Objektorientierte Programmierung

Kapitel 6: Wertausdrücke (Expressions)

Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2014/15

http://www.informatik.uni-halle.de/~brass/oop14/

Inhalt

1. Einführung
   - Wertberechnung und Zustandsänderung, Struktur
2. Operatorsyntax
   - Operatorsyntax, Prioritäten, Operatorbaum
   - Arithmetische Operatoren
3. Typen
   - Typ-Korrektheit, Überladene Operatoren, Typ-Anpassung
4. Boolesche Ausdrücke
   - Boolesche Ausdrücke, Vergleichsoperatoren
   - Logische Verknüpfungen (+ Bit-Operatoren)
5. Zuweisungen
   - Zuweisungen, Variable vs. Wert, Typ-Cast, Inkrement
6. Syntax
   - Zusammenfassung/Ausblick (+ Syntaxdiagramme)

Wertausdrücke (1)

- Wertausdrücke (oder Ausdrücke, engl. Expressions) sind syntaktische Konstrukte, die zu einem Element eines Datentyps ausgewertet werden können (z.B. zu einer ganzen Zahl, einer Gleitkommazahl, einem Objekt).
- Beispiel: `i + 1` ist ein Wertausdruck.
- Der Wert ist von der aktuellen Belegung der Variablen abhängig (Inhalt der Variablen).

  Die Werte aller Variablen, bisher erfolgte Ausgaben, aktuelle Eingabeposition, etc. fasst man im Begriff "Zustand" oder "Berechnungs-Zustand" zusammen.
- Wenn z.B. `i` gerade den Wert `3` hat, so hat der Wertausdruck `i + 1` den Wert `4`.

Wertausdrücke (2)

- In Sprachen wie Pascal spezifizieren Wertausdrücke nur die Berechnung eines Wertes, und bewirken aber keine Zustandsänderung (keine Änderung von Variablenwerten).

  Eher unabsichtlich könnte das dort durch den Aufruf einer Funktion doch passieren, wenn die Funktion z.B. eine Zuweisung an eine globale Variable enthält. Aber das gilt dort als schlechter Stil oder ist sogar verboten.

- In Java sind dagegen wie in C/C++ auch Zuweisungen Wertausdrücke, also z.B. `i = 5`.

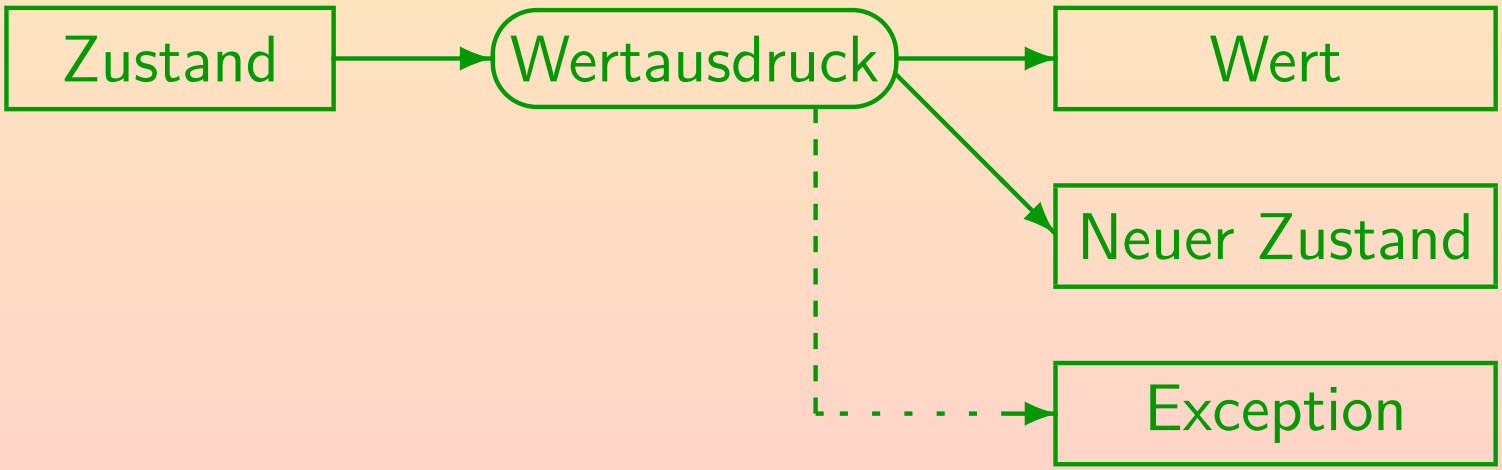
  C erlaubt kompakte Formulierungen (manchmal als kryptisch empfunden).

- Methoden-Aufrufe in Wertausdrücken können selbstverständlich auch Zustandsänderungen bewirken.

  Die Methode kann die Werte von Variablen in den Objekten (Attribute) verändern. In Pascal waren Funktionen (mit Rückgabewert) und Prozeduren (mit Zustands-Änderung) zumindest theoretisch getrennt.

Wertausdrücke (3)

- Es ist auch möglich, dass die Auswertung eines Ausdrucks eine “Exception” (Ausnahme, Fehler) auslöst, z.B. bei Division durch 0.
- Die Bedeutung eines Wertausdrucks kann man also so veranschaulichen:



- Normalerweise ist das primäre Ergebnis der berechnete Wert.

Wertausdrücke (4)

- Elementare Wertausdrücke sind:
  - Konstanten/Datentyp-Literale, z.B. `0.5`.
  - Namen von Variablen, z.B. `x`.
- Aus diesen ganz einfachen Wertausdrücken können komplexere zusammengesetzt werden, u.a. durch
  - Anwendung eines Operators, z.B. `x + 0.5`.
  - Aufruf einer Methode/Funktion, z.B. `Math.sin(x)`.

    In Sprachen wie C heißt es "Funktion", und dort kann man auch einfach "`sin(x)`" schreiben (in diesem Beispiel entspricht es ja auch einer mathematischen Funktion). Die Objektorientierung hat die neue Bezeichnung "Methode" für Funktionen in Klassen eingeführt.
- Da dies wieder Wertausdrücke sind, kann man daraus auch noch komplexere zusammensetzen.

Wertausdrücke (5)

- Semantisch (hinsichtlich der Bedeutung) besteht zwischen einem Methoden/Funktions-Aufruf und der Anwendung eines Operators kein Unterschied:
  - `x + 0.5` bedeutet auch nur, dass die Additionsfunktion auf den aktuellen Wert von `x` und die Zahl `0.5` angewendet wird.
  - Bei passender Deklaration einer Funktion `MyClass.add` könnte man auch `MyClass.add(x, 0.5)` schreiben.

    Eine solche Funktion ist nicht in Java vordefiniert, aber man kann man sie sich leicht selbst deklarieren. In C++ kann Operatoren tatsächlich eine Funktion hinterlegt werden, um die Operatoren auch für neue (benutzer-definierte) Datentypen anwendbar zu machen. In Java geht das nicht.

Wertausdrücke (6)

- Syntaktisch (hinsichtlich der Art, wie man es aufschreibt) besteht natürlich ein Unterschied:
  - Bei einem Funktionsaufruf, z.B. `Math.sin(x)`, wird zuerst der Name der Funktion geschrieben, und dann in Klammern die Eingabewerte (durch `','` getrennt, falls es mehrere sind).

    Da `sin` eine statische Methode der Klasse `Math` ist, schreibt man den Namen der Klasse, dann einen Punkt "`.`", gefolgt vom Namen der Methode. Für normale Methoden würde man vor dem Punkt ein Objekt angeben (entfällt bei aktuellem Objekt bzw. aktueller Klasse).
  - Ein Operator wie `+` wird zwischen seine Eingabewerte geschrieben, z.B. `x + 0.5`.

    Klammern sind nur manchmal notwendig (abhängig von Priorität / Bindungsstärke der Operatoren, siehe unten).

Eingabewerte einer Funktion

- Die Eingabewerte einer Funktion heißen auch die Argumente der Funktion.
- Bei "`Math.sin(0.0)`" wird `Math.sin` also mit dem Argument `0.0` aufgerufen.

  Noch eine Feinheit: Bei "`Math.sin(x+y)`" wäre `x+y` das Argument, tatsächlich übergeben wird aber der Wert dieses Ausdrucks (ein `double`-Wert). Die Verwendung des Begriffes in der Literatur ist aber nicht einheitlich.
- Die Anzahl der Argumente heißt auch die Stelligkeit der Methode/Funktion bzw. des Operators, z.B. ist
  - `Math.sin` eine einstellige Funktion, und
  - die Addition eine zweistellige Funktion.
- Die Argumente eines Operators heißen auch Operanden.

Inhalt

1. Einführung
   - Wertberechnung und Zustandsänderung, Struktur
2. Operatorsyntax
   - Operatorsyntax, Prioritäten, Operatorbaum
   - Arithmetische Operatoren
3. Typen
   - Typ-Korrektheit, Überladene Operatoren, Typ-Anpassung
4. Boolesche Ausdrücke
   - Boolesche Ausdrücke, Vergleichsoperatoren
   - Logische Verknüpfungen (+ Bit-Operatoren)
5. Zuweisungen
   - Zuweisungen, Variable vs. Wert, Typ-Cast, Inkrement
6. Syntax
   - Zusammenfassung/Ausblick (+ Syntaxdiagramme)

Operatorsyntax (1)

- Nach der Anzahl von Operanden unterscheidet man:
  - unäre Operatoren: Ein Argument
  - binäre Operatoren: Zwei Argumente
  - ternäre Operatoren (selten): Drei Argumente
- Manchmal wird auch das gleiche Symbol für unterschiedliche Typen von Operatoren verwendet:
  - `-x`: unäres Minus
  - `x-y`: binäres Minus.

  Es werden ganz unterschiedliche Maschinenbefehle erzeugt (unterschiedliche Funktionen).

Operatorsyntax (2)

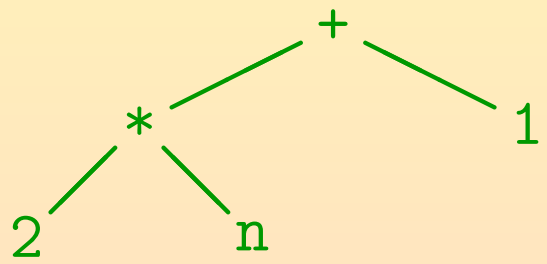
- Nach der Position des Operators (im Vergleich zu den Operanden) unterscheidet man:
  - Präfix-Operatoren: Operator (unär) steht vor dem Operand, z.B. `-x`.
  - Postfix-Operatoren: Operator (unär) steht nach dem Operand, z.B. `i++`.

    Bedeutung von ++: Erhöhe Wert von `i` um 1 (Post-Increment, s.u.).
  - Infix-Operatoren: Operator (binär) steht zwischen den Operanden, z.B. `x + y`.
  - Mixfix-Operatoren: Operator (beliebig) besteht aus mehreren Teilen, z.B. `b ? x : y`.

    Bedeutung: Wenn `b` wahr ist, dann `x`, sonst `y` (bedingter Ausdruck, s.u.).

Operatorsyntax (3)

- Die Struktur von komplexen Wertausdrücken, z.B. `2 * n + 1`, kann man als "Operatorbaum" veranschaulichen:



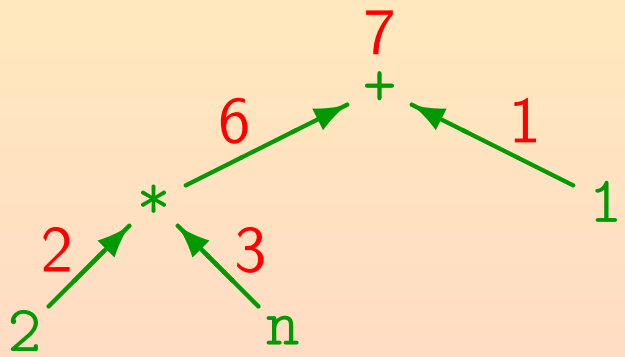
- In der Informatik wachsen Bäume verkehrt herum:
  - Der oberste Knoten ist die Wurzel des Baumes.
  - Hat Knoten X (oben) eine Verbindung zu Knoten Y (unten), so heißt Y Nachfolger/Kind von X.
  - Knoten ohne Nachfolger heißen Blätter.
- Entspricht Diagrammen von Hiearchien, Verzeichnisstruktur.

Operatorsyntax (4)

- Im Operatorbaum ist jeder innere Knoten (d.h. alle Knoten außer den Blättern) mit einem Operator (oder dem Namen einer Funktion/Methode) markiert.
- Blätter sind markiert mit:
  - Namen von Variablen,
  - Konstanten (Datentyp-Literalen), oder
  - Namen von nullstelligen Funktionen/Methoden.
- Die Operanden eines Operators sind die Teilbäume, die mit den Kindknoten des Operators beginnen.
- Klammern tauchen im Operatorbaum nicht auf, sie können aber natürlich seine Struktur beeinflussen.

Operatorsyntax (5)

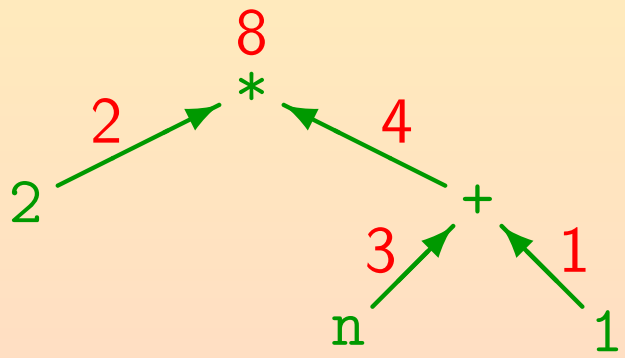
- Werte fließen im Operatorbaum von unten nach oben.
- Hat z.B. `n` gerade den Wert 3, so ergibt sich:



- Der Wert des gesamten Ausdrucks ist der Wert, der sich an der Wurzel des Baums errechnet: 7.

