# Softwareentwicklung (Modul Softwaretechnik)
# Grundlagen der Programmiersprache Java
# BEngNT2007

Matthias Krause
Hochschule für Telekommunikation Leipzig
Deutsche Telekom AG

22. September 2008



**Prüfungsbedingungen:** Die Prüfung im Fach Softwaretechnik (Datenbanken, Softwareentwicklung) wird als 90 minütige schriftliche Klausur durchgeführt. Es werden Aufgaben beider Teilmodule zu einem ähnlichen Anteil vertreten sein. Als **Hilfsmittel** ist ein beidseitig beschriebenes Blatt A4 („Spickzettel“) erlaubt.

Dieses Skript soll im Rahmen von Vorlesungen wie „Fortgeschrittene Programmierung 1“ oder „Software-Entwicklung“ benutzt werden und den Hörenden helfen, weniger abschreiben zu müssen. Das Skript ersetzt nicht das Lesen von Originalliteratur und die selbständige kreative Beschäftigung mit der Programmierung.

## 1 Einführung

Anliegen der Vorlesung ist

- das Kennenlernen der Sprache Java mit ihrer Syntax
- unter Nutzung der Kommandozeile und einer IDE (Forte für Java, VisualCafe, Eclipse),
- das Kennenlernen der Arbeit mit dem Java-API,
- Objektorientierung,
- Ausnahmebehandlung,
- Programmierung graphischer Oberflächen (GUI),
- Eventhandling,
- Streams,
- Multithreading,
- Netzwerksprogrammierung (Sockets),
- Datenbankzugriffe

Literatur:

- Handbuch der Javaprogramierung [4],
  html-Version auf `www.javabuch.de`
- Java ist auch eine Insel [6],
  html-Version auf `http://www.galileocomputing.de/openbook/javainsel6/`
- Javatutorial [1]
  und andere Dokumentationen (engl.)
- Java Code Conventions (auf java.sun.com) (engl.)
- Learning Java [5] (engl.)

## 1.1 Die Geschichte von Java

- Anfang der 90er Jahre suchte James Gosling bei Sun nach Werkzeugen zur Entwicklung plattformunabhängiger Software
- 1991 wurde mit der Konzeption von OAK begonnen, später wurde das Projekt in Java umbenannt (Java ... in den USA Ausdruck für Kaffee)
- 1995/1996 kam Java 1.0 heraus, unter der Bezeichnung Java Software Development Kit SDK (Abk. JDK)
- 1998/99 Java 1.2 (jetzt als Java 2 bezeichnet)
- derzeit aktuelle Version ist 1.5 (Java 5),
  na, ja ... diese Aussage ist auch schon wieder veraltet!

## 1.2 Eigenschaften von Java

Java basiert auf Unicode (www.unicode.org), einem 2-Byte-Zeichensatz, bei dem das höherwertige eine Seite, das niederwertige Byte ein Zeichen innerhalb dieser Seite codiert
die 1.Seite entspricht dem Zeichensatz 8859-1, weitere Seiten allen möglichen nationalen Alphabeten
(für Startrek-Fans: klingonisch wurde am 18.9.1997 beantragt, am 21.5.2001 als nicht codierungswürdig abgelehnt („determined to be inappropriate for encoding“))

- Java-Bytecode wird interpretierend in einer virtuellen Maschine (VM) abgearbeitet, Plattformspezifika stecken in der VM
- der Bytecode selbst ist plattformunabhängig
- Objektorientierung, es gibt nichts (!) außerhalb von Klassen (im Gegensatz z.B. zum main() in C++)
- Sicherheitskonzept für Applets (Sandbox-Prinzip ohne System-/Hardware- Zugriffe)
- Threads (Nebenläufigkeit)
- Graphik (plattformunabhängige GUI und Vektorgraphik)
- Netzwerkfähigkeit (Sockets, verteilte Anwendungen)
- modularer Aufbau (Verteilung der Klassen auf Dateien)
- Anbindung anderer Programmiersprachen möglich (nicht in Applets)

Als wesentlicher Nachteil von Java erweist sich der Performanceverlust durch die VM (ca. 1/15 der Geschwindigkeit gegenüber C/C++ bei Solaris). Durch die Plattformunabhängigkeit lassen sich auch nicht alle Systemvorteile ausreizen.
Auf der anderen Seite sind natürlich Plattformunabhängigkeit, Standardisierung, Netzwerkfähigkeit ganz entscheidende Vorteile von Java.

## 1.3 Installation und Werkzeuge

Java erhält man auf den Seiten von Sun (java.sun.com), das SDK, die Doku und das Tutorial liegen auch auf “henry“:

- aktuelles JDK für die entsprechende Plattform (für MS-Windows: Windows Offline Installation, Multi-language, ggf. im Paket mit IDE NetBeans)
- Java Dokumentation (incl. API-Beschreibung):
- Java Tutorial

und Massen anderer Infos
Man installiert das JDK in ein Installationsverzeichnis INST_PATH,
setzt Umgebungsvariablen an entsprechender Stelle (z.B. bei Win9x in der autoexec.bat, ...

```
set PATH = %PATH%;INST_PATH\bin
set CLASSPATH = .;INST_PATH\lib\rt.jar
```

... heutzutage geschieht das meist durch den Installationsprozeß.)

Java-Werkzeuge des JDK sind:

- Compiler: javac
- Interpreter: java
- Appletviewer: appletviewer
- Debugger: jdb
- „Disassembler“: javap
- Dokumentationsgenerator: javadoc
- Archiv-Tool: jar

mit denen man jetzt an der Kommandozeile (oder auch integriert in Editoren wie emacs, WinEdit o.a.) arbeiten kann.

Zur bequemeren Bedienung könnte man jetzt eine IDE wie Forte, NetBeans, Eclipse installieren, die ganz oder in ihren Basisversionen frei sind.

Alternativ (anstelle JDK und Forte) kann man auch eine kommerzielle IDE wie VisualCafe (Symantec), JBuilder (Borland / Inprise), Visual Age (IBM), Oracle JDeveloper o.a. installieren.

Heute ist wohl **Eclipse** die IDE der Wahl.

Arten von Java-Anwendungen/-Programmen

- Text-Anwendungen (Konsolenanwendungen)
  laufen an einer Textkonsole ohne graphische Elemente
- graphische Anwendungen (GUI ... Graphical UserInterface)
  innerhalb graphischer Oberflächen wie X Windows, Microsoft Windows o.ä.
- Applets, das sind spezielle Java-Klassen, die innerhalb eines Web-Browsers laufen
- Servlets, das sind spezielle Java-Klassen, die innerhalb eines Web-Servers mit einer Servlet-Maschine laufen
- Java Beans, Enterprise Java Beans (EJB)
  Service-Klassen, die von anderen Klassen verwendet werden können
- ...

# 2 Elemente der Sprache Java

## 2.1 Struktur eines Java-Programms

Die **Erweiterte Backus-Naur-Form (EBNF)** ist eine formale Metasprache zur Beschreibung von Grammatiken. Eine EBNF definiert Produktionsregeln ( `symbol = symbolfolge ;` ), in denen eine Symbolfolge (rechts vom =) einem Symbol zugeordnet wird. Dieses Symbol kann ein Terminal sein, dann besteht es nur aus Symbolen, die nicht weiter aufgelöst werden müssen, oder aber ein Nichtterminalsymbol, dann stehen auf der rechten Seite auch Symbole, die weiter verfolgt werden müssen.

Einige Beispiele für Symbolfolgen:

```
AB              A gefolgt von B
A|B             A oder B
[A]             nichts oder A
{A}             nichts oder A oder AA oder ...
( A | B ) C     AC oder BC
{ "{" }         nichts oder { oder {{ oder ...
[']']           nichts oder ]
```

| Verwendung | Zeichen |
|---|---|
| Definition | = (auch := oder ::=) |
| Endezeichen einer Produktionsregel (Definition) | ; |
| Logisches Oder | \| |
| Option (0 bis 1) | [ ... ] |
| Optionale Wiederholung (0 bis mehrmals) | { ... } |
| Gruppierung | ( ... ) |
| Anführungszeichen (double quotes) | " ... " |
| Anführungszeichen (single quotes) | ' ... ' |
| Kommentar | (* ... *) |
| Spezielle Sequenz | ? ... ? |
| Ausnahme | - |

In der BNF (Vorstufe zu EBNF) ist es üblich, Nichtterminale in den Produktionsregeln in spitze Klammern einzufügen. In EBNF muß das nicht unbedingt geschehen. Diese Schreibweise wird aber hier bei der (nicht in EBNF formulierten) Syntaxdarstellung für Java verwendet. Die spitze Klammer klammert dann etwas ein, was vom Programmierer einzusetzen ist.
z.B. `System.out.println ( <einString> )`

In der EBNF ist es üblich, nicht weiter auflösbare Symbole in den Produktionsregeln in Anführungszeichen zu schreiben (doppelte, Gänsefüßchen, `"..."`, double quotes ODER einfache, single quotes, `'...'`). Dadurch werden in der EBNF vorkommende Zeichen (wie z.B. `{...}`) von den gleichen in der Sprache beschriebenen Zeichen zu unterscheiden.

```
Java-Quelltext := { kommentar |
                    leerzeichen |
                    sprachelement
                  }

leerzeichen := space | tab | CR | LF | FF

sprachelement := schluesselwort |
                 bezeichner (identifier) |
                 literal |
                 separator |
                 operator

bezeichner := buchstabe [ buchstabenoderzahlenfolge ]

buchstabe := "a" .. "z" | "A" .. "Z" | umlaute | ... | "$" | _

separator := ";" | "," | "{" | "}" | ...
```

**reservierte Wörter (Schlüsselwörter)**

| | | | | |
|---|---|---|---|---|
| abstract | do | implements | package | throw |
| boolean | double | import | private | throws |
| break | else | inner + | protected | transient |
| byte | extends | instanceof | public | try |
| byvalue + | final | int | rest + | var + |
| case | finally | interface | return | void |
| cast + | float | long | short | volatile |
| catch | for | native | static | while |
| char | future + | new | super | byvalue |
| class | generic + | null | switch | |
| const + | goto + | operator + | synchronized | |
| continue | if | outer + | this | |
| default | | | | |

+ ... reserviert, aber noch nicht benutzt

**Bestandteile eines Programms bzw. Java-Files** sind

1. **Package-Anweisung** (optional)

```
1  package packagename; // z.B. package hftl.fpr1.testpackage;
```

2. **Import-Anweisungen** (optional, per default ist immer das Packet `java.lang.*` importiert)

```
3  import packagename.*; //gesamtes Package
4    // z.B. import java.applet.*;
5  import packagename.klassenname; //einzelne Klasse
6    // z.B. import java.applet.Applet
```

   **Importierte Klassen** können im weiteren Quelltext durch Angabe des Namens ohne Angabe des Packages angesprochen werden

   (z.B. `class Test extends Applet`),

   **ohne Import** ist der volle Name incl. Package zu nennen

   (z.B. `class Test extends java.applet.Applet`)

   eine weitere Auswirkung auf den Codeumfang der Klasse hat die Importanweisung NICHT, das Package bzw. die Klasse muß nach wie vor im CLASSPATH zu finden sein! Ein Linken wie bei klassischen Compilersprachen gibt es nicht.

3. **Klassen- und Schnittstellendefinitionen**,

```
8  class MyClass {
9    ...
10 }
11
12 interface MyInterface {
13   ...
14 }
```

   (falls davon max. eine `public` ist, muß die Datei den Namen dieser Klasse tragen `KlassenName.java`.)
   Üblicherweise enthält eine Datei `KlassenName.java` genau eine öffentliche Klasse `KlassenName`.

4. **Kommentare** an beliebigen Stellen

**Die Klassendefinition**

```
klassendefinition :=
[modifier ...] "class" klassenname
                         ["extends" klasse2]
                         ["implements" schnittstellenliste]
"{" klassen-koerper  "}"

modifier := "public" |
            "abstract" |
            "final"
```

In Java gibt es nichts außerhalb von Klassen, selbst das Hauptprogramm `main` ist, im Gegensatz zu C++ und anderen OO-Programmiersprachen, Bestandteil (s)einer Klasse. Falls die Klasse nicht explizit abgeleitet wurde, erbt sie von java.lang.Object.
Von einer als `final` deklarierten Klasse können keine Unterklassen erzeugt werden.

**Kommentare**

```
1  // Standard - Kommentar
2  /*   ....
3       Auskommentierung
4       ....              */
5
6  /**
7       javadoc - Kommentar
8  */
```

Der 2. und 3. Typ sind untereinander nicht schachtelbar (das erste */ beendet den Kommentarbereich), können aber mit //-Kommentaren gemischt werden, welch wiederum durch ein Zeilenende abgeschlossen werden.

**Ein erstes Konsolen-Programm**

```
1 public class HalloWelt {
2   // das bekannteste Programm der Welt
3   public static void main (String[] args) {
4     System.out.println("Hallo Welt");
5   }
6 }
```

und die Übernahme von einem Kommandozeilenparameter:

```
1 public class PrintArg {
2   // druckt das 1. Argument nach dem Programmnamen
3   public static void main (String[] args) {
4     System.out.println("Kommandozeilen-Argument: " + args[0]);
5   }
6 }
```

Kommandozeilen-Argumente-Strings sind ähnlich wie in C Parameter der main-Methode, nur daß man hier kein Pointer of Pointer of Character benötigt, der erste Pointer wird durch das Array, der zweite durch den Typ String ersetzt.

Mit `args.length` erfahren wir die **Länge** des Arrays `args`:

```
    System.out.println("Anzahl : " + args.length);
```

Die Standard-Ausgabe landet auf dem Stream `System.out` (`out` ist eine statische Variable der Klasse `System`, einen print-Befehl, der wie in C zur Sprache direkt gehört, gibt es hier nicht.

Und wenn wir vergessen, daß Argument in der Kommendozeile auch anzugeben,
z.B. so: `java PrintArg firstArgument`
dann fliegt uns natürlich eine Exception um die Ohren. (siehe dazu 2.5.8, S.14 )

**Übung 1** *Erste Programme sollen übersetzt (mit* `javac Name.java`*) und ausgeführt (mit* `java Name`*) werden:*

- *HalloWelt und*
- *PrintArg, drucken Sie auch die Zahl der Kommandozeilenargumente aus! Vergessen Sie auch mal, die Kommandozeilenargumente anzugeben! Was passiert? Lesen Sie genau die Ausgabe!*

## 2.2 Variablen

### 2.2.1 Elementare Datentypen und Strings

| Datentyp | $N_{Bytes}$ | Bereich | Erläuterung |
|---|---|---|---|
| byte<br>short<br>int<br>long | 1<br>2<br>4<br>8 | $-2^{8N-1}....2^{8N-1}-1$ | Ganzzahlen<br>mit Vorzeichen<br>(Zweier-<br>komplement) |
| float<br>double | 4<br>8 | $\pm 1,4*10^{-45}..\pm 3,4*10^{38}$<br>$\pm 4,9*10^{-324}..\pm 1,7*10^{308}$ | Gleitkommazahlen<br>nach IEEE 754 |
| char | 2 | | Unicode |
| boolean | | true, false | Boolscher Typ |

**default** ist immer 0, 0.0, false

Bei der Bestimmung der Min-Max-Werte helfen **Wrapperklassen**,
diese existieren zu allen elementaren Datentypen im Package `java.lang.*` (siehe API-Dokumentation), Name ist jeweils der Typname mit großem Anfangsbuchstaben (aber `Integer` statt `Int`)

```
...
System.out.println(Float.MIN_VALUE + " " + Float.MAX_VALUE);
System.out.println(Double.MIN_VALUE + " " + Double.MAX_VALUE);
...
```

### Die Klasse String

Die Klasse String dient zur Aufbewahrung von Zeichen. Sie beschreibt JAVA-Objekte, die über eine Reihe von Methoden verfügen.
wichtige Operation: + (für Stringverknüpfung/Concatenation)
siehe Java-API-Dokumentation

### 2.2.2 Deklaration

```
[modifier ...] typname variablenname [initialisierung] [ , ... ] ;

modifier = sichtbarkeitsmodifier |
           "static" |
           "final"

sichtbarkeitsmodifier = "public" |
                        "protected" |
                        "private"
```

Ein paar kleine Beispiele:

```
  static double PI = 3.14159;
  int i,j,k; // macht 3 Speicherzellen fuer int-Zahlen
             // Initialisierung VOR erstem Lesen erforderlich!

  MyClass myvar; // macht eine Referenzvariable auf ein Objekt vom Typ MyClass,
                 // enthaelt null
  MyClass myvar2 = new MyClass(); // macht eine Referenzvariable auf ein
                 // Objekt vom Typ MyClass UND dieses Objekt selbst
```

Variablen (außer static) müssen initialisiert werden: durch Einsatz als LValue (LeftValue ... links vom = in einer Zuweisung) vor dem ersten Lesezugriff, was natürlich auch in der Deklaration passieren kann.

**static**-Variable sind Klassenvariablen, die einmal pro Klasse angelegt werden und nicht an die Existenz von Instanzen (Objekten) gebunden sind. Hiermit kann man zum Beispiel die erzeugten Instanzen zählen. Sie werden durch voranstellen des Klassenamens adressiert.
z.B. `Math.sin(Math.PI) // statische Methode und Variable der Klasse Math`

**final**-Variable sind Konstanten, deren Wert nicht mehr verändert werden darf (Vorsicht: wenn es sich um eine Referenz auf ein Objekt handelt, darf der Inhalt des Objektes wohl verändert werden, die Variable/Referenz darf allerding nie auf ein anderes Objekt zeigen)

### 2.2.3 Literale

**Zahlenliterale**

| | |
|---|---|
| Ganzzahlliterale (per default vom Typ int) | 1, -345, 43678235 |
| (mit nachgestelltem l,L sind vom Typ long) | 1l, -345L, 436782358639L |
| in hexadezimaler Darstellung<br>in oktaler Darstellung (mit führender Null) | 0xabf8<br>07, 0126 |
| Gleitkommaliterale (per default vom Typ double)<br>können mit Zehnerpotenz multipliziert werden | 1., 3.14, .342<br>1e5, 4.7E6 |
| (mit nachgestelltem f,F sind vom Typ float) | 1f, .3278F, 1e20f |

**Zeichenliterale (Darstellung in Hochkommata(quotes): 'literal' !!)**

| | |
|---|---|
| gewöhnliches Literal | k, r |
| oktale Darstellung | \377, \6 |
| Unicode-Darstellung | \u0030 |
| Escape-Sequenzen | \n, \b, \r, \t, \f, \', \" |
| String | "zeichenliteralfolge\n" |

**Übung 2** *(Übung zu den Datentypen und Literalen:) Verursachen Sie Über- und Unterlauf von Ganzzahlen und Gleitkommazahlen, wann kommt es zu Ausnahmen, wann zu „neuen" Werten ( Infinity, -Infinity, -0.0, NaN)*

Verwenden Sie Gleitkommazahlen NIE als Laufvariable (oder anderen Integer-Ersatz)!

```
double w1 = 1.23456789;
double w2 = 0.00056789;
double d1 = w1;
double d2 = w1 -2 * w2 + w2 + w2;
System.out.println(" d1 == d2 ? " + (d1 == d2));
// Ausgabe kann false sein (Compilerabhaengig)
```

Vergleichen Sie 1.1 == 1.1f!

