Kapitel 4

Kontrollstrukturen

Ziele

- Kontrollstrukturen in imperativen Programmen kennenlernen und verstehen.
- Realisierung der Kontrollstrukturen in Java.

Anweisungen

- **Anweisungen (Statements)** 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

(*Statement* wird später erweitert)

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

σ Anweisung σ'

alter Zustand neuer Zustand

Deklarationsanweisungen

Eine Deklarationsanweisung nennt man auch **lokale Variablendeklaration**.

Syntax (Wiederholung):

VariableDeclaration =

Type VariableDeclarator {" , " *VariableDeclarator*} " ; "

VariableDeclarator = *NamedVariable* [" = " *Expression*]

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

Wirkung:

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)]`

$\sigma = [\,]$ int $x = 5, y = 7;$ $\sigma' = [(x,5),(y,7)]$

Zuweisung

Zuweisungen

x = x + 2;

Syntax:

Assignment = *Variable* " = " *Expression* " ; "

x == x + 2

Ausdruck vom Typ boolean

Wirkung:

1. Bei der **Zuweisung** 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;` `[(x,17),(y,7)]`

σ σ'

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

int x = 3;
x = true;

Nebenbedingung:

(1) Der Typ der Variablen muss mit dem Typ des Ausdrucks verträglich sein.
(2) Die Variable muss vorher deklariert sein.

Zuweisung: Abkürzende Schreibweisen

Abkürzungen

`x++;` steht für `x = x + 1;`

`x--;` steht für `x = x - 1;`

`x` *`operation`*`= Ausdruck;` steht für `x = x` *`operation 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);`

Block

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

Wiederholung im EBNF

Terminalsymbol im Java

Syntax:

Block = " { " { *Statement* } " } "

Wirkung:

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

Zwischenzustand

Beispiel:

```
[(x,5),(y,7)] x = 2*x+y; [(x,17),(y,7)] y = x-2; [(x,17),(y,15)]
```

Also: `[(x,5),(y,7)] {x = 2*x+y; y = x-2;} [(x,17),(y,15)]`

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:

1.
```
{
    int wert = 0;
    wert = wert + 17;
1.1 {
        int wert = 0;
        int total = 100;
        wert = wert - total;
    }
    wert = 2 * wert;
}
```

Gültigkeitsbereich von `total`

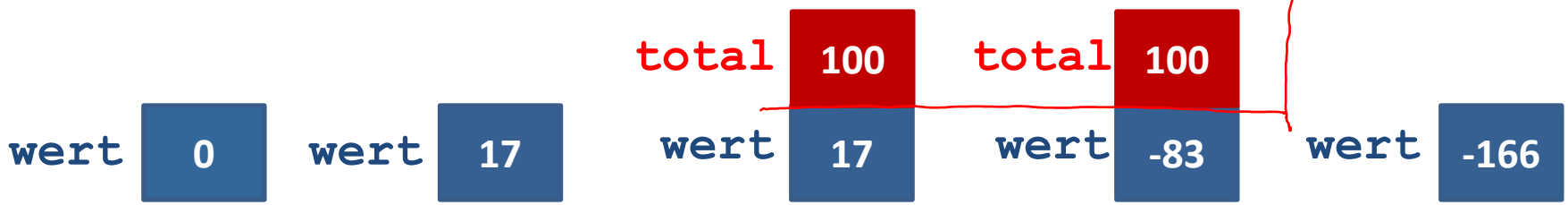
Gültigkeitsbereich von `wert`

Veränderung des Speichers

1.