Operatorsyntax (6)

- Man beachte, dass der Ausdruck `2 * n + 1` nicht für folgenden Operatorbaum steht:



- Der Grund ist die bekannte Regel “Punktrechnung vor Strichrechnung”. Falls man die obige Struktur will, muss man Klammern setzen: `2 * (n + 1)`.

Operatorsyntax (7)

- Operatoren haben “Prioritäten”.

  Auch “Bindungsstärken” genannt.

- “*” hat höhere Priorität als “+”.

  Man kann auch sagen: “*” bindet stärker als “+”.

- Um die implizite Klammerung explizit zu machen, kann man so vorgehen, dass in der Reihenfolge der Prioritäten jeder Operator zusammen mit den jeweils kürzestmöglichen Operanden links und/oder rechts eingeklammert wird.

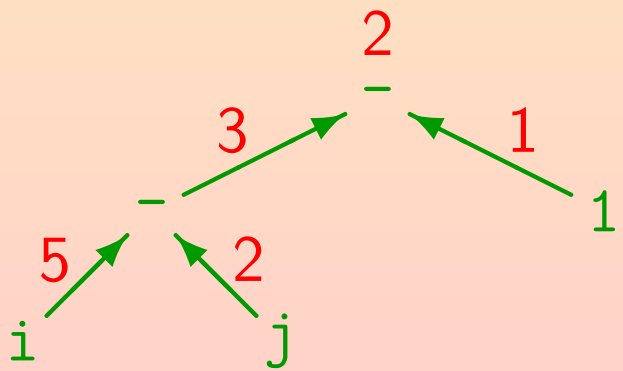
  Wenn Prioritäten als Zahlwerte angegeben werden, ist möglich, dass kleinere Werte eine höhere Priorität bedeuten (z.B. Priorität 1 für die Operatoren, die zuerst drankommen). Wenn man sich unsicher ist, schaue man die Prioritäten für + und * an und denke an “Punktrechnung vor Strichrechnung”.

Operatorsyntax (8)

- Beispiel:
  - Betrachtet sei wieder `2 * n + 1`.
  - Der Operator höchster Priorität ist `*`.
  - Links steht ohnehin nur `2`, rechts bestünde die Wahl zwischen den Operanden `n` und `n + 1`.
  - Es wird der kürzere gewählt, also `n`, und der Operator mit seinen Operanden in Klammern eingeschlossen: `(2 * n) + 1`.
  - Wenn jetzt `+` drankommt, muss links der komplette geklammerte Ausdruck gewählt werden.

Operatorsyntax (9)

- Die obigen Regeln legen noch nicht die Struktur von `i - j - 1` fest.
- In diesem Fall wird von links geklammert, also `(i - j) - 1`.
- Ist `i` $= 5$ und `j` $= 2$, so ergibt sich:



Operatorsyntax (10)

- Um diese implizite Klammerung zu beschreiben, sagt man: “der Operator - ist linksassoziativ”.

  Das Assoziativgesetz gilt aber natürlich gerade nicht für -, denn $(a - b) - c$ ist im allgemeinen verschieden von $a - (b - c)$. Selbst für den Operator +, für den mathematisch das Assoziativgesetz gelten würde, ist es in der Berechnung möglicherweise aufgrund unterschiedlicher Rundung oder arithmetischen Überläufen verletzt. Deswegen legen Java/C/C++ auch hier eine klar definierte implizite Klammerung fest (von links).

- Selbstverständlich darf man auch eigentlich überflüssige Klammern setzen, um die Struktur noch deutlicher zu machen, z.B. `(i - j) - 1`.

  Ein Wertausdruck in Klammern ist wieder ein Wertausdruck. Man könnte sogar `((i - j) - ((1)))` schreiben. Übertrieben viele Klammern sind aber stilistisch schlecht. Der Operatorbaum wird davon nicht beeinflusst.

Operatorsyntax (11)

- In Java/C/C++ sind fast alle Operatoren linksassoziativ, ausgenommen sind nur:
  - Präfixoperatoren, hier gibt es ja nur die Möglichkeit, implizit von rechts zu klammern, z.B. `- -x` bedeutet `-(-x)`.

    Das Leerzeichen zwischen den beiden `-` ist hier wichtig, sonst liefert die lexikalische Analyse den Dekrement-Operator `--`.
  - Zuweisungen, z.B. steht `a = b = c` für `a = (b = c)`.

    Hierzu muss man wissen, dass der Wert einer Zuweisung der zugewiesene Wert ist (s.u.). Der Wert von `c` wird also erst in `b` gespeichert und dann in `a`. Die Regel gilt auch für `+=` etc.
  - Der bedingte Ausdruck, z.B. wird `a?b:c?d:e` als `a?b:(c?d:e)` verstanden.

Operatorsyntax (12)

- Aufgabe:
  - Zeichnen Sie einen Operatorbaum für

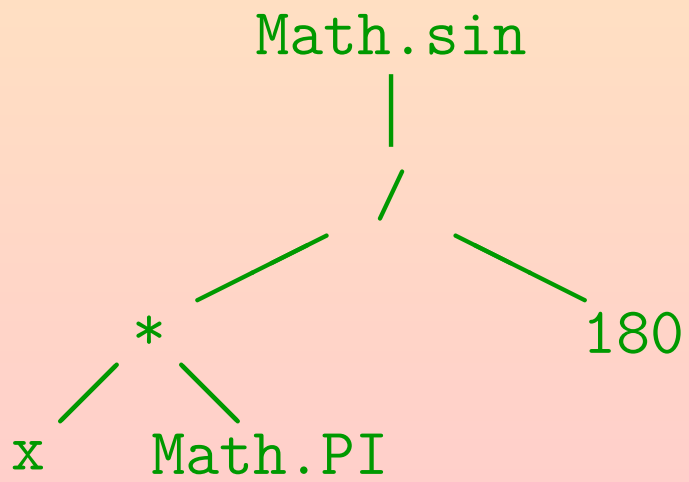
    ```
    a + 2 * (b - c + 1)
    ```

    (die Operatoren `+` und `-` haben gleiche Priorität).
  - Was ist der Wert dieses Ausdrucks für `a=5`, `b=8`, `c=3`?
- Hinweis: Wenn keine Klammern verwendet werden, muss bei einem linksassoziativen Operator der Operator im linken Kindknoten höhere oder gleiche Priorität haben, der Operator im rechten Kindknoten echt höhere Priorität.
- Bemerkung: Die in der Mathematik übliche Notation $2n + 1$ ist in Java/C/C++ (und fast allen anderen Programmiersprachen) nicht erlaubt: Man muss den Multiplikationsoperator explizit schreiben.

Operatorbaum: Erweiterungen (1)

- Auch die Struktur komplexerer Wertausdrücke kann gut in einem Operatorbaum visualisiert werden.
- Prinzip: Die Argumente eines Operators/Funktion/Methode entsprechen den darunterliegenden Teilbäumen.

  Teilbaum: Ein Knoten mit allen darunter hängenden Knoten. Das ist selbst wieder ein Baum und entspricht der Schachtelung von Wertausdrücken.
- Beispiel: `Math.sin(x * Math.PI / 180)`

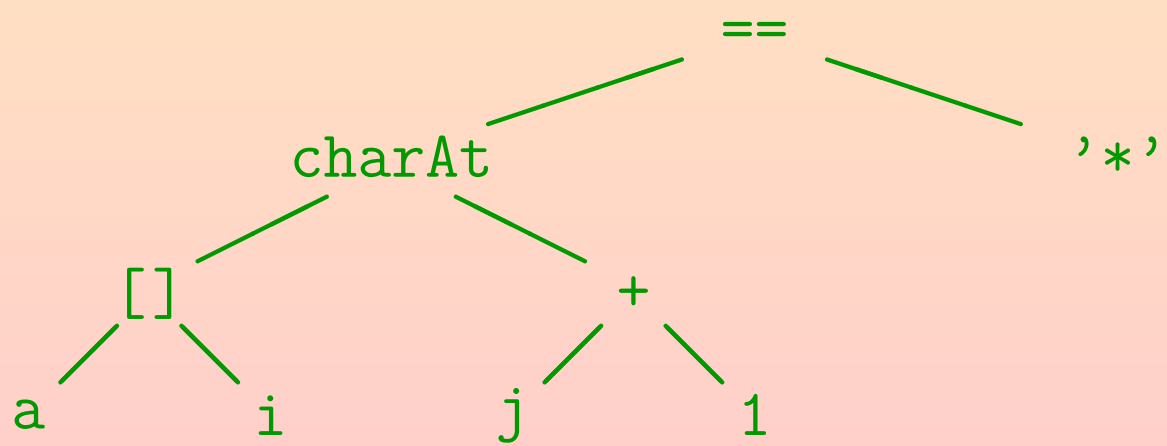


Operatorbaum: Erweiterungen (2)

- Kanten entsprechen Werten, die bei Ausführung des Programms (zur Laufzeit) weitergegeben werden.

  Während bei statischen Methoden wie `Math.sin` der Klassenname "`Math`" nur Bestandteil des Namens ist (zur Compilezeit wichtig, um die Methode zu finden/auszuwählen), hat eine normale Methode zur Laufzeit auch Zugriff auf ein aktuelles Objekt, das mit dem Ausdruck vor dem "`.`" berechnet wird.

- Beispiel: `a[i].charAt(j+1) == '*'`



Prioritätsstufen (von hoch nach niedrig)

1	`o.a, i++, a[], f(), ...`	Postfix-Operatoren
2	`-x, !, ~, ++i, ...`	Präfix-Operatoren
3	`new C(), (type) x`	Objekt-Erzeugung, Cast
4	`*, /, %`	Multiplikation etc.
5	`+, -`	Addition, Subtraktion
6	`<<, >>, >>>`	Shift
7	`<, <=, >, >=, instanceof`	kleiner etc.
8	`==,!=`	gleich, verschieden
9	`&`	Bit-und, logisches und
10	`^`	Bit-xor, logisches xor
11	`|`	Bit-oder, logisches oder
12	`&&`	logisches und (bedingt)
13	`||`	logisches oder (bedingt)
14	`?:`	Bedingter Ausdruck
15	`=, +=, -=, *=, /=, ...`	Zuweisungen

Arithmetische Operatoren (1)

- `+`: Addition
- `-`: Subtraktion
- `*`: Multiplikation
- `/`: Division

  Wenn man `/` auf zwei ganze Zahlen anwendet, erhält man eine ganze Zahl, und zwar wird immer Richtung 0 gerundet, d.h. für positives Ergebnis wird abgerundet, z.B. ist `5/3` = `1`, für negatives Ergebnis wird aufgerundet, z.B. `-5/3` = `-1`. Es ist immer `(a/b)*b + a%b = a` garantiert (außer für `b` = `0`, das gibt einen Fehler: `ArithmeticException`). Ist einer der beiden Operanden eine Gleitkommazahl, erhält man die normale Division (mit einer Gleitkommazahl als Ergebnis).

Arithmetische Operatoren (2)

- %: Divisionsrest (Modulo)

  Z.B. `8 % 3 = 2`.

  In Java ist der Divisionsrest-Operator auch für Gleitkommazahlen definiert, z.B. ist `8.5 % 3.0 = 2.5` (in C++ wäre das ein Typfehler).

- − (unär): Negation/Komplement
- + (unär): Identität
- Es gibt drei Prioritätsstufen:
  - Die unären Operatoren binden am stärksten,
  - dann die Punktrechnung (*, /, %),
  - und zuletzt die Strichrechnung (binäres +, −).

Inhalt

1. Einführung
   - Wertberechnung und Zustandsänderung, Struktur
2. Operatorsyntax
   - Operatorsyntax, Prioritäten, Operatorbaum
   - Arithmetische Operatoren
3. Typen
   - Typ-Korrektheit, Überladene Operatoren, Typ-Anpassung
4. Boolesche Ausdrücke
   - Boolesche Ausdrücke, Vergleichsoperatoren
   - Logische Verknüpfungen (+ Bit-Operatoren)
5. Zuweisungen
   - Zuweisungen, Variable vs. Wert, Typ-Cast, Inkrement
6. Syntax
   - Zusammenfassung/Ausblick (+ Syntaxdiagramme)

Typ-Korrektheit (1)

- Operatoren und Funktionen können nur auf Werte der korrekten Datentypen angewendet werden.
- Wenn z.B. `s` den Typ `String` hat, wird `s / 2` einen Typfehler liefern:

```
operator / cannot be applied to
                    java.lang.String,int
```

- Jede Methode/Jeder Operator ist nur für Eingabewerte von bestimmten Datentypen definiert, und produziert dann jeweils einen Ausgabewert eines definierten Datentyps.

Typ-Korrektheit (2)

- Der Divisions-Operator `/` ist z.B. für Eingabewerte vom Typ `int` definiert, und produziert dann einen Wert vom Typ `int` als Ergebnis.

  Natürlich ist er auch für einige andere Datentypen definiert, s.u.

