

# Grundlagen der Programmierung 2

## - die Programmiersprache Java

D\_KMI/AI\_2017

Matthias K. Krause  
Hochschule für Telekommunikation Leipzig

25. Januar 2018

(c: k-tki-gpr2 )

**Prüfungsbedingungen:** Die Modulprüfung wird als 90 minütige Klausur durchgeführt. Es werden Anteile von C und Java geprüft. Als **Hilfsmittel** ist ein beidseitig handbeschriebenes Blatt A4 („Spickzettel“) erlaubt.

Voraussetzung für die Teilnahme an der Prüfung ist das Belegen der PVL, die mit den parallel zur Vorlesung laufenden Übungen in Semester 1 und 2 verbunden sind.

Dieses Skript soll im Rahmen von Vorlesungen zur Programmierung benutzt werden und den Hörenden helfen, weniger abschreiben zu müssen. Das Skript ersetzt nicht das Lesen von Literatur wie

- Handbuch der Javaprogrammierung [5]

- Java ist auch eine Insel [7]

- und dem Javatutorial [3]

und die selbständige kreative Beschäftigung mit der Programmierung.

### Zeitplan für das Modul (d\_ai/kmi)

- **4.Präsenz FH (11.KW, ca. 3 UE):**

Einführung (Grundlagen, Syntax)

Klassen/Objekte

Ausnahmebehandlung

**Selbststudium** (Syntax, Arrays, Ausnahmebehandlung, OOP)

- **AZ-Phase (14.KW:** Übungen (OOP, Algorithmierung, Ausnahmebehandlung), PVL-Aufgaben

- **5.Präsenz FH (16.KW, ca. 3 UE):**

Tools, Interfaces,

Eventhandling und Graphikprogrammierung, Menues,

Streams, Java Collections Framework (JCF)

**Selbststudium** (OOP, Tools (jar, javadoc), Collections, Graphik/Event, Multithreading)

- **AZ-Phase (19.KW):** Übungen, PVL-Aufgaben

- **6.Präsenz FH (22.KW, ca. 3 UE):**

Reguläre Ausdrücke (RE), Multithreading (mit Synchronisation)

**Selbststudium** (RE, Multithreading, Prüfungsvorbereitung)

- **2.Prüfungswoche FH (26.KW, MP):** Modulprüfung (C und Java)

# Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>4</b>
1.1	Die Geschichte von Java . . . . .	4
1.2	Eigenschaften von Java . . . . .	5
1.3	Installation und Werkzeuge . . . . .	6
<b>2</b>	<b>Elemente der Sprache Java</b>	<b>7</b>
2.1	Struktur eines Java-Programms . . . . .	7
2.2	Variablen . . . . .	10
2.2.1	Elementare Datentypen und Strings . . . . .	10
2.2.2	Deklaration . . . . .	10
2.2.3	Literale . . . . .	11
2.3	Felder (Arrays) in Java . . . . .	12
2.4	Methoden . . . . .	14
2.4.1	Deklaration . . . . .	14
2.4.2	Überladen (to overload) von Methoden . . . . .	16
2.5	Kontrollfluß . . . . .	16
2.5.1	Die Verzweigung (if else) . . . . .	16
2.5.2	Mehrfach-Verzweigung (switch) . . . . .	17
2.5.3	Die abweisende Schleife (while) . . . . .	17
2.5.4	Die nicht-abweisende Schleife (do while) . . . . .	18
2.5.5	Die Zählschleife (for) . . . . .	18
2.5.6	Die erweiterte for-Schleife (Enhanced for statement) . . . . .	19
2.5.7	Verlassen von Schleifen/Verzweigungen mit break und continue . . . . .	19
2.5.8	Ausnahmebehandlung . . . . .	20
2.6	Operatoren . . . . .	23
<b>3</b>	<b>Objektorientierung</b>	<b>26</b>
3.1	Grundbegriffe . . . . .	26
3.2	Sichtbarkeit . . . . .	27
3.3	Klassen in Java . . . . .	27
3.3.1	Das Paket Reflection . . . . .	33
3.4	Schnittstellen (Interfaces) . . . . .	35
3.5	Packages und Archiv-Tool jar . . . . .	37
3.6	javadoc . . . . .	37
<b>4</b>	<b>Spezielle Klassen und Probleme</b>	<b>39</b>
4.1	Grundlegende graphische Anwendungen . . . . .	39
4.1.1	Applets . . . . .	39
4.1.2	Frames . . . . .	39
4.1.3	Fensterklassen, Componenten, Container, Button, ... . . . .	40
4.1.4	Das Layout . . . . .	40
4.1.5	Ein zusammenfassendes Beispiel . . . . .	41
4.1.6	Ein wenig Swing . . . . .	42
4.2	Eventhandling . . . . .	43
4.2.1	Begriffe . . . . .	43
4.2.2	Implementation eines ListenerInterfaces . . . . .	43
4.2.3	Beauftragung einer inneren Klasse . . . . .	45
4.2.4	Menues und ActionEvents . . . . .	47
4.3	Ein- und Ausgabe . . . . .	49
4.3.1	Die Nutzung der Stream-Klassen . . . . .	49
4.3.2	Ein GUI-Beispiel (unter dem AWT) . . . . .	53
4.3.3	Serialisierung . . . . .	54
4.4	Containerklassen und Stringhandling . . . . .	57
4.4.1	Containerklassen - Java Collections Framework . . . . .	57
4.4.2	StringTokenizer, String, ... . . . .	59
4.4.3	Reguläre Ausdrücke (Regular Expressions ... RE) . . . . .	60
4.5	Threadprogrammierung . . . . .	64
4.5.1	Einfache Threads . . . . .	64
4.5.2	Threadsynchronisation . . . . .	66
4.6	Netzwerkprogrammierung . . . . .	71

4.6.1	Allgemeines . . . . .	71
4.6.2	TCP-Sockets . . . . .	72
4.6.3	Ein variables Serversystem . . . . .	74
4.7	Datenbankprogrammierung mit JDBC . . . . .	76

# 1 Einführung

Anliegen der Lehrveranstaltung ist

- das Kennenlernen der Sprache Java mit ihrer Syntax
- unter Nutzung der Kommandozeile und einer IDE (z.B. Eclipse),
- das Kennenlernen der Arbeit mit dem Java-API,
- Algorithmierung im statischen (rein prozeduralen) und Objekt-Kontext,
- Ausnahmebehandlung,
- Objektorientierung und Datenkapselung,
- Programmierung graphischer Oberflächen (GUI) und Eventhandling,
- Streams,
- reguläre Ausdrücke,
- Multithreading (mit Synchronisation)

Darüber hinausgehende Themengebiete wurden zur Information und zum Weiterlesen im Text des Skriptes belassen, werden aber nicht behandelt und geprüft.

