

# Programmieren I

## Kontrollstrukturen



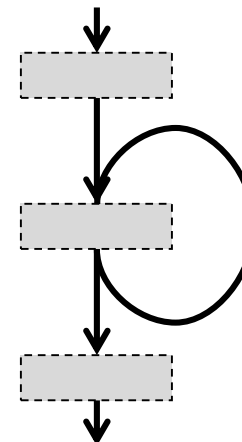
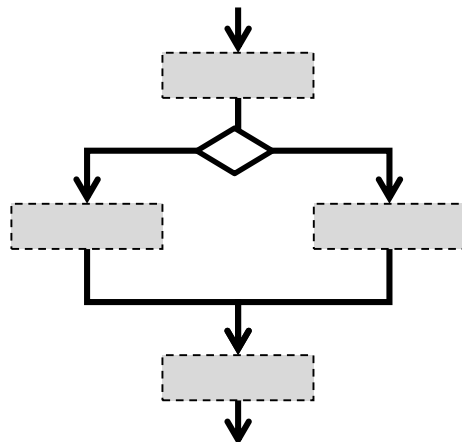
Heusch 8  
Ratz 4.5

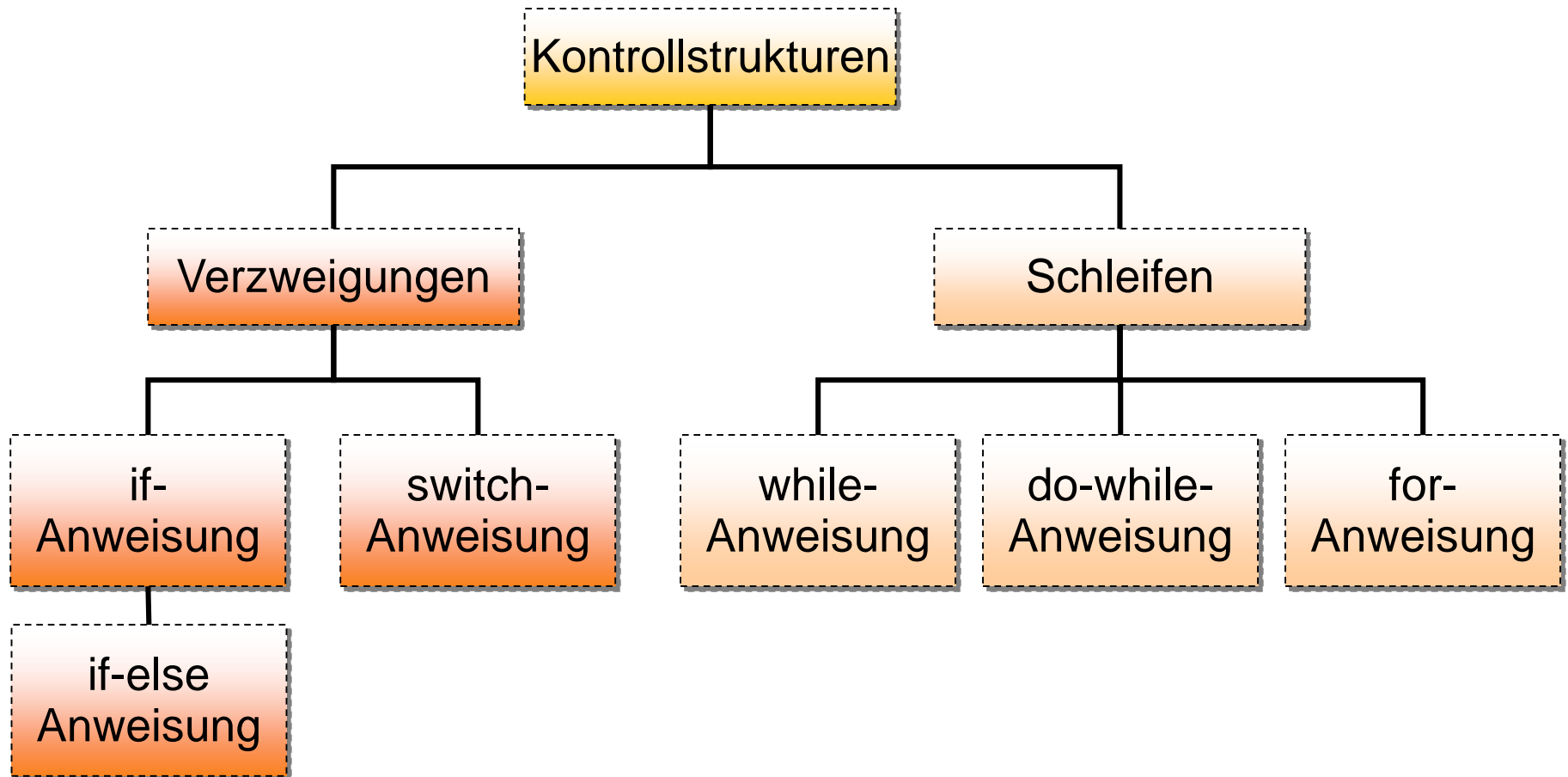
Institut für Automation und angewandte Informatik

```
final List<String> allResults = new ArrayList<String>();  
final Map<String, Integer> typeWordResultCount = new HashMap<String, Integer>();  
final Map<String, Integer> typePoints = new HashMap<String, Integer>();  
evaluation.put(type, typePoints);  
  
for (final Sheet sheet : this.sheets) {  
    final String sheetResult = sheet.getPlayerInput(type);  
    if (sheetResult.startsWith(start) && this.isValidWord(sheetResult, type)) {  
        validWordCountForType++;  
        allResults.add(sheetResult);  
    }  
}
```

