

# Programmieren I

## Primitive Datentypen



Heusch 6  
Ratz 4.3, 4.4

Institut für Automation und angewandte Informatik

```
final List<String> allResults = new ArrayList<String>();  
final Map<String, Integer> typeWordResultCount = new HashMap<String, Integer>();  
final Map<String, Integer> typePoints = new HashMap<String, Integer>();  
evaluation.put(type, typePoints);  
  
for (final Sheet sheet : this.sheets) {  
    final String sheetResult = sheet.getPlayerInput(type);  
    if (sheetResult.startsWith(start) && this.isValidWord(sheetResult, type)) {  
        validWordCountForType++;  
        allResults.add(sheetResult);  
    }  
}
```

# Java-Bezeichner

- Für Variablen, Methoden, Klassen und Schnittstellen werden Bezeichner – auch *Identifizierer* (von engl. identifizieren) genannt – vergeben.
- Ein Bezeichner ist eine *Folge von Zeichen*, die fast beliebig lang sein kann.
  - Die Zeichen sind Elemente (Teilmenge) aus dem *gesamten Unicode-Zeichensatz*. Zulässig sind Java-Buchstaben (inkl. Zeichen wie '\_', '\$' und '€') und Java-Ziffern.
  - Jedes Zeichen ist für die Identifikation wichtig. Das heißt, ein Bezeichner, der 100 Zeichen lang ist, muss auch immer mit allen 100 Zeichen korrekt angegeben werden.
- Jeder Java-Bezeichner muss mit einem Unicode-Buchstaben *beginnen*. Unicode-Buchstaben sind z. B. die Buchstaben aus den Bereichen 'a' bis 'z' und 'A' bis 'Z', aber auch '\_' und '\$'.
- Nach dem ersten Zeichen können neben Buchstaben inkl. '\_' und '\$' auch Ziffern 0-9 folgen.
- Es wird zwischen *Groß- und Kleinschreibung unterschieden*.
- Leerzeichen sind innerhalb von Bezeichnern verboten.

# Beispiele für Bezeichner

## ■ Gültige Bezeichner:

- `mami`
- `kulliReimtSichAufUlli`
- `IchMussWilliAnrufen`
- `RAPHAEL_IST_LIEB`

## ■ Ungültige Bezeichner:

- `2und2macht4`
- `hose gewaschen`
- `hurtig!`
- `class`

# Reservierte Schlüsselwörter

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

\*derzeit nicht verwendet



**Heusch 5.2**  
**Ratz 4.1.4**

# Konventionen für Bezeichner

Bezeichner	Konvention	Beispiel
<b>Packages</b>	Kleinschreibung Trennung mit Punkt („.“)	<code>de.dhbw.ka.prog</code> <code>java.util</code>
<b>Klassen, Interfaces</b>	<i>UpperCamelCase</i> ( <i>erster Buchstabe groß</i> ) Klassen: <Nomen> Interfaces: <Adjektiv>	<code>SomeExample</code>  <code>LegalNotice</code> <code>Consumable</code>
<b>Methoden</b>	<i>lowerCamelCase</i> ( <i>erster Buchstabe klein</i> ) <Verb><Nomen>	<code>getInstance</code> <code>performAction</code>
<b>Variablen</b>	<i>lowerCamelCase</i> ( <i>erster Buchstabe klein</i> )	<code>index</code> <code>targetValue</code>
<b>Konstanten</b> (static, final)	UPPER_SNAKE_CASE (Großschreibung Teilworte mit Unterstrich („_“) getrennt)	<code>MAX_VALUE</code> <code>PI</code> <code>CONF_FILE_NAME</code>

# Quelltext dokumentieren

- Mit Kommentaren den Überblick bewahren
  - Kommentare erleichtern es, zu einem späteren Zeitpunkt die Funktionsweise einzelner Teile nachzuvollziehen.
  - Kommentare werden direkt in den Quelltext eingefügt.
  - Kommentare werden vom Compiler überlesen.  
Hierzu müssen sie entsprechend gekennzeichnet werden.
- In Java gibt es zwei grundlegende Kommentartypen:
  - Einzeilige Kommentare `//`
    - Kommentar bis zum Ende der Zeile.
    - Alle Zeichen hinter `//` werden vom Compiler überlesen.
  - Mehrzeilige Kommentare `/*` `*/`
    - Kommentar über mehrere Zeilen.
    - Ab der Zeichenkombination `/*` werden alle Zeichen im Quelltext vom Compiler überlesen, bis die Zeichenkombination `*/` auftritt.