- Man kann dies in folgender Form aufschreiben:

  $$/ : \texttt{int} \times \texttt{int} \rightarrow \texttt{int}$$

  Dies ist die in der Mathematik übliche Notation, kein Java. Die Spezifikation von Eingabe- und Ausgabetyp einer Methode in der API-Dokumentation haben wir in Kapitel 5 angeschaut, eigene Funktionen behandeln wir später.

- Die Spezifikation konkreter Eingabetypen und des zugehörigen Resultattyps nennt man auch die Signatur des Operators/der Methode (Funktion).

Typ-Korrektheit (3)

- Der gleiche Operator kann mit mehreren verschiedenen Signaturen definiert sein, er könnte dann auch völlig unterschiedliche Funktionen berechnen.
- Man nennt solche Operatoren "überladen".
- Z.B. liefert `7.0 / 2.0` das Ergebnis `3.5`, degegen liefert `7 / 2` den Wert `3`.
- Es gibt also zwei verschiedene Varianten der Division (eigentlich noch mehr, s.u.):
  - `/`: $\texttt{int} \times \texttt{int} \to \texttt{int}$
  - `/`: $\texttt{double} \times \texttt{double} \to \texttt{double}$

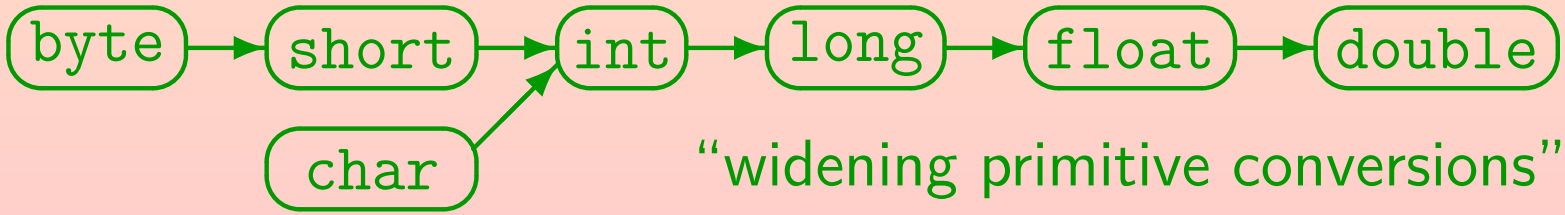
Typ-Korrektheit (4)

- Auch die anderen arithmetischen Operatoren sind mit verschiedenen Typvarianten überladen, selbst wenn es da nicht so offensichtlich ist.
- Z.B. muss der Compiler für eine Integer-Addition und eine Gleitkomma-Addition ganz unterschiedliche Maschinenbefehle erzeugen.

  Die Zahlen sind ja ganz verschieden in Bitmustern codiert.
- Auch Werte unterschiedlicher Längen (etwa `int` und `long`) müssen anders behandelt werden.
- Besonders offensichtlich ist es bei `+`, das neben der Addition auch die String-Konkatenation bedeuten kann.

Typ-Korrektheit (5)

- Die arithmetischen Operatoren `+`, `-`, `*`, `/`, `%` haben insgesamt vier Varianten der Form $T \times T \to T$
  - $\texttt{int} \times \texttt{int} \to \texttt{int}$
  - $\texttt{long} \times \texttt{long} \to \texttt{long}$
  - $\texttt{float} \times \texttt{float} \to \texttt{float}$
  - $\texttt{double} \times \texttt{double} \to \texttt{double}$
- Haben die Operanden eines Operators einen unterschiedlichen (oder zu kleinen) Typ, so wird der kleinere Typ gemäß folgender Liste in den größeren umgewandelt:



Typ-Anpassungen: Details (1) *

- Meistens ändert sich der Wert durch die Typerweiterung nicht, nur bei Umwandlung von `int` und `long` nach `float`, sowie von `long` nach `double` kann es zu Rundungsfehlern kommen.
- Beispiel: Wenn man 123456789 nach `float` umwandelt, erhält man 123456792.

  `float` hat nur 6–7 signifikante Dezimalstellen. Der Rest ist mehr oder weniger zufälliges Rauschen. Da die Zahlen intern binär mit Exponent zur Basis 2 dargestellt werden, und nicht mit Exponent zur Basis 10, hat man nicht einfach 6 Dezimalstellen und dann lauter Nullen. Genauer kann man in den 24 Bit für die Mantisse bei `float` ganze Zahlen bis 2^{24}, das ist etwas mehr als 16 Millionen, exakt darstellen. Bei `double` können alle `int`-Werte exakt dargestellt werden.

Typ-Anpassungen: Details (2)

- In der obigen Liste fehlte z.B.

  $$\texttt{short} \times \texttt{short} \to \texttt{short}$$

- Man kann aber z.B. auch `s + t` schreiben, wenn `s` und `t` den Typ `short` haben.

- Hierzu gibt es "numeric promotions" (Typ-Vergrößerungen): Alle Typen, die kleiner als `int` sind, nämlich `byte`, `short`, `char`, werden automatisch in `int` umgewandelt.

  Die eigentliche Rechnung findet dann mit zwei `int`-Werten statt.

- Beim Beispiel

  ```
  s + 1
  ```

  hat das Datentyp-Literal (die Konstante) `1` den Typ `int`, so dass hier ohnehin die Regel greift, dass der kleinere Typ in den größeren umgewandelt wird.

Typ-Anpassungen: Details (3)

- Der Zweck dieser automatischen Typ-Vergrößerungen ist es, dem Programmierer des Compilers das Leben etwas zu erleichtern: Er muss für die arithmetischen Operatoren nicht ganz so viele Fälle zu behandeln.

  Eventuell war das historisch auch in C wichtig, weil früher Argumenttypen beim Funktionsaufruf nicht unbedingt bekannt waren. Es wurde durch die "integral promotions" z.B. niemals ein `char`-Wert übergeben, sondern immer mindestens ein `int`. So konnte es weniger Typfehler geben.
  In Java sind die Argumenttypen beim Funktionsaufruf aber immer bekannt, und nun können auch Werte der kleinen arithmetischen Typen ohne Umwandlung übergeben werden.
  Falls der Compiler-Entwickler möchte, könnte er natürlich z.B. einen Maschinenbefehl für die Addition einzelner Bytes einsetzen, wenn der Benutzer keinen Unterschied bemerken kann.

Typ-Anpassungen: Details (4) *

- Beispiel:

```
byte b1 = 1;
byte b2 = 2;
byte b3 = b1 + b2;
```

- Der Compiler meldet den Fehler:

```
Test.java:10: possible loss of precision
found   : int
required: byte
        byte b3 = b1 + b2;
                     ^
```

Die beiden `byte`-Werte werden zuerst in `int` umgewandelt, dann wird addiert, das Ergebnis ist `int`, und es ist für den Compiler nicht offensichtlich, dass das Ergebnis klein genug für eine `byte`-Variable ist. Bei den ersten beiden Zuweisungen mit Konstanten rechts erkennt der Compiler dagegen, dass die Werte jeweils in ein `byte` passen (obwohl sie formal `int`-Werte sind). Das gilt aber nur für konstante Ausdrücke (auch mit +).

Typ-Anpassungen: Details (5) *

- Die Regeln sind:
  - Ist `double` der Typ von einem der beiden Operanden, wird der andere Operand in diesen Typ konvertiert.
  - Ist andernfalls einer vom Typ `float`, wird der andere in diesen Typ konvertiert.
  - Trifft auch das nicht zu, aber ist einer vom Typ `long`, wird der andere in diesen Typ konvertiert.
  - Ansonsten sind beide `int` oder werden nach `int` konvertiert.

    Es gibt außerdem spezielle Klassen wie `Integer`, die einen `int`-Wert in ein Objekt verpacken (s.u.). Werden Objekte dieser Klassen in einem arithmetischen Ausdruck verwendet, werden die enthaltenen Werte automatisch "ausgepackt" (vor der Typ-Vergrößerung).
- Beispiel: `x + 1` mit `x` vom Typ `float`. Der `int`-Wert `1` wird nach `float` umgewandelt, dann `float`-Addition.

Typ-Anpassungen: Details (6) *

- Bei Shift-Operationen (s.u.) gibt es auch eine Variante

$$\texttt{int} \times \texttt{long} \rightarrow \texttt{int}$$

  Dort können die beiden Argumente einen unterschiedlichen Typ haben. Der Typ `long` für die rechte Seite ist ohnehin merkwürdig, da man schlecht um mehr als 31 Bit verschieben kann.

- In C++ sind die Regeln wesentlich komplizierter.

  Es gibt dort noch `unsigned`-Typen, und der Wertebereich der Typen ist implementierungsabhängig (z.B. ist `int` auf 16-Bit Maschinen 16-Bit groß). Dadurch ist die Teilmengenbeziehung zwischen den Werten der Typen teilweise implementierungsabhängig (z.B. `unsigned int` passt nicht immer in `long`). Außerdem gibt es weitere automatische Umwandlungen, z.B. von `bool` und von Aufzählungstypen nach `int`. Dagegen gibt es keine "Wrapper-Klassen" wie `Integer` (sie werden auch nicht benötigt, da der Template-Mechanismus in C++ auch mit primitiven Typen funktioniert, s.u.). Bei Java funktioniert `%` auch für `double`, bei C++ nur für ganze Zahlen.

String-Konkatenation

- Der Operator + ist auch noch überladen mit der Signatur

$$\texttt{String} \times \texttt{String} \to \texttt{String}$$

- Er konkateniert dann die beiden Zeichenketten, d.h. erzeugt ein neues `String`-Objekt, was die beiden Zeichenketten "aneinandergeklebt" enthält.
- Für die Operanden von + (und nur dort) findet die Typ-Anpassung "String Conversion" statt:
  - Ist einer der beiden Operanden vom Typ `String`,
  - wird der andere in einen `String` umgewandelt.

    Das geschieht mit der Methode `toString()`, die jedes Objekt zur Verfügung stellt. Werte primitiver Typen werden in ein Objekt der entsprechenden "Wrapper-Klasse" umgewandet, z.B. geschieht die Umwandlung `int`→`String` mit `toString()` der Klasse `Integer`.

Inhalt

1. Einführung
   - Wertberechnung und Zustandsänderung, Struktur
2. Operatorsyntax
   - Operatorsyntax, Prioritäten, Operatorbaum
   - Arithmetische Operatoren
3. Typen
   - Typ-Korrektheit, Überladene Operatoren, Typ-Anpassung
4. Boolesche Ausdrücke
   - Boolesche Ausdrücke, Vergleichsoperatoren
   - Logische Verknüpfungen (+ Bit-Operatoren)
5. Zuweisungen
   - Zuweisungen, Variable vs. Wert, Typ-Cast, Inkrement
6. Syntax
   - Zusammenfassung/Ausblick (+ Syntaxdiagramme)

Boolesche Ausdrücke (1)

- Bisher wurden Wertausdrücke (Expressions) gezeigt, die Zahlen berechneten (arithmetische Ausdrücke).
- Es gibt aber für jeden Datentyp von Java Wertausdrücke, die diesen Typ liefern.

  Z.B. ist eine Variable vom Typ T ein Wertausdruck vom Typ T.
- Wertausdrücke vom Typ `boolean` werden u.a. als Bedingung in `if` und `while` verwendet:

```
if( [          ] ) {
    ...
}
```

- Z.B. ist `i < 100` ein Wertausdruck vom Typ `boolean`. Der Operator < hat hier die Signatur:

$$\texttt{int} \times \texttt{int} \rightarrow \texttt{boolean}$$

Boolesche Ausdrücke (2)

- Selbstverständlich können solche Ausdrücke auch verwendet werden, um den Wert einer Variablen vom Typ `boolean` zu berechnen:

```
boolean b = [          ];
```

- Variablen sind besonders einfache (atomare) Wertausdrücke von ihrem jeweiligen Typ, daher geht z.B. auch:

```
if(b) { ... }
```

- Wertausdrücke werden auch zur Berechnung der Eingabewerte für Methoden/Funktionen (Argumentwerte) benutzt.
  Z.B. hat `println` auch eine Variante für boolesche Werte:

```
System.out.println(i < 10);
```

Dies gibt `true` oder `false` aus (je nachdem, ob `i` kleiner als 10 ist).

Vergleichsoperatoren (1)

- `==`: gleich
- `!=`: verschieden
- `<`: kleiner
- `<=`: kleiner oder gleich (kleinergleich)
- `>`: größer
- `>=`: größer oder gleich (größergleich)
- Diese Operatoren haben geringere Priorität als die arithmetischen Operatoren, z.B. wird `a - b > 10` korrekt als `(a - b) > 10` verstanden.

Vergleichsoperatoren (2)

- Die Vergleichsoperatoren < u.s.w. haben die Signatur(en)

  $$T \times T \rightarrow \texttt{boolean}$$

  wobei T ein Zahl-Datentyp (`int`, `long`, `float`, `double`) ist.

  Da vor dem Vergleich die kleineren numerischen Typen `byte`, `short` und `char` in `int` umgewandelt werden, sind diese auch möglich. Ebenso wird "Unboxing" auf die "Wrapper-Classes" `Integer` u.s.w. angewendet, d.h. der im Objekt gespeicherte Wert wird entnommen und verglichen. Selbstverständlich kann man auch z.B. ein `double` mit einem `int` vergleichen, es wird dann vor dem Vergleich das `int` in den größeren Typ `double` umgewandelt (wie bei +). Vergleiche auf == und != sind bei Gleitkommazahlen wegen Rundungsfehlern und der näherungsweisen Zahldarstellung problematisch.