Literatur:

- Handbuch der Javaprogrammierung [5],  
html-Version auf [www.javabuch.de](http://www.javabuch.de)
- Java ist auch eine Insel [7],  
html-Version auf <http://openbook.galileocomputing.de/javainsel7/>
- Javatutorial [3]  
und andere Dokumentationen (engl.)
- Java Code Conventions (auf [java.sun.com](http://java.sun.com)) (engl.)
- Learning Java [6] (engl.)
- <http://www.gailer-net.de/tutorials/java5/index.html> (ein Studententip)

## 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 1.8 (Java 8)

## 1.2 Eigenschaften von Java

Java ist eine objektorientierte Programmiersprache, deren Code in Klassen strukturiert ist und auf Unicode ([www.unicode.org](http://www.unicode.org)) basiert. Für die interne Darstellung wird UTF-16 verwendet (siehe z.B. [7, Kap.4]). Das ist ein 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.

(eine Info am Rande 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 hat in seinem API umfangreiche Features integriert, die in anderen Programmiersprachen als separate Pakete “irgendwie“ hinzugeholt/besorgt werden müssen:

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

Als wesentlicher Nachteil von Java erwies sich früher der Performanceverlust durch die VM (ca. 1/15 der Geschwindigkeit gegenüber C/C++ bei Solaris). Durch JIT(just in time)-Compiler (seit Java 1.1) und das HotSpot-System (seit Java 1.2) wird dieser Performanceverlust ausgeglichen, die Performance erreicht die nativer Sprachen (siehe auch [https://en.wikipedia.org/wiki/Java\\_performance](https://en.wikipedia.org/wiki/Java_performance)).

Durch die Plattformunabhängigkeit lassen sich auch nicht alle Systemvorteile ausreizen.

Auf der anderen Seite sind natürlich Plattformunabhängigkeit, Standardisierung, Netzwerkfähigkeit ganz entscheidende Vorteile von Java, nicht zu vergessen die über `javadoc` aus Quelltextkommentaren erzeugte Dokumentation, die eine wesentliche Stärke von Java darstellt.

### 1.3 Installation und Werkzeuge

Java erhält man auf den Seiten von Sun bzw. Oracle (seit der Übernahme 2010) ([java.sun.com](http://java.sun.com) bzw. <http://www.oracle.com/technetwork/java/index.html>):

- ein aktuelles JDK der Java SE (Standard Edition) für ihre entsprechende Plattform, es kann die aktuelle Version 8, aber auch die Versionen 6/7 der Java SE verwendet werden, in jedem Fall ist ein JDK und nicht nur eine JRE herunterzuladen
- und unbedingt die dazugehörige Java Dokumentation (incl. API-Beschreibung)
- und ev. ein Java Tutorial

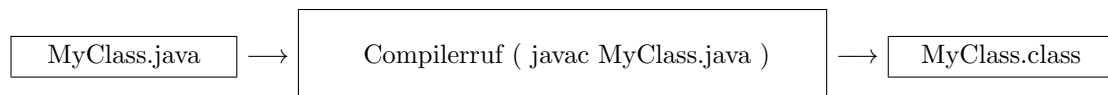
und Massen anderer Infos

Java-Werkzeuge des JDK sind:

- Compiler: `javac`
- Interpreter: `java`
- Appletviewer: `appletviewer`
- Debugger: `jdb`
- „Disassembler“: `javap`
- Dokumentationsgenerator: `javadoc`
- Archiv-Tool: `jar`
- C-Header-Generator für JNI: `javah`

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

- Übersetzen des Javaquelltextes (der Klasse `MyClass`) in Java-Bytecode



- und Abarbeitung der Klasse `MyClass` in einer virtuellen Maschine (VM, JVM)



Zur bequemerer Bedienung könnte man jetzt eine IDE wie Eclipse oder Netbeans installieren, oder etwas Kommerzielles.

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
- ...

## 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
"class"	die Zeichenfolge class
[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.

<pre> Java-Quelltext := { kommentar                       leerzeichen                       sprachelement                   } leerzeichen := space   tab   CR   LF   FF  sprachelement := schlusselwort                     bezeichner (identifizier)                     literal                     separator                     operator  bezeichner := buchstabe [ buchstabenoderzahlenfolge ]  buchstabe := "a" .. "z"   "A" .. "Z"   umlaute   ...   "\$"   "_" separator := ";"   ","   "{"   "}"   ... </pre>
--

## reservierte Wörter (Schlüsselwörter)

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

\* not used  
\*\* added in 1.2  
\*\*\* added in 1.4  
\*\*\*\* added in 5.0

(siehe [https://docs.oracle.com/javase/tutorial/java/nutsandbolts/\\_keywords.html](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html))

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 Paket `java.lang.*` importiert)

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

```
6 class MyClass {
7     ...
8 }
9
10 interface MyInterface {
11     ...
12 }
```

(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

Syntax (EBNF):

```
klassendefinition :=  
{modifier} "class" klassenname  
                        ["extends" oberklasse]  
                        ["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, welche 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 Kommandozeile auch anzugeben,

z.B. so: `java PrintArg firstArgument`

dann fliegt uns natürlich eine Exception um die Ohren. (siehe dazu 2.5.8, S.20 )

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

Der **default**-Wert ist immer 0, 0.0 oder false (Bits auf 0).

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`)

```
1 ...
2 System.out.println(Float.MIN_VALUE + " " + Float.MAX_VALUE);
3 System.out.println(Double.MIN_VALUE + " " + Double.MAX_VALUE);
4 ...
```

Die verwendeten Variablen sind **static**-Member der Klassen (s. 2.2.2, S.11)

#### Die Klasse String

Die Klasse `String` dient zur Aufbewahrung von Zeichen. Sie beschreibt Objekte, in die ein Array von `char` eingebettet ist, dessen Größe unveränderbar ist.

Eine wichtige Operation mit Operanden vom Typ `String` ist `+` (Stringverknüpfung/Concatenation).

### 2.2.2 Deklaration

Syntax (EBNF):

```
{modifier} typename variablenname [initialisierung] [ , ... ] ;

modifier = sichtbarkeitsmodifier |
           "static" |
           "final"

