Einführung in die Informatik

Jochen Hoenicke

Software Engineering
Albert-Ludwigs-University Freiburg

Sommersemester 2014

Variablendeklarationen

Variablendeklaration

UNI FREIBURG

Syntax

$$\textit{Deklaration} ::= \textit{Datentyp}\ \textit{Bezeichner}\ (\ ,\ \textit{Bezichner})^*\ ;$$

Eine Variablendeklaration reserviert einen Speicherbereich für ein oder mehrere Variablen. Zum Beispiel reserviert

```
int i, j;
```

Speicher für zwei Ganzzahl(Integer)-Variablen `i` und `j`.
Eine Deklaration wird wie (fast) alle Anweisungen mit einem Semikolon abgeschlossen.

Datentypen

UNI FREIBURG

Syntax

$$\textit{Deklaration} ::= \textit{Datentyp}\ \textit{Bezeichner}\ (\ ,\ \textit{Bezichner})^*\ ;$$

Datentyp kann sein:

- `byte`, `short`, `int`, `long`: Ganzzahlen mit unterschiedlichen Wertebereichen (byte: $-128 \ldots 127$, short: $-32768 \ldots 32767$, int: $-2^{31} \ldots 2^{31} - 1$, long: $-2^{63} \ldots 2^{63} - 1$).
- `char`: einzelnes Unicodezeichen. Dies wird auch wie eine Ganzzahl zwischen 0 und 65535 behandelt, z.B. `'A' == 65`.
- `boolean`: Wahrheitswerte, also `true` und `false`.
- `float`, `double`: Fließkommazahlen mit verschiedenen Wertebereichen und Genauigkeit.
- `String`: Zeichenketten
- ...: Man kann auch seine eigenen Typen bauen (später).

Zuweisungen

Zuweisungen

Syntax

$$\textit{Zuweisung} ::= \textit{LValue} = \textit{Ausdruck} ;$$

$$\textit{LValue} ::= \textit{Bezeicher} \mid \ldots$$

Eine Zuweisung weist einer Variablen (genauer: einem LValue) das Ergebnis einer Berechnung (Ausdruck) zu.
Auch eine Zuweisung wird mit einem Semikolon abgeschlossen.
Die Zuweisungen werden in der Reihenfolge ausgeführt, in der sie im Programm stehen. Es ist keine mathematische Gleichheit.

```
int x;
x = 5;       // x ist jetzt 5
x = 7;       // x ist jetzt 7
x = x + 1; // x ist jetzt 8 (7 + 1)
```

Ausdrücke

Syntax

$$
\begin{aligned}
\textit{Ausdruck} ::= & \ \textit{Ausdruck}\ \textit{Binär-Operator}\ \textit{Ausdruck} \\
| & \ \textit{Unär-OperatorAusdruck} \\
| & \ (\textit{Datentyp})\ \textit{Ausdruck} \\
| & \ (\textit{Ausdruck}) \\
\textit{Binär-Operator} ::= & \ \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{>=} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{<} \mid \ldots \\
\textit{Unär-Operator} ::= & \ \texttt{+} \mid \texttt{-} \mid \texttt{!} \mid \texttt{\~{}} \mid \texttt{++} \mid \texttt{--} \mid
\end{aligned}
$$

Ein Ausdruck (engl. Expression) ist eine Berechnung. Die Syntax ist noch größer als oben angegeben.
Man kann Ausdrücke Klammern, ansonsten gilt Punkt- vor Strichrechnung.

