_VALUE}$



## In der Tretmühle (II)

- Behebungsmöglichkeiten:

```
for (long i = START; i <= END; i++)
```

oder

```
int i = START;  
do {  
    count++;  
} while (i++ != END);
```

→ Bei der Arbeit mit Ganzzahl-Typen immer an Überläufe denken!

3.1 Ablaufstrukturen

→ 3.2 Methoden

# Sortieren (I)

- häufig auftretendes *Problem*: **Sortieren**, z. B. von
  - Dateien nach Dateinamen alphabetisch
  - Telefonbucheinträgen nach Namen
  - Aktienkursen nach Gewinnen und Verlusten
  - ...
  - Zahlen einer Zahlenfolge nach Wert (Abstraktion der vorherigen Bsp.)
- *Spezifikation*
  - Problem: Sortieren einer Zahlenfolge
  - gegeben: Zahlenfolge  $a_0, a_1, \dots, a_{n-1}$
  - gesucht: Anordnung der Eingabewerte nach steigendem Wert, so dass  $a_i \leq a_{i+1}$
- Beispiel:
  - Startfolge: 11, 7, 8, 3, 15, 13, 9, 19, 18, 10, 4
  - Zielfolge: 3, 4, 7, 8, 9, 10, 11, 13, 15, 18, 19

# Sortieren (II)

- Da das Sortieren einer großen Menge von Elementen (z. B. alle Telefonbucheinträge einer Großstadt) unter Umständen sehr zeitaufwändig ist, hat die Informatik dafür viele verschiedene Strategien entwickelt und untersucht.
- Wir beginnen mit einer sehr einfachen Strategie:
  - **Sortieren durch Minimumssuche**
  - auch bekannt als: Sortieren durch Auswählen, SelectionSort

z. B. Sortieren  
von Spielkarten





# Videos zu MinSort (engl. selection sort) (Auswahl)



[http://www.youtube.com/watch?v=TW3\\_7cD9L1A](http://www.youtube.com/watch?v=TW3_7cD9L1A)



[http://www.youtube.com/watch?v=INHF\\_5RlxTE](http://www.youtube.com/watch?v=INHF_5RlxTE)

# Algorithmus MinSort

- *Idee für Algorithmus MinSort( $a_0, a_1, \dots, a_{n-1}$ )*
  - Suche ein Element  $a_k$  der Folge mit dem kleinsten Wert.
  - Füge  $a_k$  an das Ende einer neuen Folge an, die am Anfang leer ist.
  - Entferne  $a_k$  aus der Eingabefolge und verfare mit der Restfolge genauso.
- *Strukturierung der Idee mittels Kontrollstrukturen*

*weise restfolge die gegebene Folge zu;*  
*solange restfolge nicht leer ist, führe aus:*
  - suche ein Element  $a_k$  mit dem kleinsten Wert in restfolge;*
  - füge  $a_k$  an ergebnisfolge an;*
  - entferne  $a_k$  aus restfolge;**gib ergebnisfolge aus;*

# Ablauf der Sortierung einer Folge von Zahlen mittels **MinSort**

- gegebene Folge: 11, 7, 8, 3, 15, 13, 9, 19, 18, 10, 4

Durch- lauf	<i>restfolge</i>	$a_k$	<i>ergebnisfolge</i>
1	11, 7, 8, 3, 15, 13, 9, 19, 18, 10, 4	3	3
2	11, 7, 8, 15, 13, 9, 19, 18, 10, 4	4	3, 4
3	11, 7, 8, 15, 13, 9, 19, 18, 10	7	3, 4, 7
4	11, 8, 15, 13, 9, 19, 18, 10	8	3, 4, 7, 8
5	11, 15, 13, 9, 19, 18, 10	9	3, 4, 7, 8, 9
6	11, 15, 13, 19, 18, 10	10	3, 4, 7, 8, 9, 10
7	11, 15, 13, 19, 18	11	3, 4, 7, 8, 9, 10, 11
8	15, 13, 19, 18	13	3, 4, 7, 8, 9, 10, 11, 13
9	15, 19, 18	15	3, 4, 7, 8, 9, 10, 11, 13, 15
10	19, 18	18	3, 4, 7, 8, 9, 10, 11, 13, 15, 18
11	19	19	3, 4, 7, 8, 9, 10, 11, 13, 15, 18, 19

# Algorithmus MinSort

- *Strukturierung der Idee mittels Kontrollstrukturen*

*weise restfolge die gegebene Folge zu;*

*solange restfolge nicht leer ist, führe aus:*

*suche ein Element  $a_k$  mit dem kleinsten Wert in restfolge;*

*füge  $a_k$  an ergebnisfolge an;*

*entferne  $a_k$  aus restfolge;*

*gib ergebnisfolge aus;*

- *Pseudocode*

*restfolge := gegebenefolge;*

*solange restfolge nicht leer ist, führe aus: {*

*$a_k := \text{minSuche}(\text{restfolge});$*

*fuegeAn( $a_k$ , ergebnisfolge);*

*entferne( $a_k$ , restfolge);*

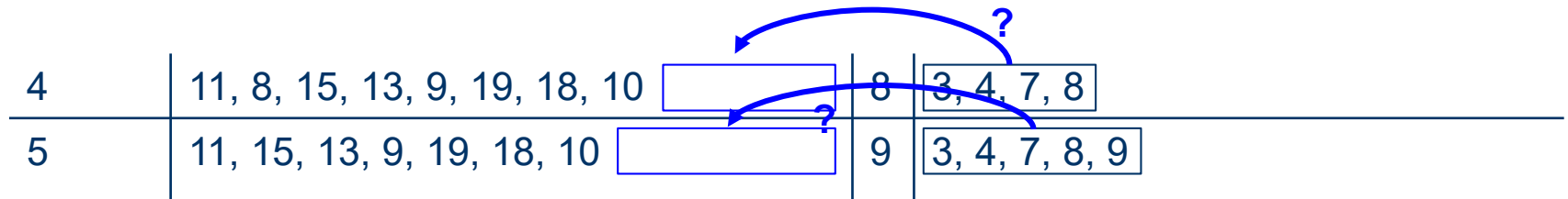
*}*

*gib ergebnisfolge aus;*

**Methodenaufruf:**  
**Unterprogramm** minSuche  
anwenden auf *restfolge*

# Verbesserung des Algorithmus MinSort

- nachteilig: Verschwendung von Speicherplatz



- Ziel: nur eine Folge verwenden, Speicherplatz sparen
- Idee: Elemente der gegebenen Folge vertauschen
- *Idee für verbesserten Algorithmus MinSort2 ( $a_0, a_1, \dots, a_{n-1}$ ):*
  - Suche ein Element der Folge mit dem kleinsten Wert. Vertausche das erste Element der Folge mit diesem Wert.
  - Suche in der Restfolge ab dem zweiten Element ein Element mit dem kleinsten Wert. Vertausche das zweite Element der Folge mit diesem Wert.
  - Führe das Verfahren mit der Restfolge ab dem dritten, vierten usw. und bis zum vorletzten Element aus.

