Java-Grundlagen

Hubert Grassmann

27. April 2011

1 Vorbemerkungen

Java ist nicht „einfach“ zu erlernen, es ist eine „richtige“ Programmiersprache, deren Regeln man sich erarbeiten muß.

Java-Programme werden in einen „Bytecode“ übersetzt, zur Ausführung ist ein Interpreter notwendig. Daraus resultiert das Vorurteil, daß Java langsam sei. Daran ist etwas Wahres: Rechenintensive Programme können im Vergleich zu C-Programmen bis zum 10-fachen der Rechenzeit benötigen. Daran arbeiten die Java-Entwickler. „Viele Beobachter gehen davon aus, daß Java-Programme in 2-3 Jahren genausoschnell wie C/C++-Programme laufen.“ Dieses Zitat stammt aus dem Jahr 1997.

Das Abtract Windowing Toolkit (AWT), das Graphik-Routinen bereitstellt, werden wir manchmal nutzen, ohne daß hier auf dessen Funktionalität eingegangen wird.

Die zu bearbeitenden Daten werden als „Variablen“ bezeichnet (dies hat nichts mit dem gleichnamigen Begriff aus der Analysis zu tun!),

„Konstanten“ sind spezielle Variable, deren Wert beim Programmstart feststeht und der nicht mehr geändert werden kann.

Jede im Programm verwendete Variable hat einen Namen, der aus den Zeichen A, B, ... , Z, a, b, ... z, 0, ... , 9 und dem Unterstrich zusammengsetzt ist; dabei darf das erste Zeichen keine Ziffer sein.

Vor einer ersten Verwendung einer Vaiablen muß der „Typ“ der Variablen festgelegt werden (die Variable wird „deklariert“). Der Typ einer Variablen legt einerseits fest, wieviel Speicherplatz hierfür bereitgestellt werden soll; andererseits wird anhand der Typen während der Übersetzung festgestellt, ob mit den Operanden auszuführende Operationen erlaubt sind, eine Typunverträglichkeit führt zu einem Syntaxfehler. Dies ist für den Programmierer oft ärgerlich, aber immer hilfreich, weil sich bei Syntaxfehlern (neben Schreibfehlern) oft echte Programmierfehler entdecken lassen. Allerdings sind verschiedene Programmiersprachen unterschiedlich streng bei der Typ-Prüfung. Manche Sprachen lassen sogar arithmetische Operationen zwischen Zahlen und Wahrheitswerten zu. Auf der anderen Seite ist manchmal eine Typ-Umwandlung notwendig, hierzu werden spezielle Funktionen bereitgestellt (`(float) 1.0`), oder die Programmierer greifen zu „dirty tricks“, wobei sie bei Java auf Granit beißen. Viele Fehler, die z.B. beim Programmieren in C++ dadurch entstehen, daß der Programmierer für die Speicherverwaltung selbst zuständig ist, können bei Java nicht mehr auftreten. Es gibt keine Pointer.

2 Primitive Typen

Bei Java stehen die folgenden Grundtypen zur Verfügung (alle Zahlen sind vorzeichenbehaftet):

Typ	Größe	Bemerkung
byte	1 Byte	-128 ... 127
short	2 Byte	- 32768 ... 32767
int	4 Byte	2 Milliarden
long	8 Byte	? (ÜA) ..... bisher alle ganzzahlig
float	4 Byte	Gleitkommazahl, IEEE 754-1985 Standard
double	8 Byte	max. 10^{317}
char	2 Byte	Unicode-Zeichensatz, noch lange nicht ausgefüllt
boolean	1 Bit	**true** oder **false**

3 Operatoren

Mit Hilfe von Operatoren kann man aus Variablen Ausdrücke zusammensetzen, es stehen u.a. die arithmetischen Operatoren

