5 Mehr Java

Java ist statisch typisiert, d.h., Variablen, Ergebnisse von Ausdrücken, etc. haben einen Datentyp, der schon bei der Kompilierung festgelegt wird.

Java unterscheidet zwei Arten von Typen:

- Basistypen / Primitive Datentypen
  `byte`, `char`, `short`, `int`, `long`, `float`, `double`, `boolean`
- Referenzdatentypen
  kann man auch selber definieren

Beispiel – Statische Typisierung

```
a = 5
a = a + 1
a = "Hello World." # a is now a string
a = a + 1          # runtime error
```

Python

```
int a;
a = 5;
a = "Hello World." // will not compile
```

Java

5.1 Basistypen

Primitive Datentypen

- Zu jedem Basistypen gibt es eine Menge möglicher Werte.
- Jeder Wert eines Basistyps benötigt den gleichen Platz, um ihn im Rechner zu repräsentieren.
- Der Platz wird in Bit gemessen.

Wie viele Werte kann man mit n Bit darstellen?

Primitive Datentypen – Ganze Zahlen

Es gibt vier Sorten ganzer Zahlen:

Typ	*Platz*	*kleinster Wert*	*größter Wert*
`byte`	8	−128	127
`short`	16	−32 768	32 767
`int`	32	−2 147 483 648	2 147 483 647
`long`	64	−9 223 372 036 854 775 808	9 223 372 036 854 775 807

Die Benutzung kleinerer Typen wie `byte` oder `short` spart Platz.

Primitive Datentypen – Ganze Zahlen

Literale:

- dezimale Notation
- hexadezimale Notation (Präfix 0x oder 0X)
- oktale Notation (Präfix 0)
- binäre Notation (Präfix 0b oder 0B)
- Suffix l☹ oder L für long
- '_' um Ziffern zu gruppieren

Verwenden Sie niemals l als long-Suffix, da dieses leicht mit 1 verwechselt werden kann.

_ darf nur **zwischen** Ziffern stehen, d.h. weder am Anfang noch am Ende.

Übung:
Geben Sie eine reguläre Grammatik an, die diese Regeln abbildet

Beispiele

- 192, 0b11000000, 0xC0, 0300 sind alle gleich
- 20_000L, 0xABFF_0078L
- 09, 0x_FF sind ungültig

Primitive Datentypen – Ganze Zahlen

Achtung: Java warnt nicht vor Überlauf/Unterlauf!!!

Beispiel:

```
1 int x = 2147483647; // groesstes int
2 x = x + 1;
3 write(x);
```

liefert: -2147483648

- In realem Java kann man bei der Deklaration einer Variablen ihr direkt einen ersten Wert zuweisen (Initialisierung).
- Man kann sie sogar (statt am Anfang des Programms) erst an der Stelle deklarieren, an der man sie braucht!

Primitive Datentypen – Gleitkommazahlen

Es gibt zwei Sorten von Gleitkommazahlen:

Typ	*Platz*	*kleinster Wert*	*größter Wert*	*signifikante Stellen*
`float`	32	ca. $-3.4 \cdot 10^{38}$	ca. $3.4 \cdot 10^{38}$	ca. 7
`double`	64	ca. $-1.7 \cdot 10^{308}$	ca. $1.7 \cdot 10^{308}$	ca. 15

$$x = s \cdot m \cdot 2^e \qquad \text{mit } 1 \leq m < 2$$

- Vorzeichen s: 1 bit
- reduzierte Mantisse $m - 1$: 23 bit (`float`), 52 bit (`double`)
- Exponent e: 8 bit (`float`), 11 bit (`double`)

Primitive Datentypen – Gleitkommazahlen

Literale:

- dezimale Notation.
- dezimale Exponentialschreibweise (`e`, `E` für Exponent)

Mantisse und Exponent sind dezimal; Basis für Exponent ist 10;

- hexadezimale Exponentialschreibweise. (Präfix `0x` oder `0X`, `p` oder `P` für Exponent)

Mantisse ist hexadezimal; Exponent ist dezimal und muß vorhanden sein; Basis für Exponent ist 2;

- Suffix `f` oder `F` für `float`, Suffix `d` oder `D` für `double` (default is `double`)

Beispiele

- `640.5F` == `0x50.1p3f`
- `3.1415` == `314.15E-2`
- `0x1e3_dp0`, `1e3d`
- `0x1e3d`, `1e3_d`, `0x50.1`

In der hexadezimalen Notation, gibt der Exponent die Anzahl der Bitpositionen an, um die das Komma verschoben wird.

Wenn der Exponent in der hexadeximalen Notation, hexadezimal wäre, wüßten wir nicht, ob ein finales '`f`' zum Exponenten gehört, oder ein `float`-Suffix sein soll.

`0x1e3d` ist ein `int` und keine Gleitkommazahl

`1e3_d` ist ungültig, da '_' nicht zwischen 2 Ziffern steht (`d` ist keine Ziffer sondern das `double`-Suffix)

Primitive Datentypen – Gleitkommazahlen

- Überlauf/Unterlauf bei Berechnungen liefert `Infinity`, bzw. `-Infinity`
- Division Null durch Null, Wurzel aus einer negativen Zahl etc. liefert `NaN`

Weitere Basistypen

Typ	*Platz*	*Werte*
`boolean`	1	`true,false`
`char`	16	alle(?) Unicode-Zeichen

Unicode ist ein Zeichensatz, der alle irgendwo auf der Welt gängigen Alphabete umfasst, also zum Beispiel:

- die Zeichen unserer Tastatur (inklusive Umlaute);
- die chinesischen Schriftzeichen;
- die ägyptischen Hieroglyphen ...

Literale:

- `char`-Literale schreibt man in Hochkommata: `'A'`, `'\u00ED'`,`';'`, `'\n'`.
- `boolean`-Literale sind `true` und `false`.

Die ursprüngliche Idee war, dass `char` alle Unicodezeichen enthält. Nach der Einführung von Java, hat sich der Unicodestandard geändert. Deshalb kann ein `char` nur Zeichen der sogenannten Basic Multilingual Plane speichern. Andere Unicodezeichen werden über Strings codiert.

5.2 Strings

Der Datentyp `String` für Wörter ist ein Referenzdatentyp (genauer eine `Klasse` (dazu kommen wir später)).

Hier nur drei Eigenschaften:

- Literale vom Typ `String` haben die Form `"Hello World!"`;
- Man kann Wörter in Variablen vom Typ `String` abspeichern;
- Man kann Wörter mithilfe des Operators `'+'` konkatenieren.

Beispiel

```
String s0 = "";
String s1 = "Hel";
String s2 = "lo Wo";
String s3 = "rld!";

write(s0 + s1 + s2 + s3);
```

...liefert: Hello World!

5.3 Auswertung von Ausdrücken

Funktionen in Java bekommen Parameter/Argumente als Input, und liefern als Output den Wert eines vorbestimmten Typs. Zum Beispiel könnte man eine Funktion

```
int min(int a, int b)
```

implementieren, die das Minimum ihrer Argumente zurückliefert.

Operatoren sind spezielle vordefinierte Funktionen, die in Infix-Notation geschrieben werden (wenn sie binär sind):

```
a + b = +(a,b)
```

Funktionen, werden hier nur eingeführt, weil wir sie bei der Ausdrucksauswertung benutzen möchten. Eine detaillierte Einführung erfolgt später.

5.3 Auswertung von Ausdrücken

Ein Ausdruck ist eine Kombination von Literalen, Operatoren, Funktionen, Variablen und Klammern, die verwendet wird, um einen Wert zu berechnen.

Beispiele: (`x` z.B. vom Typ `int`)

- `7 + 4`
- `3 / 5 + 3`
- `min(3,x) + 20`
- `x = 7`
- `x *= 2`

Operatoren

Unäre +/--Operatoren konvertieren byte, short, char zuerst nach int.

Man kann keinen legalen Ausdruck bilden, bei der die Assoziativität der Postfix-Operatoren (Gruppe Priorität 2) eine Rolle spielen würde.

Unäre Operatoren:

symbol	*name*	*types*	*L/R*	*level*
++	Post-inkrement	(var) Zahl, char	keine	2
--	Post-dekrement	(var) Zahl, char	keine	2
++	Pre-inkrement	(var) Zahl, char	rechts	3
--	Pre-dekrement	(var) Zahl, char	rechts	3
+	unäres Plus	Zahl, char	rechts	3
-	unäres Minus	Zahl, char	rechts	3
!	Negation	boolean	rechts	3

Die Spalte „L/R“ beschreibt die Assoziativität des Operators.

Die Spalte „level“ die Priorität.

Im Folgenden sind (für binäre Operatoren) beide Operanden jeweils vom gleichen Typ.

„Zahl“ steht hier für einen der Zahltypen byte, short, int, long, float oder double.

Achtung

Diese Beschreibung der Vorrangregeln in Form von Prioritäten für Operatoren findet sich nicht im Java Reference Manual. Dort wird nur die formale kontextfreie Grammatik von Java beschrieben. Die Vorrangregeln leiten sich daraus ab und erleichtern den Umgang mit Ausdrücken, da man nicht in die formale Grammatik schauen muß um einen Ausdruck zu verstehen.

Es gibt im Internet zahlreiche teils widersprüchliche Tabellen, die die Vorrangregeln von Java-Operatoren beschreiben :(. Die gesamte Komplexität der Ausdruckssprache von Java läßt sich wahrscheinlich nicht in dieses vereinfachte Schema pressen.

Prefix- und Postfixoperator

- Die Operatoranwendungen `++x` und `x++` inkrementieren beide den Wert der Variablen `x` (als Seiteneffekt).
- `++x` tut das, bevor der Wert des Ausdrucks ermittelt wird (Pre-Inkrement).
- `x++` tut das, nachdem der Wert ermittelt wurde (Post-Inkrement).
- `b = x++;` entspricht:

```
b = x;
x = x + 1;
```

- `b = ++x;` entspricht:

```
x = x + 1;
b = x;
```

Die Entsprechung gilt z.B. für `int`s. Für `short`s müßte es heißen:

```
b = x;
x = (short) (x + 1);
```

da `x = x + 1` nicht compiliert wenn `x` ein `short` ist.

`(short)` ist hier ein Typecast-Operator, den wir später kennenlernen.

Operatoren

Binäre arithmetische Operatoren:

`byte`, `short`, `char` werden nach `int` konvertiert

symbol	*name*	*types*	*L/R*	*level*
*	Multiplikation	Zahl, char	links	4
/	Division	Zahl, char	links	4
%	Modulo	Zahl, char	links	4
+	Addition	Zahl, char	links	5
-	Subtraktion	Zahl, char	links	5

Konkatenation

symbol	*name*	*types*	*L/R*	*level*
+	Konkatenation	String	links	5

Operatoren

Für Referenzdatentypen (kommt später) vergleichen die Operatoren == und != nur die Referenzen.