# Ablauf der Sortierung einer Folge von Zahlen mittels **MinSort2**

Durchlauf												$a_k$
1	11,	7,	8,	3,	15,	13,	9,	19,	18,	10,	4	3
2	3,	7,	8,	11,	15,	13,	9,	19,	18,	10,	4	4
3	3,	4,	8,	11,	15,	13,	9,	19,	18,	10,	7	7
4	3,	4,	7,	11,	15,	13,	9,	19,	18,	10,	8	8
5	3,	4,	7,	8,	15,	13,	9,	19,	18,	10,	11	9
6	3,	4,	7,	8,	9,	13,	15,	19,	18,	10,	11	10
7	3,	4,	7,	8,	9,	10,	15,	19,	18,	13,	11	11
8	3,	4,	7,	8,	9,	10,	11,	19,	18,	13,	15	13
9	3,	4,	7,	8,	9,	10,	11,	13,	18,	19,	15	15
10	3,	4,	7,	8,	9,	10,	11,	13,	15,	19,	18	18
11	3,	4,	7,	8,	9,	10,	11,	13,	15,	18,	19	19
12	3,	4,	7,	8,	9,	10,	11,	13,	15,	18,	19	

# Java-Programm MinSort2 (I)

```
class MinSort2 {  
  
    static int minSuche2(int[] r, int start) {  
        // s. naechste Folie  
    }  
  
    public static void main(String[] args) {  
        int[] a = {11, 7, 8, 3, 15, 13, 9, 19, 18, 10, 4};  
        int k;    // speichert den Minimumindex  
        for (int i = 0; i < a.length - 1; i++) {  
            k = minSuche2(a, i);  
            int tmp = a[i];  
            a[i] = a[k];  
            a[k] = tmp;  
        }  
        for (int i = 0; i < a.length; i++) {  
            System.out.println(a[i]);  
        }  
    }  
}
```

Hauptprogramm ist  
selbst Methode

Aufruf der Methode (des  
Unterprogramms) mit  
den Argumenten **a** und **i**

Vertauschen der  
Werte von **a[i]**  
und **a[k]**

# Java-Programm MinSort2 (II)

```
class MinSort2 {
```

Rückgabety  
der Methode

Name der  
Methode

Parameter der  
Methode

```
    static int minSuche2(int[] r, int start) {  
        // gibt den Index eines Elements von r mit kleinstem  
        // Wert im Bereich ab Index start zurueck  
        int wmerker = r[start]; // merkt den kleinsten Wert  
        int imerker = start;    // merkt Index zum kleinsten Wert  
        for (int i = start; i < r.length; i++){  
            if (r[i] < wmerker) {  
                wmerker = r[i];  
                imerker = i;  
            }  
        }  
        return imerker;  
    }
```

Methodenrumpf

```
    public static void main(String[] args) {  
        // s. vorherige Folie  
    }
```

**return-Anweisung:** Ende der Ausführung der Methode und Rückgabe des in der Variable **imerker** gespeicherten Werts an die aufrufende Stelle

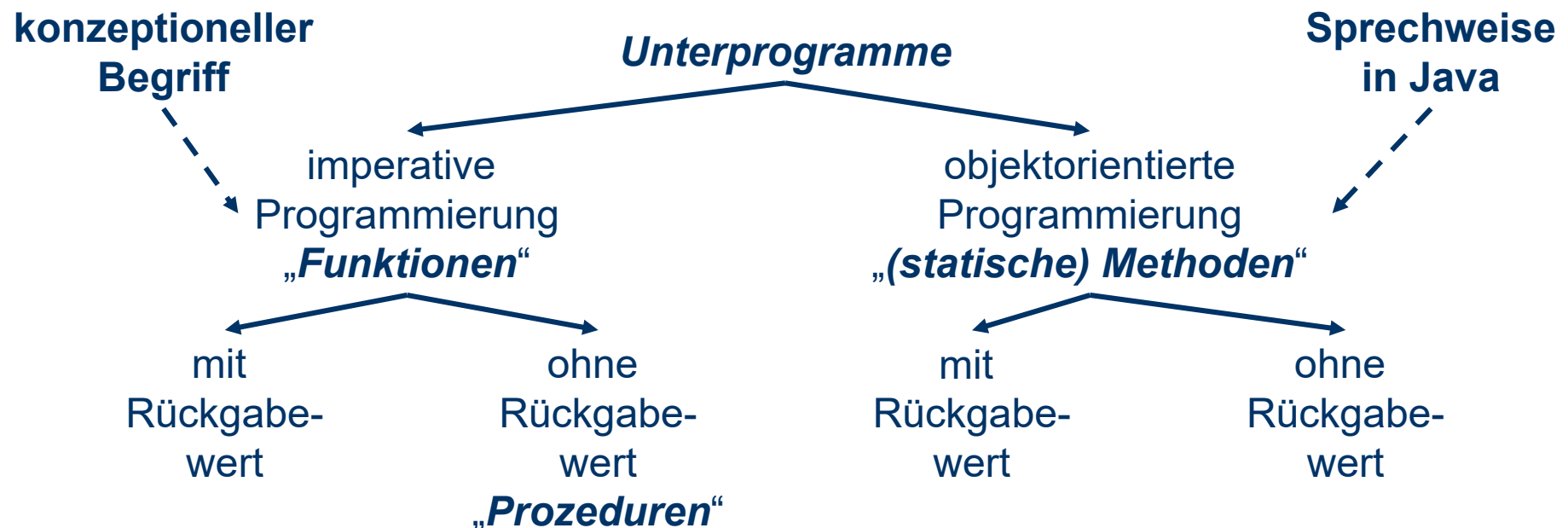


# Einschub: Kommentare

- In Java unterscheidet man zwischen Kommentaren zur Implementierung und Kommentaren zur Dokumentation:
    - **Implementierungskommentare**
      - werden gekennzeichnet durch `//` (sog. **Zeilenkommentar**; alles danach bis zum Ende der Zeile gilt als Kommentar) oder `/* ... */` (sog. **Blockkommentar**; alles dazwischen - ggf. auch über mehrere Zeilen - gilt als Kommentar),
      - dienen der Erläuterung des Programmtextes,
      - werden auch verwendet, um vorübergehend nicht benötigte Programmteile „auszukommentieren“.
    - **Dokumentationskommentare**
      - werden durch `/**...*/` gekennzeichnet,
      - erlauben standardisierte Beschreibung von Programmtexten.
      - Mittels des Werkzeugs `javadoc` lassen sich solche Dokumentationskommentare aus Quelltexten extrahieren und `html`-Seiten generieren (analog zur Dokumentation der Java-Standard-Bibliotheken).
- Details: <http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>