# Beispiel (Kommentare)

```
class Comment {  
    public static void main( String args[] ) {  
        System.out.println( "Hello World" );  
  
    /* zu Testzwecken als Kommentar:  
        System.out.println( "Hello Europa" );  
  
    */  
    }  
}
```

// Definition der Klasse

// Mit diesem Befehl wird  
// ein Text ausgegeben

// Diese Ausgabe ist als  
// Kommentar gekennzeichnet  
// und wird daher nicht  
// ausgeführt

# Anweisungen

- Eine Anweisung (Statement) ist ein vom Programmierer erstellter Befehle zur Lösung einer Aufgabe.
- Ein Programm besteht aus eine Folge von Anweisungen, die in einer bestimmten Reihenfolge ausgeführt werden.
- Syntax für Anweisungen:
  - Eine Anweisung ist eine einfache Anweisung oder ein Anweisungsblock
  - Eine einfache Anweisung wird mit einem Semikolon abgeschlossen
  - Ein Anweisungsblock fasst mehrere Anweisungen in geschweiften Klammern { } zusammen
  - Die Klammern { } müssen immer paarweise auftreten
  - Anweisungsblöcke können geschachtelt werden
  - Auch leere Anweisungen (nur Semikolon) sind erlaubt



# Beispiel (Anweisung)

```
statement;           // Einfache Anweisung

{                   // Beginn eines Anweisungsblocks
    statement1;
    statement2;
    ...             // Weitere Anweisungen
}                   // Ende des Anweisungsblocks
```

# Datentypen

- In den Anweisungen eines Programms wird mit Daten gearbeitet, die sich in ihrer Art unterscheiden:
  - Zahlen (numerische Daten)
  - Zeichen (alphanumerische Daten)
  - Boolesche (logische) Daten (Wahrheitswerte)
- Java legt mit den Datentypen fest, welche Daten jeweils zulässig sind. Sie unterscheiden sich durch:
  - ihren zulässigen Wertetyp und -bereich
  - in der Größe des dafür benötigten Speicherplatzes
- In Java gibt es zwei Arten von Datentypen:
  - Primitive Datentypen (oder einfache Datentypen)
  - Referenzdatentypen (u.a. Klassen)

# Die primitiven Datentypen



**Heusch 6**  
**Ratz 4.3**

## ■ Numerische Datentypen

### ■ Ganzzahl-Datentypen

- Für ganze Zahlen (ohne Nachkommastellen) mit Vorzeichen.
- Vier verschiedene Varianten: `byte`, `short`, `int`, `long`

### ■ Gleitkommazahl-Datentypen

- Für gebrochene Dezimalzahlen mit Vorzeichen.
- Zwei verschiedene Varianten: `float` und `double`
- Nicht jede Zahl darstellbar → Rundungsfehler

## ■ Zeichen-Datentyp

- Zur Darstellung alphanumerischer Zeichen wird der Datentyp `char` verwendet.
- Er enthält ein beliebiges UniCode-Zeichen.

## ■ Boolescher (logischer) Datentyp

- Zur Repräsentation von Wahrheitswerten wird der Datentyp `boolean` verwendet.
- Werte `true` und `false` (boolesche Literale)

# Interne Darstellung von Zeichen (1)

- Es gibt zwei Gruppen von Zeichen: *druckbare Zeichen* (darstellbare Zeichen) und *Steuerzeichen*.
  - In die erste Gruppe fallen Groß- und Kleinbuchstaben, Ziffern und Sonderzeichen (!, :, +, etc.).
  - Zur zweiten Gruppe gehören z.B. der Zeilenvorschub und der horizontale Tabulator.
- Für die Speicherung und die Ein-/Ausgabe werden Zeichen als *Bitkombinationen* codiert.
  - Java verwendet den Unicode, der den ASCII-Code als Teilmenge enthält.
  - Die Bitkombination (der Code) eines Zeichen wird hier in zwei Bytes (16 Bit) *gespeichert*.
  - Die Bitkombination eines Zeichens kann auch *als Zahl aufgefasst* werden. Z.B. entspricht die Bitkombination des Buchstabens A die Zahl 65, des Buchstabens a die Zahl 97 und des Zeichens 1 der Zahl 49.