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

Ein paar kleine Beispiele:

```
1 static double PI = 3.14159;
2 int i,j,k; // macht 3 Speicherzellen fuer int-Zahlen
3           // Initialisierung VOR erstem Lesen erforderlich!
4
```

```

5 MyClass myvar; // macht eine Referenzvariable auf ein Objekt vom Typ MyClass,
6               // enthaelt null
7 MyClass myvar2 = new MyClass(); // macht eine Referenzvariable auf ein
8                               // 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 bzw. Member 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 Klassennamens 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 allerdings nie auf ein anderes Objekt zeigen)

### 2.2.3 Literale

#### Zahlenlitterale

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

#### Zeichenlitterale (Darstellung in Hochkommata(quotes): 'literal' !!)

gewöhnliches Literal	k, r
oktale Darstellung	\377, \6
Unicode-Darstellung	\u0030
Escape-Sequenzen	\n, \b, \r, \t, \f, , \"
String	\"zeichenlitteralfolge\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 „komischen“ 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`! Sind die Zahlen gleichgroß, wenn nein, warum nicht?  
Warum sind `1.25` und `1.25f` gleichgroß?

## 2.3 Felder (Arrays) 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 [4, S.58])

### Deklaration mit Speicherbereitstellung und default-Initialisierung

**Elementare Datentypen** mit **static**-Modifier werden mit default (siehe 2.2.1, S.10) 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 [] f = null; // f enthaelt null
int [] f = { }; // f zeigt auf ein Feld der Laenge 0
int [] f = { 1, 2 }; // f zeigt auf ein Feld der Laenge 2

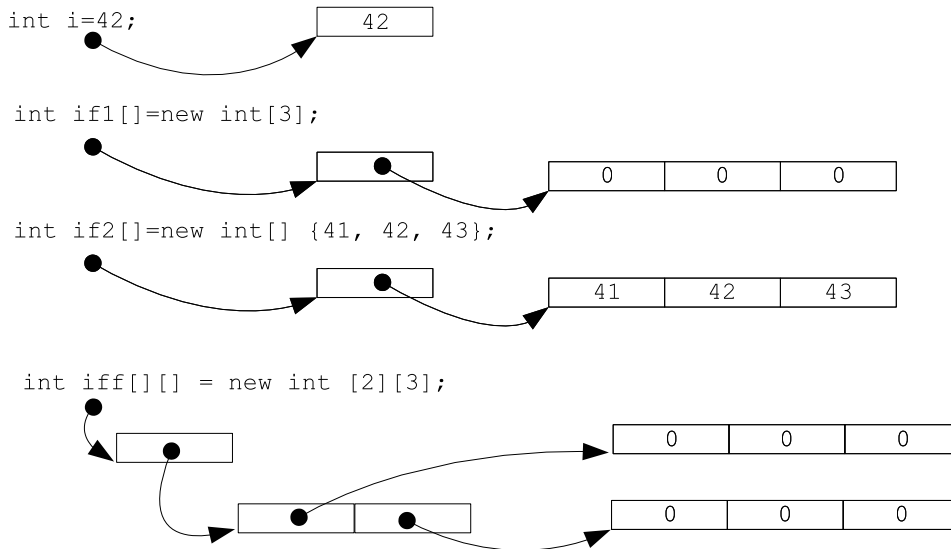
int i[] [] = new int[] [] {{1,2,3},{4,5,6,7},{8,9}};
int i[] [] = {{1,2,3},{4,5,6,7},{8,9}};
Classname c[] = {new Classname (Parms),
                 new Classname (Parms) }
```

Zu Variablen und Arrays in Java gelten einige Regeln ...

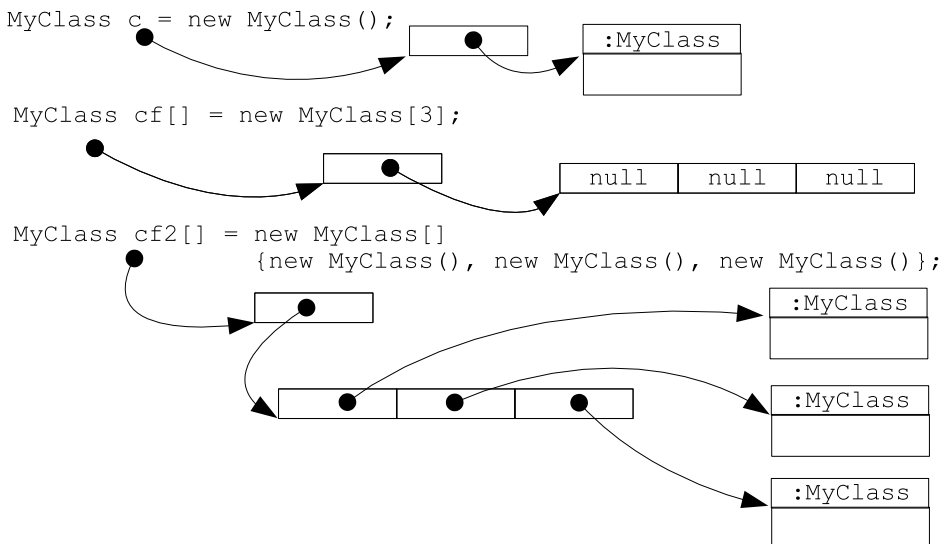
- Eine Variable/Speicherzelle, die einen elementaren Datentyp repräsentiert, enthält **immer** die entsprechenden Daten.
- Eine Variable/Speicherzelle, die einen abstrakten Datentyp (also ein Objekt) repräsentiert, enthält **immer** eine Referenz auf das entsprechende Objekt.
- Eine Variable/Speicherzelle, die ein Array repräsentiert, enthält **immer** eine Referenz auf das entsprechende Array.
- Jedes Array liegt im Speicher als eindimensionales Array vor, mehrdimensionale Arrays dann als Kaskaden eindimensionaler Arrays, die letzte Dimension eines mehrdimensionalen Array entspricht einem eindimensionalen Array des entsprechenden Datentyps.
- Ein eindimensionales Array eines elementaren Datentyps enthält als Elemente die Daten, ein eindimensionales Array eines abstrakten Datentyps die entsprechenden Referenzen auf die Objekte.

..., die durch folgende Abbildungen veranschaulicht werden:

## Elementare Datentypen und ihre Arrays



## Abstrakte Datentypen und ihre Arrays



Wo liegen die skizzierten Variablen, Arrays und Objekte? Hierbei muß man die Variablen, die immer elementare Datentypen oder Referenzen auf Arrays oder Objekte enthalten, und ebendiese Arrays und Objekte unterscheiden:

- Alle Objekte und Arrays liegen auf dem Heap (globaler dynamischer Speicher),
  - alle Objektvariablen liegen im Speicherbereich des entsprechenden Objektes,
  - die Klassenvariablen liegen im Speicherbereich des entsprechenden Klassenbeschreibungs-Objektes, da sie ja nur einmal pro Klasse existieren,
  - alle lokalen (innerhalb von Methoden definierten) Variablen liegen auf dem Stack des jeweiligen Methodenrufes, der zum Zeitpunkt des Aufrufes angelegt und beim Verlassen der Methode wieder frei gegeben wird.
- Der Stack ist ein linearer Speicherbereich, der von oben (bei hohen Adressen) beginnend alle Daten jedes Methodenrufes enthält, solange die Methode nicht beendet wurde.

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 Pascal!

## 2.4 Methoden

### 2.4.1 Deklaration

Syntax (EBNF):

```
methodendekl := {modifier} typename {"[]" } methodenname "(" [parameterliste] ")"
               {"[]" }
               ["throws" ausnahmentypliste]
               ( "{" anweisungsfolge "}" | ";" )

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

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

typename := EDT (Standard-Java-Typ) | ADT (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. Das wird in abstrakten Klassen gemacht, die nur als Oberklassen in Klassenhierarchien dienen, von denen selbst aber keine Objekte instanziiert werden dürfen.

**Finale Methoden** (**final**) dürfen in Unterklassen nicht überschrieben werden

Der Typ des **Rückgabewertes** kann `void`, ein elementarer Datentyp, ein abstrakter Datentyp (Klassenname) oder ein Array eines beliebigen Datentyps sein. Im Quelltext beendet

```
return [rueckgabewert]
```

die Abarbeitung, das ist zwingend in Methoden anzugeben, 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     }
8 }
```

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

```
1 public class ArrayMethod {
2     public static void main (String[] args) {
3         double [] feld = macheFeld(100);
4         ...
5     }
6     public static double [] macheFeld (int laenge) {
7         double f [] = new double[laenge];
8         for (int i=0; i < f.length; i++) f[i] = ... ; // Initialisierung
9         return f;
10    }
11    public void sortiereFeld(double [] feld) {
12        // hier werden die Elemente feld[i] ( i = 0 .. feld.length-1 ) sortiert
13    }
14 }
```

**Übung 4** Skizzieren Sie die Lage der lokalen Variablen `feld` (in `main(...)` und `macheFeld(...)` und des Arrays, welches außerhalb der Methoden angelegt wird (in Anlehnung an die Skizzen zu Arrays im Speicher (S.13), der Stackbereich der Methoden kann als Kasten symbolisiert werden, der wiederum Variablen enthalten kann)!