Beispiel: Ausdruck

```
int x;
boolean b;
x = 5 + 4 * 3;     // x ist jetzt 17
x = 2 * (x + 1); // x ist jetzt 36
x = -x + 1;        // x ist jetzt -35
b = (x > 0);       // b ist jetzt false
b = (x != 0);      // b ist jetzt true
x = x / 4;         // x ist jetzt -8 (rundet)
x = x % 3;         // x ist jetzt -2 (Rest)
```

Typkonvertierungen

Typkonvertierungen `(Datentyp)Ausdruck` wandeln einen Wert in einen anderen Typ um. Dabei kann Präzision verloren gehen oder ein Überlauf auftreten.
Beispiel:

```
short s; int i; double d;
d = 123456.789
i = (int) d;    // i ist jetzt 123456
s = (short) i; // s ist jetzt -7616
```

Typkonvertierungen passieren auch implizit aber nur, wenn keine Präzision verloren geht:

```
i = s; // okay
d = s; // okay
s = d; // Compiler-Fehler
s = i; // Compiler-Fehler
```

Operationen: Integer

- +, −, ∗: Verhalten sich (außer bei Überlauf) wie man erwarten würde.
- /: Division, rundet zu Null. Division durch Null führt zu einem Programmabsturz.
- %: Rest bei der Division.
- Unäres +,−: Wie man erwartet.
- Unäres ~: vertauscht 0 und 1 in der Binärdarstellung (not).
- &, |, ^: Logisch Und bzw. Oder bzw. Exklusiv-Oder auf der Binärdarstellung.
- <<,>>,>>>: Shiften auf der Binärdarstellung.

Operationen: Fließkommazahlen

Bei Fließkommazahlen rundet die Division nicht und es gibt auch keinen Programmabsturz beim Teilen durch 0.
Aber Vorsicht:

```
double d,e;
d = 7 / 5;      // d ist jetzt 1.0
d = 7.0 / 5.0; // d ist jetzt 1.4
```

Ob / rundet hängt davon ab, ob mindestens ein Operand eine Fließkommazahlen ist.
Integer werden automatisch bei Bedarf nach Fließkommazahlen konvertiert.
Umgekehrt muss man eine explizite Typkonvertierung verwenden.

```
int i;
double d;
i = 7;          // i ist jetzt 7
d = i;          // d ist jetzt 7.0
d = d / 5;      // d ist jetzt 1.4
i = (int) d; // i ist jetzt 1
```

Operator + und Zeichenketten

Der Operator + ist für ganze Zahlen, Fließkommazahlen und Zeichenketten definiert.

1. Ist einer der Operanden eine Zeichenkette, so wird der andere automatisch in eine Zeichenkette konvertiert und die Zeichenketten aneinandergehängt.
2. Ist einer der Operanden eine Fließkommazahl, so werden Fließkommazahlen addiert.
3. Ist einer der Operanden vom Type `long` so wird mit dem Typ `long` addiert.
4. Ansonsten wird mit dem Typ `int` addiert.

Zahlen können nicht mit `(String)i` in Zeichenketten konvertiert werden. Stattdessen kann man die Funktion `String.valueOf(i)` verwenden oder `"" + i`.

Integer, Short, Byte

Die Arithmetischen Operationen gibt es nicht für `byte` und `short`. Es wird implizit auf `int` konvertiert:

```
short s1, s2;
int i;
i = s1 + s2;  // implizite Konvert. nach int
s1 = s1 + s2; // Compiler-Fehler!
s1 = (short) (s1 + s2); // so funktioniert es
```

Die Typen `byte` und `short` sollten nur gebraucht werden, wenn Speicherplatz wichtig ist (z.B. in großen Feldern, später).

Long-Literale

Eine Zahl, z.B. 123456, wird in Java als `int`-Konstante gesehen. Für große Zahlen kann man ein `L` anhängen um auszudrücken, dass es ein `long` sein soll.

```
long l;
l = 12345678901234567890;  // Compilerfehler
l = 12345678901234567890L; // richtig
l = 1 << 40;  // falsch; keine Warnung!
l = 1L << 40; // richtig
```

Der Ausdruck `1L << 40` berechnet 2^{40} (eine Eins mit 40 Nullen in der Binärdarstellung).