# Interne Darstellung von Zeichen (2)

Erste Zeichen des Unicode-Zeichensatzes (entspr. ASCII-Code)

Dez. Hex. Zeich.			Dez. Hex. Zeich.			Dez. Hex. Zeich.			Dez. Hex. Zeich.		
0	0000	NUL	32	0020		64	0040	@	96	0060	`
1	0001	SOH	33	0021	!	65	0041	A	97	0061	a
2	0002	STX	34	0022	"	66	0042	B	98	0062	b
3	0003	ETX	35	0023	#	67	0043	C	99	0063	c
4	0004	EOT	...			68	0044	D	100	0064	d
5	0005	ENQ	45	002D	-	69	0045	E	101	0065	e
6	0006	ACK	46	002E	.	...			...		
7	0007	BEL	47	002F	/	87	0057	w	119	0077	w
...			48	0030	0	88	0058	X	120	0078	x
10	000A	LF	49	0031	1	89	0059	Y	121	0079	y
...			50	0032	2	90	005A	Z	122	007A	z
12	000C	FF	...			91	005B	[	123	007B	{
13	000D	CR	57	0039	9	92	005C	\	124	007C	
...			58	003A	:	93	005D	]	125	007D	}
30	001E	RS	...			94	005E	^	126	007E	~
31	001F	US	63	003F	?	95	005F	_	127	007F	DEL

# Literale für primitive Datentypen (1)

- Im Quelltext verwendete Werte (z.B. Zahlen) werden als *Literale* bezeichnet
- Numerische Datentypen
  - Für numerische Werte des Integer-Datentyps können die Vorzeichen + bzw. – und die Ziffern 0..9 verwendet werden. Das Vorzeichen + kann auch entfallen.
  - Bei *Gleitkommazahlen* wird als Dezimaltrennzeichen der Punkt . verwendet.
  - Die Exponentialschreibweise für Gleitkommazahlen sieht z.B. folgendes vor: 4.56e3   4.56E3   -5.677e90   +6.777e-8
  - Für den Datentyp `long` wird das Suffix L verwendet, z.B. 126L
  - Für den Datentyp `float` wird das Suffix F (f) verwendet, z.B. 100.0F
  - Standarddatentypen: `int` (falls in `int` darstellbar) und `double`
- Boolescher Datentyp
  - Boolesche Literale sind `true` und `false`

# Literale für primitive Datentypen (2)

## ■ Alphanumerischer Datentyp **char** für Zeichen

- Einzelne Zeichen werden durch Apostrophe ' eingeschlossen.

```
char letter = 'X';
```

- Zeichen können auch als Escape-Sequenzen dargestellt werden. Die Escape-Sequenz ist in Apostrophe ' zu setzen.
- Mit der Escape Sequenz '\uhhhh' kann der Unicode für das gewünschte Zeichen direkt angegeben werden (h: Hexadezimalziffer).

Escape-Sequenz	Bedeutung
\uhhhh Bsp: \u0045	Ein spezielles Zeichen (im Beispiel das Zeichen E mit dem hexadezimalen UniCode 0045)
\b	Backspace
\t	Tabulator
\n	line feed
\f	form feed
\r	carriage return
\"	Anführungszeichen
\'	Hochkomma
\\	Backslash

# Liste der primitiven Datentypen in Java

Typ	Größe	Minimum	Maximum	Defaultwert	Beispiel
boolean	8 bit	-	-	false	false
char	16 bit	Unicode 0	Unicode $2^{16}-1$	\u0000 (null)	'a'
byte	8 bit	-128	+127	(byte) 0	34
short	16 bit	$-2^{15}$	$+2^{15}-1$	(short) 0	21345
int	32 bit	$-2^{31}$	$+2^{31}-1$	0	42322554
long	64 bit	$-2^{63}$	$+2^{63}-1$	0L	134567L
float	32 bit	$2^{-149}$	$(2-2^{-32}) \cdot 2^{127}$	0.0f	3.1415f
double	64 bit	$2^{-1074}$	$(2-2^{-52}) \cdot 2^{1023}$	0.0d	3.253245
void	-	-	-		



# Variablen (1)

## ■ Was sind Variablen?

- Die Anweisungen eines Programms arbeiten mit *Daten*, die *unterschiedliche Werte (variable Werte)* annehmen können, z.B. Zwischenergebnisse einer Berechnungen.
- Hierzu werden so genannte Variablen verwendet, für die entsprechende Speicherplatz im Arbeitsspeicher des Computers reserviert wird.
- Zugriff auf den Speicherplatz über den Variablennamen (Bezeichner).
- Zur Deklaration von Variablen bedarf es Namen und Datentyp.