# Arten von Kontrollstrukturen

- Neben der Sequenz (Aneinanderreihung von Anweisungen) werden zwei Arten von Kontrollstrukturen unterschieden:
  - Verzweigungen (entspricht „bedingten Anweisungen“ bei Algorithmen)
    - Es gibt alternative Programmteile, in die, abhängig von einer Bedingung, beim Programmablauf verzweigt wird.
  - Schleifen (engl. loops) (entspricht Wiederholungsanweisungen)
    - Ein Programmteil kann, abhängig von einer Bedingung, mehrmals durchlaufen werden.

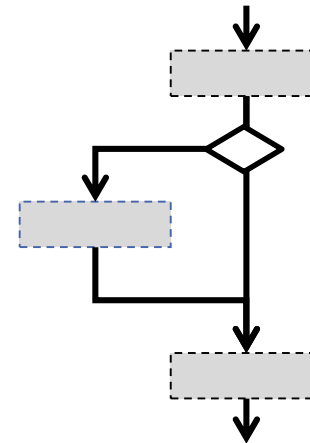




# if-Anweisung

- Die (einfache) `if`-Anweisung besteht aus dem Schlüsselwort `if`, dem zwingend ein Ausdruck mit dem Typ `boolean` in Klammern folgt. Es folgt eine Anweisung oder ein Anweisungsblock. Syntax:

```
if ( expression )  
    statement(_block)
```



- Ist das Ergebnis des Ausdrucks *expression* wahr (`true`), so wird die Anweisung ausgeführt. Ist das Ergebnis des Ausdrucks falsch (`false`), so wird mit der ersten Anweisung nach der `if`-Anweisung (bzw. dem Block) fortgefahren.

# if-Anweisung (Beispiele)

## ■ Beispiele:

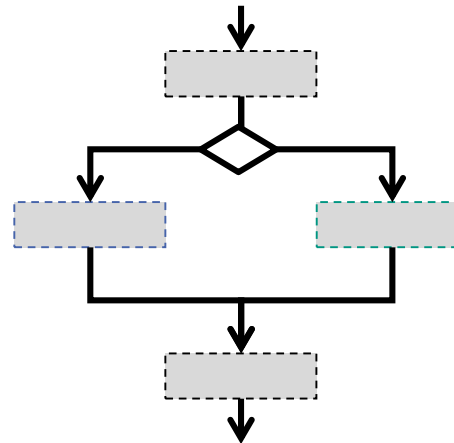
```
// if gefolgt von einer einzelnen Anweisung
if (x < y)
    System.out.println("x is lower than y");

// if gefolgt von einem Anweisungsblock
if (y == 0 || y > 10000) {
    x = 0;
    System.out.println("x is now zero");
}
```

# if-else-Anweisung

- Neben der *einfachen* bedingten Anweisung existiert die *vollständige* bedingten Anweisung. Hier steht zusätzlich hinter dem Schlüsselwort `else` eine alternative Anweisung. Syntax:

```
if ( expression )  
    statement1  
else  
    statement2
```



- Falls der Ausdruck *expression* wahr ist, wird *statement1* ausgeführt, andernfalls *statement2*. Es wird also in jedem Fall eine Anweisung ausgeführt.