`+, −, *, /, %`

zur Verfügung (`%` ist der modulo-Operator), darüberhinaus gibt es Zuweisungsoperatoren (man spart Schreibarbeit, aber es wird etwas unübersichtlich), den Inkrementoperator ++, den Dekrementoperator −−, Bitoperatoren, Verschiebungsoperatoren, die Vergleichsoperatoren

`==, !=, <>, <=, >=`

sowie die logischen Operatoren `&&, ||, !`. Wenn in einem Ausdruck mehrere logische Operatoren vorkommen, so werden die Ergebnisse von links nach rechts ausgewertet. Die Auswertung wird abgebrochen, sobald das Ergebnis feststeht (das Ergebnis einer || -Operation ist **true** oder das Ergebnis einer &&-Operation ist **false**); das ist nützlich, wenn z.B. weiter rechts stehende Operanden gar nicht existieren.

Bei Java werden die Operatoren in Ausdrücken, in denen unterschiedliche Zahltypen vorkommen, automatisch in den den übergeordneten Zahltyp konvertiert. Das sollte man ausprobieren. Es darf nämlich nicht falsch verstanden werden! Ein häufiger Fehler ist

```
float x;
x = 1 / 2;
```

Nun, das ist völlig korrekt, es wird nur nicht das berechnet, was man vielleicht denkt: x ist eine Gleitkommazahl, also wird das Ergebnis gleich 0.5 sein. Nein, 1 und 2 sind ganzzahlig (im Gegensatz zu 1.0 und 2.0) und der Term 1/2 wird ganzzahlig berechnet, ist also gleich 0. An diese Stelle gehört ein ganz großes Ausrufungszeichen.

Ein Programm setzt sich aus Anweisungen zusammen, z.B.

```
a = b + c; x = y * z;
```

Rechts vom Gleichheitszeichen steht ein Ausdruck, links eine Variable, die den Wert des berechneten Ausdrucks erhält.

Einer Charakter-Variablen weist man mit `a = 'a'` einen konstanten Wert zu; bei einer String-Variablen (einer Zeichenkette) sind die Anführungszeichen zu verwenden: `s = "abc"`; numerische Konstanten werden als Gleitkommazahlen gedeutet, wenn irgendetwas darauf hinweist, das es welche sein sollen, z.B. `3.14`, `2f`, `0F`, `1e1`, `.5f`, `6.`; wenn `f` oder `F` (für **float**) fehlt, aber ein Dezimalpunkt oder ein Exponent auf eine Gleitkommazahl hinweist, wird **double** angenommen.

Obwohl nicht alle Operatoren eingeführt wurden, soll hier die Vorrangs-Reihenfolge dargestellt werden; wenn davon abgewichen werden soll, sind Klammern zu setzen:

. [] ()

einstellig: + − ~ ! ++ −− **instanceof**

zweistellig: * / % + − < <= >= > == != & ^ | && || ?: =

Variablen, die einem der bisher genannten primitiven Datentypen zugehören, stehen nach ihrer Deklaration sofort zur Verfügung, können also belegt und weiter verarbeitet werden. Die Deklarationen müssen nicht, wie in anderen Sprachen, am Anfang eines Blocks stehen, jedenfalls aber vor der ersten Verwendung.

Bei einer Division durch Null entsteht der Wert $\pm\infty$, wenn der Zahlbereich überschritten wird, ensteht NaN.

4 Felder

Ein Feld (array) ist eine Folge von Variablen ein- und desselben Datentyps (der nicht primitiv sein muß, ein Feld von gleichartigen Feldern ist eine Matrix usw.). Um mit einem Feld arbeiten zu können, muß es

- deklariert werden,

```
int folge [];

int [] folge ;

float [][] matrix;
```

- Speicherplatz erhalten,

```
folge = new int[5];
```

  damit stehen `folge [0], ... , folge [4]` zur Verfügung,

- gefüllt werden:

```
folge [1] = 7;
```

Eine andere Möglichkeit besteht darin, das Feld sofort zu initialisieren:

```
int folge[] = {1,2,3,4,5};
```

dann gibt folge.length die Länge an.

