Objektorientierte Programmierung

Kapitel 4: Lexikalische Syntax

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2018/19

http://www.informatik.uni-halle.de/~brass/oop18/

Inhalt

1. Compiler
2. Unicode
3. Leerplatz, Kommentare
4. Bezeichner
5. Datentyp-Konstanten
6. Operatoren

Aufgabe eines Compilers

- Übersetzung von Programmen aus einer Programmiersprache A in eine Programmiersprache B.
  - Eine Maschine für B existiert (Hardware, Interpreter, Compiler in implementierte Sprache C).
  - A ist für den Programmierer bequemer als B.

    Zumindest für bestimmte Aufgaben. Je nach Problem können unterschiedliche Sprachen besonders geeignet sein.
- Der Computer versteht die Sprache A über den Compiler.

  Der Compiler $A \rightarrow B$ macht aus einer Maschine für B eine Maschine für A.

Phasen eines Compilers (1)

- Compiler sind große Programme. Üblich: Einteilung in möglichst unabhängige Module/Arbeitsphasen:
  - Lexikalische Analyse (Scanner)
  - Syntaktische Analyse (Parser)
  - Semantische Analyse (u.a. Typprüfung)
  - Erzeugung von Zwischencode
  - Codeoptimierung
  - Codeerzeugung

    Besonders praktisch ist hier, dass man die von einem Modul zum nächsten übergebenen Daten relativ leicht ausgeben kann.

Phasen eines Compilers (2)

- Trennung lexikalische Syntax (Scanner) vs. kontextfreie Syntax (Parser):
  - Der Scanner kondensiert mit einem einfachem (sehr effizienten) Verfahren die Eingabe (Zeichenfolge) in eine kürzere Folge von Wortsymbolen.
  - Das komplexere Analyseverfahren im Parser muss dann nur noch eine kürzere Eingabe verarbeiten.
  - Einfache Regel für Leerplatz: Kann zwischen je zwei lexikalischen Einheiten (Wortsymbolen) eingestreut werden.
- Die mehrphasige Übersetzung erklärt auch, warum manchmal Fehler weiter unten im Programm zuerst gemeldet werden.

  Z.B. wird beim aktuellen Java-Compiler ein Syntaxfehler gemeldet, und nachdem man ihn korrigiert hat, ein Typfehler (semantische Analyse) weiter vorn.

Lexikalische Analyse (1)

- Eingabe: Programm als Folge von Zeichen.
- Ausgabe: Folge von Token (Wortsymbole).
- Leerplatz ("white space") und Kommentare werden entfernt.

  Leerplatz: Leerzeichen, Zeilenumbrüche, etc. Siehe unten.

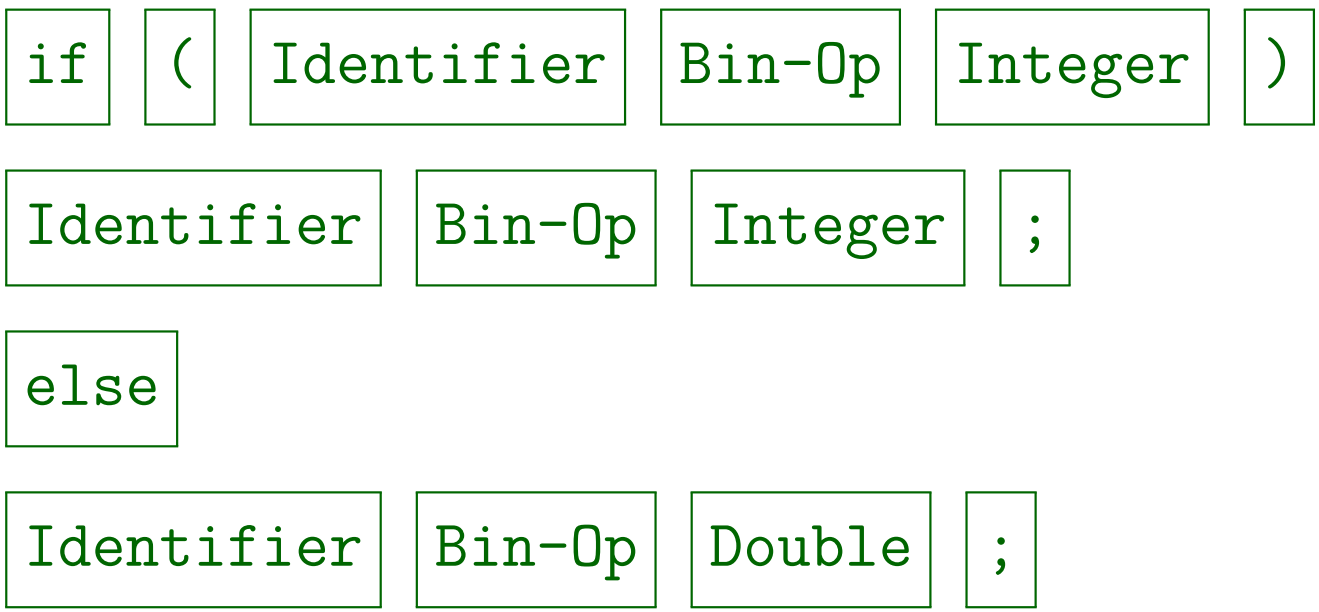
- Zwischen manchen Token ist Leerplatz
  - nötig, um sie zu trennen (z.B. `else x`),
  - zwischen manchen ist er optional (z.B. `x=1`).

Lexikalische Analyse (2)

- Eingabe:

```
if(total_amount >= 100)
        // Keine Versandkosten
        shipping = 0;
else
        shipping = 5.95;
```

- Ausgabe:



Lexikalische Analyse (3)

- Manche Token haben außer ihrem Typ noch einen Wert, den der Scanner an den Parser übergibt:
  - Datentyp-Konstanten / Literale (z.B. `Integer`) haben einen Wert (z.B. 100).

    Eventuell führt der Scanner auch schon die Umwandlung von einer Ziffernfolge in die interne binäre Repräsentation durch.
  - Bezeichner (`Identifier`) haben einen Namen.

    Bezeichner werden in eine Symboltabelle eingetragen. Dort können weitere Daten hinterlegt werden, wie z.B. der Typ der Variablen.
  - Gibt es nur einen gemeinsamen Tokentyp für alle Operatoren, so muß der genaue Operator durch einen zusätzlichen Wert identifiziert werden.

Inhalt

1. Compiler
2. Unicode
3. Leerplatz, Kommentare
4. Bezeichner
5. Datentyp-Konstanten
6. Operatoren

Unicode (1)

- Java basiert auf dem Unicode-Zeichensatz.

  Ziel des Unicode-Zeichensatzes ist es, alle Schriftzeichen in gedruckten Dokumenten zu umfassen, also z.B. auch chinesische, kyrillische und arabische [http://www.unicode.org/]. Ursprünglich sollten 16-Bit ausreichen (65 536 Zeichen), aber inzwischen gibt es 17 "Ebenen" zu je 16 Bit. Java verwendet UTF-16, eine 16-Bit Codierung von Unicode. Variablen des Typs `char` können auch nur 16 Bit speichern, d.h. in (seltenen) Ausnahmefällen nur einen Teil eines Zeichencodes, der aus zwei 16 Bit Einheiten besteht.

- Während ein Java-Programm intern (im Compiler) als Folge von Unicode-Zeichen behandelt wird, kann es in der Text-Datei auch mit anderer Codierung stehen.

  Auch Unicode kann unterschiedlich codiert werden, z.B. wird mit UTF-8 nur ein Byte für die klassischen ASCII-Zeichen benötigt, mit UTF-16 dagegen zwei Byte. Unicode definiert nur, wie das Zeichen zu einem "Code Point" (Nummer) aussieht. Die Codierung in Bits/Bytes ist eine andere Frage.

Unicode (2)

- Man kann dem Compiler sagen, welche Codierung er für die Eingabedatei annehmen soll, z.B.
  - `javac -encoding ISO8859-1 Hello.java`
  - `javac -encoding utf8 Hello.java`
  - `javac -encoding ASCII Hello.java`
- Zum Teil ist eine automatische Erkennung anhand des "Byte Order Marks" am Anfang der Datei möglich.

  Für UTF-16 ist es Standard, dass das "Byte Order Mark" (16-Bit Zahl `U+FEFF`) an den Anfang der Datei geschrieben wird. Bei UTF-8 kann man diesen Wert auch einfügen (codiert als drei Bytes `EF`, `BB`, `BF`) aber nicht jeder Editor macht das. Die Datei wäre dann nicht mehr kompatibel mit Programmen, die ASCII erwarten. Deutsche Umlaute werden in ISO Latin-1 (8859-1) und UTF-8 unterschiedlich codiert (1 Byte vs. 2 Byte).