## 2.3 Felder in Java

### Deklaration ohne Speicherbereitstellung

Die definierte Variable enthält immer eine Referenz auf ein Array (Feld), niemals das Feld selbst:

```
int i []; // Referenz auf ein eindim. Array
int [] i; // dito
int i[][]; // macht eine Referenz auf ein Array-of-Array
int [][] i; // dito
int [] i []; // dito
```

(mehrdimensionale Felder siehe auch [2, S.58])

### Deklaration mit Speicherbereitstellung und default-Initialisierung

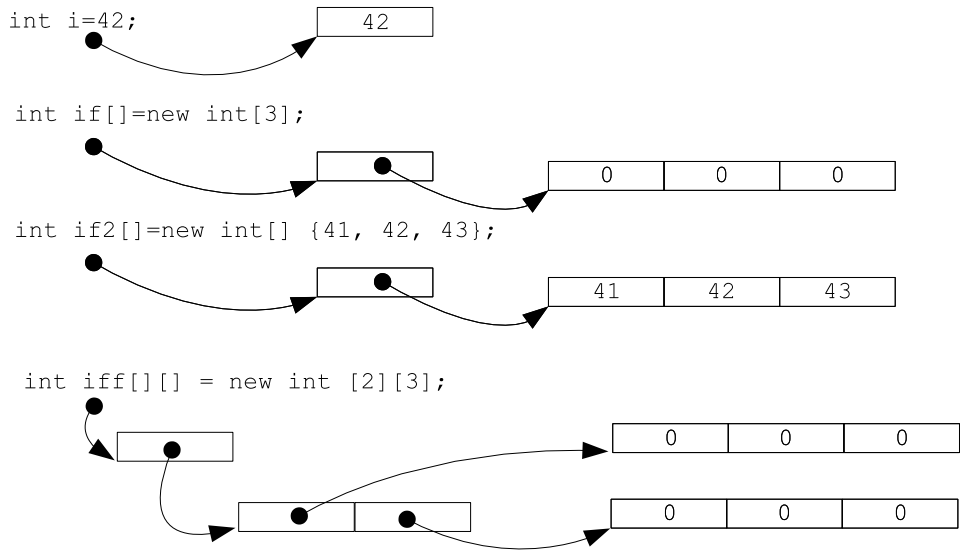
**Elementare Datentypen** mit `static`-Modifier werden mit default (siehe 2.2.1, S.6) initialisiert.
**Felder** werden immer mit default initialisiert.
**Vorsicht:** `int i[4];` ist **nicht erlaubt** (im Gegensatz zu C)
dafür schreibt man:

```
int i[] = new int[4];
int i[][] = new int[5][8];
Classname c[] = new Classname [8];// schafft nur ein Feld von Referenzen
```

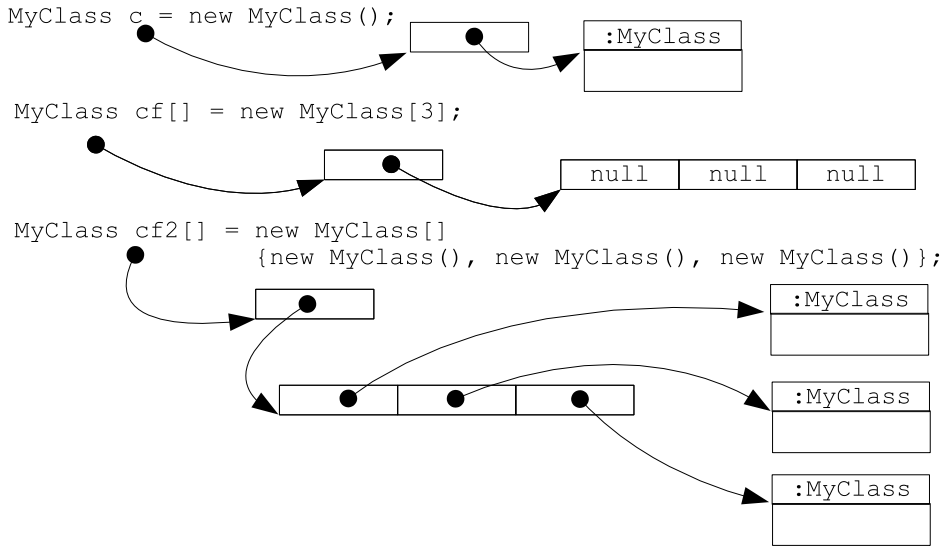
### Deklaration mit Speicherbereitstellung und expliziter Initialisierung

```
int i[][] = new int[][]{{1,2,3},{4,5,6,7},{8,9}};
int i[][] = {{1,2,3},{4,5,6,7},{8,9}};
Classname c[] = {new Classname (Parms),
                 new Classname (Parms) }
```

Elementare Datentypen und ihre Arrays




Abstrakte Datentypen und ihre Arrays




Beispiel zur dynamischen Definition von Feldern anhand des Aufbaus eines Pascalschen Dreiecks:

```
1  class Pascal {
2
3     public static void main (String [] a) {
4         int zeilen= Integer.parseInt(a[0]);
5         int [][] d = new int [zeilen][]; // Initialisierung von d und
6                                          // 1. Dimension von d
7
8  // zuerst wird ein Pascalsches Dreieck (Binomialkoeffizienten) aufgebaut:
9
10        for (int i=0; i<d.length; i++) {
11            d [i] = new int [i+1]; // init. die 2. Dim. fuer ein Element der 1. Dim.
12            d[i][0] = 1;
13            d[i][d[i].length - 1] = 1;
14            for (int j=1; j < d[i].length-1; j++) {
15                d[i][j]=d[i-1][j-1] + d[i-1][j];
16            }
17        }
18
```

```
19 // und dann wird die Speicherstruktur d (das P. Dreieck) zeilenweise ausgedruckt:
20
21       for (int i=0; i<d.length; i++) {
22           for (int j=0; j<d[i].length; j++) {
23               System.out.print(" " + d[i][j]);
24           }
25           System.out.println();
26       }
27   }
28 }
```

Die folgende Übung verdeutlicht, daß die erste Dimension eines zweidimensionalen Feldes eindimensionale Felder beinhaltet

**Übung 3** *Wie sieht das Array d nach Abarbeitung der folgenden Befehle aus?*

```
int d[][];
d    = new int[][]{{1,2,3},{4,5,6,7},{5,4,3,2,1,0}};
d[2] = new int [] {8,9,10,11,12};
```

*Finden Sie die Lösung, ohne das Programm abzuarbeiten! Testen Sie das mit dem Druckteil der obigen Klasse* `Arrays`*!*

## 2.4 Methoden

### 2.4.1 Deklaration

```
methodendekl := [modifier ...] typname [feldklammerfolge] methodenname ([parameterliste])
                          [feldklammerfolge]
                          [throws ausnahmentypliste]
                { anweisungsfolge }  |  ;

modifier := sichtbarkeitsmodifier |
            "static" |
            "final"  |
            "abstract"

sichtbarkeitsmodifier := "public" |
                         "protected" |
                         "private"

feldklammerfolge := "[]" ...

typname := Standard-Java-Typ | Klassenname
```

**statische Methoden** (`static`) sind Klassenmethoden, die einmal pro Klasse angelegt werden und nicht an die Existenz von Instanzen (Objekten) gebunden sind (z.B. die Methode main als Hauptprogramm für die VM oder die Methoden der Klasse Math)

**abstrakte Methoden** (`abstract`) werden noch nicht implementiert, der Programmkörper {anweisungsfolge} kann wegfallen

**finale Methoden** (`final`) dürfen in Unterklassen nicht überschrieben werden

`return [rueckgabewert]` beendet die Abarbeitung, ist zwingend anzugeben in Methoden, deren Typ sich von `void` unterscheidet

Beispiel:

```
1 public class BMI {
2   public static void main (String[] args) {
3     System.out.println(bmi(1.83,85.));
4   }
5   public static double bmi (double groesse, double gewicht){
6       return gewicht / (groesse*groesse);
7   }
```

```
}
```

Beispiel für eine Methode, die vom Typ eines **Array von Strings** (Zeile 6) ist:

```
public class ArrayMethod {
  public static void main (String[] args) {
    String [] st = s();
    System.out.println(st[0] + st[1]);
  }
  public static String [] s () {
      String [] st = { "Hallo ", "Welt"};
      return st;
  }
}
```

**Übung 4** *Skizzieren Sie die Lage der lokalen Variablen* `st` *(in* `main(...)` *und* `s(...)` *und des Arrays of String, welches außerhalb der Methoden angelegt wird! (in Anlehnung an die Skizzen zu Arrays im Speicher (S.9), der Stackbereich der Methoden kann als Kasten symbolisiert werden, der wiederum Variablen enthalten kann).*

**Übung 5** *Entwickeln Sie eine Testmethode*

*`public static int countPairs (double [][] d , double epsilon)`,*

*die auf einer 2-dimensionalen Datenstruktur (Aufrufparameter der Methode) prüft, ob die Bedingung*

$$|d[i][j] - d[i][j+1]| < epsilon$$

*erfüllt ist, d.h., benachbarte Elemente haben eine Differenz, deren Betrag kleiner als epsilon ist. Diese Paare sollen gezählt, ihre Anzahl zurückgegeben werden!*

### 2.4.2 Überladen (to overload) von Methoden

Als Überladen bezeichnet man die Koexistenz gleichnamiger Methoden mit unterschiedlichen Parameterlisten, ein unterschiedlicher Rückgabetyp ist nicht ausreichend.
die Auswahl der Methode beim Aufruf erfolgt anhand der aktuellen Parameterliste.

**VORSICHT**: nicht mit **Überschreiben** (to override) von Methoden (=Polymorphie, bei Vererbung) verwechseln!

```
/*
   Typen der Literale: 3L 4l ... long-Literale
                       1 2 5  ... int-Literale
*/

public class Ueberladen {
  public static void main (String[] args) {
      // max-Rufe mit int- bzw. long-Literalen
    System.out.println(max(1,2)); // int, int
    System.out.println(max(1l,2)); // long, int
    System.out.println(max(1,2l)); // int, long
    System.out.println(max(1l,2l)); // long, long
  }
  public static long max (long z1, long z2){
    System.out.println("long max(long "+z1+", long "+z2+")");
    return z1>z2?z1:z2;
  }
  public static long max (long z1, int z2){
    System.out.println("long max(long "+z1+", int "+z2+")");
    return z1>z2?z1:z2;
  }
  public static long max (int z1, long z2){
    System.out.println("long max(int "+z1+", long "+z2+")");
    return z1>z2?z1:z2;
```

```
25    }
26    public static int max (int z1, int z2){
27      System.out.println("int max(int "+z1+", int "+z2+")");
28      return z1>z2?z1:z2;
29    }
30  }
```

## 2.5 Kontrollfluß

| | |
|---|---|
| Verzweigungen | if, switch |
| Schleifen | do, while, for |
| Ausnahmbehandlung | try, catch, finally |

### 2.5.1 Die Verzweigung (if else)

```
if
  anweisung1
[else
  anweisung2]

anweisungX := anweisung |
              verbundanweisung

verbundanweisung := { anweisungsfolge }

anweisungsfolge := anweisung [ anweisung ... ]
```

else bezieht sich immer auf das letzte davorstehende else-lose if

### 2.5.2 Mehrfach-Verzweigung (switch)

```
switch (ganzzahlausdruck) {
  case konst1: anweisung1 break;
  case konst2: anweisung2 break;
  ...
  default: anweisungn break;
}

konstX := Ausdruck, der zur Compile-Zeit bekannt ist,
          also ggf. auch eine final-Variable

anweisungX := anweisungsfolge := s.o.}
```

die AnweisungX, für die ganzzahlausdruck == konstX gilt, wird abgearbeitet.

ACHTUNG: bei fehlenden break wird die passende anweisungX und danach alle folgenden bis zum Ende bzw. zum nächsten break abgearbeitet, somit ist switch de facto ein berechneter Sprung

### 2.5.3 Die abweisende Schleife (while)

```
while (bool_ausdruck)
  anweisung

anweisung := s.o.
```

### 2.5.4 Die nicht-abweisende Schleife (do while)

```
do  anweisung
while (bool_ausdruck)

anweisung := s.o.
```

### 2.5.5 Die Zählschleife (for)

```
for (initialisierung; bedingung; ausdruck)
  anweisung

bedingung := boolscher ausdruck
```

die Zählschleife kann auch folgendermaßen dargestellt werden:

```
initialisierung;
while (bedingung) {
  anweisung
  ausdruck;
}
```

**Übung 6** *Entwickeln Sie ein Programm in Anlehnung an Übung 1, das in einer Schleife alle Kommandozeilenargumente ausdruckt!*

**Übung 7** *Entwickeln Sie ein Programm, das die Fakultät einer an der Kommandozeile als Argument übergebenen Zahl ausgibt! Verwenden Sie einen rekursiven Algorithmus in der Methode*
`public static int fakultaet(int zahl) {...} !`

**Übung 8** *Entwickeln Sie ein Programm, welches die (positive) Quadratwurzel eines als Kommandozeilenargument übergebenen nichtnegativen Zahl berechnet und ausgibt! Die Berechnung soll intern in einer Methode*
`public static double sqrtHeron(double zahl, double epsilon) {...}`
*realisiert werden und nach dem Verfahren nach Heron stattfinden, das durch folgende Kommentar skizziert wird:*

```
// Wurzelziehen aus positiven Zahlen mit dem Verfahren nach Heron
//  W = Wurzel aus zahl
//  zahl < 0 : keine Lösung
//  zahl = 0 : W = 0
//  zahl > 0 : Zahlenreihe W(0), W(1), ..., W(n), W(n+1)
//               W(0) = Zahl
//               W(n+1) = 1/2 (W(n) + zahl/W(n))
//               konvergiert gegen W
//  ...
//  geeignete Schleife mit geeigneter Abbruchbedingung fuer die Naeherung (Epsilon)
```

**Übung 9** *Schreiben Sie eine Programm, in dessen main-Methode je ein Feld von x- und y-Werten berechnet und ausgegeben wird. Die y-Werte sollen z.B. den Sinus der x-Werte bilden, x soll in einem definierten Intervall liegen (z.B. $0 \ldots 2\pi$)! Die Felder und Konfigurationsvariablen (Intervallgrenzen, Punktezahl) sollen in sinnvoll benannten Klassenvariablen liegen!*

**Übung 10** *Entwickeln Sie eine Methode*
`public static double [][]multiplyMatrices(double [][] a, double [][] b) {...}`
*welche die Ergebnismatrix einer Matrizenmultiplikation $a \times b$ zurückgibt. Die Elemente der Ergebnismatrix $r$ berechnen sich zu*

$$r_{ij} = \sum_{n=0}^{Ba-1} a_{in} b_{nj}$$

*Gehen Sie davon auf, daß die Arrays* `a` *und* `b` *rechteckig sind und die Breite von* `a` *(Ba) gleich der Höhe von* `b` *(Hb) ist.*

### 2.5.6 Die erweiterte for-Schleife (Enhanced for statement)

Stellt im Prinzip eine foreach-Schleife dar.

Syntax (EBNF):

```
for "(" [VariableModifiers] Type Identifier ":" Expression ")" Statement

Expression := einFeld | Iterable
```

Ein Beispiel:

```
int [] feld = {1,2,3,4,5};
for (int i : feld) System.out.println(i);
```

### 2.5.7 Verlassen von Schleifen mit break und continue

**break** beendet die Ausführung einer Schleife

**continue** beendet die Ausführung der aktuellen Schleifen-Anweisung und geht zur anstehenden Prüfung für den nächsten Durchlauf

Syntax:

```
[label1:] schleife1 {
  ...
  [label2:] schleife2 {
    ...
    break | continue [label]
    ...
  }
  ...
}
```

die Angabe des Parameters label bewirkt, daß das Verlassen sich auf die Schleifenanweisung der mit label markierten Schleife bezieht

Ein Beispiel:

```
int i1=1;
int zeilen=5;
eins: while (true) {
  int i2=1;
  zwei: while (true) {
    System.out.print(" " + i2++);
    if (i2 > i1) {
      i1++;
      System.out.println();
      if (i1 > zeilen) break eins;
      break;
    }
  }
}
```

**Übung 11** *Was gibt das Programm aus?*

### 2.5.8 Ausnahmebehandlung

Ausnahmen sind Klassen bzw. Objekte, die das Auftreten von Fehlern anzeigen, sie sind von der Klasse Exception abgeleitet

```
java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      ...
```

und werden vom Progamm bei Bedarf mittels `throw` geworfen.

**Ausnahmen** (außer Runtime Exceptions und deren Kindklassen) müssen

- mit folgendem Konstrukt (`try-catch`) behandelt (mit `catch` gefangen) werden:

```
try {
    anweisungsfolge0
             // die eigentlichen Anweisungen,
             // die ev. Exceptions werfen koennen
}
catch (ausnahmetyp1 aname1) {
    anweisungsfolge1
}
... // ggf. weitere catch Anweisungen

finally {
    anweisungsfolgeN // wird immer durchlaufen
}

anweisungsfolgeX := anweisungsfolge
                 := anweisung [ anweisung ... ]
```

- oder aber von der Methode **weitergeworfen** werden, was mittels *throws* in der Kopfzeile kundgetan werden muß:

```
void setWert(int i) throws WertInvalid { ... }
```

Bei **RuntimeException** kann auf Ausnahmebehandlung verzichtet werden, was natürlich dann die unschöne Folge hat, daß die Ausnahme auf die Konsole purzelt und das Programm ggf. beendet wird!

**kleines Beispiel zu Ausnahmen**

1. Auftreten von RuntimeExceptions des Systems
2. Abfangen dieser Ausnahmen (allgemein als Exceptions und speziell nach Typ)
3. eigene Ausnahmeklassen, abfangen oder weiterwerfen

Das Programm

```
1 class Except1 {
2   // druckt das 1. Argument nach dem Programmnamen
3   static void main (String[] args) {
4     System.out.println("1.Kommandozeilen-Argument: " + args[0]);
5     System.out.println("nach moeglichen Eingabefehlern");
6   }
7 }
```

wirft eine Ausnahme vom Typ `ArrayIndexOutOfBoundsException`, wenn kein Kommandozeilenargument da ist, diese Exception ist eine `Runtime Exception`, die nicht unbedingt abgefangen werden muß.

