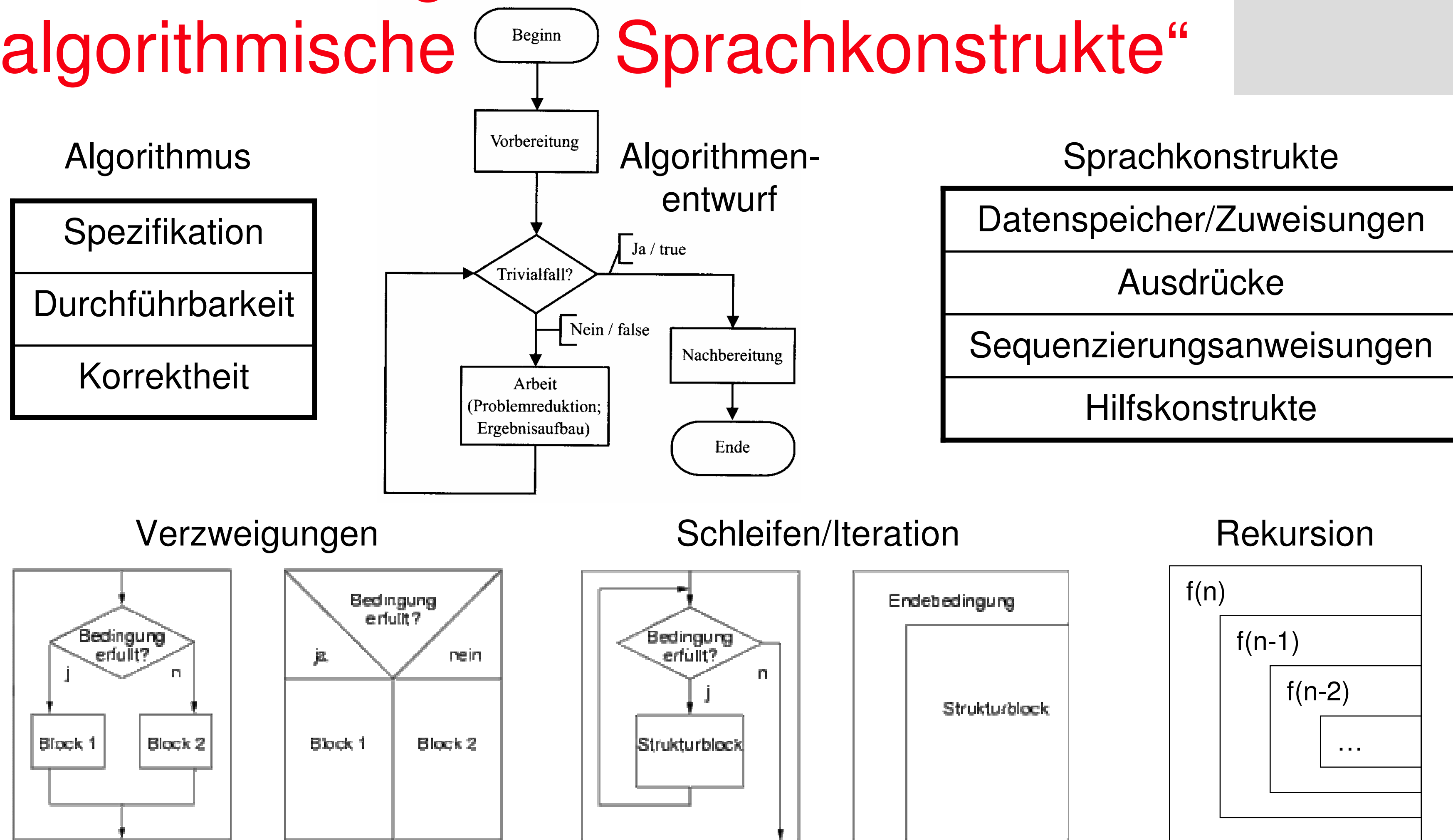
Universität Bremen

Grammatik, Bezeichner, Datentypen, Entwicklungszyklus

Thomas Röfer

Syntax/Semantik
Chomsky-Grammatiken
(Erweiterte) Backus-Naur-Form
Bezeichner
Datentypen, Literale
Entwicklungszyklus