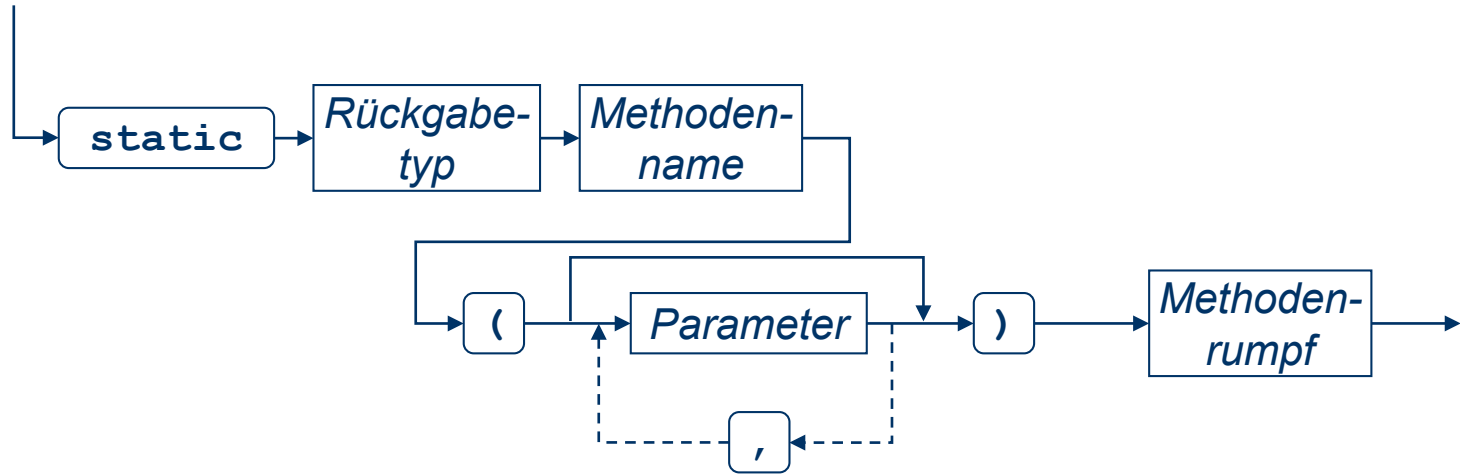
# Unterprogramm, Funktion, Prozedur, Methode

- benannte Folge von Anweisungen, zusammengefasst zu einem Block (Strukturierung von Programmen)
- Programmstück, das abhängig von Eingabewerten etwas berechnen und an mehreren Stellen eines Programms ausgeführt werden kann (Wiederverwendungsaspekt)
- Begriffe:



# Syntaxdiagramme zur Deklaration von Methoden

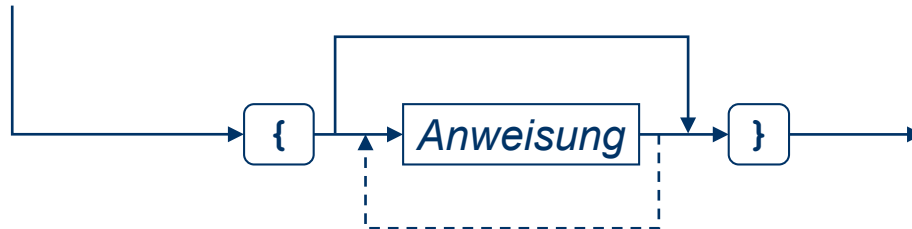
statische Methode



Parameter



Methodenrumpf



# Deklaration von Methoden (I)

- Methoden müssen im Rumpf der Klasse deklariert werden:

```
static <Rückgabetyp> <Methodenname> (<Param 1>, ...,  
                                     <Param n>) {  
    <Anweisung 1>  
    ...  
    <Anweisung m>  
}
```

sog. **Signatur** (auch: **Schnittstelle**) der Methode

sog. **Rumpf** der Methode

- Hinweis: es gibt auch nicht-statische **Methoden** (Konzept aus der Objektorientierung), deren Deklaration analog, nur ohne das Schlüsselwort **static** erfolgt. (Details später)
- Deklarierte Methode kann in einem Programmstücks (in einer Methode) aufgerufen (ausgeführt) werden.

# Deklaration von Methoden (II)

- *Deklaration als statische Methode:*

- ☐ Schlüsselwort **static** am Beginn der Deklaration

- *Rückgabedatentyp:*

- ☐ gibt an, von welchem Datentyp der von der Methode zurückgegebene Wert ist
- ☐ kein Rückgabebetyp: Schlüsselwort **void** anstelle des Rückgabedatentyps

**static void eineMtdOhneRueckgabeWert(int n) {...}**

- *Methodenname:*

- ☐ bezeichnet die Methode
- ☐ dient zum Aufruf der Methode an der Stelle, an der sie verwendet werden soll
- ☐ gleiche Regeln wie für die Bildung von Variablennamen

# Deklaration von Methoden (III)

- *Liste von (formalen) Parametern*
  - Eingabewerte an Methoden übergeben, die in Berechnung eingehen
- *Block (Rumpf der Methode)*
  - enthält die eigentlichen Anweisungen der Methode.
  - Falls der Rückgabebetyp der Methode nicht `void` ist, enthält der Rumpf eine (oder mehrere) sog. **Rückgabeeanweisung(en)** (auch: **return-Anweisungen**), mittels derer ein zuvor berechneter Wert an die jeweils aufrufende Stelle zurückgegeben wird.
  - Hinweis: Methode (gilt nicht für Typ `void`) hat genau einen (möglicherweise komplexen, Details bei Objektorientierung) Rückgabewert; je nach Verlauf der Berechnungen kann der aber von verschiedenen Stellen der Methode an die aufrufende Stelle zurückgegeben werden.

# Formale Parameter, Parameterliste

- definieren die Schnittstelle der Methode zur Übergabe bzw. Übernahme von Eingabewerten
- Deklaration eines formalen Parameters:
  - analog Variablendeklaration
  - Form: **<Datentyp> <Parametername>**
- Deklaration im Rahmen der Methodendeklaration:
  - einzeln, oder im Fall von mehreren durch Kommata getrennt, in Klammern nach dem Methodennamen
  - Folge von Parameterdeklarationen heißt auch ***Parameterliste***
- Beispiele:
  - `static int minSuche2(int[] r, int start) {...}` hat die formalen Parameter `int[] r` und `int start`
  - Soll Methode kein Wert übergeben werden, wird Platz zwischen den Klammern () leer gelassen, z. B.  
`static int eineMethodeOhneParameter() {...}`

# Hinweis zur Programmzeilenformatierung

- im Falle eines zu langen Kopfes einer Methodendeklaration wird folgende Formatierung empfohlen

- anstatt:

```
static void eineMethode(int ersterParameter, int  
                        zweiterParameter) {  
  
    ...  
  
}
```

- besser:

```
static void eineMethode(int ersterParameter,  
                        int zweiterParameter) {  
  
    ...  
  
}
```



# Methodenrumpf (I)

- enthält ggfs. Deklaration von sog. **lokalen Variablen**
- Folge von Anweisungen mit lesenden und schreibenden Zugriffen auf formale Parameter und lokale Variablen

Unterschied zwischen formalen Parametern und lokalen Variablen: formalen Parametern muss vor Verwendung im Rumpf kein Wert zugewiesen werden, denn sie erhalten einen beim Aufruf

- eine oder mehrere **Rückgabeeanweisungen** (**return**-Anweisung)
- Fall 1: Rückgabebetyp ist *nicht void*

- Form: **return** **<Rückgabewert>;**

Keine Klammern um Ausdruck  
<Rückgabewert> setzen.