Vergleichsoperatoren:

symbol	*name*	*types*	*L/R*	*level*
>	größer	Zahl, char	keine	7
>=	größergleich	Zahl, char	keine	7
<	kleiner	Zahl, char	keine	7
<=	kleinergleich	Zahl, char	keine	7
==	gleich	alle	links	8
!=	ungleich	alle	links	8

Operatoren

Boolsche Operatoren:

symbol	*name*	*types*	*L/R*	*level*
&&	Und-Bedingung	boolean	links	12
||	Oder-Bedingung	boolean	links	13

Operatoren

Zuweisungsoperatoren:

symbol	*name*	*types*	*L/R*	*level*
=	Zuweisung	(links var) alle	rechts	15
*=,/=, %=, +=, -=	Zuweisung	(links var) alle	rechts	15

Für die letzte Form gilt:

$$v \circeq a \quad \Longleftrightarrow \quad v = (type(v)) \ (v \circ a)$$

Operatoren

Ein Seiteneffekt sind Änderungen von Zuständen/Variablen, die durch die Auswertung des Ausdrucks entstehen.

Warnung:

- Eine Zuweisung `x = y;` ist in Wahrheit ein Ausdruck.
- Der Wert ist der Wert der rechten Seite.
- Die Modifizierung der Variablen `x` erfolgt als Seiteneffekt.
- Das Semikolon `';'` hinter einem Ausdruck wirft nur den Wert weg.

In C ist diese Art des Fehlers noch wesentlich häufiger, da auch z.B. `x = 1` (für `int x`) in der Bedingung vorkommen kann. Das Ergebnis des Ausdrucks (1) wird in den boolschen Wert `true` konvertiert. Letzteres ist in Java nicht möglich.

Fatal für Fehler in Bedingungen:

```
boolean x = false;
if (x = true)
   write("Sorry! This must be an error ...");
```

In Java kann man durch das ';' aus den meisten Ausdrücken eine Anweisung machen, die nur den Seiteneffekt des Ausdrucks durchführt.

5.3 Auswertung von Ausdrücken

Assoziativität

- Die Assoziativität entscheidet über die Reihenfolge bei Operatoren gleicher Priorität. (links = der linkeste Operator wird zuerst ausgeführt)
- Alle Operatoren einer Prioritätsgruppe haben dieselbe Assoziativität.
- Bis auf Zuweisungsoperatoren (=, +=, etc.) sind alle binären Operatoren linksassoziativ.
- unäre Operatoren, die ihr Argument rechts erwarten sind rechtsassoziativ
- unäre Operatoren, die ihr Argument links erwarten (postfix-Operatoren ++, --) sind linksassoziativ
- Der ternäre Bedingungsoperator (später) ist rechtsassoziativ

5.3 Auswertung von Ausdrücken

Die Auswertung eines Ausdrucks liefert

- eine Variable (**var**),
- einen reinen Wert (**val**) oder
- void (**void**)

In den ersten beiden Fällen hat der Ausdruck dann einen

- Typ, z.B.: **int**, und einen
- Wert, z.B.: **42**

Für z.B. Zuweisungen muss die Auswertung des Ausdrucks auf der linken Seite eine Variable ergeben!!!

5.3 Auswertung von Ausdrücken

In Java werden Unterausdrücke von links nach rechts ausgewertet. D.h. um den Wert einer Operation zu berechnen:

- werte (rekursiv) alle Operanden von links nach rechts aus
- führe die Operation auf den Resultaten aus

Ausnahmen: ||, &&, und der ternäre Bedingungsoperator :?, werten nicht alle Operanden aus (Kurzschlussauswertung).

Man sollte nie Ausdrücke formulieren, deren Ergebnis von der Auswertungsreihenfolge abhängt!!!

Eine Kurzschlussauswertung ist natürlich ok. Dafür gibt es sehr nützliche Anwendungen.

In C/C++, ist die Auswertungsreihenfolge nicht definiert, d.h., sie ist compilerabhängig.

Den Bedingungsoperator lernen wir später kennen.

5.3 Auswertung von Ausdrücken

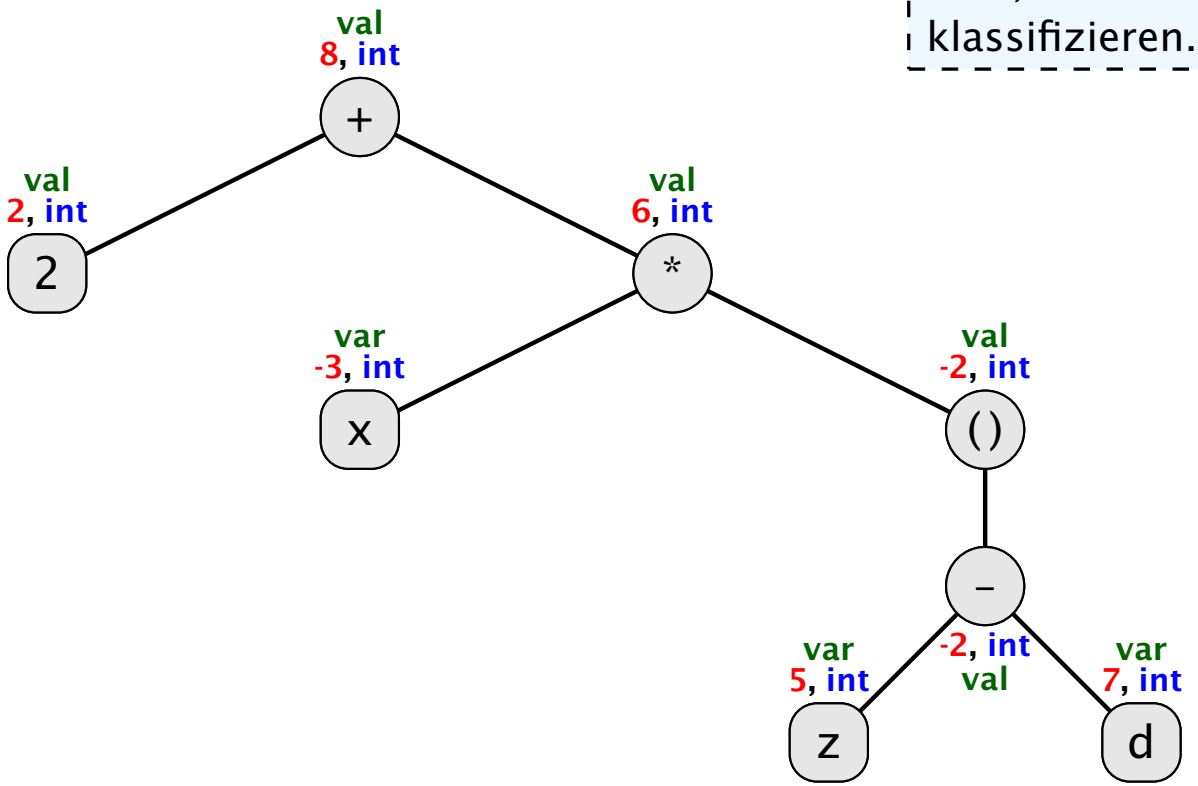
Im Folgenden betrachten wir Klammern als einen Operator der nichts tut:

symbol	*name*	*types*	*L/R*	*level*
( )	Klammerung	alle	links	0

Beispiel: 2 + x * (z - d)

Punkt geht vor Strichrechnung.

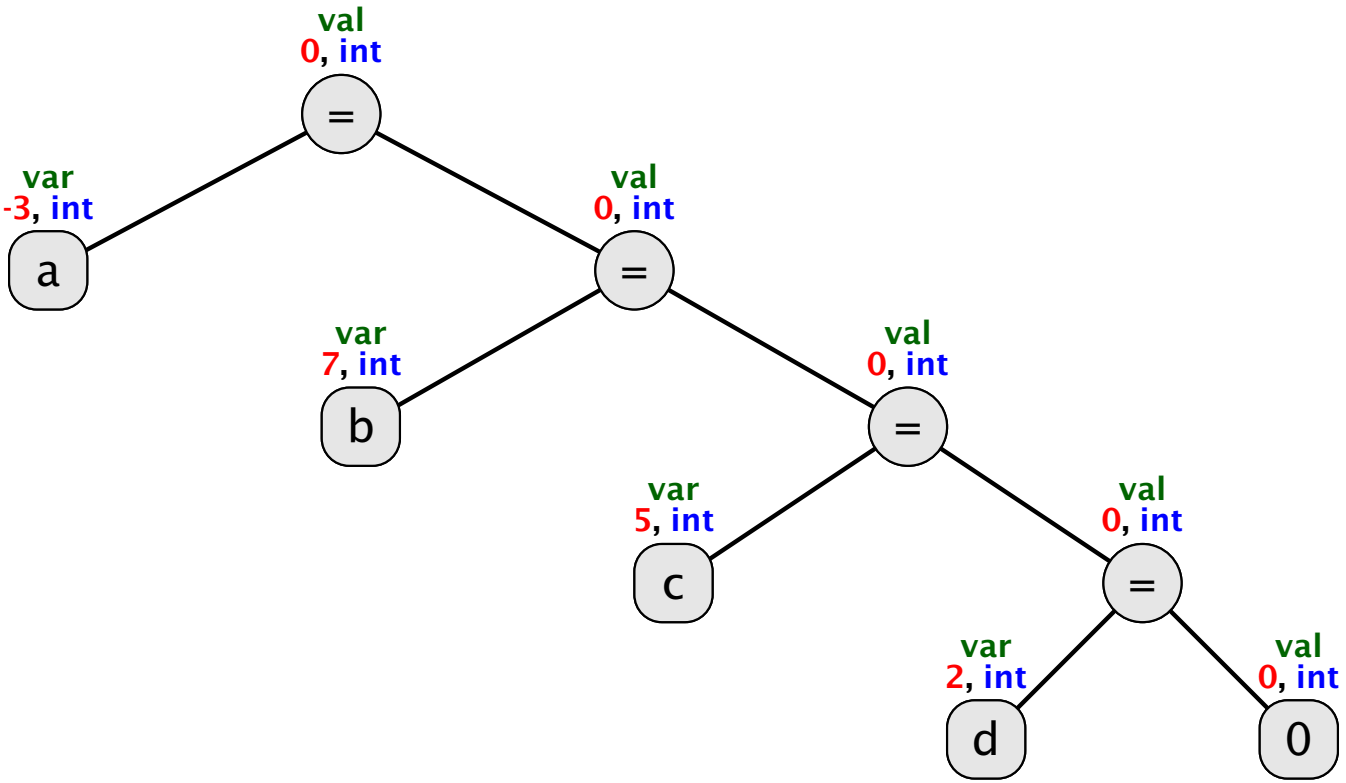
Ganzahlliterale sind vom Typ `int`, wenn nicht z.B. ein `L` angehängt wird, um das Literal als `long` zu klassifizieren.