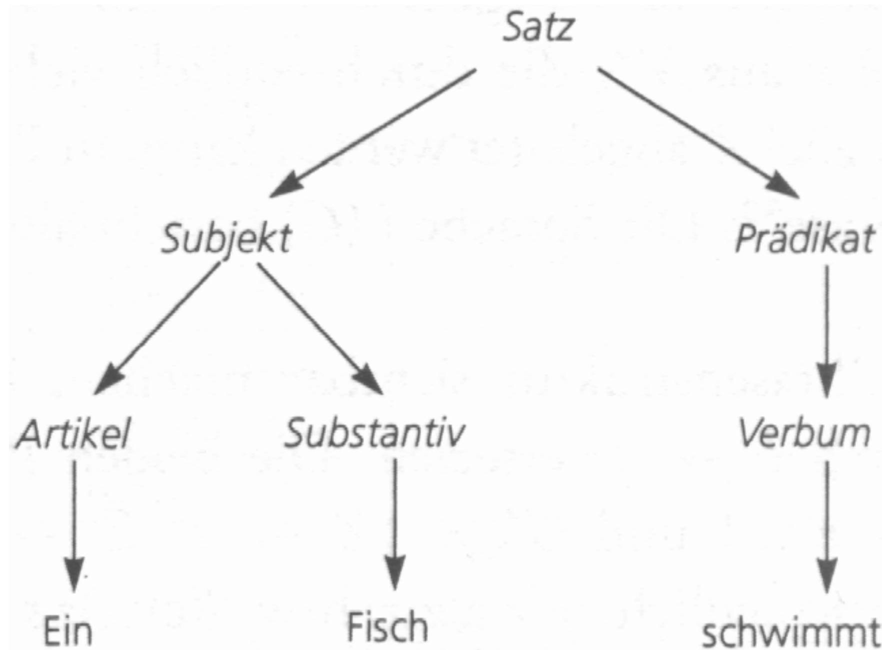
Rückblick „Algorithmen und algorithmische Sprachkonstrukte“



Syntax und Semantik

- **Syntax**
  - Korrekte Art und Weise, sprachliche Elemente zusammenzuführen und zu Sätzen zuzuordnen
  - Ist durch formale *Grammatik* eindeutig beschrieben
  - *Lexikalische Analyse*: Zerlegung in *Tokens*
  - Tokens: Bezeichner, Literale, Schlüsselwörter, Operatoren…
- **Semantik**
  - legt die *Bedeutung* der Sprachkonstrukte fest
  - Formale Beschreibung der Semantik sehr aufwendig (aber möglich)
    - *daher oft informelle Beschreibung der Semantik*

Chomsky-Grammatiken

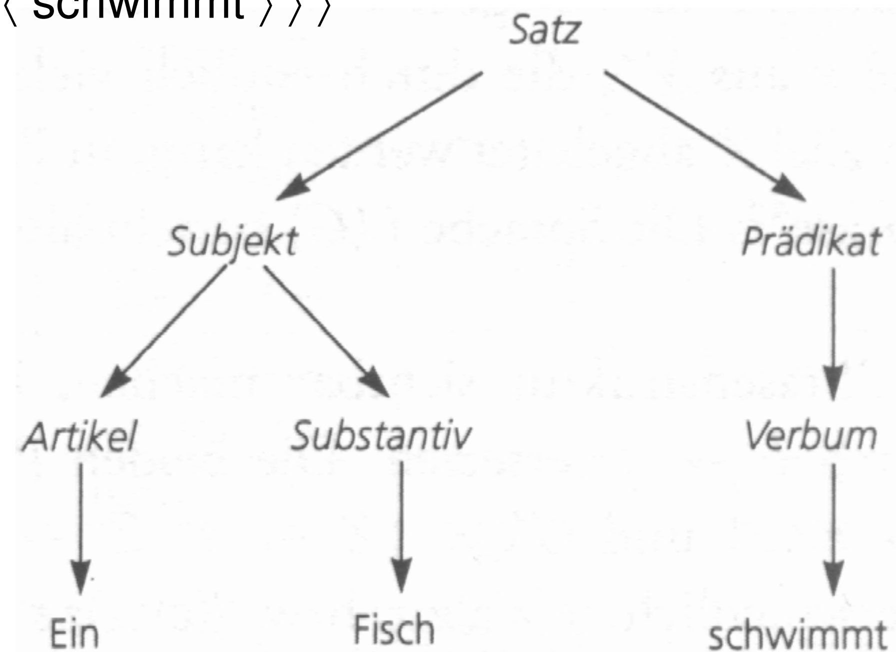


- **Produktionen**
  - *Satz* → *Subjekt Prädikat*
  - *Subjekt* → *Artikel Substantiv*
  - *Subjekt* → *Substantiv*
  - *Prädikat* → *Verbum*
  - *Artikel* → Ein
  - *Substantiv* → Fisch
  - *Substantiv* → Fische
  - *Verbum* → schwimmt
  - *Verbum* → schwimmen

