2. Die Programmiersprache Java

2.1 Was ist Java?

Java ist

- einfach zu lernen
- objektorientiert
- maschinenunabhängig
- eine der dominierenden und zukunftsträchtigen Sprachen in der Industrie (wie C, C++)
- mit reichhaltigen Klassenbibliotheken ausgestattet (z. B. Swing)

Was ist Java? (Fortsetzung)

- für moderne Programmiertechniken ausgelegt wie
  - Parallelverarbeitung mit Threads
  - Komponenten mit Beans
  - verteiltes Rechnen mit RMI
  - Multimedia-Programmierung mit dem Java Media Framework

Objektorientierung

Moderne Programmiersprachen sind gegenwärtig meist objektorientiert. Beispiele hierfür sind C++, Smalltalk, Eiffel und Delphi. Java ist hierbei eine Weiterentwicklung von C++.

„Objektorientierte Programmierung ist eine Implementierungsmethode, bei der Programme als kooperierende Ansammlungen von Objekten angeordnet sind. Jedes dieser Objekte stellt eine Instanz einer Klasse dar, und alle Klassen sind Elemente einer Klassenhierarchie, die durch Vererbungsbeziehungen gekennzeichnet ist.“ (Booch 1994).

* einige Texte zu diesem Kapitel sind dem Buch von Stephan Fischer und Abdulmotaleb El Saddik: Open Java, Springer-Verlag, 1999 entnommen.

Objekt und Klasse

Der Grundbegriff der Objektorientierung ist das **Objekt**. Ein Objekt ist ein abgeschlossenes Element eines Programms, das darauf ausgelegt ist, bestimmte Aufgaben zu erfüllen. Ein Objekt hat eine eindeutige Identität, die es von allen anderen Objekten unterscheidet, und ein bestimmtes Verhalten, das vom Zustand des Objekts abhängen kann.

Objekte mit gleichem Verhalten werden in **Klassen** gruppiert. Eine Klasse beschreibt also das Verhalten und die Struktur von Objekten. Klassen werden dazu verwendet, Objekte (Instanzen) zu erzeugen (diese zu *instantiieren*). Klassen werden in Java durch das Schlüsselwort `class` beschrieben.

Attribute und Verhaltensweisen

Im Allgemeinen besteht jede Klasse aus zwei Komponenten: **Attributen** und **Verhaltensweisen**.

Attribute, die durch Variablen definiert werden, legen die Erscheinung, den Zustand und andere Merkmale der Klasse fest. Das **Verhalten** einer Klasse wird durch Methoden definiert.

Objekte kommunizieren miteinander, indem sie **Nachrichten** an andere Objekte senden. Empfängt ein Objekt eine Nachricht, so führt es daraufhin eine entsprechende **Methode** aus, die angibt, wie auf das Empfangen der Nachricht reagiert werden soll. Das Senden von Nachrichten kann mit dem Funktions- oder Prozeduraufruf in prozeduralen Programmiersprachen verglichen werden.

Vererbung (1)

Vererbung ist ein Mechanismus, der es einer Klasse ermöglicht, ihre gesamten Eigenschaften von einer anderen Klasse zu erben. Vererbung wird in der OOP benutzt, um zwei Konzepte zu beschreiben, die häufig zusammen verwendet werden: die **Spezialisierung** von Klassenschnittstellen und den **Import** von Implementierungen.

Spezialisierung dient dazu, ein System von Klassen nach einer Teilmengenbeziehung zu strukturieren. Einen Klasse **A** ist dann eine Spezialisierung einer anderen Klasse **B**, wenn alle Objekte von **A** die von **B** definierte Schnittstelle ebenfalls erfüllen, **A** also die Schnittstelle von **B erweitert** (`extends`). Zu beachten ist, dass einen Klasse mit erweiterter Schnittstelle eine Untermenge von Objekten beschreibt, da die erweiterte Beschreibung spezieller ist.

Vererbung (2)

Import beschreibt hingegen die Wiederverwendung existierender Implementierungen (oder Teile davon). Wenn eine Klasse **A** eine Klasse **B** erweitert, so kann sich dies also nicht nur auf deren Schnittstelle beziehen, sondern auch auf deren Implementierung, in der weitere Funktionen hinzugefügt oder vorhandene an den neuen Kontext angepasst werden können.

Polymorphie

Der Begriff **Polymorphie** (griech: Vielgestaltigkeit) besagt, dass eine Variable eines bestimmten Typs in die eines abgeleiteten Untertyps umgewandelt werden kann. Eine Variable vom (allgemeineren) Typ **Fahrzeug** könnte beispielsweise in den (spezielleren) Typ **Auto** umgewandelt werden, da sie dadurch lediglich um einige Eigenschaften erweitert wird (Konzept der Vererbung). Objektorientierte Programmiersprachen unterstützen in der Regel das Konzept der Polymorphie.

Dynamisches Binden

Die durch die Polymorphie eingeführten Möglichkeiten der Wiederverwendung und Erweiterbarkeit werden erst dann richtig ausgenutzt, wenn das Binden der Methoden einer Klasse an die eintreffenden Nachrichten dynamisch zur Laufzeit erfolgt – im Unterschied zum statischen Binden, das schon zur Übersetzungszeit des Programms vorgenommen wird. Man spricht auch von **Late Linking**, da die Komponente nicht in das aufrufende Programm eingebunden, sondern nur benutzt wird. Dieses Konzept ist für ein erweiterbares System sehr nützlich, führt allerdings oft auch zu Performance-Einbußen.

Abschließende Bemerkung

Wir werden uns in dieser einführenden Vorlesung im Wesentlichen auf den **imperativen Kern** der Sprache Java beschränken. Im Hauptstudium gibt es weiterführende Vorlesungen über die objektorientierte Programmierung, die ebenfalls die Sprache Java verwenden. Hier werden die weiterführenden Konzepte und die wichtigsten Klassenbibliotheken von Java ausführlich vorgestellt.

2.2 Erste Beispiele

Wir beginnen mit einigen Beispielen, um den Umgang mit dem Computer und dem Interpreter zu erlernen.

In der **Quelldatei** steht das Java-Programm, das der Programmierer erstellt hat (der „source code“).

Der **Java-Compiler** übersetzt das Quellprogramm in Bytecode und legt diesen in der Klassendatei ab.

In der **Klassendatei** steht der Bytecode, eine maschinenunabhängige, ausführungsnahe Repräsentation des Programms.

Der **Java-Interpreter** führt den Bytecode aus (interpretiert ihn).

Ein kleines Beispiel: Die Klasse Zaehler

(Stören Sie sich nicht daran, wenn Sie dieses Beispiel einstweilen noch nicht ganz verstehen!)

```
// Zaehler.java
class Zaehler {
  private int wert;
  int wert() { return wert; }
  void wert(int i) { wert = i; }
  void inkrementiere() { ++wert; }
  void dekrementiere() { --wert; }
}
```

Die Klasse ZaehlerTest (1)

Dient zum Testen der Klasse Zaehler.

```
// ZaehlerTest.java
import java.io.*;
class ZaehlerTest {
  public static void main(String[] args) {
    Zaehler z = new Zaehler();
    BufferedReader in = new BufferedReader(new
        InputStreamReader(System.in));
    PrintWriter out = new PrintWriter(System.out, true);
    for (;;) {
      out.println("------------------ \nZählerstand: "
          + z.wert() + "\n------------------");
      char akt = '+';
```

Die Klasse ZaehlerTest (2)

```
do {
        out.print("Aktion (+/-): ");
        out.flush();
        try {
              akt = in.readLine().charAt(0);
        } catch (IOException ioe) { }
      } while (akt != '+' && akt != '-');
      if (akt == '+')
        z.inkrementiere();
      else
       z.dekrementiere();
    }
  }
}
```

Ausführen des Programms ZaehlerTest (1)

Wenn wir den Java-Interpreter starten, beginnt die Ausführung per Konvention immer mit der Klasse `main`. Wir sehen beispielsweise:

```
------------------------------

Zählerstand: 0

------------------------------

Aktion (+/-): +

------------------------------

Zählerstand: 1

------------------------------

Aktion (+/-): +
```

Ausführen des Programms ZaehlerTest (2)

```
-------------------------------

Zählerstand: 2

-------------------------------

Aktion (+/-): -

-------------------------------

Zählerstand: 1

-------------------------------

Aktion (+/-):
```

Da wir keine spezielle Eingabe als Endebedingung vorgesehen haben, brechen wir das Programm mit CTRL-C ab.

