

## Kapitel 4: Anweisungen und Kontrollstrukturen

Prof. Dr. David Sabel

Lehr- und Forschungseinheit für Theoretische Informatik  
Institut für Informatik, LMU München

WS 2018/19

Stand der Folien: 7. November 2018

Die Inhalte dieser Folien basieren – mit freundlicher Genehmigung – tlw. auf Folien von Prof. Dr. Rolf Hennicker aus dem WS 2017/18 und auf Folien von PD Dr. Ulrich Schöpp aus dem WS 2010/11



### Anweisungen

- **Anweisungen (Statements, Befehle)** sind die Grundelemente imperativer Programmiersprachen zur Festlegung (Kontrolle) des Ablaufs eines Programms.
- Wir unterscheiden folgende **grundlegende Arten** von Anweisungen:

#### Syntax:

Statement = VariableDeclaration (Deklarationsanweisung)  
 | Assignment (Zuweisung)  
 | Block (Block)  
 | Conditional (Fallunterscheidung)  
 | Iteration (Wiederholungsanweisung)

- Durch Ausführung einer Anweisung wird ein „alter“ **Zustand** in einen „neuen“ **Zustand** überführt (**Semantik!**)

### Ziele

- **Kontrollstrukturen** in **imperativen** Programmen kennenlernen und verstehen
- Kontrollstrukturen:
  - Variablendeklarationen
  - Zuweisungen
  - Blöcke
  - Fallunterscheidung (Verzweigung)
  - Schleifen
- **Syntax** und **Semantik**
- **Realisierung** der Kontrollstrukturen in **Java**

### Deklarationsanweisungen

Statement = VariableDeclaration (Deklarationsanweisung)  
 | Assignment (Zuweisung)  
 | Block (Block)  
 | Conditional (Fallunterscheidung)  
 | Iteration (Wiederholungsanweisung)

## Deklarationsanweisungen (2)

Eine Deklarationsanweisung nennt man auch **lokale Variablendefinition**

**Syntax** (Wiederholung):

```
VariableDeclaration =  
    Type VariableDeclarator { ", " VariableDeclarator } ";"  
VariableDeclarator = NamedVariable [ "=" Expression ]
```

**Beispiel:** `int x = 5, y = 7;`

**Wirkung** (Semantik):

Es wird ein **Speicherplatz angelegt**, auf den mit dem symbolischen **Namen** der deklarierten **Variablen** zugegriffen werden kann.

Dort wird ein „Default“-Wert oder der durch Expression bestimmte Initialwert gespeichert. Z.B. [(x, 5), (y, 7)].

## Zuweisungen

|                                 |                          |
|---------------------------------|--------------------------|
| Statement = VariableDeclaration | (Deklarationsanweisung)  |
| Assignment                      | (Zuweisung)              |
| Block                           | (Block)                  |
| Conditional                     | (Fallunterscheidung)     |
| Iteration                       | (Wiederholungsanweisung) |

## Deklarationsanweisungen (3)

**Beachte:**

Bereits deklarierte Variablen **dürfen nicht noch einmal** deklariert werden!

**Beispiel:**

```
1 public class TestVarDeclTwice {  
2     public static void main(String[] args) {  
3         int x = 7;  
4         int x = 8;  
5         System.out.println(x);  
6     }  
7 }
```

```
javac TestVarDeclTwice.java  
TestVarDeclTwice.java:4: error: variable x is already  
                        defined in method main(String[])  
    int x = 8;  
        ^  
1 error
```

## Zuweisungen (2)

**Syntax:**

Assignment = Variable "=" Expression ";"

**Wirkung:**

- 1 Zunächst wird der Wert von Expression im „alten“ Zustand berechnet.
- 2 Dieser Wert wird im Nachfolgezustand der Variablen Variable als neuer Wert zugewiesen.