Programmodifikation zum Abfangen aller Ausnahmen:

```
  ...
    try {
      System.out.println("Kommandozeilen-Argument: " + args[0]);
```

```
    }
    catch (Exception e){
      System.out.println("Schade: " + e);
    }
  ...
```

Parsen von Strings mit Hilfe der Wrapperklassen Integer, Double, ...

```
int i = Integer.parseInt(String s);
double d = Double.parseDouble(String s);
...
```

Programmodifikation zum Abfangen der Ausnahmen vom Typ `ArrayIndexOutOfBoundsException`:

```
...
    catch (ArrayIndexOutOfBoundsException e){
...
```

Erzeugen zusätzlicher Ausnahmen durch das Umwandeln der Strings in Zahlen und gezieltes Abfangen

```
class Except2 {
  // parst als int und druckt die zwei Argumente nach dem Programmnamen
  static void main (String[] args) {
    int i0,i1;
    System.out.println("1.Kommandozeilen-Argument: " + args[0]);
    i0 = Integer.parseInt(args[0]);
    System.out.println("2.Kommandozeilen-Argument: " + args[1]);
    i1 = Integer.parseInt(args[1]);
    System.out.println("nach moeglichen Eingabefehlern");
  }
}
```

Abfangen oder Nichtabfangen der Ausnahmen

```
class Except2 {
  // druckt das 1. Argument nach dem Programmnamen
  static void main (String[] args) {
    int i0,i1;
    try {
      System.out.println("1.Kommandozeilen-Argument: " + args[0]);
      i0 = Integer.parseInt(args[0]);
      System.out.println("2.Kommandozeilen-Argument: " + args[1]);
      i1 = Integer.parseInt(args[1]);
      System.out.println("end try");
    }
    catch (NumberFormatException e){
      System.out.println("NumberFormat: " + e);
    }
    // catch (ArrayIndexOutOfBoundsException e){
    //   System.out.println("ArrayOut...: " + e);
    // }
    finally {
      System.out.println("finally");
      return; //moegliche Aktion, da keine sinnvollen Parameter da sind
    }
    System.out.println("ausserhalb try catch finally");
  }
}
```

die erste (!) Ausnahme im try - Block bewirkt das Verlassen des Blockes und das Abarbeiten des entsprechenden ersten passenden catch - Blockes , nicht gefangene Ausnahmen werden weitergeworfen

**Der try/catch - Mechanismus**

- sichert den Stack und die Register zum Zeitpunkt des try - Statements,
- der try - Block wird abgearbeitet,
  - und bei Auftreten der ersten Ausnahme verlassen,
  - es wird nach einer passenden catch - Methode unter schrittweisem Zurückrollen des Stacks gesucht
  - und die jeweiligen finally - Anweisungen abgearbeitet.
  - der erste gefundene catch-Zweig wird abgearbeitet, Stack und Register zum Zeitpunkt des try wiederhergestellt

Fängt eine Methode Ausnahmen außer RuntimeExceptions nicht, so ist das im Methodenkopf über eine **throws**-Klausel zu vermerken.

```
... methodname () throws exceptiontype { ... body ... }
```

Ein **Beispiel für das bedingte Werfen einer eigenen Exception-Klasse** aus einer Methode heraus. Die Methode `static int indDiv (int z, int n)` soll das Ergebnis der Division $n/z$ zurückgeben, wenn diese ohne Rest erfolgt, ansonsten soll sie eine `IntDivHasRestException` werfen.

```
1  class MathUtilities {
2    // die Divisionsmethode
3    static int intDiv (int z, int n) throws IntDivHasRestException {
4      int Rest;
5      if ((Rest = z % n) == 0) return (z / n);
6      else throw new IntDivHasRestException("Rest " + Rest + " vorhanden");
7    }
8    // das Hauptprogramm mit Ruf der Divisionsmethode und Exceptionhandling:
9    public static void main(String args[]) {
10     // hier ohne Parameter-Behandlung
11     int Z=Integer.parseInt(args[0]);
12     int N=Integer.parseInt(args[1]);
13     //************************************
14     try {
15       System.out.println("" + Z + " / " + N + " = " + intDiv (Z,N));
16     }
17     catch(IntDivHasRestException e) {
18       System.out.println("Div: " + e);
19     }
20     catch(ArrayIndexOutOfBoundsException e) {
21       System.out.println("Index: " + e);
22     }
23     finally {
24       System.out.println("finally-Zweig");
25     }
26     //************************************
27   }
28 }
```

Die nächsten Zeilen zeigen die Definition der Exception-Klasse, was natürlich eigentlich nur Sinn hat, wenn die Instantiierung NICHT 1:1 an `Exception` weitergereicht wird, ansonsten könnte gleich mit `Exception` gearbeitet werden!

```
30 class IntDivHasRestException extends Exception {
31   IntDivHasRestException(String s){super(s);}
32   IntDivHasRestException(){super();}
33   //zusaetzlich koennen weitere Aktionen realisiert werden
34 }
```

Ein **zusammenfassendes Beispiel**, welches **Objektorientierung und Ausnahmebehandlung** behandelt, soll in **Übung 17** auf S.24 entwickelt werden

## 2.6 Operatoren

**Übung 12** *Wiederholen Sie die Wirkungweise von Operatoren! Lesen Sie dazu im Javatutorial den Abschnitt Operatoren, die gute Zusammenfassung dort sollten Sie sich ausdrucken (Java Tutorial [1, file://./opsummary.html])!*

**Vorrang (precedence) der Operatoren** (ähnlich C(++))

```
postfix operators      [] . (params) expr++ expr--
unary operators        ++expr --expr +expr -expr ~ !
creation or cast       new (type)expr
multiplicative         * / %
additive               + -
shift                  << >> >>>
relational             < > <= >= instanceof
equality               == !=
bitwise AND            &
bitwise exclusive OR   ^
bitwise inclusive OR   |
logical AND            &&
logical OR             ||
conditional            ?:
assignment             = += -= *= /= %= &= ^= |= <<= >>= >>>=
```

**Übung 13** *Was druckt die folgende Sequenz aus? (Finden Sie zunächst die Lösung, ohne das Programm abzuarbeiten! Testen Sie danach!)*

```
1    int i,j;
2    i=8; j=1;
3    System.out.println( i==8 || j++==1);
4    System.out.println( j );
5    i=8; j=1;
6    System.out.println( i==8 | j++==1);
7    System.out.println( j );
8    i=8; j=1;
9    System.out.println( ++i==8 || j==3);
10   System.out.println( i );
11   i=8; j=1;
12   System.out.println( i++==8 || j==3);
13   System.out.println( i );
```

Ein Beispiel für Operatoren:

```
1  class ParseInt{
2    static void main(String [] a){
3      int b=Integer.parseInt(a[0]);
4      // b wird bitweise abgetastet
5      int m = Integer.MIN_VALUE; //MSB ist 1
6      for (int i=0; i<32; i++){
7        System.out.print(( b & m >>> i )== 0 ? "0" : "X");
8      //System.out.print(((b &(m >>> i))== 0)? "0" : "X");
9      }
10   }
11 }
```

**Übung 14** *Analysieren Sie das Beispiel! Was passiert? Rufen Sie sich Ihr Wissen über Zahlendarstellungen (wie Zweierkomplement) in Erinnerung!*

Mit den bitweisen Operatoren (`& |`) können u.U. nur Daten des gleichen Datentyps verknüpft werden. In den Wrapperklassen der elementaren Datentypen gibt es Transformationsfunktionen, die zwischen den Datentypen unter Beibehaltung des Bitmusters transformieren,
z.B. `long static Double.doubleToLongBits(double zahl)`.

**Übung 15** *Schreiben Sie ein Programm, welches das Bitmuster einer als Kommandozeilenparameter übergebenen Gleitpunktzahl-Zahl (float oder auch double) ausgibt! Realisieren Sie intern die Analyse (einer float-Zahl) mittels einer Methode*
*`public static String getFloatBitsAsString(float zahl){...}`*
*Der Programmruf soll z.B. so aussehen:*

```
C:\tools> java PFloat 1.00
00111111100000000000000000000000
```

*(Nutzen Sie ggf. die Methoden*
*`Float.floatToIntBits(float value)` oder `Double.doubleToLongBits(double value)` !)*

# 3 Objektorientierung

## 3.1 Grundbegriffe

Abstract Data Types (ADT) bestehen aus

- Daten
- Methoden
- Regelung von Zugriffsrechten

**Klasse:** Bauvorschrift für einen ADT
Vererbung (Klassen und Unterklassen)
**Exemplare von Klassen:** Instanzen, Objekte
**Abstrakte Klasse:** Klasse, von der keine Objekte instantiiert werden können
**Schnittstelle:** Spezialform der abstrakten Klasse, die nur Methodendeklarationen ohne Implementierung und finale Variablen (Konstanten) enthält
**Nachricht, Message:** Ruf einer Methode von B durch A → A sendet Message an B
Kapselung, Information Hiding, Sichtbarkeit

## 3.2 Sichtbarkeit

Bei der Frage der Sichtbarkeit geht es immer darum, ob für ein Objekt einer Klasse oder die Klasse selbst (in statischen Bestandteilen) auf einem anderen Objekt der gleichen oder einer anderen Klasse oder dieser Klasse selbst eine Information (Variable oder Methode) sichtbar oder nicht ist. Dabei schließt das Sehen lesen und ggf. ändern ein.

| Modifier | in gleicher Klasse | in einer Unterklasse | im gleichen Package | überall |
|---|---|---|---|---|
| `private` | sichtbar | | | |
| `protected` | sichtbar | sichtbar(*) | sichtbar | |
| `public` | sichtbar | sichtbar | sichtbar | sichtbar |
| ohne (Package) | sichtbar | | sichtbar | |

(*) sichtbar sind nur die in der Oberklasse definierten Methoden auf Instanzen der eigenen Klasse, nicht auf den Instanzen der Oberklasse!

Gedanken zum Einsatz der Sichtbarkeitsmodifier finden sich auf Seite 23.

## 3.3 Klassen in Java

Klassendefinition (EBNF):

```
modifier "class" klassenname
                   ["extends"     oberklassenname]
                   ["implements"  interfaceliste ]
"{" classbody "}"

interfaceliste = interfacename { "," interfacename }
```

Erzeugen von Exemplaren (Instanzen) (Quellcode, kein EBNF):

```
// Deklaration:
klassenname instanzname;

// Instanziierung:
instanzname = new klassenname (parameterliste);

// Deklaration und Instantiierung:
klassenname instanzname = new klassenname (parameterliste);
```

Ein einfachen Beispiel einer Klasse **Person** mit Daten (Attributen) und Methoden:

```
1  public class Person {
2    // eine Eigenschaft, Variable der Klasse
3    public String name  = "N.N.";
4
5    // main hat hier keine eigenstaendige Aufgabe, da Person aus anderen
6    // Klassen heraus genutzt wird
7    public static void main (String [] a) {
8        System.out.println("Ich bin doch nur eine arme Klasse");
9    }
10
11   // Konstruktoren
12   public Person () { }
13   public Person (String n) {
14     this.name = n;
15   }
16
17   // eine Methode zum Drucken von Informationen
18   public void drucke (){
19     System.out.println("*****************************************");
20     System.out.println("Name: " + name);
21     System.out.println("Ich bin ein " + this.getClass().getName() );
22   }
23 }
```

Ein **Konstruktor** ist eine Methode mit dem Namen der Klasse ohne expliziten Rückgabewert, die einmal zur Instantiierung eines Objektes abgearbeitet wird.

Die Definition des Konstruktors, dem ein String-Parameter übergeben wird, hat zur Folge, daß der parameterlose Konstruktor, der ansonsten ohne Erwähnung existiert, auch definiert werden muß.

Die Instantiierung / Nachrichten von `Person`-Objekten aus der Klasse Persontest heraus:

```
1  class PersonTest {
2    static void main (String [] a) {
3      Person m1 = new Person ("Peterle");
4      m1.drucke();
5    }
6  }
```

Modifizieren Sie, indem Sie `private` vor `drucke()` schreiben! Was passiert? Warum?

**Aufbau einer Klassenhierarchie mit Person als Oberklasse** (Vererbung, Polymorphie (Überschreiben) von Methoden):

Die Klasse `Mitarbeiter` wird von `Person` abgeleitet, sie erbt alle Eigenschaften (Variablen und Methoden), hat aber darüber hinaus eigene Eigenschaften (`double gehalt;`)

```
class Mitarbeiter extends Person {
  public double gehalt = 0;

  public Mitarbeiter (String n) {
    super (n);
  }
  public Mitarbeiter (String n, double g) {
    super (n);
    this.gehalt = g;
  }
  public void drucke (){
    super.drucke();
    System.out.println("Gehalt: " + gehalt);
  }
}
```

Zu beachten ist hier der direkte Aufruf der entsprechenden in der Oberklasse definierten Methode durch das Schlüsselwort **super**! Damit können am Anfang der Methode explizit die Fähigkeiten der Methode der Oberklasse eingebunden werden, womit man sich z.B. in `drucke()` die nochmalige Behandlung der Eigenschaften, die es schon in der Oberklasse gibt, sparen kann.

Die Klasse `Chef` wiederum wird von `Mitarbeiter` abgeleitet, sie hat zusätzlich die Eigenschaft `String Bereich;` ...

```
class Chef extends Mitarbeiter {
  public String Bereich = "N.N.";
  public Chef(String n, double g, String b) {
    super (n, g);
    this.Bereich = b;
  }
  public Chef(String n, double g) {
    super (n, g);
  }

  public void drucke() {
    super.drucke();
    System.out.println("Ich bin Chef von " + Bereich);
  }
}
```

... während der `Kunde`, der von `Person` abgeleitet ist, als Eigenschaft über eine `Kundennummer` verfügt:

```
class Kunde extends Person {
  public String KdNummer = "--";

  public Kunde (String n) {
    super (n);
  }
  public Kunde (String n, String KdNr) {
    super (n);
    this.KdNummer = KdNr;
  }
  public void drucke (){
    super.drucke();
```

```
13      System.out.println("Kundennummer: " + KdNummer);
14    }
15  }
```

... mit Aufrufmöglichkeiten, die aus verschiedenen Klassen heraus stattfinden können:

```
1   class PersonTest {
2     static void main (String [] a) {
3       Chef c1 = new Chef ("BigBoss",25000,"Alles");
4       Chef c2 = new Chef ("BrauchtePosten",15000);
5       Mitarbeiter m1 = new Mitarbeiter ("Mallocher",3000);
6
7       Person [] p = new Person [4];
8
9       p[0] = c1;
10      p[1] = c2;
11      p[2] = m1;
12      p[3] = new Kunde("Knollo Bertius","1-3-432");
13
14      for (int i=0; i<p.length; i++) {
15          System.out.println("-------------------------------------------------");
16          System.out.println(p[i].toString());
17          p[i].drucke(); // das ist spaete Bindung
18      }
19    }
20  }
```

```
21  class AnotherClass {
22    static void main (String [] a) {
23      Person [] p = { new Chef ("BigBoss",25000,"Alles"),
24                      new Chef ("BrauchtePosten",15000),
25                      new Mitarbeiter ("Mallocher",3000)};
26
27      for (int i=0; i<p.length; i++) p[i].drucke();
28    }
29  }
```

Der erfolgende Ruf der Methode `drucke()` auf den `Person`-Objekten in Zeile 17 und 26 legt noch nicht fest, welche Methode zur Laufzeit gerufen wird (die von `Mitarbeiter`, `Chef`,...). So etwas wird als **späte Bindung** bezeichnet.

Eine Möglichkeit, eine Kopie eines Objektes anzulegen, ergibt sich mit der Implementation des Interfaces **Clonable** und der Methode **clone()** von Object:

```
1   public class Person implements Cloneable {
2     ...
3     public Person clonen () {
4         Person p = null;
5         try {
6              p = (Person)this.clone();
7         }
8         catch (Exception e) { System.out.println("Fehler beim clonen: " + e);}
9         return p;
10    }
11    ...
12
13    public static void main (String [] args){
14      ...
15      Mitarbeiter m1 = new Mitarbeiter ("Mallocher",3000);
```

```
16      Mitarbeiter m2;
17      m2 = m1.clonen();
18      ...
19    }
20
21  }
```

### Gedanken zu Sichtbarkeit und Verstecken (Hiding) von Klassenbestandteilen

Der Tabelle zur Sichtbarkeit (S.19) kann man entnehmen, welche Informationen/ Bestandteile (d.h. Variablen und Methoden) einer Klasse für wen sichtbar sind. Die folgenden Ausführungen sollen helfen, den passenden Sichtbarkeitsmodifier auszuwählen:

- Soll eine Klasse in einem API / einer Bibliothek von anderen Klassen genutzt werden, so müssen die entsprechenden Methoden und natürlich die Klasse selbst als **public** deklariert werden, da die rufende Klasse im Allgemeinen in einem anderen Package liegen und nicht von der zu nutzenden Klasse abgeleitet sein wird!
- Sollen Informationen dagegen **nur** innerhalb der eigenen Klasse genutzt werden, werden sie als **private** deklariert. Dann kann zwar jedes Objekt die Informationen auf jedem Objekt der Klasse sehen, aber der Quelltext, der genau das realisiert, liegt ja in der Hand des Programmierers der Klasse, so daß nichts Ungewolltes passieren kann. Abgeleitete Klassen sehen diese Informationen auch nicht, so daß sie ohne jegliche Konflikte Informationen mit gleichen Namen deklarieren können.
- Liegen verschiedene Klassen innerhalb eines Packages, so sehen sie gegenseitig alle Informationen mit der **Package-Sichtbarkeit** (d.h. **ohne** Sichtbarkeitsmodifier). Somit können sie sich gegenseitig Dienste anbieten, die von niemandem außerhalb des Packages gesehen und genutzt werden können.
- Eine Sonderrolle spielt **protected**, diese Informationen werden prinzipiell in Kindklassen gesehen, sie werden somit auch vererbt, so daß sie in den Kindklassen genutzt werden können. D.h., es gibt die in der Oberklasse deklarierten und definierten Informationen in der Kindklasse! Allerdings kann die Kindklasse die mit `protected` geschützten Informationen auf der Oberklasse **nicht** sehen! Warum ist das so? Im Zuge der Programmentwicklung kann man eine Klasse schaffen, die Ausgangspunkt für Vererbung ist. Die Kindklassen sehen die Variablen und Methoden und können sie innerhalb ihrer Klassen uneingeschränkt nutzen, damit wird das Wissen weitergegeben. Andererseits weiß man, daß die Kindklassen die in Objekten der Oberklasse enthaltenen Informationen nicht sehen und somit nicht "ausspionieren" können, denn das könnte, da man ja den Quelltext der Kindklassen nicht beeinflussen kann, geschehen.