Eine einfache Benutzeroberfläche

Im **Java Development Toolkit (JDK)** von SunSoft gibt es eine Vielzahl von Klassen, die eine einfache, maschinenunabhängige Programmierung von grafischen Benutzeroberflächen ermöglichen. Wenn man den Abstract Window Toolkit (AWT) importiert, stehen diese Klassen zum Aufruf zur Verfügung.

Wir schreiben nun eine neue Klasse `ZaehlerFrame` mit einer grafischen Oberfläche, die die zeilenorientierte Klasse `ZaehlerTest` ersetzt.

Beispiel: ZaehlerFrame (1)

```
// ZaehlerFrame.java
import java.awt.*;
import java.awt.event.*;
class ZaehlerFrame extends Frame {
  private Button plus, minus;
  private TextField stand;
  private Zaehler z;
  ZaehlerFrame(String s) {
    super(s);
    z = new Zaehler();
    setLayout(new GridLayout(2, 2));
```

Beispiel: ZaehlerFrame (2)

```
    add(new Label("Zählerstand: ", Label.RIGHT));
    add(stand = new TextField(10));
    stand.setText(String.valueOf(z.wert()));
    stand.setEditable(false);
    add(plus = new Button("Inkrementiere"));
    add(minus = new Button("Dekrementiere"));
    ButtonListener lis = new ButtonListener();
    plus.addActionListener(lis);
    minus.addActionListener(lis);
    pack();
    setVisible(true);
}
```

Beispiel: ZaehlerFrame (3)

```
public static void main(String[] args) {
  new ZaehlerFrame("Zähler-Test");
}
class ButtonListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    if (e.getActionCommand().equals("Inkrementiere"))
      z.inkrementiere();
    else
      z.dekrementiere();
    stand.setText(String.valueOf(z.wert()));
    }
  }
}
```

Beispiel: ZaehlerApplet (1)

Ein Applet ist ein Java-Programm, das nicht eigenständig, sondern nur von einem Web-Browser aus ausgeführt werden kann. Der Browser lädt den Bytecode dynamisch vom Server und bringt ihn zur Ausführung.

```
// ZaehlerApplet.java
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class ZaehlerApplet extends Applet {
  private Button plus, minus;
  private TextField stand;
  private Zaehler z;
```

Beispiel: ZaehlerApplet (2)

```
  public void init() {
    z = new Zaehler();
    setLayout(new GridLayout(2, 2));
.............Rest wie ZaehlerFrame
}
  class ButtonListener implements ActionListener {
    ....wie in ZaehlerFrame
  }
}
```

Einbettung des ZaehlerApplets in eine html-Seite

```
<h1>Das Zähler-Applet</h1>
<hr>
<center>
<applet code = ZaehlerApplet width = 250 height = 70>
</applet>
</center>
<p>
Hier steht der <a href= “ZaehlerApplet.java“>Java-
Code.</a>
```

2.3 Lexikalische Konventionen

Syntax und Semantik

Zur maschinellen Verarbeitung eines Algorithmus muss dieser in einer wohl definierten Sprache (frei von Mehrdeutigkeiten und Ungenauigkeiten) ausgedrückt werden. Eine solche Sprache heißt **Programmiersprache**.

Zur Ausführung eines Programms, das in einer Programmiersprache vorliegt, muss der Computer in der Lage sein,

1. die Symbole, in denen jeder Algorithmus-Schritt ausgedrückt ist, zu verstehen (**Syntax**),
2. dem Algorithmus-Schritt in Form von auszuführenden Maschinenbefehlen eine Bedeutung zuordnen (**Semantik**),
3. die entsprechenden Operationen auszuführen.

Syntax-Diagramme

Motivation

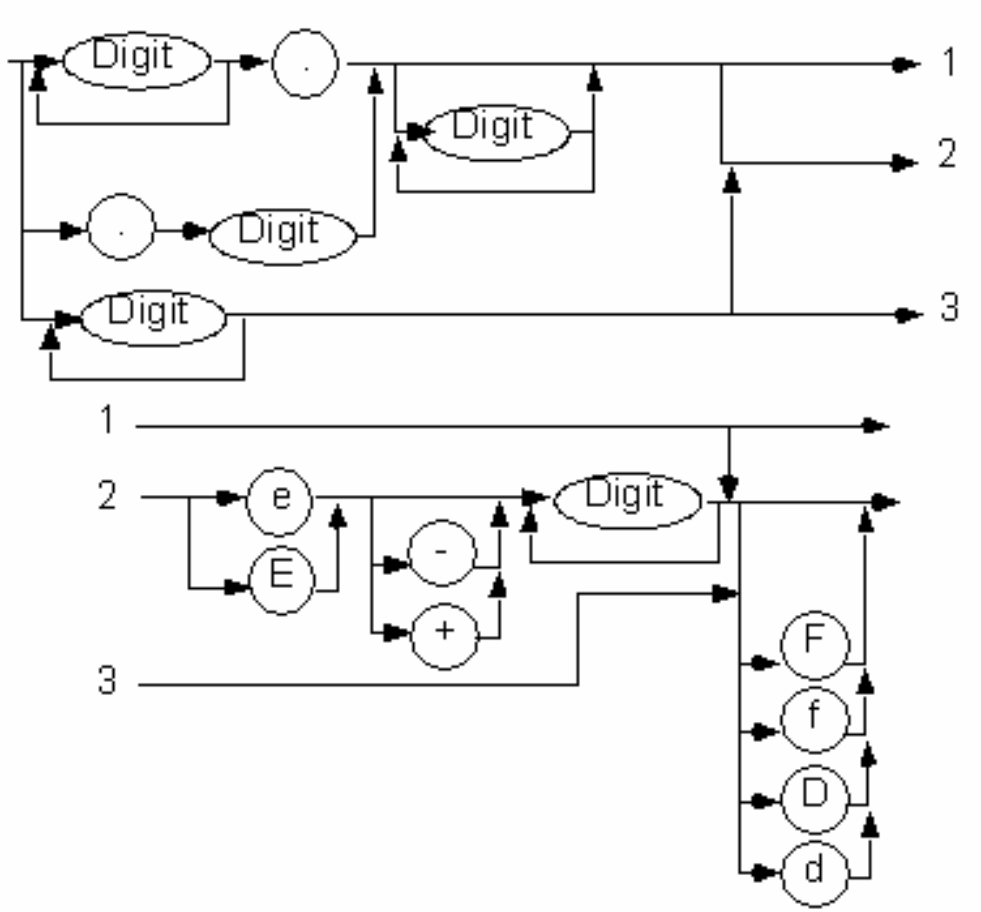
Eine formal definierte Syntax stellt eindeutig klar, welche Ausdrücke einer Sprache gültig sind und welche nicht. Wir führen dazu **Syntax-Diagramme** ein. Sie sind bestehen aus

- Terminalsymbolen (Kreisen),
- Nichtterminalsymbolen (Quadraten) und
- gerichteten Kanten (Pfeilen), die zulässige Wege durch das Diagramm darstellen.

Wir können den gesamten Sprachumfang unserer Programmiersprache durch solche Syntax-Diagramme definieren. *Ein gültiger Ausdruck liegt dann und nur dann vor, wenn er durch einen Pfad durch die Diagramme generiert werden kann.*

Beispiel für ein Syntax-Diagramm

FloatingPointLiteral

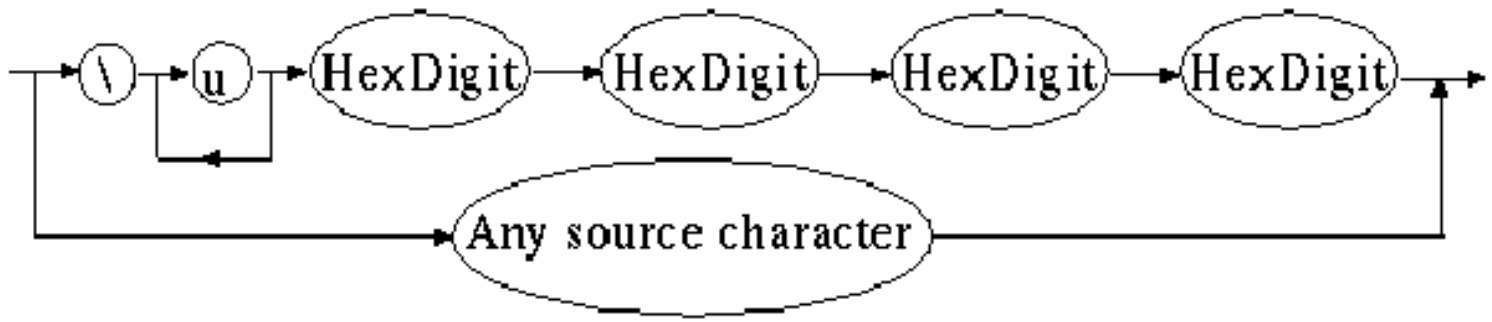


Unicode (1)

Java-Übersetzungseinheiten werden im **Unicode** geschrieben. Dies ist ein Zwei-Byte-Code, mit dem man viele länderspezifische Zeichen, z. B. deutsche Umlaute, französische Akzente oder griechische Buchstaben darstellen kann. Die aktuelle Norm findet man in: The Unicode Standard, Version 2.0, The Unicode Consortium, Addison-Wesley, 1996. Da es derzeit nur wenige Dateisysteme, Editoren usw. gibt, die Unicode verarbeiten, sind „Unicode-Escapes“ definiert, mit denen man Unicode-Zeichen im ASCII-Code darstellen kann.