**Beispiel:** [(x, 5), (y, 7)] `x = 2*x+y`  $= 2 \cdot x + y$ ; [(x, 17), (y, 7)]  
 $2 \cdot 5 + 7 = 17$

**Beachte** den Unterschied zwischen „=" (Zuweisung) und „==" (Vergleich)!

**Nebenbedingung für die Zuweisung** Variable "=" Expression ";" :

- Der Typ der Variablen Variable muss mit dem Typ des Ausdrucks Expression verträglich sein.
- Die Variable Variable muss vorher deklariert sein.

### Abkürzungen:

$x++$ ;                   steht für  $x = x + 1$ ;  
 $x--$ ;                   steht für  $x = x - 1$ ;  
 $x \text{ operation} = \text{Ausdruck}$ ;   steht für  $x = x \text{ operation} \text{ Ausdruck}$ ;

### Beispiele:

$x += y$ ;       steht für  $x = x + y$ ;  
 $b \&= c$ ;       steht für  $b = b \& c$ ;  
 $x += 3 * y$ ;   steht für  $x = x + (3 * y)$ ;

## Blöcke (2)

Ein **Block** fügt **mehrere Anweisungen** durch geschweifte Klammern **zu einer einzigen Anweisung zusammen**.

### Syntax:

Block = "{ { Statement } }"

### Wirkung:

- Die Anweisungen im Block werden in der Reihenfolge der Aufschreibung hintereinander ausgeführt.
- Der durch einen Block bewirkte Zustandsübergang erfolgt also durch Hintereinanderausführung der Zustandsübergänge der einzelnen Anweisungen.

Statement = VariableDeclaration (Deklarationsanweisung)  
          | Assignment (Zuweisung)  
          | **Block** (Block)  
          | Conditional (Fallunterscheidung)  
          | Iteration (Wiederholungsanweisung)

## Blöcke (3)

Beispiel:

$[(x, 5), (y, 7)] \{x = 2*x+y; y = x-2;\} [(x, 17), (y, 15)]$

denn:

$[(x, 5), (y, 7)] \quad x = 2*x+y; \quad [(x, 17), (y, 7)]$

und  $[(x, 17), (y, 7)] \quad y = x-2; \quad [(x, 17), (y, 15)]$

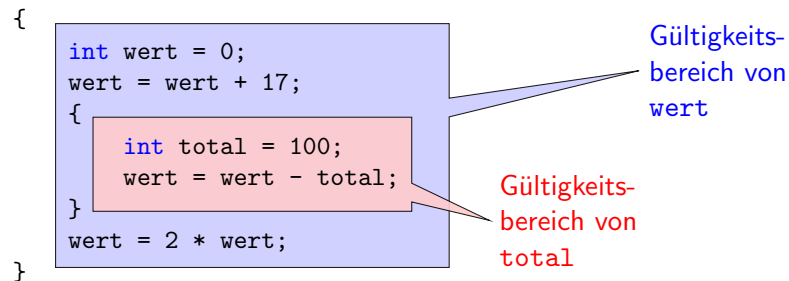
### Allgemein:

wenn  $\sigma_i$  Statement  $_{i+1}$   $\sigma_{i+1}$  für  $i = 0, \dots, n-1$ ,  
dann  $\sigma_0 \{ \text{Statement}_1; \dots, \text{Statement}_n \} \sigma_n$

## Gültigkeitsbereich

- Der **Gültigkeitsbereich** einer lokalen Variablen ist der Block, in dem die Variable deklariert wurde. Außerhalb dieses Blocks **existiert die Variable nicht**.
- Blöcke können geschachtelt werden.
- In einem untergeordneten Block sind Variable eines übergeordneten Blocks gültig und dürfen dort **nicht noch einmal** deklariert werden.