- Bei == und != sind zusätzlich noch `boolean` und Referenz-Typen (Klassen, Arrays, Interfaces) erlaubt.

Vergleichsoperatoren (3)

- Die aus der Mathematik bekannte Schreibweise

  $$1 \leq n \leq 100$$

  funktioniert in allen mir bekannten Programmiersprachen nicht.

- Wenn man

  ```
  1 <= i <= 100
  ```

  schreibt, wird das implizit von links geklammert:

  ```
  (1 <= i) <= 100
  ```

- Das Ergebnis von `(1 <= i)` ist ein boolescher Wert. Diesen kann man nicht mit einer Zahl (`100`) vergleichen. Der Compiler meldet einen Typfehler.

  In C/C++ ist dagegen `1 <= i <= 100` ein legaler Ausdruck. Dort können boolesche Werte automatisch in `int`-Werte umgewandelt werden (`0` oder `1`).

Symbol für Gleichheitstest (1)

- Beachte: Der Vergleich auf Gleichheit == darf nicht mit der Zuweisung = verwechselt werden.

  In Pascal wird die Zuweisung ":=" geschrieben, dann kann man "=" für den Vergleich nehmen. Da Zuweisungen viel häufiger als Gleichheitsvergleiche sind, hat man sich in C entschieden, "=" für die Zuweisung zu verwenden.

  - Die Zuweisung "`a = b`" speichert den Wert von `b` in der Variablen `a`.
  - Der Vergleich "`a == b`" prüft, ob die Werte von `a` und `b` gleich sind.

    Hier muss a nicht unbedingt eine Variable sein, sondern kann ein beliebiger Ausdruck sein.

- Dies ist ein häufiger Anfängerfehler!

Symbol für Gleichheitstest (2)

- Vergleiche von booleschen Variablen mit `true` und `false` sind umständlich (schlechter Stil). Statt

  ```
  if(b == true) { ... }
  ```

  schreibe man besser

  ```
  if(b) { ... }
  ```

  Wenn die Bedingung umgekehrt erfüllt sein soll, wenn die Variable `false` ist, verwende man die Negation "`!`" (s.u.).

- Die umständliche Version ist auch insofern eine Falle, als

  ```
  if(b = true) { ... }
  ```

  gültiger Java-Code ist, aber nicht tut, was beabsichtigt ist!

  In C/C++/Java ist eine Zuweisung auch ein Wertausdruck (s.u.). Daher kann sie auch als Bedingung verwendet werden (der Wert einer Zuweisung ist der zugewiesene Wert). Hier wird also `true` der Variablen `b` zugewiesen, und anschließend als Wert der Bedingung verwendet.

Symbol für Gleichheitstest (3)

- Bei C/C++ tritt dieses Problem häufiger auf, weil dort ein `int`-Wert auch als Wahrheitswert interpretiert werden kann:

  ```
  if(a = b) { ... }
  ```

  war z.B. auch für `int`-Variablen `a` und `b` legal.

- Das geht in Java wegen der strikteren Typ-Struktur nicht: Die Bedingung in `if` und `while` muss ein Ausdruck vom Typ `boolean` sein.

  C hatte keinen booleschen Datentyp, sondern hat dafür immer `int` verwendet, C++ wollte kompatibel sein und hat den Typ `bool` etwas halbherzig eingeführt. Da dieser Fehler häufig vorgekommen ist, haben bessere Compiler eine Warnung ausgegeben. Warnungen unterscheiden sich von Fehlermeldungen dadurch, dass das Programm trotzdem übersetzt wird (es ist ja legal), und man die Warnungen abschalten kann. Bei Java kann das nur für boolesche Variablen vorkommen, der Compiler gibt aber auch dann keine Warnung aus (auch nicht mit `-Xlint:all`).

Logische Operatoren (1)

- `&&`: Logisches "und" (Konjunktion).

P	Q	P && Q
false	false	false
false	true	false
true	false	false
true	true	true

P && Q ist dann und nur dann wahr, wenn P und Q beide wahr sind.

- Signatur: `boolean` $\times$ `boolean` $\rightarrow$ `boolean`.
- `&&` hat eine geringere Priorität als die Vergleiche.
- Z.B. funktioniert `1 <= i && i <= 100` wie erwartet.

Logische Operatoren (2)

- `||`: Logisches "oder" (Disjunktion).

P	Q	P `||` Q
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

P `||` Q ist wahr genau dann, wenn mindestens einer der Operanden P und Q wahr ist.

- Signatur: $\texttt{boolean} \times \texttt{boolean} \to \texttt{boolean}$.
- `||` hat eine geringere Priorität als `&&`.

  `&&` entspricht in logischen Ausdrücken der Punktrechung, `||` der Strichrechnung.

  Ohne Klammern bekommt man eine Disjunktion von Konjunktionen.

Logische Operatoren (3)

- `!`: Logisches "nicht" (Negation).

P	! P
`false`	`true`
`true`	`false`

`!` P ist wahr genau dann, wenn P falsch ist.

- Signatur: `boolean` $\rightarrow$ `boolean`.
- `!` hat eine sehr hohe Priorität.

  Wie alle Präfixoperatoren.

- Z.B. sind bei `!(n>100)` die Klammern nötig.

  Natürlich könnte man auch einfach `n <= 100` schreiben.

Logische Operatoren (4)

- C/C++/Java garantieren, dass bei P && Q der zweite Operand (Q) nur dann ausgewertet wird, wenn der erste (P) wahr ist ("short-circuit-evaluation").

  Wenn `P` falsch ist, steht das Ergebnis der Konjunktion ja schon fest: Es ist sicher falsch. Man braucht `Q` nicht mehr auszuwerten. Bei Pascal wird das nicht garantiert. Java hat auch noch einen zweiten Konjunktionsoperator `&`, bei dem immer beide Operanden ausgewertet werden (→ nächste Folie).

- Z.B. kann `n != 0 && a/n > 2` keinen Fehler geben.

  In Pascal müßte man dagegen explizit ein `if` verwenden, wenn man sichergehen möchte, dass auch ein anderer Compiler (bzw. eine neue Compilerversion) niemals die Division ausführt, wenn `n` gleich `0` ist.
  In Java nennt man `&&` auch den "conditional conjunction operator".

- Entsprechend wird bei `||` der zweite Operand nur dann ausgewertet, wenn der erste falsch ist.

Logische Operatoren (5)

- Java hat noch folgende Operatoren zur Verknüpfung boolescher Werte, die garantiert immer beide Operanden auswerten (wieder `boolean` $\times$ `boolean` $\rightarrow$ `boolean`):

  Das ist auch anders als in Pascal, weil dort diese Frage offen gelassen ist, so dass der Programmierer des Compilers entscheiden kann (Freiraum für mögliche Optimierungen). Wenn die Bedingungen einfach sind, sind die Varianten auf dieser Folie auf modernen (pipelined) CPUs vermutlich schneller, weil es nicht zu falsch vorhergesagten Sprüngen kommt.

  - `P & Q`: Logisches und: `P` und `Q` müssen beide wahr sein.
  - `P | Q`: Logisches oder: mindestens eins muss wahr sein.
  - `P ^ Q`: Logisches "exklusiv-oder": genau eins muss wahr sein.

    Dies hat die gleiche Wirkung wie `P != Q`. Entsprechend ist `P == Q` die Äquivalenz: `P` und `Q` sind entweder beide wahr, oder beide falsch.

Bit-Operatoren (1)

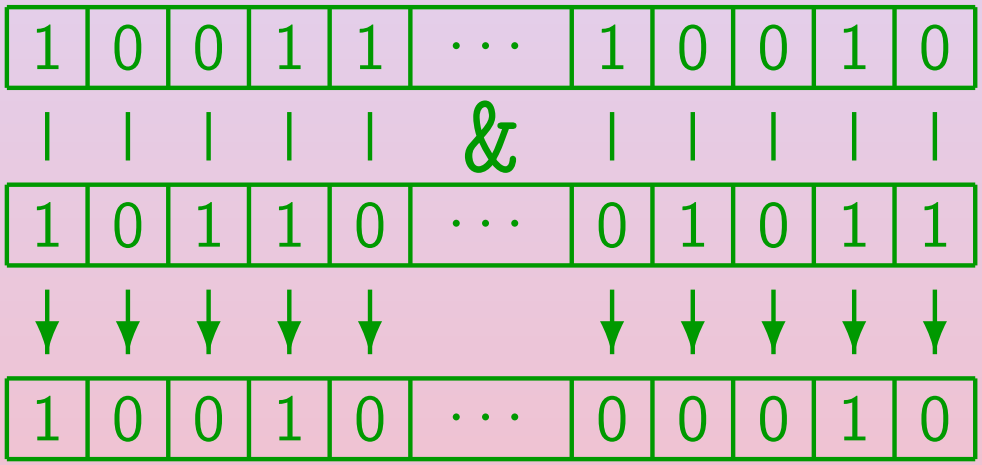
- Man kann ganze Zahlen auch als Folgen von Bits auffassen.
- Z.B. wird `5` intern (im Dualsystem) als `0...00101` dargestellt.

  Das am weitesten rechts stehende Bit hat den Wert $2^0 = 1$, das nächste den Wert $2^1 = 2$, das dritte den Wert $2^2 = 4$, u.s.w. (im Dezimalsystem werden entsprechend Zehner-Potenzen verwendet).

- Die üblichen logischen Verknüpfungen können jetzt auf jede Bitposition angewendet werden, wobei `1` "`true`" entspricht und `0` "`false`".
- Da `int` 32 Bit groß ist, können mit einem Maschinenbefehl z.B. 32 "und"-Verknüpfungen durchgeführt werden.

Bit-Operatoren (2)

- Beispiel:



- Z.B. können Mengen (mit bis zu 32 Elementen in der Grundmenge) so effizient repräsentiert werden.

  Jede Bitposition steht für ein Element der Grundmenge. Ist das Bit 1, so ist das Element enthalten, ist es 0, so ist es nicht enthalten. & wäre dann der Mengenschnitt $\cap$, und | die Vereinigung $\cup$.

Bit-Operatoren (3)

- Die bitweisen logischen Verknüpfungen sind:
  - `&`: Bit-und
  - `|`: Bit-oder
  - `^`: Bit–exklusiv-oder (XOR)
  - `~`: Bit-Komplement (Negation)

A	B	A & B	A | B	A ^ B	~A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Bit-Operatoren (4)

- Alle binären Bit-Operatoren haben die Signatur(en)

$$T \times T \to T,$$

wobei T einer der folgenden Typen ist: `int`, `long`.

Es geht auch `boolean`, aber dann fasst man es als logische Verknüpfung auf. Die Wirkung entspricht aber einer 1 Bit breiten Zahl.

- Der Präfixoperator `~` hat natürlich nur ein Argument: Die Signatur ist: $T \to T$ (mit T wie oben).
- Z.B. würde die Anwendung eines Bit-Operators auf einen `float`-Wert einen Typfehler geben.

Bit-Operatoren (5)

- Z.B. kann man mit

  ```
  if((s & 0x8) != 0) { ... }
  ```

  testen, ob das Bit an Position 4 gesetzt ist.

  - Falls es gesetzt ist, liefert `s & 0x8` das Ergebnis `0x8`.
  - Falls es nicht gesetzt ist, ist das Ergebnis 0.

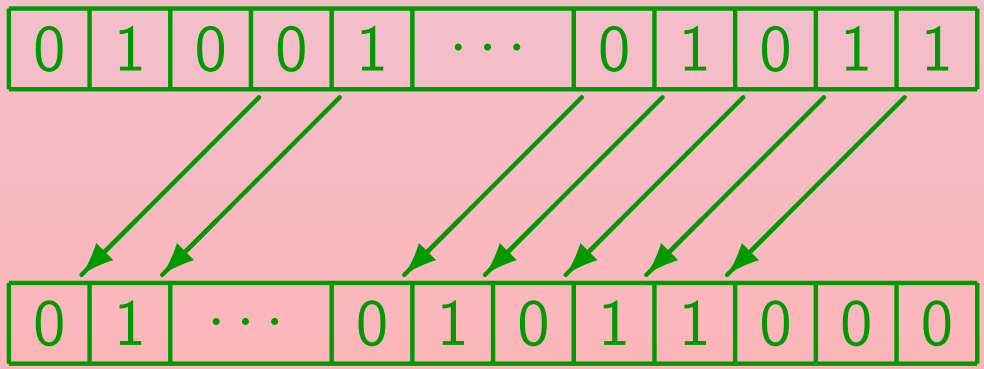
  Man könnte auch `(s & 0x8) == 0x8` schreiben.

  Die Hexadezimalnotation ist natürlich nicht nötig, aber so braucht man keine großen Zweierpotenzen auswendig zu lernen: Jede Ziffer entspricht 4 Bit. Zum Beispiel ist `0x8` der Wert `1000` im Dualsystem.

  Die Klammern um "`(s & 0x8)`" sind hier notwendig, da die Bitoperationen eine geringere Priorität als die Vergleiche haben. Das widerspricht eigentlich der Intuition, aber sie werden ja gleichzeitig als logische Operationen eingesetzt, und diese sollten geringere Priorität als die Vergleiche haben.

Bit-Operatoren (6)

- Es gibt noch drei weitere Bit-Operationen:
  - `<<`: Links-Shift.
  - `>>`: Rechts-Shift mit Vorzeichen-Erhaltung.
  - `>>>`: Rechts-Shift mit 0-Erweiterung.
- Z.B. liefert `n << 3` den Wert der Zahl `n` um drei Bit-Positionen nach links verschoben.