Unicode (2)

EscapedSourceCharacter



HexDigit (hexadezimale Ziffer) :
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F (Zahlensystem zur Basis 16)

source character :
a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z

Unicode-Beispiele

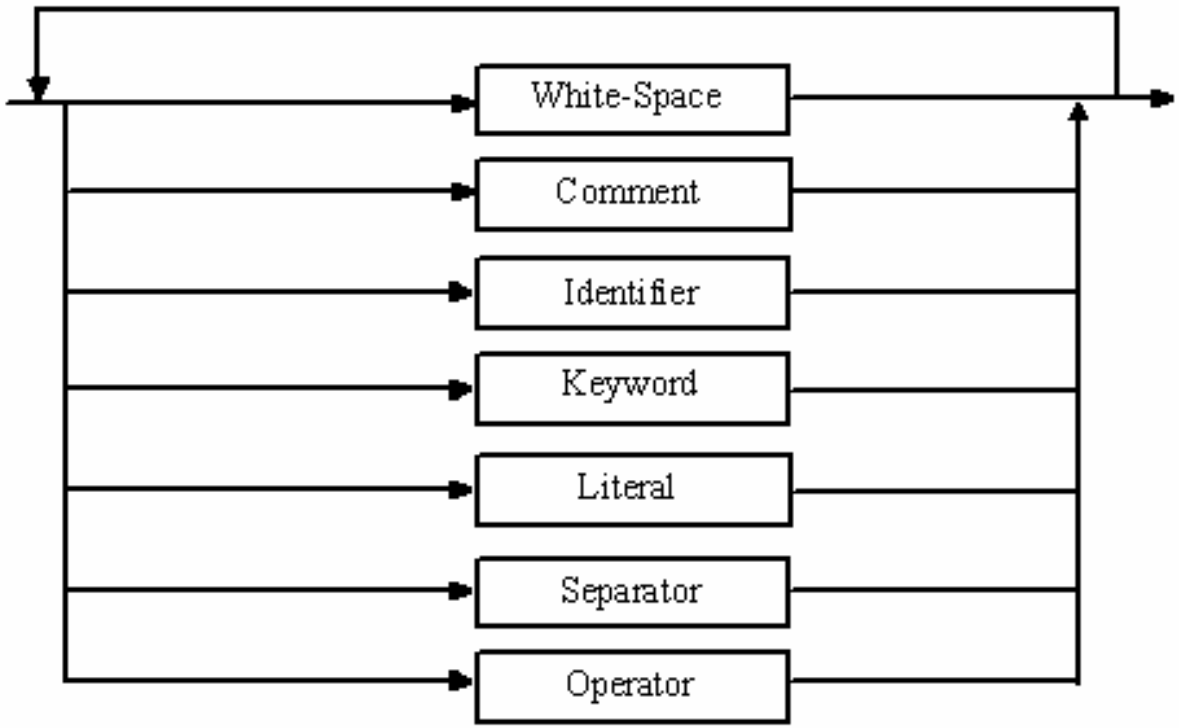
• \u7933	korrekt
• \uFFA3	korrekt
• \uuuuu3059	korrekt
• \uuuuuuuuuuu0000	korrekt
• \uu89232	zu viele Hex-Ziffern
• \7230	u fehlt
• \u89W3	W ist keine Hex-Ziffer

Syntaktische Basiselemente (Token)

Die kleinsten Einheiten, aus denen sich eine Java-Übersetzungseinheit zusammensetzt, nennt man **lexikalische Elemente** oder **Token**. In Java gibt es sieben Kategorien von Token:

- White Space
- Kommentar
- Bezeichner (*identifier*)
- Schlüsselwort (*keyword*)
- Literal
- Interpunktionszeichen (*separator*) und
- Operator.

Syntax-Diagramm für Token



Eindeutige Zerlegung des Quellcodes in Token

Bei der Analyse einer Übersetzungseinheit werden vom Compiler immer die größtmöglichen lexikalischen Elemente (Token) gebildet: Wenn ein kürzeres Token in einem längeren enthalten ist, wird das längere ausgewertet.

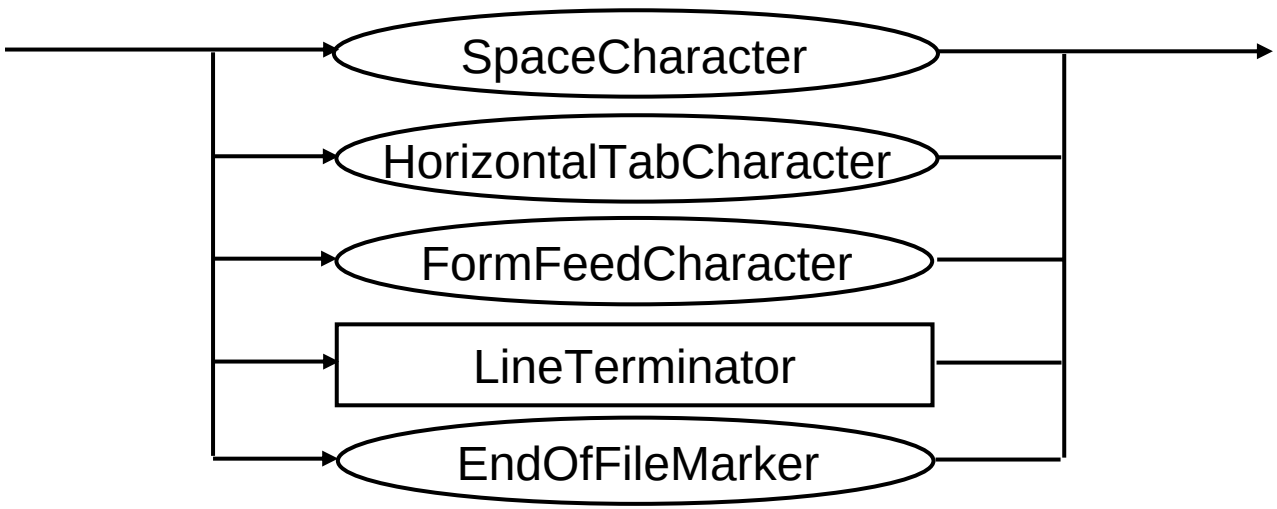
Beispiel

`++wert` in der Klasse `Zaehler` wird ausgewertet als Inkrement-Operator, gefolgt von einem Bezeichner ("identifier") und nicht als zwei positive Vorzeichen vor einem Bezeichner.

White Space

Ein “white space“ dient zur Separation von Token. Er ist insofern semantisch wichtig und darf nicht weggelassen werden. So bedeutet zum Beispiel “+ +wert“ etwas anderes als „++wert“. Mehrere “white spaces“ hintereinander haben dieselbe Bedeutung wie ein einzelner “white space“. Wir verwenden “white spaces“, um unsere Programme lesbarer zu machen.

Syntax-Diagramm



Kommentar (1)

Es gibt drei Arten von Kommentaren:

Die **traditionellen** Kommentare sind in /* und */ eingeschlossen.

Zeilenkommentare beginnen mit // und erstrecken sich bis zum Ende der Zeile.

Dokumentationskommentare sind in /** und */ eingeschlossen.