- **Begriffe**
  - Vokabular (*Nichtterminale* + Terminale)
  - Startsymbol/Ziel
  - Sprachschatz (alle möglichen terminalen Zeichenreihen ohne Grammatik)
  - Satzform/Phrase

Chomsky-Grammatiken (2)

- **Struktur**
  - ⟨ ⟨ ⟨ Ein ⟩ ⟨ Fisch ⟩ ⟩ ⟨ ⟨ schwimmt ⟩ ⟩ ⟩
  - schwach äquivalent: ⟨ ⟨ Ein ⟩ ⟨ ⟨ Fisch ⟩ ⟨ schwimmt ⟩ ⟩ ⟩
  - strukturäquivalent: ⟨ ⟨ Ein ⟨ Fisch ⟩ ⟩ ⟨ ⟨ schwimmt ⟩ ⟩ ⟩
- **Parsen**
  - Ist eine Zeichenreihe eine Phrase?
    → Zerteilung (*parsing*)
  - Umkehr des Ableitungssystem
    → Reduktions-/Zerteilungssystem
- **Chomsky-Grammatik**
  - Ein Grammatik aus Terminalen, Nichtterminalen, Produktionen und einem Ziel

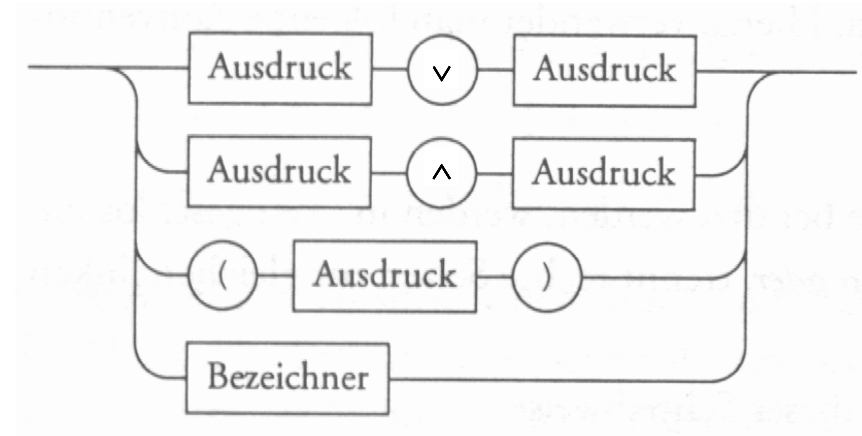


Backus-Naur-Form

- **BNF**
  - Ausdruck = Ausdruck '$\vee$' Ausdruck |
Ausdruck '$\wedge$' Ausdruck |
'(' Ausdruck ')' |
Bezeichner
  - Bezeichner = 'a' | 'b' | … | 'z'
- **Zerlegung 1**
  - Ausdruck = $(a \vee b) \wedge c \vee d$
  - Ausdruck = $(a \vee b) \wedge c$ Ausdruck = d
  - Ausdruck = $(a \vee b)$ Ausdruck = c   Bezeichner = d
  - Ausdruck = $a \vee b$ Bezeichner = c
  - Ausdruck = a   Ausdruck = b
  - Bezeichner = a   Bezeichner = b

Backus-Naur-Form

- **BNF**
  - Ausdruck = Ausdruck '∨' Ausdruck | Ausdruck '∧' Ausdruck | '(' Ausdruck ')' | Bezeichner
  - Bezeichner = 'a' | 'b' | … | 'z'