- Beispiel: **return merker;**

- Rückgabe des Wertes, der sich durch Auswertung des Ausdrucks **<Rückgabewert>** ergibt

- Abarbeitung des Methodenrumpfs wird nach Auswertung der **return**-Anweisung beendet. Dann wird an der Aufrufstelle in der aufrufenden Methode fortgefahren.

# Methodenrumpf (II)

- Fall 2: Rückgabetyp ist `void`
  - Methoden mit Rückgabetyp `void` geben keinen Wert zurück
  - Optional kann Ausführung des Methodenrumpfs bei Bedarf explizit mit `return;` beendet werden.
  - Falls keine `return`-Anweisung angegeben wurde, wird der Methodenrumpf komplett abgearbeitet und anschließend zur aufrufenden Stelle zurückgekehrt.

# Java-Programm MinSort2

```
class MinSort2 {  
  
    static int minSuche2(int[] r, int start){  
        ...  
        return imerker;  
    }  
  
    public static void main(String[] args){  
        int[] a = {11, 7, 8, 3, 15, 13, 9, 19, 18, 10, 4};  
        int k;    // speichert den Minimumindex  
        for (int i = 0; i < a.length - 1; i++) {  
            k = minSuche2(a, i);  
            int tmp = a[i];  
            a[i] = a[k];  
            a[k] = tmp;  
        }  
        for (int i = 0; i < a.length; i++) {  
            System.out.println(a[i]);  
        }  
    }  
}
```

Wert von **imerker**  
zurückgegeben

„aufrufende Stelle“:  
Wert von **imerker** wird  
**k** zugewiesen

# Hilfreiche Analogie: mathematische Funktionen

- Statische Methode mit Rückgabe

```
static int minSuche2(int[] r,  
                    int start) {  
    ...  
}
```

- Mathematische Funktionen

$$f(x) = 3x + 17$$

$$f(x, y) = xy + 2x + 2y + 42$$

$$g(x) = 5x + f(x)$$

- *Methodenname* **minSuche2** entspricht *f* bzw. *g*
- *formale(r) Parameter* entsprechen dem/den *Funktionsparameter(n)*
- *Rückgabewert* entspricht dem *Funktionswert*
- Aufruf von **minSuche2** in **main** entspricht Verwendung von *f* in *g*
- Wesentliche Unterschiede:
  - Methodenparameter können beliebige Typen sein, z. B. Zahlen, Zeichen, Texte etc.
  - Methoden können Seiteneffekte haben.

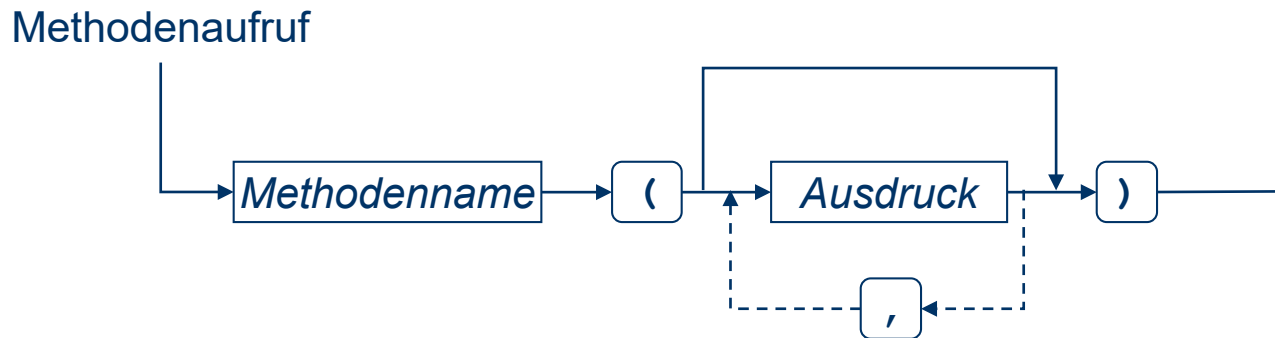
# Methodenaufruf (I)

- Aufruf einer Methode erfolgt durch Angabe ihres Namens, gefolgt von einer geklammerten Liste von **Argumenten**

- Beispiel: `k = minSuche2(a, i);`

- allgemeine Form

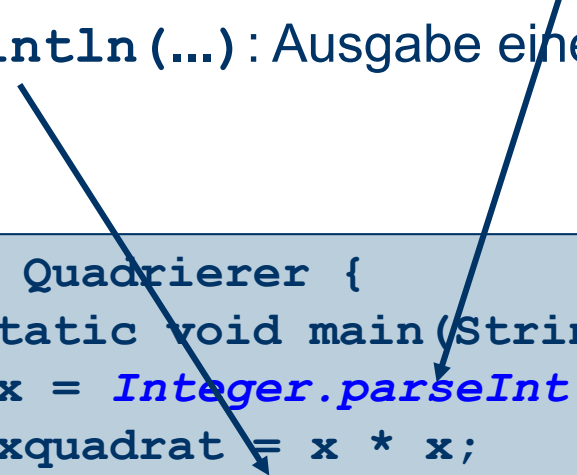
`<Methodenname> (<argument 1>, ..., <argument n>);`



# Methodenaufruf (II)

- Aufruf von in Java bereits vorhandenen Methode
  - Umwandlung von String in int: `Integer.parseInt(...)`
  - `System.out.println(...)`: Ausgabe eines Strings auf Standardausgabe

```
public class Quadrierer {  
    public static void main(String[] args) {  
        int x = Integer.parseInt(args[0]);  
        int xquadrat = x * x;  
        System.out.println(xquadrat);  
    }  
}
```



# Beachten der Schnittstelle der Methode beim Aufruf

- Als Argument kann alles eingesetzt werden, was auf der rechten Seite einer Wertzuweisung zu einer Variablen vom Typ des jeweiligen formalen Parameters stehen kann.
- Beispiel:
  - gegeben: `static int minSuche2(int[] r, int start) {...}`  
sowie Deklaration und Initialisierung von `int[] a` und `int i`
  - dann:

```
k = minSuche2(a, i);           // ok
l = minSuche2(a, "Unsinn");    // liefert Fehler
```
- Für jeden formalen Parameter der Parameterliste muss genau ein Argument angegeben werden.
- Beispiel:

```
l = minSuche2(a);              // liefert Fehler
```