Bit-Operatoren (7)

- Ein Shift um drei Bit-Positionen nach links entspricht einer Multiplikation mit $8 = 2^3$.

  Üblicherweise sind Shift-Operationen schnell, während Multiplikationen einige Taktzyklen länger dauern. Wenn der Compiler eine Multiplikation mit einer Zweierpotenz entdeckt, wird er selbständig einen Shift-Befehl benutzen.

- Beim Rechts-Shift gibt es zwei Varianten, die sich nur bei negativen Zahlen unterscheiden:
  - Bei `>>` wird das Vorzeichenbit vervielfacht, d.h. bei negativen Zahlen (erstes Bit 1) werden 1-Bits hereingeschoben, bei positiven Zahlen (erstes Bit 0) 0-Bits.
  - Bei `>>>` werden immer 0-Bits von links hereingeschoben.
- Bei `int`-Werten werden vom rechten Operanden nur die untersten 5 Bit benutzt (Shift-Distanz von 0 bis 31).

Bedingter Ausdruck (1)

- Der bedingte Ausdruck hat die Form `A ? B : C`.
- Falls `A` wahr (`true`) ist, wird `B` geliefert, sonst `C`.

  Es wird zuerst `A` ausgewertet, und dann, je nach Ergebnis, `B` oder `C`.

  Der andere Teil wird nicht ausgewertet.
- Z.B. kann man das Minimum von `i` und `j` auf folgende Weise berechnen:

```
(i < j)? i : j
```

- In den meisten anderen Programmiersprachen würde man eine `if`-Anweisung benutzen.

  Das kann man in C/C++/Java natürlich auch machen, aber mit dem bedingten Ausdruck sind manchmal kompaktere Formulierungen möglich. Vermutlich spielte für die Einführung von bedingten Ausdrücken in C auch die Verwendung von Macros eine Rolle, dies ist für Java aber nicht mehr relevant.

Bedingter Ausdruck (2)

- Bei `A?B:C` muss `A` den Typ `boolean` oder `Boolean` haben.
- Falls `B` und `C` beide vom gleichen Typ T sind, hat auch der bedingte Ausdruck `A?B:C` diesen Typ.

$$\_?\_:\_ : \texttt{bool} \times T \times T \to T$$

Im Gegensatz zu C++ liefert der bedingte Ausdruck immer einen Wert, keine Variable. Außerdem ist `void` als Ergebnistyp verboten (in Java formal kein Typ).

- Ansonsten gibt es relativ komplizierte Regeln für die Umwandlung in einen gemeinsamen Typ:

$$\_?\_:\_ : \texttt{bool} \times S_1 \times S_2 \to S$$

Im wesentlichen wird der größere der beiden Typen genommen, ggf. wird auch auf einer Seite ein "Unboxing" ausgeführt. Außerdem wird berücksichtigt, dass Datentyp-Literale für ganze Zahlen immer mindestens den Typ `int` haben:
Ist der konstante Ausdruck klein genug, setzt sich ggf. ein kleiner Typ wie `byte` durch. Bei Objekt-Referenzen wird der kleinste gemeinsame Obertyp genommen.

Inhalt

1. Einführung
   - Wertberechnung und Zustandsänderung, Struktur
2. Operatorsyntax
   - Operatorsyntax, Prioritäten, Operatorbaum
   - Arithmetische Operatoren
3. Typen
   - Typ-Korrektheit, Überladene Operatoren, Typ-Anpassung
4. Boolesche Ausdrücke
   - Boolesche Ausdrücke, Vergleichsoperatoren
   - Logische Verknüpfungen (+ Bit-Operatoren)
5. Zuweisungen
   - Zuweisungen, Variable vs. Wert, Typ-Cast, Inkrement
6. Syntax
   - Zusammenfassung/Ausblick (+ Syntaxdiagramme)

Zuweisungen (1)

- Allgemein haben Zuweisungen die Form

  ⟨Variable⟩ = ⟨Ausdruck⟩

  So ist die Zuweisung zunächst ein Ausdruck, und kann auch Teil eines größeren Wertausdrucks sein (s.u.). Wenn man einen Ausdruck als eigenständige Anweisung benutzen will, muss man ihn mit "`;`" abschließen. Deswegen steht im Normalfall nach der Zuweisung noch ein Semikolon. Dadurch wird sie zur Anweisung (engl. "Statement", siehe nächstes Kapitel). Die Variable links kann auch durch einen komplexeren Ausdruck berechnet werden (s.u.).

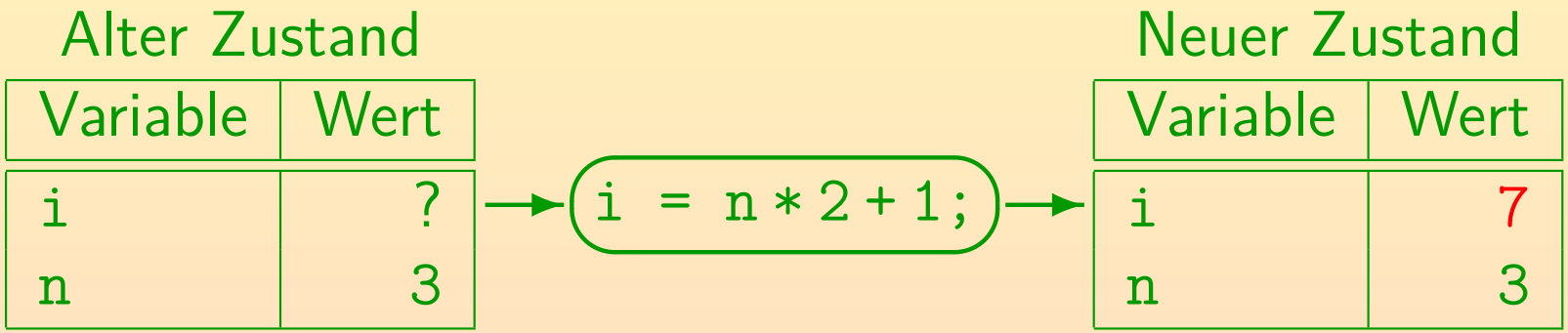
- "`=`" speichert den Wert des rechten Operanden in die Variable auf der linken Seite (⟵), z.B.

  ```
  i = n * 2 + 1;
  ```

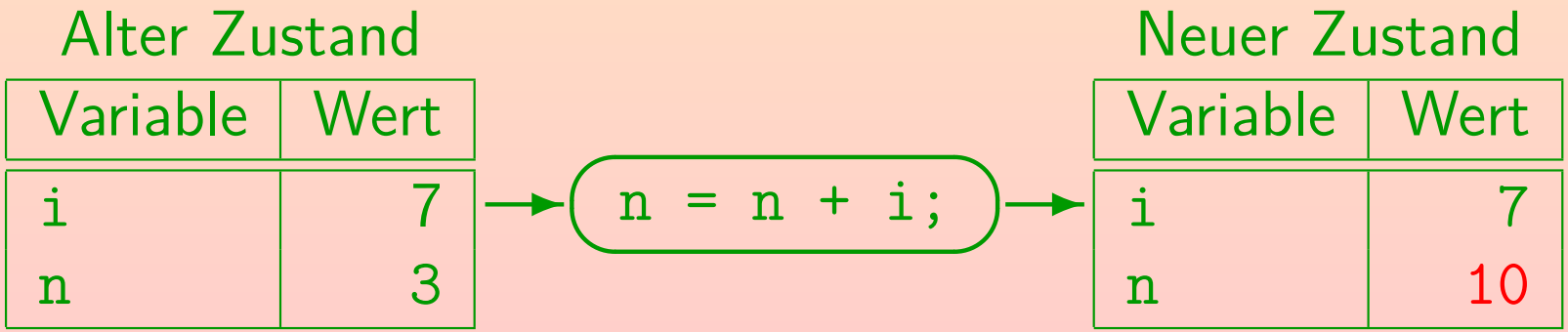
  Wenn `n` den Wert 3 hat, hat `i` anschließend den Wert 7.

Zuweisungen (2)

- Eine Zuweisung ändert den Zustand des Programms, der insbesondere die aktuellen Werte der Variablen enthält.



- Der Ausdruck auf der rechten Seite wird im alten Zustand ausgewertet, anschließend wird der Zustand geändert:



Zuweisungen (3)

```
int i;
int n = 3;
```

Variable	Wert
i	?
n	3

```
i = n * 2 + 1;
```

Variable	Wert
i	7
n	3

```
n = n + i;
```


Variable	Wert
i	7
n	10

```
System.out.println(n*100+i);
```

Ausgabe:	1007

Aufgabe

Was gibt dieses Programm aus?

```
(1)  class ZustandsFolge {
(2)      static public void main(String[] args) {
(3)          int i;
(4)          int j;
(5)
(6)          i = 27;
(7)          j = i - 20;
(8)          i = i % 5;
(9)
(10)         System.out.println(i * j);
(11)     }
(12) }
```

Linke Seite von Zuweisungen (1)

- Auf der linken Seite einer Zuweisung kann nicht nur eine einzelne Variable stehen, sondern auch ein Ausdruck, der zu einer Variablen ausgewertet werden kann:
  - Array-Zugriff:

```
a[i - 1] = 5;
```

    Der Index kann durch einen beliebig komplexen Ausdruck berechnet werden. Wenn `i` hier den Wert 7 hat, wird 5 in `a[6]` gespeichert. Es gibt später noch ein eigenes Kapitel über Arrays.

  - Zugriff auf eine Variable in einem Objekt (Attribut):

```
o.name = "Lisa";
```

    Hier muss `o` ein Objekt von einem Klassentyp `C` sein, der ein Attribut `name` vom Typ `String` hat (das nicht `private` ist, es sei denn, diese Zeile steht in einer Methode der Klasse `C` selbst).

Linke Seite von Zuweisungen (2) *

- Das kann man in langen Ketten kombinieren, auch mit Methoden-Aufrufen:

```
Vorlesung.suche("OOP").belegungen()[i].
          student().kann_programmieren = true;
```

Hier ist angenommen, dass die Klasse `Vorlesung` eine statische Methode `suche` hat, die das Vorlesungs-Objekt zu einem gegebenen Vorlesungs-Titel findet. Dieses Vorlesungs-Objekt hat nun eine Methode `belegungen`, die ein Array von Belegungs-Objekten liefert. Die Klasse `Belegung` hat eine Methode `student`, die den Studenten liefert, der die Vorlesung belegt hat. Die Klasse `Student` hat schließlich ein Attribut `kann_programmieren`, in das mit dieser Anweisung der Wert `true` gespeichert wird. Falls diese Anweisung nicht in der Klasse `Student` steht, darf das Attribut `kann_programmieren` nicht `private` deklariert sein. Auch die aufgerufenen Methoden müssen entsprechend zugreifbar sein. Natürlich ist der lange Ausdruck unübersichtlich, man würde es besser in mehrere Zuweisungen mit Hilfsvariablen aufteilen.

Linke Seite von Zuweisungen (3)

- Um die Typstruktur einer Zuweisung beschreiben zu können, muss folgendes unterscheiden:
  - Variable vom Typ T
  - Wert vom Typ T
- Wir schreiben im folgenden $Var(T)$ für Variablen vom Typ T.

  Man kann $Var(T)$ als eigenen Typ verstehen. $Var(T)$ ist aber kein Java-Code. Statt $Var(T)$ wird auch "Lvalue vom Typ T" gesagt: "L", weil es auf der linken Seite der Zuweisung stehen darf, also eine Hauptspeicheradresse hat.
- Die Signatur des Zuweisungsoperators = ist (vereinfacht):

$$Var(T) \times T \to T$$

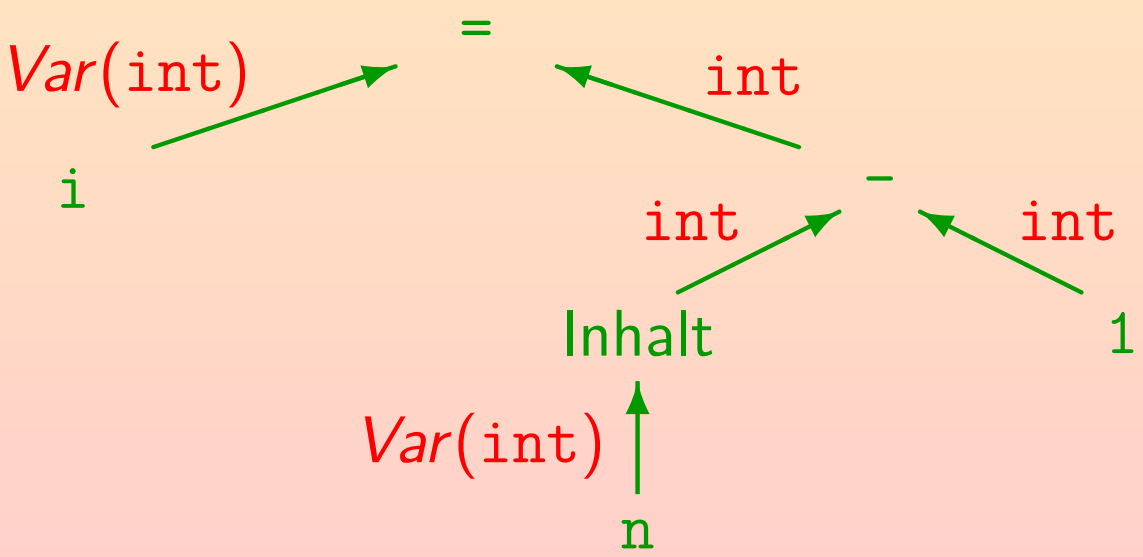
  wobei T ein beliebiger Typ ist.