Fließkomma-Literale

Eine Fließkommazahl wird als `double` (hohe Genauigkeit) interpretiert, `float` (einfache Genauigkeit) wird mit `F` gekennzeichnet.

```
System.out.println(3.1415926535897932);
// druckt 3.141592653589793
System.out.println(3.1415926535897932F);
// druckt 3.1415927
System.out.println((double) 0.1F);
// druckt 0.10000000149011612}
```

Zeichen (Character)

Der Typ `char` steht für ein Zeichen, ist aber gleichzeitig eine Zahl, die nach `int` konvertiert werden kann.

```
System.out.println('a');                    // druckt a
System.out.println((int)'a');               // druckt 97
System.out.println((char)97);               // druckt a
System.out.println((char)('A' + 1));// druckt B
System.out.println((char)('A' - 1));// druckt @
```

Die Zahlen/Zeichenzuordnung folgt Unicode (ISO 10646).
Weil die Zeichen im lateinischen Alphabet aufeinanderfolgen, kann man leicht den Buchstabenwert bestimmen.

```
char c; int i;
i = c - 'A' + 1; // Buchstabenwert (A=1..Z=26)
```

Vergleichsoperatoren

Die Operatoren < (kleiner),<= (kleiner oder gleich), > (größer), >= (größer oder gleich),== (gleich), != (ungleich) vergleichen zwei Werte und liefern einen Wahrheitswert.

```
boolean b;
b = 3 <= 5; // liefert true
b = 5 > 5;  // liefert false
b = 5 == 5; // liefert true
b = 5 != 5; // liefert false
b = 3.14159 <= Math.PI; // liefert true
b = 3.14159 == Math.PI; // liefert false
```

Vergleichsoperationen

UNI FREIBURG

Achtung

Auf Zeichenketten funktionieren ==, != nicht immer. Auf Fließkommazahlen können durch unterschiedliche Rundungsfehler zwei Zahlen verschieden werden.

```
boolean b; double d;
b = IOTools.readString() == "test"; // falsch
d = 1000000.1;
b = (d - 1000000 == 0.1); // liefert false
```

Operatoren auf Wahrheitswerten

Die Operatoren `&&`, `||`, `!`, `^` stehen für logisches UND, ODER, NICHT und XOR.

```
boolean b;
b = true && true; // liefert true
b = false && b;   // liefert false
b = b || true;    // liefert true
```

Die Operanden werden (wie immer) von links nach rechts ausgewertet. Steht bei `||` oder `&&` das Ergebnis bereits nach dem ersten Operanden fest, wird der zweite *nicht* ausgewertet (Kurzschlussauswertung).

```
b = false && IOTools.readInteger("x: ") == 5;
```

Hier fragt das Programm nicht nach `x`, da das Ergebnis sowieso `false` ist.

Der ?:-Operator

Es gibt einen ternären Operator

$$\textit{Ausdruck} ::= \textit{Ausdruck}\,?\,\textit{Ausdruck}\,:\,\textit{Ausdruck}$$

zum Beispiel

```
int x,y,a,m;
a = x >= 0 ? x : -x // liefert abs(x)
m = x >= y ? x : y  // liefert max(x,y)
```

- Der Operator entspricht dem if-then-else-Operator in funktionalen Programmiersprachen.
- Der erste Ausdruck muss den Typ `boolean` haben, die anderen beiden Ausdrücke müssen den gleichen Typ haben. Wenn der erste Ausdruck zu `true` auswertet, wird der zweite Ausdruck ausgewertet, ansonsten der dritte. Das Ergebnis der Auswertung wird zurückgegeben.
- Es gilt auch hier Kurzschlussauswertung: der Teil der nicht gebraucht wird, wird auch *nicht* ausgewertet.

Zuweisungen mit Operator

Man kann die meisten Operatoren mit einer Zuweisung kombinieren. Die Anweisung

```
i += j;
```

ist eine Abkürzung für

```
i = i + j;
```

Zuweisungen dürfen in Java auch als Ausdrücke (ge/miss)braucht werden.

```
i = (j = 4) + (k = 7);
```

weist `j` den Wert 4, `k` den Wert 7 und `i` den Wert 11 zu.
Das sollte aber nur in Ausnahmefällen verwendet werden.