# Hinweis zur Programmzeilenformatierung

- Lange Methodenaufrufe sollten geeignet formatiert werden, um die Programmlesbarkeit zu erhöhen.
- Lange Argumente untereinander setzen.
- Anstatt:

```
ergebnis = eineMethode("eine Zeichenkette",  
    "noch eine Zeichenkette");
```

besser:

```
ergebnis = eineMethode("eine Zeichenkette",  
    "noch eine Zeichenkette");
```



# Wirkung eines Methodenaufrufs (I)

- Jedem formalen Parameter wird der Wert des/seines Arguments zugewiesen;  
analog einer Variablendeklaration mit Wertzuweisung.
- Beispiel:
  - Deklaration `static int minSuche2(int[] r, int start) {...}`  
mit formalen Parameter `int[] r` und `int start`
  - Argumente: `a`, `i`
  - muss verstanden werden als Wertzuweisung `r = a` und `start = i`
- Anweisungen im Rumpf der Methode mit dem gegebenen Namen (hier: von `minSuche2`) werden ausgeführt.
- Ausführung des Rumpfes endet, wenn `return`-Anweisung erreicht wird (oder bei letzter Anweisung im Fall von `void`-Methoden);  
der sich dort ergebende Rückgabewert wird an die aufrufende Stelle zurückgegeben (nicht bei `void`) und kann dort verwendet werden.

# Wirkung eines Methodenaufrufs (II)

## Methode

```
static int square(int x) {  
    int y = x * x;  
    return y;  
}
```

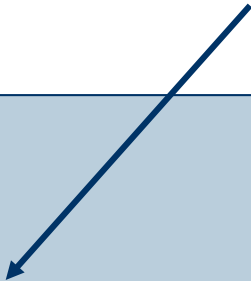
Anlegen einer lokalen Variable  
für formalen Parameterwert;  
Wertzuweisung des Arguments

## Der Aufruf

```
...  
q = square(5);  
...
```

entspricht:

```
...  
int result;  
{  
    int x = 5;           // Parameter zuweisen  
    int y = x * x;       // Methodenrumpf  
    result = y;          // Rückgabe Ergebnis  
}  
q = result;  
...
```



# Wirkung eines Methodenaufrufs (III)

- üblich: Methodenaufruf innerhalb eines Ausdrucks
- Beispiel:

```
int x = 5;  
int wert = square(x) + x;
```

- Zurückgegebener Wert des Aufrufs `square(x)` wird zur Auswertung des Ausdrucks `square(x) + x` verwendet.
- Methoden, die keinen Wert zurückgeben, werden als alleinstehende Anweisung aufgerufen.

# Einschub: Fehlersuche (I)

- Wie geht man vor, wenn beim Programmieren ein Fehler auftritt?
- Fehlermeldungen des Übersetzers interpretieren:
  - Übersetzung des Programms bricht mit Fehlermeldung ab oder während der Ausführung tritt ein Fehler auf, der zum Abbruch und zu einer Fehlermeldung führt.
  - Fehlermeldungen der Übersetzers genau lesen: oft enthalten solche Meldungen sehr konkrete Hinweise zum Fehler (Zeilennummer im Programm, Art des Fehlers).
  - Wenn mehrere Fehlermeldungen auftreten, diese der Reihe nach bearbeiten, denn manchmal sind weitere Meldungen Folgefehler.

# Einschub: Fehlersuche (II)

- Logische Fehler finden:
  - Programm wird übersetzt und ausgeführt, Ergebnis ist aber nicht das erwartete.
  - Einfache Strategie:
    - An verschiedenen Stellen im Programm jeweils aktuelle Variablenwerte per Ausgabeanweisung auf der Standardausgabe ausgeben.
    - Prüfen, bis zu welcher Stelle diese den Erwartungen entsprechen.
    - Nicht vergessen, diese Ausgabenanweisungen nach dem Test wieder zu entfernen.
  - Komfortabler:
    - Sog. **Debugger** einer Programmierumgebung einsetzen.
    - Debugger ermöglicht i. d. R. das Setzen von sog. **Haltepunkten**, an denen die Ausführung unterbrochen wird.
    - Danach schrittweise Ausführung und Inspektion von Variablenwerten möglich.

# Einschub: Fehlersuche (III)

- Testwerkzeuge verwenden  
(z. B. JUnit, [www.junit.org](http://www.junit.org))
  - Grundidee von JUnit:
    - Programmierer/-in spezifiziert Menge von Testfällen
    - JUnit stellt Testrahmen zur Verfügung:  
Grundidee: Testfall wird in eine Klasse (s. Objektorientierung) gekleidet und erhält einheitliche Schnittstelle zur Ausführung.
    - JUnit liefert die fehlgeschlagenen Testfälle (in textueller oder grafischer Anzeige)
  - Verallgemeinerung: xUnit
    - auch für C (cUnit), C++ (cppUnit), .NET (xUnit.net), Smalltalk (SUnit), Datenbanken (DBUnit), ...

# Sichtbarkeitsbereich von Deklarationen

- Methoden sind voneinander unabhängige Unterprogramme, in denen somit auch Variablen deklariert werden können.
- Darf eine Variable mit gleichem Namen in zwei unterschiedlichen Unterprogrammen verwendet werden?

Ja, in unterschiedlichen Blöcken dürfen Variablen gleichen Namens deklariert werden.

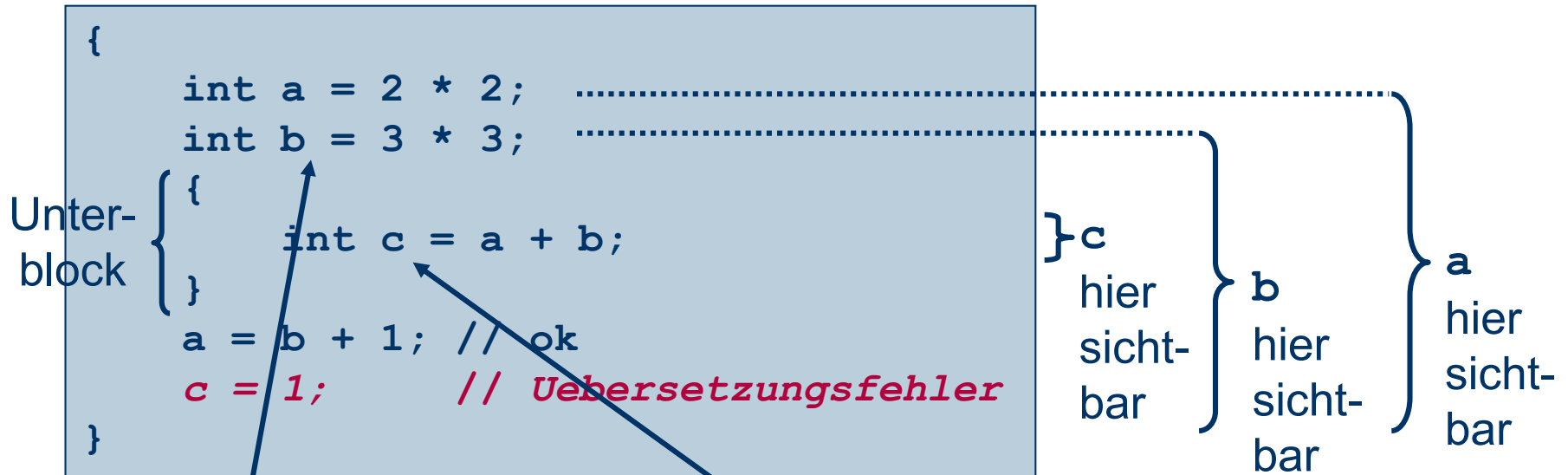
- Warum ist das so?