1.1

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



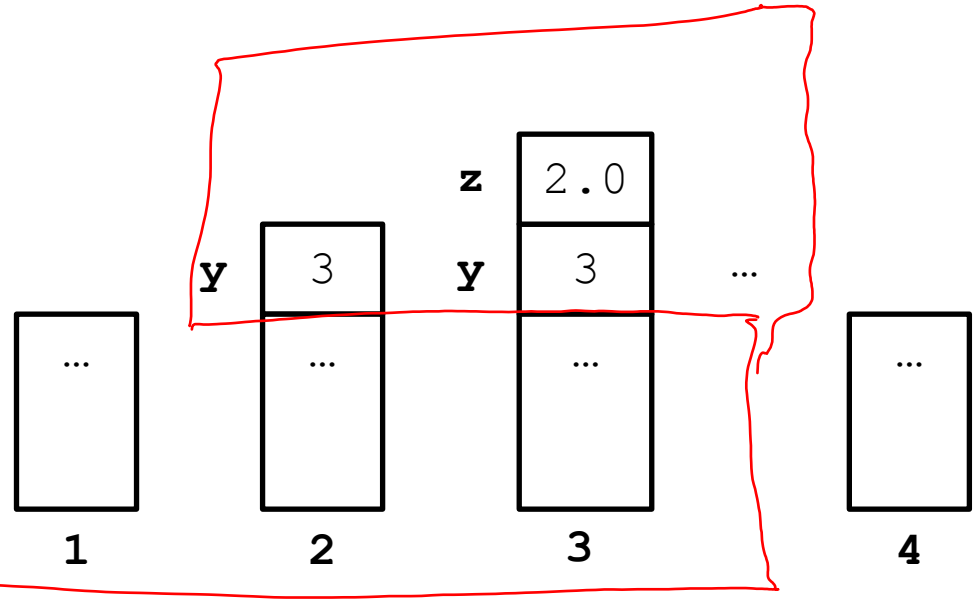
1.

1.1

Lokale Variablen im Speicher

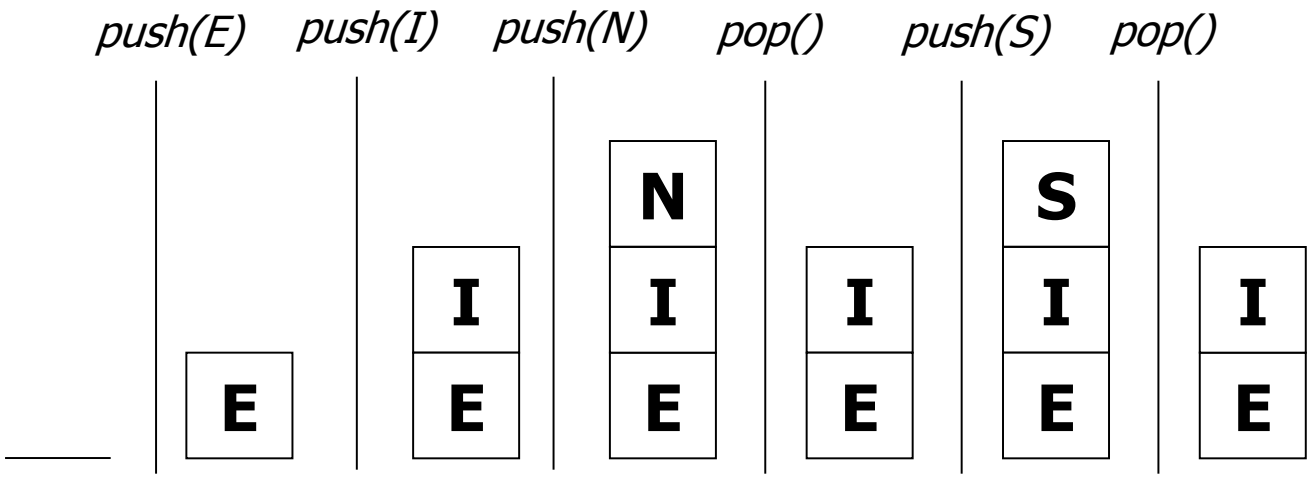
- Lokale Variablen werden **stapelartig** im Speicher abgelegt.
- Wird eine Variablendeklaration abgearbeitet, so wird ein neuer Speicherplatz für diese Variable oben auf den Stapel gelegt (allokiert).
- Am Ende eines Blocks werden alle Variablen (von oben) vom Stapel genommen, die in diesem Block deklariert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

```
1 ...
2   { int y = 3;
3     double z = 2.0;
      ...
    }
4 ...
```