Pre-/Postinkrement Operatoren

Man kann eine Variable gleichzeitig auslesen und um eins inkrementieren.
Der Ausdruck `i++` (Postinkrement) liest `i` aus, inkrementiert `i`, rechnet aber mit dem alten Wert. Zum Beispiel

```
int i, j;
i = 1;
j = (i++) + (i++) + (i++) + (i++) + (i++);
```

addiert die Zahlen von 1 bis 5.
Der Ausdruck `++i` (Preinkrement) inkrementiert `i`, liest `i` aus und rechnet mit dem neuen Wert. Zum Beispiel

```
i = 1;
j = (++i) + (++i) + (++i) + (++i) + (++i);
```

addiert die Zahlen von 2 bis 6.
Analog dekrementieren `i--` und `--i` den Wert von `i`.
Die Anweisung `i++;` kann als Abkürzung für `i = i + 1;` benutzt werden.

Operator-Präzedenzen

Name	Operator	Assoz.	
Methodenaufruf	`( )`	-	höchste
Komponentenzugriffe	`., [ ]`	links, -	Präzedenz
Postfix-Operatoren	`++, --` (Postin-/-dekrement)	-	
Unäre Operatoren	`+,-,~,!`	-	
	`++, --` (Prein-/-dekrement)	-	
Typkonvertierung	(*Datentyp*)	-	
Multiplikation	`*,/,%`	links	
Addition	`+,-`	links	
Schiebeoperation	`<<,>>,>>>`	links	
Relationen	`<=,<,>=,>, instanceof`	(links)	
Gleichheit	`==,!=`	links	
Bitweises UND	`&`	links	
Bitweises ODER	`|`	links	
Bitweises XOR	`^`	links	
Logisches UND	`&&`	links	
Logisches ODER	`||`	links	
Ternär	`?:`	rechts	
Zuweisung	`=, +=, -=, *=, /=, %=,`	rechts	niedrigste
	`&=, |=, ^=, <<=, <<=, >>=, >>>=`		Präzedenz

Deklaration mit Initialisierung

Eine Deklaration kann mit einer Zuweisung zu einer Anweisung kombiniert werden:

```
int i = 4;
```

reserviert den Speicherbereich für `i` und initialisiert `i` auf 4.
Es geht auch mit mehreren Variablen:

```
int i = 4, j = i + 1, k = j + 1;
```

reserviert den Speicherbereich für `i`,`j` und `k` und initialisiert die Variablen auf 4, 5 bzw. 6.

Methodenaufrufe

Methodenaufrufe

UNI FREIBURG

Syntax

Methodenaufruf ::= *Primary* . *Bezeichner* (*Ausdruck* (, *Ausdruck*)*)

Ein Methodenaufruf ist eine Anweisung, oder ein Ausdruck (wenn die Methode einen Wert zurückliefert).
Wir haben bereits Beispiele gesehen:

```
String name = IOTools.readString("Name: "); // 1
System.out.println("Hallo " + name);         // 2
```

In 1 besagt `IOTools`, wo die Methode zu finden ist, `readString` ist der Name der Methode und `"Ihr Name: "` das Argument oder Parameter.
Die Methode `readString` gibt ihren Parameter als Prompt aus, liest eine Zeichenkette ein und liefert diese zurück.
Was genau `IOTools` und `System.out` sind, werden wir erst klären können, wenn wir die objektorientierte Programmierung einführen.

IOTools

Die Klasse `IOTools` hält verschiedene Methoden bereit, um Werte einzulesen:

```
int i = IOTools.readInteger();
double j = IOTools.readDouble("j: ");
boolean b = IOTools.readBoolean("b: ");
System.out.println("" + i + j + b);
```

Der Parameter ist optional und ist ein Prompt der ausgegeben wird.
Die Funktionen warten bis eine Eingabe vom Benutzer kommt.

```
25
j: 1e2
b: ja
Eingabefehler java.lang.NumberFormatException: For input strin
Bitte Eingabe wiederholen...
b: true
25100.0true
```