Variable ist (nur) in dem Block, in dem sie deklariert wurde, sowie in allen **Unterblöcken** (weiterer Block, der zwischen den geschweiften Klammern eines Blocks auftritt) dieses Blocks sichtbar und damit verwendbar.

- sog. **Sichtbarkeitsbereich** (engl: **Scope**) der Deklaration: Abschnitte eines Programms, in dem die Variable benutzt werden darf
- Variable heißt **lokal** bezüglich des Blocks, in dem sie deklariert wurde.

# Sichtbarkeitsbereich von Deklarationen

- Beispiel für geschachtelte Blöcke mit Sichtbarkeitsbereichen:



Variablen **a** und **b** können im Unterblock verwendet werden, da sie im umgebenden Block deklariert wurde.

Variable **c** kann nicht im umgebenden Block verwendet werden, Versuch würde zu Fehlermeldung führen.



# Rechnerinterne Verwaltung lokaler Variablen mittels Stapel

- Jedes Programm besitzt zur Laufzeit einen sogenannten **(Programm-) Stapel** (engl. **(program) stack**), auf dem lokale Variablen bzgl. Blöcken oder Methoden gespeichert werden.
- Beim Betreten eines Blocks bzw. beim Aufruf einer Methode werden die zugehörigen lokalen Variablen auf dem Stapel erzeugt und konzeptuell in der Reihenfolge ihrer Erzeugung nacheinander auf den Stapel gelegt.
- Beim Verlassen des Blocks bzw. beim Beenden des Aufrufs werden die lokalen Variablen in umgekehrter Reihenfolge gelöscht und vom Stapel entfernt.
- All das geschieht automatisch.

# Lokale Variablen in Blöcken (I)

①

```
{  
  int a = 2 * 2;  
  int b = 3 * 3;  
  {  
    int c = a + b;  
  }  
  a = b + 1; //okay  
  c = 1; //Ue-Fehler  
}
```

a → 4

②

```
{  
  int a = 2 * 2;  
  int b = 3 * 3;  
  {  
    int c = a + b;  
  }  
  a = b + 1; //okay  
  c = 1; //Ue-Fehler  
}
```

b → 9  
a → 4

③

```
{  
  int a = 2 * 2;  
  int b = 3 * 3;  
  {  
    int c = a + b;  
  }  
  a = b + 1; //okay  
  c = 1; //Ue-Fehler  
}
```

c → 13

b → 9

a → 4

④

```
{  
  int a = 2 * 2;  
  int b = 3 * 3;  
  {  
    int c = a + b;  
  }  
  a = b + 1; //okay  
  c = 1; //Ue-Fehler  
}
```

b → 9

a → 10

# Lokale Variablen in Blöcken (II)

⑤

```
{  
    int a = 2 * 2;  
    int b = 3 * 3;  
    {  
        int c = a + b;  
    }  
    a = b + 1; //okay  
    c = 1; //Ue-Fehler  
}
```

b → 9  
a → 10

- Problem: c liegt nicht mehr auf dem Stapel

- *in Java in dieser Form nicht erlaubt* (aber in anderen Sprachen):

```
int a = 1;  
int b = 2;  
{  
    int a = 2; // jetzt gaebe es zwei Variablen "a"  
    ...  
}
```

# Lokale Variablen von Methoden (I)

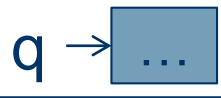
①

## Deklaration

```
static int square(int x) {  
    int y = x * x;  
    return y;  
}
```

## Aufruf

```
...  
q = square(5);  
...
```



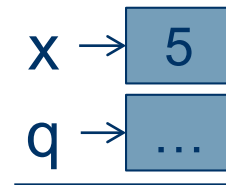
②

## Deklaration

```
static int square(int x) {  
    int y = x * x;  
    return y;  
}
```

## Aufruf

```
...  
q = square(5);  
...
```



# Lokale Variablen von Methoden (II)

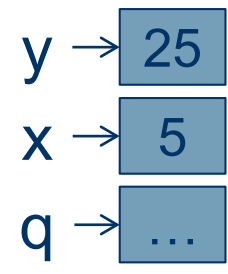
③

Deklaration

```
static int square(int x) {  
    int y = x * x;  
    return y;  
}
```

Aufruf

```
...  
q = square(5);  
...
```



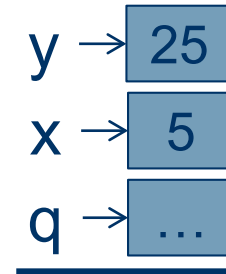
④

Deklaration

```
static int square(int x) {  
    int y = x * x;  
    return y;  
}
```