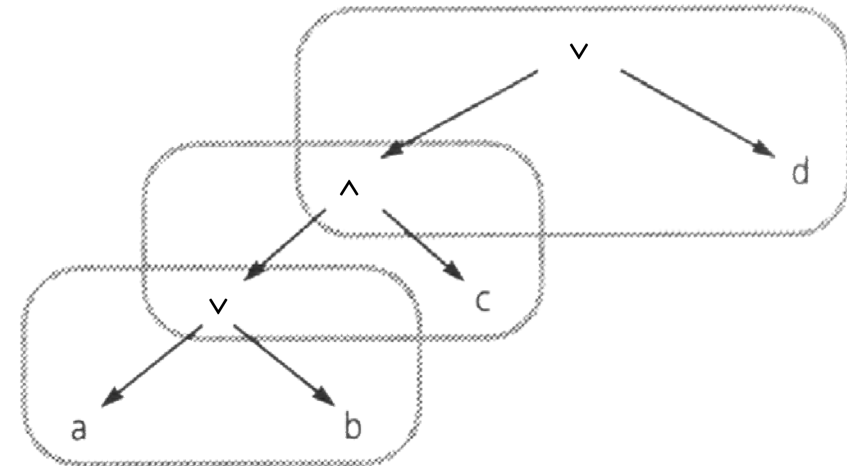
- **Zerlegung 2**
  - Ausdruck = (a ∨ b) ∧ c ∨ d
  - Ausdruck = (a ∨ b) Ausdruck = c ∨ d
  - Ausdruck = a ∨ b Ausdruck = c Ausdruck = d
  - Ausdruck = a Ausdruck = b Bezeichner = c Bezeichner = d
  - Bezeichner = a Bezeichner = b

Erweiterte Backus-Naur-Form

- **Bedeutung**
  - […] bezeichnet einen optionalen Teil auf der rechten Seite
  - (…) umschließt eine Gruppe von Zeichen
  - { … } Inhalt der Klammer wird beliebig oft wiederholt (auch 0-mal)
  - | trennt Alternativen
- **EBNF in EBNF**
  - Produktion = Bezeichner '=' Ausdruck
  - Ausdruck = Term { '|' Term }
  - Term = Faktor { Faktor }
  - Faktor = Bezeichner | '\"' Literal '\"' |
    '(' Ausdruck ')' |
    '[' Ausdruck ']' |
    '{' Ausdruck '}'

Zerlegung mit EBNF

- **EBNF**
  - Ausdruck = Term { '∨' Term }
  - Term = Faktor { '∧' Faktor }
  - Faktor = '(' Ausdruck ')' | Bezeichner
  - Bezeichner = 'a' | 'b' | … | 'z'



Kantorowitsch-Baum

- **Zerlegung**
  - Ausdruck = (a ∨ b) ∧ c ∨ d
  - Term = (a ∨ b) ∧ c　　Term = d
  - Faktor = (a ∨ b)　　Faktor = c　　Faktor = d
  - Ausdruck = a ∨ b　　Bezeichner = c　　Bezeichner = d
  - Term = a　　Term = b
  - Faktor = a　　Faktor = b
  - Bezeichner = a　　Bezeichner = b

Bezeichner

- **Schreibung**
  - Erstes Zeichen muss ein Buchstabe, '_' oder '$' sein
  - Alle weiteren Zeichen können Buchstaben, Ziffern, '_' oder '$' sein
  - Groß- und Kleinschreibung wird unterschieden
- **EBNF**
  - Identifier = FirstChar { FurtherChar }
  - FirstChar = '_' | '$' | 'A' | 'B' | … | 'Z' | 'a' | 'b' | … | 'z' | …
  - FurtherChar = FirstChar | '0' | '1' | … | '9'
- **Schlüsselwörtern (Nicht für Bezeichner verwenden)**
  - abstract, assert, 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

Konventionen für Bezeichner

- **Variablen**
  - Variablen beginnen mit Kleinbuchstaben: stream, name
  - Mehrere Worte werden durch Großbuchstaben aneinander gefügt: thisIsAVeryLongName
  - '_' und '$' werden nicht verwendet
  - Für Laufvariablen in Schleifen: i, j, k, …
- **Konstanten**
  - Nur Großbuchstaben, Ziffern und '_' MAX_WORDS, Math.PI
- **Klassen**
  - Genau wie Variablen, beginnen aber mit einem Großbuchstaben System, Factorial