5 Programmsteuerung

Um den Programmfluß durch die bearbeiteten Daten steuern zu können, benötigt man Auswahlanweisungen; es gibt die **if**-, **if**–**else**- und die **switch**-Anweisung.

```
if ((a != b) || (c == d))
{
  a:= 2; // hier sind 2 Anweisungen zu einen Block zusammengefasst worden
  b:= c + d;
}
else
  a:= 0;
```

Die Auswertung der logischen Bedingungen erfolgt in der angegebenen Reihenfolge; es wird höchstens einer der Blöcke abgearbeitet. Der **else**-Zweig kann entfallen.

Zählergesteuerte Wiederholungsanweisung:

Syntax: **for** (init ; test ; update) anweisung;

```
for (i = 1; i <= 10; i++)
{
  a = a * a;
  b = b * a;
}
```

```
for (i = a.length - 1, i >= 0; i--)
  s = s + a[i];
```

Kopfgesteuerte Wiederholungsanweisung:

Syntax: **while** (condition) anweisung;

```
while (true)
  a = a * a;          // dies ist eine Endlosschleife
```

Die while-Schleife wird solange wiederholt, wie die angegebene Bedingung wahr ist. Innerhalb der Schleife sollte etwas getan werden, was den Wahrheitswert der Bedingung ändert.

Fußgesteuerte Wiederholungsanweisung:

Syntax: **do** anweisung; **while** (condition);

```
do
{
  a = a * a;
  b = b * a;
}
while (b < 20);
```

Die **do**-Schleife ist nicht-abweisend, sie wird mindestens einmal durchlaufen.

Schleifen können auch geschachtelt werden:

```
for (i = 1; i <= 10; i++)
  for (j = 1; j <=10; j++)
    a[i][j] = b[j][i];
```

Ich habe mit Pascal den Test gemacht, ob (bei nicht-quadratischen Feldern) die Reihenfolge des Durchlaufs der Schleifenvariablen eine Auswirkung auf die Rechenzeit hat und keinen Unterschied feststellen können. Man sollte aber beachten: In `for`-Schleifen wird normalerweise auf indizierte Variable zugegriffen und die Berechnung der Adresse eines Feldelements aus dem Index braucht Zeit. Man sollte solche Dereferenzierungen so selten wie möglich durchführen.

Es ist sicher ein schlechter Progammierstil, wenn man gezwungen ist, die Berechnung innerhalb einer Schleife abzubrechen oder die Schleife ganz zu verlassen. Für diesen Notfall stehen die Anweisungen **continue**

und **break** zur Verfügung, die auch noch mit Sprungmarken versehen werden können (das erinnert doch sehr an die **goto**–Möglichkeiten, mit der man sich Programm-Labyrithe schaffen kann).

Schließlich sind noch Anweisungen zur Ein- und Ausgabe von Daten notwendig.

```
System.out.print(Zeichenkette);
```

gibt die Zeichenkette aus. Zeichenketten lassen sich mit dem +-Operator verknüpfen: wenn a eine numerische Variable ist, so wird sie innerhalb der print-Anweisung mittels "␣" + a in eine Zeichenkette umgewandelt und ausgegeben. Wenn println verwendet wird, so erfolgt ein Zeilenvorschub.

Die Eingabe ist etwas schwieriger: Hier müssen mögliche Fehler explizit abgefangen werden (es muß aber nichts getan werden):

```
public class cat
{
  public static void main(String args[])
  {
    DataInput d = new DataInputStream(System.in);
    String line;
    try
    {
      while ((line = d.readLine()) != null)
      System.out.println(line);
    }
    catch (IOException ignored)
    { }
  }
}
```

Wir lesen eine ganze Zahl, indem wir eine Zeichenkette lesen und diese umwandeln:

```
import java.io.*;
public static int readint()
{
  String s; int i, k;
  DataInput d = new DataInputStream(System.in);
```

```
  try
  {
    s = d.readLine();
    i = 0;
    for (k=0; k < s.length(); k++)
      i = 10 * i + ((int)s.charAt(k) - 48);
    return i;
  }
  catch (IOException ignored) {}
  return(0);
}
```