# Variablen (2)

- Voraussetzung für die Nutzung von Variablen
  - Eine Variable muss deklariert sein, bevor sie Daten aufnehmen kann.
  - Eine Variable muss einen Wert enthalten, bevor sie verwendet werden kann.

```
int a; // Variable mit Namen "a" deklariert  
a = 12; // Variable "a" bekommt Wert 12
```

```
// Variable deklarieren und direkt Wert zuweisen  
int b = 14;
```

# Gültigkeitsbereich lokaler Variablen

- Java kennt verschiedene Arten von Variablen.
- Variablen, die innerhalb eines Anweisungsblocks (z.B. einer Methode) deklariert werden, heißen **lokale Variablen**.
- Lokale Variablen sind nur innerhalb des deklarierenden Blocks gültig → *lokaler Gültigkeitsbereich*.
- Eine lokale Variable verliert mit der schließenden Klammer `}` des Blocks ihre Gültigkeit.
- Ein Bezeichner in Java muss innerhalb seines Gültigkeitsbereichs *eindeutig* sein.
  - Innerhalb des Blocks (auch in inneren Blöcken) darf keine Variable mit gleichem Namen deklariert werden.
- Lokale Variablen primitiver Datentypen erhalten keinen Default-Wert! Hier meckert der Compiler.



Heusch 5.2  
Ratz 4.4

# Beispiel für den Gültigkeitsbereich von Variablen

## Zulässig

```
{  
    int number;  
}  
// Die Variable number  
// ist hier nicht mehr  
// gültig  
  
{  
    int number;  
    /* jetzt existiert  
     * wieder eine lokale  
     * Variable mit dem  
     * Namen number */  
}
```

## Unzulässig

```
{  
    int number;  
  
    {  
        int number;  
        // unzulässig  
    }  
}  
  
/* Diese Variablendeklaration  
 * ist nicht zulässig, da die  
 * Variable number aus dem  
 * übergeordneten Block noch  
 * gültig ist. */
```

# Syntax der Variablendeklaration

**Syntax:  
siehe  
Anhang**

- ```
type identifier [= init_value]  
{, identifier [= init_value] };
```
- Die Variablendeklaration besteht aus einem *Datentyp* (*type*) und dem *Namen* (Identifizierer, Bezeichner) der Variablen (*identifier*).
- Die Namen der Variablen werden direkt nach dem Datentyp, getrennt durch ein oder mehrere Leerzeichen, angegeben.
- Der Name der Variablen muss sich an die Vorgaben für Bezeichner halten.
- Die Deklaration einer Variablen ist eine Anweisung und wird mit einem Semikolon abgeschlossen.
- Mehrere Variablen desselben Typs können in einer Anweisung deklariert werden. Die Variablennamen werden dabei durch Kommata getrennt.
- Nach einem Zuweisungsoperator = kann der Variablen ein Anfangswert (*init\_value*) zugewiesen werden (optionale Initialisierung).

# Namenskonventionen und Beispiele

- Variablennamen beginnen üblicherweise mit einem Kleinbuchstaben (*Konvention*, keine zwingende Vorgabe), z.B. `time`
- Soll ein Name aus mehreren Wörtern zusammengesetzt werden, werden die Wörter direkt hintereinander geschrieben (ohne Trennzeichen).

Zur Abgrenzung werden die (folgenden) Wörter mit Großbuchstaben begonnen, z.B. `maxTextLength`, `timeToEnd`

- Gültige Beispiele:

```
float price;  
double extent, radius;  
int minCount = 2;  
char a = 'c';
```

- Ungültige Beispiele:

```
double &count; // Bezeichner von Variablen  
                // dürfen kein & enthalten  
float a b c;   // mehrere Variablennamen durch Kommata trennen
```