Elementare Datentypen in Java

Datentyp	Default	Speicherplatz	Wertebereich
byte	0	1 Byte (8 Bits)	-128 bis 127
short	0	2 Bytes (16 Bits)	-32768 bis 32767
int	0	4 Bytes (32 Bits)	-2147483648 bis 2147483647
long	0	8 Bytes (64 Bits)	-9223372036854775808 bis 9223372036854775807
float	0.0	4 Bytes (32 Bits)	±1.40239846E-45 bis ±3.40282347E+38
double	0.0	8 Bytes (64 Bits)	±4.94065645841246544E-324 bis ±1.79769313486231570E+308
boolean	false	? (min. 1 Bit)	false, true
char	'\u0000'	2 Bytes (16 Bits)	'\u0000' bis '\uFFFF'

Ganzzahlige Datentypen

- **Zahlensysteme**
  - Dezimal, es gibt die Ziffern 0 bis 9
  - Binär, nur die Ziffern 0 und 1
  - Oktal, nur die Ziffern 0 bis 7
  - Hexadezimal, die Ziffern 0 bis 9 und A bis F

4 2 = 4*10+2

1 0 1 0 1 0 = 32+8+2

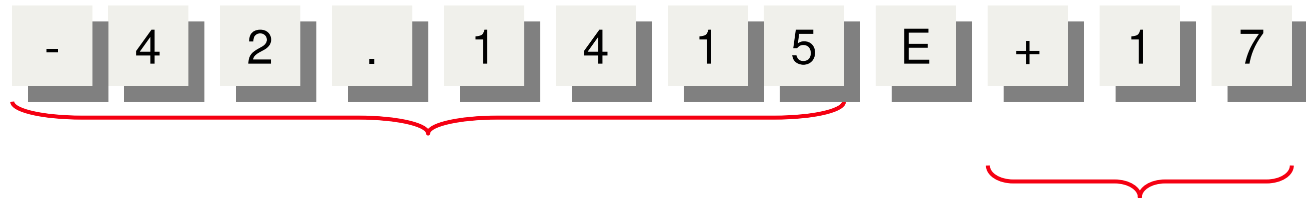
5 2 = 5*8+2

2 A = 2*16+10

- **Literale**
  - Dezimal: 42, +42, -42, 42L, 42l
  - Oktal: 052, +052, -052, 052L, 052l
  - Hexadezimal: 0x2A, +0x2a, -0x2A, 0x2aL, 0x2Al

Ein *L* am Ende bedeutet *long*

Fließkommazahlen



- **Format**
  - Mantisse
  - Exponent
  - Wert = Mantisse × 10^{Exponent}
- **Literale**
  - Float: 0.0f, .382f, 3.1415F, -3.1415f, 17E7f, 17e-7f, 17E+7F
  - Double: 0.0, .0392039029303, -3.141592653589d, 1E+100

Ein *F* am Ende bedeutet *float*, ein *D double*. Wird nichts angegeben, ist der Typ *double*

Zeichen

- **Literale**
  - Normal: 'a', 'b', 'c'
  - Wagenrücklauf: '\r'
  - Zeilenvorschub: '\n'
  - Seitenvorschub: '\f'
  - Tabulatorsprung: '\t'
  - Backspace: '\b'
  - Hochkomma: '\''
  - Backslash: '\\'
  - Unicode Zeichen: '\u12ab'

\ = Escape-Zeichen

```
f\u006fr(int i = 0; i < 10; ++i)
```

=

```
for(int i = 0; i < 10; ++i)
```

Java-Quelltexte werden im Unicode verarbeitet.
An jeder Stelle kann \uXXXX stehen.

Zeichenketten

- **String ist eine Klasse, kein Basistyp**
  - Daher kann man Ausdrücke schreiben, wie z.B. *s.equals("Hallo")*
- **Literale**
  - "Hallo", "wie geht's?"
  - "Ich sage \"mir geht's gut\"" → *Ich sage "Mir geht's gut"*
- **Beispiele**
  - System.out.println("\"DM\"\t\"Euro\"\n1\t0,51");
  - "DM"   "Euro"
    1   0,51

In String-Literalen können alle Escape-Sequenzen aus char-Literalen verwendet werden