Noch ein Beispiel für das Rechnen mit beliebig langen Zahlen:

```
import java.math.*;
import java.util.*;
import java.io.*;
class v2
{
  public static BigInteger lies() // soll eine lange Zahl lesen
  {
    BigInteger a;
    String s;
    System.out.println("->␣");
    DataInput d = new DataInputStream(System.in);
    try
    {
      s = d.readLine();
      a = new BigInteger(s);
      return (BigInteger) a;
    }
    catch (IOException ignored)
    {}
    a = new BigInteger("0");
    return(a);
  }

  public static void schreib(BigInteger a)
  {
    String s = a.toString();
    System.out.println("␣"+s);
  }

  public static void main(String args[])
  {
    BigInteger a,b,c;
    a = lies();
```

```
    schreib(a);
    b = lies();
    schreib(b);
    c = a.multiply(b);
    schreib(c);
  }
}
```

6 Methoden

Rechenabläufe, die mehrmals (mit verschiedenen Parametern) wiederholt werden, oder die man zur besseren Strukturierung eines Programms vom Rest des Programms abtrennen will, realisiert man durch Methodenaufrufe. Dazu müssen also Methoden definiert werden. Die Kopfanweisung einer Methodendeklaration kann folgende Form haben:

```
public static TYP name (f1, f2, ...)
```

Dabei gibt `TYP` den Ergebnistyp der Metode an, der mit der **return**-Anweisung erzeugt wird; damit wird die Arbeit der Methode beendet.

Das folgende Beispiel realisiert das sogenannte Hornerschema zur Berechnung des Werts $f(x_0)$ für ein Polynom

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0,$$

dabei wird die Ausklammerung

$$f(x) = (\ldots((a_n x + a_{n-1})x + a_{n-2})x + \ldots + a_1)x + a_0$$

genutzt, um das Berechnen von Potenzen zu vermeiden.

```
import java.io.*;
public class horner
{
```

```
public static double horn (double[] a, double x0)
{
  int i, n;
  float f;
  n = a.length - 1;
  f = a[n];
  for (i = n-2; i >= 0; i--)
    f = f * x0 + a[i];
  return f;
}

public static void main(String arg[])
{
  float a[] = {1,1,1};          // also x^2+x+1
  float h;
  h = horn(a, 2.0f);
  System.out.println(h);
}
```

Falls keine Parameter übergeben werden (müssen), ist eine leere Parameterliste ( ) zu übergeben. Es sind rekursive Aufrufe möglich.

Beim Aufruf eines Unterprogramms wird bei nichtprimitiven Typen direkt auf den Speicherplatz, den die Parameter belegen, zugegriffen (call by reference), d.h. auch die Eingangsparameter können verändert werden (Vorsicht!). Als aktuelle Parameter können nicht nur Variable, sondern auch Ausdrücke übergeben werden (die kann man naturgemäß nicht ändern). Eine sorgfältige Dokumentation aller Parameter ist unbedingt anzuraten.

Eine einfache Sortiermethode:

```
public static void sort(int[] feld)
{
  int i, j, l = feld.length - 1, tausch;
  for (i = 1; i <= l-1; i++)
    for (j = i+1; j <= l; j++)
      if (feld[i] >= feld[j])
      {
        tausch = feld[i];
        feld[i] = feld[j];
        feld[j] = tausch;
      }
```

```
}
```

7 Programmeinheiten

Die Java-Programmeinheiten sind Klassen. In einer Klasse können Objekte und Methoden vereinbart werden. Ein ausführbares Programm (eine Applikation) enthält eine main-Methode, an die Parameter von der Komandozeile übergeben werden können. In jeder Klasse kann (zu Testzwecken) eine main-Methode implementiert werden.

```
public class Klasse
{
  public static void main(String arg[])
  {
    int i;
    if (arg.length == -1)
      System.out.println("Eingabe␣erwartet");
    else
    {
      for (i = 0; i <= arg.length - 1; i++)
        System.out.print(arg[i] + "␣");
      System.out.println();
    }
  }
}
```

