3 Gemeinsamkeiten und Unterschiede zwischen Java und C

3.1 Äußerlichkeiten

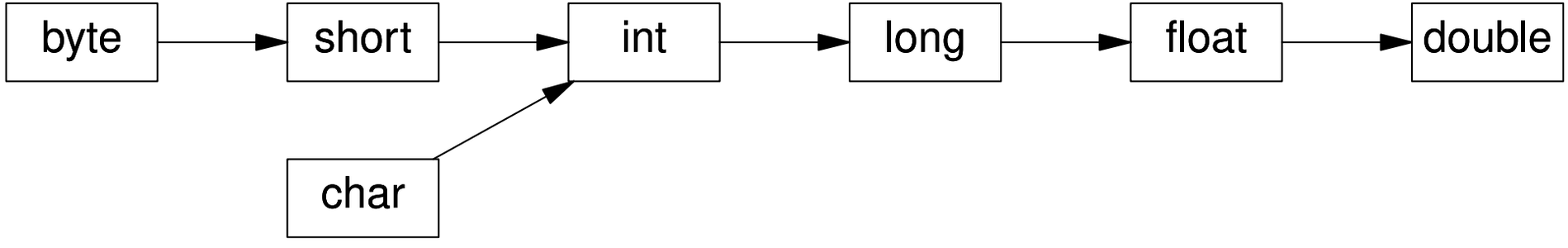
- Kommentare in Java erstrecken sich entweder (wie in C) von `/*` bis `*/` („Blockkommentare“) oder von `//` bis zum Zeilenende („Zeilenkommentare“).
- Blockkommentare können (wie in C) nicht verschachtelt werden, d. h. ein Blockkommentar endet immer beim ersten Auftreten von `*/`.
- Wenn man „echte“ Kommentare als Zeilenkommentare formuliert, kann man Blockkommentare zum „Auskommentieren“ von Programmabschnitten verwenden.
- In Java gibt es keinen Präprozessor, d. h. insbesondere kein `#include` und kein `#define`.
- In C besitzt die Hauptfunktion `main` Resultattyp `int` (der Resultatwert stellt den Exitstatus des Programms dar, der an das Betriebssystem zurückgegeben wird), in Java ist der Resultattyp `void` (der Exitstatus ist standardmäßig 0).
  Auch die Parameterliste von `main` ist unterschiedlich (vgl. § 4.6.3).

3.2 Datentypen

3.2.1 Elementare Typen

Typ	*Java*	*C*
boolean	vorhanden	nicht vorhanden
char	16 Bit	8 Bit
byte	8 Bit	entspricht `signed char`
short	16 Bit	meist 16 Bit
int	32 Bit	meist 32 oder 16 Bit
long	64 Bit	meist 64 oder 32 Bit
unsigned short	vgl. `char`	Größe wie `short`
unsigned int	nicht vorhanden	Größe wie `int`
unsigned long	nicht vorhanden	Größe wie `long`
float	32 Bit	meist 32 Bit
double	64 Bit	meist 64 Bit

- Ganzzahlige Arithmetik wird grundsätzlich mit `int`- oder `long`-Werten ausgeführt; „kürzere“ Werte werden ggf. automatisch expandiert.
- Umwandlungen von „kürzeren“ in „längere“ Typen erfolgen bei Bedarf implizit, während andere Umwandlungen explizit erfolgen müssen.



- Beispiel: Umwandlung von Klein- in Großbuchstaben

```
char c = 'x'; // ... oder ein anderer Kleinbuchstabe.

// Fehler: Rechte Seite hat Typ int => Zuweisung an char verboten.
c = c + 'A' - 'a';

// Korrekt: Rechte Seite explizit in char umwandeln.
c = (char)(c + 'A' - 'a');

// Auch korrekt: Bei += etc. wird automatisch umgewandelt!
c += 'A' - 'a';
```

3.2.2 Zeiger und Referenzen

- In Java gibt es keine expliziten Zeigertypen (wie z. B. `int*` oder `int (*) ()`) und somit auch keinen Adress- (`&`) und Inhaltsoperator (`*`).
- Stattdessen sind alle nicht-elementaren Typen (d. h. Klassen, Schnittstellen und Reihen) *Referenztypen*, d. h. implizit Zeigertypen.
- Die *Nullreferenz* `null`, die mit allen Referenztypen kompatibel ist, repräsentiert eine Referenz auf „nichts“.
- Anstelle von Funktionszeigern kann man Referenzen auf Objekte verwenden, die die gewünschte „Funktion“ (d. h. Methode) besitzen (vgl. § 7.1).

