3.4 Anweisungen

3.4.1 Unterschied zwischen Ausdrücken und Anweisungen

- *Ausdrücke* (expressions) sind entweder atomar (z. B. `1` oder `x`) oder Anwendungen von Operatoren auf geeignete Teilausdrücke (z. B. `x + 1` oder `0 < x && x < n`).

- Sie werden zur Laufzeit eines Programms *ausgewertet* und liefern damit einen Wert. (Ausnahme: Aufrufe von Methoden mit Resultattyp `void` sind Ausdrücke, liefern aber keinen Wert.)

- *Anweisungen* (statements, z. B. `while (...) ...`) werden zur Laufzeit *ausgeführt* und liefern keinen Wert.

- Bestimmte Ausdrücke (sog. *Anweisungs-Ausdrücke*, siehe § 3.4.3) können als Anweisungen verwendet werden, indem sie mit einem Semikolon abgeschlossen werden. Ihr Wert wird dann ignoriert.

3.4.2 Erläuterungen zur Darstellung

- Schwarz geschriebene Zeichen oder Zeichenfolgen wie z. B. `for`, `{` und `}` müssen „wörtlich“ verwendet werden.
- Blau geschriebene Namen wie z. B. `label` und `statement` sind Platzhalter.
- Rot geschriebene Zeichen wie z. B. **`{`** und **`}`** sind EBNF-Metazeichen mit folgender Bedeutung (EBNF = Extended Backus-Naur Form):
  - `x` **`|`** `y` bedeutet, dass an dieser Stelle entweder `x` oder `y` stehen kann.
  - **`[`** `x` **`]`** bedeutet, dass an dieser Stelle null- oder einmal `x` stehen kann.
  - **`{`** `x` **`}`** bedeutet, dass an dieser Stelle beliebig viele `x` nacheinander stehen können (d. h. auch nullmal `x` ist zulässig).
- Zum Beispiel:
  - `break` **`[`**`label`**`]`** bedeutet, dass nach dem Schlüsselwort `break` optional irgendeine Marke `label` stehen kann.
  - `{` **`{`** `declaration` **`|`** `statement` **`}`** `}` bedeutet, dass zwischen geschweiften Klammern (schwarze geschweifte Klammern) beliebig viele (rote geschweifte Klammern) Deklarationen (`declaration`) oder (roter Strich) Anweisungen (`statement`) stehen können.

3.4.3 Atomare Anweisungen

- Alle im folgenden genannten Anweisungen müssen mit einem Semikolon abgeschlossen werden, das im folgenden jeweils weggelassen wird.
- leere Anweisung (z. B. als Schleifenrumpf: `while ((x /= 2) > 1) ;`)
- Ausdrucks-Anweisung, bestehend aus einem Anweisungs-Ausdruck:
  - Zuweisung (z. B. `x = 1`)
  - Präfix- oder Postfix-Inkrement- oder -Dekrement-Ausdruck (z. B. `++x` oder `x--`)
  - Methodenaufruf (z. B. `System.out.println(5)`)
  - Objekterzeugung (z. B. `new int [10]` oder `new Account("Heinlein")`)

  In C kann eine Ausdrucks-Anweisung aus einem beliebigen Ausdruck bestehen.
- `break [label]`
  - Zum vorzeitigen Beenden einer Anweisung
  - Ohne Marke `label` nur direkt oder indirekt in einer Schleife oder `switch`-Anweisung
  - Mit Marke direkt oder indirekt in einer passend markierten beliebigen Anweisung (vgl. § 3.4.4)

- `continue [label]`
  - Zum vorzeitigen Fortsetzen einer Schleife mit dem nächsten Durchlauf
  - Grundsätzlich nur direkt oder indirekt in einer Schleife
  - Mit Marke `label` nur direkt oder indirekt in einer passend markierten Schleife (vgl. § 3.4.4)
- `yield expression`
  - Zum Zurückgeben eines Werts `expression` in einem `switch`-Ausdruck (vgl. § 3.4.7)
- `return [expression]`
  - Zum vorzeitigen Beenden einer Methode oder eines Konstruktors
  - und/oder zum Zurückgeben eines Resultatwerts `expression`
- `throw expression`
  - Zum Werfen einer Ausnahme `expression` (vgl. Kapitel 8)
- Im Gegensatz zu C, gibt es keine `goto`-Anweisung!
  `break`- und `continue`-Anweisungen mit Marke (die es in C nicht gibt) erlauben nur Sprünge „von innen nach außen“, d. h. keinen beliebigen „Spaghetti-Code“.

