

# **Praktische Informatik 1**

## **Grammatik, Java-Programme, Bezeichner, Datentypen**

Thomas Röfer

- Syntax/Semantik
- Chomsky-Grammatiken
- (Erweiterte) Backus-Naur-Form
- Eigenständige Java-Programme
- Bezeichner
- Datentypen, Literale
- Musterlösung, Übungsblatt

# Zeichen Literale

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

\ = Escape-Zeichen

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

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

||

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


# 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 

- 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 == -1
- c = (char) 32

Bei der Typumwandlung von double/float in einen Ganzzahltyp wird immer in Richtung 0 abgerundet

- Durch Funktionen

- String s = Integer.toString(i);
- i = Integer.parseInt(s);
- d = Double.parseDouble(s);



# Hüllklassen (Wrapper-Klassen)

- Konstruktion
  - `Type(type t)`
    - z.B. `Integer i = new Integer(1234);`
- Wert abfragen
  - `type typeValue()`
    - z.B. `int j = i.intValue();`
- Hilfsmethoden
  - `Type.parseType (String s)`
    - z.B. `Integer.parseInt("1234");`
  - `Type.toString(type t)`
    - z.B. `Integer.toString(1234);`

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

# Hüllklassen (Wrapper-Klassen)

## Konstanten

- Generell
  - *Type.MAX\_VALUE*
    - z.B. *Byte.MAX\_VALUE*
  - *Type.MIN\_VALUE*
- Float und Double
  - *Type.NEGATIVE\_INFINITY*
    - z.B. *Double.NEGATIVE\_INFINITY*
  - *Type.POSITIVE\_INFINITY*
  - *Type.NaN*

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

# Autoboxing

- Java wandelt automatisch einen Wert eines Basisdatentypen in ein Objekt der entsprechenden Hüllklasse um
- Dadurch lassen sich Werte von Basisdatentypen wie Objekte behandeln
  - z.B. in vordefinierte sog. Collections einfügen
- Funktioniert nur für Zuweisungen und Parameterübergaben

```
java.util.Vector<Integer> v =  
    new java.util.Vector<Integer>();  
v.add(17);  
// entspricht v.add(new Integer(17));
```

```
5.toString(); // Geht nicht!
```