### Einsatz von getter/setter-Methoden

Eine in der Welt der Objektorientierung oft genutzte und kontrovers diskutierte Methode ist das Verstecken der konkreten Implementation / Ablage von Informationen und der Zugriff auf diese Informationen mittels **getter/setter-Methoden**, wie es z.B. bei der Gestaltung von Java-Beans üblich ist. Dabei wird eine Information (z.B. eine Variable `name`) als `private` deklariert, so daß sie außerhalb der Klasse nicht gesehen wird. Der lesende bzw. schreibende Zugriff erfolgt über öffentliche Methoden (hier `getName` und `setName`):

```
1  ...
2    private MyType name;
3    public MyType getName() {
4      return name;
5    }
6    public void setName(MyType name) {
7      this.name = name;
8    }
9  ...
```

An diesem Beispiel (MyType könnte String sein) ist der weitergehende Sinn des Versteckens der Variable `name` nicht ersichtlich, die Variable könnte durchaus `public` sein, auf die Methoden könnte verzichtet werden. Anders wäre das, wenn die interne Implementation der eigentlichen Information von den außen sichtbaren Typen der getter/setter-Methoden abweicht, was das folgende Beispiel verdeutlichen soll:

```
...
  private String name; // z.B. name = "Lorenz,Peter"
  public String getVorname() {
    // gibt den Teil nach dem Komma zurueck, also "Peter"
    String [] n = name.split(",");
    if (n.length == 2) return n[1];
    else               return "";
  }
  public void setVorname(String vorname) {
    // Aendert den Teil nach dem Komma
    // keine Kommas eingeben!!!
    this.name = getNachname() + "," + vorname;
  }
  public String getNachname() {
    // gibt den Teil vor dem Komma zurueck, also "Peter"
    String [] n = name.split(",");
    if (n.length == 2) return n[0];
    else               return "";
  }
  public void setNachname(String nachname) {
    // Aendert den Teil vor dem Komma
    // keine Kommas eingeben!!!
    this.name = nachname + "," + getNachname();
  }
...
```

Darüber, wie Vorname und Nachname in der Klasse abgelegt sind, besteht außerhalb der Klasse keinerlei Information, ohne Veränderung der Signatur der getter- und setter-Methoden kann man nun die interne Ablage der Daten ändern! Natürlich müßten die setter-Methoden so gestaltet sein, daß keine Kommas übergeben werden können, die die innere Datenstruktur stören würden, bei Vorhandensein eines Kommas könnte eine Exception geworfen werden! Oder ein eingegebenes Komma müßte im String speziell markiert sein!

**Übung 16** *Schreiben Sie eine Klasse Punkt mit zunächst zwei (private) double Variablen für x- und y-Wert und entsprechenden getter- und setter-Methoden (getX(), setX(...), ...). Als Alternative könnten auch zunächst public Variablen ohne getter- und setter-Methoden verwendet werden.*
*In einer main-Methode soll ein Feld von Objekten dieser Klasse instantiiert werden, die y-Werte sollen z.B. den Sinus der x-Werte bilden, x soll in einem definierten Intervall liegen (z.B. 0 ...2π), die Wertepaare sollen ausgedruckt werden (als Modifikation der Aufgabe 9, S.13)!*

**Übung 17** *Entwickeln Sie eine Testmethode*
`public static MyDimension testMatrix(double [][] d) throws ThisIsNotAMatrixException,`
*die auf einer 2-dimensionalen Datenstruktur (Aufrufparameter der Methode) prüft, ob es sich dabei um ein Rechteck (also im mathematischen Sinne um eine Matrix) handelt!*
*Wenn nicht, soll die Methode eine* `ThisIsNotAMatrixException` *werfen, wenn ja, soll ein Objekt einer (ggf. zu schaffenden) Klasse zurückgegeben werden, welches die Eigenschaften* `width` *und* `height` *hat, denen die Breite und Höhe der Matrix übergeben wird!*
*Geben Sie auch den Code dieser* `MyDimension`*-Klasse (falls Sie keine geeignete fertige Klasse finden) und der Ausnahme-Klasse an!*
*Rufen Sie die Testmethode mit einer Matrix und mit einer "Nicht-Matrix" als Parameter auf, geben Sie die Größe aus und realisieren Sie das Exceptionhandling.*

### 3.3.1 Das Paket Reflection

(import java.lang.reflect.*;)
Was kann mit dem Paket gemacht weren?

- ermitteln des Typs von Objekten
- ermitteln von Attributen (Fields), Methoden, Konstruktoren, Modifiern, Superklassen

- dynamisches Instantiieren von Objekte, deren Typ erst zur Laufzeit bekannt sind
- dynamisches Beeinflussen von Attributwerten
- dynamisches Rufen von Methoden

Welche Klassen sind daran beteiligt:

- java.lang.Object
- java.lang.Class
- java.lang.reflect.Array
- java.lang.reflect.Constructor
- java.lang.reflect.Field
- java.lang.reflect.Method
- java.lang.reflect.Modifier

Wesentlich für die Angabe von Typen, isbesondere für Parameterlisten von Konstruktoren und Methoden, ist die Klasse Class (eine Art Laufzeit-Typbeschreibung). Entsprechende Class-Objekte können folgendermaßen geschaffen werden:

- Class.forName(*KlassenName*)
- *KlassenName*.class
- *objektName*.getClass()
- und für die elementaren Datentypen:
  *WrapperKlasse*.TYPE bzw. *Void*.TYPE
  oder auch
  *elemDataTypeName*.class bzw. *void*.class

Erzeugung von Instanzen auf dynamische Art (in den Beispielen wird **keine saubere Fehlerbehandlung** durchgeführt, alle Ausnahmen werden pauschal weitergeworfen):
Zunächst wird eine Klasse mit der Reflection-API nach Konstruktoren und deren Parametern befragt:

```
1  import java.lang.reflect.*;
2  ...
3    public static void main (String [] a) throws Exception {
4      Class m = Class.forName("Chef") ;
5      Constructor [] cf = m.getDeclaredConstructors ( );
6      for (int i=0;i<cf.length;i++) {
7        System.out.println("Konstruktor " + i + " von " + cf[i].getName());
8        Class cp [] = cf[i].getParameterTypes();
9        for (int j=0; j<cp.length; j++) {
10         System.out.println(" Parameter " + j + " : " + cp[j]);
11       }
12     }
```

ein Objekt wird mit dem Default-Konstruktor instanziiert

```
14     //*****************************************************
15     Person p0 = (Person) m.newInstance();
16     System.out.println ( p0 + " --- "+ p0.getClass().getName());
17     p0.drucke();
```

und letztendlich wird ein Objekt mit beliebigem Konstruktor instanziiert

```
19      //*****************************************************
20      Class ConstrParChef[] = {
21                          String.class,
22                          double.class, // oder Double.TYPE
23                          String.class
24      };
25      Constructor cc = m.getDeclaredConstructor (ConstrParChef);
26      Object [] po = {"derNeueChef", new Double( 12345.), "viele Abt."};
27      // oder auch ueber Kommandozeilenargumente
28      // Object [] po = {a[0], new Double(a[1]), a[2]};
29      Person p = (Person) c.newInstance(po);
30      p.drucke(po);
31    }
32  }
```

Mit der Methode getMethods() der Klasse Class erhält man ein Array von Objekten der Klasse Method, die wiederrum Methoden besitzt, mit denen Methoden aufgerufen werden können (z.B. invoke()).

**Übung 18** *Befragen Sie mit Reflection eine beliebige Klasse nach den deklarierten Methoden und deren Parametertypen, nehmen Sie von der Konsole die auszuführende Methode und Parameterwerte entgegen und rufen Sie diese Methode auf. Beschränken Sie sich der Einfachheit halber auf statische Methoden (also eine entsprechende eigene Klasse und/oder etwas wie java.lang.Math).*

## 3.4 Schnittstellen (Interfaces)

sind abstrakte Klassen, die nur

- Konstantenvereinbarungen (implizit public, static und final)
- und Methodendeklarationen (implizit abstract und public)

enthalten.
Die sie implementierenden Klassen müssen alle Methoden implementieren.
→ deshalb dürfen Interfaces **nicht wachsen**, ein Hinzufügen einer neuen Methode zu einem Interface bewirkt, daß alle Klassen, die das Interface implementieren, überarbeitet werden müssen!

Interfacedefinition:

```
[public] interface interfacename [extends SuperInterface] {
  interfacebody
}
```

Interface-Implementierung:

```
[public] class ClassName [extends SuperClassName] [implements InterfaceListe] {
  classbody
}
```

Die implementierende Klasse ist typkompatibel zu

- der Elternklasse
- und allen implementierten Interfaces

sowie allen deren Oberklassen/-interfaces.

**Beispiele zu Interfaces** zu finden im bald folgenden Abschnitt Eventhandling 4.2 (S.31)

## 3.5 Packages und Archiv-Tool jar

ohne eine Package-Anweisung wie
`package pack_name;`
sind die Klassen im Default-Package, wobei `pack_name` mehrere durch . getrennte Teile haben kann
**kleines Beispiel** für Package in Unterverzeichnis: Unterverzeichnisname und Packagename müssen übereinstimmen

```
aktuelles Verzeichnis
  z.java
  z.class
  Verz. a
    aclass.java (package a; ...)
    aclass.class
    Verz. b
      bclass.java (package a.b; ...)
      bclass.class
```

**Archiv-Tool jar**
Syntax von jar:

```
Syntax: jar {ctxu}[vfmOM] [JAR-Datei] [Manifest-Datei] [-C dir] Dateien ...
Optionen:
    -c  neues Archiv erstellen
    -t  Inhaltsverzeichnis f"ur Archiv auflisten
    -x  benannte (oder alle) Dateien aus dem Archiv extrahieren
    -u  vorhandenes Archiv aktualisieren
    -v  ausf"uhrliche Ausgabe f"ur Standardausgabe generieren
    -f  Namen der Archivdatei angeben
    -m  Manifestinformationen aus angegebener Manifest-Datei einbeziehen
    -0  nur speichern; keine ZIP-Komprimierung verwenden
    -M  keine Manifest-Datei f"ur die Eintr"age erstellen
    -i  Indexinformationen f"ur die angegebenen JAR-Dateien generieren
    -C  ins angegebene Verzeichnis wechseln und folgende Datei einbeziehen
Falls eine Datei ein Verzeichnis ist, wird sie rekursiv verarbeitet.
Der Name der Manifest-Datei und der Name der Archivdatei m"ussen
in der gleichen Reihenfolge wie die Flags 'm' und 'f' angegeben werden.
```

**Generieren eines Archivs**
`jar -cfv pack_a.jar a`
ausgeführt im aktuellen Verz., packt Verz. a rekursiv ein!
mit dem Mounten von `pack_a.jar` bzw. dem Eintrag in den CLASSPATH sind die Packages a.* und a.b.* im CLASSPATH von Java
**Erstellung einer Manifest-Datei:**
`jar cvmf manifest.txt pack_a.jar a`
Die Textdatei manifest.txt dient als Vorlage für die Manifestdatei meta-inf/Manifest.mf
der Eintrag `Main-Class: a.aclass` bewirkt, daß das Archiv oder besser gesagt die Haupt-Klasse `aclass` des Archivs mit `java -jar pack_a.jar` direkt ausgeführt werden kann

## 3.6 javadoc

Dokumentationskommentare sehen in einem einfachen Fall folgendermaßen aus:

```
1  /**
2   * tut dies und das
3   * @param p1 enthaelt dies
4   * @param p2 enthaelt das
5   * @return der Ergebnisvector
6   */
7  public Vector Rechner (String p1, boolean p2) {
8    ...
9  }
```

... und der Aufruf

```
javadoc [ options ] { package | sourcefile }
```

... die Links zu den Java-Standard-Klassen erhält man mit
`javadoc -link url packagenamen(mit Leerzeichen getrennt)` ,

... wobei URL auch eine file-URL auf dem lokalen System sein kann, z.B.
`javadoc -link file:///C:/jdk131/docs/api package1 package2`

... und die Doku sollte natürlich irgendwo (in einem docs-Verzeichnis) liegen und nicht bei den Quellen:
`javadoc -d ../docs package1 package2`

# 4 Spezielle Klassen und Probleme

## 4.1 Grundlegende graphische Anwendungen

anhand des AWT (Abstract Windowing Toolkit) erläutert
daneben gibt es noch Swing mit seinen Klassen

### 4.1.1 Applets

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class MyApplet extends Applet {

  public void init() {
    // Initialisierungsaufgaben, die einmal zu Beginn zu erfolgen haben,
    // werden hier gemacht (da Applet keinen Konstruktor hat)
  }

  // paint wird von der VM (und nicht vom Programmierer!) gerufen,
  // das uebergebene Graphics-Objekt ist sozusagen das Blatt Papier,
  // auf dem paint malen darf
  public void paint(Graphics g) {
    // Graphics (und somit g) verfuegt ueber viele Zeichnen-Methoden
    g.drawString("Hallo, Welt! " + getParameter("derPara"),
                 getSize().width/2-40, getSize().height/2-5);
  }
}
```

... und die benötigte, einbettende **Webseite** in einfachster Form:

```
<HTML>
<body>
<applet code="MyApplet.class" derPara=" und nun?"
        width="300" height="300">
</applet>
</body>
</HTML>
```

Ein Ruf von **repaint()** veranlaßt das Neuzeichnen des Applets, falls nötig!

### 4.1.2 Frames

```
import java.awt.*;
import java.awt.event.*;

public class EmptyFrame extends Frame {


  public EmptyFrame () {
    // Der Inhalt der Titelleiste des Fensters
    super("Ein leeres Fenster");
    // Groesse und Position
    setSize(500,300);
    setLocation(100,100);
    setVisible(true); // Sichtbarmachung
```

```
14    }
15
16    public static void main (String [] a) {
17      new EmptyFrame();
18    }
19
20  }
```

Warum dieses Fenster nicht zugeht, wenn wir auf das Kreuz klickern oder Alt-F4 drücken, erfahren wir im Abschnitt Eventhandling 4.2.3 (S.34)!

### 4.1.3 Fensterklassen, Componenten, Container, Button, ...

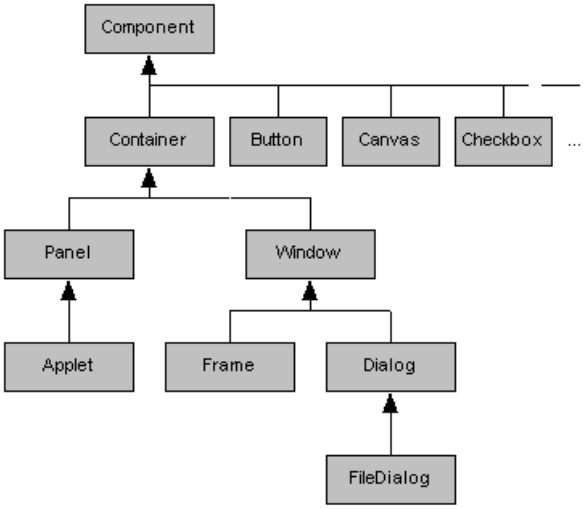



Abb. Hierarchie der Fensterklassen (aus [3, Kap.27, Fenster])

Dabei ist jeder **Container** geeignet, etwas anderes zu enthalten, verfügt also über `add(Component)`-Methoden,

während ein **Component**

- selbst etwas darstellt, also zumeist über eine `paint(Graphics)`-Methode verfügt, die genutzt werden kann aber nicht muß (auf einem Button wird sicher nur selten die `paint(...)`-Methode gerufen, wichtig ist das Zusammenspiel von `repaint(...)`, `update(...)` und `paint(...)`, durch Überschreiben von `update(...)` kann das Löschen verhindert werden, siehe API-Dok.)
- und über die `add(...)`-Methode eines Containers zu diesem hinzugefügt werden kann.

Somit können solche Objekte nach Bedarf ineinander verschachtelt werden.

### 4.1.4 Das Layout

Mit Layoutmanagern wie

- FlowLayout
- GridLayout
- BorderLayout
- GridBagLayout

kann der Inhalt einer Komponente (z.B. eines Frames) geordnet werden. Der Inhalt selbst wird mittels `add(...)` hinzugefügt. (siehe [4])

```
1  setLayout(new FlowLayout());
2
3  add(new Button("Weiter"));
4  add(new ...);
5  ...
```

Mit dem `FlowLayout` werden so viele Komponenten nebeneinander platziert, wie (in Abhängigkeit von der Breite) in eine Zeile passen.

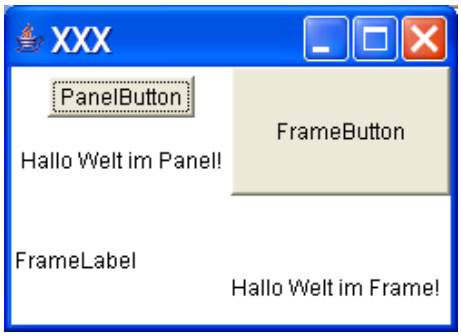
Soll manuell Größe und Position bestimmt werden, wird das **Null-Layout** verwendet:

```
setLayout(null);
Button but = new Button("Weiter");
add(but);
but.setBounds(x,y,width,height); //in java.awt.Component
...
```

**Übung 19** *Schreiben Sie eine Klasse, die einen Frame erweitert und in einem GridLayout über 2 Zeilen und 2 Spalten verfügt. In der oberen Zeile sollen zwei Buttons mit unterschiedlicher Beschriftung liegen, unten links ein Label! Lesen Sie dazu genau die API-Beschreibung der Klasse GridLayout!*

### 4.1.5 Ein zusammenfassendes Beispiel

Im folgenden Beispiel wollen wir sehen, wie wir das Aussehen von Componenten mit ihrer `paint(...)`-Methode beeinflussen können und diese oder Standardkomponenten (wie `Button`) dann über die `add(...)`-Methode von Containern zu diesen hinzufügen können. Die Beeinflussung des Layouts im Container mittels `setLayout(...)` wird anhand eines `GridLayout` demonstriert.




Ein `HelloFrame`, von `Frame` abgeleitet (Zeile 1), enthält, mittels 2x2-`GridLayout` (Zeile 8) angeordnet,

- ein `HelloPanel` (Zeile 3, 10, 33), von `Panel` abgeleitet (eine Componente, die Container ist), dieses `HelloPanel`
  - enthält wiederum einen Button
  - und zeichnet in `paint(...)` einen String,
- einen `Button`,
- ein `Label`,
- und zeichnet in `paint(...)` einen String.