# Werte zuweisen

- Deklaration und Initialisierung von Variablen
  - Mit der *Deklaration* einer Variablen ist durch den Datentyp lediglich festgelegt, welche Daten in der Variablen gespeichert werden können
  - Mit der Deklaration einer lokalen Variablen erfolgt keine automatische *Wertzuweisung*  
Der Wert ist noch *unbestimmt*, die Variable ist noch nicht initialisiert
  - → explizite Initialisierung (erste Wertzuweisung) möglich
- Der Wert einer Variable kann jederzeit geändert werden
  - Syntax für Wertzuweisungen an Variablen

`identifizier = expression ;`
  - Nach dem Namen der Variablen folgt der Zuweisungsoperator = und der Ausdruck
  - Der Ausdruck kann ein Literal, oder ein komplexer Ausdruck sein
  - Die Wertzuweisung wird mit einem Semikolon abgeschlossen

# Beispiele

```
// Deklarationen
int i;
int k;
char c;
double d = 1.7; // zusätzlich zur Deklaration
                // Initialisierung

// korrekte Wertzuweisungen
i = 20;          // Leerzeichen sind erlaubt
k=i;            // ohne Leerzeichen geht es auch
c = 'a';         // auch Mischen möglich

// fehlerhafte Wertzuweisungen
i = 0.15;        // 0.15 ist kein gültiger int-Wert
v = 7;          // Keine Variable mit Namen v deklariert.
c="Test";       // "Test" ist kein char-Wert
c='Test';       // 'Test' ebenfalls nicht
```



# Typkompatibilität und Typkonversion

## ■ Typkompatibilität

- Einer Variablen kann nur ein Wert des Datentyps zugewiesen werden, den sie selbst besitzt oder den sie aufgrund ihres Wertebereichs umfasst.
- Beispiel: Der Datentyp `int` ist kompatibel zum Datentyp `double`, aber `double` ist nicht kompatibel zu `int`.

## ■ Typkonversion

- Bei kompatiblen Typen führt Java automatisch eine implizite Typkonversion durch.
- Sind die Typen nicht kompatibel, so kann eine explizite Typkonversion (`cast`) durchgeführt werden.
- Logische Werte (`boolean`) können nicht umgewandelt werden.

## ■ Syntax für die explizite Typkonversion

```
(type) expression;
```

# Beispiele:

```
int position = 100;      // korrekt
double size = 1.45;      // korrekt
double weight = 80;      // korrekt

int number1 = 1.23;      // falsch, Fehlermeldung

int number2 = (int)1.23;  // korrekt durch erzwungene
                        // Typenumwandlung

float length1 = 1.45;     // falsch, Literal ist vom Typ
                        // double. Compiler liefert:
                        // "possible loss of precision"

float length2 = 1.45F;    // korrekt
float length3 = 1.45f;    // korrekt
```

# Symbol. Konstanten – unveränderliche Variablen

- In Java werden symbolische Konstanten durch unveränderliche Variablen dargestellt.
- Sobald während der Programmausführung eine Wertzuweisung erfolgt ist, ist diese endgültig (**final**). Es kann keine zweite Wertzuweisung an eine Konstante erfolgen nachdem sie einen Wert erhalten hat (auch in Zuweisung möglich).
- Syntax der Deklaration und Initialisierung von symbolischen Konstanten

```
final type identifier [= init_value]  
           {, identifier [= init_value] };  
identifier = expression;
```

- Beispiele:

```
final double MWST = 0.19;  
final double PI;  
PI = 3.141592;
```

- Konvention: Üblicherweise werden Konstanten mit Großbuchstaben geschrieben.

# Zeichenketten

- Für Zeichenketten (Strings) gibt es keinen primitiven Datentyp.
- Für Zeichenketten gibt es in Java die Klasse String.
- Diese Klasse wird später ausführlich behandelt.
- Da in den folgenden Beispielen zuweilen aber Strings verwendet werden, vorab schon ein paar Informationen
  - String-Literale (Zeichenketten im Quellprogramm) werden in doppelte Anführungszeichen " gesetzt, z.B. "Test".
  - Eine Zeichenkette kann mit + an eine andere Zeichenkette angehängt werden (Stringverkettung, Konkatination).  
Beispiel: "Hal" + "lo"
  - Zeichenketten besteht aus Unicode-Zeichen.