  Außer den $Var(T)$-Typen und `void` (das zählt formal nicht als Typ).

  Eine Zuweisung liefert den zugewiesen Wert.

Linke Seite von Zuweisungen (4)

- Es gibt eine automatische Umwandlung von Variablen zu Werten: Wenn ein Wert vom Typ T benötigt wird, und man hat eine Variable vom Typ T, nimmt der Compiler automatisch den in der Variablen gespeicherten Wert.
- Die Zuweisung "`i = n - 1`" ist so zu verstehen:



Linke Seite von Zuweisungen (5)

- Die implizite Operation "Inhalt" (zum Übergang von *Var*(*T*) nach *T*) ist so selbstverständlich und häufig, dass sie meist nicht in den Auswertungsbäumen gezeigt wird.
- Folgende Zuweisung ist ein Fehler:

  ```
  (a+1) = 5
  ```

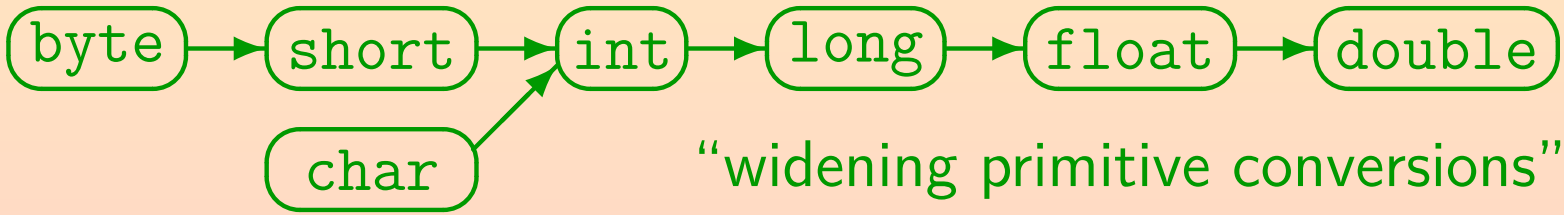
  Der Ausdruck `a+1` auf der linken Seite liefert einen Wert vom Typ `int`, und nicht eine Variable vom Typ *Var*(`int`).

  Man könnte auf die Idee kommen, dass Java in diesem Fall 4 in die Variable a speichern sollten, aber das geht nur in Computeralgebra-Systemen.
- Auch Methodenaufrufe liefern nur Werte (oder `void`, d.h. nichts), aber keine Variablen.

  Sie können aber ein Objekt oder Array liefern, und dann erhält man mit "`.Attribut`" bzw. "`[i]`" eine Variable.

Zuweisungen: Typanpassungen (1)

- Der Typ des Wertes auf der rechten Seite der Zuweisung muss nicht genau mit dem Typ der Variablen links übereinstimmen, sondern es können verschiedene Typanpassungen erfolgen.
- Zum Beispiel kann man einen Wert eines kleineren primitiven Typs an eine Variable eines größeren Typs zuweisen:



Man kann einen Wert eines Typs weiter links an eine Variable mit Typ weiter rechts zuweisen, z.B. `char` an `float`. Dies sind die gleichen Umwandlungen, wie sie auch zur Vereinheitlichung der Typen vor einer arithmetischen Operation verwendet werden (siehe Folie 33). Dort wurde allerdings immer mindestens nach `int` konvertiert.

Zuweisungen: Typanpassungen (2)

- Zuweisungen von einem Wert eines größeren Typs an eine Variable eines kleineren Typs sind normalerweise verboten.

  Der Wert könnte dabei zerstört werden. Wenn man es wirklich will, kann man einen Typ-Cast verwenden (siehe nächste Folie).

- Ausnahme: Auf der rechten Seite steht ein konstanter Ausdruck (ohne Variablen und Methoden-Aufrufe).
  - In diesem Fall kennt der Compiler schon den genauen Wert.

    Der Ausdruck wird zur Compile-Zeit ausgewertet, nicht erst zur Laufzeit (d.h. nicht erst bei Ausführung des Programms).
  - Er prüft, ob der Wert klein genug für den Variablen-Typ ist.
  - Falls ja, ist die Zuweisung von dem formal größeren primitiven Typ an den kleineren erlaubt.

    Das gilt bei Zuweisungen an `byte`, `char`, `short`, sowie `Byte`, `Character`, `Short`, wenn rechts ein Ausdruck vom Typ `byte`, `char`, `short`, `int` steht.

Zuweisungen: Typanpassungen (3) *

- Bei der Zuweisung sind noch weitere Typ-Anpassungen möglich, die aber alle Konstrukte verwenden, die in der Vorlesung erst später behandelt werden:
  - Man kann einen Wert eines Untertyps an eine Variable eines Obertyps zuweisen, insbesondere also ein Objekt einer Unterklasse an eine Variable einer Oberklasse.

    Unterklassen entsprechen Spezialfällen oder Teilmengen.

    Z.B. könnte `Student` eine Unterklasse von `Person` sein.

    Alle Referenztypen (inklusive Arrays) sind Untertypen von `Object`.
  - Bei der Zuweisung kann "Unboxing" stattfinden.

    D.h. ein primitiver Wert kann aus einem Objekt der entsprechenden Klasse "ausgepackt" werden, z.B. kann man auf der rechten Seite der Zuweisung ein `Integer`-Objekt haben, und links eine `int`-Variable.

    Nach dem Auspacken können noch Typ-Vergrößerungen stattfinden.
  - Entsprechend ist umgekehrt ein "Boxing" möglich.

Explizite Typumwandlung: Cast (1)

- Man kann eine Typumwandlung explizit verlangen mit einem Ausdruck der Form

  `(⟨Typ⟩) ⟨Ausdruck⟩`

- Wenn z.B. `i` eine Variable vom Typ `int` ist, kann man ihren Inhalt auf eigene Gefahr in eine `byte`-Variable speichern:

```
byte b = (byte) i;
```

- Dabei werden die höherwertigen Bits einfach abgeschnitten: Wenn der Wert zu groß für ein Byte war, wird er zerstört.

  Dabei kann z.B. auch aus einem positiven Wert ein negativer werden, z.B. wird 255 zu -1. Der Programmierer sollte sicher sein, dass das nicht passieren wird, d.h. dass die Werte immer klein genug sind.

- Dies ist als Cast oder Type-Cast bekannt.

  Deutsch u.a. "Gipsverband": Es ist nicht schön und nur ein letztes Mittel.

Explizite Typumwandlung: Cast (2)

- Bei einem Cast von einer Gleitkommazahl in einen ganzzahligen Typ werden die Nachkommastellen abgeschnitten.

  Die Details sind recht kompliziert: `NaN` (not-a-number, Fehlerwert) wird auf `0` abgebildet. Zu große Werte werden auf den größtmöglichen Wert von `int` bzw. `long` abgebildet, bei Typen kleiner als `int` werden anschließend vorne Bits gestrichen. Entsprechend bei zu kleinen Werten.

- Während Java sich bei primitiven Typen auf den Programmierer verläßt, findet bei der Typumwandlung von Referenztypen ggf. ein Test zur Laufzeit statt.

  Wenn man ein Objekt einer `Person`-Variable (Oberklasse) mit einem Cast an eine `Student`-Variable (Unterklasse) zuweisen will, prüft Java bei der Programm-Ausführung, dass es sich wirklich um ein `Student`-Objekt handelt (sonst Fehler `ClassCastException`). Falls schon zur Compilezeit feststeht, dass die Typumwandlung sicher scheitern wird, meldet der Compiler den Fehler.

Zuweisungen in Ausdrücken

- Weil eine Zuweisung selbst ein Ausdruck ist, kann man sie in größere Ausdrücke einbauen (Stilfrage).

  Der Wert einer Zuweisung ist der Wert der Variablen nach der Zuweisung (d.h. der zugewiesene Wert, aber nach eventuellen Typanpassungen).

- Z.B. kann man eine Zahl einlesen und sofort vergleichen:

```
while((n = input.nextInt()) > 0) { ... }
```

- Die Klammern um die Zuweisung sind nötig, da der Zuweisungs-Operator eine sehr niedrige Priorität hat.

- Man kann z.B. folgendermaßen zwei Variablen auf 0 setzen:

```
i = j = 0;
```

  Es ist eine Stilfrage, ob man solche Mehrfachzuweisungen tatsächlich nutzt. Eventuell sind einzelne Zuweisungen klarer.

Seiteneffekte

- Normalerweise ist der offizielle Hauptzweck eines Wertausdrucks die Berechnung eines Wertes.
- Bewirkt er darüber hinaus eine Zustandsänderung (durch eine Zuweisung oder Ein-/Ausgabe), so sagt man, er habe einen Seiteneffekt.

  Von einer expliziten Zuweisung, die nicht Teil eines größeren Ausdrucks ist, würden die meisten Leute wohl nicht sagen, dass sie einen Seiteneffekt hat. Hier ist die Zustandsänderung ja der Hauptzweck. Die Sprechweise stammt aus Sprachen, bei denen Zuweisungen nicht selbst ein Wertausdruck sind.
- Es ist stilistisch fragwürdig, wenn ein Ausdruck nach einer Zustandsänderung noch weitere Berechnungen enthält.

  Die Zustandänderung könnte durch eine eingebettete Zuweisung, einen Inkrement-/Dekrement-Operator (s.u.) oder einen Methoden-Aufruf mit Zustandsänderung erfolgen.

Abkürzung: Zuweisung *

- Zuweisungen der Form

  $$\texttt{X = X } \theta \texttt{ Y}$$

  mit $\theta \in \{+, -, *, /, \%, <<, >>, >>>, \&, |, \hat{}\}$ kommen häufig vor.
- In C/C++/Java gibt es dafür die Abkürzung

  $$\texttt{X } \theta\texttt{= Y}$$

- Z.B. kann man `X += 2` statt

  $$\texttt{X = X + 2}$$

  schreiben.

  Eine Feinheit ist, dass `X` bei der Abkürzung nur einmal ausgewertet wird.
  Das ist wichtig, falls `X` selbst Zuweisungen/Seiteneffekte enthält.

Inkrement/Dekrement (1)

- Die Erhöhung einer Variablen um 1 (“inkrementieren”) ist besonders häufig, daher kann `i = i + 1` abgekürzt werden zu `i++` oder `++i`. Unterschied:
  - `i++` liefert den alten Wert der Variablen `i` und addiert dann 1 auf (“postincrement”).

    Daher entspricht `i++` nicht genau `i=i+1`. Dies liefert den neuen Wert.
  - `++i` addiert erst 1 auf, und liefert dann den neuen Wert der Variablen (“preincrement”).
- Hat z.B. `i` vorher den Wert `3`, so würde
  - `i++` den Wert `3` liefern, aber
  - `++i` den Wert `4`.

  In beiden Fällen hätte die Variable hinterher den Wert `4`.

Inkrement/Dekrement (2)

- Falls man den Wert der Variablen nicht braucht, sondern sie nur erhöhen (inkrementieren) will, ist es egal, ob man `i++;` oder `++i;` schreibt.

  Die erste Variante sieht man häufiger.

- Es ist aber z.B. auch Folgendes möglich:

  ```
  eingaben[i++] = n;
  ```

  Hier wird der Wert `n` in das Array `eingaben` gespeichert, und der Index `i` für die nächste freie Position gleich erhöht. Wenn `i` am Anfang auf `0` initialisiert ist, ist diese Variante richtig, und nicht `eingaben[++i]`.

- Entsprechend kann man mit `i--` und `--i` den Wert von `i` um `1` verringern (dekrementieren).

- Im ersten Semester müssen Sie diese Operatoren nicht verwenden, `i = i + 1;` geht auch.

Auswertungsreihenfolge (1)

- Ein wesentlicher Unterschied von Java zu C++ ist, dass
  - bei Java die Auswertung eines Ausdrucks strikt von links nach rechts erfolgt,
  - bei C++ der Compiler bei fast allen Operatoren die Auswertungsreihenfolge wählen darf.

    D.h. ob linker oder rechter Operand zuerst. Klammern bestimmen nur die Baumstruktur des Ausdrucks, nicht die Auswertungsreihenfolge.
- In C++ hat der Compiler so mehr Optimierungsmöglichkeiten.

  Wenn er z.B. einen Variablenwert schon in einem Register der CPU stehen hat, kann er damit gleich weiterrechnen.
- In Java stand die Portabilität im Vordergrund: Man wollte immer ein definiertes Ergebnis.

  Ein anderer Compiler soll nicht das Verhalten des Programms ändern.

Auswertungsreihenfolge (2) *

- Beispiel: Angenommen, `i` hat den Wert 3 und man führt folgende Zuweisung aus:

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

- Bei C++ kann je nach Compiler `j` anschließend den Wert 6 oder 7 haben (Ergebnis ist nicht definiert).

  7 ergibt sich, wenn die rechte Hälfte zuerst ausgewertet wird.

- In Java hat `j` anschließend sicher den Wert 6.

  Wenn man die beiden Summanden vertauscht, also das Inkrementieren links macht, ist das Ergebnis 7.

- Es ist aber schlechter Stil, wenn die Auswertungsreihenfolge eine Rolle spielt: Man sollte solchen kryptischen Code vermeiden.