3.2.3 Reihen (arrays)

Eindimensionale Reihen

- Reihentypen in Java legen nur den Typ der Elemente fest, nicht jedoch ihre Anzahl, z. B.:

```
double [] a;                // Reihe von double-Werten.
```

- Reihenobjekte müssen zur Laufzeit explizit mit `new` erzeugt werden, z. B.:

```
a = new double [10];    // Reihe mit 10 double-Werten erzeugen.
```

- Anders als in C, muss die Länge bzw. Elementzahl kein konstanter Ausdruck sein, z. B.:

```
// Kommandozeilenargument args[0] in eine ganze Zahl umwandeln
// und eine Reihe mit entsprechend vielen double-Werten erzeugen.
a = new double [Integer.parseInt(args[0])];
```

- Bei Angabe einer negativen Länge erhält man eine Ausnahme des Typs `NegativeArraySizeException`. (Die Länge 0 ist zulässig.)

- Die Länge einer Reihe `a`, d. h. die Anzahl ihrer Elemente, kann mittels `a.length` abgefragt werden.

- Bei der Erzeugung einer Reihe werden ihre Elemente mit einem typabhängigen Standardwert initialisiert:
  - `false` für Typ `boolean`
  - `'\0'` für Typ `char`
  - `0` bzw. `0.0` für alle numerischen Typen
  - `null` für alle Referenztypen (Klassen, Schnittstellen, Reihen)

- Nach Erzeugung einer Reihe kann ihre Länge nicht mehr verändert werden. Um eine Reihe quasi zu „vergrößern“ (oder zu „verkleinern“), muss man eine neue Reihe mit der gewünschten Länge erzeugen und die Elemente umkopieren.

- Für einen ganzzahligen Wert `i` zwischen 0 einschließlich und `a.length` ausschließlich liefert `a[i]` das `i`-te Element der Reihe `a`.

- Für andere Werte von `i` erhält man eine Ausnahme des Typs `ArrayIndexOutOfBoundsException`, d. h. es findet Bereichsprüfung statt.

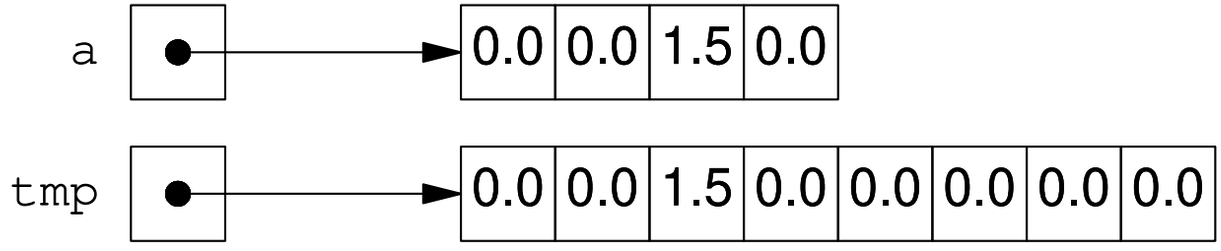
- Wenn `a` die Nullreferenz `null` ist, produzieren die Ausdrücke `a.length` und `a[i]` eine Ausnahme des Typs `NullPointerException`.

- Wenn `a` und `b` Reihenvariablen sind, wird bei einer Zuweisung `a = b` lediglich die *Referenz* `b` kopiert, aber nicht die Elemente der von `b` referenzierten Reihe.

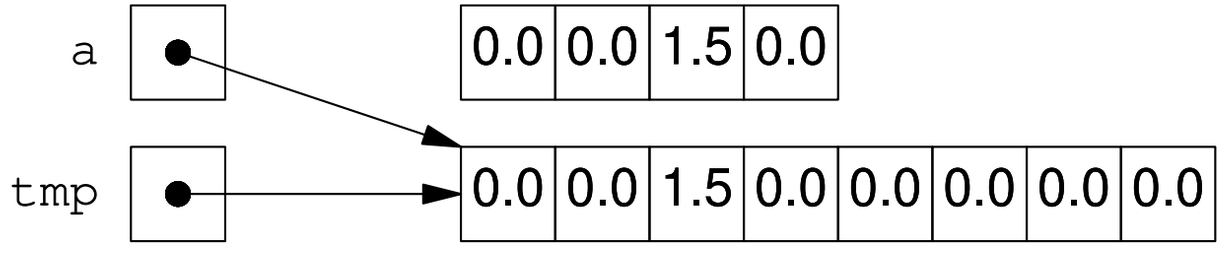
Beispiel

```
double [] a = new double [4];
a[2] = 1.5;

double [] tmp = new double [8];
for (int i = 0; i < a.length; i++) tmp[i] = a[i];
```




```
a = tmp;
```