```
public class HelloFrame extends Frame {

  HelloPanel hp;

  // im Konstruktor wird der Frame zusammengebaut
  public HelloFrame () {
    super("XXX");
    // ein Layoutmanager mit 2 Zeilen und 2 Spalten erhaelt die Aufgabe
    setLayout(new GridLayout(2,2)); // der Objektanordnung

    // jetzt werden nach und nach mit add(...) die Zellen des 2x2-Gitters befuellt
    hp = new HelloPanel(); // benanntes Objekt des Typs HelloPanel
    add(hp); // Zelle 1,1

    Button b = new Button("FrameButton");
```

```
16      b.setSize(800,1000);
17      add(b);  // Zelle 1,2
18
19      add( new Label("FrameLabel") ); //anonymes Objekt in Zelle 2,1
20
21      // Groesse, Position, Sichtbarkeit
22      setSize(250,180);
23      setLocation(100,100);
24      setVisible(true);
25    }
26
27    // zusaetzlich kann etwas auf dem Frame gemalt werden
28    public void paint(Graphics g) {
29      g.drawString("Hallo Welt im Frame!", getSize().width/2, getSize().height-20);
30    }
31
32    public static void main (String [] a) {
33      HelloFrame cm = new HelloFrame();
34    }
35  }
36
37  class HelloPanel extends Panel {
38
39    public HelloPanel() {
40      // enthaelt einen Button
41      add(new Button("PanelButton"));
42    }
43
44    public void paint(Graphics g) {
45      // und wird mit einem String bemalt
46      g.drawString("Hallo Welt im Panel!" , 5, getSize().height-15);
47    }
48  }
```

Dieses Programm zeigt beispielhaft die Wirkung von `paint(...)` und `add(...)`, natürlich wird man im Allgemeinen die `paint`-Methode eines **Containers** nicht rufen, sondern den Container mit `add(...)` mit anderen Componenten (wie `Button`, `TextField`, `TextArea`, `Label`, `Panel`, `Canvas`, ...) füllen. Die `paint`-Methode ihrerseits wird auf **Componenten, die keine Container** sind (wie z.B. `Canvas`) genutzt, wenn man rein graphische, nicht interaktive Inhalte wie z.B. einen Kurververlauf darstellen will.

## 4.2 Eventhandling

### 4.2.1 Begriffe

#### Ereignistypen

```
EventObject
  AWTEvent
    ComponentEvent  ... die Low-Level-Events
      FocusEvent
      InputEvent
        MouseEvent
        KeyEvent
      ...
    ActionEvent     ... die Semantic-Events
    ItemEvent
    TextEvent
    AdjustmentEvent
```

Mit dem Auftreten eines Ereignisses wie das Drücken einer Maustaste wird ein EventObjekt instantiiert. Dieses Eventobjekt wird über die Empfängermethoden von der Ereignisquelle zum -empfänger transportiert.

**Ereignisquellen**
verschiedene höhere Programmobjekte

**Ereignisempfänger**

```
FocusListener
MouseListener
ActionListener
...
```

In den nächsten Abschnitten anhand verschiedener Entwurfsmuster gezeigt, wie die Event-Empfänger auf unterschiedliche Art und Weise in das Geschehen eingebunden werden können.

### 4.2.2 Implementation eines ListenerInterfaces

ein kleines Beispiel in Form eines Applets, welches ein MouseListener-Interface implementiert und damit auch die Methoden des MouseListener implementieren muß:

```
1  import java.applet.Applet;
2  import java.awt.*;
3  import java.awt.event.*;
4
5  public class ClickMe extends Applet implements MouseListener {
6    private Spot spot = null;
7    private static final int RADIUS = 7;
8
9    public void init() {
10     addMouseListener(this);
11   }
12
13   public void paint(Graphics g) {
14     g.setColor(Color.lightGray);
15     g.fillRect(0, 0, getSize().width - 1, getSize().height - 1);
16     g.setColor(Color.black);
17     g.drawRect(0, 0, getSize().width - 1, getSize().height - 1);
18     g.drawString("Klick mich!", getSize().width/2-40, getSize().height/2-5);
19     //draw the spot
20     g.setColor(Color.red);
21     if (spot != null) {
22       g.fillOval(spot.x - RADIUS, spot.y - RADIUS, RADIUS * 2, RADIUS * 2);
23     }
24   }
25   public void mousePressed(MouseEvent event) {
26     if (spot == null) {
27       spot = new Spot(RADIUS);
28     }
29     spot.x = event.getX();
30     spot.y = event.getY();
31     repaint();
32   }
33   public void mouseClicked(MouseEvent event) {}
34   public void mouseReleased(MouseEvent event) {}
35   public void mouseEntered(MouseEvent event) {}
36   public void mouseExited(MouseEvent event) {}
37 }
```

**... und der Spot**

```
1  public class Spot {
2    public int size;
3    public int x, y;
4
5    public Spot(int intSize) {
6      size = intSize;
```

```
    x = -1;
    y = -1;
  }
}
```

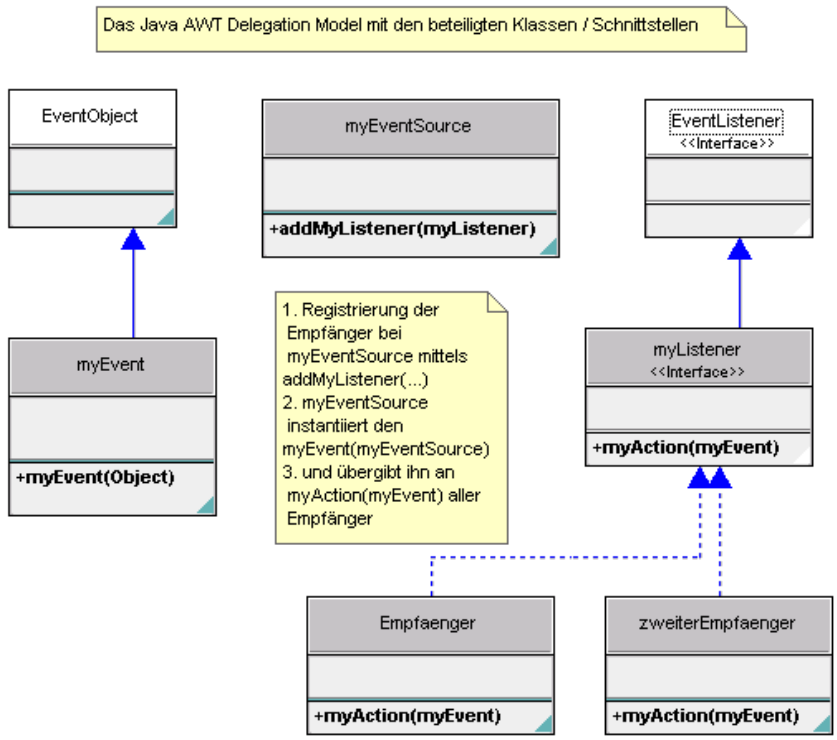
... und die benötigte, einbettende **Webseite** in einfachster Form:

```
<HTML>
<body>
<applet code="ClickMe.class"
        width="300" height="300">
</applet>
</body>
</HTML>
```

(mehr Infos zu HTML: Stefan Münz: Selfhtml aus dem WWW downloaden)

**Übung 20** *Bauen Sie das Applett in einen Frame um!*

- *Modifizierung des Spots, die Farbe soll im Spot gehalten werden, benutzen der entspechenden Klasse für Farben*
- *der Spot soll auch beim Loslassen der Maustaste erscheinen, aber in anderer Farbe*
- *die Schrift soll sich ändern, wenn die Maus das Window betritt bzw. verläßt*
- *und als Abschluß soll der Spot sich selbst darstellen:*
  *public void draw (Graphics)*




### 4.2.3 Beauftragung einer inneren Klasse

**xxxAdapter** sind Implementationen von **xxxListenern** mit leeren Methodenkörpern, durch ihre Nutzung erspart man sich Schreibarbeit, da nur die genutzten Methoden überschrieben werden.
Wenn ein **xxxListener**-Interface nur eine Methode enthält, gibt es keinen **xxxAdapter**, man kann auch einfach die Methode implementieren.

### innere anonyme Klasse

Hier ist sichtbar, was getan werden muß, um ein Fenster beenden zu können, es muß nämlich ein WindowListener auf dem Fenster angemeldet werden, der das Fenster schließt!

```
import java.awt.*;
import java.awt.event.*;

public class EmptyFrameWithInnerAnonymClass extends Frame {

  public EmptyFrameWithInnerAnonymClass () {
    super("Ein leeres Fenster");

    addWindowListener(
      new WindowAdapter() {
        public void windowClosing(WindowEvent event) {
          System.out.println("WindowClosing...");
          setVisible(false);
          dispose();
        }
        public void windowClosed(WindowEvent event)
        {
          System.out.println("terminating program...");
          System.exit(0);
        }
      }
    );

    setSize(500,300);
    setLocation(100,100);
    setVisible(true);
  }

  public static void main (String [] a) {
    EmptyFrameWithInnerAnonymClass cm = new EmptyFrameWithInnerAnonymClass();
  }

}
```

Dateien nach dem Kompilieren:

```
EmptyFrameWithInnerAnonymClass.java
EmptyFrameWithInnerAnonymClass.class
EmptyFrameWithInnerAnonymClass$1.class
```

### innere benannte Klasse

```
import java.awt.*;
import java.awt.event.*;

public class EmptyFrameWithInnerClass extends Frame {


  public EmptyFrameWithInnerClass () {
    super("Ein leeres Fenster");

    addWindowListener( new MyWindowCloser() );

    setSize(500,300);
    setLocation(100,100);
    show();
  }
```

```
16
17    public static void main (String [] a) {
18      EmptyFrameWithInnerClass cm = new EmptyFrameWithInnerClass();
19    }
20
21
22    class MyWindowCloser extends WindowAdapter {
23      public void windowClosing(WindowEvent event) {
24        System.out.println("WindowClosing...");
25        setVisible(false);
26        dispose();
27      }
28      public void windowClosed(WindowEvent event) {
29        System.out.println("terminating program...");
30        System.exit(0);
31      }
32    }
33
34  }
```

Dateien nach dem Kompilieren:

```
EmptyFrameWithInnerClass.java
EmptyFrameWithInnerClass.class
EmptyFrameWithInnerClass$MyWindowCloser.class
```

**normale Klasse**

Soll die Klasse vielleicht mehrfach verwendet werden, kann man natürlich auch eine ganz normale Klasse verwenden!

**Übung 21** *Modifizieren Sie die Klasse aus Übung 19 so, daß die beiden Buttons jeweils eine unterschiedliche Ausschrift in der unteren linken Zelle (z.B. als Label) bewirken!*

### 4.2.4 Menues und ActionEvents

Für die Bedienungsmöglichkeit mit Maus oder Tastatur ist es günstig, von den sogenannten Low-Level-Events zu Semantic-Level-Events (wie z.B. Action-Events) überzugehen. Das folgende Beispiel zeigt, neben der Möglichkeit, ein AWT-Menu aufzubauen, die Reaktion auf und die Bestimmung der Quelle von ActionEvents.

```
1   import java.awt.*;
2   import java.awt.event.*;
3   import java.io.*;
4
5   public class ActionGUI3 extends Frame implements ActionListener {
6
7     TextField tf;
8     TextArea ta;
9
10    MenuBar mb;
11    Menu m;
12    MenuItem mbItemCalc, mbItemExit;
13
14    Button button;
15
16    public static void main (String [] a) {
17      new ActionGUI3();
18    }
19
20    public ActionGUI3 () {
21      super("Rechner");
```

```
22      addWindowListener(
23        new WindowAdapter() {
24          ...
25        }
26      );
```

Der Aufbau des Menus und das zugehörige Eventhandling erfolgt im Konstruktor:

Die Menuleiste **(MenuBar) mb** wird mit **setMenuBar(mb)** auf dem Frame angemeldet und erhält mittels **add()** seine Aufklappmenus (**Menu**) zugeordnet, diese wiederum erhalten mittels **add()** Menuzeilen (**MenuItem**) zugeordnet.
Man beachte, daß es hier zwei **ActionListener** gibt, einer ist der Frame selbst, der andere ein anonymer, der auf der Menuzeile `mbItemExit` angemeldet wird (immer zu finden über die **add...Listener()**-Methoden:

```
28      // das AWT - Menu ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
29      mb = new MenuBar();
30      m = new Menu("... und nun?");
31      mbItemCalc = new MenuItem("Kopieren, aber schnell!");
32      mbItemExit = new MenuItem("Ende");
33      m.add(mbItemCalc);
34      mbItemCalc.addActionListener(this);
35      m.add(mbItemExit);
36      mbItemExit.addActionListener(new ActionListener() {
37        public void actionPerformed(ActionEvent evt) {
38          System.out.println("... ActionListener im Menu ...");
39          setVisible(false);
40          dispose();
41          System.exit(0);
42        }
43      });
44      mb.add(m);
45      setMenuBar(mb);
```

```
47      // der Fensterinhalt ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
48      setLayout(new FlowLayout());
49
50      Label l = new Label ("String: ");
51      add (l);
52
53      tf = new TextField(30);
54      add (tf);
55      ta = new TextArea(10,60);
56      add (ta);
57
58      button = new Button("Kopiere");
59      button.addActionListener(this);
60      add(button);
61
62      setSize(500,300);
63      setLocation(100,100);
64      setVisible(true);
65      ta.append("gestartet\n");
66    }
```

... nun folgen die Reaktionen auf die ActionEvents, die mit der Maus oder der Tastatur ausgelöst werden können, die Behandlung der ActionEvents der verschiedenen Quellen (Menu, Button) erfolgt hier in einer Methode, für die Bestimmung der Quelle gibt es verschiedene Möglichkeiten wie die Bestimmung des ActionCommand der Quelle ...

```
68 // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
69   public void actionPerformed(ActionEvent event) {
70
71     String c = event.getActionCommand();
72 //  liefert das ActionCommand der Quelle, das ist
73 //  per default auch Label der Quelle,
74 //  quelle.setActionCommand (String ac); // Setzt das ActionCommand auf quelle
75 //  quelle.setLabel(...)              // setzt die Beschriftung auf quelle
76
77 //  Moeglichkeit der gezielten Reaktion und der Aenderung der Bedeutung von Schaltern
78
79     if (c.equals("Kopiere"))
80                       System.out.println("fester String: Button ist Quelle");
81     if (c.equals("Kopieren, aber schnell!"))
82                       System.out.println("fester String: Menu-Eintrag ist Quelle");
83
84 //  VORSICHT :
85 //  Die Nutzung des Strings auf dem Schalter etc. ist wegen Internationalisierung
86 //  und möglichen Änderungen nicht zu empfehlen (zumindest der Vergleich gegen einen
87 //  festen String), ma
```

. . . oder auch die Bestimmung der Quelle an sich über die Referenz:

```
89 //  ... besser ist public Object getSource() auf einem EventObject
90
91     Object o = event.getSource();    // liefert die Quelle
92
93     if (o.equals(mbItemCalc)) System.out.println("Object: Menu-Eintrag ist Quelle");
94     if (o.equals(button))   System.out.println("Object: Button ist Quelle");
95
96 //  ... oder der Vergleich gegen den dynamisch erhaltenen String
97
98     if (c.equals(button.getActionCommand()))
99                     System.out.println("aktueller String: Button ist Quelle");
100    if (c.equals(mbItemCalc.getActionCommand()))
101                    System.out.println("aktueller String: Menu-Eintrag ist Quelle");
102
103 //  ... oder auch die Trennung von Label (als Beschriftung)
104 //      und ActionCommand (als Code für das Quellobjekt)
105
106    String s = tf.getText();
107    System.out.println("Commandsource: \"" + c +"\" String: \"" + s + "\"");
108    ta.append(s + "\n");
109  }
110 }
```

**Übung 22** *Bauen Sie zu Aufgabe 16 (S.24) ein graphisches Userinterface (z.B. ein Panel, welches dann in einem Frame oder auch in einem Applet verwendet wird). Dieses Panel soll 3 Texteingabefelder und einen Button enthalten. In den Texteingabe-Komponenten sollen untere und obere Intervallgrenze und Punktzahl eingegeben werden. Der Button soll das Berechnen und zeilenweise Ausdrucken der x-/y-Wertepaare der Klasse* `Punkt` *auf die Konsole (Standardausgabe) anstoßen :*

```
x-Wert1 ; y-Wert1
x-Wert2 ; y-Wert2
x-Wert3 ; y-Wert3
```

*Dazu muß die Berechnungsschleife aus dem* `main(...)` *von* `Punkt` *an eine passende Stelle verlegt werden.*

## 4.3 Ein- und Ausgabe

### 4.3.1 Die Nutzung der Stream-Klassen

Standard-E/A-Ströme:

- System.in
- System.out
- System.err

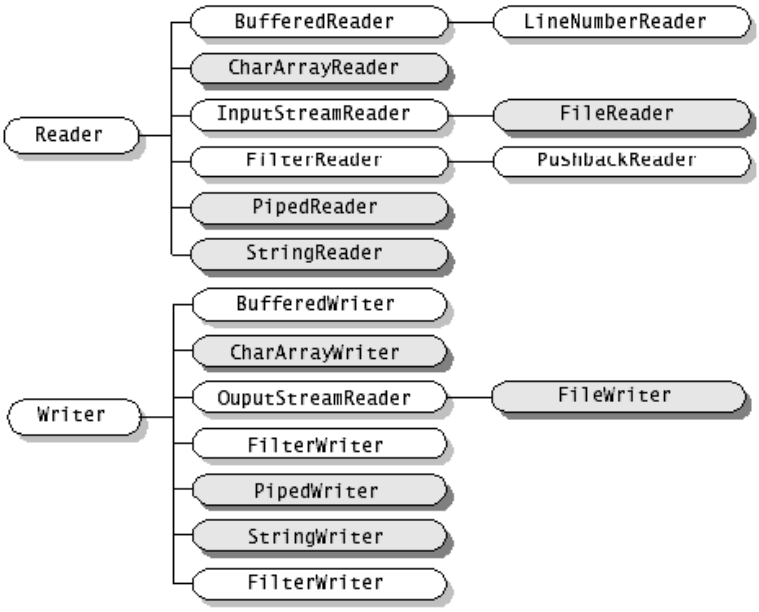
Byte-weises Lesen/Schreiben auf den Standard-Streams:

```
1 import java.io.*;
2 public class simpleread {
3   public static void main(String[] args) throws IOException {
4     int c;
5     while ((c = System.in.read()) != -1)
6        System.out.write(c);
7   }
8 }
```

c = -1 bedeutet Ende der Eingabe
etwas Buffering ist auch hier

**Übung 23** *Zu welcher Klasse gehört System.in (siehe API)?*
*Geht das wirklich (siehe Klassendeklaration)? Befragen Sie das Objekt, zu welcher Klasse es gehört! Wie passiert das?*

**Character Streams**



### Byte Streams

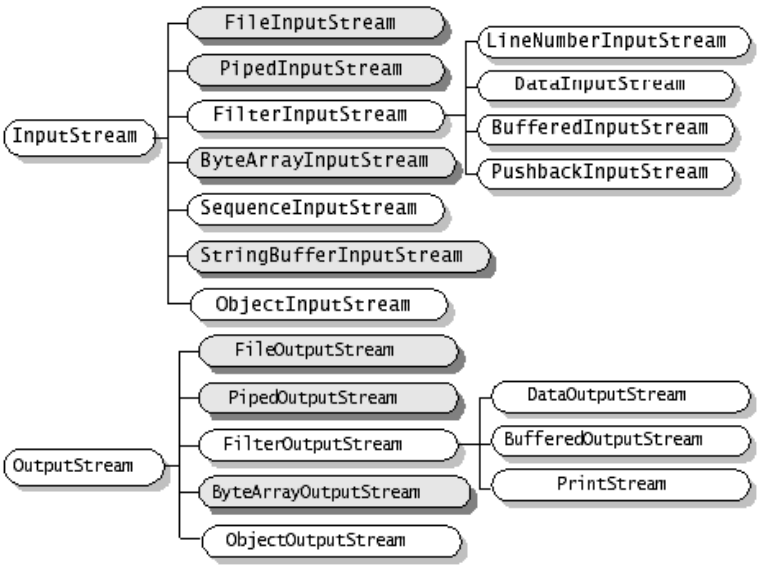



Abb. aus dem Java-Tutorial (java.sun.com)

Zeilenweises Lesen von Strings von der Standard-Eingabe

```
1  import java.io.*;
2  public class ConsoleRead {
3    public static void main(String[] args) throws IOException {
4
5      BufferedReader i = new BufferedReader(new InputStreamReader(System.in));
6      String s = i.readLine();
7
8      System.out.println("--- "+s);
9    }
10 }
```

. . . oder (seit JDK 1.5). . .