Aufgabe

Was gibt dieses Programm aus?

```
(1)  class Aufgabe {
(2)      static public void main(String[] args) {
(3)          int n;
(4)          boolean b;
(5)
(6)          n = 5 / 2 + 4 * 5 % 2;
(7)          n--;
(8)          b = (n % 3 == 0);
(9)          b = b && n < 50;
(10)         if(b)
(11)             n = n + 100;
(12)
(13)         System.out.println(++n);
(14)     }
(15) }
```

Inhalt

1. Einführung
   - Wertberechnung und Zustandsänderung, Struktur
2. Operatorsyntax
   - Operatorsyntax, Prioritäten, Operatorbaum
   - Arithmetische Operatoren
3. Typen
   - Typ-Korrektheit, Überladene Operatoren, Typ-Anpassung
4. Boolesche Ausdrücke
   - Boolesche Ausdrücke, Vergleichsoperatoren
   - Logische Verknüpfungen (+ Bit-Operatoren)
5. Zuweisungen
   - Zuweisungen, Variable vs. Wert, Typ-Cast, Inkrement
6. Syntax
   - Zusammenfassung/Ausblick (+ Syntaxdiagramme)

Zusammenfassung/Ausblick (1)

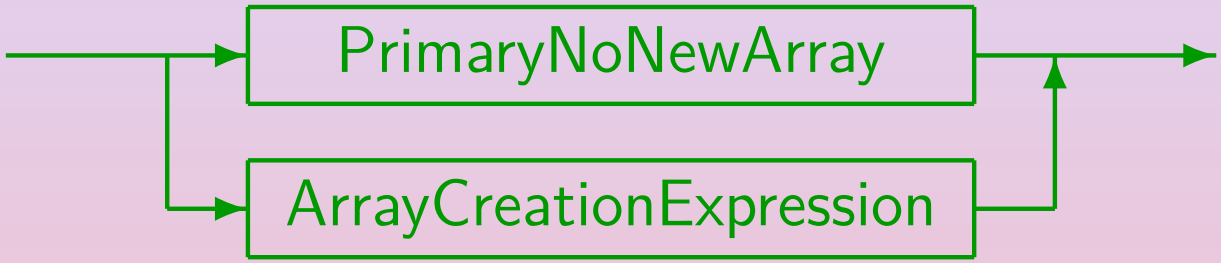
- Ausdrücke (Expressions) in Java:
  - Datentyp-Literale, z.B. `12`
  - Variablen, z.B. `i`
  - Zugriff auf Attribute, z.B. `obj.attr`
  - Zugriff auf statische Attribute, z.B. `Class.attr`
  - Zugriff auf Arrays, z.B. `a[i]`
  - Aufruf einer statischen Methode, z.B. `Class.meth(...)`
  - Aufruf einer normalen Methode, z.B. `obj.meth(...)`
  - Aktuelles Objekt: `this`
  - Zugriff auf Attribute/Methoden der Oberklasse: `super.attr`
  - Geklammerter Ausdruck, z.B. `(i+1)`

Zusammenfassung/Ausblick (2)

- Ausdrücke (Expressions) in Java, Forts.:
  - Erzeugung eines neuen Objektes, z.B. `new Class(...)`
  - Erzeugung eines neuen Arrays, z.B. `new Class[i]`
  - Arithmetischer Ausdruck, z.B. `i*2`
  - String-Konkatenation, z.B. `"Preis: " + preis`
  - Vergleich, z.B. `i == 0`
  - Logische Verknüpfung, z.B. `i >= 0 && i <= 100`
  - Zuweisung, z.B. `i = 0`
  - Abkürzung für Wertänderungen einer Variablen, z.B. `i += 2`
  - Post/Pre-Increment/Decrement, z.B. `i++`
  - Bedingter Ausdruck, z.B. `(i > j)? i :j`
  - Typ-Umwandlung (Cast), z.B. `(byte) i`

Primary Expressions: Syntax (1)

- Primary (Primary Expression, Prioritätsstufe 1):



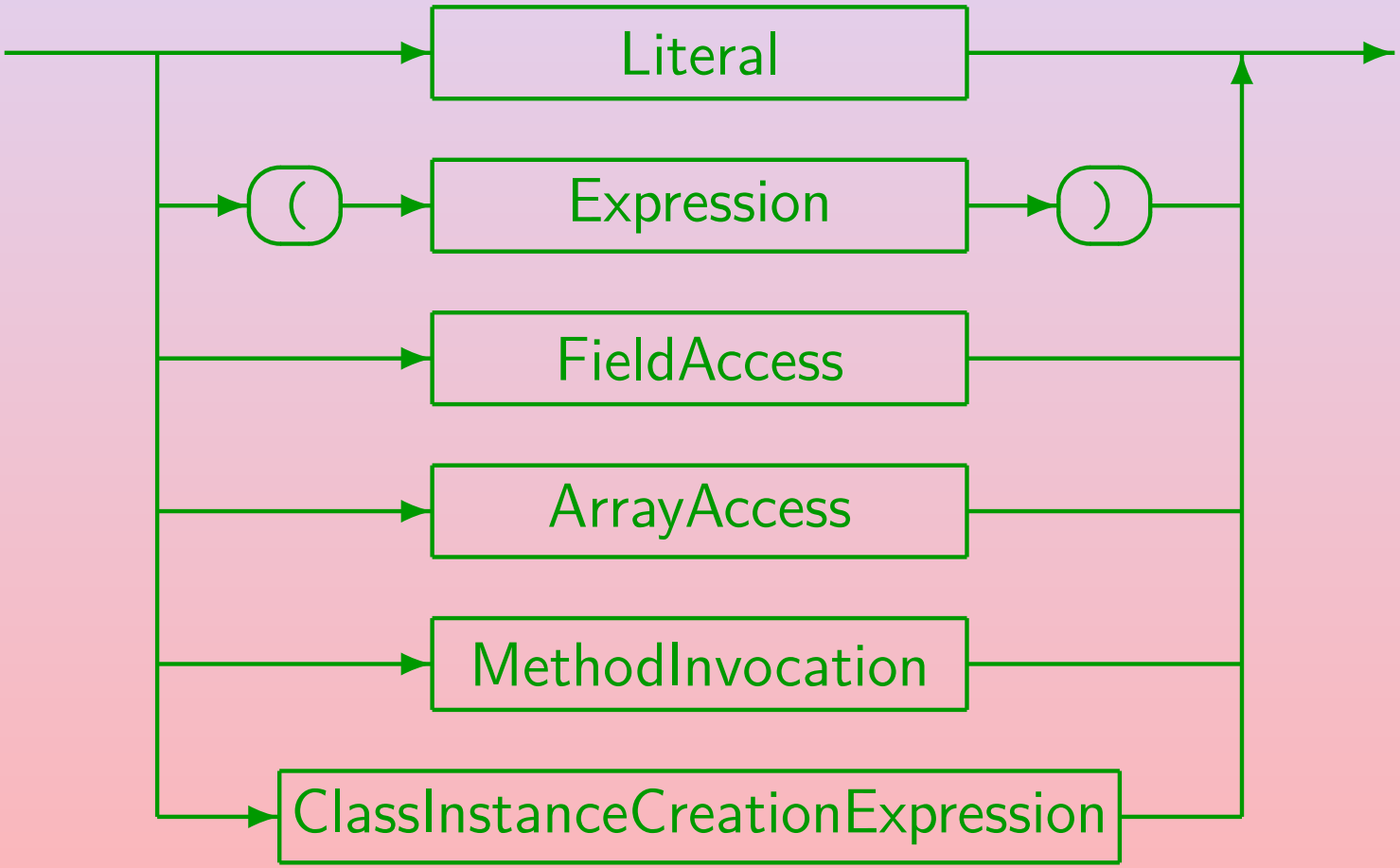
"Primary Expressions" sind die Basis, gewissermaßen die einfachsten Expressions, aus denen dann mit Operatoren größere aufgebaut werden.

Der Grund, warum "ArrayCreationExpression", also z.B. `new int[5]` getrennt behandelt werden muss, liegt in einer Mehrdeutigkeit bei mehrdimensionalen Arrays (werden erst später behandelt):
"`new int[5][3]`" soll eine Matrix erzeugen, und nicht etwa ein Zugriff auf ein gerade erzeugtes Array sein. Wenn man das will, muss man Klammern setzen: "`(new int[5])[3]`".

Die normalen lokalen Variablen kommen erst eine Prioritätsstufe später bei den "Postfix Expressions" (wegen mehrteiligen Bezeichnern mit "`.`").

Primary Expressions: Syntax (2)

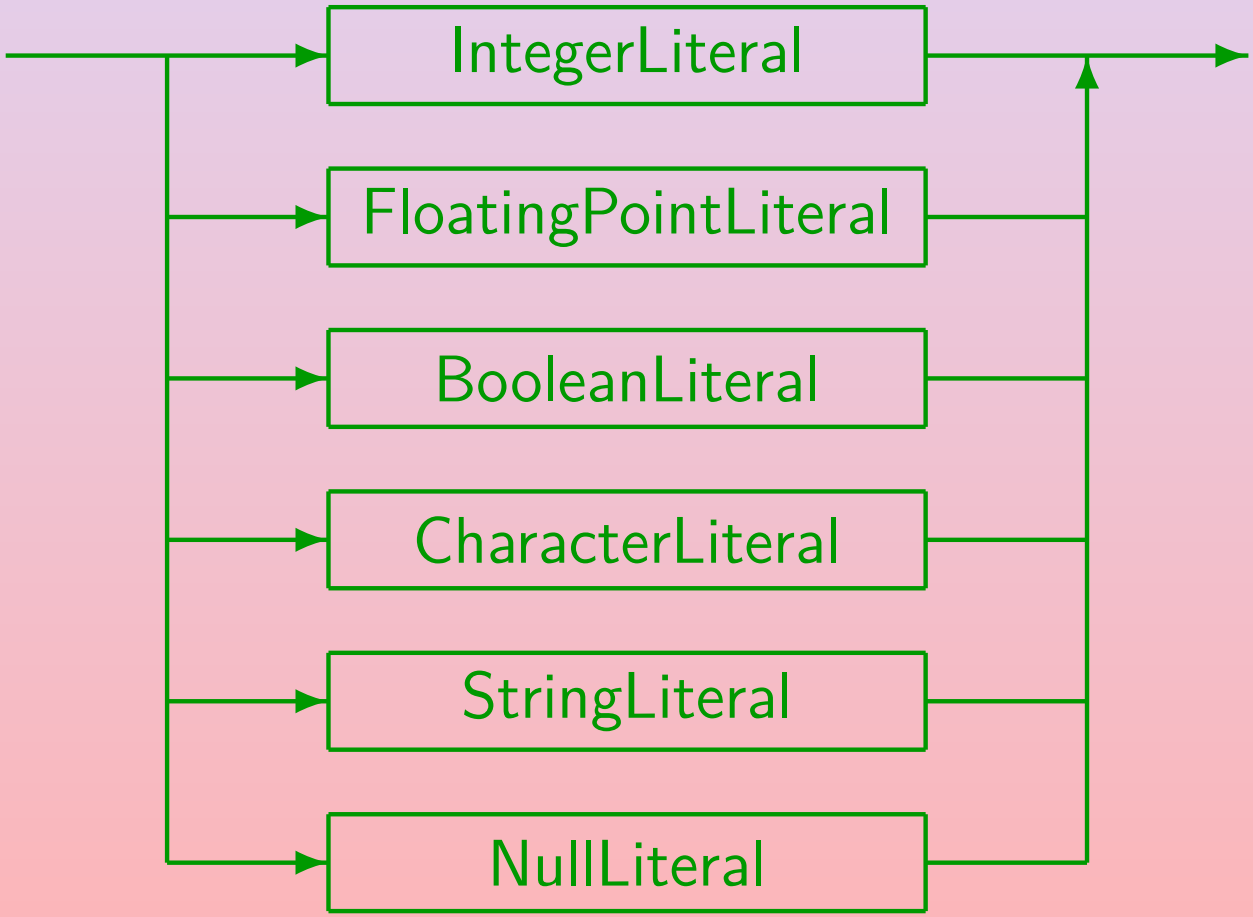
- PrimaryNoNewArray:



Diese Definition ist etwas vereinfacht. Wir hatten z.B. `this` noch nicht.