x -3 d 7 z 5

Beispiel: a = b = c = d = 0

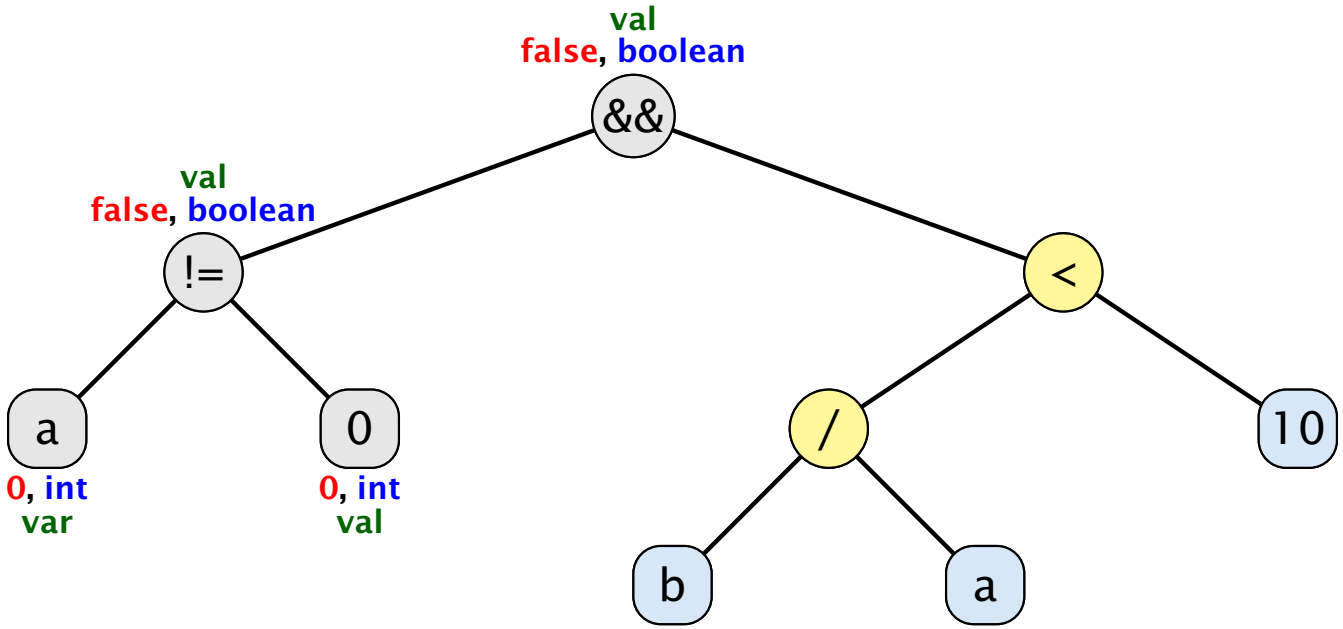
Das funktioniert nur, da der Zuweisungsoperator rechtsassoziativ ist.



a 0 b 0 c 0 d 0

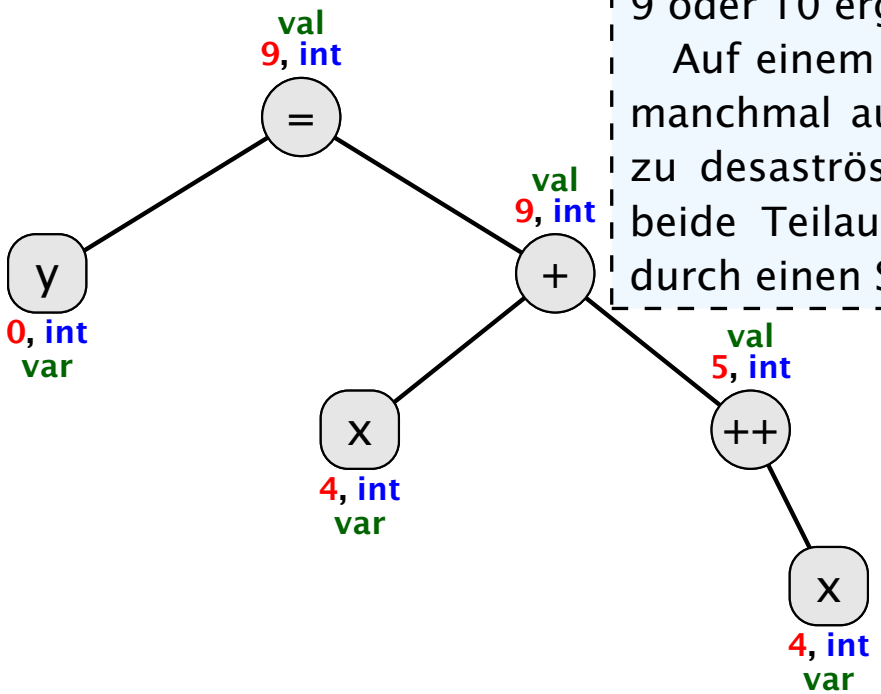
Beispiel: a != 0 && b/a < 10

Die vollständige Auswertung der Operanden würde hier zu einem Laufzeitfehler führen (Division durch Null).
Mit Kurzschlussauswertung ist alles ok.



a 0 b 4

Beispiel: y = x + ++x



In C ist die Reihenfolge der Auswertung von Unterausdrücken nicht definiert. Auf einem sequentiellen Rechner hängt die Reihenfolge vom Compiler ab und in diesem Beispiel könnte dies das Resultat 9 oder 10 ergeben.

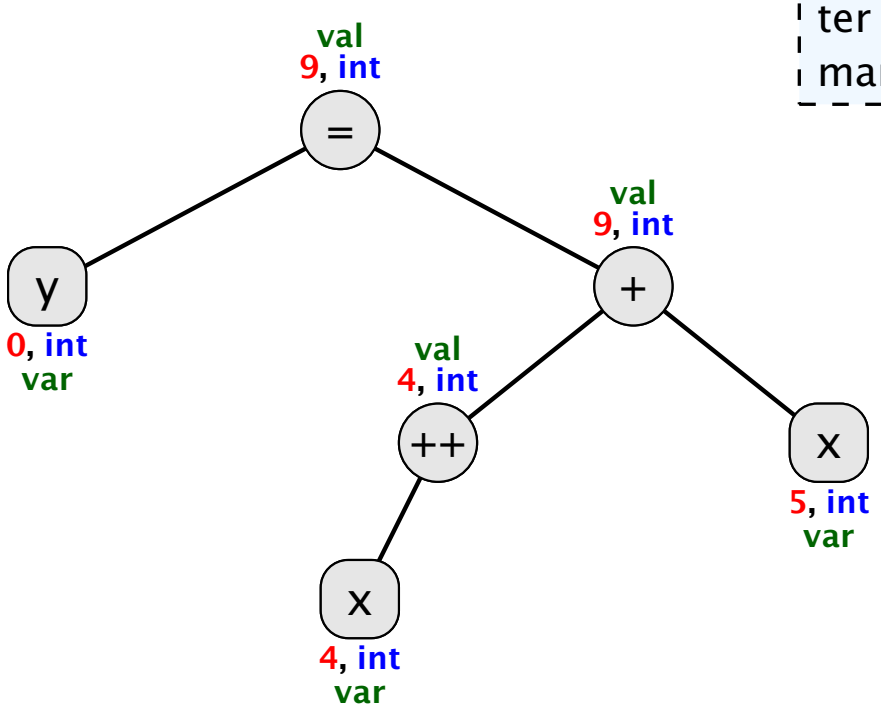
Auf einem Parallelrechner können Teilausdrücke manchmal auch parallel ausgewertet werden, was zu desaströsen Konsequenzen führen kann, falls beide Teilausdrücke eine Variable enthalten, die durch einen Seiteneffekt verändert wird.

x | 5 | y | 9 |

Beispiel: y = x++ + x

Der Postfix-Operator ändert die Variable nach dem der Wert des Teilausdrucks bestimmt wurde.

Wenn die Variable im Ausdruck später nochmal ausgewertet wird, bekommt man den neuen Wert.



x 5 y 9

Impliziter Typecast

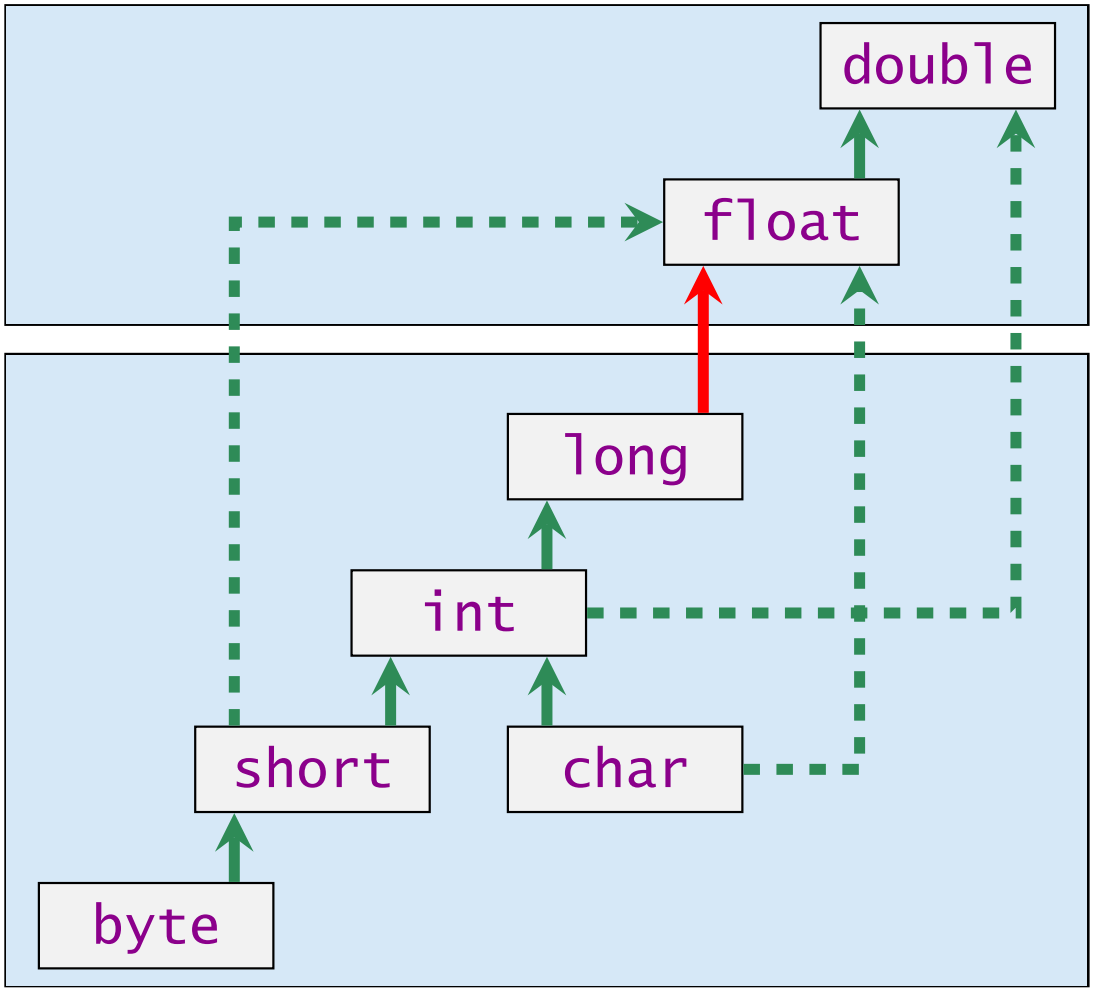
Wenn ein Ausdruck vom TypA an einer Stelle verwendet wird, wo ein Ausdruck vom TypB erforderlich ist, wird