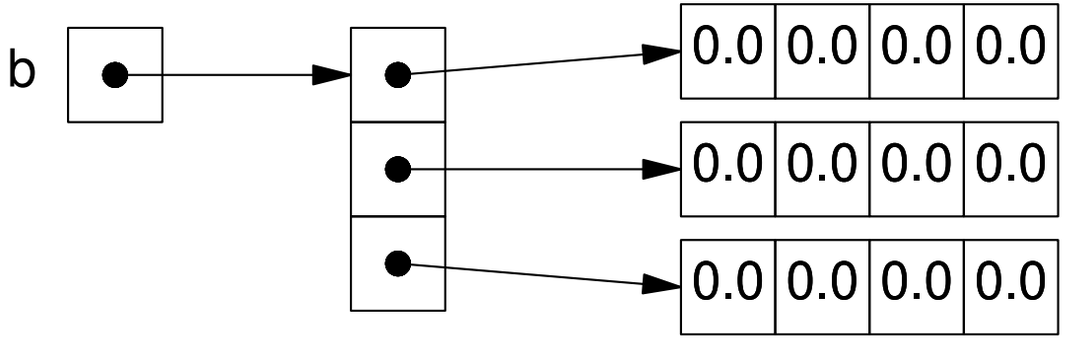

```
// Die nicht mehr referenzierte Reihe wird (zu einem nicht näher
// spezifizierten Zeitpunkt) vom System automatisch freigegeben.
```

Mehrdimensionale Reihen

- Mehrdimensionale Reihen erhält man indirekt als Reihen von Reihen, z. B.:

```
int m = ..., n = ...;
double [] [] b = new double [m] [n];
for (int i = 0; i < m; i++) {
  for (int j = 0; j < n; j++) {
    System.out.println(b[i][j]);
  }
}
```

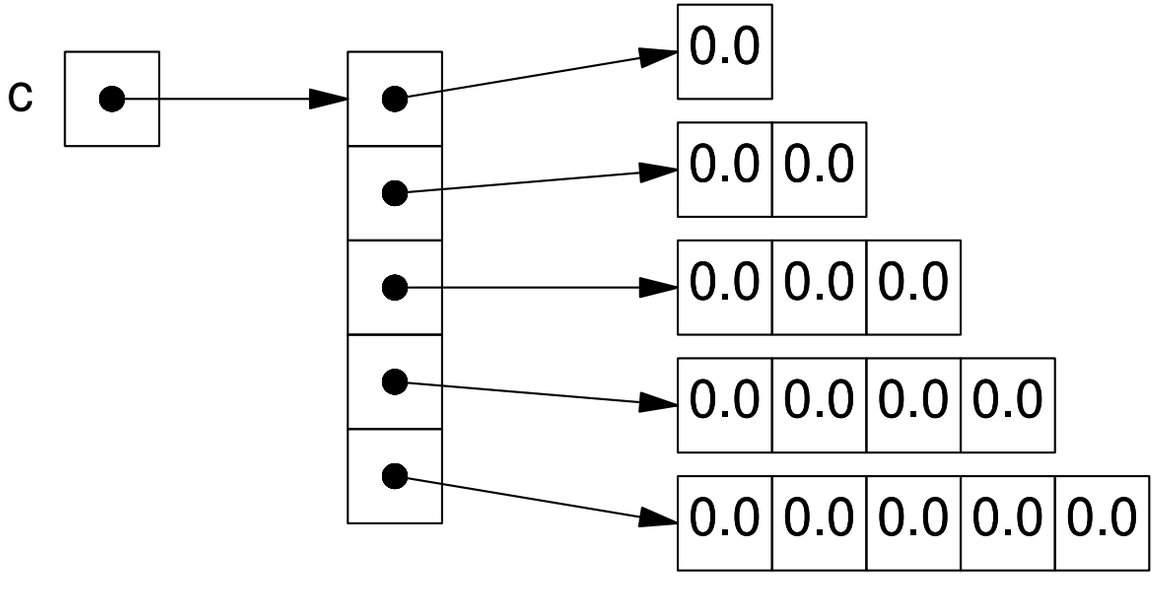
Hier ist `b` (eine Referenz auf) eine Reihe mit `m` Elementen (d. h. `b.length` liefert `m`); jedes dieser Elemente ist selbst wieder (eine Referenz auf) eine Reihe mit `n` Elementen des Typs `double` (d. h. `b[i].length` liefert jeweils `n`).