Methoden für Objekte

```
public class rational
{
  int zaehler, nenner;        // das Objekt rational hat 2 Komponenten
  public int ganzerTeil()
  {
    return this.zaehler / this.nenner;
  }
}

import rational;
import java.io.*;
public class test
```

```
{
  public static void main(String a[])
  {
    rational s = new rational();
    c.zaehler = 5;
    c.nenner = 2;
    System.out.println(c.ganzerTeil());
  }
}
```

Die Superklasse object enthält Methoden, die für alle Arten von Objekten zugänglich sind, z.B.

`a.equals(b)` Vergleich, nicht: `a == b`

`a.clone(b)` kopiert, nicht: `b = a`

`s = a.toString()` ergibt eine (druckbare) Zeichenkette.

Wenn eine Klasse nur Variable (sog Instanzmerkmale), aber keine Methoden enthält, so entspricht dies dem Verbunddatentyp aus C oder Pascal (struct bzw. record).

Objekte werden angelegt, indem eine Variable vom Typ der Klasse deklariert und ihr mit Hilfe des **new**-Operators ein neu erzeugtes Objekt zugewiesen wird:

```
rational c; c = new rational();
```

oder einfacher

```
rational c = new rational();
```

Attribute von Klassen, Methoden und Variablen

public: in der Klasse, in abgeleiteten Klassen und beim Aufruf einer Instanz der Klasse verfügbar;

private: nur innerhalb der Klasse verfügbar;

static: nicht an die Existenz eines konkreten Objekts gebunden (wie oben `ganzerTeil`, `equals` usw.), existiert solange wie die Klasse;

final: unveränderbar (Konstanten).

Klassenvariable (Attribut **static**) werden nur einmal angelegt und existieren immer; sie können von jedermann verändert werden (sie sind so etwas wie globale Variable). Die Bezugnahme geschieht mit

```
klasse.name
```

Nicht-statische Methoden werden beim Erzeugen einer Instanz der Klasse verfügbar:

```
class c {
  public c {};
  public String i() {};
}
```

Aufruf:

```
c nc = new c();
String x;
x = nc.i();
```

8 Programmelemente – Zusammenfassung

- Anweisungen (können auch Deklarationen enthalten),
- Blöcke

```
{anw1;
 anw2;
 dekl1;     //nur im Block gueltig
 anw3;
}
```

Ein Block ist nach außen nur *eine* Anweisung (nach **if** (...) usw. darf nur eine Anweisung stehen, z.B. ein Block).

- Methoden: sie haben Namen, evtl. Parameter und Rückgabewerte; ihre Parameter sind als Referenzen (Zeiger) angelegt (call by reference);
- Klassen: sie enthalten Variable, die den Zustand von Objekten beschreiben, und Methoden, die das Verhalten von Objekten festlegen; Klassen sind schachtelbar: ihre Variablen können den Typ derselben Klasse haben; damit sind rekursive Datenstrukturen (Listen) realisierbar;

Beispiel: Polymome

```
import java.math.*;
import java.io.*;
import java.util.*;
import hgR.*;

public class hgP
{
  public hgR co;
  public int ex;
  public hgP next;

  public hgP()   // Konstrukteur
  {
    this.co = hgR.assign(0);
    this.ex = 0;
    this.next = null;
  }

  public static boolean iszero(hgP p)
  {
    return (!(p instanceof hgP) || hgR.zero(p.co));
  }

  public static void writep(hgP p)
  {
    if(!(p instanceof hgP))
      System.out.print("␣NULL␣");
    else if (hgR.zero(p.co))
```

```
        System.out.print("␣zero␣");
      while (p instanceof hgP)
      {
        if (hgR.positiv(p.co))
          System.out.print("+");
        hgR.aus(p.co);
        System.out.print("x^" + p.ex);
        p = p.next;
      }
      System.out.print("␣␣");
    }
}
```