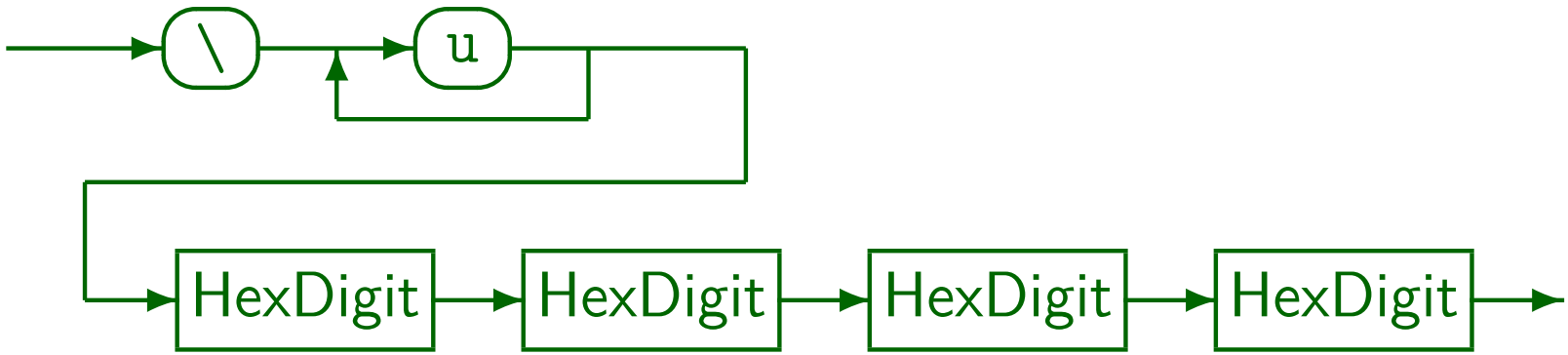
Unicode (3)

- Java erlaubt, beliebige Unicode-Zeichen eingeben zu können, selbst wenn man eine beschränkte Codierung verwendet (z.B. ASCII):
  - Die Zeichenfolge `\uXXXX` im Java-Programm wird durch das Unicode-Zeichen mit Code `XXXX` ersetzt.

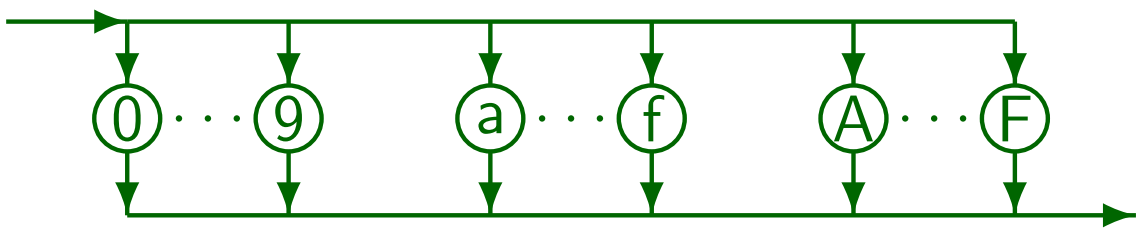
    Den Code muss man dazu hexadezimal aufschreiben, d.h. zur Basis 16 anstatt zur üblichen Basis 10. Man braucht dazu 16 Ziffern, nämlich die üblichen Ziffern 0–9 sowie `A` (Wert 10), `B` (11), `C` (12), `D` (13), `E` (14), `F` (15). Jede Ziffer entspricht 4 Bit ($16 = 2^4$). Aber: Keine Panik! Das braucht man sehr selten (wenn man z.B. deutsche Umlaute in Strings haben will, aber die Datei unbedingt in reinem ASCII codiert sein muss — sonst könnte man ja einfach die Umlaute tippen).
  - Z.B.: `\u00C4` ist Ä, `\u00E4`: ä, `\u00D6`: Ö, `\u00F6`: ö, `\u00DC`: Ü, `\u00FC`: ü, `\u00DF`: ß, `\u00A7` ist §, `\u20AC`: €.

Unicode (4)

- UnicodeEscape:



- HexDigit:



Unicode (5)

- Die Ersetzung der "Unicode Escapes" `\uXXXX` geschieht vor aller anderen Verarbeitung durch den Compiler.

  Wenn man sehr kryptische Programme schreiben will, kann man auch normale ASCII-Zeichen so codieren, auch Zeichen, die eine besondere Bedeutung in Java haben (nicht nur Zeichen in Strings). Ein codiertes "\" nimmt allerdings nicht an weiterer Ersetzung dieser Codierung teil.

- In `\uXXXX` kann man statt einem "`u`" beliebig viele schreiben.

  Dadurch kann man eine beliebige Datei reversibel in ASCII überführen: Alle Nicht-ASCII Zeichen werden durch `\uXXXX` codiert (wobei für `XXXX` natürlich der passende Code eingesetzt wird). Falls die Eingabe schon Codes dieser Art enthält, wird ein "`u`" mehr eingefügt.

Inhalt

1. Compiler
2. Unicode
3. Leerplatz, Kommentare
4. Bezeichner
5. Datentyp-Konstanten
6. Operatoren

Leerplatz, Kommentare (1)

- Zwischen zwei Token ist eine beliebige Folge von Leerzeichen, Tabulatorzeichen, Zeilenumbrüchen, Formfeed (neue Seite) sowie Kommentaren erlaubt.

  Zeilenumbrüche, Einrückungen, und Kommentare können verwendet werden, um Programme lesbarer zu gestalten. Der Compiler ignoriert diese Dinge.

- Für den Zeilenumbruch akzeptiert Java drei Alternativen:
  - Carriage Return gefolgt von Linefeed (z.B. Windows),

    Carriage Return `CR`: ASCII Code 13, Linefeed/Newline `LF`: ASCII-Code 10.
  - Nur Linefeed (z.B. UNIX),
  - Nur Carriage Return (z.B. altes MacOS).
- Ein Ctrl-Z (ASCII 26, "`SUB`") am Dateiende wird ignoriert.

Leerplatz, Kommentare (2)

- Üblich ist folgende Einrückung:

```
anz_stellen = 1;
while(n >= 10) {
    n = n / 10;
    anz_stellen = anz_stellen + 1;
}
```

- Die abhängigen Anweisungen werden also eingerückt.

  In "Code Conventions for the Java Programming Language" wird eine Einrückung um vier Zeichen empfohlen, wozu bei tieferen Schachtelungen auch Tabulator-Zeichen benutzt werden können. Am verbreitetsten ist, alle 8 Zeichen eine Tabulator-Position zu haben. Ein einzelnes "Tab"-Zeichen wirkt dann also wie 8 Leerzeichen, genauer positioniert es auf die nächste durch 8 teilbare Spaltenposition (falls man bei 0 anfängt zu zählen).
  Man kann die Tabulatorbreite im Editor möglicherweise einstellen, aber ein Druckprogramm verhält sich dann eventuell anders.

Leerplatz, Kommentare (3)

- Editoren mit Syntax-Unterstützung für Java können die Einrückung automatisch machen.

  Es gibt auch "Pretty Printer", die Programmtext nachträglich formatieren. Es kommt dabei allerdings nicht in allen Fällen das heraus, was man manuell gemacht hätte.

- Man sollte möglichst Zeilen breiter als 80 Zeichen vermeiden.

  80 Zeichen sind eine übliche Standardbreite für Editorfenster, und Druckprogramme sollten 80 Zeichen pro Zeile ausgeben können (noch sicherer: 79 Zeichen). Falls die Zeilen (z.B. durch Einrückungen) sehr lang werden, sollte man über eine Strukturierung mit Prozeduren nachdenken. Dies ist mein persönlicher Stil, Sie dürfen gerne anderer Meinung sein. Dem Compiler ist es egal.

Leerplatz, Kommentare (4)

- Zeilenstruktur, Einrückungen, u.s.w. werden schon in der lexikalischen Analyse entfernt.
- Der “eigentliche Compiler” sieht sie gar nicht mehr.
- Z.B. geschieht hier nicht, was der Programmierer erwartet:

```
if(x < 0)
    x = -x;
    System.out.println('-');
System.out.println(x);
```

- Es gibt (in den meisten Compilern) nicht einmal eine Warnung.