- Reihen von unterschiedlich großen Reihen kann man z. B. wie folgt erzeugen:

```
int n = ...;
double [] [] c = new double [n] [];
for (int i = 0; i < n; i++) {
  c[i] = new double [i+1];
}
```

Hier ist `c` (eine Referenz auf) eine Reihe mit `n` Elementen, die anfangs alle den Wert `null` besitzen; anschließend wird an jedes Element `c[i]` (eine Referenz auf) eine Reihe mit `i+1` Elementen des Typs `double` zugewiesen.



Reiheninitialisierer

- Reihen können auch durch Reiheninitialisierer erzeugt werden, z. B.:

```
double [] a = { 1.0, 2.0, 3.0 }; // double-Reihe mit 3 Elementen.

a = new double [] { 4.0, 5.0 };  // double-Reihe mit 2 Elementen.

double [] [] b = { { 1.0 }, { 2.0, 3.0 } };    // Zweidim. Reihe.
```

- Bei der Deklaration einer Reihenvariablen (erste und dritte Zeile im vorigen Beispiel) kann der Reiheninitialisierer `{ ..... }` direkt verwendet werden, in allgemeinen Ausdrücken (zweite Zeile) muss er mit einem `new`-Ausdruck kombiniert werden.

- Um mehrdimensionale Reihen zu erzeugen und zu initialisieren, können Reiheninitialisierer verschachtelt werden.

- Die Elemente eines Reiheninitialisierers können beliebige Ausdrücke sein.
  Bei mehrdimensionalen Reihen können sie selbst wieder Reihen sein.

3.2.4 Zeichenketten (strings)

- Zeichenkettenkonstanten wie z. B. `"abc"` besitzen in Java den vordefinierten Typ `String`.
- Die Länge eines `String`-Objekts `s` kann mit `s.length()` abgefragt werden. (Achtung: Die Länge einer Reihe `a` erhält man mittels `a.length`, ohne Klammern.)
- Für `i` zwischen 0 einschließlich und `s.length()` ausschließlich liefert `s.charAt(i)` das `i`-te Zeichen der Zeichenkette `s`. Für andere Werte von `i` erhält man eine Ausnahme des Typs `StringIndexOutOfBoundsException`.
- Wenn `s` die Nullreferenz `null` ist, produzieren die Ausdrücke `s.length()` und `s.charAt(i)` (sowie alle anderen Methodenaufrufe auf `s`) eine Ausnahme des Typs `NullPointerException`.
- Zeichenketten können mit dem Verkettungsoperator `+` verkettet werden, d. h. für `String`-Objekte `s` und `t` liefert `s + t` ein neues `String`-Objekt, das die Zeichen von `s` und `t` nacheinander enthält.
  Dementsprechend weist `s += t` die Verkettung von `s` und `t` wieder an die `String`-Variable `s` zu.
- Wenn nur ein Operand einer Verkettungsoperation eine Zeichenkette ist, wird der andere Operand automatisch in eine Zeichenkette umgewandelt.

- Beispiel:

```
int x = 2, y = 3;
System.out.println(
 "Die Summe von " + x + " und " + y + " ist " + (x + y) + ".");
```

- `s.equals(t)` überprüft, ob die Zeichenketten `s` und `t` (d. h. eigentlich die von `s` und `t` referenzierten `String`-Objekte) „gleich“ sind, d. h. die gleichen Zeichen in der gleichen Reihenfolge enthalten.

- Der Vergleich `s == t` überprüft lediglich, ob die Referenzen `s` und `t` gleich sind, d. h. ob `s` und `t` *dasselbe* `String`-Objekt referenzieren.

- Die Klasse `String` bietet zahlreiche weitere Operationen an, z. B. Suchen einzelner Zeichen oder Zeichenfolgen in einer Zeichenkette, Bilden von Teilzeichenketten usw.