Typkonvertierung

- **Automatisch**
  - byte → short → int → long → float → double
  - char → int
- **Manuell**
  - byte b; short s; int i; long l; float f = 1.5; double d = -1.5; char c;
  - b = (byte) f; (b == 1)
  - f = (float) 178.2
  - l = (int) d; (l == -2)
  - c = (char) 32
- **Durch Funktionen**
  - String s = Integer.toString(i);
  - i = Integer.parseInt(s);
  - d = Double.parseDouble(s);

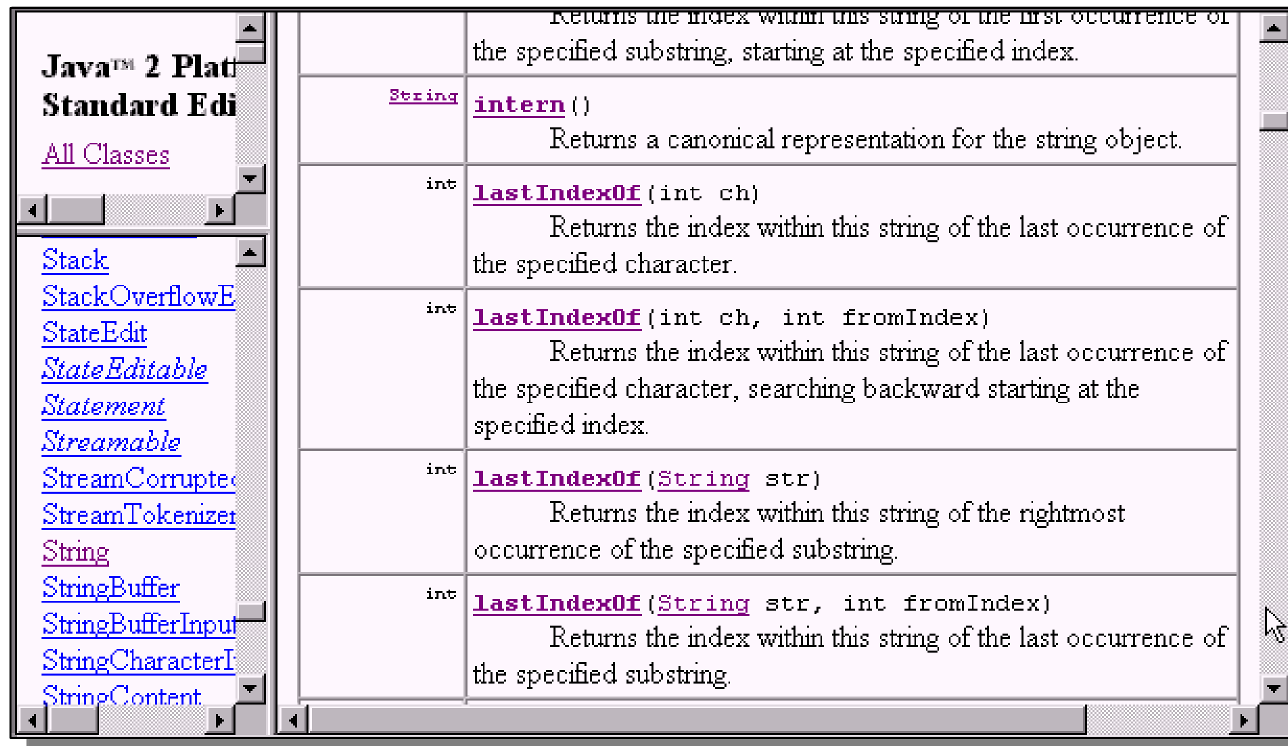
Bei der Typumwandlung von double/float in einen Ganzzahltyp wird immer abgerundet