Leerplatz, Kommentare (5)

- Kommentare haben in Java zwei mögliche Formen:
  - Von `//` bis zum Ende der Zeile ("end-of-line comment").
  - Von `/*` bis `*/` ("traditional comment").

    Man kann solche Kommentare nicht schachteln.
- Die zweite Form ist etwas gefährlich: Vergisst man, den Kommentar zu schließen, werden möglicherweise größere Teile des Programms übersprungen.

  Bis zum Ende des nächsten Kommentars. Kommentare dieser Form werden auch verwendet, um bewußt einen Teil des Programmtextes "auszukommentieren" (temporär zu entfernen). Das funktioniert allerdings nicht, wenn der Programmtext schon einen solchen Kommentar enthält: Dann endet die Auskommentierung am Ende des enthaltenen Kommentars.

Leerplatz, Kommentare (6)

- Speziell für Java entwickelte Editoren (oder mit Java-Modus) stellen Kommentare in einer anderen Farbe als normalen Programmtext dar.
- Im allgemeinen ist es hilfreicher, größere Blöcke von Kommentar zu haben, als jede einzelne Zeile zu kommentieren.
- Klassisches schlechtes Beispiel:

```
i = i + 1; // Erhöhe i um 1.
```

- Das Programm `javadoc` wertet Kommentare aus, die mit `/**` beginnen, und erstellt daraus Programm-Dokumentation.

Inhalt

1. Compiler
2. Unicode
3. Leerplatz, Kommentare
4. Bezeichner
5. Datentyp-Konstanten
6. Operatoren

Bezeichner (1)

- Ein Bezeichner ("identifier") ist ein Name für eine Variable, eine Prozedur, einen Datentyp, etc.
- Ein Bezeichner ist eine Folge von Buchstaben und Ziffern, die mit einem Buchstaben beginnt. Die Zeichen "`_`" und "`$`" zählen dabei als Buchstaben.

  Das Zeichen "$" wird in Quellcode verwendet, der von Programmen erzeugt wurde, z.B. kann man aus einer Grammatik-Spezifikation automatisch einen Parser erzeugen lassen. In handgeschriebenem Java-Code sollte man es besser vermeiden.
- Beispiele:
  - Korrekt: `x`, `x2`, `X2B`, `das_ist_ein_Bezeichner`.
  - Nicht korrekt: `25m`, `KD#`, `a b`, `a/*sowas*/b`.

Bezeichner (2)

- Groß- und Kleinschreibung werden unterschieden, "`x`" und "`X`" sind zwei verschiedene Namen.

  Es ist guter Stil, leicht zu verwechselnde Namen zu vermeiden. Daher sollte man normalerweise nicht gleichzeitig beide Namen benutzen.

- Umlaute und nationale Zeichen sind in Bezeichnern möglich, aber bei erfahrenen Programmierern eher unüblich.

  In älteren Programmiersprachen geht es meist nicht. In Java ist es auch möglich, dass man die Codierung der Quelldatei beim Aufruf des Compilers explizit angeben muss.

- Namen sollten möglichst einheitlich nach bestimmten Konventionen gewählt werden, dann kann man sie sich leichter merken.

  Auch wenn mehrere Personen zusammen an einem Projekt arbeiten, sollte ein einheitlicher Programmierstil verwendet werden.

Bezeichner (3)

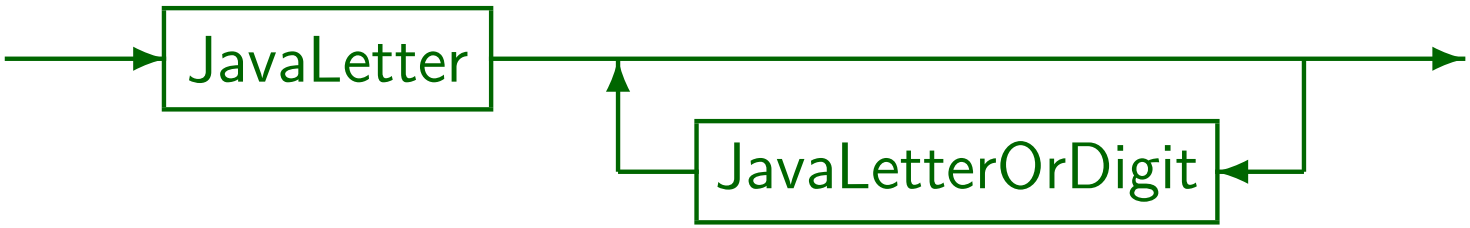
- Bezeichner sollten zum Verständnis des Programms hilfreich sein ("selbstdokumentierend").
- Einbuchstabige Bezeichner sind normalerweise nur akzeptabel, wenn sie nur in einem kleinen Abschnitt des Programms verwendet werden (wenige Zeilen).

  Oder man nichts über die Daten weiss (beliebiger Wert des Datentyps).

- Folgende einbuchstabigen Bezeichner sind üblich:
  - `i`, `j`, `k`, `n`, `m` für ganze Zahlen,
  - `c` für einzelne Zeichen,
  - `s` für Zeichenketten (Strings),
  - `x`, `y`, `z` für Gleitkommazahlen (oder Koordinaten!),
  - `o` für beliebige Objekte,
  - `e` für Exceptions.

Bezeichner (4)

- Identifier:



- Dabei sind die Schlüsselworte (siehe unten) ausgeschlossen.

  Ebenso die booleschen Literale `true` und `false` und das "Null Literal" `null`.

- "JavaLetter" enthält die ASCII Buchstaben `A`, ..., `Z` und `a`, ..., `z` und den Unterstrich "`_`" und das Dollarzeichen "`$`".

  Außerdem weitere Unicode-Buchstaben, z.B. die deutschen Umlaute. Genauer: Alle Zeichen, für die `isJavaIdentifierStart` der Klasse `Character` wahr ist: [http://docs.oracle.com/javase/7/docs/api/java/lang/Character.html]

- "JavaLetterOrDigit" enthält zusätzlich die Ziffern `0`, ..., `9`.

  Genauer: Alle Zeichen, für die die Methode `isJavaIdentifierPart` wahr liefert.

Schlüsselworte (1)

- Manche Buchstabenfolgen haben eine spezielle Bedeutung in Java, z.B. `if`, `while` ("Schlüsselworte").
- Diese Buchstabenfolgen sind Ausnahmen zu der Regel, dass man beliebige Folgen von Buchstaben als Bezeichner für Variablen etc. verwenden darf.
- Sie sind von der Sprache Java reserviert und heißen daher auch "reservierte Worte".
- Z.B. kann man keine Variable mit Namen "`if`" deklarieren:

```
int if; // Syntaxfehler!
```

Schlüsselworte (2)

- Solche Fehler geben oft ziemlich merkwürdige Fehlermeldungen des Compilers:

```
Hello.java:5: not a statement
        int if;
        ^
Hello.java:5: ';' expected
        int if;
           ^
...
8 errors
```

- Bei Editoren, die etwas Java-Syntax kennen, werden reservierte Worte/Schlüsselworte in einer anderen Farbe als normale Bezeichner angezeigt.

Schlüsselworte (3)

- Ansonsten wird vom Programmierer erwartet, dass er die Schlüsselworte der Sprache auswendig kennt und als Bezeichner vermeidet.
- Das ist naturgemäß bei Sprachen einfacher, die wenig reservierte Worte haben:
  - C hat 32
  - C++ hat 74
  - Java hat 50 (plus 3 Literale)
  - SQL hat (je nach Dialekt) ca. 300.
- Es gibt auch Sprachen ganz ohne reservierte Worte.

Schlüsselworte (4)

```
abstract    double      int         super
assert      else        interface   switch
boolean     enum        long        synchronized
break       extends     native      this
byte        final       new         throw
case        finally     package     throws
catch       float       private     transient
char        for         protected   try
class       if          public      void
const       goto        return      volatile
continue    implements  short       while
default     import      static
do          instanceof  strictfp
```

Schlüsselworte (5)

- Die Schlüsselworte `const` und `goto` sind zwar reserviert, werden aber (bisher) von der Sprache nicht verwendet.

  Der Java-Compiler kann bessere Fehlermeldungen erzeugen, wenn Umsteiger von C oder C++ diese Schlüsselworte verwenden (wenn sie nicht reserviert wären, müsste der Compiler sie als Variablennamen akzeptieren).

- Zusätzlich zu den obigen 50 Worten können auch noch die folgenden Worte nicht als Bezeichner verwendet werden:
  - `true`: Datentyp-Literal für Wahrheitswert "wahr",
  - `false`: Datentyp-Literal für Wahrheitswert "falsch",
  - `null`: Datentyp-Literal für leere Objekt-Referenz.