3.4.4 Zusammengesetzte Anweisungen mit beliebigen Teilanweisungen

- Markierte Anweisung (für `break` und `continue` mit Marke):
  - `label: statement`
  - Wenn eine Anweisung `statement` mit einer Marke `label` markiert ist, kann sie mit der Anweisung `break label` vorzeitig beendet werden.
  - Wenn die Anweisung eine Schleife ist, kann sie mit der Anweisung `continue label` auch vorzeitig fortgesetzt werden.
- Anweisungsblock:
  - `{ { statement | declaration } }`
  - Ein Anweisungsblock kann neben Anweisungen `statement` auch lokale Variablen- und Klassendeklarationen `declaration` in beliebiger Reihenfolge enthalten.
- Verzweigung:
  - `if (condition) statement`
  - `if (condition) statement else statement`
  - Die Anweisungen `statement` sind häufig Anweisungsblöcke.
  - Anders als in C, muss die Bedingung `condition` Typ `boolean` besitzen.

- Schleifen:
  - `while (condition) statement`
  - `do statement while (condition);`
  - `for (init; condition; update) statement`
    - Initialisierungsteil `init` und Aktualisierungsteil `update` können aus mehreren Anweisungs-Ausdrücken (vgl. § 3.4.3) bestehen, die durch Kommas getrennt sind.
    - Der Initialisierungsteil kann auch eine Deklaration einer oder mehrerer lokaler Variablen sein (die dann nur innerhalb der Schleife gültig sind).
    - Jeder Teil innerhalb der Klammern kann leer sein, wobei eine leere Bedingung immer erfüllt ist.
  - `for (type var : expression) statement`
    - Vor dem Doppelpunkt muss eine Deklaration einer lokalen Variablen `var` mit Typ `type` stehen (die dann nur innerhalb der Schleife gültig ist).
    - Der Ausdruck `expression` muss entweder eine Reihe oder ein Objekt liefern, das die Schnittstelle `Iterable` implementiert (vgl. § 6.2.2).
    - Der Schleifenrumpf wird nacheinander für jedes Element dieser Reihe bzw. dieses Objekts ausgeführt, wobei die Variable in jedem Durchlauf das entsprechende Element enthält, zum Beispiel:

```
double [] a = ......;
for (double x : a) System.out.println(x);
```

3.4.5 Zusammengesetzte Anweisungen mit Blöcken als Teilanweisungen

- Fallunterscheidung:
  - `switch (expression) block`
  - Anders als in C, müssen `case`- und `default`-Marken direkt im abhängigen Anweisungsblock stehen.
  - Ohne `break`, `continue`, `return` oder `throw` am Ende eines Falls „fällt“ man direkt in den nächsten Fall. Das kann durch Verwendung von Pfeilen statt Doppelpunkten nach den `case`-Marken vermieden werden (siehe Beispiel in § 3.4.7).
  - Der Typ des Ausdrucks `expression` muss `char`, `byte`, `short`, `int`, `String` oder ein Aufzählungstyp sein, die `case`-Marken müssen dazu passende konstante Werte (oder konstante Ausdrücke) sein.
- Synchronisation paralleler Threads:
  - `synchronized (expression) block`
- Auffangen von Ausnahmen und/oder Ausführen von Terminierungsanweisungen:
  - `try [ ( resources) ] block { catch (type var) block } [ finally block ]`
  - Es muss mindestens ein `catch`-Block oder der `finally`-Block oder `resources` vorhanden sein (vgl. § 8.6).