- entweder der Ausdruck vom TypA in einen Ausdruck vom TypB gecasted (impliziter Typecast),
- oder ein Compilerfehler erzeugt, falls dieser Cast nicht (automatisch) erlaubt ist.

Beispiel: Zuweisung

```
long x = 5;
int  y = 3;
x = y; // impliziter Cast von int nach long
```

Erlaubte Implizite Typecasts – Numerische Typen



Gleitkommazahlen

Man nennt diese Art der Casts, widening conversions, da der Wertebereich im Allgemeinen erweitert wird.

ganze Zahlen, char

Keine Typumwandlung zwischen boolean und Zahltypen (weder implizit noch explizit).

Konvertierung von `long` nach `double` oder von `int` nach `float` kann Information verlieren wird aber automatisch durchgeführt.

Welcher Typ wird benötigt?

Operatoren sind üblicherweise überladen, d.h. ein Symbol (+, -, ...) steht in Abhängigkeit der Parameter (Argumente) für unterschiedliche Funktionen.

$$+ : \texttt{int} \longrightarrow \texttt{int}$$
$$+ : \texttt{long} \longrightarrow \texttt{long}$$
$$+ : \texttt{float} \longrightarrow \texttt{float}$$
$$+ : \texttt{double} \longrightarrow \texttt{double}$$

$$+ : \texttt{int} \times \texttt{int} \longrightarrow \texttt{int}$$
$$+ : \texttt{long} \times \texttt{long} \longrightarrow \texttt{long}$$
$$+ : \texttt{float} \times \texttt{float} \longrightarrow \texttt{float}$$
$$+ : \texttt{double} \times \texttt{double} \longrightarrow \texttt{double}$$

$$+ : \texttt{String} \times \texttt{String} \longrightarrow \texttt{String}$$

Es gibt keinen +-Operator für short, byte, char.

Der +-Operator für Strings macht Konkatenation.

Der Compiler muss in der Lage sein während der Compilierung die richtige Funktion zu bestimmen.

Impliziter Typecast

Der Compiler wertet nur die Typen des Ausdrucksbaums aus.

- Für jeden inneren Knoten wählt er dann die geeignete Funktion (z.B. `+ : long` $\times$ `long` $\longrightarrow$ `long` falls ein +-Knoten zwei `long`-Argumente erhält).
- Falls keine passende Funktion gefunden wird, versucht der Compiler durch implizite Typecasts die Operanden an eine Funktion anzupassen.
- Dies geschieht auch für selbstgeschriebene Funktionen (z.B. `min(int a, int b)` und `min(long a, long b)`).
- Der Compiler nimmt die Funktion mit der speziellsten Signatur.

Speziellste Signatur

1. Der Compiler bestimmt zunächst alle Funktionen, die passen könnten (d.h. die vorliegenden Typen können durch widening conversions in die Argumenttypen der Funktion umgewandelt werden).
2. Eine Funktion f_1 ist spezifischer als eine andere f_2, wenn die Argumenttypen von f_1 auch für einen Aufruf von f_2 benutzbar sind (z.B. `min(int, long)` spezifischer als `min(long, long)` aber nicht spezifischer als `min(long, int)`).
   Dieses definiert eine partielle Ordnung auf der Menge der Funktionen.
3. Unter den möglichen Funktionen (aus Schritt 1) wird ein kleinste Element bzgl. dieser partiellen Ordnung gesucht. Falls genau ein kleinstes Element existiert, ist dies die gesuchte Funtion. Andernfalls ist der Aufruf ungültig. (Beachte: Rückgabetyp spielt für Funktionsauswahl keine Rolle).

Ordnungsrelationen

Relation $\preceq$: TypA $\preceq$ TypB falls TypA nach TypB (implizit) gecasted werden kann:

- **reflexiv:** $T \preceq T$
- **transitiv:** $T_1 \preceq T_2 \wedge T_2 \preceq T_3 \Rightarrow T_1 \preceq T_3$
- **antisymmetrisch:** $T_1 \preceq T_2 \wedge T_2 \preceq T_1 \Rightarrow T_1 = T_2$

d.h., $\preceq$ definiert Halbordnung auf der Menge der Typen.

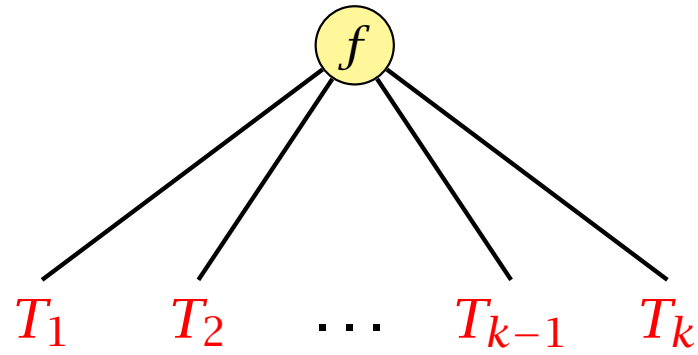
Relation $\preceq_k$: $(T_1, \ldots, T_k) \preceq_k (T'_1, \ldots, T'_k)$ falls $T_i \preceq T'_i$ für alle $i \in \{1, \ldots, k\}$:

- **reflexiv:** $\mathcal{T} \preceq_k \mathcal{T}$
- **transitiv:** $\mathcal{T}_1 \preceq_k \mathcal{T}_2 \wedge \mathcal{T}_2 \preceq_k \mathcal{T}_3 \Rightarrow \mathcal{T}_1 \preceq_k \mathcal{T}_3$
- **antisymmetrisch:** $\mathcal{T}_1 \preceq_k \mathcal{T}_2 \wedge \mathcal{T}_2 \preceq_k \mathcal{T}_1 \Rightarrow \mathcal{T}_1 = \mathcal{T}_2$

d.h., $\preceq_k$ definiert Halbordnung auf Menge der k-Tupel von Typen

Wir betrachten Relation auf der Menge von Parametertupeln für die f implementiert ist. Aus Antisymmetrie folgt, dass keine zwei Funktionen das gleiche k-Tupel an Parametern erwarten.

$$\begin{array}{ll} R_1 & f(\mathcal{T}_1) \\ R_2 & f(\mathcal{T}_2) \\ & \vdots \\ R_\ell & f(\mathcal{T}_\ell) \end{array}$$



$\mathcal{T}_1, \ldots, \mathcal{T}_\ell$ sind k-Tupel von Typen für die eine Definition von f existiert.

$\mathcal{T} = (T_1, \ldots, T_k)$ ist das k-tupel von Typen mit dem f aufgerufen wird.

Menge aller möglichen Funktionen/Tupel:

$$M := \{\mathcal{T}_i | \mathcal{T} \preceq_k \mathcal{T}_i\} \ .$$

Wähle kleinstes Element aus M falls M ein eindeutig kleinstes Element besitzt (sonst Compilerfehler).

Impliziter Typecast – Numerische Typen

Angenommen wir haben Funktionen

```
int min(int a, int b)
float min(float a, float b)
double min(double a, double b)
```

definiert.

```
1 long a = 7, b = 3;
2 double d = min(a, b);
```

würde die Funktion `float min(float a, float b)` aufrufen.

Impliziter Typecast

Bei Ausdrücken mit Seiteneffekten (Zuweisungen, ++ , --) gelten andere Regeln:

Beispiel: Zuweisungen

$$= : \texttt{byte}^* \times \texttt{byte} \longrightarrow \texttt{byte}$$
$$= : \texttt{char}^* \times \texttt{char} \longrightarrow \texttt{char}$$
$$= : \texttt{short}^* \times \texttt{short} \longrightarrow \texttt{short}$$
$$= : \texttt{int}^* \times \texttt{int} \longrightarrow \texttt{int}$$
$$= : \texttt{long}^* \times \texttt{long} \longrightarrow \texttt{long}$$
$$= : \texttt{float}^* \times \texttt{float} \longrightarrow \texttt{float}$$
$$= : \texttt{double}^* \times \texttt{double} \longrightarrow \texttt{double}$$

Es wird nur der Parameter konvertiert, der nicht dem Seiteneffekt unterliegt.