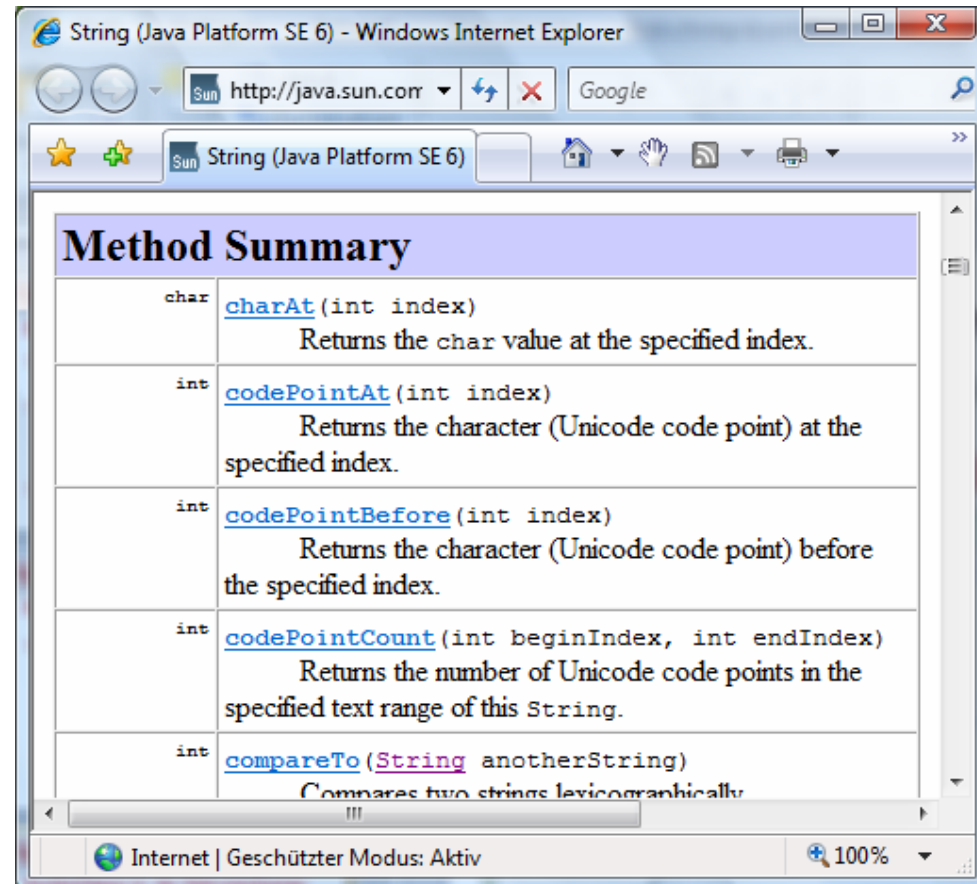
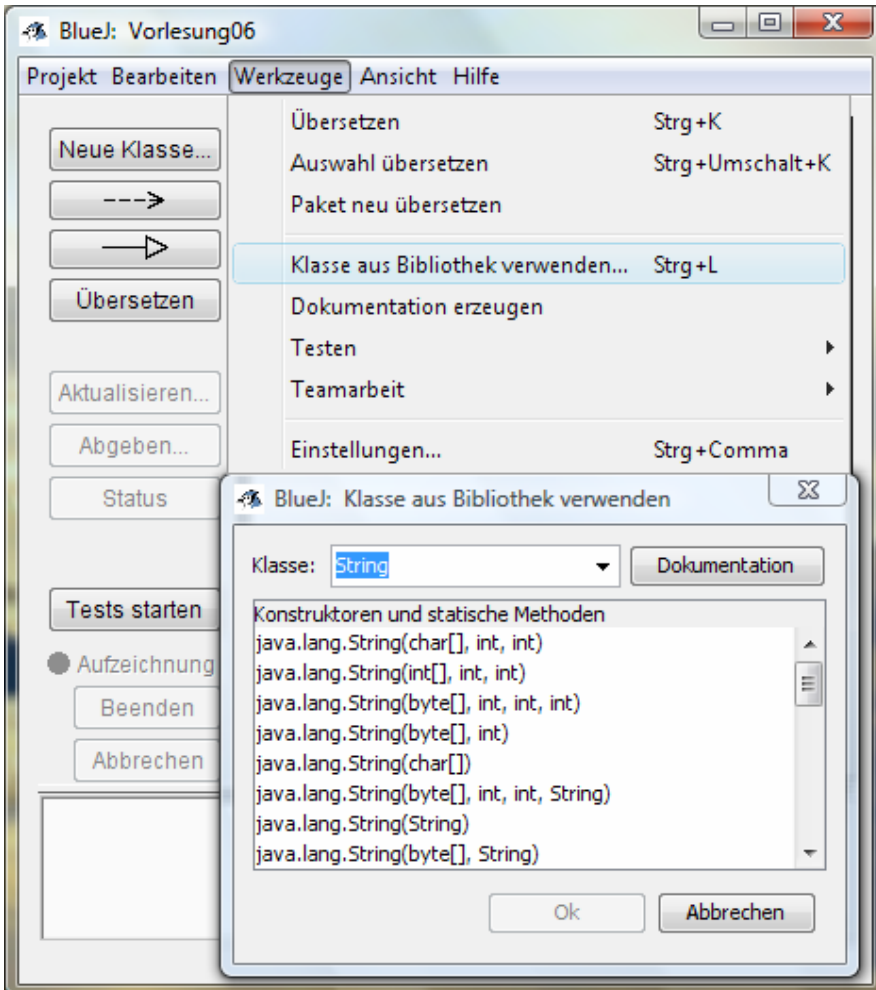
# Autounboxing

- Java wandelt automatisch ein Objekt einer Hüllklasse in einen Wert des entsprechenden Basisdatentypen um

```
int[] a = {1, 2, 3, 4, 5};  
int sum = 0;  
int i = 0;  
while (i < a.length) {  
    sum = sum + a[i];  
    i = i + 1;  
}
```

```
Integer[] a = {1, 2, 3, 4, 5};  
Integer sum = 0;  
Integer i = 0;  
while (i < a.length) {  
    sum = sum + a[i];  
    i = i + 1;  
}
```

# Nachschlagen in der Dokumentation



# Musterlösung zu Übungsblatt 3

- Aufgabe 1
  - Beim Beweis beschreiben, was man tut und warum man es tut
- Aufgabe 2
  - Das Sieb des Eratosthenes testet keine Teilbarkeit!
  - Nicht mehr jede Zeile einzeln dokumentieren
  - Listings sind Teil der schriftlichen Abgabe
    - Sonst können die Tutoren keine Anmerkungen an den Code schreiben

Praktische Informatik I WS 2007/08

## Übungsblatt 3

Musterlösung

Aufgabe 1 Können Sie das auch beweisen, Columbo? (50%)

Folgende Funktion berechnet  $\sum_{i=0}^{n-1} a_i$ , d.h. die Summe aller Elemente der Reihung  $a$ , wobei  $n$  die Anzahl der Elemente ist:

```
1 int calcSum(int[] a) {
2     int n = a.length;
3     int sum = 0;
4     int i = 0;
5     while (i < n) {
6         sum = sum + a[i];
7         i = i + 1;
8     }
9     return sum;
10 }
```

Beweist mit der Methode von Floyd.