Exkurs: Stack (auch Keller, Stapelspeicher)

- Ein **Stack** ist eine Datenstruktur, in die Elemente eingefügt und in entgegen gesetzter Reihenfolge wieder herausgenommen werden können.
  (LIFO — Last In, First Out)
- Grundoperationen
  - *push(e)* — legt das Element *e* oben auf den Stapel
  - *pop()* — entfernt das oberste Element vom Stapel (und liefert es als Ergebnis)
  - *top()* — liefert das oberste Element als Ergebnis, ohne den Stapel zu verändern



Fallunterscheidungen (Bedingte Anweisungen)

Syntax:

Conditional = *IfStatement* | *SwitchStatement* (wird nicht behandelt)

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

Beispiel:

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

$y = |x|$

Nebenbedingung:

Der Typ 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.

$(x >= 0) = \ldots$

Fallunterscheidungen: 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);
```

Was ist hier semantisch falsch?

```
if (kontoStand >= betrag)
    kontoStand = kontoStand - betrag - abhebe_gebuehr;
else
    gebuehren = (abhebe_gebuehr + ueberzieh_gebuehr);
    kontoStand = kontoStand - betrag - gebuehren;
```

Der Betrag und die Gebühren werden hier immer vom letzten Kontostand abgezogen. Falls `(kontoStand >= betrag)` ist dies ein zusätzlicher Abzug. Richtig ist:

```
if (kontoStand >= betrag)
    kontoStand = kontoStand - betrag - abhebe_gebuehr;
else {
    gebuehren = (abhebe_gebuehr + ueberzieh_gebuehr);
    kontoStand = kontoStand - betrag - gebuehren;
}
```

Blockbildung ist nötig, wenn ein Fall mehrere Anweisungen umschließen soll!

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 hat die Anweisung keine Wirkung.)

Um dies deutlich zu machen, hätte man das `else` in der Formatierung einrücken sollen.

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.");
```

Wenn das Konto gesperrt ist, passiert nichts und es geht nach der if-Anweisung weiter.

Dangling else (3)

```
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.");
```

Wiederholungsanweisungen (Iterationen)

Wir unterscheiden 3 Arten von Wiederholungsanweisungen:

<u>Syntax:</u>

Iteration =
WhileStatement
| *ForStatement*
| *DoStatement* (wird nicht behandelt)

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

While-Anweisungen

Syntax:

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

Beispiel:

int s = 0; Akkumulator
int i = 1;

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

Iterator

berechnet $\sum_{i=1}^{100} i$

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: Beispiele

Beispiel: Zahlen von 1 bis 10 ausdrucken

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

Beispiel: Quersumme einer Zahl x berechnen:

```
{ int x = 352;
  int qs = 0;
  while (x > 0) {
    qs = qs + x % 10;
    x = x / 10;
  }
}
```

Eingabe

Akkumulator

qs	0	2	7	10
x	352	35	3	0

Methodische Richtlinien

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

```
{ int x = 352;
  int qs = 0;
  while (x > 0) {
    qs = qs + x % 10;
  }
}
```

Terminiert nicht für `x > 0` !

For-Anweisungen

- Häufige Form einer Schleifenanweisung ist:

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

- Abkürzende Schreibweise durch eine For-Anweisung:

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

Block

Bevorzugte Form von For-Anweisungen

```
for (Deklaration der Iteratorvariablen mit Startwert ;
     Test, ob Iterator den Endwert noch nicht erreicht hat ;
     Iteratoränderung ) {
         Schleifenrumpf
}
```

- Erst wird die Deklaration und Initialisierung der Iteratorvariablen ausgeführt.
- Solange der Iterator den Endwert nicht erreicht hat, wird der Schleifenrumpf gefolgt von der Iteratoränderung ausgeführt.
- Die in der Initialisierung deklarierte Iteratorvariable ist nur im Rumpf gültig.
- Der Schleifenrumpf ist eine Anweisung, die selbst wieder eine For-Anweisung sein kann (geschachteltes „for").
- Im Schleifenrumpf soll der Iterator, der Startwert und der Endwert nicht verändert werden.