Objekt-Methoden

Man kann auch Methoden auf Objekten, z. B. auf Zeichenketten, aufrufen.
Siehe `http://docs.oracle.com/javase/6/docs/api/java/lang/String.html`.

```
String s = "Hallo Welt";
int i = s.length();                  // i ist 10
s = s.substring(1,4);                // s ist "all"
s = s.replaceAll("l", "ba"); // s ist "ababa"
boolean b = s.equals("ababa"); // true
```

Blöcke

Block

Syntax

$$\textit{Anweisung} ::= \{\ \textit{Anweisung}^*\ \}$$

Mehrere Anweisungen können zu einem Block zusammengefasst werden, indem man sie in { und } einklammert.

```
int j;
{
    int i = 4;
    j = i + 1;
}
{
    int i = j;
    System.out.println(i); // gibt 5 aus.
}
```

Variablen „leben“ nur in dem Block, in dem sie deklariert wurden.

Verzweigungen

If-Anweisung

Bisher wurden unsere Programme immer von vorne nach hinten ausgeführt. Manchmal will man aber in Abhängigkeit von Werten andere Operationen ausführen.

```
int a = readInteger();
int b = readInteger();
if (b == 0) {
  System.err.println("oo");
} else {
  System.err.println(a / b);
}
```

Verschachtelte If-Anweisung

Man kann die if-Anweisung auch verschachteln.

```
int a = readInteger();
int b = readInteger();
if (b == 0) {
  if (a < 0) {
    System.err.println("-oo");
  } else {
    System.err.println("oo");
  }
} else {
  System.err.println(a / b);
}
```

Syntax einer If-Anweisung

UNI FREIBURG

Syntax

If-Anweisung ::= `if (`*Ausdruck*`)` *Anweisung* (`else` *Anweisung*)$^?$

- Der Ausdruck muss vom Typ `boolean` (Wahrheitswert) sein. Wenn der Ausdruck wahr ist, wird die erste Anweisung ausgeführt, ansonsten die zweite. Die zweite Anweisung kann auch weggelassen werden.
- Eine Anweisung kann auch ein Block sein (also geschweifte Klammern). Eine einzelne Anweisung kann auch ohne geschweifte Klammern stehen; dies ist aber gefährlich, da man schnell vergisst die Klammern hinzuzufügen, wenn man die Anweisung erweitert.
- Man sollte zur besseren Lesbarkeit die Unteranweisungen immer in neuen Zeilen schreiben und richtig einrücken.

Switch-Anweisung

Wenn man mehrere Verzweigungen braucht, bietet sich eine Switch-Anweisung an.

```
  int tage = readInteger("Wieviele Tage sind seit dem "
      + "1. Januar 2000 vergangen? ");
  System.out.print("Heute ist ");
  switch (tage % 7) {
  case 0:
    System.out.println("Samstag");
    break;
  case 1:
    System.out.println("Sonntag");
    break;
...
  case 6:
    System.out.println("Freitag");
    break;
  default:
    System.out.println("etwas schiefgelaufen.");
    break;
  }
```

Switch-Anweisung (2)

- Eine Switch-Anweisung hat die Form

```
  switch (Ausdruck) {
(
  case Konstante:
    Anweisung*
    break;
)*
  default:
    Anweisung*
    break;
  }
```

- Switch-Anweisungen funktionieren mit ganzen Zahlen und Aufzählungstypen. Seit Java 7 auch mit Zeichenketten.
- Wenn man das `break` vergisst führt Java den nächsten Fall gleich mit aus.

Schleifen

Schleifen

Manchmal möchte man eine Anweisung mehrmals wiederholen.
Java bietet drei verschiedene Arten von Schleifen an:

- Die While-Schleife.
  Die Laufbedingung wird vor jeder Ausführung des Schleifenrumpfs geprüft.
- Die For-Schleife.
  Variante der While-Schleife bei der normalerweise die Anzahl der Ausführungen im Voraus feststeht.
- Die Do-While-Schleife
  Die Laufbedingung wird nach jeder Ausführung des Schleifenrumpfs geprüft.