# Anhang: Beschreibung der Syntax von JAVA

## Bei Sprachen unterscheidet man Syntax und Semantik:

- **Syntax:** Legt fest, welche Ausdrücke *erlaubt* sind (sprachliche Form).
  - Deutsch: „Der Baum blüht.“ und „Der Baum spaziert.“ sind syntaktisch korrekte Sätze.
  - *Beispiel* aus Java: Nach dem Wort `class` folgt ein Wort und anschließend eine in geschweifte Klammern eingeschlossene Zeichenfolge.
- **Semantik:** Legt die *Bedeutung* eines Ausdrucks der Sprache fest.
  - Deutsch: Die beiden Sätze oben haben unterschiedliche (mehr oder weniger sinnvolle) Bedeutungen.
  - *Beispiel* aus Java: Das Schlüsselwort `class` leitet eine Klassendefinition ein. Das Wort direkt nach `class` ist der Name der Klasse.

# Formale Sprachen

- Programmiersprachen sind **formale Sprachen**.
- Für *formale* Sprachen ist die Syntax exakt definiert. Es ist festgelegt:
  - Zulässige *Zeichen* der Sprache
  - Zulässige Ausdrücke (*Wörter*, Zeichenketten aus den Zeichen der Sprache)
- Beispiel:

Gegeben sei das Alphabet (die Menge von Zeichen)  $A = \{"a", "b", "c"\}$ .  
Die **Sprache**  $L_1$  sei die Menge aller Wörter über  $A$ , die mit "a" beginnen und mit "a" enden.  
Dann sind folgende Zeichenketten Wörter von  $L_1$ :

"a", "aa", "aaa", "aba", "aca", "aaaa", "aaba", "aaca", ...

Keine Wörter von  $L_1$  sind dagegen:

"", "b", "c", "ab", "ca", "bcc", ...

# Grammatik (1)

- Die meisten Sprachen haben *unendlich viele* Wörter (potentielle Zeichenketten), so dass man durch Aufzählung der Wörter die Sprache nicht definieren kann.
- Eine **Grammatik** im Sinne der Informatik ist eine allgemeine, eindeutige Beschreibung einer *formalen* Sprache mit Hilfe von Regeln.
- Sinn und Zweck einer Grammatik: Sie soll eine *endliche Beschreibung* einer Sprache ermöglichen. (Vgl. Vorlesung „Theoretische Informatik“)

## Grammatik (2)

Eine Grammatik  $G = (T, N, P, S)$  besteht aus 4 Teilen:

- T Die Menge der *Terminalsymbole*. Aus diesen werden die Wörter der Sprache gebildet.  
Terminalsymbole sind Zeichen des zugrunde liegenden Alphabets.
- N Die Menge der *Nichtterminalsymbole* (Hilfssymbole/Variablen für syntaktische Einheiten).  
Sie treten in der Sprache nicht auf, sondern müssen durch Terminalsymbole ersetzt werden.
- P Die Menge der *Produktionen* (auch Produktionsregeln oder Regeln genannt).  
Eine Regel  $X \rightarrow Y$  besagt, dass ein Teilwort  $X$  durch ein Teilwort  $Y$  ersetzt werden kann, wobei  $X$  und  $Y$  aus Terminal- und Nichtterminalsymbolen bestehen.
- S Das *Startsymbol* (ein spezielles Element aus  $N$ ), von dem ausgehend die Produktionen angewandt werden.



# Kontextfreie Grammatiken

- Im Allgemeinen können linke und rechte Seite von Produktionen aus beliebigen Kombinationen von Terminal- und Nichtterminalsymbolen bestehen.
- Die für die Informatik wichtigste Kategorie von Grammatiken sind diejenigen, deren Produktionen auf der linken Seite aus *genau einem Nichtterminalsymbol* bestehen.
- Diese Grammatiken heißen *kontextfrei*.

# Backus-Naur-Form (BNF)

- Die Backus-Naur-Form (BNF) ist eine häufig verwendete *Beschreibungsform für kontextfreie Grammatiken*.
- Die BNF wird insbesondere zur Beschreibung der Syntax von *Programmiersprachen* verwendet.
- Die linke Seite jeder Regel besteht aus *einem* Nichtterminalsymbol.
- Die rechte Seite einer Regel besteht aus einer beliebig langen Kette, die Terminal- und Nichtterminalsymbole enthalten kann.
  - Die „leere“ rechte Seite ( $\lambda$ ) ist erlaubt.
- Linke und rechte Seite einer Regel werden durch das Zeichen „ $::=$ “ voneinander getrennt.
- Schreibweise in dieser Vorlesung:
  - Terminalsymbole sind **fett** gedruckt.
  - Nichtterminalsymbole sind *kursiv* gedruckt.
  - Symbole der Metasprache BNF sind in **Standard-Schrift** gedruckt.