- Anders als in C, sind Zeichenketten keine `char`-Reihen, und sie besitzen kein abschließendes Nullzeichen.

- Außerdem sind Zeichenketten *unveränderbar*, d. h. der Inhalt eines `String`-Objekts kann nach seiner Erzeugung nicht mehr geändert werden.
  Um eine Zeichenkette quasi zu „ändern“, muss man eine neue Zeichenkette mit geändertem Inhalt erzeugen.

3.3 Operatoren

3.3.1 Übersicht

Operator	*Anwendung*	*Bedeutung*
`new`	präfix	Objekterzeugung (vgl. § 3.2.3 und § 4.9)
`.name`	postfix	Feldzugriff (vgl. § 4.5)
`.name(args)`	postfix	Methodenaufruf (vgl. § 4.6)
`[index]`	postfix	Reihenelementzugriff (vgl. § 3.2.3)
`++`	postfix	Inkrement
`--`	postfix	Dekrement
`++`	präfix	Inkrement
`--`	präfix	Dekrement
`+`	präfix	Identische Funktion (unäres Plus)
`-`	präfix	Arithmetische Negation (unäres Minus)
`~`	präfix	Bitweises Komplement
`!`	präfix	Logische Negation
`(type)`	präfix	Typumwandlung (cast)

Operator	*Anwendung*	*Bedeutung*
`*`	infix	Multiplikation
`/`	infix	Division
`%`	infix	Rest bei Division
`+`	infix	Addition oder Verkettung von Zeichenketten
`-`	infix	Subtraktion
`<<`	infix	Bitverschiebung nach links
`>>`	infix	Arithmetische Bitverschiebung nach rechts
`>>>`	infix	Logische Bitverschiebung nach rechts
`< > <= >=`	infix	Numerische Vergleiche
`instanceof`	infix	Dynamischer Typtest (vgl. § 5.7)
`==`	infix	Test auf Gleichheit
`!=`	infix	Test auf Ungleichheit
`&`	infix	Bitweise oder Boolesche Und-Verknüpfung
`^`	infix	Bitweise oder Boolesche Exklusiv-Oder-Verknüpfung
`|`	infix	Bitweise oder Boolesche (Inklusiv-)Oder-Verknüpfung
`&&`	infix	Logische Und-Verknüpfung
`||`	infix	Logische Oder-Verknüpfung
`? :`	infix	Verzweigung
`=`	infix	Zuweisung
`+= -= ...`	infix	Kombinierte Zuweisung
`->`	infix	Lambda-Ausdruck (vgl. § 7.2)

3.3.2 Erläuterungen

- Blau geschriebene Wörter sind Platzhalter.
- In den Tabellen besitzen alle Operatoren einer Gruppe denselben Vorrang, der höher ist als der Vorrang der nächsten Gruppe.
- Daher ist der Ausdruck `-a++ * b < c + d` beispielsweise äquivalent zu `((-(a++)) * b) < (c + d)`.
- Die Zuweisungsoperatoren und der Verzweigungsoperator sind rechtsassoziativ, alle anderen binären Operatoren linksassoziativ.
- Daher ist der Ausdruck `a - b - c` beispielsweise äquivalent zu `(a - b) - c` (und nicht zu `a - (b - c)`), während `a = b = 1` äquivalent zu `a = (b = 1)` ist.

3.3.3 Anmerkungen

- Anders als in C, wird der linke Operand eines Operators garantiert vor dem rechten ausgewertet.
  (Zum Beispiel liefert der Ausdruck `a[i++] - a[i++]` für `i = 0` garantiert den Wert von `a[0] - a[1]`; in C wäre z. B. auch `a[1] - a[0]` zulässig.)
- Ebenso werden die Argumente eines Methodenaufrufs garantiert von links nach rechts ausgewertet.
  (Zum Beispiel wird beim Ausdruck `f(i++, i++)` für `i = 0` garantiert der Methodenaufruf `f(0, 1)` ausgeführt; in C wäre z. B. auch `f(1, 0)` zulässig.)
- Die logischen Operatoren `!`, `&&` und `||` können nur auf `boolean`-Werte angewandt werden und liefern als Resultat wieder einen `boolean`-Wert (`true` oder `false`).
  (In C können sie auf Werte aller elementaren Typen angewandt werden und liefern als Resultat einen `int`-Wert, 1 oder 0.)
- Bei den Operatoren `&&` und `||` wird der rechte Operand (wie in C) nur ausgewertet, wenn dies zur Ermittlung des Ergebnisses notwendig ist.
  (Zum Beispiel produziert der Ausdruck `a != null && a.length > 0` für `a` gleich `null` keine `NullPointerException`, weil `a.length` in diesem Fall gar nicht ausgewertet wird.)