    In der Java-Spezifikation steht, dass diese drei Worte formal nicht Schlüsselworte seien, sondern Literale. Sie können aber auch nicht als Bezeichner verwendet werden. Der praktische Unterschied ist mir unklar.

Schlüsselworte (6)

- Zum Vergleich: Schlüsselworte in C:

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

- Wenn man die Schlüsselworte einer Sprache alle erklären kann, hat man schon einen großen Teil der Sprache verstanden.

Inhalt

1. Compiler
2. Unicode
3. Leerplatz, Kommentare
4. Bezeichner
5. Datentyp-Konstanten
6. Operatoren

Konstanten/Literale

- Datenwerte kann man in Programmen als Konstanten aufschreiben:
  - `true` und `false` für den Datentyp `boolean`.
  - z.B. `123` für den Datentyp `int`.
  - z.B. `1.23` oder `1.23E-4` für den Datentyp `double`.
  - z.B. `'a'` für den Datentyp `char`.
  - z.B. `"abc"` für Strings.
- Manchmal spricht man auch von Literalen, um den Unterschied zu symbolischen Konstanten wie `Math.PI` zu betonen.

Ganzzahlige Konstanten (1)

- Eine ganze Zahl wird normalerweise dezimal als Folge von Ziffern 0 bis 9 dargestellt.
- Vermeiden Sie führende Nullen, wenn Sie nicht genau wissen, was Sie tun (Oktalschreibweise, Basis 8).

  Die Oktal- (Basis 8), die Hexadezimal- (Basis 16) oder die Binär- (Basis 2) Schreibweise sind nützlich, wenn man Bitfolgen darstellen will (z.B. als effiziente Codierung mehrerer boolescher Werte in einem `int`). Eines Tages werden Sie das brauchen, aber eher nicht in dieser Vorlesung.
- Formal gehört das Minuszeichen - nicht zur Konstante, sondern ist ein Operator, den man darauf anwendet.

  Der Compiler führt Berechnungen mit Konstanten schon zur Compile-Zeit aus, so dass keinen Laufzeit-Unterschied gibt. Man darf aber z.B. "`- 1`" schreiben, weil es zwei Token sind. Die spezielle Konstante 2147483648 (2^{31}) darf nur als Argument von "-" verwendet werden (sonst Überlauf).

Ganzzahlige Konstanten (2)

- Die Notation im Binär-, Oktal- oder Hexadezimalsystem ist nützlich, wenn man eigentlich mit Bitmustern arbeiten will.

  Oktal entspricht jede Ziffer 3 Bits, hexadezimal 4 Bits.

- Wenn die Zahl mit `0` beginnt, wird sie als Angabe im Oktalsystem (zur Basis 8) verstanden.

  Die Ziffern `8` und `9` sind dann natürlich verboten.

  Z.B. $123 = 1 * 10^2 + 2 * 10 + 3$ und $0123 = 1 * 8^2 + 2 * 8 + 3 * 1 = 83$.

- Man kann Zahlen auch hexadezimal aufschreiben, dazu muß die Konstante mit `0x` oder `0X` beginnen.

  Hexadezimal: Zur Basis 16. Zusätzliche Ziffern: `a/A` (10), `b/B` (11), `c/C` (12), `d/D` (13), `e/E` (14), `f/F` (15). Z.B. `0xFF`$=15 * 16 + 15 = 255$.

- Auch binär (Basis 2) ist möglich, die Zahl muss dann mit `0B` oder `0b` beginnen, und darf nur die Ziffern `0`, `1` enthalten.

Ganzzahlige Konstanten (3)

- Es ist (seit Java 7) möglich, Unterstriche "_" in Zahlen zur Strukturierung und Verbesserung der Lesbarkeit zu verwenden, z.B.

  `123_456` oder `0xFFFF_FFFF`

  Ganz am Anfang und ganz am Ende der Ziffernfolge ist kein Unterstrich erlaubt. Es darf also auch nach `0x` nicht gleich ein Unterstrich folgen. Bei Oktalschreibweise ist es nach der führenden `0` dagegen erlaubt (das ist ja schon eine Ziffer). Auch mehrere Unterstriche nacheinander wären legal.

- In Java bis Version 6 gibt es wie in C und C++ keine Binärschreibweise und keine Unterstriche in Zahlkonstanten.

  Wenn man Programme entwickelt, stellt sich natürlich die Frage, ob man die neuesten Sprach-Features ausnutzen will. Dann können die Programme auf einem Rechner mit älteren Java-Installationen nicht übersetzt bzw. ausgeführt werden. Java 7 gibt es seit Juli 2011.

Ganzzahlige Konstanten (4)

- Wir haben bisher nur den Datentyp `int` für ganze Zahlen gebraucht. Es gibt aber noch andere Typen, nämlich
  - `long` für sehr große Zahlen (64 Bit)
  - `byte` (8 bit) und `short` (16 Bit) für kleine Zahlen.
- Man braucht diese Typen nicht gleich.

  Sie sind eher für Spezialanwendungen und werden in Kapitel 5 näher behandelt.
- Wenn man an eine Zahlkonstante (Ziffernfolge) den Suffix `L` oder `l` anhängt, hat die Konstante den Typ `long`, sonst den Typ `int`.

  In C++ werden Konstanten, die zu groß für den Typ `int` sind, automatisch als `long` aufgefasst. Wenn man dagegen in Java den Suffix weglässt, und die Konstante zu groß für ein `int` ist, muß der Compiler einen Fehler melden.

Ganzzahlige Konstanten (5)

- Es gibt formal keine Konstanten der Typen `byte` und `short`.
- Eine Zuweisung an eine Variable vom Typ `byte`, `short`, `char` ist aber möglich,
  - wenn auf der rechten Seite ein konstanter Ausdruck vom Typ `int` steht,
  - und der Wert in den jeweiligen Typ der Variable passt.
- Beim Methoden-Aufruf findet dagegen keine automatische Typ-Anpassung von konstanten Ausdrücken statt.

  Man muss explizit eine Typ-Umwandlung vornehmen, z.B. "`(byte) 0`". Der Grund ist, dass es mehrere Methoden mit gleichem Namen geben kann, die sich in den Typen der Parameter unterscheiden. Eine automatische Typ-Anpassung würde die Auswahlregeln verkomplizieren.

Ganzzahlige Konstanten (6)

Aufgabe:

- Was gibt dieses Programm aus?

```
class Literale {
    public static void main(String[] args) {
        System.out.println(1_00);
        System.out.println(0x64);
        System.out.println(0144);
    }
}
```

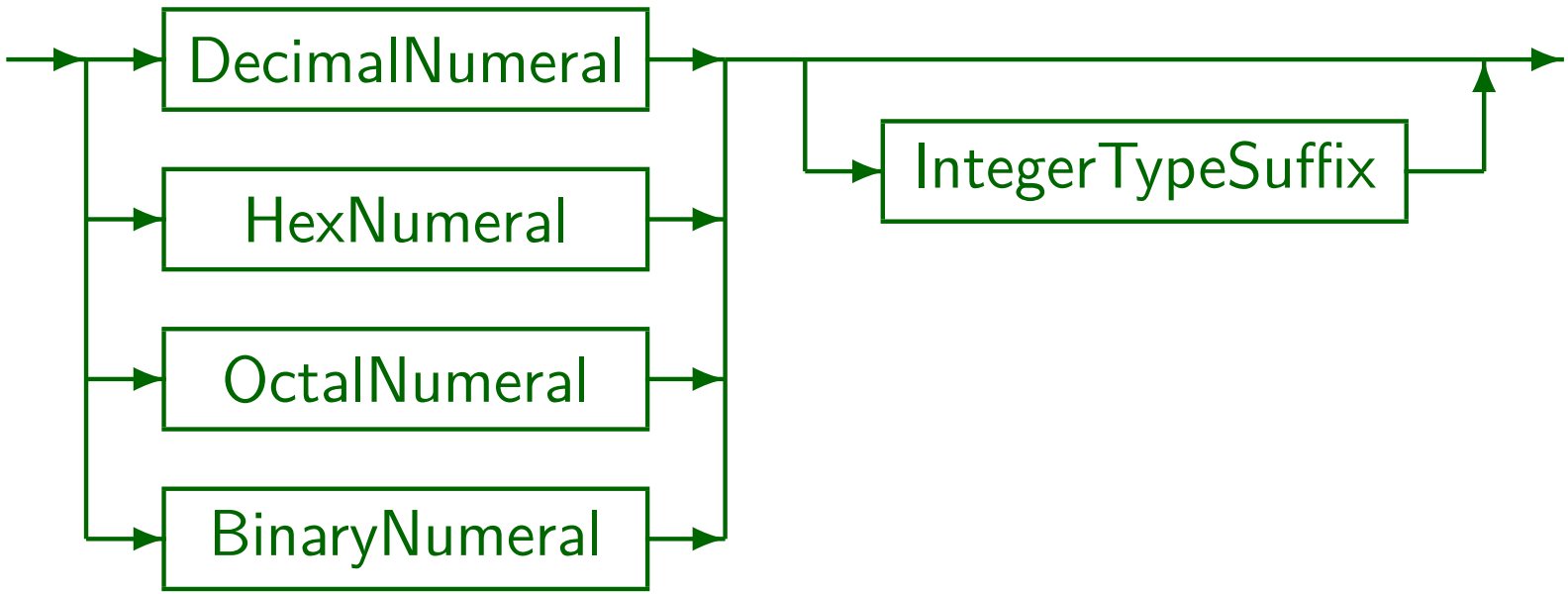
- Tipp: Wenn Sie eine ganze Zahl (vom Typ `int`) z.B. oktal ausgeben wollen, schreiben Sie:

```
System.out.println(String.format("%o", 100));
```