Es gilt:

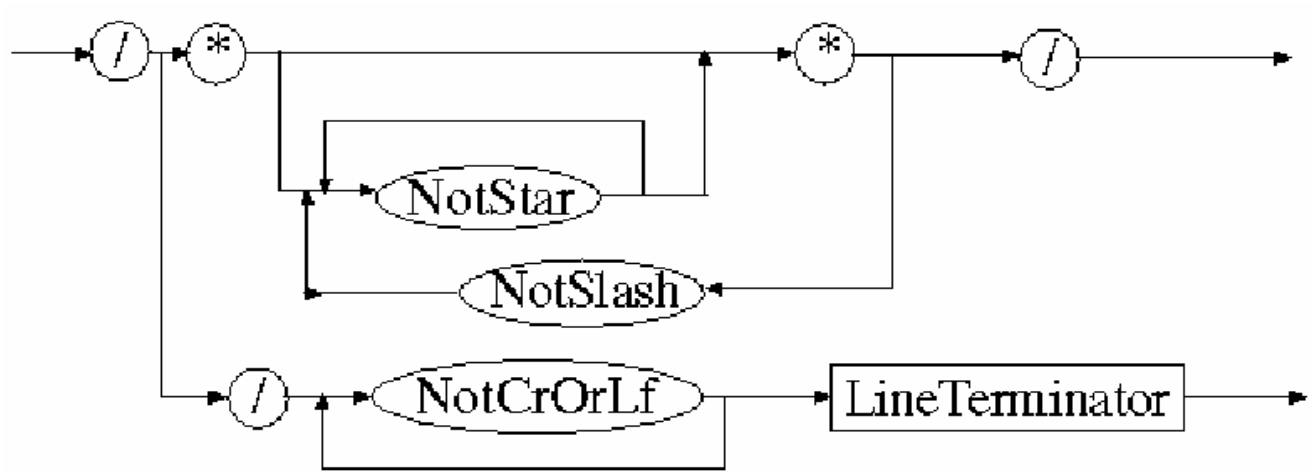
- Kommentare können nicht geschachtelt werden.
- /* und */ haben keine besondere Bedeutung in Zeilenkommentaren.
- // hat keine besondere Bedeutung in Kommentaren, die mit /** oder /* beginnen.

Kommentar (2)

Dokumentationskommentare werden wirksam, wenn man die entsprechende Java-Datei mit dem JDK enthaltenen Programm **javadoc** verarbeitet. Wenn wir einfach **javadoc** eingeben, erhalten wir die Aufrufoptionen.

javadoc legt eine HTML-Datei an, die zu Dokumentationszwecken zusammen mit den **class-Dateien** ausgeliefert werden kann.

Syntax-Diagramm (ohne Dokumentationskommentare)



Bezeichner (identifier)

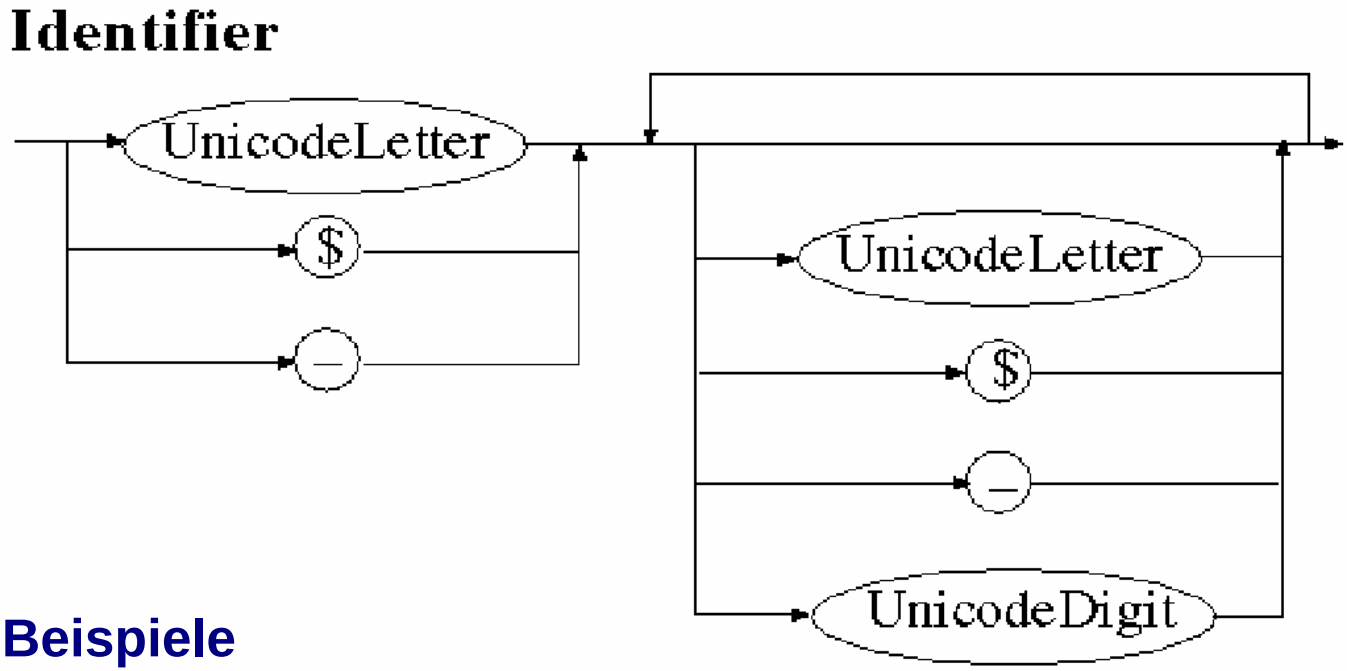
Bezeichner sind Namen, die wir für die von uns deklarierten Klassen, Methoden, Variablen, Interfaces und Pakete wählen. Java-Bezeichner bestehen aus beliebig langen Folgen von Unicode-Buchstaben und -Ziffern. Sie müssen mit einem Buchstaben beginnen. “`A-Z`“, “`a-z`“, “`_`“ und “`$`“ sind Unicode-Buchstaben, “`0-9`“ sind Unicode-Ziffern.

Die Java-Schlüsselwörter sowie die Literale `true, false` und `null` sind nicht als Bezeichner einsetzbar.

In ASCII-basierten Systemen können die “Unicode-Escapes“ `\u0000-\uffff` verwendet werden.

In unserer ersten Übersetzungseinheit `Zaehler.java` haben wir die Bezeichner `Zaehler, wert, i, inkrementiere` und `dekrementiere` benutzt.

Syntax-Diagramm für Bezeichner



Beispiele

Art des Bezeichners	Name
class	Button, BufferedReader
interface	Runnable, DataInput
method	start(), getAbsolutePath()
variable	offset, anotherString

Beispiele für Bezeichner

HowToProgramJava	korrekt
_HowToProgramJava	korrekt
How_To_Program_Java	korrekt
HowTo$Program$Java	korrekt
How2ProgramJava	korrekt
2ProgramJava	falsch: Zahl am Anfang
$$$$$$$_____	korrekt
$3473109743	korrekt
\u4839	korrekt (Unicode-Bezeichner)

Schlüsselwörter

Die folgende Tabelle enthält die **Java-Schlüsselwörter**. Diese sind nicht als Bezeichner einsetzbar.

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

Neben den Schlüsselwörtern sind auch die speziellen Literale `true`, `false` und `null` nicht als Bezeichner einsetzbar.

2.4 Typen und Werte

Jedem von uns deklarierten Bezeichner **muss genau ein Typ** zugeordnet sein, der festlegt, welche Operationen für den Bezeichner definiert sind, wieviel Speicherplatz zu reservieren ist und welche Werte dem jeweiligen Speicherinhalt entsprechen. Mit jedem Typ ist auch ein **Wertebereich** festgelegt. Das ist die Menge der Werte, die eine Variable dieses Typs annehmen kann,.

Java ist eine Sprache mit **strenger Typprüfung**. Jede Variable und jeder Ausdruck hat einen Typ, der beim Übersetzten bekannt ist. Zusätzlich werden von uns vorgenommene **Typkonversionen** statisch (beim Übersetzen) und, sofern dies erst zur Laufzeit möglich ist, dynamisch geprüft.

Datentypen

In Java werden zwei grundlegende Typen unterschieden: **elementare Typen** und **Referenztypen**.

- Die elementaren Typen bestehen aus dem logischen Typ `boolean` und den **numerischen Typen**. Numerische Typen sind die **ganzzahligen Typen** `byte, short, int, long` und `char` sowie die Gleitpunkttypen `float` und `double`.
- Referenztypen sind **Klassen**, **Interfaces** und **Felder**.

Klasse und Objekt als Typ und Instanz (1)

Ein **Objekt** ist in Java eine Ausprägung (Instanz) einer **Klasse**. Man kann also eine Klasse als einen Typ auffassen. Für einen solchen Typ können zur Laufzeit Instanzen (= Objekte) kreiert und wieder zerstört werden.

Alle Objekte sind implizit Instanzen der Klasse `Object` und verfügen damit über deren Methoden; u. a.

```
public String toString() { ..... }

public boolean equals (Object obj) { ..... }

protected Object clone()
          throws CloneNotSupportedException { ..... }

protected void finalize() throws Throwable { ..... }
```

Klasse und Objekt als Typ und Instanz (2)

Mittels `toString` ist es möglich, jedes Objekt in eine für das Objekt charakteristische Zeichenkette umzuwandeln (und diese z. B. auszugeben). Mit `equals` können wir vergleichen, ob zwei Objekte identisch sind. `clone` erzeugt eine Kopie eines Objekts, und `finalize` ist eine Methode, die aufgerufen wird, bevor ein Objekt wieder zerstört wird. Objekte werden grundsätzlich erst zur Laufzeit dynamisch erzeugt. Werte eines Referenztyps sind Referenzen (Zeiger) auf Objekte.