- Bei einer Verzweigung `x ? y : z` wird (wie in C), abhängig vom Wert des ersten Operanden (`x`), entweder nur der zweite (`y`) oder nur der dritte Operand (`z`) ausgewertet.
- Anders als in C, wird bei ganzzahliger Division garantiert „in Richtung 0“ gerundet. (In C ist das Rundungsverhalten bei negativen Operanden nicht festgelegt.)
- Für den Rest bei ganzzahliger Division gilt für alle möglichen Werte von `x` und `y`: `x % y == x - x/y * y`.
- Zum Beispiel:

```
 14 /  3  →  4        14 %  3  →  2
-14 /  3  → -4       -14 %  3  → -2
 14 / -3  → -4        14 % -3  →  2
-14 / -3  →  4       -14 % -3  → -2
```

- Für positives `y` erhält man mittels `(x % y + y) % y` für beliebige `x` den mathematisch korrekten Wert `x` mod `y`, der immer zwischen 0 und `y - 1` liegt.
- In C führt der Operator >> entweder eine logische oder eine arithmetische Verschiebung nach rechts durch (was nur bei vorzeichenlosen Werten gleichbedeutend ist); der Operator >>> existiert in C nicht.
- Der Komma-Operator, mit dem man in C die sequentielle Ausführung von Teilausdrücken beschreiben kann, existiert nicht.

3.3.4 Fehler und Ausnahmen

- Wenn `new` wegen Speicherplatzmangels kein Objekt erzeugen kann, erhält man einen Fehler des Typs `OutOfMemoryError`.
- Wenn man mit `new` eine Reihe mit negativer Länge erzeugen will, erhält man eine Ausnahme des Typs `NegativeArraySizeException`.
- Ein Feldzugriff, ein Methodenaufruf oder ein Reihenelementzugriff über eine Nullreferenz produziert eine Ausnahme des Typs `NullPointerException`.
- Ein Reihenelementzugriff mit einem unzulässigen Index produziert eine Ausnahme des Typs `ArrayIndexOutOfBoundsException`.
- Eine unzulässige Typumwandlung produziert eine Ausnahme des Typs `ClassCastException` (sofern sie nicht schon vom Übersetzer als unzulässig erkannt wird; vgl. § 5.8).
- Eine ganzzahlige Division durch 0 (bzw. eine entsprechende Modulo-Operation) produziert eine Ausnahme des Typs `ArithmeticException`.
  Eine Gleitkommadivision wie z. B. `1.0/0.0` oder `-1.0/0.0` liefert jedoch einen unendlichen Wert mit entsprechendem Vorzeichen, während `0.0/0.0` den Sonderwert „not a number“ liefert.

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` liefern 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 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 nur direkt oder indirekt in einer passend markierten Schleife (vgl. § 3.4.4)
- `yield expression`
  - Zum Zurückgeben eines Werts in einem `switch`-Ausdruck (vgl. § 3.4.7)
- `return [expression]`
  - Zum vorzeitigen Beenden einer Methode oder eines Konstruktors
  - und/oder zum Zurückgeben eines Resultatwerts
- `throw expression`
  - Zum Werfen einer Ausnahme (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 „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` vorzeitig fortgesetzt werden.
- Anweisungsblock:
  - `{ { declaration | statement } }`
  - Ein Anweisungsblock kann neben Anweisungen auch lokale Variablen- und Klassendeklarationen in beliebiger Reihenfolge enthalten.
- Verzweigung:
  - `if (condition) statement`
  - `if (condition) statement else statement`
  - Anders als in C, muss die Bedingung Typ `boolean` besitzen.

- Schleifen:
  - `while (condition) statement`
  - `do statement while (condition);`
  - `for (init; condition; update) statement`
    - Initialisierungs- (`init`) und Aktualisierungsteil (`update`) können aus mehreren Anweisungs-Ausdrücken bestehen, die durch Komma 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 stehen (die dann nur innerhalb der Schleife gültig ist).
    - Der Ausdruck nach dem Doppelpunkt 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.

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 kontrollierenden Ausdrucks 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
```