Bei der **Parameterübergabe** wird **immer** die übergebene Variable (der aktuelle Parameter) beim Aufruf in die Parameterspeicherzelle auf dem Stackbereich der Methode (formaler Parameter) **kopiert**.

- Handelt es sich bei dem Parameter um einen elementaren Datentyp, wird der Wert in die Methode kopiert, kann hier verwendet werden, gelangt aber nicht wieder nach außen (**Call by value**).
- Handelt es sich dagegen bei dem Parameter um eine Objekt- oder Arrayvariable, die immer Referenzen (also Adressen) sind, wird diese Adresse in die Methode kopiert, kann hier verwendet werden, gelangt aber auch nicht wieder nach außen. Über diese Adresse kann aber zum existierenden Objekt oder Array gegangen und dort ggf. auch Änderungen durchgeführt werden (**Call by reference**).

**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 Rückgabotyp 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!

```

1  /*
2   Typen der Literale: 3L 4l ... long-Literale
3                       1 2 5 ... int-Literale
4  */
5
6  public class Ueberladen {
7      public static void main (String[] args) {
8          // max-Rufe mit int- bzw. long-Literalen
9          System.out.println(max(1,2)); // int, int
10         System.out.println(max(1l,2)); // long, int
11         System.out.println(max(1,2l)); // int, long
12         System.out.println(max(1l,2l)); // long, long
13     }
14     public static long max (long z1, long z2){
15         System.out.println("long max(long "+z1+", long "+z2+"");
16         return z1>z2?z1:z2;
17     }
18     public static long max (long z1, int z2){
19         System.out.println("long max(long "+z1+", int "+z2+"");
20         return z1>z2?z1:z2;
21     }
22     public static long max (int z1, long z2){
23         System.out.println("long max(int "+z1+", long "+z2+"");
24         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
Ausnahmehandlung	try, catch, finally

### 2.5.1 Die Verzweigung (if else)

Syntax (EBNF):

```

"if" "(" bedingung ")"
    anweisung1
["else"
    anweisung2]

```

bedingung                := boolscher Ausdruck



```

anweisungX      := anweisung
anweisung       := einzelanweisung
                  | verbundanweisung
einzelanweisung := ausdruck ";"
                  | if-Anweisung
                  | schleife
                  | ...
verbundanweisung := "{" anweisungsfolge "}"
anweisungsfolge := { anweisung }

```

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

ein kleines Codebeispiel:

```

1 int i = ...;
2 if (i==42) {
3     System.out.println("Das ist die Antwort!!!");
4     ...
5 } else {
6     System.out.println("Das war doch nix!");
7 }

```

### 2.5.2 Mehrfach-Verzweigung (switch)

Syntax (EBNF):

```

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

```

konstX := Ganzzahl-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

ein kleines Codebeispiel:

```

1 int note = ...; // eine Zahl zwischen 1 und 5
2 switch (note) {
3     case 1: System.out.println("Das ist sehr gut!"); break;
4     case 2: System.out.println("Das ist gut!"); break;
5     case 3: System.out.println("Das ist befriedigend!"); break;
6     default: System.out.println("Das muss besser werden!"); break;
7 }

```

### 2.5.3 Die abweisende Schleife (while)

Syntax (EBNF):

```

"while" "(" bedingung ")"
  anweisung

anweisung := s.o.
bedingung := s.o.

```

ein kleines Codebeispiel:

```
1 int i=0;
2 while (i++ < 10)
3     System.out.println(i*i);
```

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

Syntax (EBNF):

```
"do" anweisung
"while" "(" bedingung ")" ";"

anweisung := s.o.
bedingung := s.o.
```

ein kleines Codebeispiel:

```
1 int i = 0;
2 do
3     System.out.println (i*i);
4 while (++i < 10);
```

#### 2.5.5 Die Zählschleife (for)

Syntax (EBNF):

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

bedingung := s.o.
anweisung := s.o.
```

die Zählschleife kann auch folgendermaßen dargestellt werden (keine EBNF!):

```
1 initialisierung;
2 while (bedingung) {
3     anweisung
4     ausdruck;
5 }
```

ein kleines Codebeispiel:

```
1 for (int i=0; i<10; i++) {
2     int quad = i*i;
3     System.out.println(i + " --> " + quad);
4 }
```

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

**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 ein 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.  $0 \dots 2\pi$ )! 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

$$r_{ij} = \sum_{n=0}^{Ba-1} a_{in}b_{nj} \quad (\text{oder kürzer: } r_{ij} = \sum_n a_{in}b_{nj})$$

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
```

Iterable ist ein Interface des Java Collections Framework (JCF) (siehe S.58)

ein kleines Codebeispiel:

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

## 2.5.7 Verlassen von Schleifen/Verzweigungen mit break und continue

**break** beendet die Ausführung der 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 der Marke `label`: gekennzeichneten Schleife bezieht, ansonsten wird bei verschachtelten Schleifen die innere verlassen.

Ein Beispiel:

```
1 int i1=1;
2 int zeilen=5;
3 eins: while (true) {
4     int i2=1;
5     zwei: while (true) {
6         System.out.print(" " + i2++);
7         if (i2 > i1) {
8             i1++;
9             System.out.println();
10            if (i1 > zeilen) break eins;
11            break;
12        }
13    }
14 }
```

**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 Programm bei Bedarf mittels `throw` geworfen:

```
throw new MyExceptionClass("Alles ist im Eimer!");
```

Eine Ausnahme tritt auf, wenn sie mittels `throw` geworfen wird oder aus einer Methode heraus weiter geworfen wird (... `throws` ... im Kopf der Methode).

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

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

```
try {
    anweisungsfolge0
        // die eigentlichen Anweisungen,
        // die ev. Exceptions vom Typ Ausnahmetyp1 werfen koennen
}
catch (Ausnahmetyp1 aname1) { // faengt Objekte von Ausnahmetyp1 und Kindklassen
    anweisungsfolge1
}
... // ggf. weitere catch Anweisungen

finally {
    anweisungsfolgeN // wird immer durchlaufen
}

anweisungsfolgeX := anweisungsfolge
```

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

```
void myMethod (...) throws Ausnahmetyp1 { ... }
```

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(s);
double d = Double.parseDouble(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

```
1 class Except2 {
2     // parst als int und druckt die zwei Argumente nach dem Programmnamen
3     static void main (String[] args) {
4         int i0,i1;
5         System.out.println("1.Kommandozeilen-Argument: " + args[0]);
6         i0 = Integer.parseInt(args[0]);
7         System.out.println("2.Kommandozeilen-Argument: " + args[1]);
8     }
9 }
```