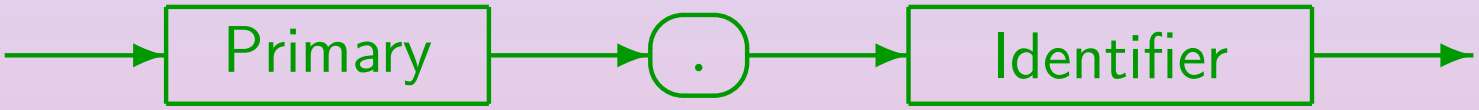
Primary Expressions: Syntax (3)

- Literal:



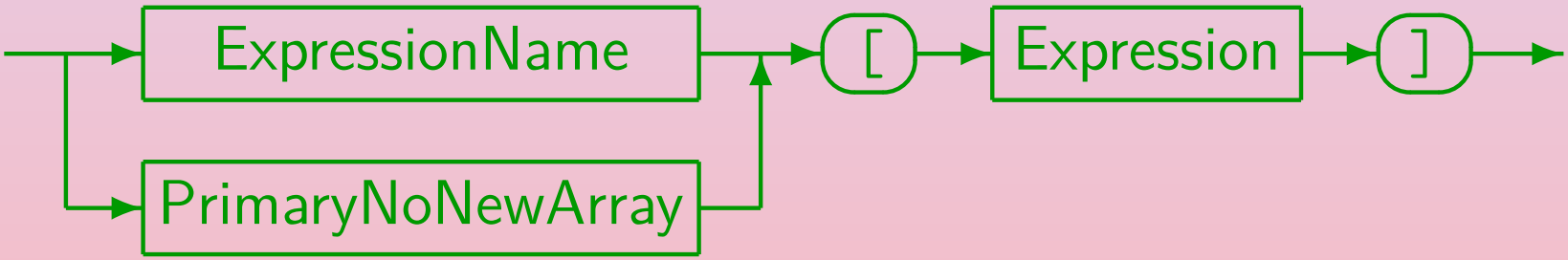
Primary Expressions: Syntax (4)

- FieldAccess:

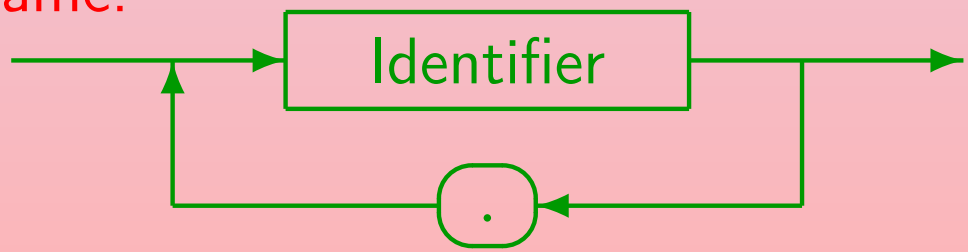


Das ist etwas vereinfacht, später kommt "`super.`".

- ArrayAccess:



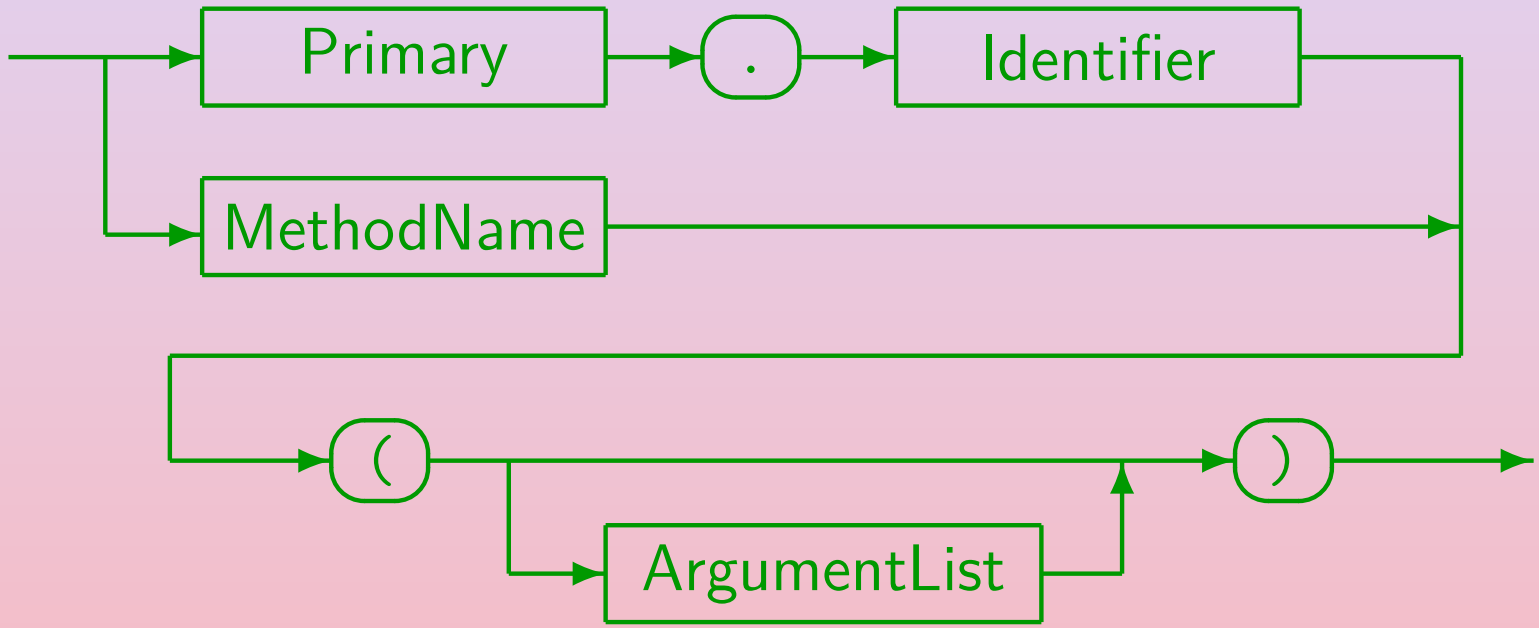
- ExpressionName:



Z.B. "`Paket.Klasse.StatischeVariable`".

Primary Expressions: Syntax (5)

- MethodInvocation:



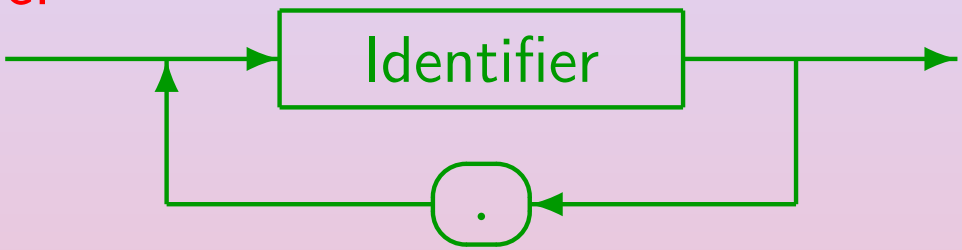
Das ist wieder etwas vereinfacht, später kommt u.a. "`super.`".

Da das Objekt, für das die Methode aufgerufen wird, selbst mit einer "Primary Expression" berechnet wird, kann es z.B. Resultat eines Methodenaufrufs oder eines Zugriffs auf ein Array sein.

Die Variante mit "MethodName" deckt u.a. statische Methoden ab.

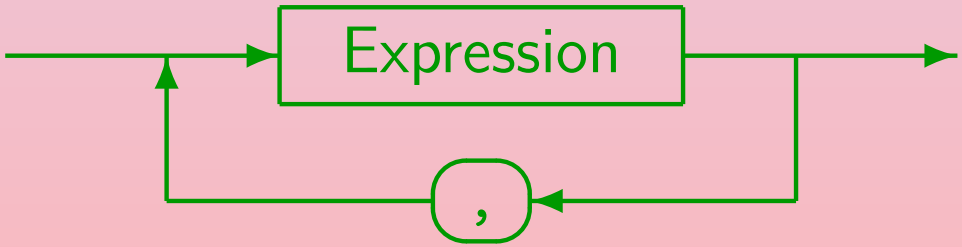
Primary Expressions: Syntax (6)

- MethodName:



Z.B. "`Paket.Klasse.StatischeMethode`".

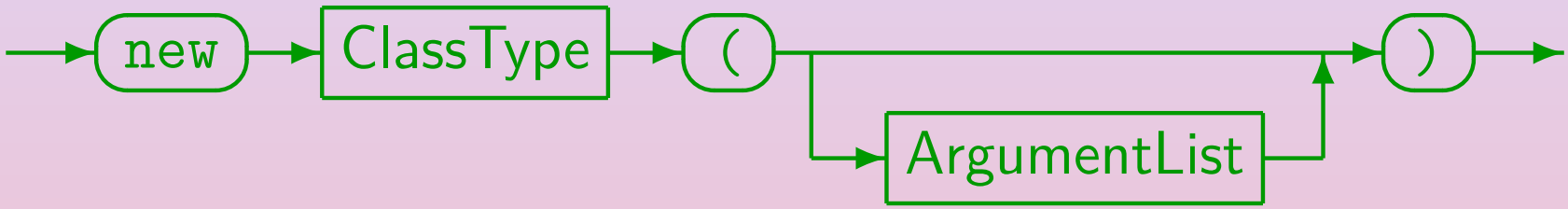
- ArgumentList:



Die Argumente werden durch Kommata "," getrennt. Jedes Argument kann mit einer beliebig komplizierten Expression berechnet werden.

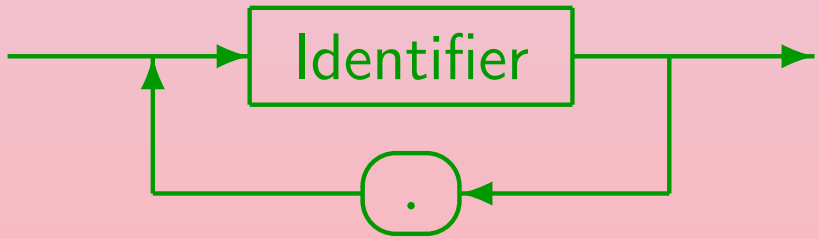
Primary Expressions: Syntax (7)

- ClassInstanceCreationExpression:



Dies ist deutlich vereinfacht. Es gibt z.B. noch Typ-Argumente. Später mehr.

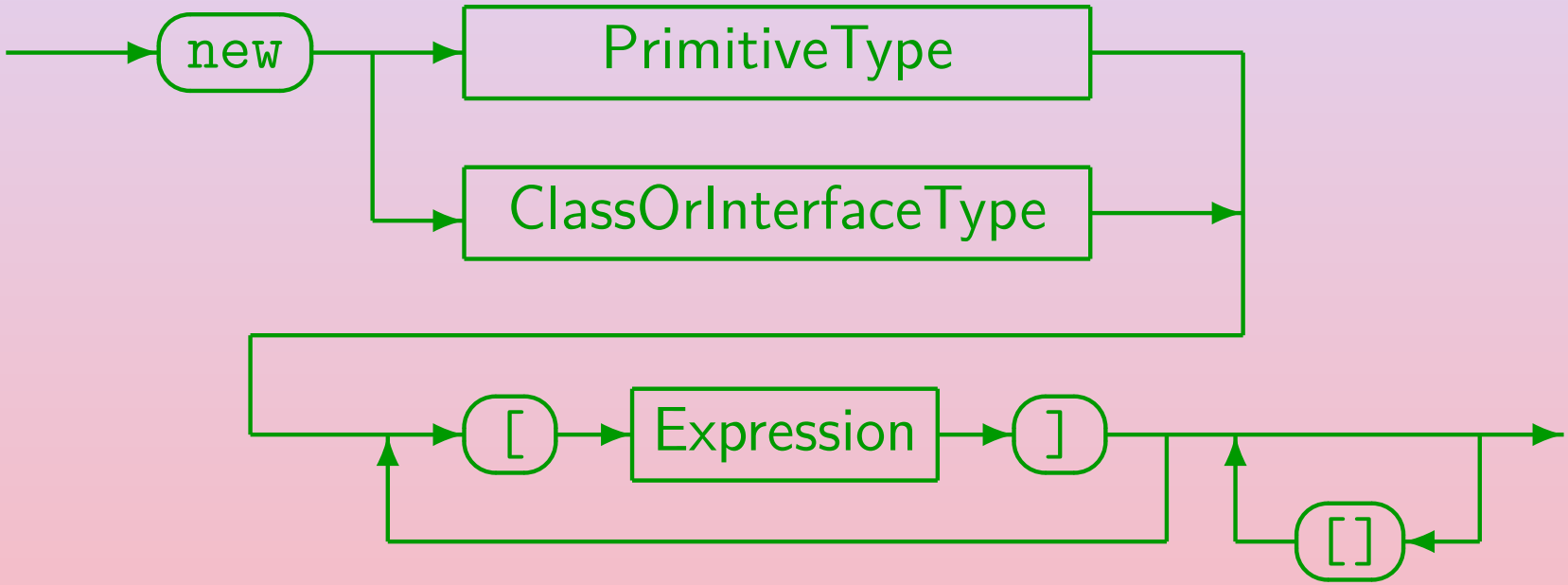
- ClassType, ClassOrInterfaceType:



Z.B. "`java.util.Scanner`". Dem Klassennamen kann ein Paketname vorangestellt werden. Dieser kann wieder aus mehreren Teilen bestehen.

Primary Expressions: Syntax (8)

- ArrayCreationExpression:



Die Möglichkeit, mehrere Klammerpaare (am Ende auch leere Klammerpaare) zu verwenden, hängt wieder mit mehrdimensionalen Arrays zusammen, die in dieser Vorlesung erst später besprochen werden.
Außerdem wurden initialisierte Arrays weggelassen.

Prioritätsstufen (von hoch nach niedrig)

1	`o.a, i++, a[], f(), ...`	Postfix-Operatoren
2	`-x, !, ~, ++i, ...`	Präfix-Operatoren
3	`new C(), (type) x`	Objekt-Erzeugung, Cast
4	`*, /, %`	Multiplikation etc.
5	`+, -`	Addition, Subtraktion
6	`<<, >>, >>>`	Shift
7	`<, <=, >, >=, instanceof`	kleiner etc.
8	`==,!=`	gleich, verschieden
9	`&`	Bit-und, logisches und
10	`^`	Bit-xor, logisches xor
11	`|`	Bit-oder, logisches oder
12	`&&`	logisches und (bedingt)
13	`||`	logisches oder (bedingt)
14	`?:`	Bedingter Ausdruck
15	`=, +=, -=, *=, /=, ...`	Zuweisungen