**Schleifeninvariante.** Da die Funktion die Lösung Stück für Stück in der Variablen *sum* aufbaut, wird folgende Schleifeninvariante verwendet, die ausdrückt, dass die Gesamtsumme über alle Elemente der Reihung stets den bereits in *sum* aufsummierten Elementen plus der Summe der noch nicht verarbeiteten Elemente entspricht:

$$\sum_{j=0}^{n-1} a_j = \text{sum} + \sum_{j=i}^{n-1} a_j$$

**Vor der Schleife.** Hier ist zu zeigen, dass die Schleifeninvariante gilt, nachdem alle Variablen initialisiert wurden, was offensichtlich der Fall ist:

$$\begin{aligned} \sum_{j=0}^{n-1} a_j &= \text{sum} + \sum_{j=i}^{n-1} a_j \quad || \text{sum} = 0, i = 0 \\ &= 0 + \sum_{j=0}^{n-1} a_j \\ &= \sum_{j=0}^{n-1} a_j \end{aligned}$$

# Dubletten – 1. Platz

```

int a=0;
int b=0;
int d=0;
while(z < (n+1)) // Wird ausgeführt solange z kleiner ist als di
{
    alle[z]=z; // Erstellt den Wert z im Array.
    z=z+1; // Erhöht den Wert z.
}
while(y<n) // Wird ausgeführt bis die letzte Zahl des Arrays erreicht
{
    if(raus[y]==false) // Wenn y nicht rausgeschmissen
    {
        while(x>y) // Wird solange
        {
            if(raus[x]==false) //
            {
                b=alle[x]%alle[y]
                if(b==0) // Gibt
                {
                    raus[x]=true
                }
                x=x-1; // x wird
            }
            else
            {
                x=x-1; // x wird
            }
        }
        y=y+1; // y wird erhöht u
        x=n; // x wird zurück au
    }
    else
    {

```

```

int rest=0; // hier wird der Rest der Modulteilung current
int menge=0; // In dieser Variable wird gespeichert, wieviel
int zaehler=0; // zaehler ist eine simple Zählvariable um d
while(array < (grenze+1)) // Eine Schleife, die das Programm
{
    zahlen[array]=array; // In diesem Array werden die Werte
    array=array+1; // Der Indexwert wird erhöht um im Index
}
while(lastprime<grenze) // Diese Schleife wird solange ausge
{
    if(noprime[lastprime]==false) // Ist die Zahl nicht
    {
        while(current>lastprime) // Schleife, die
        {
            if(noprime[current]==false) // W
            {
                rest=zahlen[current]%zah
                if(rest==0) // Wenn curr
                {
                    noprime[current]
                }
                current=current-1; // H
            }
            else
            {
                current=current-1;
            }
        }
        lastprime=lastprime+1; //lastprime wird
        current=grenze; // current wird wieder a
    }
    else
    {

```



# Dubletten – 2. Platz

```
public class Primzahlen {  
    static int sieb(int max) {  
        boolean[] reihung = new boolean[max+1];    // Es  
        int anzahl = 0;  
  
        for (int i=2; i<=max; i++) {    // Diese Schleife  
            if (reihung[i]==false) {    // Folgendes wird  
                for (int k=2*i; k<=max; k=k+i) {    // D  
                    reihung[k] = true;  
                }  
                System.out.println(i);    // Weil i eine  
                anzahl++;    // Hier wird mitgezaehlt, v  
            }  
        }  
        return anzahl;  
    }  
}
```

```
public class Primzahlen {  
    static int sieb(int max) {  
        boolean[] reihung = new boolean[max+1];  
        int anzahl = 0;  
  
        for (int j=2; j<=max; j++) {  
            if (reihung[j]==false) {  
                for (int k=2*j; k<=max; k=k+j) {  
                    reihung[k] = true;  
                }  
                System.out.println(j);  
                anzahl++;  
            }  
        }  
        return anzahl;  
    }  
}
```



# Dubletten – 3. Platz

```
public class Eratosthenes {  
    static int primesUpToNumber (int n) { //max=n  
        boolean prime[] = new boolean [n+1]; {  
            int i;  
  
            int z=0;  
            for (i=0; i<=n; i++) prime[i] = true; //setze alle auf true  
            for (i=2; i<=n; i++) {  
                if (prime[i]) {  
                    z++;  
                    System.out.println (i);  
                    for (int v=i+i; v<=n; v=v+i) prime[v] = false;  
                }  
            }  
            return z;  
        }  
    }  
}
```

```
public class Eratosthenes {  
    public static void prime (int e) {  
        int j = e +1;  
        boolean p[] = new boolean [j]; {  
            int i=0;  
  
            int z=0;  
            for (i=0; i<=e; i++) p[i] = true;  
            for (i=2; i<=e; i++) {  
                if (p[i]) {  
                    z++;  
                    System.out.println(i);  
                    for (int v=i+i; v<=e; v=v+i) p[v] = false;  
                }  
            }  
            System.out.println("Gefundene Primzahlen: "+z);  
        }  
    }  
}  
}>>>> file: uebung03.java
```