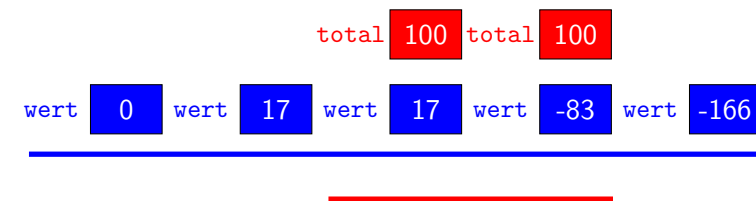
### Beispiel:



## Veränderung des Speichers

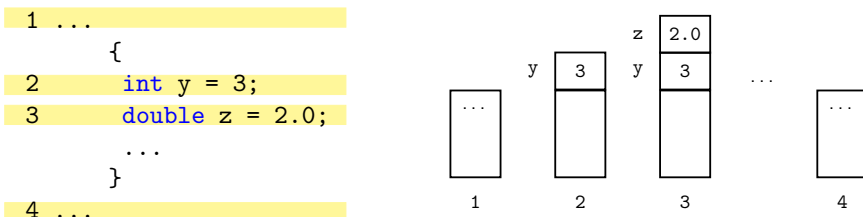
```

{
  int wert = 0;
  wert = wert + 17;
  {
    int total = 100;
    wert = wert - total;
  }
  wert = 2 * wert;
}
    
```



## Lokale Variablen im Speicher

- Lokale Variablen werden **stapelartig** im Speicher abgelegt.
- Wird eine **Variablendeklaration** abgearbeitet, so wird ein **neuer Speicherplatz** für diese Variable **oben auf den Stapel** gelegt (allokiert).
- Am Ende eines Blocks werden alle Variablen (von oben) vom Stapel **entfernt**, die in diesem Block deklariert wurden.

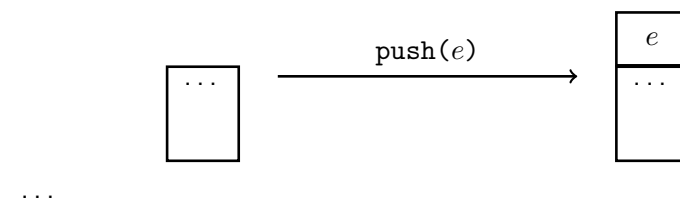


## Exkurs: Stack (auch Keller, Stapelspeicher) (2)

- Ein **Stack** ist eine Datenstruktur, in die Elemente eingefügt und in entgegengesetzter Reihenfolge wieder herausgenommen werden können.

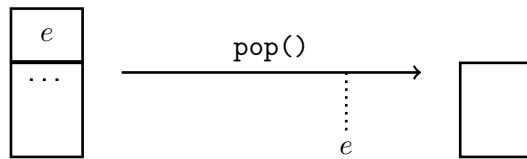
LIFO = **Last In, First Out**

- Grundoperationen:
  - **push(*e*)**: legt das Element *e* oben auf den Stapel



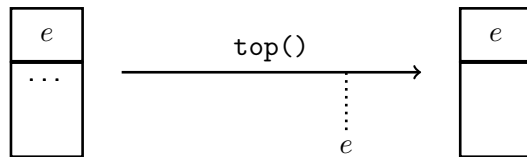
## Exkurs: Stack (auch Keller, Stapelspeicher)

- `pop()`: entfernt das oberste Element (und liefert es als Ergebnis)

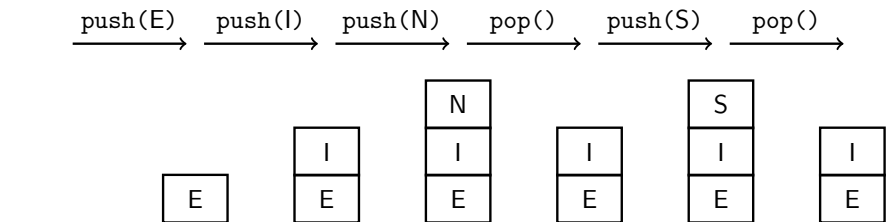


...

