Einfache Rechenstrukturen und Kontrollfluss II

Martin Wirsing

in Zusammenarbeit mit
Moritz Hammer und Axel Rauschmayer

http://www.pst.informatik.uni-muenchen.de/lehre/SS06/infoII/

SS 06

Ziele

- Lernen imperative Programme in Java mit
  Zuweisung, seq. Komposition, Block,
  Fallunterscheidung und Iteration
  zu schreiben
- Lernen Kontrollflussdiagramme zur Veranschaulichung einzusetzen

Zuweisung (Wdh.) und sequentielle Komposition

- **Beispiel**

Deklaration der lokalen Variable `total`

```
int total = 100;
total = 2* total + 100;
```

Sequentielle Komposition

Zuweisung

- Bei der **Zuweisung**

<VarName> = <Expression>;

wird der Wert w der <Expression> im „alten“ Zustand berechnet und im Nachfolgezustand der Variablen <VarName> als neuer Wert zugewiesen.

- **Sequentielle Komposition**

wird durch Hintereinanderschreiben ausgedrückt; d.h. es gibt kein spezielles Zeichen für seq. Komposition (wie bei **BNF**).

Block

- Ein **Block**

```
{ <Statement>
}
```

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

- Durch einen Block werden **Sichtbarkeit** und **Gültigkeitsbereich** von Variablen begrenzt:

Lokale Variablen sind nur innerhalb des umfassenden Blocks gültig und sichtbar.

- In Java sind auch geschachtelte Mehrfachdeklarationen von Variablen gleichen Namens verboten: Lokale Variablen in inneren Blocks schränken die Sichtbarkeit von weiter außen definierten lokalen Variablen **nicht** ein;

d.h. sie verursachen **keine „Verschattungen“** .

Gültigkeitsbereich

- Der **Gültigkeitsbereich** einer lokalen Variablen oder Konstante ist der **die Deklaration umfassende Block**
- Außerhalb dieses Blocks existiert die Variable *nicht!*

Beispiel:

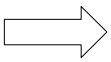
```
1. {
     int wert = 0;
     wert = wert + 17;
1.1  {
        int total = 100;
        wert = wert - total;
     }
     wert = 2 * wert;
   }
```

Block 1.1 Gült.ber. `total`

Block 1. Gült.ber. `wert`

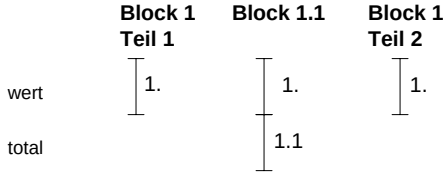
Pulsierender Speicher

- Die Menge der gültigen lokalen Variablen verändert sich **kellerartig** mit jedem Eintritt und Austritt aus einem Block:
  Bei Eintritt kommen neue Variablendeklarationen (als letzte) hinzu,
  die beim Austritt (als erste) wieder ungültig werden.



Pulsierender Speicher, implementiert durch Laufzeitkeller

Beispiel: **Abarbeitung von**



Fallunterscheidung

Die Fallunterscheidung in Java hat die Form

```
if ( <Bool Expression> ) <Statement>
```

bzw.

```
if ( <Bool Expression> ) <Statement> else <Statement>
```

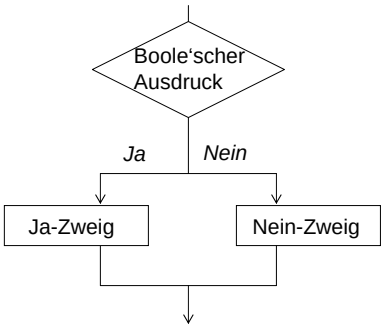
Beispiel: Abhebung von Konto 1

```
if (kontoStand >= betrag)
    kontoStand  = kontoStand - betrag;
```

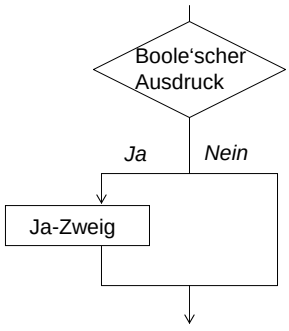
Beispiel: Abhebung von Konto 2

```
if (kontoStand >= betrag)
    kontoStand  = kontoStand - betrag;
else
    kontoStand  = kontoStand – betrag – UEBERZIEH_GEBUEHR;
```


