Imperative Programmierung in Java: Kontrollfluß II

Martin Wirsing

in Zusammenarbeit mit
Matthias Hölzl, Piotr Kosiuczenko, Dirk Pattinson

15/04/03

Ziele

- Lernen imperative Programme in Java mit
  Zuweisung, Block, Fallunterscheidung, Iteration
  zu schreiben
- Lernen Kontrollflußdiagramme zur Veranschaulichung einzusetzen

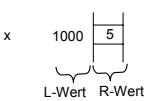
Lokale Variable (Wdhg.)

```
int x = 5;
```

„ x hat den Wert 5“

Textuelle Beschreibung
als **Paar aus Name und Wert**: `( x,5 )`

Im Speicher:



Zustand (Wdhg.)

- Ein Zustand ist eine **Belegung der Variablen mit Werten**.
- Der Zustand der lokalen Variablen wird beschrieben als **Liste von Variablennamen und zugehörigen Werten**.

Beispiel: `int x = 5, y = 7; boolean aussage=true;`
Textuell: `[(x,5), (y,7), (aussage, true)]`

Im Speicher:

x	1000	5
y	1001	7
aussage	1002	true

Deklaration lokaler Variablen und Zuweisung (Wdhg.)

Deklaration lokaler Variablen:

```
<Type>  <VarName> = <Expression>;
```

Zuweisung:

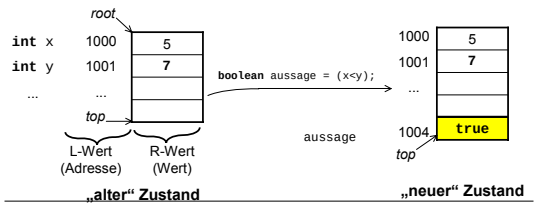
```
<VarName> = <Expression>;
```

Beispiel:

```
int x = 5, y = 7;
boolean aussage = (x<y);
x = 2*x + y;
```


Deklaration lokaler Variablen (Wdhg.)

Durch eine lokale Deklaration wird eine neue Speicherzelle für die lokale Variable reserviert und mit ihrem Initialwert belegt, (der aus dem „alten“ Zustand berechnet wird).



„alter“ Zustand **„neuer“ Zustand**

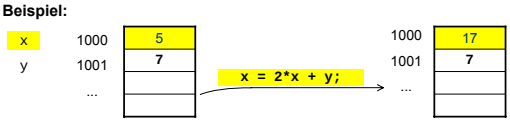
Zuweisung (Wdhg.)

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.

Beispiel:



„alter“ Zustand **„neuer“ Zustand**

Zuweisung: Textuelle Darstellung

Beispiel textuell:

„alter“ Zustand	`s = [(x, 5), (y,7), (b, true)]`
Zuweisung	`x = 2*x+y;`
„neuer“ Zustand	`s = [(x, 17), (y,7), (b, true)]`

Sequentielle Komposition

Sequentielle Komposition

wird durch Hintereinanderschreiben ausgedrückt.

Beispiel:

```
int total = 100;
total = total + 100;
```


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`

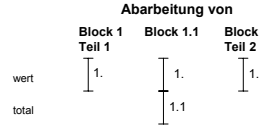
Block 1. Gült.ber. `wert`

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>
```

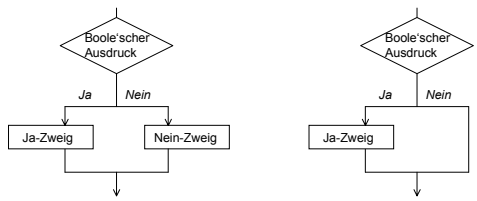
Beispiel: Kontofluß 1

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

Beispiel: Kontofluß 2

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


Graphische Notation



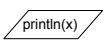
(a) „Long **if**“ (b) „Short **if**“

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 in **else**-Zweig sehr wichtig:

Vergessen führt zu falschen Ergebnissen

Beispiel:

```
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;
}
```

Überziehgsgebühr nur, 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.

Iteration

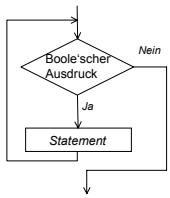
Drei Konstrukte zur Iteration:

- while
- 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;
     Test, ob counter bei end;
           aendere counter)
{
     ...//counter, start, end und increment werden
      //hier nicht geaendert!
}
```

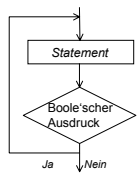
- Außerdem sollte der Zähler **counter** in der Schleifen-Initialisierung deklariert werden.

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 1

- Grundlegende **imperative Konstrukte** von Java sind:
  - Deklaration lokaler Variablen, Zuweisung, Sequ. Komposition,
  - 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.
- Zur Vermeidung des Dangling-**else**-Problems dürfen „Ja" – und „Nein"-Zweig von **if** sowie die Anweisung von **while** kein „short-**if**" sein.

Zusammenfassung 2

- Es gibt 3 Arten von Iterationen: **while**-, **for**- und **do**-Schleifen.

  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;

  do-Schleifen sind passend, wenn der Schleifenrumpf **mindestens einmal** ausgeführt werden muß.
- Kontrollflußdiagramme dienen zur graphischen Darstellung von Programmabläufen. Sie können sehr gut zur Veranschaulichung iterativer Programme eingesetzt werden.