Wenn eine Klasse importiert wurde, reicht der Klassenname zur Dereferenzierung einer Methode / Variablen aus (nicht der ganze Name ist nötig);

- Pakete: Sammlungen von Klassen; jeder Methoden- oder Variablenname besteht aus drei Teilen:
  - Paketname,
  - Klassen- oder Objektname,
  - Methoden- oder Variablenname, z.B.

    java.lang.Math.sqrt

  Java weiß natürlich, wo es seine eigenen Pakete zu suchen hat, das obige steht in

  .../ java/ classes /java/lang/Math

  evtl. gibt es unter java auch nur eine zip-Datei, die gar nicht ausgepackt werden muß (der Compiler weiß, wie er damit umzugehen hat).
- Applikationen (= Programme): Klassen mit main-Methoden;
- Applets: werden aus einer HTML-Seite heraus aufgerufen.

Eigene Pakete erstellt man, indem man den beteiligten Klassen sagt, daß sie beteiligt sind:

```
package P;
public class A
{
...
}
```

Dies gehört in die Datei A.java, wie ja der Klassenname sagt. Woher kann aber der Compiler eine Information über das Paket P erhalten? Es gibt keine Datei mit diesem Namen. Man kann sein Paket aber auch nicht unter .../classes unterbringen, denn dort hat man keine Schreibrechte. Also: Die Datei A.java gehört ins Verzeichnis

```
./P
```

Dort wird sie gesucht und gefunden. Pakete nutzt man mit

```
import P.*;
```

Die Klassen im aktuellen Verzeichnis . werden zu einem namenlosen Paket zusammengefaßt; deren Methoden können *ohne* Importe verwendet werden. Das Paket java.lang wird automatisch importiert, man hat also automatisch eine Funktion wie Math.sin zur Verfügung.

9 Entwicklung eines Programms

- Quelltext erstellen, z.B T.java,
- Übersetzen des Quelltexts mittels

  javac T.java

  ggf. werden Syntaxfehler angezeigt, die zu korrigieren sind. Wenn keine Fehler vorhanden sind, entsteht die Datei

T.class

dies ist keine ausführbare Datei (kein Maschinencode), sondern „Java-Bytecode“, der auf allen Plattformen durch einen Java-Interpreter ausgeführt werden kann:

java T

10 Dateibehandlung

Dies wollen wir uns nur anhand einiger Beispiele (Rezepte) ansehen, ohne auf die Bedeutung der Funktionen einzugehen.

```
import java.io.*;
public class ausgabe        // Guido Krueger
{
  public static void main(String arg[])
  {
    BufferedWriter f;
    String s;
    if (arg.length == -1)
      System.out.println("Dateiname erwartet");
    else
    {
      try
      {
        f = new BufferedWriter(new FileWriter(arg[0]));
        for (int i = 1; i <= 100; i++)
        {
          s = "Dies ist die " + i + ". Zeile";
          f.write(s);
          f.newLine();
        }
        f.close();
      }
      catch(IOException e)
      {
        System.out.println("Fehler");
      }
    }
  }
}
```

```
import java.io.*;
public class nummer
{
  public static void main(String arg[])
  {
    LineNumberReader f;
    String line;
    try
    {
      f = new LineNumberReader(new FileReader("nummer.java"));
      while ((line = f.readLine()) != null)
      {
        System.out.print(f.getLineNumber() + ": ");
        System.out.println(line);
      }
      f.close();
    }
    catch(IOException e)
    {
      System.out.println("Fehler");
    }
  }
}
```