Aufruf

```
...  
q = square(5);  
...
```



# Lokale Variablen von Methoden (III)

⑤

## Deklaration

```
static int square(int x) {  
    int y = x * x;  
    return y;  
}
```

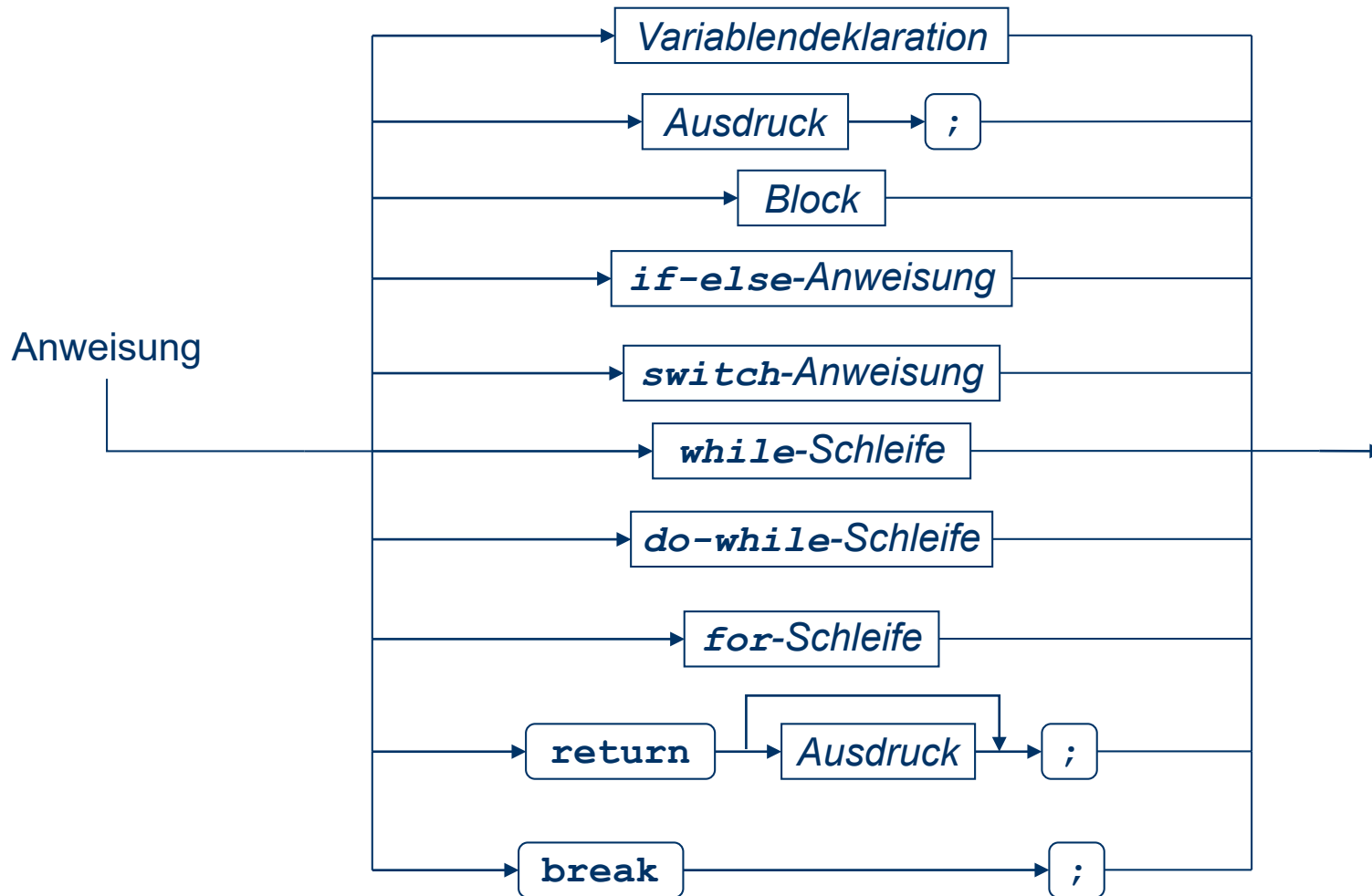
## Aufruf

```
...  
q = square(5);  
...
```

tatsächlich ist Stapel noch etwas komplexer aufgebaut, enthält z. B. Rücksprungziele etc.

q → 25

# Zwischenbilanz (I)



# Zwischenbilanz (II)

- Wir kennen nun beinahe alle wesentlichen Grundkonstrukte, die für die Erstellung von Programmen benötigt werden:
  - Variablen primitiver und einfacher, zusammengesetzter Datentypen
  - Operatoren und Ausdrücke
  - Ablaufstrukturen und Methoden
- Uns fehlt noch:
  - **Rekursion**: dabei geht es um Algorithmen, die zur Bestimmung eines Ergebnisses sich selbst verwenden.
  - **Objektorientierung**: dabei geht es – vereinfachend gesagt – um komplexe, aus den primitiven Datentypen zusammengesetzte Objekte und deren Verknüpfung mit zugehörigen Operationen.



# Abschließendes Beispiel: Spiel „Schiff versenken“

## ■ Spielidee

- Vereinfachte Version von „Schiffe versenken“: Nur ein Schiff.
- Spieler gibt Koordinatenpaar an, auf das geschossen werden soll.
- Computer antwortet mit „Treffer“ oder „Wasser“.
- Wurde das ganze Schiff versenkt, gewinnt der Spieler.
- Bei zu vielen Fehlversuchen verliert der Spieler.

## ■ Datenspeicherung

- Spielfeld
- Anzahl der Treffer und Fehlversuche

10	~	~	~	~	~	~	~	~	~	~
9	~	~	~	~	~	~	~	~	~	~
8	~	~	~	~	~	~	~	~	~	~
7	~	~	~	~	~	~	~	~	~	~
6	~	~	~	X	~	~	~	~	~	~
5	~	~	~	X	~	~	~	~	~	~
4	~	~	~	X	~	~	~	~	~	~
3	~	~	~	X	~	~	~	~	~	~
2	~	~	~	~	~	~	~	~	~	~
1	~	~	~	~	~	~	~	~	~	~
	A	B	C	D	E	F	G	H	I	J

# Schiff versenken: Konstanten und Initialisierung (I)

```
public class SchiffVersenken {  
    final static int SPIELFELD_MAX = 10;  
    final static int SCHIFF_LAENGE = 4;  
    final static int VERSUCHE = 5;  
  
    final static char SCHIFF = 'X';  
  
    static char[][] generiereFeld() {  
        // Zweidimensionales Array erstellen  
        char[][] feld = new char[SPIELFELD_MAX][SPIELFELD_MAX];  
  
        // Feld mit Wasser-Symbol füllen  
        for (int y = 0; y < SPIELFELD_MAX; y++) {  
            for (int x = 0; x < SPIELFELD_MAX; x++) {  
                feld[y][x] = '~';  
            }  
        }  
        // ...  
    }  
}
```

Spielfeldgröße

Länge eines  
Schiffes

Symbol für ein Schiff:  
Als Konstante, da  
mehrmals im Code  
verwendet

# Schiff versenken: Konstanten und Initialisierung (II)

```
// Zufaelliche Startposition des Schiffes bestimmen
```

```
int xPos = zufallsKoordinate();
```

```
int yPos = zufallsKoordinate();
```

```
boolean senkrecht = muenzWurf();
```

Wählt zufällig  
eine passende  
Koordinate

```
// Schiff einzeichnen
```

```
for (int i = 0; i < SCHIFF_LAENGE; i++) {
```

```
    feld[yPos][xPos] = SCHIFF;
```

```
    if (senkrecht) {
```

```
        yPos++;
```

```
    } else {
```

```
        xPos++;
```

```
    }
```

```
}
```

```
return feld;
```

Zufällig true  
oder false

Liegt das Schiff senkrecht, muss die  
y-Koordinate erhöht werden, sonst  
die x-Koordinate

```
} // Ende generiereFeld()
```

# Schiff versenken: Zufallsmethoden

```
static boolean muenzWurf() {  
    return Math.random() < 0.5;  
}
```

Zufällige `double`-Zahl  
aus dem Bereich [0;1)

```
static int zufallsKoordinate() {  
    return (int) (Math.random() * (SPIELFELD_MAX - SCHIFF_LAENGE));  
}
```

Verhindert  
„Herausragen“  
des Schiffes aus  
dem Spielfeld

# Schiff versenken: Textuelle Ausgabe

```
static void feldAusgeben(char[][] feld, boolean schiffZeigen) {
    for (int y = SPIELFELD_MAX - 1; y >= 0; y--) {
        if (y + 1 < 10) {
            System.out.print(" ");
        }
        System.out.print((y+1) + " ");
        for (int x = 0; x < SPIELFELD_MAX; x++) {
            char s = feld[y][x];
            // Schiffe verstecken, wenn ohne Loesung
            if ((s == SCHIFF) && (! schiffZeigen)) {
                s = '~'; // Wasser anzeigen
            }
            System.out.print(s + " ");
        }
        System.out.println();
    }
    System.out.println("    A B C D E F G H I J");
}
```

Index 0 soll in der untersten Zeile dargestellt werden, Ausgabe muss aber von oben nach unten erfolgen.