```
1  import java.util.Scanner;
2    ...
3      Scanner i = new Scanner (System.in);
4      String s = i.nextLine();
5    ...
```

. . . das zeilenweise Schreiben von Strings irgendwohin (z.B. ein File) macht man mit einem **PrintWriter**:

```
1  import java.io.*;
2
3    ...
4      PrintWriter out = new PrintWriter(filename);
5      out.println(s);
6    ...
```

### Beispiele zum Dateikopieren und -modifizieren mit Stream-Operationen

Es folgt zunächst ein Beispiel für das Kopieren einer Datei ohne Buffering unter Nutzung der **Klasse File**. Ob intern über Byte- oder Charakterströme gegangen wird, spielt keine Rolle, da es keinerlei interne Weiterverarbeitung gibt. Zunächst arbeiten wir zeichenweise (char oder byte) . . .

```
1  import java.io.*;
2
3  public class MyFileCopy {
```

```
4
5   public static void main(String[] args) throws IOException {
6     File inputFile = new File("in.txt");
7     File outputFile = new File("out.txt");
8
9     // die Character-Streams
10    FileReader in = new FileReader(inputFile);
11    FileWriter out = new FileWriter(outputFile);
12
13    // oder an deren Stelle die Byte-Variante
14  // FileInputStream in = new FileInputStream(inputFile);
15  // FileOutputStream out = new FileOutputStream(outputFile);
16
17    int c;
18    while ((c = in.read()) != -1)
19       out.write(c);
20    in.close();
21    out.close();
22  }
23
24 }
```

... und nun mit Input-Filter für zeilenweises Lesen auf dem Characterstream

- `BufferedReader`
- oder statt dessen `LineNumberReader` (bietet zusätzlich die Verwaltung/Zählung der gelesenen Zeilenzahl)

und zeilenweise Ausgabe mit Output-Buffering (`BufferedWriter`) , hier wird intern mit Strings (also Charakterdaten) gearbeitet:

```
8     ...
9     // die Character-Streams
10    FileReader in = new FileReader(inputFile);
11    FileWriter out = new FileWriter(outputFile);
12
13    LineNumberReader i = new LineNumberReader(in);
14    BufferedWriter o = new BufferedWriter(out);
15
16    String s;
17    while ((s = i.readLine()) != null) {
18      System.out.println(s);
19      o.write(s,0,s.length());
20      o.newLine();
21    }
22    i.close();
23    o.close();
24    //Experimentieren ohne close, und mit flush
25    ...
```

... und nun die Ausgabe jeder Zeile mit Zeilennummer:

```
15    ...
16    String s;
17    while ((s = i.readLine()) != null) {
18      s = i.getLineNumber() + ":" + s;
19      System.out.println(s);
20      o.write(s,0,s.length());
21      o.newLine();
22    }
23    ...
```

### 4.3.2 Ein GUI-Beispiel (unter dem AWT)

Im folgenden Beispiel soll vor allem auf die Trennung zwischen Visualisierung und Fachinhalten hingewiesen werden, die durch die Aufspaltung in 2 Klassen realisiert wird.
Hingewiesen werden soll noch auf die Zeilen 15 und 21: Hier wird aus einer inneren anonymen Klasse heraus auf eine lokale Variable der umgebenden Methode zugegriffen (Zeile 21). Damit das möglich ist, muß diese Variable als `final` deklariert werden!

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import java.io.*;
4
5  public class myFileCopy3Frame extends Frame {
6
7     public myFileCopy3Frame () {
8       super("Kopiertest");
9       addWindowListener(
10        ...
11      );
12      setLayout(new FlowLayout());
13      Label l = new Label ("Dateiname (Quelle)");
14      add (l);
15      final TextField tf = new TextField(30);
16      add (tf);
17      Button button = new Button("Kopiere");
18      button.addMouseListener(
19        new MouseAdapter(){
20          public void mouseClicked(MouseEvent event) {
21            String n = tf.getText();
22            String f = "";
23            System.out.println("Maus gepresst, Inputfile: " + n);
24            try { f = CpLNr.CopyLineNr(n, n + ".out"); }
25            catch ( IOException e) {System.out.println("Problems mit " + n);}
26            System.out.println(f);
27          }
28        }
29      );
30      add(button);
31      TextArea ta = new TextArea(10,60);
32      add (ta);
33    }
34
35    public static void main (String [] a) {
36      myFileCopy3Frame cm = new myFileCopy3Frame();
37      cm.setSize(600,300);
38      cm.setLocation(100,100);
39      cm.show();
40    }
41 }
42
43 class CpLNr {
44   public static void main(String[] args) {}
45
46   public static String CopyLineNr(String inname, String outname) throws IOException {
47
48     FileReader in = new FileReader(new File(inname));
49     FileWriter out = new FileWriter(new File(outname));
50     LineNumberReader i = new LineNumberReader(in);
51     BufferedWriter o = new BufferedWriter(out);
52
53     String s, g="";
54     while ((s = i.readLine()) != null) {
```

```
      s = i.getLineNumber() + ":" + s;
      System.out.println(s);
      o.write(s,0,s.length());
      o.newLine();
      g += s + "\n";
    }
    i.close();
    o.close();
    return g;
    //Experimentieren ohne close, und mit flush
  }
}
```

### 4.3.3 Serialisierung

#### Grundlagen

Bei der Serialisierung wird ein Objekt in eine Folge von Bytes zerlegt (serialisiert) zum Zweck des Transportes mittels eines Streams, welcher irgendwohin geht (File, Socket, andere Stelle in gleicher VM). Der entgegengesetzte Vorgang heißt Deserialisierung).

In der API-Dokumentation kann man sich bei den In- und OutputStrömen über passende Klassen und Methoden der Klassen

- `ObjectInputStream`
- `ObjectOutputStream`

informieren, die zum Transport serialisierter Objekte (Interface `Serializable`!) geeignet sind.

Das folgende Listing zeigt beispielhaft, wie Objekte serialisiert/ deserialisiert werden:

```
FileOutputStream fs = new FileOutputStream("test.serial");
ObjectOutputStream os = new ObjectOutputStream(fs);
os.writeObject("Hallo");
os.writeObject(new Time(10, 30));
os.writeObject(new Time(11, 25));
os.close();

FileInputStream fs = new FileInputStream("test2.serial");
ObjectInputStream is = new ObjectInputStream(fs);
Time time = (Time)is.readObject();
System.out.println(time.toString());
is.close();
```

Ein Objekt kann nur serialisiert werden, wenn es das Interface `Serializable` implementiert!

#### Beispiel zur Serialisierung eines Frames

**Die Serialisierung eines Frames** bei Programmende und darauffolgende Deserialisierung beim nächsten Programmstart hat zur Folge, daß Eigenschaften des Frames wie Position und Größe des Frames gespeichert und wiederhergestellt werden:

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class DefaultFrameSerial extends Frame implements Serializable {

  public DefaultFrameSerial () {
    super("Kopiertest");
    addWindowListener(
      // als WindowListener arbeitet ein Objekt der Klasse MyWindowSerializer
```

```
      new MyWindowSerializer()
    );
  }

  public void serialisiere() {
    try {
      System.out.println("SerialisierungsMethode");
      FileOutputStream fos = new FileOutputStream("dfs.ser");
      ObjectOutputStream os = new ObjectOutputStream(fos);
      os.writeObject(this);
      os.close();
    } catch (Exception e) {}
  }

  public static void main (String [] a) {
    DefaultFrameSerial dfs;
    FileInputStream fis;
    ObjectInputStream ois;
    try {
      fis = new FileInputStream("dfs.ser");
      ois = new ObjectInputStream(fis);
      dfs = (DefaultFrameSerial) ois.readObject();
      dfs.setVisible(true);
    } catch (FileNotFoundException e) {
      System.out.println(e);
      dfs = new DefaultFrameSerial();
      dfs.setSize(300,100);
      dfs.setLocation(100,100);
      dfs.setVisible(true);
    } catch (Exception e) {}
  }
```

Die Klasse `MyWindowSerializer` wird hier als innere Klasse realisiert, sie ist ein WindowAdapter und serialisierbar:

```
  class MyWindowSerializer extends WindowAdapter implements Serializable {
  // Serializable bewirkt, daß die Objekte der inneren Klasse mit serialisiert werden,
  // sonst fehlt der MyWindowSerializer beim 2. Lauf!!
    public void windowClosing(WindowEvent event) {
      System.out.println("Begin windowClosing");
      serialisiere();
      setVisible(false);
      dispose();
      System.out.println("Ende windowClosing");
    }
    public void windowClosed(WindowEvent event) {
      System.out.println("terminating program...");
      System.exit(0);
    }
  }
}
```

## 4.4 Containerklassen und Stringhandling

### 4.4.1 Containerklassen

Man sollte sich zu dieser Problematik die Beschreibung zum Jave Collections Framework ansehen.
In der Java-Version 1.5 gibt es einige Neuerungen, insbesondere ist durch die sogenannten **Generics**

Das ist z.B. an folgender Klassenbezeichnung ablesbar:

```
    java.util.Vector<E>
```

```
      public boolean add(E o)
      public E get(int index)
```

Ein Casten beim Auslesen von Inhalten nicht mehr nötig, der Typ wird beim Hinzufügen (mit `add(...)`) registriert und beim Auslesen dem referenzierten Objekt zugewiesen:

```
Vector v = new Vector();
E o = new E();
v.add(o); // bringt unchecked-Warnungen
          // gemischtes Speichern verschiedener Typen moeglich
E n = v.get(...);

// und mit checked option fuer die Typsicherheit:
Vector<E> v = new Vector<E>();
...  // keine Warnungen
     // nur ein Typ E im Vector
     // wobei der festgelegte Typ E eben auch Object sein kann,
     // dann passen ALLE Klassen in den Vector!
```

Mit der Typisierung auf `Object` hat man einen Container, in den alle möglichen Objekte gesteckt werden können:

```
Vector<Object> v = new Vector<Object>();
    // nur ein Typ Object im Vector
```

In Java-Version 1.4 sah das noch folgendermaßen aus:

```
    java.util.Vector
      public boolean add(Object o)
      public Object get(int index)
```

ein Casten beim Auslesen von Inhalten auf den eigentlichen Typ war dort nötig:

```
Vector v = new Vector();
E o = new E();
v.add(o);
E n = (E) v.get(...);
  // ohne Casten ist nur ein Object bekannt
  // das Objekt n wuerde die Methoden von E nicht kennen
```

**Iterator** und **Enumeration** sind Interfaces, die oft verwendet werden, wenn Listen von Ergebnissen geliefert werden, die ausgelesen werden können (Iteration in einer Richtung).
Oft liefern Methoden ihre Ergebnisse als Enumeration zurück (Vector: elements(), Hashtable: keys())

**Set** ist ein Interface, das eine Menge (ohne doppeltes Auftreten) abstrahiert.

**List** ist ein Interface, das eine geordnete Sammlung repräsentiert. Zugriff über die Position ist möglich. Ein Blick in die API-Doc zeigt, daß es Methoden gibt, mit denen Objekte (am Ende oder an beliebiger Stelle) eingefügt, ersetzt oder entfernt werden können.

**Vector** ist eine Containerklasse, das List-Interface implementiert.

**Stack** ist eine Erweiterung von Vector mit den stacktypischen push()- und pop()-Methoden.

**Dictionary** ist eine abstrakte Containerklasse von Schlüssel-/Wertepaaren, die Schlüssel sind eine Menge, also eindeutig, keiner erscheint mehrfach.

**Hashtable** ist eine von `Dictionary` abgeleitete Klasse.

**Übung 24** *Bauen Sie eine Anwendung, die eine Häufigkeitsanalyse von Worten durchführt. Die Daten (das können z.B. Namen sein) sollen auf irgendeinem Weg (z.B. durch zeilenweise Eingabe auf der*

*Konsole) in die Anwendung gelangen, dort können sie in einem Container (*`Vector`*) zwischengespeichert werden. Denken Sie sich eine geeignete Abbruchbedingung für die Eingabe aus.*
*Dann soll die Bestimmung der Häufigkeit des Auftretens der einzelnen Worte erfolgen (z.B. unter Nutzung einer Hashtable), in deren Ergebnis die Worte mit der entsprechenden Anzahl ausgegeben werden.*

**Übung 25** *Halten Sie die Menge der Punkte (x-/y-Wertepaare in der Klasse* `Punkt`*) aus Aufgabe 16 (S.24) bzw. Aufgabe 22 (S.37) nicht mehr in einem Array, sondern in einem von* `Vector` *abgeleitetem Container (Klasse* `Punkte`*), welcher auf* `Punkt` *typisiert ist!*
*Die Klasse* `Punkte` *soll nun über eine Methode* `public void xyPrint ( PrintStream p ) {...}` *verfügen, die alle Punkte wie folgt zeilenweise ausgibt:*

```
x-Wert1 ; y-Wert1
x-Wert2 ; y-Wert2
x-Wert3 ; y-Wert3
```

**Übung 26** *Aufbauend auf Aufgabe 25 (S.45) soll jetzt zusätzlich zum Parametereingabe-Panel ein Darstellungskomponente vom Typ* `Panel` *oder* `Canvas` *existieren, in der die x-/y-Wertepaare graphisch als Funktion dargestellt werden. In der zugehörigen Methode* `public void paint(Graphics g) {...}` *wird dazu eine Methode* `public void draw (Graphics g) {...}` *in* `Punkte` *aufgerufen, wo wiederum in einer Schleife gleichnamige Methoden auf allen* `Punkt`*-Objekten gerufen werden!*

### 4.4.2 StringTokenizer, String, ...

Mit dem **StringTokenizer** kann ein String in Tokens (so etwas wie lexikalische Grundeinheiten) zerlegt werden, die durch Begrenzer (im Defaultfall sind das Whitespaces) voneinander getrennt werden (siehe API). Ein StringTokenizer ist gleichzeitig Enumeration (als impl. Interface).

Der **String** selbst verfügt über eine `split(...)`-Methode, die, in Erweiterung des StringTokenizers, als Begrenzer reguläre Ausdrücke verarbeiten kann.

**Übung 27** *Nutzen Sie eine der genannten Möglichkeiten (mittels String oder StringTokenizer), um einen Text durch Auftrennung in seine Worte für Übung 24 aufzubereiten!*

Eine **CharSequence** ist ein Interface, das Lesezugriff auf verschiedene Charactersequenzen wie `String` oder `StringBuilder` liefert.

### 4.4.3 Reguläre Ausdrücke

Reguläre Ausdrücke dienen zur Mustersuche in Texten. Sie stellen nach einem formalen Schema Muster (Pattern) dar, die auf eine Folge von Charakterdaten angewandt werden, um Stellen in diesen Daten zu finden, die dem Muster entsprechen.
So kann man zum Beispiel eine Textdatei (z.B. das Log-File eines Firewalls oder Webservers, nach IP-Adressen (numerisch, z.B. 212.184.75.55) durchsuchen. Das Muster für solche Adressen ist einfach zu formulieren, vier Zifferngruppen, durch Punkte getrennt. Weiter unten werden wir eine Lösung eines ähnlichen Problems sehen.

- **java.util.regex.Pattern** definiert ein Muster (Pattern) zur Textsuche, in der Javadoc der Klasse findet man die ausführliche Beschreibung der Syntax der regulären Ausdrücke in Java
- **java.util.regex.Matcher** ist der Ausführende, der das Muster in einem konketen Text sucht

```
1 import java.util.regex.*;
2 ...
3 Pattern p = Pattern.compile("a+b"); // ein regulaerer Ausdruck "a+b"
4 Matcher m = p.matcher("aaaab");  // ein zu parsender String "aaaab"
5 boolean b = m.matches();       // true, wenn der RE den Gesamtstring matcht
```

In Zeile 3 wird ein Pattern-Objekt durch Compilieren eines regulären Ausdrucks (`"a+b"`, was ein oder mehrere a, gefolgt von einem b bedeutet) erzeugt, seiner Methode `matcher(CharSequence)` wird der zu parsende String übergeben, Rückgabewert ist ein Matcherobjekt (Zeile 4). Zu beachten ist, daß CharSequence ein Interface ist, welches durch einen gewöhnlichen String implementiert wird, aber auch durch einen StringBuilder, mit dem über eine append()-Methode Daten hinzugefügt werden können (z.B. zeilenweise aus einem InputStreamReader).
Auf diesem Objekt kann nun z.B. mit der Methode matches() (Zeile 5) getestet werden, ob der reguläre Ausdruck den Gesamtstring matcht, was hier der Fall ist.
Einige Methoden der Klasse Matcher:

- `public boolean matches()`: prüft, ob der RE den String als Ganzes matcht
- `public boolean lookingAt()`: prüft vom Stringanfang an, ob der RE an irgendeiner Stelle des Strings matcht
- `public boolean find()`: prüft ab Ende der letzten Suche (letzter find()-Ruf), ob der RE matcht, und bleibt nach der Fundstelle stehen
- `public String group()`: liefert den dem RE entsprechenden Substring aus der letzten Suche zurück
- `public int start()`: liefert die Startposition des dem RE entsprechenden Substring aus der letzten Suche zurück
- `public int end()`: liefert die Endposition (Position nach dem letzten Zeichen) des dem RE entsprechenden Substring aus der letzten Suche zurück

Folgendes Beispiel sucht aus einer Zeichenfolge alle Fließkommazahlen (mit Punkt, Ziffern vor dem Punkt müssen, nach dem Punkt können stehen.