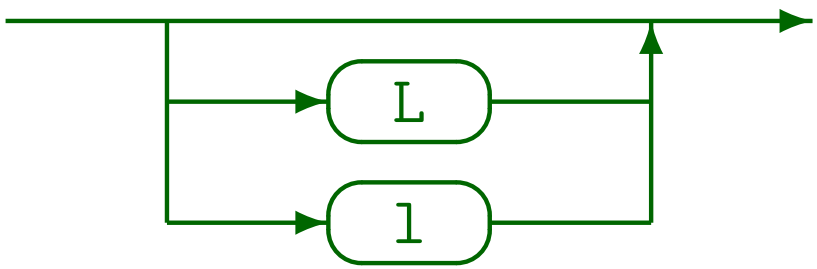
  Hexadezimal geht mit `"%x"`.

Ganzzahlige Konstanten: Syntax (1)

- IntegerLiteral:

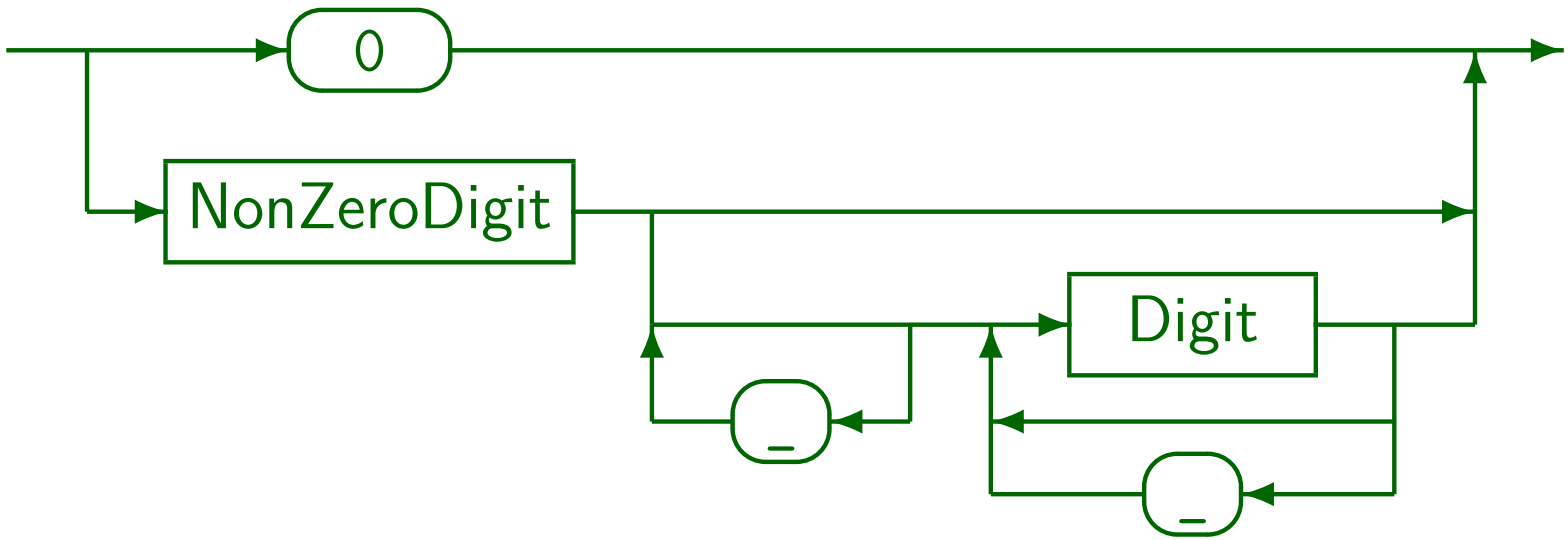


- IntegerTypeSuffix:



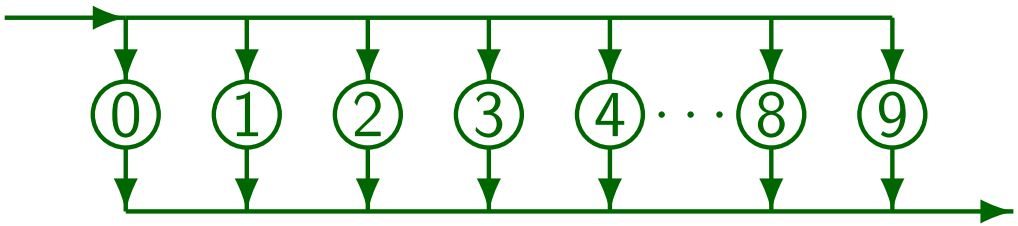
Ganzzahlige Konstanten: Syntax (2)

- DecimalNumeral:

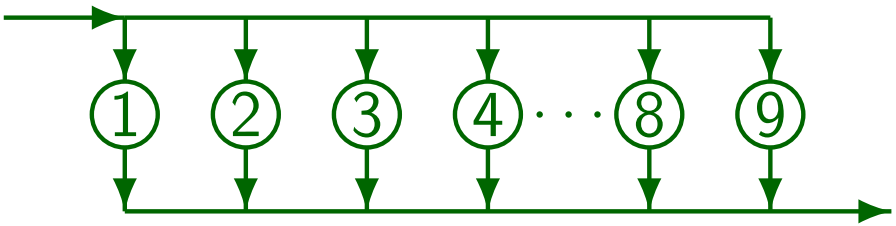


Ganzzahlige Konstanten: Syntax (3)

- Digit:

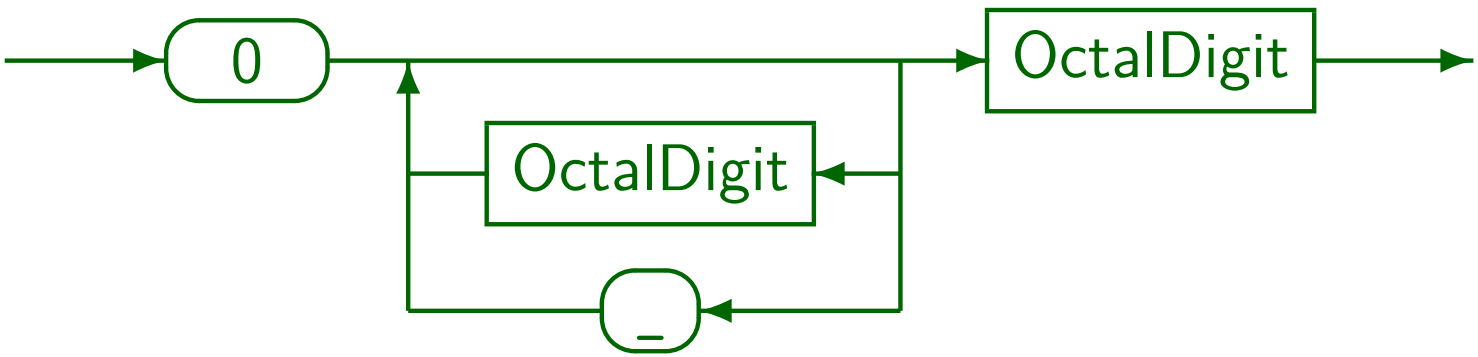


- NonZeroDigit:

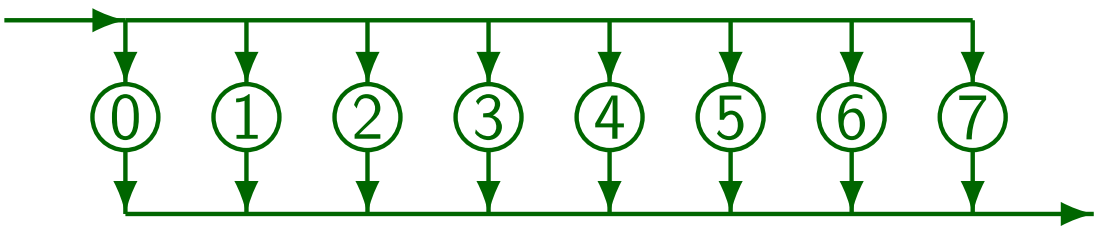


Ganzzahlige Konstanten: Syntax (4)

- OctalNumeral:

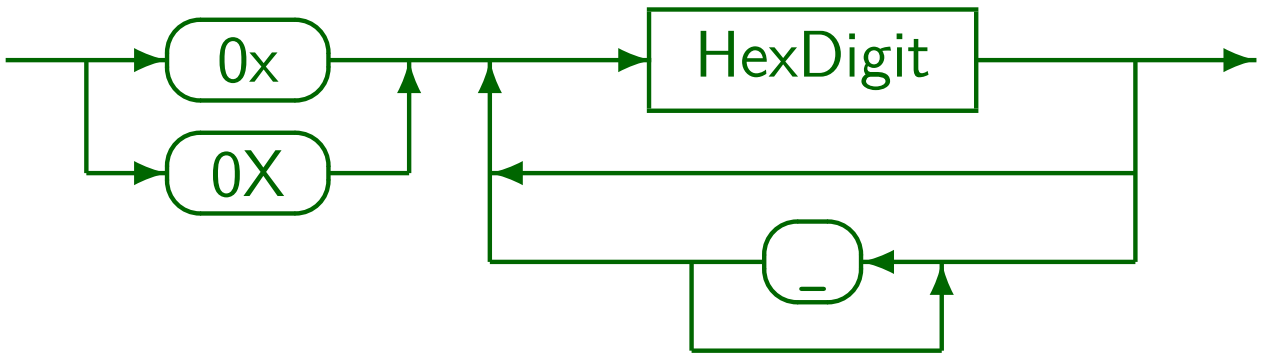


- OctalDigit:

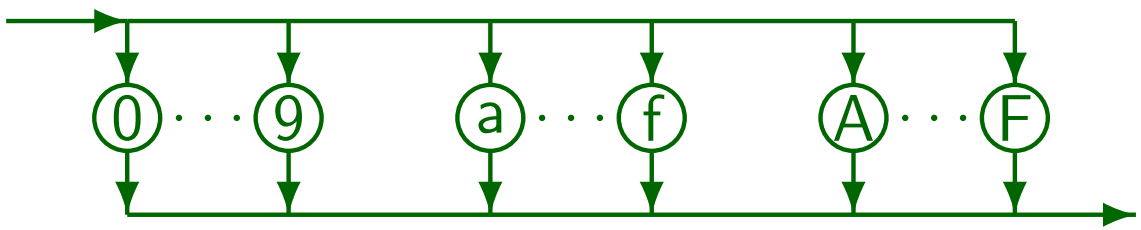


Ganzzahlige Konstanten: Syntax (5)

- HexNumeral:

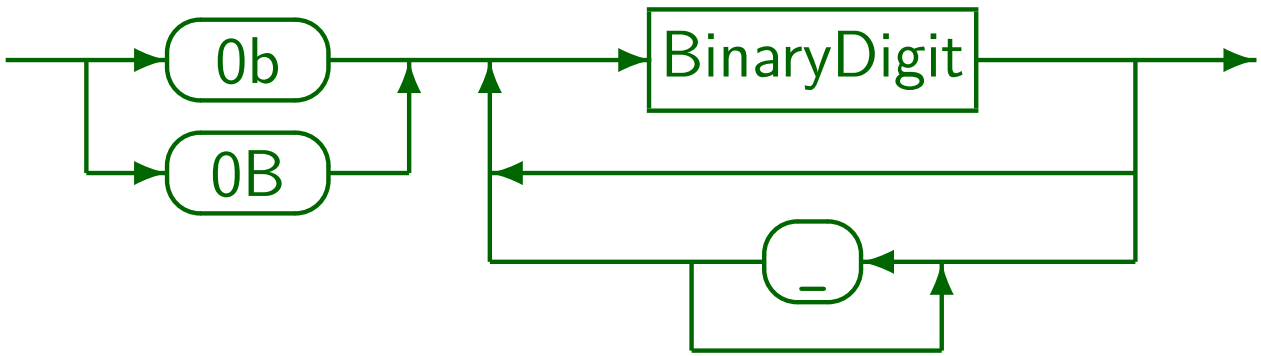


- HexDigit:

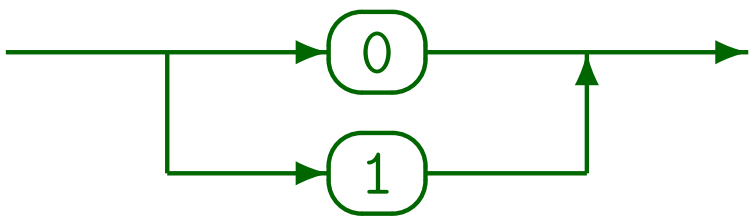


Ganzzahlige Konstanten: Syntax (6)

- BinaryNumeral:



- BinaryDigit:



Gleitkomma-Konstanten (1)

- Eine Gleitkomma-Konstante (für reelle Zahlen beschränkter Genauigkeit), z.B. `12.34E-56` ($= 12.34 * 10^{-56}$) besteht aus
  - einem ganzzahligem Anteil

    Dies ist eine Folge von Dezimalziffern.
  - einem Dezimalpunkt "`.`",
  - einem gebrochenen Anteil

    Dies ist eine Folge von Dezimalziffern.
  - ein `e` or `E`,
  - einem Exponenten (zur Basis 10)

    Folge von Dezimalziffern, mit optional einem Vorzeichen.
  - Optional einem Typ-Suffix: `f`, `F`, `d`, `D`.

Gleitkomma-Konstanten (2)

- Der ganzzahlige Anteil oder der gebrochene Anteil können fehlen (aber nicht beide).

  Ein einzelner Punkt ohne etwas davor oder dahinter würde ja keinen Sinn machen. Aber z.B. `3.` und `.3` sind zulässig.

- Der Dezimalpunkt oder der Exponent (mit `e`/`E`) können fehlen (aber nicht beide).

  Wenn beide fehlen, ist es ja eine ganze Zahl.

- Z.B. sind legal: `12.3`, `12.`, `.34`, `1E0`, `1.E-2`, `.2E+5`.

- Gleitkomma-Konstanten haben den Typ `double`, nur der Suffix `f`/`F` macht es zu `float`.

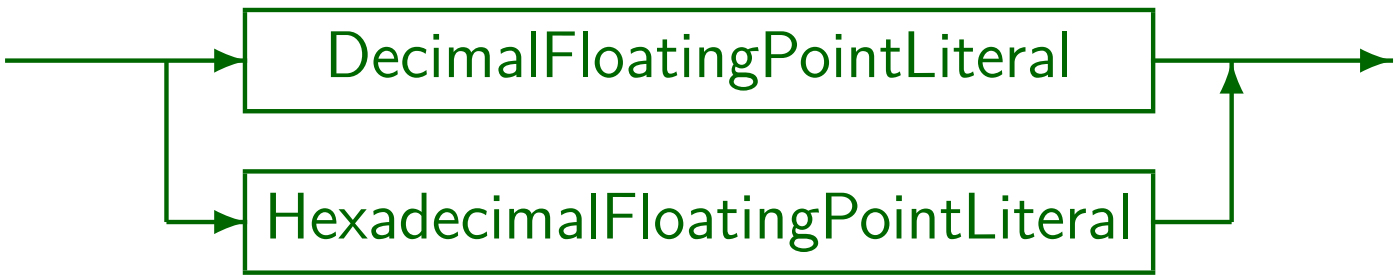
  Man darf den Suffix `d`/`D` hinschreiben, um sehr klar zu machen, dass es ein `double` ist, aber das ändert nichts. Java hat keinen Typ "`long double`" wie C++.