# if-else-Anweisung (Beispiele)

## ■ Beispiele:

```
if (x < y)
    System.out.println("x is lower than y");
else
    System.out.println("x ist greater than or equal to y");

// if gefolgt von einem Anweisungsblock
if (x < y) {
    // tausche x und y
    int a = x;
    x = y;
    y = a;
}
else
    System.out.println("x is already greater than or equal to y");
```

# Schachtelung von if-else-Anweisungen (1)

- Bei Verzweigungen mit `else` gibt es ein Problem, welches *Dangling-Else-Problem* genannt wird. Zu welcher Anweisung gehört das folgende `else`?

```
if (expression1 )  
    if (expression2 )  
        statement1;  
else  
    statement2;
```

- Die Einrückung suggeriert, dass das `else` die Alternative zur ersten `if`-Anweisung ist. Dies ist aber nicht richtig. Die Semantik von Java ist so definiert, dass das `else` zum innersten `if` gehört, das nicht abgeschlossen ist.



## Schachtelung von if-else-Anweisungen (2)

- Wenn das `else` zum äußeren `if` gehören soll, kann Klammerung Abhilfe schaffen:

Klammerung des inneren `if`

```
if ( expression1 ) {  
    if ( expression2 )  
        statement1;  
}  
else  
    statement2;
```



Klammerung aller `if`-Zweige

```
if ( expression1 ) {  
    if ( expression2 ) {  
        statement1;  
    }  
}  
else {  
    statement2;  
}
```

- Die innere `if`-Anweisung wird in einen Block eingeschlossen und damit abgeschlossen.

# "Das böse Semikolon" (1)

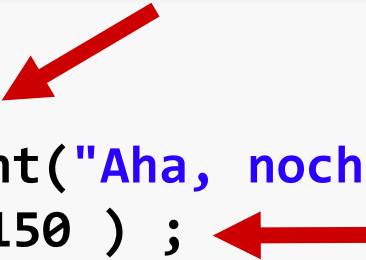
```
int alter = ...;  
if ( alter < 0 ) ;  
    System.out.print("Aha, noch im Mutterleib");  
  
if ( alter > 150 ) ;  
    System.out.println("Aha, ein neuer Abraham");
```



- Die Semikola führen dazu, dass bei Erfüllung der Bedingungen jeweils eine leere Anweisung ausgeführt wird.
- Unabhängig vom Wert der Variablen `alter` werden immer die beiden Ausgaben »Aha, noch im Mutterleib« und »Aha, ein neuer Abraham« erzeugt!

## "Das böse Semikolon" (2)

```
int alter = ...;  
if ( alter < 0 ) ;  
    System.out.print("Aha, noch im Mutterleib");  
else if ( alter > 150 ) ;  
    System.out.println("Aha, ein neuer Abraham");  
else  
    System.out.println("Aha, im richtigen Alter");
```



- Dies führt zu der Compiler-Fehlermeldung:  
'else' without 'if'.

# Bedingungsoperator (Wdh.)

- Ermittlung des Maximums von a und b:

```
int a = 3, b = 5, max;  
max = (a > b) ? a : b;
```

- Analoge Schreibweise mit `if` und `else`:

```
if ( a > b )  
    max = a;  
else  
    max = b;
```

# Mehrfachverzweigungen (1)

- Mit `if`-Verzweigungen alleine kann man durch Aufzählen aller Fälle\* grundsätzlich auskommen. Zum Beispiel kann man die drei Fälle für das Vorzeichen (Signum) einer ganzen Zahl abfragen:

```
if ( x > 0 ) signum = 1;  
if ( x == 0 ) signum = 0;  
if ( x < 0 ) signum = -1;
```

- Dieser Ansatz kostet Rechenzeit, da in jedem Fall drei Bedingungen geprüft werden, und ist sehr umständlich.

\* Es können aber leicht sehr viele Fälle entstehen!

## Mehrfachverzweigungen (2)

- In einer kleinen Programmverbesserung schachteln wir daher die Alternativen und arbeiten dann mit einer Abfolge von *sequenziell abhängigen* Alternativen:

```
if ( x > 0 )  
    signum = 1;  
else if ( x < 0 )  
    signum = -1;  
else  
    signum = 0;
```

# Programmieraufgabe: „quadratische Gleichung“ (1)

- Gegeben sei eine quadratische Gleichung  $a x^2 + b x + c = 0$  für die Unbekannte  $x$  mit beliebigen reellen Koeffizienten  $a$ ,  $b$  und  $c$ .
- Schreiben Sie eine Java-Applikation `Quadratics`, welche die Koeffizienten  $a$ ,  $b$  und  $c$  einliest und die Lösungen der quadratischen Gleichung für beliebige Koeffizienten bestimmt und ausgibt.
- Unterscheiden Sie bei der Lösung der quadratischen Gleichung folgende Fälle und geben Sie, je nach Fall, die angegebene Lösung bzw. Fehlermeldung aus:

## Koeffizienten

## Lösung(en) bzw. Fehlermeldung

1.  $a = 0$

1.1  $b = 0$

“Die Gleichung ist degeneriert.”