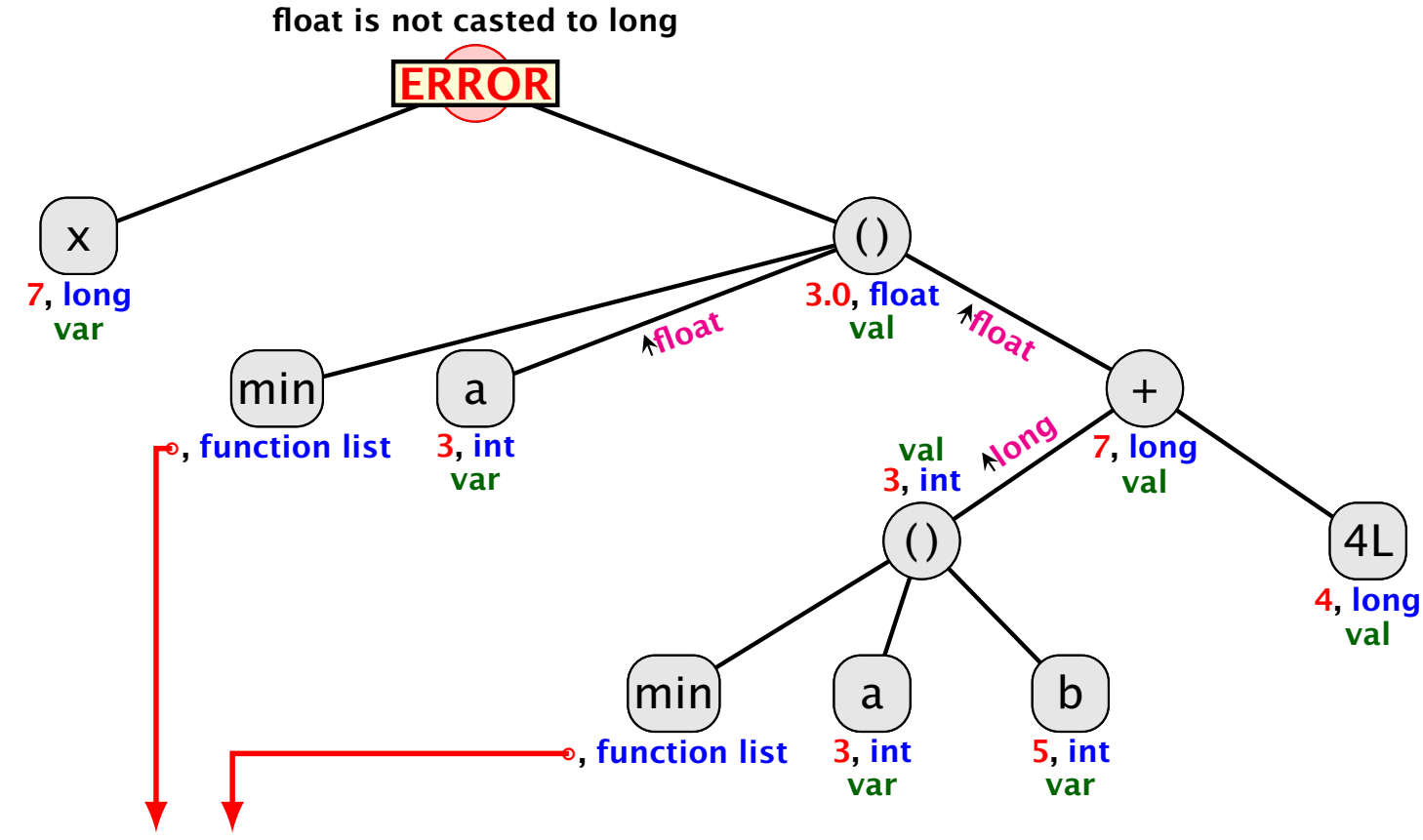
5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Wir modellieren den Funktionsaufrufoperator hier als einen Operator, der beliebig viele Argumente entgegennimmt. Das erste Argument ist der Funktionsname, und die folgenden Argumente sind die Parameter der Funktion. Üblicherweise hat der Funktionsaufrufoperator nur zwei Operanden: den Funktionsnamen, und eine Argumentliste.

Beispiel: x = min(a, min(a,b) + 4L)




```
int min(int,int)
float min(float,float)
double min(double,double)
```

long x 7 int a 3 int b 5

Achtung: Dieses ist eine sehr vereinfachte und teilweise inkorrekte Darstellung. Der eigentliche Prozess, der vom Funktionsnamen zu eigentlichen Funktion führt ist sehr kompliziert. **function list** ist auch kein Typ in Java.

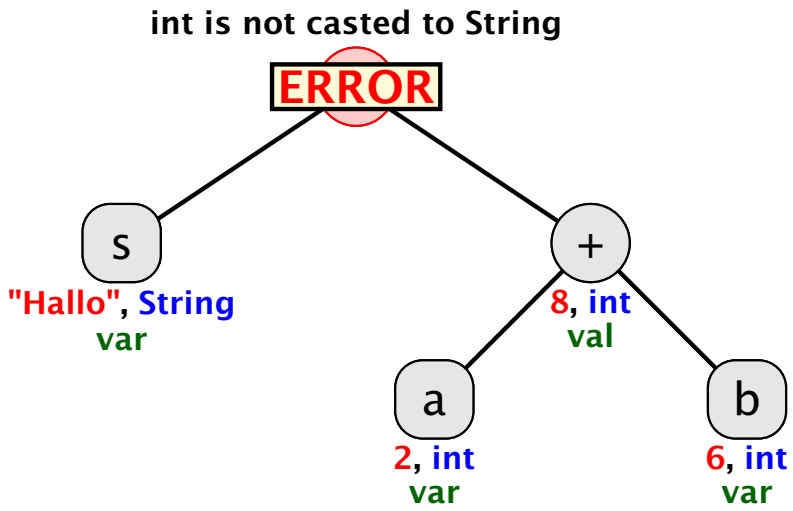
Impliziter Typecast – Strings

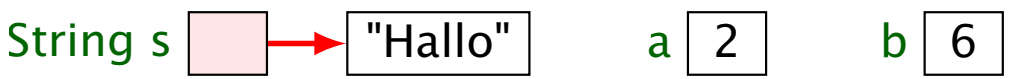
Spezialfall

- Falls beim Operator + ein Typ vom Typ String ist, wird der andere auch in einen String umgewandelt.
  $\Longrightarrow$ Stringkonkatenation.
- Jeder Typ in Java besitzt eine Stringrepresentation.

Funktioniert nicht bei selbstgeschriebenen Funktionen.

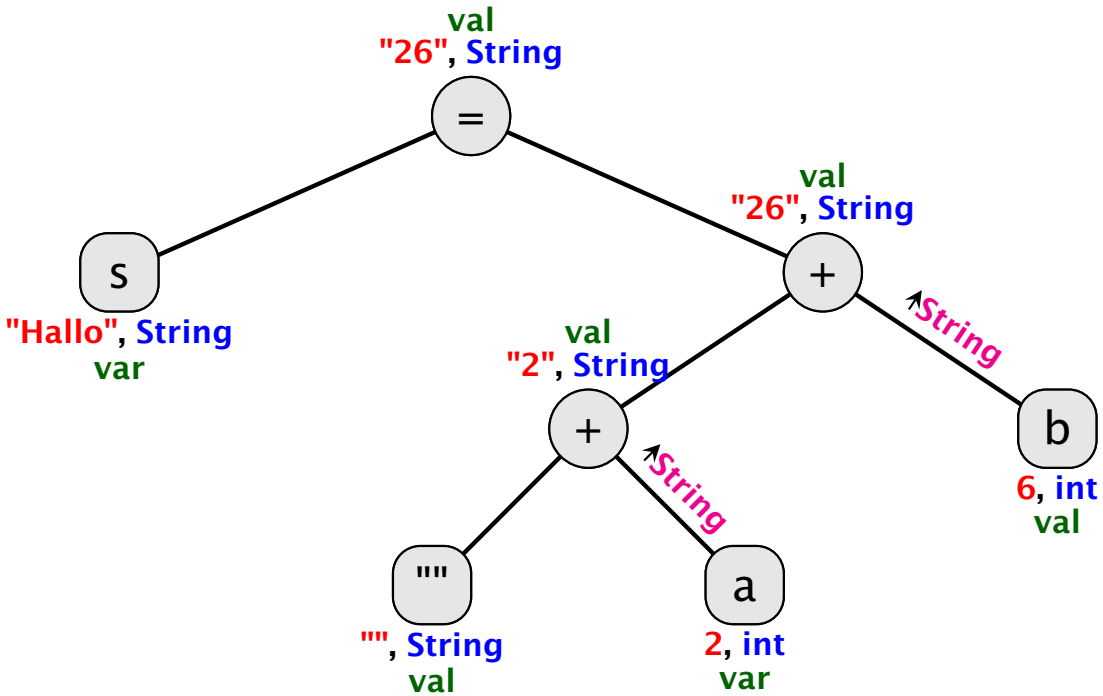
Beispiel: s = a + b





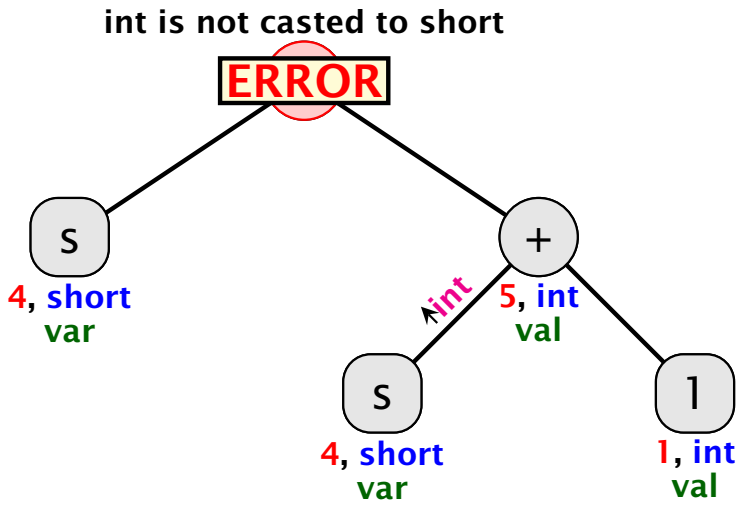
Beispiel: s = "" + a + b

Strings are immutable! Falls eine weitere Referenz auf "Hallo" verweist, hat sich für diese nichts geändert.