Gleitkomma-Konstanten (3)

- Hexadezimalschreibweise ist auch möglich, dann verwendet man `p` bzw. `P` für den Exponenten zur Basis 2, z.B. ist `0x1p-2` der Wert `0.25`.
- Dies ist hauptsächlich interessant, wenn man sich für die interne Darstellung der Zahlen interessiert.

Gleitkomma-Konstanten (4)

- FloatingPointLiteral:

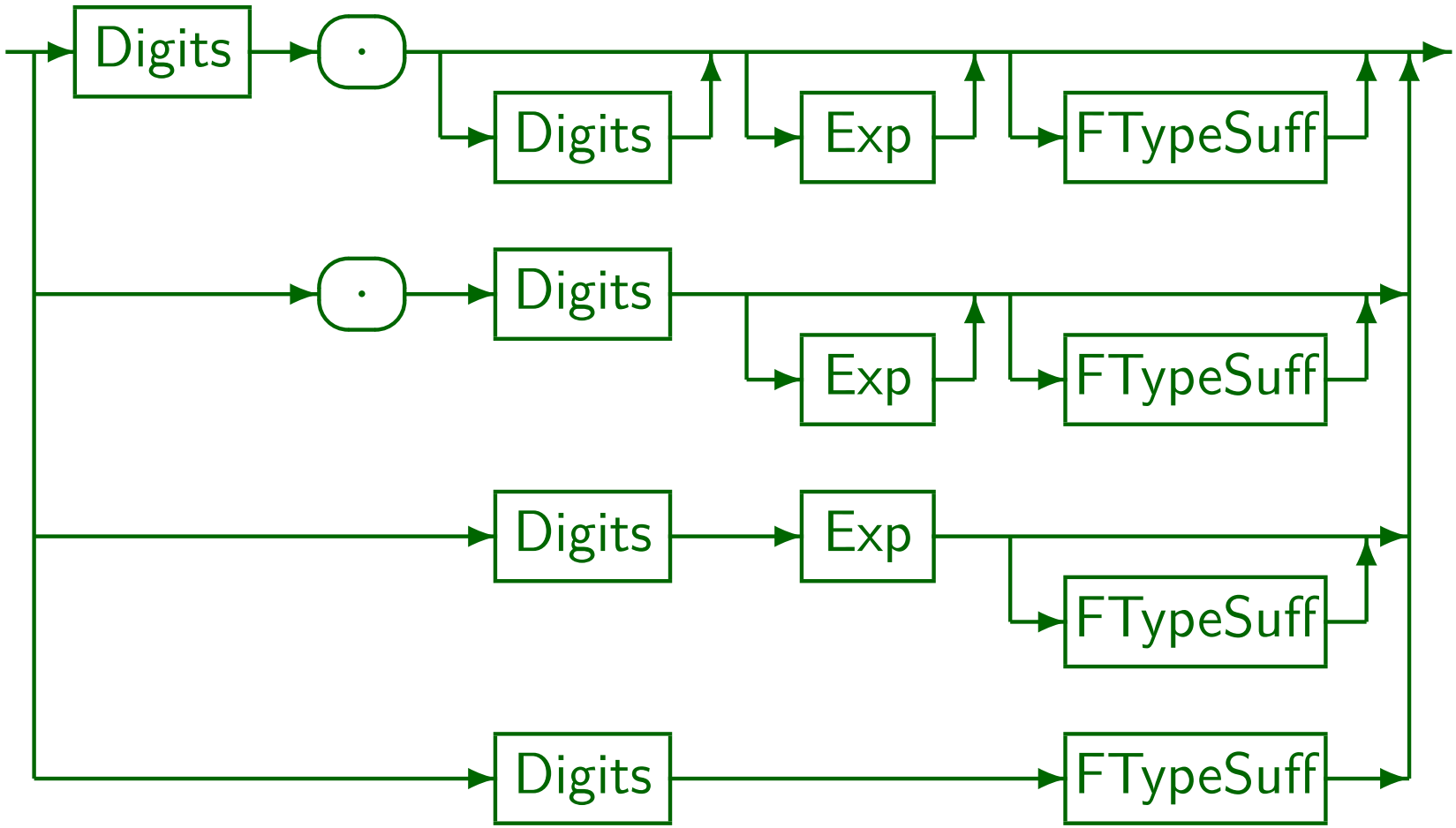


- Die hexadezimale Schreibweise von `double`-Werten ist nur für Spezialisten interessant und wird hier nicht weiter erläutert.

  Bei Bedarf können Sie die "Java Language Specification" einsehen.
  PDF: [http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf] (644+xxv Seiten)
  HTML: [http://docs.oracle.com/javase/specs/jls/se7/html/index.html]

Gleitkomma-Konstanten (5)

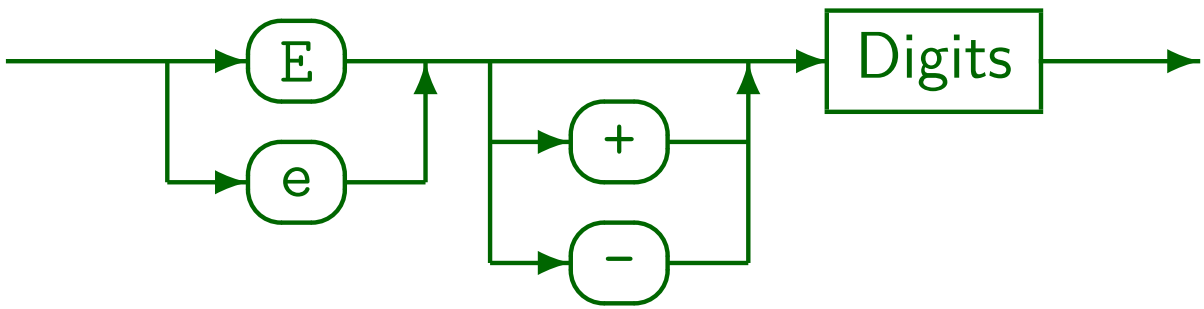
- DecimalFloatingPointLiteral:



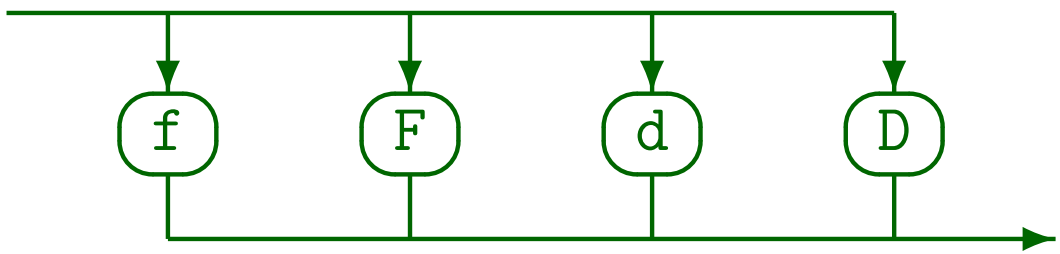
Gleitkomma-Konstanten (6)

- Digits:

- Exp (“ExponentPart”):



- FTypeSuff (“FloatTypeSuffix”):



Zeichenkonstanten (1)

- Zeichenkonstanten bestehen aus einem Zeichen in einfachen Anführungszeichen (Apostroph), z.B. `'a'`.

  Das Zeichen kann nicht ein einfaches Anführungszeichen selbst sein, auch kein Zeilenumbruch oder Rückwärtsschrägstrich `\`.

- Anstelle eines Zeichens kann man auch eine der Escape-Sequenzen verwenden, die auf der nächsten Folie aufgelistet sind.

- Die "Unicode Escapes" kann man überall verwenden, natürlich auch in Zeichenkonstanten: `'\u00CA'` wäre `'Ä'`.

  Carriage Return und Linefeed kann man so nicht eingeben, weil Zeilenumbrüche in Zeichenkonstanten verboten sind. Auch `'` und `\` gehen so nicht, weil Unicode Escapes vor der eigentlichen lexikalischen Analyse ersetzt werden. Der Scanner sieht dann also das Zeichen, und nicht `\uXXXX`. Das ist ein Unterschied zur Oktalschreibweise (siehe nächste Folie).