Wrapper-Klassen

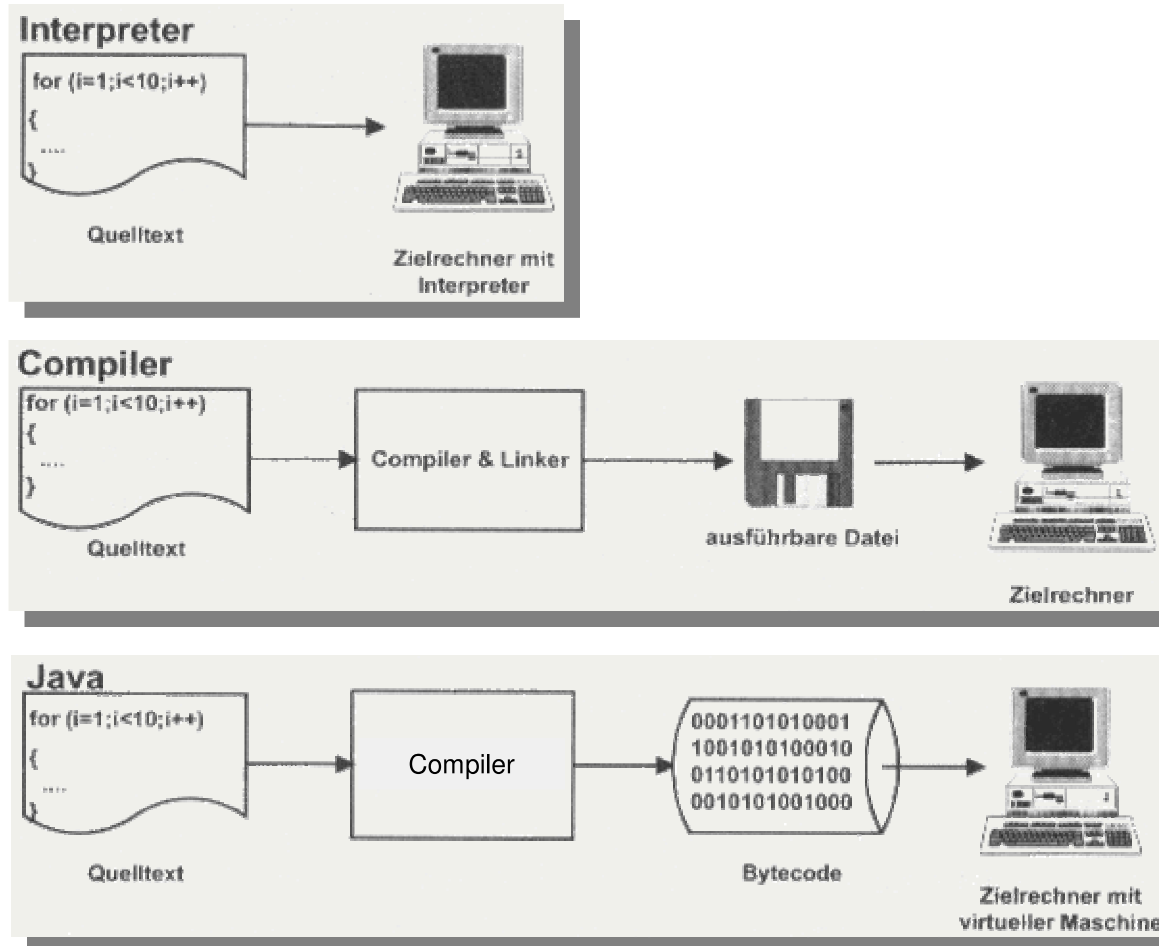
Datentyp	Wrapper
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

- **Generell**
  - *Wrapper*.MAX_VALUE
    - *z.B. Byte.MAX_VALUE*
  - *Wrapper*.MIN_VALUE
  - *Wrapper*.parse*Wrap*(String s)
    - *z.B. Integer.parseInt("1234");*
  - *Wrapper*.toString(type t)
    - *z.B. Integer.toString(1234);*
- **Float und Double**
  - Wrapper.NEGATIVE_INFINITY
    - *z.B. Double.NEGATIVE_INFINITY*
  - Wrapper.POSITIVE_INFINITY
  - Wrapper.NaN