Graphische Notation

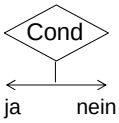


(a) „Long **if**“



(b) „Short **if**“

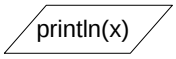
Kontrollflussdiagramme



Fallunterscheidung bzgl. der Bedingung Cond

Zuweisung

Block {x=y+1; z=y; ...}



Ein- / Ausgabe

Sequentielle Abfolge

Fehlende Begrenzungssymbole bei `if`

Java hat keine Endbegrenzung der Fallunterscheidung:

```
if (c2) S1 else S2 fi
```

Das kann zu Problemen führen bei

- Mehrfachen Anweisungen im `else`-Zweig
- Geschachtelten Fallunterscheidungen („Dangling `else`“)

Mehrfache Anweisungen im `else`-Zweig

- Blockklammern sind bei geschachtelten Fallunterscheidungen und im **`else`** Zweig sehr wichtig:
  Vergessen führt zu falschen Ergebnissen

Beispiel:

Seien `kontoStand`, `gebühren` existierende lokale Variablen (für ein Bankkontoobjekt).

```
if (kontoStand >= betrag)
{
        double neuerStand = kontoStand - betrag;
        kontoStand = neuerStand;
}
else
        kontoStand = kontoStand - betrag - UEBERZIEH_GEBUEHR;
        gebuehren  = gebuehren + UEBERZIEH_GEBUEHR;
```

Was ist falsch?

Korrektur des Beispiels

Beispiel:

```
if (kontoStand >= betrag)
{       double neuerStand = kontoStand - betrag;
        kontoStand = neuerStand;
}
else
{       kontoStand = kontoStand - betrag - UEBERZIEH_GEBUEHR;
        gebuehren  = gebuehren + UEBERZIEH_GEBUEHR;
}
```

Überziehungsgebühr nur verlangt, wenn Konto nicht gedeckt!

„Dangling `else`“

Die fehlende Endbegrenzung führt zu Mehrdeutigkeiten bei geschachtelten Fallunterscheidungen:

```
if (c1) if (c2) S1 else S2
```

Zu welchem `if` gehört der `else` Zweig?

„Dangling `else`“

- In Java sind äquivalent:

```
if (c1) if (c2) S1 else S2
```

und

```
if (c1)
        {   if (c2) S1 else S2   (Rechtsklammerung!)
        }
```

- Vermeidung der Mehrdeutigkeit in Java durch folgende Regel:

  Innerhalb einer Fallunterscheidung ist **kein „kurzes `if`“**

  (**`if`** ohne **`else`**-Zweig) erlaubt.

Mehrfache Verzweigung („switch“)

- Die Mehrfachverzweigung dient zur Auswahl aus mehreren Fällen:

Enum oder nach int konvertiebar

```
switch (Expression){
   case const1 : Statement1 break;
        . . .
   case constn : Statementn break;
   default : Statements
}
```


Mehrfache Verzweigung („switch“)

- **Beispiel Komplementärfarben:**

Sei `f` vom Typ `enum Farbe {ROT, TUERKIS, GRUEN, PURPUR, BLAU, GELB};`

```
switch (f) {
  case ROT: System.out.println(i + „hat Komplement“ + TUERKIS);
        break;
  case GRUEN: System.out.println(i + „hat Komplement“ + PURPUR);
        break;
  case BLAU: System.out.println(i + „hat Komplement“ + GELB); break;
  default : System.out.println("Komplement von " + f +
                                          " ist unbekannt.");
}
```

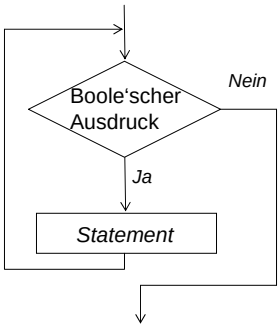
- **Achtung:**

Mit **`break`** verläßt man die **`switch`**-Anweisung, in der **`break`** steht. Ohne **`break`** wird die **`switch`**-Anweisung nur am Ende bzw. nach dem nächsten **`break`** verlassen!