# Übungsblatt 5

## • Aufgabe 1

### • EBNF

## • Aufgabe 2

### • Dasselbe Problem iterativ und rekursiv lösen

### • (Naive) rekursive Version optimieren

Praktische Informatik I WS 2007/08

## Übungsblatt 5

Abgabe: 05.12.07

### Aufgabe 1 Grammatiken mit EBNF (40%)

Gegeben ist folgende EBNF zur Beschreibung eines einfachen Taschenrechners:

```
Rechnung = Zahl {Operator Zahl};
Operator = '+' | '-' | '*' | '/';
Zahl = Ziffern ['.' Ziffern] ['e' ['+' | '-'] Ziffern];
Ziffern = Ziffer {Null | Ziffer};
Ziffer = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
Null = '0';
```

Überprüft, ob die folgenden Sätze der Grammatik entsprechen. Gebt die entsprechenden Regeln an, um eure Antwort zu begründen.

- a)  $18e6+170+37.92$
- b)  $17.0139^{*}11$
- c)  $5203-23/98e$
- d)  $24.25/7.36-81$
- e)  $9841.41$

Erläutert den Schwachpunkt dieser Grammatik.

Erweitert die gegebene Grammatik, sodass die allseits bekannte Regel „Punkt- vor Strichrechnung“ berücksichtigt wird. Erläutert eure Lösung.

**Hinweis.** Die nichtterminalen Symbole, d.h. die Regeln, werden „von innen nach außen“ abgearbeitet. Enthält also eine Regel weitere Regeln, werden zunächst diese ausgewertet.

### Aufgabe 2 Wie Hasen sich vermehren – Eine iterative und eine rekursive Sicht in Java (60%)

Sinngemäß formulierte der Italiener Leonardo Pisano – er nannte sich *Filius Bonacci* – in seinem 1202 veröffentlichten Werk *Liber abaci* oder *Buch über den Abakus* folgende Übungsaufgabe zur Addition:

Ein neugeborenes Hasenpaar wird in einen umzäunten Garten gesetzt. Jedes Hasenpaar erzeugt während seines Lebens jeden Monat ein weiteres Paar. Ein neugeborenes

# Übungsblatt 5 – Aufgabe 1

## Beispiel

$z = 0x7F$   
 $OH = x7F$   
 $H = 7F$   
 $HZ = 7 \quad HZ = F$

$z = 0$   
 $OH =$   
 $0$

Zahl = 0 OktHex | Dez  
 Dez = DezZ1 {DezZ}  
 OktHex = 'x' Hex | Okt  
 Hex = HexZ {HexZ}  
 Okt = ~~OktZ~~ {OktZ}  
 HexZ = '0' | ... | '9' | 'A' | ... | 'F'  
 DezZ = '0' | DezZ1  
 DezZ1 = '1' | ... | '9'  
 OktZ = '0' | ... | '7'

## Übungsblatt 5 – Aufgabe 2

Monat	0	1	2	3	4	5	6	7	8	9
Hasen	0	1	1	2	3	5	8	13	21	34

Monat	10	11								
Hasen	55	89								

$1 + 1$  ↗

# Übungsblatt 5 – Aufgabe 2

## Eine andere rekursive Funktion

- Approximation der Zahl  $e$

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} + \Lambda$$

```
static double e(int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return 1 / factorial(n) + e(n - 1);  
    }  
}
```

```
static double factorial(int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}
```

# Übungsblatt 5 – Aufgabe 2

## Effizientere Varianten

```
static double e(int n) {  
    return e2(n)[0];  
}  
  
static double[] e2(int n) {  
    if (n == 0) {  
        return new double[]{1, 1};  
    } else {  
        double[] r = e2(n - 1);  
        double f = n * r[1];  
        return new double[] {1 / f + r[0], f};  
    }  
}
```

```
static double e(int n) {  
    return e2(n, new double[n + 1]);  
}  
  
static double e2(int n, double[] f) {  
    if (n == 0) {  
        f[0] = 1;  
        return 1;  
    } else {  
        double r = e2(n - 1, f);  
        f[n] = n * f[n - 1];  
        return 1 / f[n] + r;  
    }  
}
```