- `top()`: liefert das oberste Element als Ergebnis, ohne den Stapel zu verändern



## Stacks: Beispiel



## Fallunterscheidungen (Bedingte Anweisung)

Statement = VariableDeclaration (Deklarationsanweisung)  
 | Assignment (Zuweisung)  
 | Block (Block)  
 | **Conditional** (**Fallunterscheidung**)  
 | Iteration (Wiederholungsanweisung)

## Fallunterscheidungen (Bedingte Anweisung) (2)

### Syntax:

Conditional = IfStatement | SwitchStatement

IfStatement = "if" "(" Expression ")" Statement [ "else" Statement ]

### Beispiel

```
if (x >= 0) y = x;
else y = -x;
```

### Nebenbedingung:

Der Type des Ausdrucks Expression muss boolean sein.

### Wirkung:

- Wenn die Auswertung von Expression im aktuellen Zustand den Wert `true` ergibt, wird das **erste Statement** ausgeführt.
- Wenn die Auswertung von Expression im aktuellen Zustand den Wert `false` ergibt **und** ein `else`-Zweig vorhanden ist, wird das **zweite Statement** ausgeführt.

## Fallunterscheidungen (Bedingte Anweisung) (3)

Formaler: (4 Fälle wegen true/false und mit else/ohne else)

- Wenn Expression im Zustand  $\sigma$  zu true ausgewertet, und  $\sigma$  Statement<sub>1</sub>  $\sigma'$ , dann  $\sigma$  if (Expression) Statement<sub>1</sub> else Statement<sub>2</sub>  $\sigma'$
- Wenn Expression im Zustand  $\sigma$  zu false ausgewertet, und  $\sigma$  Statement<sub>2</sub>  $\sigma'$ , dann  $\sigma$  if (Expression) Statement<sub>1</sub> else Statement<sub>2</sub>  $\sigma'$
- Wenn Expression im Zustand  $\sigma$  zu true ausgewertet, und  $\sigma$  Statement  $\sigma'$ , dann  $\sigma$  if (Expression) Statement  $\sigma'$
- Wenn Expression im Zustand  $\sigma$  zu false ausgewertet, dann  $\sigma$  if (Expression) Statement  $\sigma$

## Was ist hier semantisch falsch?

```
if (kontostand >= betrag)
    kontostand = kontostand - betrag - abhebe_gebuehr
else
    kontostand = kontostand - betrag;
    kontostand = kontostand - abhebe_gebuehr
    - ueberzieh_gebuehr;
```

Die letzte Zeile wird immer ausgeführt! Sie gehört nicht mehr zum else-Zweig.

Richtig ist: **Blockbildung!**

```
if (kontostand >= betrag)
    kontostand = kontostand - betrag - abhebe_gebuehr
else {
    kontostand = kontostand - betrag;
    kontostand = kontostand - abhebe_gebuehr
    - ueberzieh_gebuehr;
}
```

## Beispiele

Beispiel 1:

```
if (kontostand >= betrag)
    kontostand = kontostand
        - betrag
        - abhebe_gebuehr;
```

Beispiel 2:

```
if (kontostand >= betrag)
    kontostand = kontostand
        - betrag
        - abhebe_gebuehr;
else
    kontostand = kontostand
        - betrag
        - abhebe_gebuehr
        - ueberzieh_gebuehr;
```

## Dangling else (1)

```
if (!kontoGesperrt)
    if (kontostand >= betrag) {
        kontostand = kontostand - betrag -
            abhebe_gebuehr;
        System.out.println("Abhebung erfolgreich");
    }
else
    System.out.println("Abhebung nicht erlaubt");
```

**Vorsicht!**

Das else bezieht sich auf das **zweite** if und wird nicht ausgeführt, wenn das Konto gesperrt ist. (Dann ist die gesamte if-Anweisung ohne Wirkung).