2.5 Konstanten und Variable

Wie die meisten anderen Programmiersprachen unterscheidet Java Konstanten und Variable.

Eine **Konstante** erhält ihren Wert bereits zum Zeitpunkt der Niederschrift des Programms zugewiesen; dieser Wert kann sich zur Laufzeit des Programms niemals ändern. Konstante können symbolische Namen haben, zum Beispiel `pi`. Oder sie können unmittelbar als Wert aufgeschrieben werden, dann sind es gerade die bereits eingeführten **Literale**.

Eine **Variable** ist ein Platzhalter für einen Wert. Der aktuelle Wert kann sich zur Laufzeit des Programms (auch mehrfach) ändern.

Konstanten

Syntax für Konstante

```
final Typ VariablenName = Wert
```

oder

```
final Typ VariablenName;
```

Beispiel

```
final int x = 5
```

Wichtig:

Konstanten sind Platzhalter für Werte, während Variablen Platzhalter für Adressen sind. Daher können Konstanten ihre Werte nicht verändern.

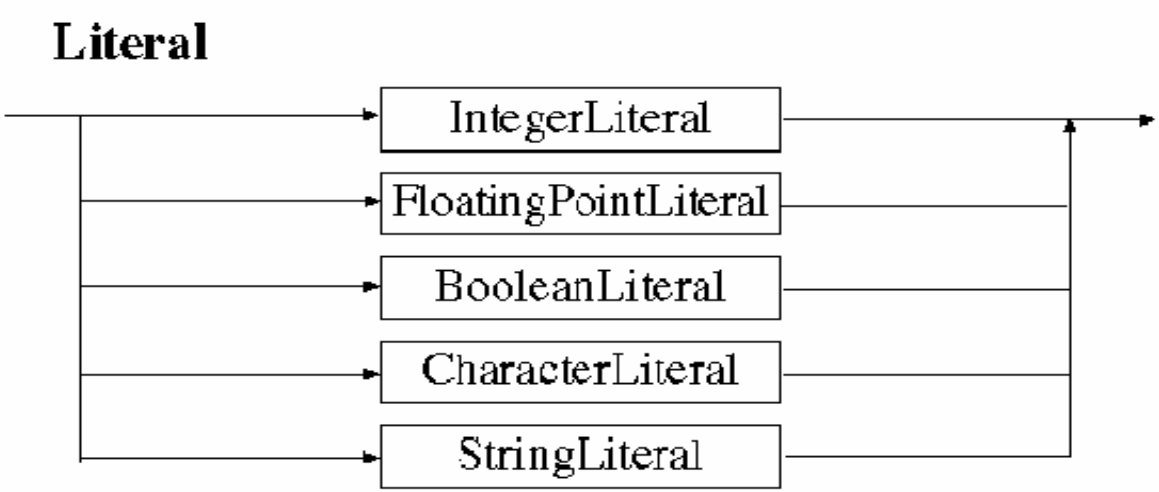
Hexadezimale Konstanten

In Hexadezimaldarstellung beginnt eine ganzzahlige Konstante mit `0x` oder `0X` und besteht aus mindestens einer weiteren Hexadezimalziffer (`0-9`, `a-f` oder `A-F`).

Literal

In einer Programmiersprache bezeichnen man als **Literal** das, was man in mathematischen Formeln als Konstante bezeichnet, also Zahlen, Boolesche Werte und Buchstabenketten, die während der Ausführungszeit des Programms ihren Wert nicht ändern können.

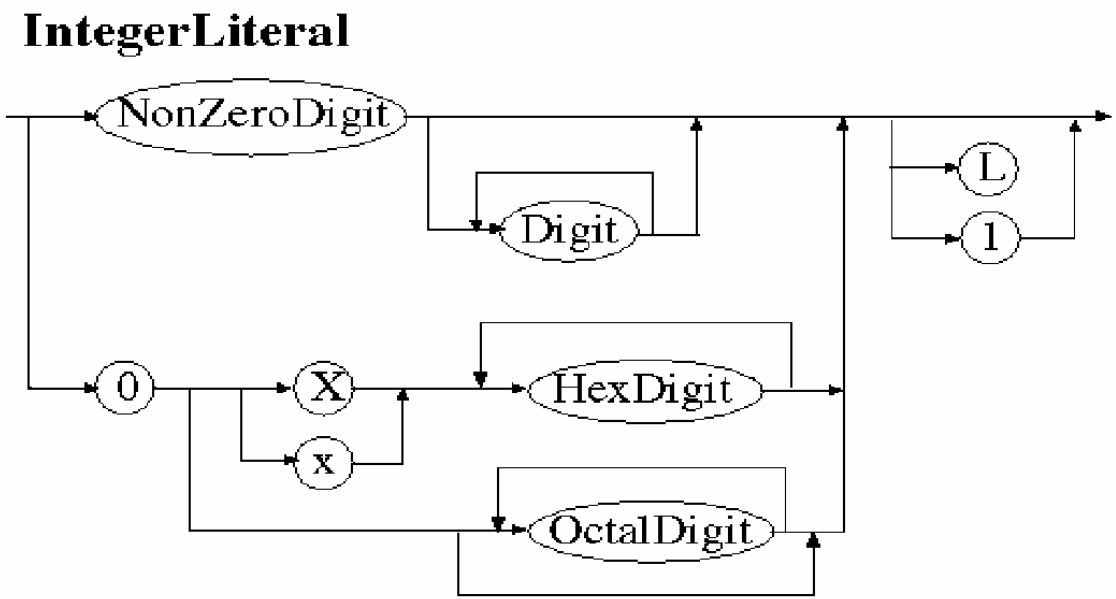
Syntax-Diagramm



Ganzzahl (1)

Ganzzahlen (integers) entsprechen den ganzen Zahlen aus der Mathematik.

Syntax-Diagramm



Ganzzahlen sind im Normalfall int, nicht long, es sei denn, sie enden mit l (kleinem L) oder L. Eine führende 0 (Null) besagt, dass entweder eine Oktalzahl oder eine Hexadezimalzahl folgt.

Ganzzahl (2)

Es gibt Ganzzahlen verschiedener Größe, wie folgt:

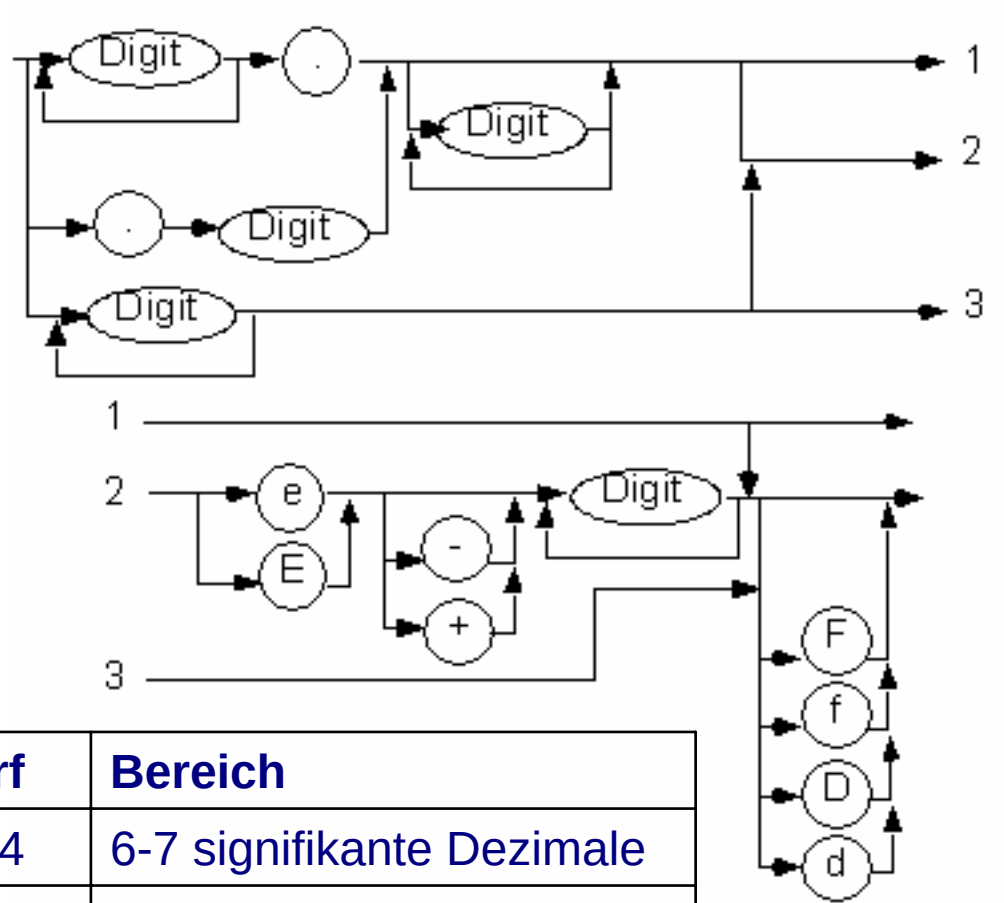
Type	Speicherbedarf	Bereich
byte	8-bit, Vorzeichen, 2er-Komplement	-128 to 127
short	16-bit, Vorzeichen, 2er-Komplement	-32768 to 32767
int	32-bit, Vorzeichen, 2er-Komplement	-2**31 to 2**31-1
long	64-bit, Vorzeichen, 2er-Komplement	-2**63 to 2**63-1
char	16-bit, kein Vorzeichen, Unicode	\u0000 to \uFFFF