```

8     i1 = Integer.parseInt(args[1]);
9     System.out.println("nach moeglichen Eingabefehlern");
10 }
11 }

```

Abfangen oder Nichtabfangen der Ausnahmen

```

1 class Except3 {
2     // druckt das 1. Argument nach dem Programmnamen
3     static void main (String[] args) {
4         int i0,i1;
5         try {
6             System.out.println("1.Kommandozeilen-Argument: " + args[0]);
7             i0 = Integer.parseInt(args[0]);
8             System.out.println("2.Kommandozeilen-Argument: " + args[1]);
9             i1 = Integer.parseInt(args[1]);
10            System.out.println("end try");
11        }
12        catch (NumberFormatException e){
13            System.out.println("NumberFormat: " + e);
14        }
15        // catch (ArrayIndexOutOfBoundsException e){
16        //     System.out.println("ArrayOut...: " + e);
17        // }
18        finally {
19            System.out.println("finally");
20            // wird IMMER abgearbeitet, falls vorhanden
21        }
22        System.out.println("ausserhalb try catch finally");
23    }
24 }

```

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, in dem Fall geht es nach dem try-catch NICHT weiter und die Methode wird abgebrochen.

### 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 ... }
```

Nicht abgefangene Ausnahmen werden in jedem Fall weitergeworfen!

Ein **Beispiel für das bedingte Werfen einer eigenen Exception-Klasse** aus einer Methode heraus. Die Methode `static int intDiv (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!

```

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

```

Exception	Behandlung	try { ... } catch { ... } (Fangen)	throws (Weiterwerfen)
Erzeugung			
Ruf einer Methode oder Feldzugriff		Zeile 15	Zeile 11, 12
throw			Zeile 6, 3

Ein **zusammenfassendes Beispiel**, welches **Objektorientierung und Ausnahmebehandlung** behandelt, soll in **Übung 17** auf S.32 entwickelt werden

## 2.6 Operatoren

Die folgende Übung ist für das Verständnis obligatorisch:

**Übung 12** *Arbeiten Sie sich in die Wirkungsweise von Operatoren ein, frischen Sie Ihre Kenntnisse aus anderen Programmiersprachen auf! Lesen Sie dazu im Handbuch der Javaprogrammierung ([5]) oder im Javatutorial den Abschnitt Operatoren, die gute Zusammenfassung dort sollten Sie sich ausdrucken (Java Tutorial [3, file://./opsummary.html])!*

## Vorrang (precedence P) und Assoziativität (A) der Operatoren (ähnlich C(++))

P		Operator	A
1	unary operators creation or cast	[] . (params) expr++ expr-- ++expr --expr +expr -expr ~ ! new (type)expr	R
2	multiplicative	* / %	L
3	additive	+ -	L
4	shift	<< >> >>>	L
5	relational	< > <= >= instanceof	L
6	equality	== !=	L
7	bitwise AND, log. AND	&	L
8	bitwise XOR, log. XOR	^	L
9	bitwise OR, log. OR		L
10	logical AND	&&	L
11	logical OR		L
12	conditional	?:	R
13	assignment	= += -= *= /= %= &= ^=  = <<= >>= >>>=	R

**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" : "1");
8             //System.out.print(((b &(m >>> i))== 0)? "0" : "1");
9         }
10    }
11 }

```

**Übung 14** Analysieren Sie das Beispiel! Was passiert? Rufen Sie sich Ihr Wissen über Zahlendarstellungen (wie Zweierkomplement) in Erinnerung und lesen Sie über Operatoren im Java-Tutorial!

Im Beispiel oben wird das Bitmuster auf die Standardausgabe gedruckt, was nicht sehr flexibel ist. Besser ist die Nutzung einer Funktion, die das Bitmuster als String liefert:



```

1 public class NumberTools {
2     public static void main (String[] args) {
3         int i = 7;
4         System.out.println("Eine int-Zahl: " + i);
5         System.out.println("Das Bitmuster: " + getIntBitsAsString(i));
6     }
7     public static String getIntBitsAsString(int zahl) {
8         // ein char-Feld als Puffer:
9         char [] bits = new char[32];
10        // die int-Zahl mit MSB = 1, sonst alle 0
11        int m = Integer.MIN_VALUE; //MSB ist 1
12        for (int i=0; i<32; i++){
13            // b wird bitweise abgetastet
14            bits[i] = ( zahl & m >>> i )== 0 ? '0' : '1' ;
15            //      (((zahl &(m >>> i))== 0)? '0' : '1');
16        }
17        return new String(bits);
18    }
19 }

```

Mit den bitweisen Operatoren (& |) können u.U. nur Daten des gleichen Datentyps verknüpft werden. Wenn wir float- oder double-Zahlen so wie oben scannen wollen, also praktischerweise auch wieder gegen eine int-Zahl, müssen wir das Bitmuster der Gleitkommazahl in eine ganze Zahl gleichen Bitmusters wandeln. 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

**Klasse (Definition):** In der objektorientierten Programmierung versteht man unter einer **Klasse** eine Bauvorschrift für ähnliche Objekte, *so kann es z.B. eine Klasse `Person` geben und zu dieser Klasse mehrere Objekte (z.B. `Peter`, `Lisa`, `Ingrid`).*

Eine Klasse definiert in ihrem Inneren alle Eigenschaften (Variablen, Daten) und Fähigkeiten (Methoden), über die die Objekte der Klasse verfügen.

Außerdem kann jede Klasse eine Methode