1.2  $b \neq 0$

$$x = -\frac{c}{b}$$

2.  $a \neq 0$

2.1  $D \geq 0$

$$x_{1,2} = \frac{-b \pm \sqrt{D}}{2a}$$

2.2  $D < 0$

“Die Lösung ist konjugiert komplex.”

mit  $D = b^2 - 4 a c$  (Diskriminante der Gleichung)

# Programmieraufgabe: „quadratische Gleichung“ (2)

- Hinweis: Die Quadratwurzel einer Zahl  $d$  können Sie wie folgt bestimmen:

`Math.sqrt(d)`

- Testdaten:

<u>Koeffizienten</u>			<u>Lösung(en) bzw. Fehlermeldung</u>	
<u>a</u>	<u>b</u>	<u>c</u>	<u>x1</u>	<u>x2</u>
-2.0	0.0	2.0	-1.0	1.0
1.0	2.0	1.0	-1.0	-1.0
0.0	1.0	1.0	-1.0	
1.0	0.0	1.0	“Die Lösung ist konjugiert komplex.“	
0.0	0.0	1.0	“Die Gleichung ist degeneriert.“	



# Die switch-Anweisung (1)

- Die `switch`-Anweisung bietet in vielen Fällen eine *einfache Form* der Mehrfachverzweigung. Syntax:

```
switch ( expression ) {  
    case constant1: [ statement(s)1 ] [ break; ]  
    case constant2: [ statement(s)2 ] [ break; ]  
    ...  
    [ default:  
        statement(s) ]  
}
```

- Der Wert des Ausdrucks *expression* wird bestimmt und nacheinander mit jedem einzelnen Fallwert (*constant<sub>x</sub>*) verglichen.  
Stimmt der Wert des Ausdrucks mit einem Fallwert überein, so wird die Anweisung (ggf. mehrere) hinter dem ":" ausgeführt.

## Die switch-Anweisung (2)

- Zulässige Datentypen des Ausdrucks und der Fallwerte sind `byte`, `short`, `int` und `char` (ganze Zahlen und Zeichen, außer `long`) sowie `enum` (Aufzählungstypen), seit Java 1.7 auch `String`.
- Alle Fallwerte (*constant $x$* ) müssen unterschiedlich sein.
- `break`; beendet die `switch`-Anweisung.
- Für den Fall, dass der Wert des Ausdrucks mit keinem Fallwert übereinstimmt, lässt sich optional die Sprungmarke `default` einsetzen.



**Heusch 8.2.2**  
**Ratz 4.5.3.2, 21.1.1.3**

# Beispiel mit Datentyp char

```
public class Switch {  
    public static void main(String args[]) {  
        for (char c = 65; c < 91; c++) { // entspr. 'A'/'Z'  
            switch (c) {  
                case 'A':  
                case 'E':  
                case 'I':  
                case 'O':  
                case 'U':  
                    System.out.println(c + " ist ein Vokal");  
                    break;  
                case 'X':  
                    System.out.println("Das " + c  
                        + " ist mein Lieblingsbuchstabe");  
                    break;  
                default:  
                    System.out.println(c + " ist ein Konsonant");  
            }  
        }  
    }  
}
```

# Beispiel mit enum

```
public enum Month { JAN, /*...*/ NOV, DEC };
```

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

```
    Month m = Month.NOV;
```

```
    int monthNumber = 0;
```

*Zugriffe normalerweise  
immer mit enum-Typenname*

```
    switch (m) {
```

```
        case JAN: monthNumber = 1; break;
```

```
        // ...
```

```
        case NOV: monthNumber = 11; break;
```

```
        case DEC: monthNumber = 12; break;
```

```
        default: monthNumber = 0; break;
```

```
    }
```

```
}
```

*Ausnahme: case-Klausel*

**i** Ab Java 1.5

**i** 2. Semester

# Beispiel mit Datentyp String

```
String month = "November";  
int monthNumber = 0;  
  
switch (month.toLowerCase()) {  
    case "january": monthNumber = 1; break;  
    // ...  
    case "november": monthNumber = 11; break;  
    case "december": monthNumber = 12; break;  
    default: monthNumber = 0; break;  
}
```

# while-Schleife

- Die `while`-Schleife ist eine „*abweisende*“ Schleife, da sie vor jedem Schleifeneintritt die Schleifenbedingung prüft:

```
while ( expression )  
    statement
```

- Der Typ des Ausdrucks *expression* muss `boolean` sein.
- Vor jedem Schleifendurchgang wird der Ausdruck ausgewertet bzw. erneut ausgewertet.  
Ist das Ergebnis `true`, so wird der Rumpf ausgeführt und anschließend der Ausdruck erneut geprüft.  
Ist das Ergebnis `false`, wird die Schleife beendet.
- Ist die Bedingung schon vor dem ersten Eintritt in den Rumpf nicht wahr, so wird der Rumpf gar nicht durchlaufen.