# Beispiel – Grammatik für $L_1$ in BNF

$T = \{ \mathbf{a}, \mathbf{b}, \mathbf{c} \}$

$N = \{ \textit{Wort}, \textit{Buchstabe}, \textit{Teilwort} \}$

$P = \{ \textit{Wort} ::= \mathbf{a} \textit{Teilwort} \mathbf{a} \quad (1)$

$\textit{Wort} ::= \mathbf{a} \quad (2)$

$\textit{Teilwort} ::= \textit{Buchstabe} \textit{Teilwort} \quad (3)$

$\textit{Teilwort} ::= \lambda \quad (4)$

$\textit{Buchstabe} ::= \mathbf{a} \quad (5)$

$\textit{Buchstabe} ::= \mathbf{b} \quad (6)$

$\textit{Buchstabe} ::= \mathbf{c} \quad (7)$

}

$S = \textit{Wort}$

# Erweiterte Backus-Naur-Form (EBNF)

- Weil BNF-Grammatiken unübersichtlich werden können, wurde die BNF um einige *Abkürzungsmöglichkeiten* erweitert.
- Dies führt zur Erweiterten Backus-Naur-Form (EBNF).
  
- Es gibt verschiedene Varianten der EBNF.
- Eine gebräuchliche sieht folgende Abkürzungsmöglichkeiten vor:
  - | der senkrechte Strich trennt Alternativen
  - [ ] eckige Klammern enthalten optionale Bestandteile
  - { } geschweifte Klammern enthalten Bestandteile, die null, ein oder mehrfach wiederholt werden dürfen
  - { }+ geschweifte Klammern mit + enthalten Elemente, die ein oder mehrfach wiederholt werden können
  - ( ) runde Klammern gruppieren mehrere Bestandteile, z.B. mehrere Alternativen

# Beispiel: Grammatik für $L_1$ in EBNF

$T = \{ \mathbf{a}, \mathbf{b}, \mathbf{c} \}$

$N = \{ \textit{Wort}, \textit{Buchstabe}, \textit{Teilwort} \}$

$P = \{$ 
  
 $\textit{Wort} \quad ::= \mathbf{a} \textit{Teilwort} \mathbf{a} \mid \mathbf{a} \quad (1+2)$ 
  
 $\textit{Teilwort} \quad ::= \textit{Buchstabe} \textit{Teilwort} \mid \lambda \quad (3+4)$ 
  
 $\textit{Buchstabe} \quad ::= \mathbf{a} \mid \mathbf{b} \mid \mathbf{c} \quad (5-7)$ 
  
 $\}$

$S = \textit{Wort}$

# Sprache – ein zweites Beispiel (1)

- Gegeben sei das Alphabet

$A = \{"1", "2", "3", "4", "5", "6", "7", "8", "9", "0", "+", "-", "*", "/", "(", "\)" \}$

- Die Menge  $L_A$  der *arithmetischen Ausdrücke* ist eine Sprache über  $A$ .

Z.B. gehören folgenden Wörter zu  $L_A$ :

122+4

$8*(4+177/3-(11*(18+7)))$

13

Die folgenden Wörter gehören nicht zu  $L_A$ :

28(18

++/-5

## Sprache – ein zweites Beispiel (2)

- EBNF-Grammatik für die Sprache  $L_A$  (einfache arithmetische Ausdrücke):

$T = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, +, -, *, /, (, ) \}$

$N = \{ \textit{Ausdruck}, \textit{Term}, \textit{Faktor}, \textit{Zahl}, \textit{Ziffer} \}$

$P = \{$ 
  
 $\textit{Ausdruck} ::= \textit{Term} \{ ( + \mid - ) \textit{Term} \}$  (1)
  
 $\textit{Term} ::= \textit{Faktor} \{ ( * \mid / ) \textit{Faktor} \}$  (2)
  
 $\textit{Faktor} ::= \textit{Zahl} \mid ( \textit{Ausdruck} )$  (3)
  
 $\textit{Zahl} ::= \{ \textit{Ziffer} \}^+$  (4)
  
 $\textit{Ziffer} ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$  (5)
  
 $\}$

$S = \textit{Ausdruck}$