```
1 Pattern p = Pattern.compile("\\d+\\.\\d*"); // der RE: \d+\.\d*
2 Matcher m = p.matcher(" eine Folge von Zahlen 1.25 15. 0.654! ");
3 while (m.find()) System.out.println(m.group());
```

In Zeile 1 wird deutlich, daß ein im RE vorkommender Backslash durch die Eigenheit, wie Java in Strings mit Backslashes umgeht, **doppelt** geschrieben werden muß. In Zeile 3 werden nun alle Zahlen untereinander ausgegeben.

**Übung 28** *Bauen Sie einen Parser, der numerische IP-Adressen in einem String findet. Geben Sie diese an der Konsole aus.*

*Verwenden Sie anstelle des zu parsenden Strings einen StringBuilder, den Sie aus einem Eingabefile (*`ZALog.txt`*) befüllen.*

*Speichern Sie die Adressen in einem Objekt einer* `List` *implementierenden Klasse.*

*Bestimmen Sie die Häufigkeit des Auftretens. (Hashtable ist dafür gut geeignet. siehe auch Übung 24)*

*Spielen Sie mit den regulären Ausdrücken (Beschreibung in der API-Doc der Klasse* `Pattern`*), schließen z.B. Sie die Portnummer mit ein.*

*(very optional) Versuchen Sie, aus einem html-File alle href-Attribute zu finden! Oder was Sie gerade finden wollen in einem File Ihrer Wahl ;-)*

## 4.5 Threadprogrammierung

### 4.5.1 Einfache Threads

Threads sind "Ablauffäden" innerhalb eines Programmes (Prozesses), sie teilen sich dessen Ressourcen. Das bedeutet, daß mit Threads paralleles Abarbeiten von Aufgaben mit weniger Aufwand als mit Prozessen möglich ist.
Thread-Programme werden durch

- Erweiterung von `Thread`

```
1 public class AThreadClass extends Thread{
2   public static void main(String[] args) {
3     AThreadClass t = new AThreadClass();
4     t.start();
5   }
6   public void run() { ... }
7 }
```

oder

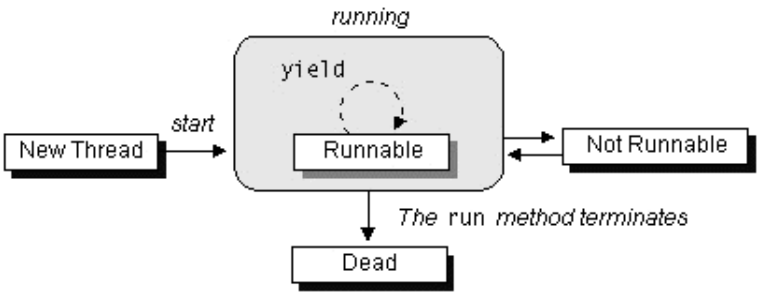
- Implementierung von `Runnable`, hier wird ein entsprechendes Objekt einem Thread-Konstruktor als Parameter übergeben, das entstehende Thread-Objekt erhält alle Fähigkeiten des Ausgangsobjektes,

```
1  public class ARunnableClass extends AClass implements Runnable{
2    public static void main(String[] args) {
3      ARunnableClass o = new ARunnableClass();
4      Thread t = new Thread(o);
5      t.start();
6    }
7    public void run() { ... }
8  }
```

geschaffen, die Methode `start()` startet den Thread, in jedem Fall steht in der Methode `run()`,

```
...
public void run() {
  ... // die Handlung des Threads
}
...
```

was der Thread während seiner Lebenszeit tut. Rufen Sie niemals die Methode `run()` auf, denn dann wird sie abgearbeitet, ohne als Thread zu laufen!




**Lebenszyklus eines Threads** (aus [1])

Threads gehen in den Zustand NotRunnable über durch

- den Aufruf von `sleep()`, hierbei wird das Eigentum am Monitor nicht abgegeben,
- den Aufruf von `wait()` zum Warten auf bestimmte Bedingungen,
- das Blockieren von I/O

... verlassen wird der Zustand wieder durch das Verstreichen der Wartezeit, den Ruf von `notify / notifyAll` bzw. Komplettierung der I/O-Operation.

```
1  public class PoliteRaceDemo {
2
3    private final static int NUMRUNNERS = 2;
4
5    public static void main(String[] args) {
6      PoliteRunner[] runners = new PoliteRunner[NUMRUNNERS];
7      for (int i = 0; i < NUMRUNNERS; i++) {
8        runners[i] = new PoliteRunner(i);
9        runners[i].setPriority(2);
10     }
11     for (int i = 0; i < NUMRUNNERS; i++)
12       runners[i].start();
13   }
14 }
15
16 public class PoliteRunner extends Thread {
17
```

```
18    private int tick = 1;
19    private int num;
20
21    public PoliteRunner(int num) {
22      this.num = num;
23    }
24
25    public void run() {
26      while (tick < 400000) {
27        tick++;
28        if ((tick % 50000) == 0) {
29          System.out.println("Thread #" + num + ", tick = " + tick);
30          yield();
31        }
32      }
33    }
34  }
```

**Aufgaben/Fragen:**
Wie macht man den Runner weniger höflich?
Wie macht man das Rennen unfair (siehe API unter Priorität)?

| **Methoden von Thread** | |
|---|---|
| public void start() | startet den Thread, die VM ruft run() |
| public void stop() | (deprecated) |
| public void run() | |
| public void interrupt() | setzt den Interruptstatus des Threads, bewirkt nichts weiter, nur das Werfen einer InterruptedException aus sleep und wait |
| public boolean isInterrupted() | liefert den Interruptstatus |
| public static boolean interrupted() | liefert den Interruptstatus des current Thread und setzt ihn zurück |
| static void sleep(int millis) | der Thread pausiert, das Werfen von InterruptedException setzt den Interruptstatus des Threads zurück |
| public final boolean isAlive() | liefert true, falls der Thread noch lebt |
| public final void join() | wartet auf das Ende diesesThreads |
| public void yield() | gibt aus purer Freundlichkeit die Aktivität ab |

**Übung 29** *Bauen Sie eine Graphikapplikation, die über einen Vector Threads verwaltet, die einmal pro Sekunde eine Ausschrift auf die Konsole schreiben. Über drei Buttons sollen über ein geeignetes Eventhandling*

- *ein neuer Thread instantiiert und der Verwaltung hinzugefügt werden,*
- *der letzte Thread aus der Verwaltung entfernt und beendet werden*
- *bzw. ein Interrupt an den letzten Thread geschickt werden.*

### 4.5.2 Threadsynchronisation

Ein **Monitor** ist ein Verfahren/Mechanismus zur Steuerung des ausschließlichen Zugriffs, eng damit verbunden ist der Begriff des wechselseitigen Ausschlusses (**Mutex** . . . mutual exclusion).
Das bedeutet beides im Prinzip die Kapselung eines **kritischen Abschnittes** in einer Klasse oder einem Objekt oder mehreren Objekten einer oder mehrerer Klassen zum Zwecke der Verhinderung des gleichzeitigen Mehrfachzugriffes. Zweck dieser Sache ist, zu gewährleisten, daß sich nur ein Thread (oder Prozeß) zu einem Zeitpunkt in diesem Bereich befinden kann. Dieser Bereich kann sich über mehrere getrennte Quelltext-Gebiete erstrecken, die Sperrinformation, d.h. die Information darüber, ob sich schon jemand (im Allgemeinen ein Thread) im Bereich befindet, ist immer an ein Objekt gebunden.
Für die Wahl des Objektes, auf welchem die Sperrinformation liegt, gibt es zwei Möglichkeiten:

**synchronized-Block:**

Syntax: `synchronized (varname) { ... }`

Sperrung auf die angegebene Variable/Objekt `varname`, hier ist die Information über das Betreten des Monitors abgespeichert, somit gilt, zeigt die Variablen `varname` getrennter synchronized-Blöcke auf das gleiche Objekt im Speicher, handelt es sich um einen gemeinsamen Monitor, unabhängig davon, ob sich die Bereiche in einem Objekt, mehreren Objekten, einer oder mehreren Klassen befinden.

**synchronized-Methode:**

Syntax: `synchronized typ methodname(...) { ... }`

Sperrung auf this (implizit), d.h. eine oder alle synchronized-Methoden sollen für eine Instanz (ein Objekt) zu bestimmter Zeit nur 1x laufen / betreten sein.

Daß die Sperrungsinformationen bei der angegebenen Variablen (bzw. this) beheimatet ist, bedeutet, daß alle synchronized-Methoden gemeinsam ein Monitor für das entsprechende Objekt sind,
ebenso alle synchronized-Blöcke, die mit einer Objektvariablen synchronisiert sind,
im Gegensatz dazu erstreckt sich der entsprechende Monitor auf alle Objekte der Klasse, wenn die Variable eine Klassenvariable (`static`) ist.
So lassen sich über synchronized-Blöcke beliebige, sich über verschiedene Objekte und auch Klassen erstreckende Monitore definieren. Z.B. kann ein Rechenthread im wait() warten, neue Rechenbefehle zu erhalten.

Auf der Sperrvariablen gibt es weitere Synchronisationsprimitive zur Kommunikation,
`wait()` ... mit der der Monitor temporär freigegeben werden kann oder
`notify()` und `notifyAll()` ... mit denen signalisiert werden kann, daß die Arbeit innerhalb des Monitors nun fertig gestellt ist. Dabei werden ein bzw. alle Threads, die `wait()` gerufen haben, geweckt. Wenn die Möglichkeit besteht, daß der geweckte Thread aufgrund irgendwelcher Bedingungen doch nicht weiterarbeiten kann/will, sollte `notifyAll()` gerufen werden!

**Aufgaben/Fragen:**

Was passiert, wenn Sie die while-Schleife beim etwas unfreundlicher gemachten PoliteRunner (s.oben) unendlich lange laufen lassen und in einen synchronized-Block einbetten, der eine Klassenvariable bzw. ein nur einmal existierendes Objekt als Sperrvariable verwendet?

**Beispiele:**
`StartThreads.java`
`StartThreadsIntTest.java`
`FrameCanvasAndThreads.java`

Was passiert bei diesem Beispiel? Welcher Designfehler wurde begangen?

```
1  public class Thread1 extends Thread {
2    public static int von;
3    public void run () {
4      String tmp = "192.168.151." + Thread1.von;
5      try {
6        con = new Socket(tmp, 3306);
7      }
8      catch (Exception e) {/*port ist zu*/ }
9    }
10
11   ... main ...
12
13     for (int i=1;i<255;i++) {
14       Thread1.von=i;
15       Thread thread = new Thread1();
16       thread.start();
17     }
18   ...
19 }
```

**Ein Beispiel zur Thread-Synchronisation:**

Eine Aufrufklasse:

```
public class ProducerConsumerTest {
    public static void main(String[] args) {
        Postfach c = new Postfach();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);

        p1.start();
        c1.start();
    }
}
```

Ein Postfach, welches kein Thread ist, aber über einen Monitor (synchronized-Methoden) verfügt. Mit der Synchronisation wird erreicht, daß nicht gleichzeitig mehrere Threads in put und get sind, was zu Datenfehlern führen könnte:

```
public class Postfach {
    private int contents;
    private boolean available = false;

    public synchronized int get() {
        while (available == false) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        available = false;
        System.out.println("Consumer got: " + value);
        notifyAll();
        return contents;
    }

    public synchronized void put(int value) {
        while (available == true) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        contents = value;
        available = true;
        System.out.println("Producer put: " + value);
        notifyAll();
    }
}
```

. . . und nun noch einen Produzenten, der Informationen im Postfach ablegt:

```
public class Producer extends Thread {
    private Postfach post;
    private int number;

    public Producer(Postfach c, int number) {
        post = c;
        this.number = number;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
                post.put(i);
            }
```

```
54              try {
55                  sleep((int)(Math.random() * 100));
56              } catch (InterruptedException e) { }
57          }
58      }
59  }
```

... und einen Konsumenten, der die Informationen aus dem Postfach abholt:

```
61  public class Consumer extends Thread {
62      private Postfach post;
63      private int number;
64
65      public Consumer(Postfach c, int number) {
66          post = c;
67          this.number = number;
68      }
69
70      public void run() {
71          int value = 0;
72          for (int i = 0; i < 10; i++) {
73              value = post.get();
74          }
75      }
76  }
```

**Übung 30** *Bauen Sie dieses Beispiel so um, daß das Postfach solange Strings entgegennehmen kann, bis eine Obergrenze erreicht ist! Auch in diesem Fall soll beim Schreibversuch in das volle und beim Leseversuch aus dem leeren Postfach gewartet werden, die entsprechenden Methoden für Producer und Consumer sind zu modifizieren! Testen und dokumentieren Sie das durch ein geeignetes Programm!*

Weiterführende Stichworte:
java.util.concurrent.atomic.AtomicInteger
Future
ThreadPool
CyclicBarrier

## 4.6 Netzwerkprogrammierung

### 4.6.1 Allgemeines

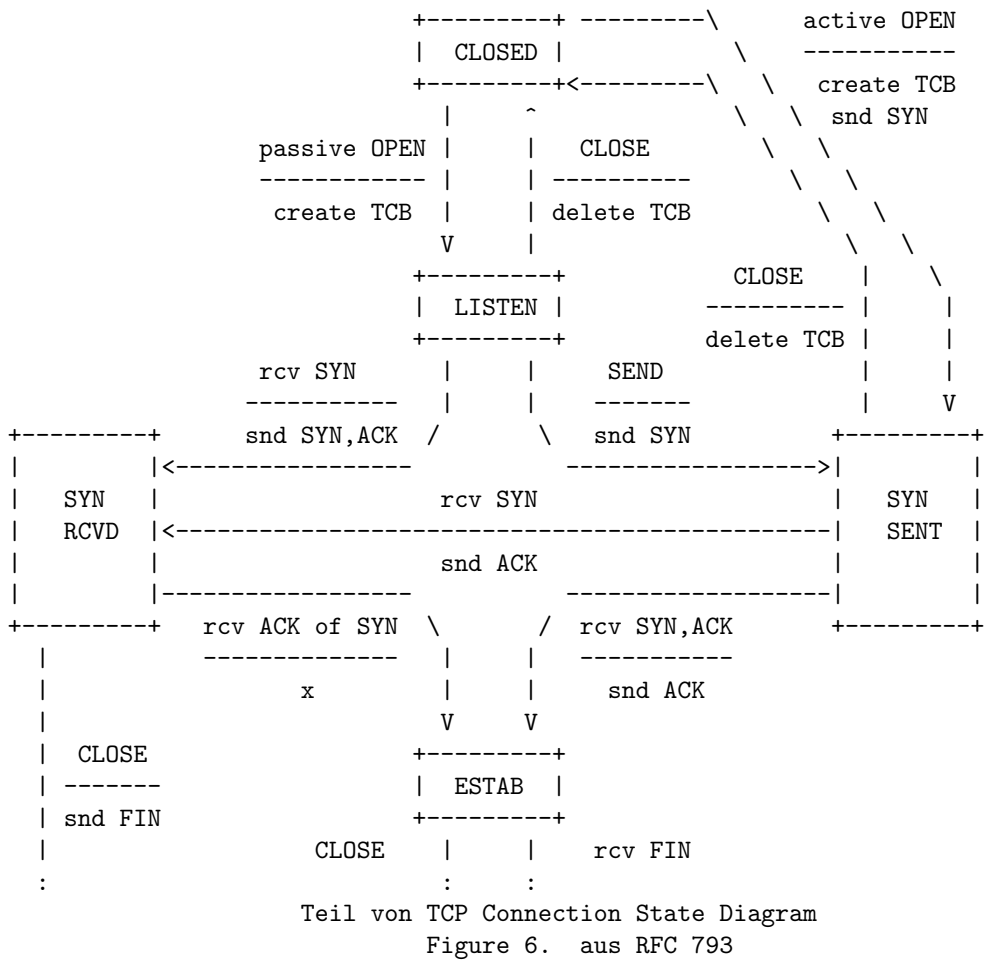
| |
|---|
| Schicht 7: Anwendungsschicht |
| Schicht 6: Darstellungsschicht |
| Schicht 5: Sitzungsschicht |
| Schicht 4: Transportschicht |
| Schicht 3: Netzwerksschicht/Vermittlungsschicht |
| Schicht 2: Leitungsschicht/Sicherungsschicht |
| Schicht 1: Physikalische Schicht |

ISO/OSI-7 Schichten-Modell

| |
|---|
| Schicht 4: Anwendungsschicht |
| Schicht 3: Transportschicht (TCP, UDP) |
| Schicht 2: Netzwerksschicht/Internetschicht (IP) |
| Schicht 1: Verbindungsschicht/Netzzugangsschicht |

Vereinfachtes 4-Schichten-Modell / TCP/IP-Referenzmodell

Abb./Begriffe nach [4] , [Wikipedia]
TCP ... Transmission Control Protocol (RFC 793), verbindungsorientiertes Protokoll
UDP ... User Datagram Protocol, verbindungsloses Protokoll
IP ... Internet Protocol (RFC 791)




Teil von TCP Connection State Diagram
Figure 6. aus RFC 793

wichtige Java-Klassen:

- `java.net.*` (Netzwerksprogrammierung)
  `DatagramPacket, DatagramSocket` (UDP)
  `ServerSocket` (TCP): wartet auf Netzwerks-Requests, entspricht dem LISTEN-Zustand des Zustandsdiagramms einer TCP-Verbindung
  `Socket` (TCP): implementiert einen Socket, entspricht dem ESTABLISHED-Zustand
  `URL`: liefert eine WWW-Ressource, kapselt TCP
- `java.io.*` (System-Input und Output mit Streams, Files, Serialisierung)
  `InputStream`
  `OutputStream`

### 4.6.2 TCP-Sockets

Zur Socketkommunikation wird zunächst ein **Server** benötigt, der im LISTEN-Zustand des Zustandsdiagramms verharrt (blockiert) und auf eine Clientanfrage wartet.