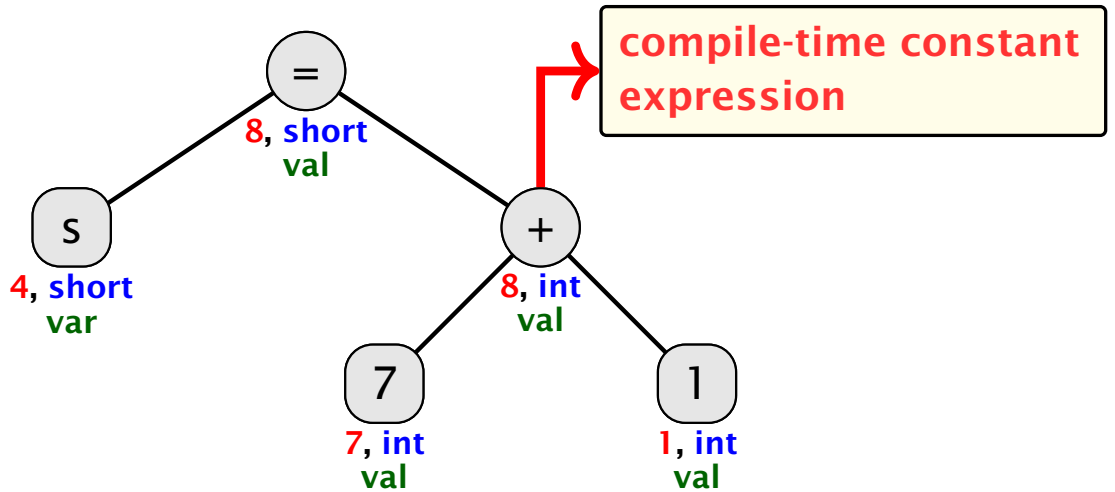
Achtung: vereinfachte Darstellung!!! Eigentlich arbeitet Java mit Ojekten vom Typ StringBuffer um den +-Operator zu realisieren...

Beispiel: s = s + 1



short s | 4 |

Beispiel: s = 7 + 1



short s 8

Wenn der `int`-Ausdruck, der zugewiesen werden soll, zu Compilierzeit bekannt ist, und er in einen `short` „passt“, wird der Cast von `int` nach `short` durchgeführt.

Funktioniert nicht für `long`-Ausdrücke, d.h., `byte b = 4L;` erzeugt einen Compilerfehler.

Expliziter Typecast

symbol	*name*	*type*	*L/R*	*level*
`(type)`	typecast	Zahl, char	rechts	3

Beispiele mit Datenverlust

- `short s = (short) 23343445;`

  Die obersten bits werden einfach weggeworfen...

- ```
  double d = 1.5;
  short  s = (short) d;
  ```

  `s` hat danach den Wert `1`.

...ohne Datenverlust:

- ```
  int   x = 5;
  short s = (short) x;
  ```

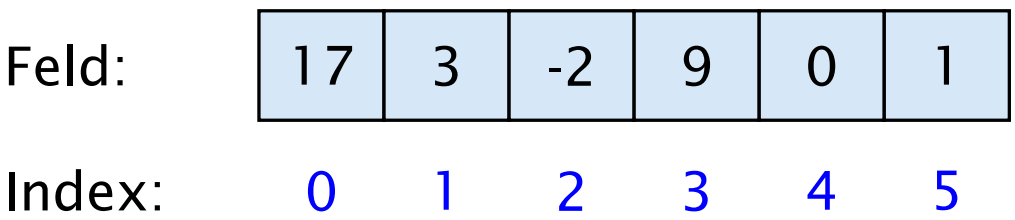
Man kann einen cast zwischen Zahltypen erzwingen (evtl. mit Datenverlust). Typecasts zwischen Referenzdatentypen kommen später.

5.4 Arrays

Oft müssen viele Werte gleichen Typs gespeichert werden.

Idee:

- Lege sie konsekutiv ab!
- Greife auf einzelne Werte über ihren Index zu!



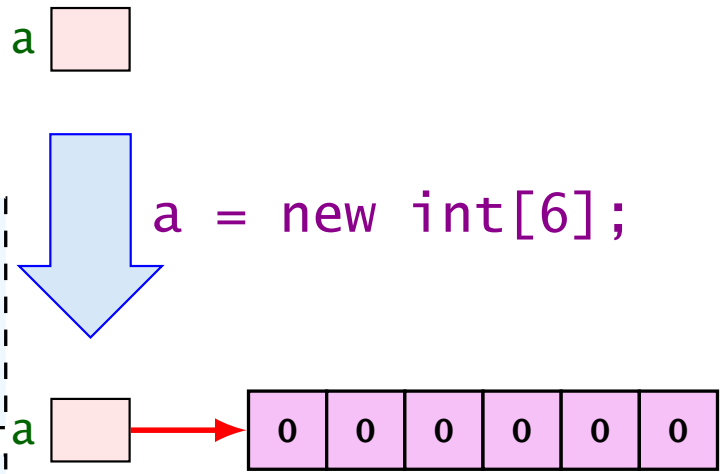
Beispiel

```
1 int[] a; // Deklaration
2 int n = read();
3
4 a = new int[n]; // Anlegen des Felds
5 int i = 0;
6 while (i < n) {
7     a[i] = read();
8     i = i + 1;
9 }
```

Einlesen eines Feldes

Beispiel

- `type[] name;` deklariert eine Variable für ein Feld (array), dessen Elemente vom Typ `type` sind.
- Alternative Schreibweise:
  `type name[];`
- Das Kommando `new` legt ein Feld einer gegebenen Größe an und liefert einen Verweis darauf zurück:

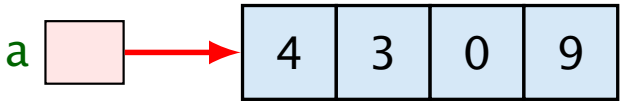


Alles was mit `new` angelegt wird, ist vorinitialisiert. Referenztypen zu `null`, Zahltypen zu `0`, boolean zu `false`.

Man kann auch leere Felder anlegen: `new int[0]`.

Was ist eine Referenz?

Eine Referenzvariable speichert eine Adresse; an dieser Adresse liegt der eigentliche Inhalt der Variablen.



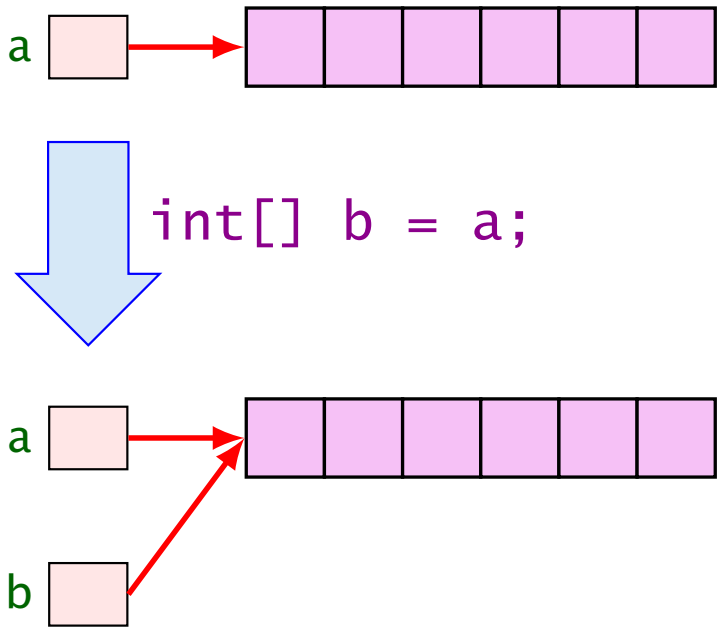
Wir können die Referenz nicht direkt manipulieren (nur über den new-Operator, oder indem wir eine andere Referenz zuweisen).

Eine Referenz zeigt daduch nie auf einen beliebigen Ort im Speicher; sie zeigt immer auf ein gültiges Objekt oder auf das `null`-Objekt.
Wir geben üblicherweise nie den Wert einer Referenzvariablen an, sondern symbolisieren diesen Wert durch einen Pfeil auf die entsprechenden Daten.

	Adresse	Inhalt
	⋮	⋮
	0000 0127	
a:	0000 0128	0000 012C
	0000 0129	
	0000 012A	
	0000 012B	
	0000 012C	0000 0004
	0000 012D	0000 0003
	0000 012E	0000 0000
	0000 012F	0000 0009
	0000 0130	
	⋮	⋮

5.4 Arrays

- Der Wert einer Feld-Variable ist also ein Verweis!!!
- `int[] b = a;` kopiert den Verweis der Variablen `a` in die Variable `b`:



Insbesondere beim Kopieren von Feldern (und anderen Referenztypen) muss man sich dessen immer bewusst sein.

- **Alle nichtprimitive Datentypen sind Referenzentypen, d.h., die zugehörige Variable speichert einen Verweis!!!**

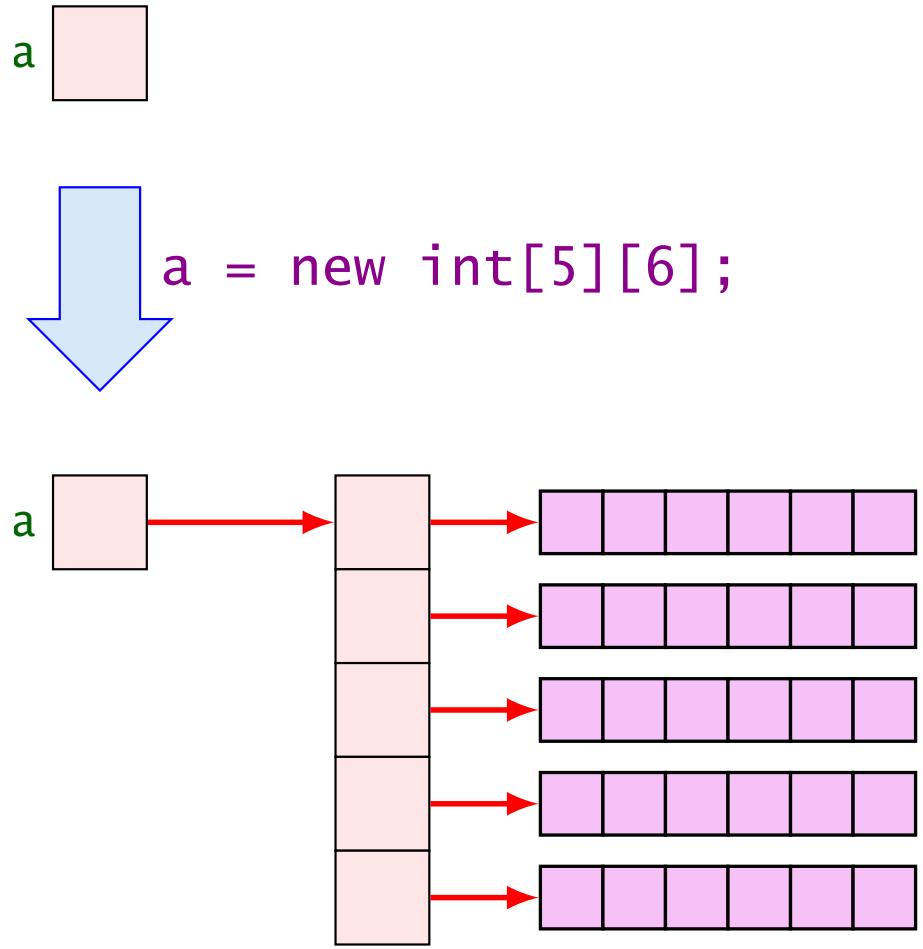
5.4 Arrays

- Die Elemente eines Feldes sind von 0 an durchnummeriert.
- Die Anzahl der Elemente des Feldes `name` ist `name.length`.
- Auf das i-te Element greift man mit `name[i]` zu.
- Bei jedem Zugriff wird überprüft, ob der Index erlaubt ist, d.h. im Intervall $\{0,\ldots,$`name.length-1`$\}$ liegt.
- Liegt der Index außerhalb des Intervalls, wird eine `ArrayIndexOutOfBoundException` ausgelöst (↑Exceptions).