Zwei Arten der Ausgabe: Schiff anzeigen oder verstecken

# Schiff versenken: `main` (I)

```
public static void main(String[] args) {  
    int versuche = VERSUCHE;  
    int treffer = 0;  
    char[][] spielfeld = generiereFeld();
```

```
    do {  
        feldAusgeben(spielfeld, false);
```

Spieler darf  
mindestens  
1x schießen  
→ do-while-  
Schleife

```
        // Koordinaten lesen und umrechnen  
        System.out.print("Schuss-Koordinaten: ");  
        java.util.Scanner input = new java.util.Scanner(System.in);  
        int xKoord = input.next().charAt(0) - 'A';  
        int yKoord = input.nextInt() - 1;
```

Eingabe auf 0-basierte  
Indizes umrechnen.

A → 0, B → 1, ...

```
        // Auf Arraygrenzen testen  
        if (xKoord < 0 || xKoord >= SPIELFELD_MAX  
            || yKoord < 0 || yKoord >= SPIELFELD_MAX) {  
            System.out.println("Fehlerhafte Koordinaten!");  
        } else {  
            // ...
```

Falscheingabe soll  
nicht zu ungültigem  
Arrayzugriff führen

# Schiff versenken: `main` (II)

```
// mit korrekte Koordinaten:
// Treffer?
if (spielfeld[yKoord][xKoord] == SCHIFF) {
    treffer++;
    spielfeld[yKoord][xKoord] = 'x';
    System.out.println("Treffer!");
} else {
    versuche--;
    spielfeld[yKoord][xKoord] = 'o';
    System.out.println("Wasser!");
}

}

} while ((treffer < SCHIFF_LAENGE) && (versuche > 0));

if (treffer == SCHIFF_LAENGE) {
    System.out.println("Schiff versenkt!");
} else {
    System.out.println("Leider verloren.");
}
feldAusgeben(spielfeld, true);

}
```

Getroffene  
Stelle markieren

Spieler darf nochmals  
schießen, wenn noch  
nicht alle Teile getroffen  
wurden und noch  
Versuche übrig sind.

Bei Spielende:  
Schiff aufdecken

# Schiff versenken: Beispiel

```
$ java SchiffVersenken  
(...)
```

```
Schuss-Koordinaten: A 1  
Wasser!
```

10	~	~	~	~	~	~	~	~	~	~
9	~	~	~	~	~	~	~	~	~	~
8	~	~	~	~	~	~	~	~	~	~
7	~	~	~	~	~	~	~	~	~	~
6	~	~	~	~	~	~	~	~	~	~
5	~	~	~	~	~	~	~	~	~	~
4	~	~	~	~	~	~	~	~	~	~
3	~	~	~	~	~	~	~	~	~	~
2	~	~	~	~	~	~	~	~	~	~
1	○	~	~	~	~	~	~	~	~	~
	A	B	C	D	E	F	G	H	I	J

```
Schuss-Koordinaten: F 9  
Wasser!
```

10	~	~	~	~	~	~	~	~	~	~
9	~	~	~	~	○	~	~	~	~	~
8	~	~	~	~	~	~	~	~	~	~
7	~	~	~	~	~	~	~	~	~	~
6	~	~	~	~	~	~	~	~	~	~
5	~	~	~	~	~	~	~	~	~	~
4	~	~	~	~	~	~	~	~	~	~
3	~	~	~	~	~	~	~	~	~	~
2	~	~	~	~	~	~	~	~	~	~
1	○	~	~	~	~	~	~	~	~	~
	A	B	C	D	E	F	G	H	I	J

```
Schuss-Koordinaten: B 4  
Treffer!
```

10	~	~	~	~	~	~	~	~	~	~
9	~	~	~	~	○	~	~	~	~	~
8	~	~	~	~	~	~	~	~	~	~
7	~	~	~	~	~	~	~	~	~	~
6	~	~	~	~	~	~	~	~	~	~
5	~	~	~	~	~	~	~	~	~	~
4	~	x	~	~	~	~	~	~	~	~
3	~	~	~	~	~	~	~	~	~	~
2	~	~	~	~	~	~	~	~	~	~
1	○	~	~	~	~	~	~	~	~	~
	A	B	C	D	E	F	G	H	I	J

```
Schuss-Koordinaten: B 5  
Wasser!
```

10	~	~	~	~	~	~	~	~	~	~
9	~	~	~	~	○	~	~	~	~	~
8	~	~	~	~	~	~	~	~	~	~
7	~	~	~	~	~	~	~	~	~	~
6	~	~	~	~	~	~	~	~	~	~
5	~	○	~	~	~	~	~	~	~	~
4	~	x	~	~	~	~	~	~	~	~
3	~	~	~	~	~	~	~	~	~	~
2	~	~	~	~	~	~	~	~	~	~
1	○	~	~	~	~	~	~	~	~	~
	A	B	C	D	E	F	G	H	I	J

```
Schuss-Koordinaten: B 3  
Wasser!
```

10	~	~	~	~	~	~	~	~	~	~
9	~	~	~	~	○	~	~	~	~	~
8	~	~	~	~	~	~	~	~	~	~
7	~	~	~	~	~	~	~	~	~	~
6	~	~	~	~	~	~	~	~	~	~
5	~	○	~	~	~	~	~	~	~	~
4	~	x	~	~	~	~	~	~	~	~
3	~	○	~	~	~	~	~	~	~	~
2	~	~	~	~	~	~	~	~	~	~
1	○	~	~	~	~	~	~	~	~	~
	A	B	C	D	E	F	G	H	I	J

```
Schuss-Koordinaten: J 9  
Wasser!
```

Leider verloren.

10	~	~	~	~	~	~	~	~	~	~
9	~	~	~	~	○	~	~	~	○	~
8	~	~	~	~	~	~	~	~	~	~
7	~	~	~	~	~	~	~	~	~	~
6	~	~	~	~	~	~	~	~	~	~
5	~	○	~	~	~	~	~	~	~	~
4	x	x	x	x	~	~	~	~	~	~
3	~	○	~	~	~	~	~	~	~	~
2	~	~	~	~	~	~	~	~	~	~
1	○	~	~	~	~	~	~	~	~	~
	A	B	C	D	E	F	G	H	I	J