Syntax einer While-Schleife

UNI FREIBURG

Syntax

While-Schleife ::= `while` (*Ausdruck*) *Anweisung*

Der Ausdruck ist die Laufbedingung, die vor jedem Schleifendurchlauf geprüft wird. Liefert sie den Wahrheitswert `true`, so wird der Schleifenrumpf (die Anweisung) ausgeführt. Anschließend wird erneut die Laufbedingung geprüft.
Wenn die Laufbedingung `false` liefert beendet die Schleife und die nächste Anweisung nach der Schleife wird ausgeführt.

Die While-Schleife

Eine While-Schleife wird solange ausgeführt, wie die Laufbedingung der Schleife erfüllt ist. Zum Beispiel

```
int x = readInteger("x:"), root = 0;
while (root * root < x)
    root++;
}
System.out.println("Die Wurzel ist " + root);
```

Das Programm berechnet die Quadratwurzel (aufgerundet).

Dreieckszahlen

Wir wollen jetzt die Zahlen von 0 bis $(n-1)$ addieren. (Richtige Informatiker fangen beim Zählen mit 0 an!)
`http://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html`

```
int n = readInteger("n: ");
int sum = 0;
int i = 0;
while (i < n) {
  sum += i;
  i++;
}
System.out.println(sum);
```

Die Variable `i` ist eine Laufvariable, die von 0 bis `n - 1` hochläuft. Bei `i == n` wird die Schleife abgebrochen.

Die For-Schleife

Es kommt recht häufig vor, dass eine Laufvariable von einem zu einem anderen Wert läuft. Java, C und andere Sprachen bieten hierfür eine „schönere“ Syntax an.

```
int n = readInteger("n: ");
int sum = 0;
for (int i = 0; i < n; i++) {
  sum += i;
}
System.out.println(sum);
```

For-Schleife ist While-Schleife

Eine For-Schleife

```
for (A; B; C) {
  D
}
```

ist nur eine Abkürzung für

```
{
  A
  while (B) {
    D
    C
  }
}
```

Normalerweise deklariert und initialisiert `A` eine Variable (die nur lokal in der Schleife lebt), `B` prüft ob diese Variable noch in den Grenzen ist und `C` inkrementiert (oder dekrementiert) die Variable.

Die Do-While-Schleife

Die Do-While-Schleife prüft die Laufbedingung erst nach dem Schleifenrumpf. Der Unterschied ist also, dass die Laufbedingung vor der ersten Ausführung des Schleifenrumpfs noch nicht gelten muss.

```
int n = IOTools.readInteger("n:");
int i = 0;
do {
    System.out.println(i);
    i++;
} while (i < n);
```

Das Programm gibt die Zahlen von 0 bis `n-1` aus. Wenn aber $n = 0$ ist, gibt das Programm trotzdem die Zahl 0 aus.
Oftmals ist es besser und richtiger eine While-Schleife zu verwenden.

Die Break-Anweisung

Manchmal möchte man erst in der Mitte der Schleife die Schleife beenden. Dafür gibt es das Kommando `break`.

```
int sum = 0;
while (true) {
    int n = IOTools.readInteger
        ("Zahl (0 bricht ab): ");
    if (n == 0) {
        break;
    }
    sum += n;
}
```

Das Kommando break verlässt sofort die umschließende Schleife oder Switch-Anweisung ohne die restlichen Kommandos auszuführen.

Die Break-Anweisung mit Label

Wenn man nicht die innerste Schleife verlassen will, kann man das mit einem Label markieren.

```
aussen:
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        if (f(i) == f(j)) {
            System.out.println("f ist nicht
                injektiv");
            break aussen;
        }
    }
}
```

Der Block der mit dem Label markiert wurde, muss keine Schleife sein.

Die Continue-Anweisung

Mit der Anweisung `continue` kann man einen neuen Schleifendurchlauf starten. Es wird dann wieder die Laufbedingung geprüft.

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (j == i)
            continue;
        System.out.println(i
            + " ist ungleich " + j);
    }
}
```

- Bei For-Schleifen wird das Inkrement noch ausgeführt.
- Auch ein `continue` kann mit einem Label versehen werden. Das Label muss aber an einer Schleife stehen.