Beispiele für Ganzzahlen

76438748	korrekt
876387432L	korrekt
843275829l	korrekt
198439ll	zwei l, falsch
98432750209	Zahl zu groß, falsch
8437658lU	U falsch
984357A439875	A falsch
439827O4379832	O falsch

Gleitkommazahl (floating point number)

Syntax-Diagramm



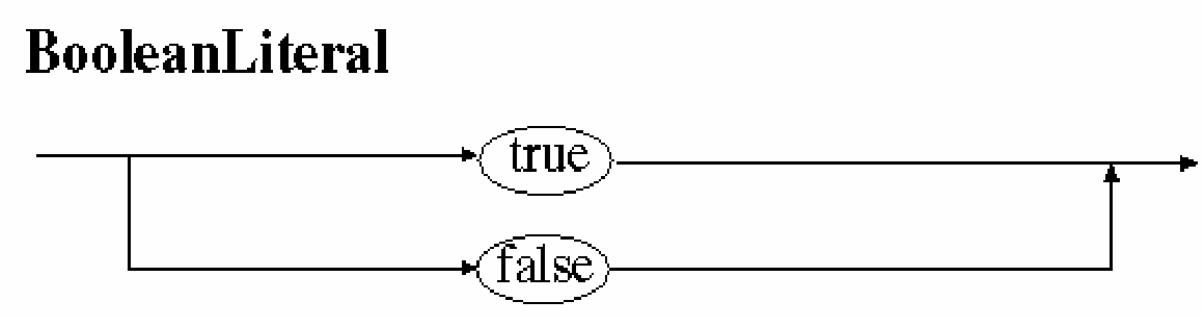
Type	Speicherbedarf	Bereich
float	32-bit, IEEE 754	6-7 signifikante Dezimale
double	64-bit, IEEE 754	15 signifikante Dezimale

Beispiele für Gleitkommazahlen

32876.34857	korrekt
.43573495	korrekt
984375e-43	korrekt
439857E+93	korrekt
84e8	korrekt
439578f	korrekt
349857943D	korrekt
49e-45.3847	Dezimalpunkt an falscher Position
.e12	falsch; nach dem Dezimalpunkt muss mindestens eine Ziffer folgen
23.e7	korrekt

Literale für Wahrheitswerte (boolean)

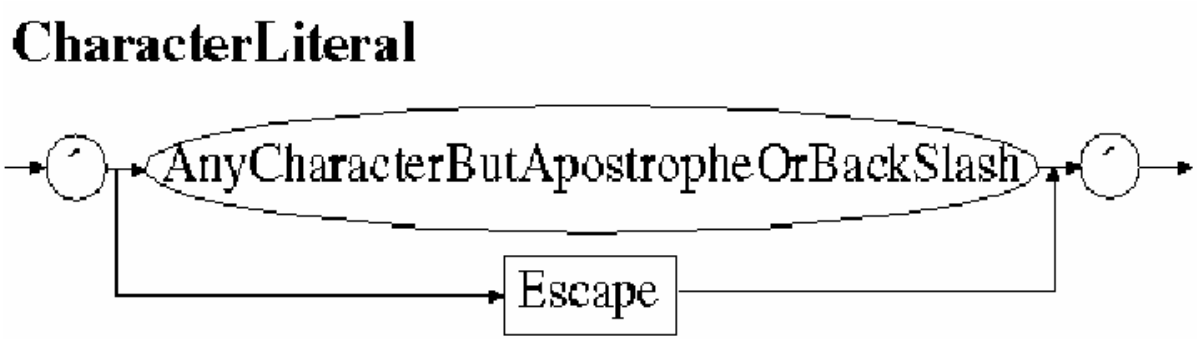
Syntax-Diagramm



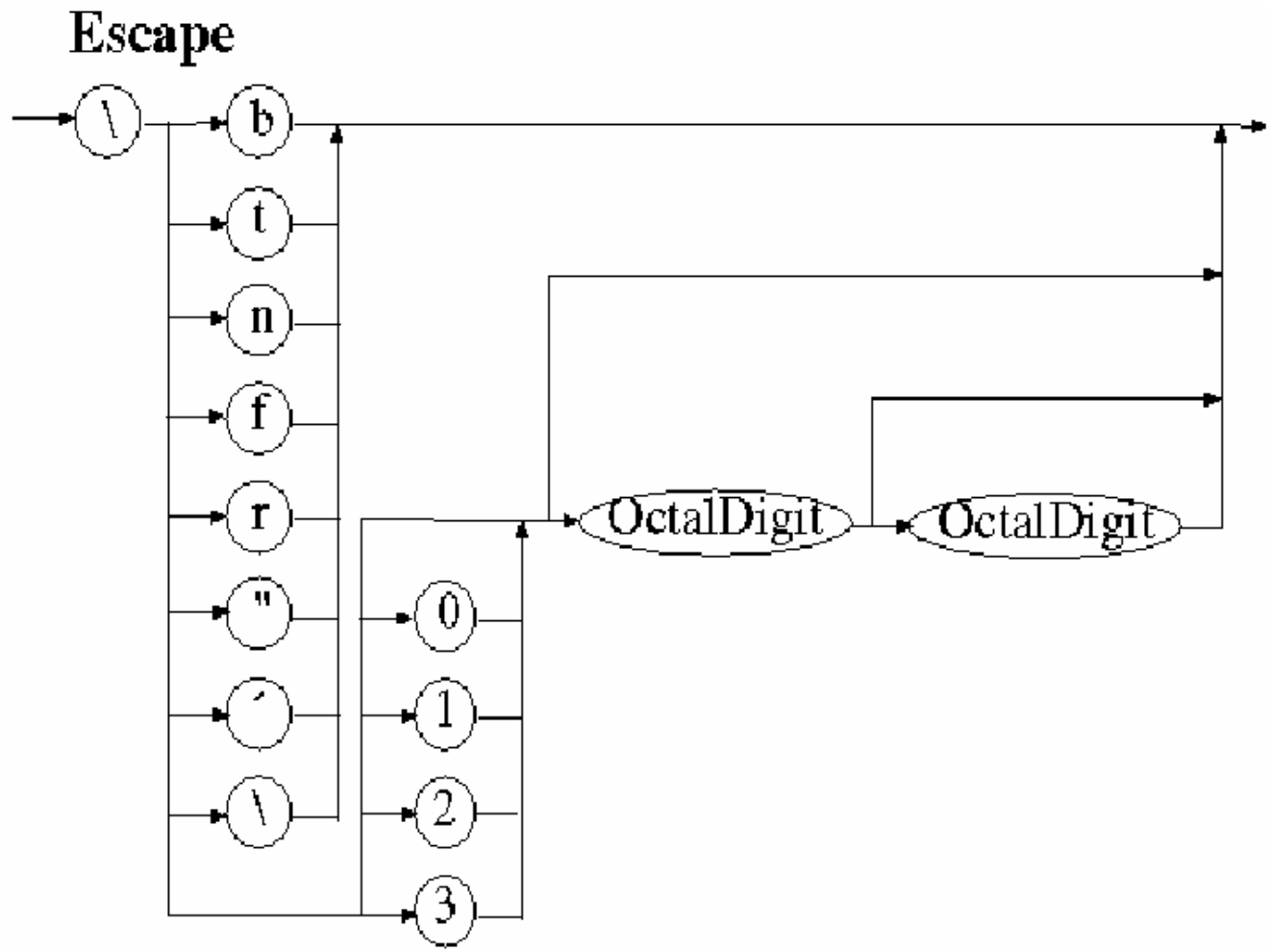
Buchstaben-Literal (character literal)

Buchstaben-Literale werden in Apostrophe eingeschlossen. Mit Escape-Sequenzen stellt man Buchstaben-Literale dar, die nicht als druckbare ASCII-Zeichen aufgeschrieben werden können.