Die Einrückung entspricht daher nicht der Semantik! (else sollte eingerückt sein)

## Dangling else (2)

```
if (!kontoGesperrt)
  if (kontostand >= betrag) {
    kontostand = kontostand - betrag -
      abhebe_gebuehr;
    System.out.println("Abhebung erfolgreich");
  }
  else
    System.out.println("Abhebung nicht erlaubt, da
      Konto nicht gedeckt.");
else
  System.out.println("Abhebung nicht erlaubt, da
    Konto gesperrt.");
```

## Vergleich: Bedingungsoperator vs. Bedingte Anweisung

Statement = ... | Conditional | ...

Conditional = IfStatement | ...

IfStatement = "if" "(" Expression ")" Statement ["else" Statement]  
vs.

Expression = ... | Expression "?" Expression ":" Expression

### Gemeinsamkeiten:

- Boolescher Ausdruck bestimmt wie die Verzweigung fortgesetzt wird

### Unterschiede:

- if-Anweisung ist eine **Anweisung**, während der Bedingungsoperator zum Formen von **Ausdrücken** verwendet wird
- else Statement ist **optional**, aber ":" Expression ist **zwingend erforderlich**

## Vergleich: Bedingungsoperator vs. Bedingte Anweisung (2)

Beispiel mit Bedingungsoperator:

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

Äquivalent dazu mit bedingter Anweisung:

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

**Falsch**, da if keine Ausdrücke, sondern Anweisungen formt:

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

**Falsch**, da nach ? und : ein Ausdruck steht und keine Anweisung

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

## Wiederholungsanweisungen

|                                 |                          |
|---------------------------------|--------------------------|
| Statement = VariableDeclaration | (Deklarationsanweisung)  |
| Assignment                      | (Zuweisung)              |
| Block                           | (Block)                  |
| Conditional                     | (Fallunterscheidung)     |
| Iteration                       | (Wiederholungsanweisung) |

## Wiederholungsanweisungen (Iterationen, Schleifen)

Wir unterscheiden 3 Arten von Wiederholungsanweisungen:

### Syntax:

```
Iteration = WhileStatement
           | ForStatement
           | DoStatement
```

Mit den dann zur Verfügung stehenden Anweisungen (insbesondere `while`-Anweisungen) können **alle berechenbaren Funktionen** programmiert werden!

## While-Anweisungen

### Syntax:

```
WhileStatement = "while" "(" Expression ")" Statement
```

### Beispiel:

```
while (i <= 100) {
    s = s+i;
    i = i + 1; // oder i++;
}
```

### Nebenbedingung:

Der Typ des Ausdrucks `Expression` muss boolean sein.

### Wirkung:

Solange die Auswertung von `Expression` den Wert `true` ergibt, wird die Anweisung `Statement` ausgeführt.

## While-Anweisungen (2)

### Formaler:

- Falls die Auswertung von `Expression` im Zustand  $\sigma$  den Wert `true` ergibt,  $\sigma$  `Statement`  $\sigma'$  und  $\sigma'$  `while Expression Statement`  $\sigma''$ , dann  $\sigma$  `while Expression Statement`  $\sigma''$
- Falls die Auswertung von `Expression` im Zustand  $\sigma$  den Wert `false` ergibt, dann  $\sigma$  `while Expression Statement`  $\sigma$

## While-Anweisungen: Beispiele

Beispiel: Zahlen von 1 bis 10 ausdrucken

```
public class WhileExampleNumbers {
    public static void main(String[] args) {
        int n = 1; // Iterator
        int end = 10;
        while (n <= end) {
            System.out.println(n);
            n++;
        }
    }
}
```



## While-Anweisungen: Beispiele (2)

Beispiel: Quersumme einer Zahl berechnen

```
public class DigitSum {
    public static void main(String[] args) {
        int x = 352;
        int digitSum = 0; // Akkumulator
        while (x > 0) {
            digitSum += x % 10;
            x = x/10;
        }
        System.out.println(digitSum);
    }
}
```