```
public static void main(String [] args)
```

enthalten, dann handelt es sich bei der Klasse um eine Java-Applikation.

**Objekt (Definition):** Ein **Objekt** oder eine **Instanz** ist ein konkretes **Exemplar einer Klasse**. Es verfügt über konkrete Werte für die durch die Klasse definierten Daten.

*Eine Klasse ist wie eine Puddingform, nach deren Muster die Objekte (Puddings) gebaut werden können. Im Unterschied zu Puddings können sich die Objekte aber stärker unterscheiden.*

Als Synonym für die Klasse wird der Begriff des **abstrakten Datentyps (ADT)** verwendet. **Abstrakte Datentypen (ADT)** bzw. Klassen stellen eine Verallgemeinerung oder Abstraktion der **elementaren und strukturierten Datentypen** (wie `int`, `double`, ... bzw. Felder, Strukturen, ...) dar.

Abstrakte Datentypen (ADT) bestehen aus

- Daten (Werte von Daten/Variablen),
- Methoden (algorithmisches Wissen, Funktionen)
- und der Regelung von Zugriffsrechten auf ihre Daten und Methoden (Sichtbarkeit).

Die Einbeziehung des algorithmischen Wissens in Form von Methoden in die Eigenschaften eines Datentyps macht einen wesentlichen Unterschied zwischen der objektorientierten und der prozeduralen Programmierung, wie sie z.B. in C abläuft, aus. Dieses algorithmische Wissen ist allerdings nicht bei dem konkreten Objekt gespeichert, das würde zu redundantem (mehrfach vorhandenem) Code führen, sondern an einer zentralen Stelle mit anderen Informationen zur Klasse.

**Vererbung:** Das Konzept der **Vererbung** beschreibt, wie Klassen in einer hierarchischen Struktur (in Java ist es ein Baum) miteinander in Beziehung stehen. Dabei erbt eine **Unterklasse/ abgeleitete Klasse** von einer **Basis-/ Ober-/ Elternklasse** alle Eigenschaften (Daten und Methoden). In Java kann jede Klasse **nur eine und genau eine Elternklasse** besitzen, mit Ausnahme der Klasse `Object`, die die Wurzel der Klassenhierarchie bildet und keine Oberklasse hat. Die Oberklasse bildet eine **Abstraktion** der Unterklasse, während die Unterklasse eine **Spezialisierung** der Oberklasse bildet.

Die Tatsache, daß eine Klasse nur eine Elternklasse haben kann, wird als **Einfachvererbung** bezeichnet, sind mehrere Oberklassen möglich, spricht man von **Mehrfachvererbung** (z.B. in C++).

*Stellen wir uns die Klasse `Person` vor, die vielleicht Daten wie `Name` und `Adresse` beinhaltet, könnten nun Klassen wie `Kunde` und `Mitarbeiter` davon abgeleitet werden und zusätzliche Daten und Methoden beinhalten.*

**Abstrakte Klasse (Definition):** Eine abstrakte Klasse ist eine Klasse, von der keine Objekte instanziiert werden können, die aber als Basisklasse bei der Vererbung in einer Klassenhierarchie dienen kann.

**Interface/ Schnittstelle (Definition):** Ein Interface ist eine Spezialform einer abstrakten Klasse, die nur Methodendeklarationen (d.h. ohne Implementierung/Methodenkörper) und finale Variablen (Konstanten) enthält.

Seit Java 1.8 ist eine **default**-Implementierung von Methoden im Interface möglich, wodurch jetzt prinzipiell Implementierungskonflikte (durch Mehrfachvererbung von Funktionalität) möglich sind. (siehe 3.4, S. 35)

Eine Klasse kann zusätzlich zu seiner Elternklasse, die sie beerbt, beliebig viele Interfaces implementieren. Da es im Java nur **Einfachvererbung** gibt, ist die Implementierung von Interfaces eine Möglichkeit, dem Objekt einer Klasse neben dem Datentyp dieser Klasse auch die Datentypen der implementierten Interfaces zu geben.

**Nachricht, Message (Definition):** In objektorientierten Systemen gilt folgende Sprechweise: Wenn ein Objekt A eine Methode von einem Objekt B aufruft, so heißt es, daß A eine Nachricht (Message) an B sendet.

## 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 30.

## 3.3 Klassen in Java

Klassendefinition (EBNF):

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

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

Erzeugen von Exemplaren (Instanzen) (kein EBNF):

```
// Deklaration:
klassenname instanzname;

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

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

### Ein einfaches Beispiel einer Klassenhierarchie

Die Oberklasse 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 eigenständige 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 zur Instanziierung 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, explizit definiert werden muß, wenn er benötigt wird.

Die Instanziierung / 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?

**Die Unterklassen der 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 (**gehalt**)

```

1 class Mitarbeiter extends Person {
2     public double gehalt = 0;
3
4     public Mitarbeiter (String n) {
5         super (n);
6     }
7     public Mitarbeiter (String n, double g) {
8         super (n);
9         this.gehalt = g;
10    }
11    public void drucke (){
12        super.drucke();
13        System.out.println("Gehalt: " + gehalt);
14    }
15 }

```

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 Eigenschaften **Bereich** und **bonus**

```

1 class Chef extends Mitarbeiter {
2     public String Bereich = "N.N.";
3     public bonus = 0.0; // ein Chef bekommt ggf. eine Belohnung
4     public Chef(String n, double g, String b) {
5         super (n, g);
6         this.Bereich = b;
7     }

```

```

8 public Chef(String n, double g) {
9     super (n, g);
10 }
11
12 public void drucke() {
13     super.drucke();
14     System.out.println("Ich bin Chef von " + Bereich);
15 }
16 public void setBonus(double bonus) {
17     this.bonus = bonus;
18 }
19
20 }

```

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

```

1 class Kunde extends Person {
2     public String KdNummer = "--";
3
4     public Kunde (String n) {
5         super (n);
6     }
7     public Kunde (String n, String KdNr) {
8         super (n);
9         this.KdNummer = KdNr;
10    }
11    public void drucke () {
12        super.drucke();
13        System.out.println("Kundennummer: " + KdNummer);
14    }
15 }

```

Referenzen auf Objekte können in Speicherzellen der entsprechenden Klasse und beliebiger Oberklassen abgelegt werden

...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();

```

Der erfolgreiche 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.

Da aber die Methode `drucke()` bereits in der Klasse `Person` bekannt ist, die auch für die Typisierung der Arrays in Zeile 7 und 23 steht, kann der Ruf wie in Zeile 17 bzw. 27 erfolgen.

Im Gegensatz dazu misslingt der Ruf `p[0].setBonus(...)` in Zeile 29, weil die Referenz `p[0]` für das System als `Person` typisiert ist (obwohl sie natürlich auf ein `Chef`-Objekt zeigt), `Person` aber die Methode `setBonus(...)` nicht kennt. Eine **Typecast** auf `Chef` in Zeile 30 macht den Ruf möglich:

```

28     ...
29     p[0].setBonus(2000.);           // bringt Compilerfehler
30     ( (Chef)p[0] ).setBonus(2000.); // ok
31     ...
32 }
33 }

```

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

Der Tabelle zur Sichtbarkeit (S.27) kann man entnehmen, welche Informationen/ Bestandteile (d.h. Variablen und Methoden) einer Klasse für wen sichtbar sind. Die folgenden Ausführungen sollen helfen, den passenden Sichtbarkeitsmodifizier 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 Programmiers 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** Sichtbarkeitsmodifizier). 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 Information für jedermann lesbar, aber nicht änderbar (oder umgekehrt) sein soll, dann kann man die entsprechende setter- (bzw. getter-)Methode nicht implementieren oder aber `private` machen,
- oder aber, wenn die interne Implementation der eigentlichen Information von den außen sichtbaren Typen der getter/setter-Methoden abweicht, was das folgende Beispiel verdeutlichen soll:

```
1 ...
2 private String name; // z.B. name = "Lorenz,Peter"
3 public String getVorname() {
4     // gibt den Teil nach dem Komma zurueck, also "Peter"
5     String [] n = name.split(",");
6     if (n.length == 2) return n[1];
7     else return "";
8 }
9 public void setVorname(String vorname) {
10    // Aendert den Teil nach dem Komma
11    // keine Kommas eingeben!!!
12    this.name = getNachname() + "," + vorname;
13 }
14 public String getNachname() {
15    // gibt den Teil vor dem Komma zurueck, also "Lorenz"
16    String [] n = name.split(",");
17    if (n.length == 2) return n[0];
18    else return "";
19 }
20 public void setNachname(String nachname) {
21    // Aendert den Teil vor dem Komma
22    // keine Kommas eingeben!!!
23    this.name = nachname + "," + getVorname();
24 }
25 ...
```

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.  $0 \dots 2\pi$ ), die Wertepaare sollen ausgedruckt werden (als Modifikation der Aufgabe 9, S.19)!

### 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.

### Übung 18 Modifizieren Sie die Methode

`public static double [][]multiplyMatrices(double [][] a, double [][] b) {...}`

aus Übung 10 (S.19) so, daß getestet wird, ob es sich bei `a` und `b` um Matrizen (rechteckige Datenstrukturen) handelt und die Breite von `a` gleich der Höhe von `b` ist. Wenn das nicht erfüllt ist, soll eine `Exception` geworfen werden, die über ihre Ursache informiert.



### 3.3.1 Das Paket Reflection

Mit dem Paket `Reflection` kann man

- den Typ von Objekten ermitteln (was zu einem Klassenbeschreibungsobjekt der Klasse `Class` führt),
- oder dieses Klassenbeschreibungsobjekt zu einer beliebigen Klasse laden,
- für eine Klasse die existierenden Attribute/Variablen (Fields), die Methoden, die Konstruktoren, die Superklasse und implementierten Interfaces und zu allen die gültigen Modifier bestimmen,
- Objekte, deren Typen zur Laufzeit bekannt sind, dynamisch instanziiieren,
- und dynamisch Attributwerte beeinflussen und Methoden rufen.

Das alles kann geschehen, ohne daß der Klassenname auch nur einmal im geschriebenen Quelltext auftritt, natürlich müssen die Klassen im `Classpath` mit ihrem Bytecode vorliegen. Man kann also in einem Stück Java-Quelltext Klassen verwenden, die zum Zeitpunkt der Erstellung dieses Quelltextes noch nicht existieren. So werden zum Beispiel Plugins verwendet, die z.B. nach und nach zu einem Browser entstehen, ohne daß sich dafür der Browser ändern muß..

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`

```
(import java.lang.reflect.*;)
```

Wesentlich für die Angabe von Typen, insbesondere für Parameterlisten von Konstruktoren und Methoden, ist die Klasse `Class` (eine Art Laufzeit-Typbeschreibung). Entsprechende `Class`-Objekte können folgendermaßen geschaffen (bzw. referenziert) 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`

Für jede durch den `ClassLoader` in die JVM geladene Klasse gibt es eine Instanz der Klasse `Class`, jede Typbestimmung auf einem Objekt gibt eine Referenz auf diese Instanz der Klasse `Class` zurück!

Erzeugung von Instanzen/Objekten beliebiger Klassen auf dynamische Art:

Zunächst wird eine Klasse (aus der oben vorgestellten Klassenhierarchie in Abschnitt 3.3 auf S.27) mit der `Reflection-API` nach Konstruktoren und deren Parametern befragt:

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

```

9      Class cp [] = cf[i].getParameterTypes();
10     for (int j=0; j<cp.length; j++) {
11         System.out.println(" Parameter " + j + " : " + cp[j]);
12     }
13 }
14 ...
15 }

```

ein Objekt wird mit dem Default-Konstruktor instanziiert

```

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

```

und letztendlich wird ein Objekt mit beliebigem Konstruktor, der durch cc referenziert und über seine Parameter identifiziert wird (Zeile 26), instanziiert:

```

20 //*****
21 Class ConstrParChef[] = {
22     String.class,
23     double.class, // oder Double.TYPE
24     String.class
25 };
26 Constructor cc = m.getDeclaredConstructor (ConstrParChef);
27 Object [] po = {"derNeueChef", new Double( 12345.), "viele Abt."};
28 // oder auch ueber Kommandozeilenargumente
29 // Object [] po = {a[0], new Double(a[1]), a[2]};
30 Person p = (Person) cc.newInstance(po);
31 p.drucke();
32 }
33 }

```

Dieses Muster der Instanziierung eines Objektes einer zur Entwicklungszeit unbekannten Klasse wird beim Design eines variablen Serversystems (S. 74, Abschnitt 4.6.3) angewendet.

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

**Übung 19** 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),  
(seit Java 1.8 ist es möglich, im Interface eine **default**-Implementierung festzulegen)

enthalten. Durch die **default**-Implementierung gibt es nun den ursprünglich vermiedenen Konflikt durch möglicherweise ererbte unterschiedliche Implementierungen

(<http://www.angelikalanger.com/Articles/EffectiveJava/72.Java8.DefaultMethods/72.Java8.DefaultMethods.html> ).

Die Implementierung aus einer Elternklasse hat dabei den Vorrang vor einer **default**-Implementierung eines Interfaces, bei mehreren gleichrangigen Interfaces muss sich der Programmierer entscheiden.

Interfaces implementierende Klassen müssen alle abstrakten Methoden implementieren → deshalb dürfen Interfaces **nicht wachsen**. (Das Hinzufügen einer neuen abstrakten 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. Damit werden Interfaces im Java verwendet, um Objekten unterschiedliche Typen zuweisen zu können.

Im folgenden Beispiel soll gezeigt werden, wie mit einem Interface `XPunkt` unterschiedlichen Klassen, die jeweils über ein Koordinatenpaar eines zweidimensionalen Raumes verfügen, ein gemeinsamer Datentyp zugewiesen werden kann, der dann an anderer Stelle weiter verwendet werden kann:

```
class Punkt implements XPunkt{
    double x,y;
    double getX(){ return x;}
    double getY(){ return y;}
    ...
}
class Complex implements XPunkt{
    double real,imag;
    double getX(){ return real;}
    double getY(){ return imag;}
    ...
}
class GPSPunkt implements XPunkt{
    double longitude, latitude, elevation;
    double getX(){ return longitude;}
    double getY(){ return latitude;}
    ...
}
```

Bis hier könnte man auch eine abstrakte Oberklasse als gemeinsamem Datentyp verwenden, was beim folgenden Beispiel, welches als Datentyp für einen Boten mit geographischen Koordinaten dient, wegen der Einfachvererbung im Java nicht mehr möglich ist:

```
class Bote extends Mitarbeiter
    implements XPunkt{
    double longitude, latitude;
    /** zur Maps-Darstellung */
    double getX(){ return longitude;}
    double getY(){ return latitude;}
    ...
}
```

Das verwendete Interface legt die Methoden fest, die von den implementierenden Klassen implementiert werden müssen:

```
interface XPunkt{
    double getX();
    double getY();
}
```

```
class Main{
    ... main (...) {
        Punkt [] funktion = ... ;
        Complex [] wurzel = ... ;
        GPSPunkt [] track = ... ;
        Bote [] schwarm = ... ;

        XYPaint zeichner = new XYPaint();

        zeichner.draw(funktion);
        zeichner.draw(wurzel);
        zeichner.draw(track);
        zeichner.draw(schwarm);
    }
}
```

Sollen die Felder in einer Klasse `XYPaint` mit einer Methode `draw(...)` graphisch dargestellt werden, so müsste es für jeden Feldtyp eine überladene `draw`-Version geben ...