Syntax-Diagramm



Syntax-Diagramm für Escape



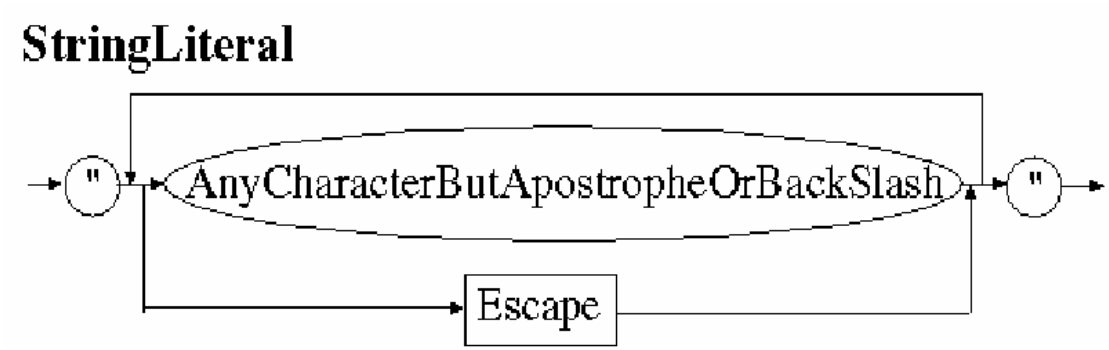
Escape-Sequenzen

Gebräuchliche Escape-Sequenzen sind:

Escape-sequenz	Unicode-Äquivalent	Bedeutung
\b	\u0008	Backspace
\t	\u0009	Horizontal tab
\n	\u000a	Linefeed
\f	\u000c	Form feed
\r	\u000d	Carriage return
\"	\u0022	Double quote
\'	\u0027	Single quote
\\	\u005c	Backslash
\xxx	\u0000 to \u00ff	Buchstabe, der dem oktalen Wert entspricht

Buchstabenfolge (string)

Syntax-Diagramm

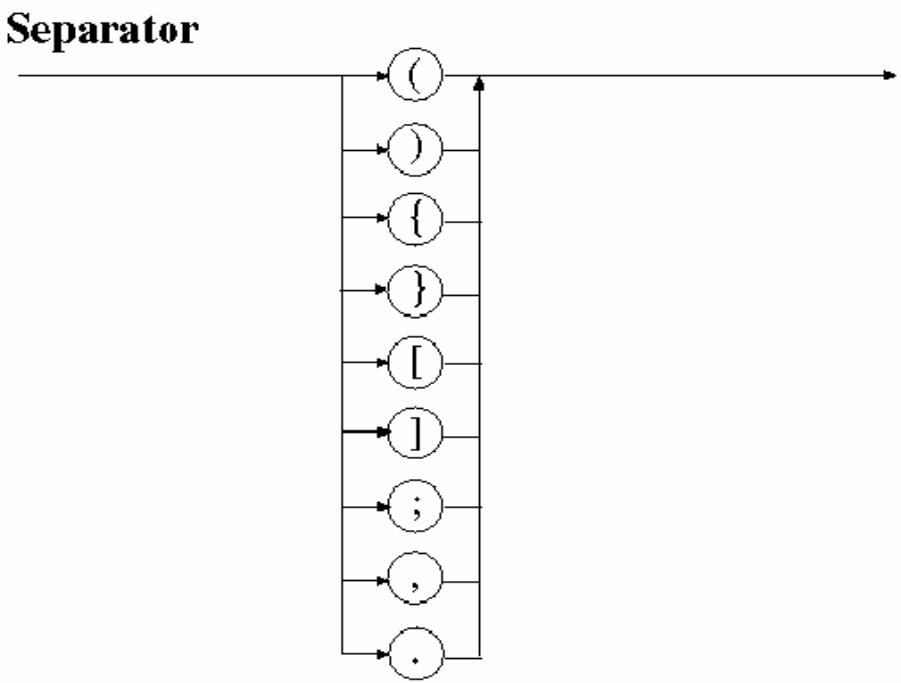


Interpunktionszeichen

Die folgenden neun Zeichen dienen in Java als Interpunktionszeichen:

( ) { } [ ] ; , .

Syntax-Diagramm:



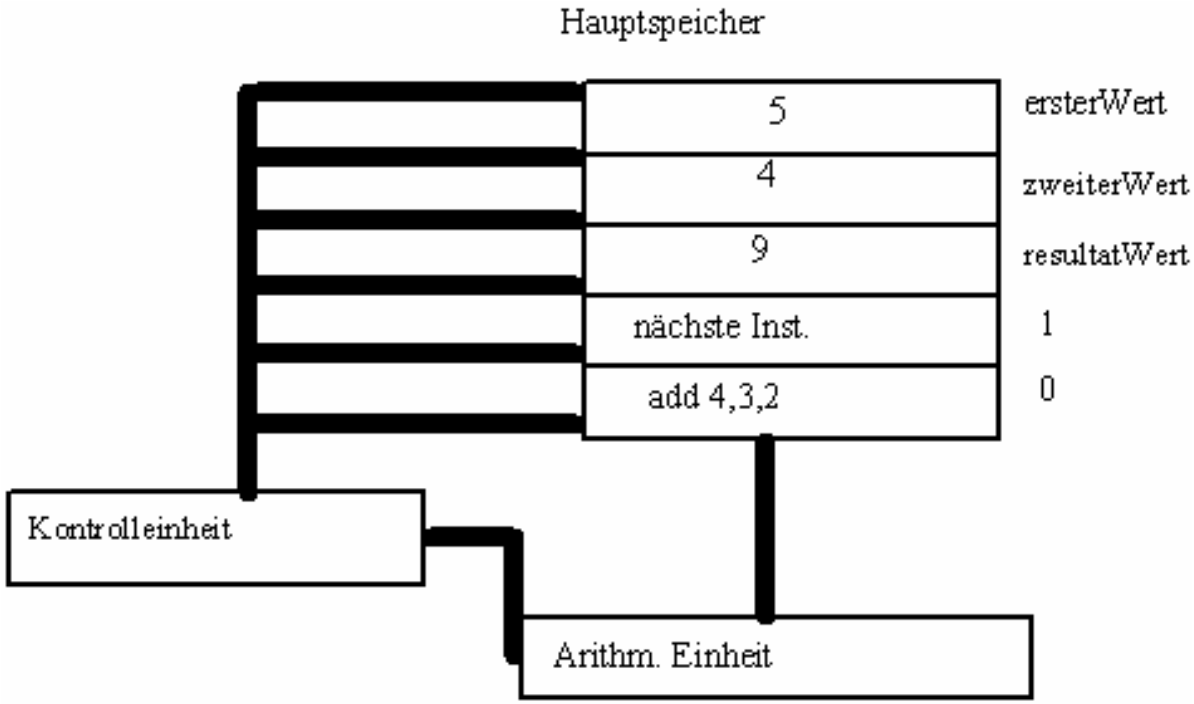
Die Nullreferenz

Die **Nullreferenz**, die anzeigt, dass eine Variable aktuell kein Objekt referenziert, wird durch das Literal `null` repräsentiert.

Variable (1)

Variable sind Platzhalter für Werte. Sie sind können auch als Namen für Speicheradressen aufgefasst werden, denen ein fester Typ zugewiesen ist.

Variable (2)



`ersterWert` **ist 5 und repräsentiert Speicheradresse 4.**

`zweiterWert` **ist 4 und repräsentiert Speicheradresse 3.**

`resultatWert` **ist 9 und repräsentiert Speicheradresse 2.**

Variablennamen

In den meisten Fällen sind **Variablennamen** Bezeichner (identifier). Als qualifizierte Namen für Variable können aber auch beliebig viele mit "." aneinander gereihte Bezeichner vorkommen, wenn die höheren Ebenen zuvor definiert und instanziiert worden sind, zum Beispiel

std.a

y.out

employee.address.zipcode

Die genaue Bedeutung dieser Bezeichnerkombinationen wird im objekt-orientierten Teil von Java erläutert.