3.4.6 Beispiel: Markierte Anweisungen

```
// Überprüfe möglichst effizient, ob jede Zeile der Matrix a
// mindestens eine positive Zahl enthält.
boolean check (double [] [] a) {
  outer:                    // Äußere for-Schleife mit Marke »outer«.
  for (int i = 0; i < a.length; i++) { // Lokale Deklaration von i.
    for (int j = 0; j < a[i].length; j++) {  // Lokale Dekl. von j.
      if (a[i][j] > 0) { // Positive Zahl in Zeile i gefunden.
        continue outer;  // Abbruch der inneren und
                         // Fortsetzung der äußeren for-Schleife.
      }
    }
    return false;        // Keine positive Zahl in Zeile i gefunden.
  }
  return true;           // Positive Zahl in jeder Zeile gefunden.
}
```

3.4.7 Beispiel: `switch`-Anweisungen und `switch`-Ausdrücke

- Die Methode `print` soll abhängig vom Wert des Parameters `which` entweder das erste oder das zweite Element der Reihe `msgs` ausgeben.

- „Klassische“ Formulierung mit einer `switch`-Anweisung wie in C (abgesehen davon, dass in C nicht anhand von Zeichenketten verzweigt werden kann):

```
void print (String [] msgs, String which) {
  int i;
  switch (which) {
  case "1st":
  case "first": // Vor jeder Marke muss case stehen.
    i = 0;
    break;      // Am Ende jedes Falls muss break o. ä. stehen.
  case "2nd":
  case "second":
    i = 1;
    break;
  default:
    throw new IndexOutOfBoundsException(which);
  }
  System.out.println(msgs[i]);
}
```

- Kürzere Formulierungsmöglichkeit seit Java 12:

```
void print (String [] msgs, String which) {
  int i;
  switch (which) {
  // Nach einem case können mehrere Marken stehen.
  // Bei Verwendung des Pfeils anstelle des Doppelpunkts
  // muss am Ende der Fälle kein break stehen.
  case "1st", "first" -> i = 0;
  case "2nd", "second" -> i = 1;
  default -> throw new IndexOutOfBoundsException(which);
  }
  System.out.println(msgs[i]);
}
```

Allgemein kann nach einem Pfeil entweder eine Ausdrucks-Anweisung (vgl. § 3.4.3) oder ein Anweisungsblock (vgl. § 3.4.4) oder eine `throw`-Anweisung (vgl. § 3.4.3) stehen.

- Noch kürzer mit einem `switch`-Ausdruck, der einen Wert zurückliefert, anstelle einer `switch`-Anweisung:

```
void print (String [] msgs, String which) {
  int i =
    switch (which) {
    case "1st", "first" -> 0;
    case "2nd", "second" -> { yield 1; }
    default -> throw new IndexOutOfBoundsException(which);
    };
  System.out.println(msgs[i]);
}
```

Wenn `switch` wie hier als Ausdruck statt als Anweisung verwendet wird, können nach einem Pfeil nicht nur Anweisungs-Ausdrücke, sondern beliebige Ausdrücke mit abschließendem Semikolon stehen. Der Wert des gesamten `switch`-Ausdrucks ist dann der Wert des jeweils ausgewerteten Ausdrucks.

Wenn nach einem Pfeil ein Anweisungsblock steht, muss er seinen Wert mit einer `yield`-Anweisung (vgl. § 3.4.3) zurückliefern, ähnlich wie eine Methode ihren Resultatwert mit `return` zurückliefert.

Achtung: Eine `break`-Anweisung bezieht sich niemals auf einen `switch`-Ausdruck, sondern immer auf eine Anweisung (vgl. § 3.4.3).