Zum Schluß noch ein schönes Beispiel:

```
import java.math.BigInteger;
import java.util.*;  // insbes. wird die Klasse Vector importiert,
                     // das ist ein Feld, das wachsen kann

public class factor  // D. Flanagan
{
  protected static Vector table = new Vector();
  static { table.addElement(BigInteger.valueOf(1));

  public static BigInteger factorial(int x)
  {
    if (x < 0)
      throw new IllegalArgumentException("x darf nicht negativ sein");
    for (int size = table.size(); size <= x; size++)
    {
      BigInteger lastfact = (BigInteger)table.elementAt(size - 1);
      BigInteger nextfact = lastfact.multiply(BigInteger.valueOf(size));
      table.addElement(nextfact);
    }
    return (BigInteger)table.elementAt(x);
  }
```

```
  public static void main(Strin[] arg)
  {
    for (int i = 1; i <= 50; i++)
      System.out.println(i + "!␣=␣" + factorial(i));
  }
}
```

11 FAQ: Wann braucht man (), { }, [], ; ?

Eine Anweisung wird mit einem Semikolon beendet.

```
input HUMath.Algebra.*;
```

War das eine Anweisung?

```
int a, s = 0, f[] = {2,3,5,7,11}, p = 1;
```

Dies sind Deklarationsanweisungen: es werden die Variablen `a`, `s`, `p` deklariert, `s` und `p` werden initialisiert; `f` ist ein Feld, das auch sofort initialisiert wird.

```
for (a = 0; a < f.length; a++)
  s = s + f[a];
```

Im Körper einer for-, while-, do-, if-Anweisung wird eine Anweisung ausgeführt.

```
for (a = 0; a < f.length; a++)
  s = s + f[a];
```

Noch einmal: Im Körper einer **for-**, **while-**, **do-**, if-Anweisung wird EINE Anweisung ausgeführt, wenn es mehrere sein sollen, sind sie mit { } zu umschließen.

```
for (a = 0; a < f.length; a++)
{
  s = s + f[a];
  p = p * f[a];
}
```

```
for (a = 0; a < f.length; a++);
  s = s + f[a];
```

Das war falsch, ist aber syntaktisch korrekt!

Die in der for-Schleife auszuführende Anweisung ist diejenige, die vor dem Semikolon steht, also die leere Anweisung: nichts.

Funktionen, Prozeduren: Methoden

Sie haben Parameter, duch Kommata getrennt, deren Type(en) angegeben werden müssen, und ein Resultat, dessen Typ angegeben werden muß:

```
public static double power(double x, int n)
{
  int i;
  double r = 1.0;
  for (i = 1; i <= n; i++)
    r = r * x;
  return r;
}
```

Die Länge eines Feldes `f` erfährt man mit `f.length`, die Länge einer Zeichenkette `s` mit `s.length()`.

Vermeiden Sie bitte `a/b`, wenn `a, b` ganzzahlig sind, und Sie nicht den Effekt haben wollen, den diese Anweisung bewirkt!

Fehlermitteilungen

```
import HUMath.Algebra.*;
public class fehler
{
  public static void main(String[] eingabe)
  {
    double x;
    // Aufruf z.B.: java fehler nein
    B.wl("Sag␣ja␣oder␣nein");
    String s = eingabe[0];
    B.wl("Du␣hast" + s + "␣gesagt");
```

```
    int x, y;
    y++;
    B.wl(y);
  }
}
```

Es erscheint:

```
hg => /home/hg/hgjava/javaneu $ javac fehler.java
fehler.java:12: x is already defined in main(java.lang.String
    int x, y;
        ^
1 error
```

Die Zeilennummer des Fehler wird angezeigt. Wir korrigieren:

```
import HUMath.Algebra.*;
public class fehler
{
  public static void main(String[] eingabe)
  {
    int y;
    y++;
    B.wl(y);
  }
}
```

```
hg => /home/hg/hgjava/javaneu $ javac fehler.java
fehler.java:7: variable y might not have been initialized
    y++;
    ^
1 error
```

Nun ja, das ist auch wahr.