Beispiele für Variable

In unserem Beispiel **`ZaehlerTest`** haben wir sechs Variablen deklariert: **`args, z, in, out`** und **`ioe`** sind Variablen eines Referenztyps. Genauer ist **`args`** Variable des Feldtyps **`String[]`**, **`z`** Variable des Klassentyps **`Zaehler`**, **`in`** Variable des Klassentyps **`BufferedReader`**, **`out`** Variable des Klassentyps **`PrintWriter`** und **`ioe`** Variable des Klassentyps **`IOException`**. Die verbleibende Variable **`akt`** hat den elementaren ganzzahligen Typ **`char`**.

Deklaration von Variablen

Bevor Variable im Programm verwendet werden, müssen sie **deklariert** werden.

```
Typ VariablenName;
```

oder

```
Typ VariablenName = Wert;
```

Beispiele:

```
int sum = 5;
double value;
```

Nutze das Komma für Mehrfachdeklarationen:

```
int sum1, sum2 = 5, sum6;
```

Typen: char, byte, short, int, long, float, double

Initialisierung von Variablen

Unter der ***Initialisierung*** einer Variablen oder symbolischen Konstanten versteht man das Kopieren eines Werts in ihren Speicherplatz direkt im Zusammenhang mit dessen Reservierung.

Beim ***Zuweisen*** wird dagegen ein Wert in eine anderweitig erzeugte Variable kopiert.

An mache Variablen ist keine Zuweisung möglich: Klassenvariable, Instanzvariable und lokale Variable, die wir als **final** spezifizieren, müssen wir bei ihrer Deklaration initialisieren oder ihnen *einmal* einen Wert zuweisen.

Auch an Methoden- und Konstruktorparameter sowie den Parameter eines "Exception Handlers" können wir nichts zuweisen – hier werden die Initialisierungen mit den Aufrufargumenten bzw. dem ausgeworfenen Ausnahmeobjekt von der VM vorgenommen.

Unterschied zwischen Initialisierung und Zuweisung

Im Unterschied zu anderen Programmiersprachen, die für Initialisierung bzw. Zuweisung verschiedene Operatoren bzw. Methoden benutzen, verfährt Java beim Initialisieren einer Variablen genau wie bei Zuweisungen: Wert und Typ des Initialisierers werden ermittelt, falls nötig (und zuweisungskompatibel) in den Typ der Variablen umgewandelt, das Resultat wird dann in deren Speicherplatz kopiert. In der Java-Literatur wird wegen dieser Analogie oft nicht präzise formuliert und von Zuweisungen gesprochen, auch wenn es sich um Initialisierungen handelt.

2.6 Typumwandlung, Ausdrücke und Operatoren

Operatoren führen Operationen auf ihren Operanden (Argumenten) aus.

Operatoren

NonAssignmentOperator

+	-	++	--
%	*	/	
<<	>>	>>>	
~	|	&	^
&&	||	!	
!=	==		
<	>=		
>	<=		
?	:		

AssignmentOperator

=	-=	*=
/=	|=	&=
^=	+=	%=
<<=	>>=	>>>= (mit und ohne sign-extension)

Additive und multiplikative Operatoren

Operator	Bedeutung	algebraisch	Java
+	Addition	f+7	f+7
-	Subtraktion	p-c	p-c
*	Multiplikation	bm	b*m
/	Division	x/y	x/y
%	modulo	r mod s	r % s

Operatoren zum Inkrementieren und Dekrementieren

++	Inkrement um 1 (als prefix oder postfix)
--	Dekrement um 1 (als prefix oder postfix)

Beispiele

```
a = 7;      // a wird 7 zugewiesen
a++;        // a wird auf 8 inkrementiert, wie a = a + 1;
            // inkrementiert wird nach der Auswertung der Variablen!
a--;        // a ist wieder 7, wie a = a - 1;
++a;        // a ist wieder 8, wie a = a + 1;
            // VORSICHT: inkrementiert wird vor der Auswertung der
               Variablen!
```

Bemerkung :

Inkrement- und Dekrement-Operatoren können Code schwer lesbar machen.

Am besten nur als Einzelanweisung benutzen, nicht in Ausdrücken.

Logische Operatoren

&&	logisches UND
||	logisches ODER
!	Logisches NICHT

Beispiele

a && b	(entspricht “a AND b“)
a || b	(entspricht “a OR b“)
!a	(entspricht “NOT a“)

Zuweisungsoperatoren

=	zuweisen	a = 2;	
+=	zuweisen und addieren	a +=2;	a = a + 2;
-=	zuweisen und subtrahieren	a -= 2;	a = a - 2;
*=	zuweisen und multiplizieren	a *= 2;	a = a * 2;
/=	zuweisen und dividieren	a /= 2;	a = a / 2;
%=	zuweisen und modulo	a %= 2;	a = a % 2;
…			

Bemerkung

Man benutze die kombinierten Zuweisungsoperatoren mit Vorsicht! Sie können den Code schwer lesbar machen.

Vergleichsoperatoren (relationale Operatoren)

Die Vergleichsoperatoren haben zwei Operanden. Ihre Auswertung ergibt einen Wahrheitswert (Booleschen Wert).

Java	Beispiel	Bedeutung
==	x == y	x ist gleich y
!=	x != y	x ist ungleich y
>	x > y	x ist größer als y
<	x < y	x ist kleiner als y
>=	x >= y	x ist größer oder gleich y
<=	x <= y	x ist kleiner oder gleich y

Achtung! Häufig werden bei der Programmierung ‘=‘ und ‘==‘ verwechselt!

Typumwandlung (1)

Da Java streng typisiert ist, kann man einer Variablen nur einen Wert vom gleichen Typ zuweisen, oder man kann mit einem Vergleichsoperator nur Ausdrücke gleichen Typs vergleichen. Gäbe es keine Typumwandlung, so wäre man in der Programmierung sehr inflexibel.

Man unterscheidet die **implizite** und die **explizite** Typumwandlung.

Bei Typumwandlungen kann es zu **Genauigkeitsverlusten** und zu **Verlusten im darstellbaren Wertebereich** (Größenverlusten) kommen.

Typumwandlung (2)

Wenn in einem Ausdruck in Java zwei Typen nicht zueinander passen, wird entweder eine

- implizierte Typumwandlung (type casting) durchgeführt

oder ein

- Syntax-Fehler gemeldet.

Es ist also in Java nicht möglich, einen Speicherplatz als Bitfolge aufzufassen und im Programm mal so und mal anders zu interpretieren!

Beispiele

```
int i = 5;
int j = 4L;          // impliziter Type-Cast von long zu int

long k;
k = i;               // impliziter Type-Cast von int zu long
```

Implizite Typumwandlung

Implizite Konversionen werden von Java in vier Situationen vorgenommen, auf die später noch genauer eingegangen wird:

- bei Zuweisungen und Initialisierungen (die Java als semantisch äquivalent betrachtet),
- bei Methoden- und Konstruktoraufrufen. Hier wird ein Argumentwert an einen Parameter übergeben.
- bei numerischen Typangleichungen im Zusammenhang mit arithmetischen Operatoren. Hier wird ein Wert in einem umfassenderen Ausdruck ausgewertet.
- bei **String**-Konversionen – hier wird ein Wert durch den Operator **+** mit einem **String**-Objekt verknüpft.

Beispiel für Typumwandlung

```
// KonversionsKontexte.java
import java.io.*;
class KonversionsKontexte {
  static PrintWriter out = new PrintWriter(System.out,true);
  static void m(double d) { out.println(”m(double) = “ + d); }
  public static void main(String[] args) {
    long l = 5;
    float x = 1.25f;
    m(x);
    x = x*l;
    out.println(“x = “ + x);
    short s = (short)x;
  }
}
```

Elementare Typvergrößerungen

Die folgenden Typumwandlungen werden als *elementare Typvergrößerungen* bezeichnet. Java nimmt diese Konversionen, falls nötig, bei Zuweisungen, Methoden- und Konstruktoraufrufen und bei der Auswertung von Ausdrücken implizit vor:

```
byte  nach  short, int, long, float oder double
short nach  int, long, float oder double
int   nach  long, float oder double
char  nach  int, long, float oder double
int   nach  long, float oder double
long  nach  float oder double
float nach  double
```

Bei den Umwandlungen ganzzahliger Typen in größere ganzzahlige Typen und bei der Umwandlung von float nach double kann es zu **keinerlei Informationsverlust** bezüglich der konvertierten Werte kommen – die Umwandlungen sind „sicher“.

Elementare Typenverkleinerungen

Die folgenden Typumwandlungen werden als **elementare Typverkleinerungen** bezeichnet:

`byte`	nach	`char`
`short`	nach	`byte oder char`
`char`	nach	`byte oder short`
`int`	nach	`byte, short oder char`
`long`	nach	`byte, short, char oder int`
`float`	nach	`byte, short, char, int, oder long`
`double`	nach	`byte, short, char, int, long, oder float`