Sie sollten beim Programmieren möglichst nie diese Exception bekommen. In anderen Sprachen (z.B. C/C++) gibt es aus Effizienzgründen keine Überprüfung. Wenn Sie in einer solchen Sprache einen solchen Fehler verursachen, ist der sehr schwierig zu finden.

Mehrdimensionale Felder

- Java unterstützt direkt nur eindimensionale Felder.
- ein zweidimensionales Feld ist ein Feld von Feldern. . .



Der new-Operator

So etwas wie `new int[3][][4]` macht keinen Sinn, da die Größe dieses Typs nicht vom Compiler bestimmt werden kann.

symbol	*name*	*types*	*L/R*	*level*
new	new	Typ, Konstruktor	links	1

Erzeugt ein Objekt/Array und liefert eine Referenz darauf zurück.

1. Version: Erzeugung eines Arrays (Typ ist Arraytyp)
   - `new int[3][7];` oder auch
   - `new int[3][];` (ein Array, das 3 Verweise auf `int` enthält)
   - `new String[10];`
   - `new int[]{1,2,3};` (ein Array mit den `int`s `1`, `2`, `3`)

2. Version: Erzeugung eines Objekts durch Aufruf eines Konstruktors
   - `String s = new String("Hello World!");`

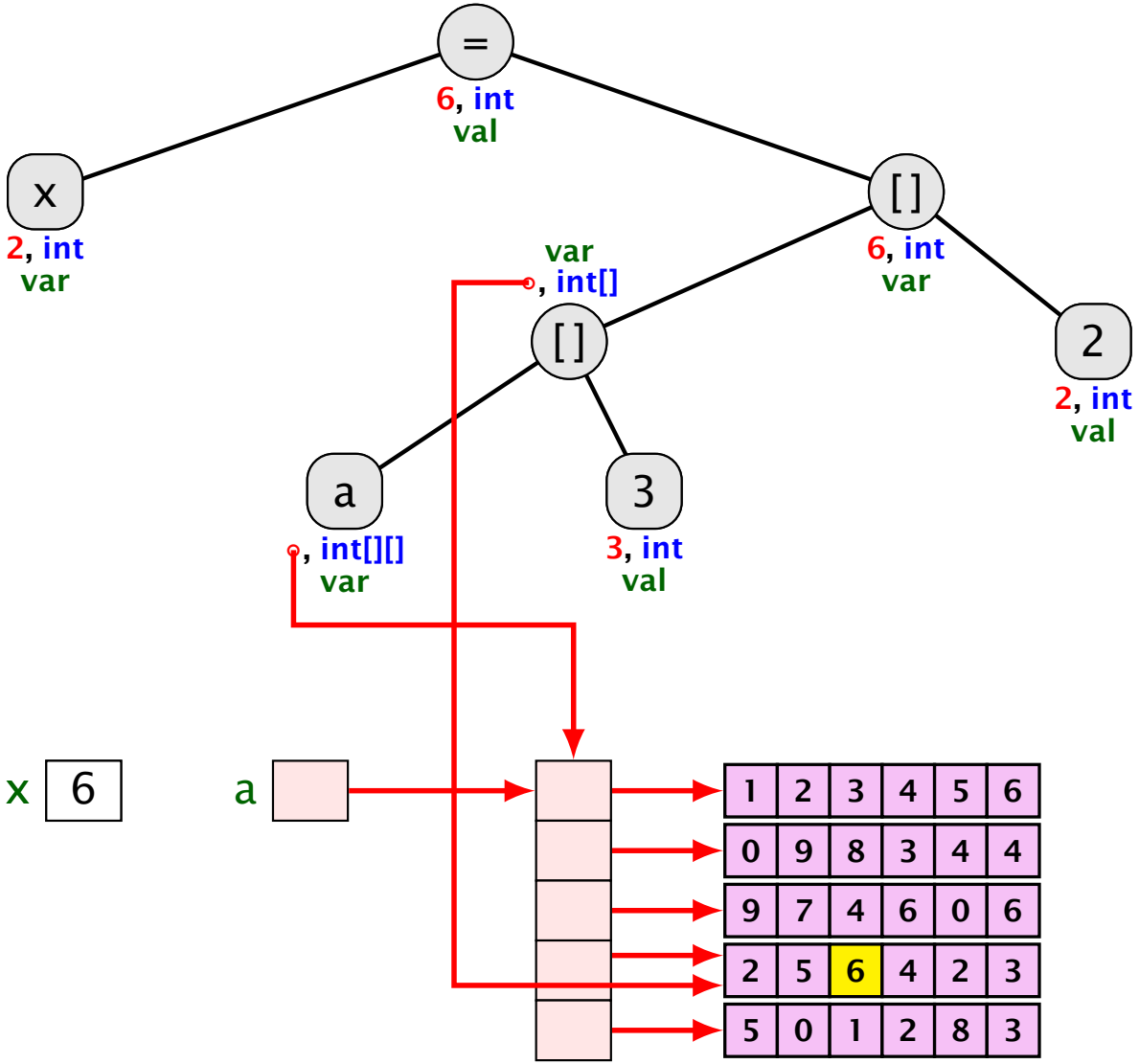
Was genau ein Konstruktor ist kommt später.

Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: x = a[3][2]



Der .-Operator

symbol	*name*	*types*	*L/R*	*level*
.	member access	Array/Objekt/Class, Member	links	1

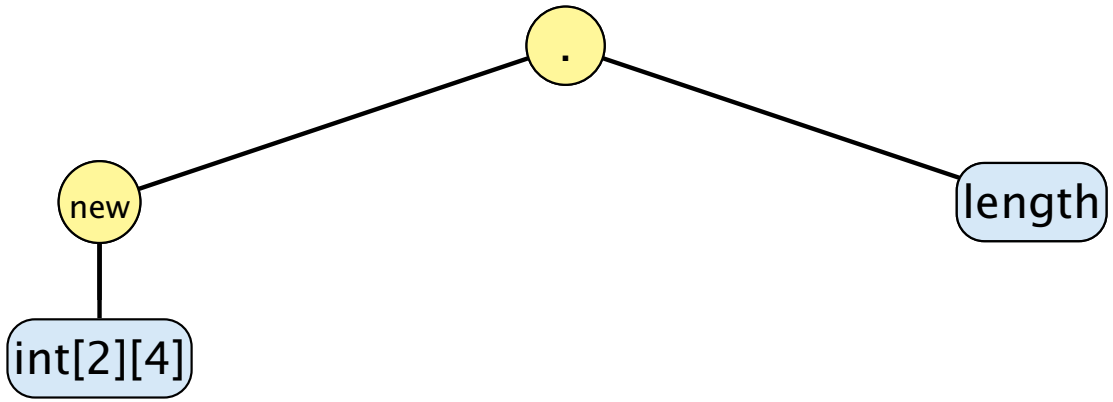
Zugriff auf Member.

Beispiel:

- `x = new int[2][4].length`
  `x` hat dann den Wert 2.

Beispiel: `new int[2][4].length`

Das Parsing für den new-Operator passt nicht in das Schema:



Beachte den Unterschied zwischen `new int[2][3]` und `(new int[2])[3]`. Bei letzterem ist das zweite Klammerpaar ein Index-Operator während es beim ersten Ausdruck zum Typ gehört.

Arrayinitialisierung

1.
```
int[] a = new int[3];
a[0] = 1; a[1] = 2; a[2] = 3;
```

2.
```
int[] a = new int[]{ 1, 2, 3};
```

3.
```
int[] a = new int[3]{ 1, 2, 3};
```

4.
```
int[] a = { 1, 2, 3};
```

5.
```
char[][] b = { {'a','b'}, new char[3], {} };
```

6.
```
char[][] b;
b = new char[][]{ {'a','b'}, new char[3], {} };
```

7.
```
char[][] b;
b = { {'a','b'}, new char[3], {} };
```

5.5 Mehr Kontrollstrukturen

Typische Form der Iteration über Felder:

- Initialisierung des Laufindex;
- `while`-Schleife mit Eintrittsbedingung für den Rumpf;
- Modifizierung des Laufindex am Ende des Rumpfs.

Beispiel

```
1 int result = a[0];
2 int i = 1;          // Initialisierung
3 while (i < a.length) {
4     if (a[i] < result)
5         result = a[i];
6     i = i + 1;    // Modifizierung
7 }
8 write(result);
```

Bestimmung des Minimums

Das For-Statement

```
1 int result = a[0];
2 for (int i = 1; i < a.length; ++i)
3     if (a[i] < result)
4         result = a[i];
5 write(result);
```

Bestimmung des Minimums

Das For-Statement

```
for (init; cond; modify) stmt
```

entspricht:

```
{ init; while (cond) { stmt modify; } }
```

Erläuterungen:

- `++i;` ist äquivalent zu `i = i + 1;`
- die `while`-Schleife steht innerhalb eines Blocks (`{...}`)