## Beispiel: Fakultät einer Zahl

Fakultät  $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$  für  $n \geq 1$  und  $0! = 1$

Entwurf eines Programms: while-Anweisung:

- 1 Variablen und Anfangswerte:  
Verwende Variablen count zum Zählen (initial 1), factorial für das Ergebnis (initial 1), und n für die Zahl n.
- 2 Schleifenbedingung: Zähler ist noch nicht größer als n, d.h.  $\text{count} \leq n$ .
- 3 Schleifenrumpf: Multipliziere Ergebnis mit dem Zähler und erhöhe den Zähler:  

```
factorial = factorial * count;
count++;
```
- 4 Terminiert die Schleife? Ja, denn count wird in jedem Durchlauf erhöht und n verändert sich nicht.

## Entwurf von While-Schleifen

### Methodische Richtlinien

- 1 Bestimmung der **Anfangswerte** der Variablen **vor** Eintritt in die While-Anweisung
- 2 Bestimmung der **Schleifenbedingung**
- 3 Formulierung des **Schleifenrumpfes**
- 4 Prüfen, dass die Schleifenbedingung nach **endlich vielen** Ausführungen des Rumpfes nicht mehr erfüllt ist.  
**Anderenfalls terminiert die While-Anweisung nicht!**

```
int x = 352;
int digitSum = 0;
while (x > 0) { // Terminiert NICHT!
    digitSum += x % 10;
}
```

## Beispiel: Fakultät einer Zahl (2)

```
public class Factorial {
    public static void main(String[] args) {
        int n = 5;
        int count = 1;
        int factorial = 1;
        while (count <= n) {
            factorial = factorial * count;
            count++;
        }
        System.out.println(factorial);
    }
}
```

## For-Anweisungen

- Häufige Form einer While-Schleife ist:

```
int i = start;        // Initialisierung einer
                    // Iteratorvariablen
while (i <= end) {   // Grenze fuer den Iterator
    ...
    i++; // konstante Aenderung des Iterators (hier +1)
}
```

- Abkürzende Schreibweise durch eine For-Anweisung:

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

## For-Schleife: Beispiele

```
public class FactorialWithFor {
    public static void main(String[] args) {
        int n = 5;
        int factorial = 1;
        for (int count=1; count <= n; count++) {
            factorial = factorial * count;
        }
        System.out.println(factorial);
    }
}
```

## For-Schleife: Beispiele (2)

```
public class FactorialWithForDownwards {
    public static void main(String[] args) {
        int n = 5;
        int factorial = 1;
        for (int count=n; count > 0; count--) {
            factorial = factorial * count;
        }
        System.out.println(factorial);
    }
}
```

## For-Schleife: Beispiele (3)

Summe der ersten 100 Zahlen:

```
public class Sum {
    public static void main(String[] args) {
        int summe = 0; // Akkumulator
        int factorial = 1;
        for (int i =0; i <= 100; i++) {
            summe = summe + i;
        }
        System.out.println("Ergebnis = " + summe);
    }
}
```

## For-Schleife: Beispiel (4)

---

Im Schleifenrumpf einer For-Anweisung kann selbst wieder eine For-Anweisung stehen. Man spricht von geschachteltem For, bzw. allgemein von geschachtelten Schleifen.

Beispiel:

```
public class SternchenDrucken {
    public static void main(String[] args) {
        int n = 5;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < i; j++) {
                System.out.print('*');
            }
            System.out.println(); // Zeilenvorschub
        }
    }
}
```

## Zusammenfassung

---

- 5 wesentliche Kontrollstrukturen der imperativen Programmierung: Variablendeklaration, Zuweisung, Block, Fallunterscheidung, Iteration
- Syntax und Semantik
- Modellierung des Zustands als Stack der momentan definierten Variablen und ihrer Werte
- Realisierung der Kontrollstrukturen in Java