Iteration

Drei Konstrukte zur Iteration:

- **`while`**
- **`for`** und erweitertes **`for`**
- **`do`**



`while`-Schleifen:

Die **`while`**-Schleife hat die Form

```
while (<Boolescher Ausdruck>)
   <Statement>
```


While-Schleifen

Beispiel 1

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

Beispiel 2

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

for- Schleifen

- Die häufigste Form der while-Schleife ist

```
int i = start;     // Initialisierung
while (i <= end)    // Bedingung
{
     ...
     i++;           // Zählerkorrektur durch
                    // konstante Änderung
}
```

- wird abgekürzt durch

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


for-Schleifen

Beispiel:

```
int end = 10;
for (int n=1; n <= end; n++)
{
   System.out.println(„tick" + n);
}
```

- Allgemein hat eine **for**-Schleife die Gestalt

  for (Initialisierung; Bedingung; Zählerkorrektur) <Statement>

- Dabei wird zunächst die Initialisierung ausgeführt.

  Dann wird, solange die Bedingung wahr ist, <Statement> ausgeführt und der Zähler geändert (gemäß der Zählerkorrektur-<expression>).

for-Schleifen

- Guter Stil ist es, **for**-Schleifen nur folgendermaßen zu schreiben:

```
for (setze counter auf start;
     pruefe, ob counter bei end;
     aendere counter)
{
     ... //counter, start, end und increment werden
         //hier nicht geaendert!
}
```

- Außerdem sollte der Zähler **counter** in der Schleifen-Initialisierung deklariert werden.

Erweitertes **for („Für jedes Element tue")**

- Die erweiterte **for**-Schleife (ab Java 5) führt den Schleifenrumpf für jedes Element der im Kopf angegebenen endlichen Menge durch; das explizite Weiterschalten der Laufvariablen ist **nicht** erforderlich.
-

```
for (<Typ> counter : endliche Menge von Elementen)
{
     ...    //counter wird hier nicht geaendert!
}
```

Beispiele Erweitertes `for`

- ```
  class Test {
  public static void main (String[] args) {
     {  for(String s : args)
           System.out.println(s);
     }}
  ```
  Aufruf `>java Test one two` ergibt
  ```
  one
  two
  ```

`values()` berechnet die Reihung (siehe später) aller Elemente von `Farbe`. Nur `Farbe` wäre nicht korrekt, da es ein Typ und keine Menge ist!

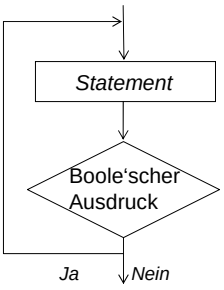
- ```
  for (Farbe f : Farbe.values())
   System.out.println(f + „ ist eine Farbe.“);
  ```


`do`-Schleifen

Die **`do`**-Schleife ist eine while-Schleife, bei der die Anweisung **mindestens einmal** aufgeführt wird. Die Bedingung wird erst nach Ausführung der Anweisung überprüft. Sie hat die Form

`do`
Statement
`while` (Boolescher Ausdruck)

Als Kontrollflußdiagramm



Zusammenfassung

- Grundlegende **imperative Konstrukte** von Java sind:
  - Deklaration lokaler Variablen, Zuweisung, sequentielle Komposition,
  - Block, Fallunterscheidung, Iteration
- Lokale Variablen müssen **vor Benutzung initialisiert** werden und werden im **Keller** gespeichert.
- Eine Fallunterscheidung erlaubt es, abhängig von einer Bedingung, verschiedene Anweisungen auszuführen.

  Mehrfachverzweigungen werden durch **`switch`** ausgedrückt.
- while-Schleifen bilden die Grundform der Iteration;

  for-Schleifen sollten verwendet werden, wenn die Schleifenvariable von einem Anfangswert bis zu einem Endwert mit einem **konstanten Inkrement** oder Dekrement läuft;

  erweiterte for-Schleifen unterstützen die Iteration über eine endliche Menge von Werten.