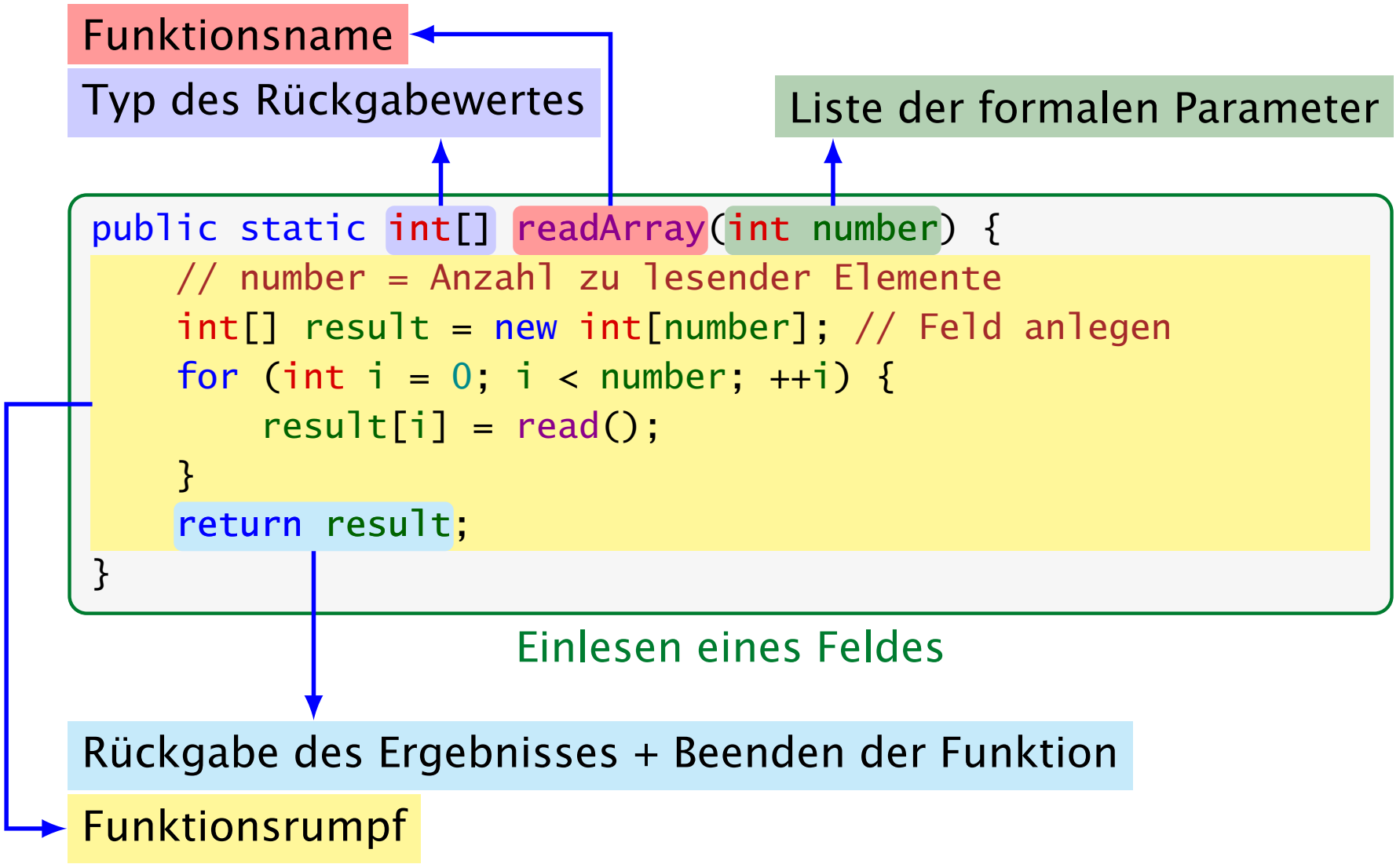
  die Variable `i` ist außerhalb dieses Blocks nicht sichtbar/zugreifbar

5.6 Funktionen und Prozeduren

Oft möchte man:

- Teilprobleme separat lösen; und dann
- die Lösung mehrfach verwenden.

Beispiel


Funktionsname
Typ des Rückgabewertes
Liste der formalen Parameter
public static int[] readArray(int number) {
// number = Anzahl zu lesender Elemente
int[] result = new int[number]; // Feld anlegen
for (int i = 0; i < number; ++i) {
result[i] = read();
}
return result;
}
Einlesen eines Feldes
Rückgabe des Ergebnisses + Beenden der Funktion
Funktionsrumpf

5.6 Funktionen und Prozeduren

Erläuterungen:

- Die erste Zeile ist der Header der Funktion.
- `public` und `static` kommen später
- `int[]` gibt den Typ des Rückgabe-Werts an.
- `readArray` ist der Name, mit dem die Funktion aufgerufen wird.
- Dann folgt (in runden Klammern und komma-separiert) die Liste der formalen Parameter, hier: `(int number)`.
- Der Rumpf der Funktion steht in geschweiften Klammern.
- `return expr;` beendet die Ausführung der Funktion und liefert den Wert von `expr` zurück.

5.6 Funktionen und Prozeduren

Erläuterungen:

- Die Variablen, die innerhalb eines Blocks angelegt werden, d.h. innerhalb von `'{'` und `'}'`, sind nur innerhalb dieses Blocks sichtbar d.h. benutzbar.
- Der Rumpf einer Funktion ist ein Block. Dort deklarierte Variablen nennt man lokale Variablen.
- Die formalen Parameter können auch als lokale Variablen aufgefasst werden.
- Bei dem Aufruf `readArray(7)` erhält der formale Parameter `number` den Wert 7 (aktueller Parameter).

Beispiel

```
public static int min(int[] b) {
    int result = b[0];
    for (int i = 1; i < b.length; ++i) {
        if (b[i] < result)
            result = b[i];
    }
    return result;
}
```

Bestimmung des Minimums

Beispiel

```
public class Min extends MiniJava {
    public static int[] readArray(int number) { ... }
    public static int min(int[] b) { ... }
    // Jetzt kommt das Hauptprogramm
    public static void main(String[] args) {
        int n = read();
        int[] a = readArray(n);
        int result = min(a);
        write(result);
    } // end of main()
} // end of class Min
```

Programm zur Minimumsberechnung

Beispiel

Erläuterungen:

- Manche Funktionen, deren Ergebnistyp `void` ist, geben gar keine Werte zurück – im Beispiel: `write()` und `main()`. Diese Funktionen heißen Prozeduren.
- Das Hauptprogramm hat immer als Parameter ein Feld `args` von `String`-Elementen.
- In diesem Argument-Feld werden dem Programm Kommandozeilen-Argumente verfügbar gemacht.

```
public class Test extends MiniJava {
    public static void main (String[] args) {
        write(args[0]+args[1]);
    }
} // end of class Test
```

Beispiel

Der Aufruf

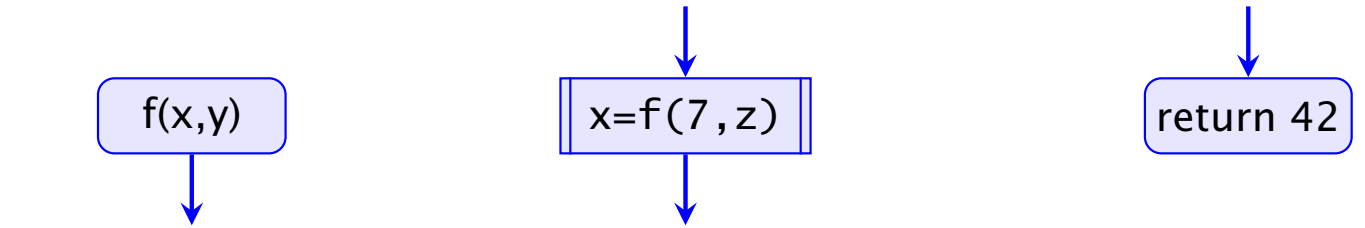
```
java Test "Hel" "lo World!"
```

liefert:

```
Hello World!
```

5.6 Funktionen und Prozeduren

Um die Arbeitsweise von Funktionen zu veranschaulichen erweitern/modifizieren wir die Kontrollflussdiagramme



Startknoten der Funktion f

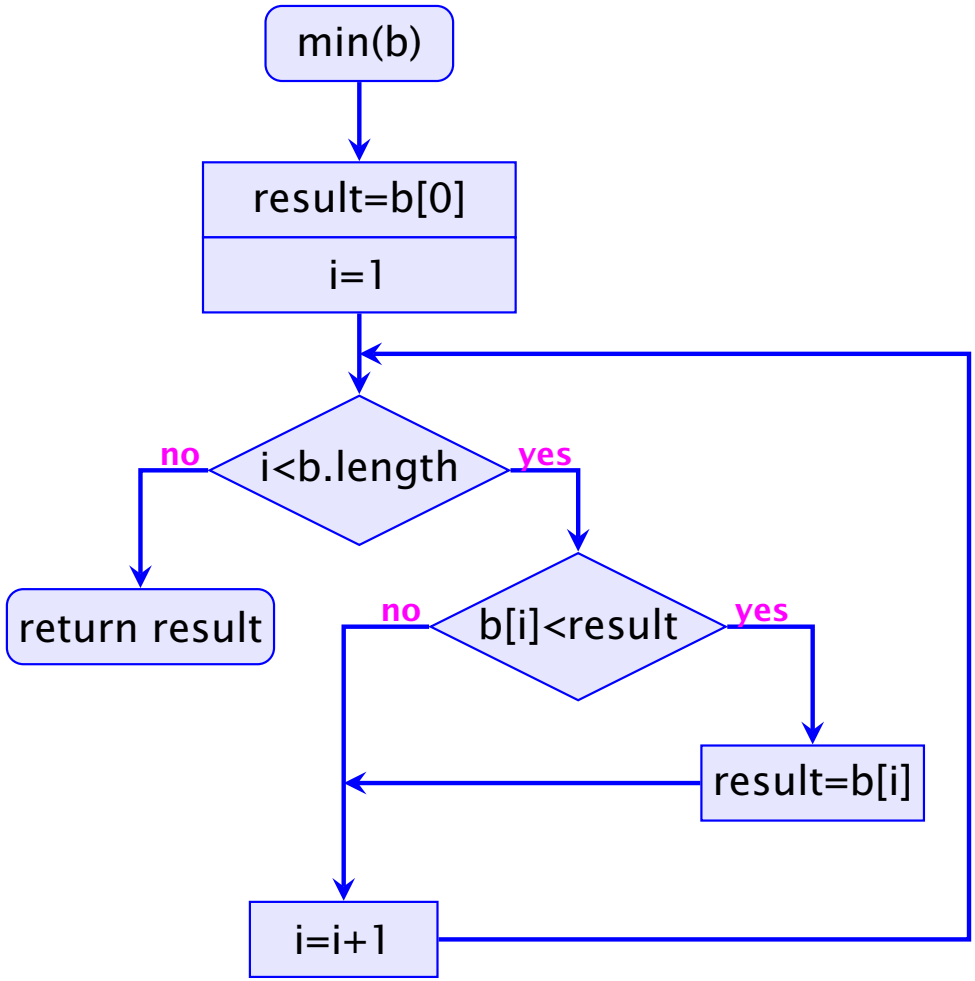
Funktionsaufruf

Endknoten

- Für jede Funktion wird ein eigenes Teildiagramm erstellt.
- Ein Aufrufknoten repäsentiert eine Teilberechnung der aufgerufenen Funktion.

5.6 Funktionen und Prozeduren

Teildiagramm der Funktion `min()`:



5.6 Funktionen und Prozeduren