3.4.8 Beispiel: try/catch/finally

```
// Auffangen unterschiedlicher Ausnahmen.
int p = -1, q = 0, r = 1000*1000*1000;
double [] a = null;
while (true) {
  try {
    a = new double [p/q + r];
    a[0] = 1;
    break;
  }
  catch (ArithmeticException e) {
    System.out.println("Division durch 0");
    q = 1;
  }
  catch (NegativeArraySizeException e) {
    System.out.println("Negative Reihenlänge");
    p = -p;
  }
  catch (OutOfMemoryError e) {
    System.out.println("Speichermangel");
    r = -1;
  }
```

```
    catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Indexfehler");
      break;
    }
    finally {
      System.out.println("Ende der try-Anweisung");
    }
  }
```

- Der obige Code produziert folgende Ausgabe:

```
Division durch 0
Ende der try-Anweisung
Speichermangel
Ende der try-Anweisung
Negative Reihenlänge
Ende der try-Anweisung
Indexfehler
Ende der try-Anweisung
```

3.5 Überladen von Namen

- In C müssen alle Funktionen eines Programms (oder zumindest einer Quelldatei) eindeutige Namen besitzen.
- In Java können Methodennamen *überladen* werden, d. h. innerhalb einer Klasse kann es mehrere Methoden mit dem gleichen Namen geben, sofern sie unterschiedliche Parameterlisten besitzen. (Unterschiedliche Resultattypen genügen nicht.)
- Beim Aufruf einer überladenen Methode vergleicht der Übersetzer die Typen der aktuellen Aufrufparameter jeweils mit den Typen der formalen Methodenparameter und wählt die *am besten passende* Methode aus (d. h. die Reihenfolge der Methodendeklarationen spielt keine Rolle).
- Eine Methode passt exakt, wenn die Typen aller Aufrufparameter mit den Typen der entsprechenden Methodenparameter übereinstimmen.
- Eine Methode passt (mit Umwandlungen), wenn sich die Typen aller Aufrufparameter implizit in die Typen der entsprechenden Methodenparameter umwandeln lassen.
- Eine Methode passt besser als eine andere, wenn sich alle Parametertypen der ersten implizit in die entsprechenden Parametertypen der zweiten umwandeln lassen.
- Wenn es keine passende Methode gibt oder mehrere Methoden „gleich gut“ passen, erhält man eine entsprechende Fehlermeldung.

Beispiel 1

```
// Methodendefinitionen.
void print (boolean x) { ..... }        // 1
void print (int x) { ..... }            // 2
void print (double x) { ..... }         // 3

// b/i/d/s/l/c/S seien Ausdrücke mit Typ
// boolean/int/double/short/long/char/String.

// Welche der Methoden 1/2/3 passt exakt (++),
// passt mit Umwandlung (+), passt nicht (-)?
// Welche Methode wird ausgewählt?
                    //  Parametertyp     Methode          Auswahl
                    //                  1   2   3
print(b);           //  boolean         ++  -   -        1
print(i);           //  int             -   ++  +        2
print(d);           //  double          -   -   ++       3
print(s);           //  short           -   +   +        2
print(l);           //  long            -   -   +        3
print(c);           //  char            -   +   +        2
print(S);           //  String          -   -   -        keine
```

Beispiel 2

```
// Methodendefinitionen.
void test (long x, double y) { ..... }          // 1
void test (double x, long y) { ..... }          // 2
void test (int x, int y) { ..... }              // 3


// Welche der Methoden 1/2/3 passt exakt (++),
// passt mit Umwandlung (+), passt nicht (-)?
// Welche Methode wird ausgewählt?
                      //  Parametertypen   Methode       Auswahl
                      //                   1   2   3
test(i, i);           //  int, int         +   +   ++    3
test(i, d);           //  int, double      +   -   -     1
test(d, i);           //  double, int      -   +   -     2
test(d, d);           //  double, double   -   -   -     keine
test(c, c);           //  char, char       +   +   +     3
test(l, l);           //  long, long       +   +   -     mehrdeutig

// Auflösung der Mehrdeutigkeit durch explizite Typumwandlungen.
test(l, (double)l);   // long, double     ++  -   -     1
test((double)l, l);   // double, long     -   ++  -     2
```