# Beispiele zur while-Schleife

```
int i = 1;
int a = 0;
while (i < 10) {
    a = a + i;
    i = i + 1;
}
System.out.println("i: " + i + ", a: " + a);
```

? Ausgabe ?

```
int i=1;
while ( i < 101 ){
    System.out.println(i);
    i++;
}
```

? Ausgabe ?



**Heusch 8.2.3**  
**Ratz 4.5.4.3**

# Vorsicht bei Gleitpunktzahlen!

```
double d = 0.0;
while ( d != 1.0 ) {
    d += 0.1;
    System.out.println( d );
}
```

Ausgabe

```
0.1
0.2
0.30000000000000004
0.4
0.5
0.6
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
1.0999999999999999
1.2
1.3    // ... bis das Auge müde wird ...
```



# do-while-Schleife

- Dieser Schleifentyp ist eine "*annehmende*" Schleife. Die Schleifenbedingung wird erst *nach* jedem Schleifendurchgang geprüft. Syntax:

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

- Beachte das Semikolon!
- Der Rumpf wird durchlaufen. Liefert der Ausdruck *expression* dann `true`, so wird der Rumpf erneut ausgeführt. Andernfalls wird die Schleife beendet, und das Programm wird mit der nächsten Anweisung nach der Schleife fortgesetzt.
- Bevor es zum ersten Test kommt, wurde der Rumpf also schon einmal durchlaufen.

# Beispiele zur do-while-Schleife

```
int i = 1;           // int i = 10;  
do {  
    System.out.println(i);  
    i++;  
} while (i < 10);
```

? Ausgabe ?

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

? Ausgabe ?

# Die for-Schleife

```
for ([init]; [loop_test]; [step])  
    statement(_block)
```

Beispiel:

```
for (int i = 1; i < 10; i++)  
    System.out.println( i );
```

- *init*: Initialisierung der Schleife
  - Wird zu Beginn genau *einmal* ausgeführt.
  - Kann eine lokale Variable deklarieren und initialisieren.  
Diese ist dann außerhalb der `for`-Anweisung nicht mehr gültig.
- *loop\_test*: Schleifenbedingung
  - Der Ausdruck *loop\_test* muss vom Typ `boolean` sein.
  - Er wird *vor* dem Durchlaufen des *Schleifenrumpfs* *statement* – also vor jedem Schleifeneintritt – ausgewertet.
  - Ergibt der Ausdruck `true`, wird der Schleifenrumpf durchlaufen.  
Bei `false` wird er nicht durchlaufen und die `for`-Anweisung beendet.
  - Ist kein *loop\_test* angegeben, so ist das Ergebnis automatisch `true`.
- *step*: Schleifen-Inkrement
  - Wird *am Ende* jedes Schleifendurchlaufs, aber noch vor dem nächsten Test der Schleifenbedingung ausgeführt.

## Noch ein Beispiel

```
public class C357 {  
  
    /*  
     * Gibt alle Zahlen von 1..200 aus,  
     * die durch 3, 5 oder 7 teilbar sind  
     */  
    public static void main(String args[]) {  
        for (int i = 1; i < 201; i++)  
            if (i % 3 == 0 || i % 5 == 0 || i % 7 == 0)  
                System.out.println(i +  
                    " ist durch 3, 5 oder 7 teilbar");  
    }  
}
```

# Break und Continue

## ■ break

- Beendigung der *Schleife* und Fortfahren mit der nächsten Anweisung nach der Schleife

```
int i=0;
while (true) {
    if (i++ > 100)
        break;
    System.out.println("i=" + i);
}
```

## ■ continue

- Abbruch *des aktuellen Schleifendurchlaufs*, Sprung ans Schleifenende und Fortfahren mit dem nächsten Schleifendurchlauf

```
int i=0;
while (i <= 100) {
    if ( (i++ % 2) == 0 )
        continue;
    System.out.println(i);
}
```



**Heusch 8.2.6**  
**Ratz 4.5.5**