Zeichenkonstanten (2)

`\n`	Newline/Linefeed (LF)
`\t`	Horizontal tab (TAB, HT)
`\b`	Backspace (BS)
`\r`	Carriage Return (CR)
`\f`	Formfeed (FF)
`\\`	Backslash (\)
`\'`	Einfaches Anführungszeichen/Apostroph (')
`\"`	Doppeltes Anführungszeichen (")
`\`*ooo*	Character with code *ooo* (in octal)

Man darf auch nur eine oder zwei Oktalziffern verwenden, aber dann kann in einer Zeichenkette natürlich keine Ziffer folgen (für einzelne Zeichen ist es kein Problem). Die Oktalschreibweise wird aus Kompatibilitätsgründen mit C angeboten, und funktioniert nur für Zeichen mit Codes bis 255 (aber auch für CR, LF, ', \). Normalerweise würde man `\uXXXX` verwenden (das geht zwar nicht für CR, LF, ', \, aber dafür hat man ohnehin spezielle Escape-Sequenzen).

Zeichenkonstanten (3)

- Da Java den Typ `char` auch als 16-Bit Zahlen ohne Vorzeichen (0 bis $2^{16} - 1 = 65535$) auffasst, sind z.B. Zuweisungen der folgenden Art möglich:

```
char c = 0;
```

  `0` ist Konstante des Typs `int`. Wenn der Compiler bei konstanten Ausdrücken erkennen kann, dass der Wert in ein `char` passt, läßt er die Zuweisung zu.

- Beachte: `'0'` steht für die Zahl 48 und nicht die Zahl 0!

  Hier wird der Unicode-Wert der Ziffer "`0`" verwendet. Die Zahlwerte der Zeichen kann man in den Unicode-Tabellen nachschauen, bis 127 reicht auch jede ASCII-Tabelle (wie etwa in Kapitel 1 abgedruckt).
  Während Java ausdrücklich Unicode verwendet, so dass der Zahlwert jeder Zeichenkonstante eindeutig festgelegt ist, gilt das nicht für Sprachen wie C++. In C++ sind `char`-Variablen normalerweise nur 8 Bit groß, und die Zeichencodierung hängt vom Betriebssystem ab. Für 16 Bit Codes gibt es `wchar_t`.

String Konstanten (1)

- Eine Zeichenketten-Konstante (String) ist eine Folge von Zeichen in (doppelten) Anführungszeichen z.B. `"abc"`.
- Die oben aufgelisteten Escape-Sequenzen können auch in Zeichenketten-Konstanten verwendet werden, z.B.

  `"eine Zeile\n"`.

  Dies ist also eine Zeichenketten-Konstante aus 11 Zeichen, wobei das letzte das Linefeed-Zeichen ist (`\u000A`, ASCII 10). Das vorletzte Zeichen ist das "e".
- Zeichenketten-Konstanten dürfen keine Zeilenumbrüche enthalten, d.h. man kann sie nicht in einer Zeile mit `"` öffnen und in der nächsten mit `"` schliessen.

  Zeilenumbrüche kann man als `\n` eingeben.

String Konstanten (2)

- Wenn man lange Texte eingeben will, kann man mehrere Zeichenketten-Konstanten verwenden, und jeweils den Konkatenations-Operator "+" dazwischen schreiben.

```
"Dies ist die erste Zeile\n" +
"und dies Zeile 2.\n"
```

- Konstante Ausdrücke werden schon zur Compilezeit ausgewertet, d.h. der Effekt ist genau gleich, wie wenn man alle Zeichen in eine lange Konstante geschrieben hätte:

```
"Dies ist die erste Zeile\nund dies Zeile 2.\n"
```

Es gibt keinen Laufzeit-Nachteil und die Abbildung gleicher Zeichenketten in dasselbe Objekt (siehe nächste Folie) gilt auch in diesem Fall.

String Konstanten (3)

- Der Compiler erzeugt Objekte der Klasse `String` für die Zeichenketten-Konstanten, und zwar für gleiche Zeichenketten-Konstanten auch nur ein Objekt.

  Da die Objekte nicht geändert werden können, ist das kein Problem.
  Es ist sogar ein Vorteil, da `String`-Variablen, die mit Zeichenketten-Konstanten initialisiert sind, mittels == verglichen werden können (gleiches Objekt).
  Ansonsten muss man für den Vergleich von Zeichenketten die Methode "`equals`" verwenden, also z.B. `s.equals("abc")`, wobei `s` eine Variable vom Typ `String` ist. Wenn `String`-Objekte zur Laufzeit erzeugt werden (z.B. aus Benutzer-Eingaben) kann es verschiedene Objekte geben, in denen die gleiche Zeichenkette gespeichert ist. Der Gleichheits-Operator == prüft aber nur, ob es sich um das gleiche Objekt handelt.

- Dies gilt auch, wenn die Zeichenketten-Konstanten in unterschiedlichen Klassen (oder sogar Packages) stehen.

Konstanten für Wahrheitswerte (1)

- Entsprechend den zwei Wahrheitswerten gibt es zwei Konstanten des Datentyps `boolean`:
  - `true`: wahr
  - `false`: falsch

Konstanten für Wahrheitswerte (2)

- Die Programmiersprache C hatte keinen eigenen Typ für Wahrheitswerte. Es wurde der Typ `int` benutzt:
  `0` war falsch, alles andere wahr.

  Manche Bedingungen lassen sich so sehr kompakt schreiben, z.B. "`if(i)`" statt "`if(i != 0)`". In Java ist der explizite Vergleich nötig. Der Vorteil ist, dass Fehler leichter bemerkt werden. C++ hat einen booleschen Typ, der allerdings "`bool`" heißt (in Java "`boolean`"). C++ erlaubt sehr großzügige automatische Umwandlungen von `int` in `bool` und umgekehrt, im Endeffekt hat man eine ganz ähnliche Situation wie in C — der Typ `bool` dient mehr zur Dokumentation. Java ist hier wesentlich strenger.

Konstante für Null-Referenzen

- `null`: Konstante für "Referenz auf nichts" (Null-Referenz).
- Man kann diesen Wert jeder Variable von einem Referenz-Typ zuweisen, dass sind Klassen, Interfaces und Arrays.

  Formal ist `null` der einzige Wert eines speziellen (unbenannten) Null-Typs, und kann aber in beliebige Referenztypen umgewandelt werden.
  Referenztypen werden später ausführlich behandelt.

Inhalt

1. Compiler
2. Unicode
3. Leerplatz, Kommentare
4. Bezeichner
5. Datentyp-Konstanten
6. Operatoren

Operatoren (1)

- `+` (Addition, String-Konkatenation), `-` (Subtraktion),
  `*` (Multiplikation), `/` (Division), `%` (Divisionsrest/Modulo).
- `==` (gleich), `!=` (ungleich), `<` (kleiner), `>` (größer),
  `<=` (kleinergleich), `>=` (größergleich).
- `&&` (bedingtes logisches und), `||` (bed. log. oder),
  `!` (logisches nicht).
- `&` (Bit-und, logisches und), `|` (Bit-oder, logisches oder),
  `^` (Bit-XOR, logisches XOR), `~` (Bit-Komplement),
  `<<` (Linksshift),
  `>>` (Rechtsshift mit Vorzeichen-Erhaltung),
  `>>>` (Rechtsshift mit Null-Erweiterung).

Operatoren (2)

- `=` Zuweisung.
- `++` (Inkrement $+1$), `--` (Dekrement -1).
- `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `>>>=` (Abkürzungen für Zuweisungen).
- `? :` (bedingter Ausdruck).
- `.` (Selektion einer Komponente eines Objektes)
- Auch einige Schlüsselworte werden wie Operatoren verwendet (z.B. `instanceof`).

Auch den Array-Zugriff `[ ]` kann man als Operator verstehen.

Weitere Token: Trennzeichen

- `;` (Ende einer Anweisung).
- `,` (Trennung von Parametern)
- `{`, `}` (begin, end: Block-Klammern, Array-Initialisierung).
- `(`, `)` (Klammern).
- `[`, `]` (Array Klammern).
- `<`, `>` (für Typ-Parameter).
- `:` (`switch`-Statement, bedingter Ausdruck, Labels für `break`).

Längste Präfixe

- Die lexikalische Analyse liefert als nächstes Token immer den längsten Präfix vom Rest der Eingabe, der noch ein gültiges Token ist.

  D.h. die lexikalische Analyse liest so lange weitere Zeichen ein, wie das aktuelle Token sich noch verlängern läßt. Erst wenn das aktuelle Token zusammen mit dem nächsten Zeichen kein gültiges Token mehr wäre, wird das aktuelle Token für beendet erklärt (und an den Parser ausgeliefert). Das nächste Zeichen gehört dann schon zum nächsten Token (oder ist Leerplatz, der Übersprungen wird).

- Z.B. würde bei der Eingabe `+++` zuerst der Operator `++` geliefert, und danach der Operator `+`.

- Man sollte solchen kryptischen Code vermeiden.