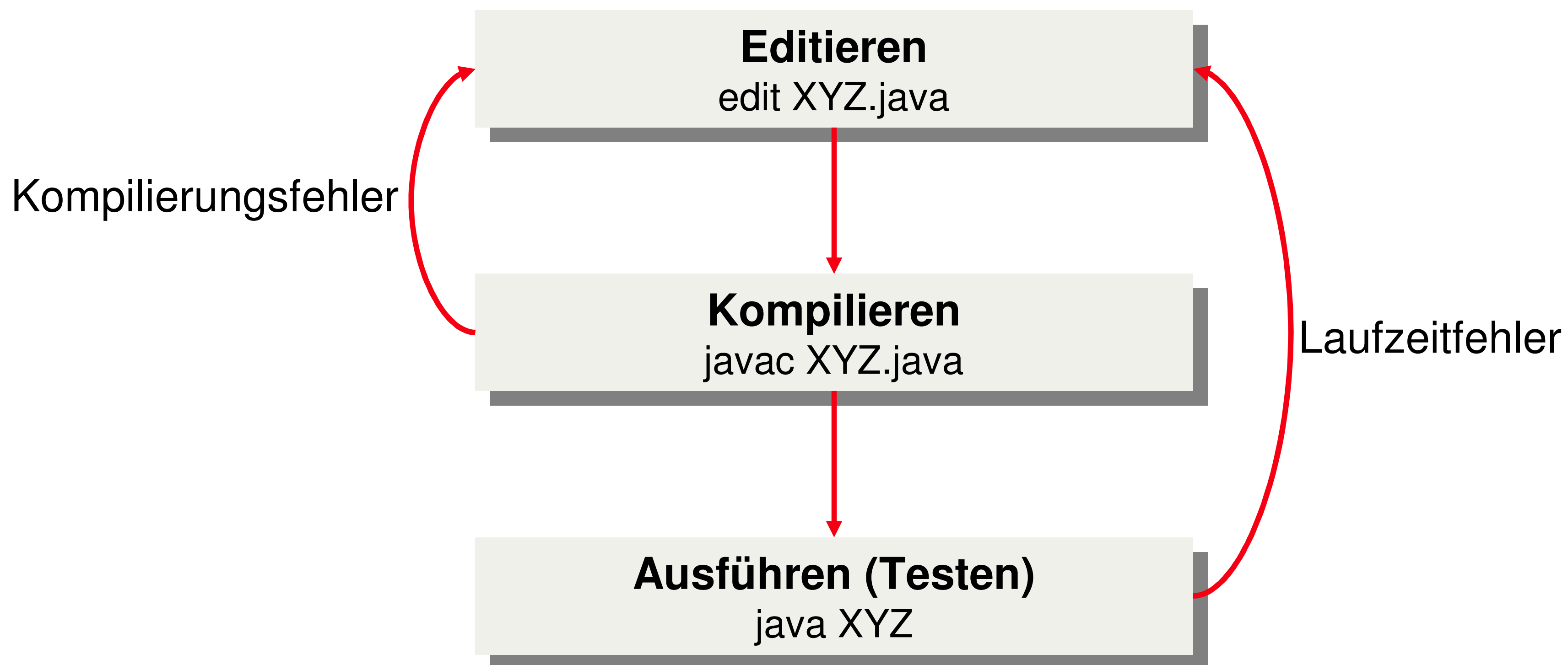
Nachschlagen in der Dokumentation



Vom Quelltext zur Ausführung



Der Entwicklungszyklus



Fehler beim Programmieren

- **Tippfehler (auch *Copy and Paste*)**
  - Semikolon vergessen, Klammern passen nicht...
  - Können vom Compiler entdeckt werden, müssen aber nicht
- **Strukturfehler**
  - Compiler verwendet andere Zuordnung als man denkt
- **Typfehler**
  - Werden bei Zuweisungen erkannt
  - Bleiben in Ausdrücken teilweise unerkannt
- **Fehler im Algorithmus**
  - Kommt bei Informatikern nicht vor ;-)

```
for(int i = 0; i < 10; ++i)
  for(int j = 0; j < 10; ++i)
    System.out.println(i+j);
```

```
if(a == 1)
  if(b == 1)
    System.out.println("ok");
else
  System.out.println("a != 1");
```

```
int i = "1";
float f = 3 / 2;
```

Fehlerbehebung

- **Kompilierungsfehler**
  - Nur den ersten Fehler beachten, die anderen könnten Folgefehler sein!
  - Fehlermeldung lesen, Quelltext bei der angegebenen Zeilennummer ansehen
    - *Dokumentation zu Fehlermeldung und falschem Konstrukt lesen*
- **Fehler während der Programmausführung**
  - Falls eine Fehlermeldung angezeigt wird, diese lesen und den Quelltext bei der angegebenen Zeilennummer ansehen
  - Testausgaben einbauen oder Debugger verwenden
    - *Aber keine Fehler durch solche Testausgaben einbauen!*
  - Verschiedene Testfälle durchspielen: Wann tritt der Fehler auf, wann nicht?

Nur ein verstandener Fehler ist ein beseitigter Fehler!

Nach Änderung des Programms kompilieren nicht vergessen!