```
ServerSocket ss = new ServerSocket (PORT);
Socket con = ss.accept();
```

Das `accept()` gibt ein Socket-Objekt (bzw. eine Referenz auf ein solches) zurück, wenn ein **Client** Kontakt aufgenommen hat. Dieser kann die Anfrage absetzen, indem er einen Socket instanziiert, der Socketkonstruktor erhält als Parameter die erforderlichen Koordinaten:

```
Socket con = new Socket (HOST, PORT);
```

Geht hier etwas schief, fliegt eine Exception (siehe API-Dokumentation). Ist die Punkt-zu-Punkt-Verbindung etabliert (ESTABLISHED im Zustandsdiagramm), können den Sockets auf beiden Seiten nach In- und OutputStreams befragt werden

```
InputStream in = con.getInputStream();
OutputStream out = con.getOutputStream();
```

und die erhaltenen Referenzen für die Kommunikation verwendet werden. Diese kann mit den aus dem Abschnitt 4.3.1 (siehe S.38) bekannten Mechanismen geschehen.

Beispiele der Serverseite geben folgende Quelltexte, die Clientseite kann mit Telnet (verbinden mittels `telnet host portnummer` , 127.0.0.1 ist der localhost) realisiert werden.

Zunächst ein sequentieller (blockierender) Echo-Server für einen Client:

```
// Ein Echo-Server, der als Diener eines Herrn mit einem Client redet
// Abbruch mit ESC-Taste im Client
import java.net.*;
import java.io.*;
class ServerMono {
  static void main (String [] a) throws java.io.IOException{
    // fuer saubere Fehlerbehandlung:
    //   entfernen der throws-Klausel, compilieren,
    //   Fehler analysieren, try catch !!!
    int port = 1234;
    ServerSocket ss = new ServerSocket (port);
    while (true) {
      int i=0;
      System.out.println("wartet auf Anfrage");
      Socket con = ss.accept();
      // accept horcht, bis jemand auf dem Port anklopft
      System.out.println("neue Verbindung");
      InputStream in = con.getInputStream();
      OutputStream out = con.getOutputStream();
      while (i!=27) { // ESC
        char c= (char) (i=in.read());
        System.out.println("empfangen :" + c + ": , Wert = " + i);
        out.write(i);
      }
      con.close();
    }
    //ss.close();
    // Schließen des ServerSockets, wenn oben keine Endlos-Schleife existiert
  }
}
```

und nun einen multithreaded Server für parallele Anfragen:

```
// Ein multithreaded Echo-Server,
//  der mit mehreren Clients reden kann
import java.net.*;
import java.io.*;
class ServerMulti {
  static void main (String [] a) throws Exception{
    int port = 1234;
    ServerSocket ss = new ServerSocket (port);
    while (true) {
      int i=0;
      System.out.println("wartet auf Anfrage");
      Socket con = ss.accept();
      System.out.println("neue Verbindung");
      // wird an den ServerThread uebergeben:
      ServerThread sthr = new ServerThread(con);
      sthr.start();
      // und der ServerMulti kann wieder zum accept gehen
```

```
    }
    //ss.close();
    // Schließen des ServerSockets, wenn oben keine Endlos-Schleife existiert
  }
}
```

... die konkrete Arbeit übernimmt ein Thread, somit kann der Hauptthread wieder in den LISTEN-Zustand (`accept()`) gehen und weitere Clientanfragen erwarten.

```
class ServerThread extends Thread {
  // realisiert die konkrete Client-Verbindung
  Socket con = null;
  public ServerThread (Socket con){
    this.con = con;
  }
  public void run() {
    int i = 0;
    System.out.println("neuer ServerThread");
    try {
      InputStream in = con.getInputStream();
      OutputStream out = con.getOutputStream();
      while (i!=27) { // ESC
        char c= (char) (i=in.read());
        System.out.println("empfangen :" + c + ": , Wert = " + i);
        out.write(i);
      }
      con.close();
    }
    catch (java.io.IOException ioe) {
      System.out.println (ioe);
    }
  }
}
```

**Ein variables Serversystem**

Es soll ein Server entwickelt werden, der multiclientfähig ist, die konkrete ServerthreadKlasse soll zur Entwicklungszeit des Servers unbekannt sein und von der abstrakten Klasse ServerThread abgeleitet sein.
Der Server soll mit dem Aufruf
`java ServerMulti ServerThreadKlasse`
gestartet werden, der 1. Kommandozeilenparameter, der den Klassennamen enthält, muß zur Instantiierung eines Objektes verwendet werden. Das kann mit Reflection (siehe Abschnitt 3.3.1, S.24) passieren:

Die **Hauptklasse** des Servers:

```
import java.net.*;
import java.lang.reflect.*;

class ServerMulti {
  static boolean STOP = false;
  public static void main (String [] a) throws Exception {

    int port = 2345;
    String ThreadKlasse = a[0];
    Class  m = Class.forName(ThreadKlasse);

 //                                  {Class-Array für die KonstruktorparameterTypen}
    Constructor c = m.getDeclaredConstructor (new Class[] {Socket.class});
 //                                          {Class.forName("java.net.Socket")}

```

```
    // ein ObjektArray für die Konstruktorparameter:
    Object [] po = new Object [1];

    ServerSocket ss = new ServerSocket (port);
    while (!STOP) {
      int i=0;
      System.out.println("wartet auf Anfrage");
      Socket con = ss.accept();
      System.out.println("neue Verbindung");
      po[0] = con;
      // wird an den ServerThread uebergeben:
      ServerThread sthr = (ServerThread) c.newInstance(po);

      sthr.start();

    }
    ss.close();
  }
}
```

Die **abstrakte Oberklasse** aller Serverthreads:

```
import java.net.*;

abstract class ServerThread extends Thread {

  Socket con = null;

  public ServerThread (Socket con){
    this.con = con;
  }

  abstract public void run();

}
```

**Eine konkrete Serverthreadklasse** muß den Konstruktor mit dem Socket-Parameter implementieren, da dieser nicht automatisch aus der Oberklasse übernommen wird:

```
import java.net.*;
import java.io.*;

class ServerThreadCharEcho extends ServerThread {
  // realisiert die konkrete Client-Verbindung

  public ServerThreadCharEcho (Socket con){
    super(con);
  }

  public void run() {
    int i = 0;
    System.out.println("neuer ServerThread");
    try {
      InputStream in = con.getInputStream();
      OutputStream out = con.getOutputStream();
      System.out.println("Klasse des InputStreams : "+in.getClass().getName());
      System.out.println("Port (remote) : "+con.getPort());
      System.out.println("Port (local) : "+con.getLocalPort());
      while ((i=in.read())!=-1) { // oder 27 für ESC
        char c= (char) i;
```

```
22          System.out.println("empfangen :" + c + ": , Wert = " + i);
23          out.write(i);
24        }
25        System.out.println("Ende ServerThread");
26        con.close();
27      }
28      catch (java.io.IOException ioe) {
29        System.out.println (ioe);
30      }
31    }
32  }
```

**Übung 31** *Bauen Sie den Server so um, daß er die Zeichen vom Client zeilenweise liest und zurückschickt! (siehe Abschnitt 4.3.1 S.38)*

**Übung 32** *Modifizieren Sie den Server aus Übung 31 so, daß er vor den zurückgegebenen String eine Zeilennummer schreibt und die Verbindung beendet, wenn ihm QUIT (bei beliebiger Groß/Kleinschreibung) geschickt wird!*

## 4.7 Datenbankprogrammierung mit JDBC

Verschaffen Sie sich

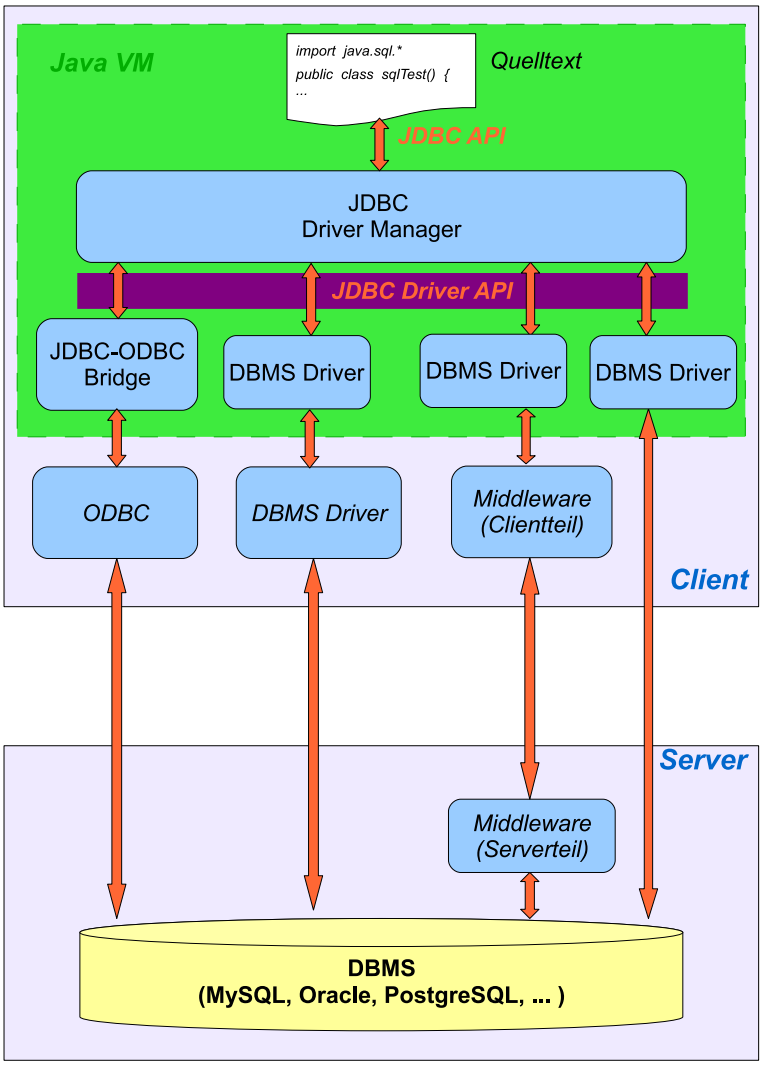
- anhand des Trails JDBC Database Access im Java-Tutorial [1]
- oder über das Kapitel Datenbankzugriffe mit JDBC in Handbuch der Javaprogrammierung [4]

einen Überblick über Datenbankzugriffe mit Java.

Java Database Connectivity (JDBC) ist ein Call-Level-Interface (CLI) zur Datenbank. Eine Alternative dazu wäre Embedded SQL, das sich in Java nicht durchgesetzt hat.

4 Typen von JDBC-Treibern

1. die JDBC-ODBC-Bridge
2. ein JDBC-Treiber, der (clientseitig) auf die proprietären Datenbanktreiber zugreift
3. ein JDBC-Treiber, der über eine entsprechende Middleware (z.B. auch Java-Netzwerkkommunikation mit Gegenstück auf dem Server) serverseitig auf die proprietären Datenbanktreiber zugreift
4. ein JDBC-Treiber, der direkt mit der Datenbank in derem Protokoll spricht




Wichtige Klassen bzw. Schnittstellen im Paket `java.sql.*`:

- `DriverManager`
- `Connection`
- `Statement`

- `PreparedStatement`
- `ResultSet`
- `DatabaseMetaData`
- ...

### Ein simples Beispiel zur Illustration

Zunächst werden die benötigten Treiber geladen (Zeile 5,6), die später durch den `DriverManager` entsprechend der angegebenen URL (Zeile 17ff.) ausgewählt werden.

```
1  import java.sql.*;
2  ...
3  ...
4  try {
5    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
6    // Class.forName("org.gjt.mm.mysql.Driver");
7  } catch(java.lang.ClassNotFoundException e) {
8    System.err.println(e.getMessage());
9  }
```

... die nächsten Zeilen zeigen zunächst, welche Treiber zur Verfügung stehen (zum Verständnis siehe API-Dokumentation). Danach (im `try`-Block) stellt der `DriverManager` eine Verbindung (`Connection`) her, auf diesem Objekt wird eine Anweisung (`Statement`) geschaffen, welche dann mit einem übergebenen SQL-String abgearbeitet wird:

```
11 for (Enumeration e = DriverManager.getDrivers() ; e.hasMoreElements() ;) {
12   System.out.println("driver: " + e.nextElement());
13 }
14
15 try {
16   Connection con;
17   Statement stmt;
18   String url = "jdbc:odbc:sample-MySQL";
19   // String url = "jdbc:mysql://localhost/mysql" ;
20
21   // jdbc:<subprotocol>:<subname>
22   // <subprotokol> = odbc | mysql | ...
23   // <subname> ist abhaengig vom subprotokoll
24   // siehe dazu den JDBC-Teil in der Doku zum jdk
25
26   con = DriverManager.getConnection(url, username, password);
27   System.out.println ("Ok, connection to the DB worked.");
28   stmt = con.createStatement();
29   ResultSet rs = stmt.executeQuery(" SELECT * FROM user;");
30   while (rs.next()) {
31     String s1 = rs.getString(1);
32     ...
33   }
34   stmt.close();
```

... hier wurde exemplarisch die erste Spalte jeder Ergebniszeile ausgedruckt, ...
... und danach kommt der `catch`-Block mit der Besonderheit von `SQLExceptions`, daß an einer eine ganze Kette weiterer hängen kann. Diese Kette wird innerhalb einer `while`-Schleife bis zum Ende durchlaufen:

```
36 } catch(SQLException ex) {
37   System.err.println("==> SQLException: ");
38   while (ex != null) {
39     System.out.println("Message: " + ex.getMessage ());
```

```
40      System.out.println("SQLState: " + ex.getSQLState ());
41      System.out.println("ErrorCode: " + ex.getErrorCode ());
42      ex = ex.getNextException();
43      System.out.println("");
44    }
45  }
46  ...
```

Dem `executeQuery(...)` eines `Statement` kann ein (variabler) String übergeben werden, dabei wird jedes mal eine Syntaxprüfung durchgeführt. Vorteile eines `PreparedStatement` sind , daß

- diese Syntaxprüfung nur einmal stattfindet
- und mit Platzhaltern variable Parameter generiert werden können,

was beides wichtig für Massendatenverarbeitung ist. Die folgenden Zeilen zeigen ein Beispiel:

```
1   PreparedStatement pstmt = con.prepareStatement("UPDATE EMPLOYEES
2                                SET SALARY = ? WHERE ID = ?");
3   pstmt.setBigDecimal(1, 153833.00);
4   pstmt.setInt(2, 110592);
5   pstmt.execute();
```

Je nachdem, ob man ein `INSERT, UPDATE, DELETE` oder aber eine Query absetzen will, nutzt man:
`boolean execute()` oder
`ResultSet executeQuery()`
auf dem `Statement` oder `PreparedStatement`

**Übung 33** *Schaffen Sie sich die Möglichkeit eines Datenbankzugriffs auf Ihrem PC! Als einfache Variante bietet sich die Kombination aus JDBC-ODBC-Bridge, ODBC-Treiber und Access- oder mySQL-Datenbank an. Testen Sie auch den speziellen MySQL-Treiber in verpackter Form (mysqldriver.jar)! (Setzen des CLASSPATH!) Schaffen Sie (mit dem DBMS, nicht über Java) eine Datenbank mit einer einfachen Tabelle (Spalten: Name, Telefonnummer) und geben Sie einige Datensätze ein!*

1. *Programmieren Sie dazu ein einfaches Java-Programm , mit dem Sie alle Datensätze ausgeben!*
2. *Als Nächstes ein einfaches Programm zur Eingabe neuer Datensätze!*
3. *Kombinieren Sie Ein- und Ausgabe der Datensätze in einem GUI!*

**Übung 34** *Entwickeln Sie eine Methode*
*`public static String [][] getTableAsStringArray(Connection con, String tablename)`,*
*die auf einer beliebigen Tabelle die Spaltennamen (in der 0. Zeile) und alle Datensätze (Werte als String) zurückgibt, nutzen Sie Metadaten! Testen Sie diese Methode durch Aufruf in der main-Methode!*

# Literatur


[1] *The Java Tutorial*. `http://java.sun.com`.

[2] F. Jobst, *Programmieren in Java*, Carl Hanser Verlag, 2001.

[3] G. Krüger, *Go To Java2*, Addison Wesley, 2000.

[4] G. Krüger, *Handbuch der Java-Programmierung*, Addison Wesley, 2002.

[5] P. Niemeyer and J. Knudsen, *Learning Java*, O'Reilly, 2000.

[6] C. Ullenboom, *Java ist auch eine Insel*, Galileo Computing, 2006.

# Inhaltsverzeichnis


**1 Einführung** **1**
- 1.1 Die Geschichte von Java . . . 2
- 1.2 Eigenschaften von Java . . . 2
- 1.3 Installation und Werkzeuge . . . 2

**2 Elemente der Sprache Java** **3**
- 2.1 Struktur eines Java-Programms . . . 3
- 2.2 Variablen . . . 6
  - 2.2.1 Elementare Datentypen und Strings . . . 6
  - 2.2.2 Deklaration . . . 7
  - 2.2.3 Literale . . . 7
- 2.3 Felder in Java . . . 8
- 2.4 Methoden . . . 10
  - 2.4.1 Deklaration . . . 10
  - 2.4.2 Überladen (to overload) von Methoden . . . 11
- 2.5 Kontrollfluß . . . 12
  - 2.5.1 Die Verzweigung (if else) . . . 12
  - 2.5.2 Mehrfach-Verzweigung (switch) . . . 12
  - 2.5.3 Die abweisende Schleife (while) . . . 12
  - 2.5.4 Die nicht-abweisende Schleife (do while) . . . 13
  - 2.5.5 Die Zählschleife (for) . . . 13
  - 2.5.6 Die erweiterte for-Schleife (Enhanced for statement) . . . 14
  - 2.5.7 Verlassen von Schleifen mit break und continue . . . 14
  - 2.5.8 Ausnahmebehandlung . . . 14
- 2.6 Operatoren . . . 18

**3 Objektorientierung** **19**
- 3.1 Grundbegriffe . . . 19
- 3.2 Sichtbarkeit . . . 19
- 3.3 Klassen in Java . . . 20
  - 3.3.1 Das Paket Reflection . . . 24
- 3.4 Schnittstellen (Interfaces) . . . 26
- 3.5 Packages und Archiv-Tool jar . . . 26
- 3.6 javadoc . . . 27

**4 Spezielle Klassen und Probleme** **28**
- 4.1 Grundlegende graphische Anwendungen . . . 28
  - 4.1.1 Applets . . . 28
  - 4.1.2 Frames . . . 28
  - 4.1.3 Fensterklassen, Componenten, Container, Button, ... . . . 29
  - 4.1.4 Das Layout . . . 29
  - 4.1.5 Ein zusammenfassendes Beispiel . . . 30
- 4.2 Eventhandling . . . 31
  - 4.2.1 Begriffe . . . 31
  - 4.2.2 Implementation eines ListenerInterfaces . . . 32
  - 4.2.3 Beauftragung einer inneren Klasse . . . 33
  - 4.2.4 Menues und ActionEvents . . . 35
- 4.3 Ein- und Ausgabe . . . 38
  - 4.3.1 Die Nutzung der Stream-Klassen . . . 38
  - 4.3.2 Ein GUI-Beispiel (unter dem AWT) . . . 41
  - 4.3.3 Serialisierung . . . 42
- 4.4 Containerklassen und Stringhandling . . . 43
  - 4.4.1 Containerklassen . . . 43
  - 4.4.2 StringTokenizer, String, ... . . . 45
  - 4.4.3 Reguläre Ausdrücke . . . 45
- 4.5 Threadprogrammierung . . . 46
  - 4.5.1 Einfache Threads . . . 46
  - 4.5.2 Threadsynchronisation . . . 48
- 4.6 Netzwerkprogrammierung . . . 51
  - 4.6.1 Allgemeines . . . 51

4.6.2 TCP-Sockets . . . . . 52
4.7 Datenbankprogrammierung mit JDBC . . . . . 57