```
class XYPaint{
    void draw(Punkt [] func){ ... }
    void draw(Complex [] feld){ ... }
    void draw(GPSPunkt [] track){ ... }
    void draw(Bote [] alle){ ... }
}
```

..., was bei den vielen denkbaren Objekttypen, die gezeichnet werden könnten, nicht sinnvoll realisierbar ist.

Über den Datentyp eines Interfaces, welches von jeder Klasse zusätzlich implementiert werden muss, kann man jedoch eine Abstraktion realisieren und kommt mit einer `draw(...)`-Methode aus:

```
void draw(XYPunkt [] punkte){ ... }
```

Weitere Beispiele zu Interfaces zu finden im bald folgenden Abschnitt Eventhandling 4.2 (S.43).

### 3.5 Packages und Archiv-Tool jar

Ein Package ist ein Ort für eine Menge fachlich zusammengehörender Klassen bzw. ebendiese Klassen. In der Datenstruktur entsprechen die Packages hierarchisch aufgebauten Unterverzeichnissen.

Die Package-Anweisung (siehe S. 8)

```
package pack_name;
```

ordnet die Klasse mit dieser Anweisung einem Package zu. Dabei kann `pack_name` mehrere durch `.` getrennte Teile haben, die den hierarchischen Unterverzeichnissen entsprechen.

Die Übereinstimmung der Unterverzeichnisnamen und Packagenamen wird im folgenden **kleinen Beispiel** deutlich, Klassen ohne Package-Anweisung liegen im **Default-Package**, das der Wurzel der Verzeichnishierarchie entspricht:

aktuelles Verzeichnis (das "default package")

```
MyZet.java (keine package-Anweisung)
MyZet.class
mainpack (Verz.)
  MyAaa.java (package mainpack;)
  MyAaa.class
  subpack (Verz.)
    MyBee.java (package mainpack.subpack;)
    MyBee.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
-O 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
```

ausgef"uhrt 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.a.class` bewirkt, daß das Archiv oder besser gesagt die Haupt-Klasse `a.class` 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 z.B. auf einen Sun/Oracle-Server `http://docs.oracle.com/javase/7/docs/api/`

... oder auch auf ein File auf dem lokalen System `file:///C:/jdk7/docs/api`

zeigen kann, 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

Diese werden anhand des AWT (Abstract Windowing Toolkit) erläutert, daneben gibt es noch Swing mit seinen Klassen, ein kurzes Beispiel findet sich in Abschnitt 4.1.6 (S.42).

#### 4.1.1 Applets

```
1 import java.applet.Applet;
2 import java.awt.*;
3 import java.awt.event.*;
4
5 public class MyApplet extends Applet {
6
7     public void init() {
8         // Initialisierungsaufgaben, die einmal zu Beginn zu erfolgen haben,
9         // werden hier gemacht (da Applet keinen Konstruktor hat)
10    }
11
12    // paint wird von der VM (und nicht vom Programmierer!) gerufen,
13    // das uebergebene Graphics-Objekt ist sozusagen das Blatt Papier,
14    // auf dem paint malen darf
15    public void paint(Graphics g) {
16        // Graphics (und somit g) verfuegt ueber viele Zeichnen-Methoden
17        g.drawString("Hallo, Welt! " + getParameter("derPara"),
18                    getSize().width/2-40, getSize().height/2-5);
19    }
20 }
```

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

```
1 <HTML>
2 <body>
3 <applet code="MyApplet.class" derPara=" und nun?"
4       width="300" height="300">
5 </applet>
6 </body>
7 </HTML>
```

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

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

#### 4.1.2 Frames

```
1 import java.awt.*;
2 import java.awt.event.*;
3
4 public class EmptyFrame extends Frame {
5
6
7     public EmptyFrame () {
8         // Der Inhalt der Titelleiste des Fensters
9         super("Ein leeres Fenster");
10        // Groesse und Position
11        setSize(500,300);
12        setLocation(100,100);
13        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 klicken oder Alt-F4 drücken, erfahren wir im Abschnitt Eventhandling 4.2.3 (S.45)!

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

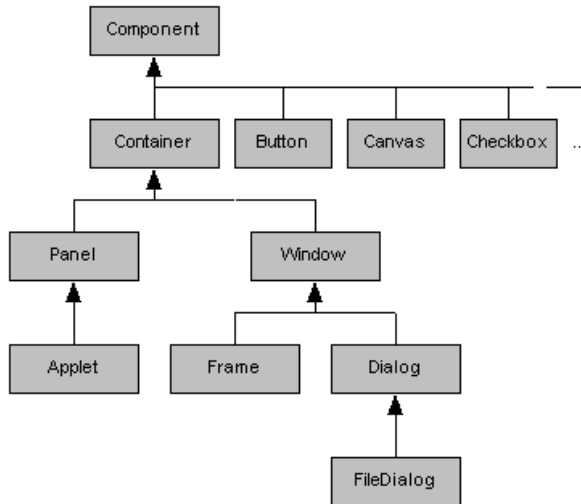


Abb. Hierarchie der Fensterklassen (aus [5, Kap.27, Fenster])

Dabei ist jeder **Container** geeignet, etwas anderes (eine **Component**) zu enthalten, verfügt also über `add(Component)`-Methoden, und ist natürlich selbst eine **Component**, somit kann man eine Hierarchie von Containern aufbauen,

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.

Ein einfaches Beispiel zur Schachtelung von Komponenten finden Sie im Abschnitt 4.1.5 (siehe S.41).

#### 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 [5])

```

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.

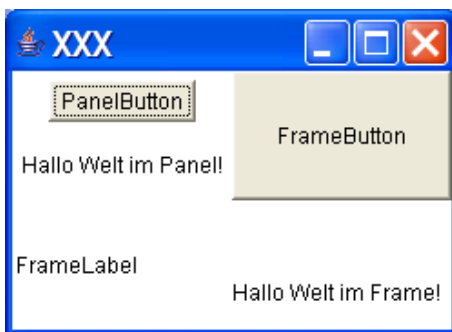
Sollen die Größe und Positionen einer Komponente direkt festgelegt, wird das **Null-Layout** verwendet:

```
1  setLayout(null);
2  Button but = new Button("Weiter");
3  add(but);
4  but.setBounds(x,y,width,height); //in java.awt.Component
5  ...
```

**Übung 20** 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 Komponenten 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 Komponente, 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`.

```
1  public class HelloFrame extends Frame {
2
3      HelloPanel hp;
4
5      // im Konstruktor wird der Frame zusammengebaut
6      public HelloFrame () {
7          super("XXX");
8          // ein Layoutmanager mit 2 Zeilen und 2 Spalten erhaelt die Aufgabe
9          setLayout(new GridLayout(2,2)); // der Objektanordnung
10
11         // jetzt werden nach und nach mit add(...) die Zellen des 2x2-Gitters befuellt
12         hp = new HelloPanel(); // benanntes Objekt des Typs HelloPanel
13         add(hp); // Zelle 1,1
14
15         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 Komponenten (wie `Button`, `TextField`, `TextArea`, `Label`, `Panel`, `Canvas`, ...) füllen. Die `paint`-Methode ihrerseits wird auf **Komponenten, die keine Container** sind (wie z.B. `Canvas`) genutzt, wenn man rein graphische, nicht interaktive Inhalte wie z.B. einen Kurververlauf darstellen will.

#### 4.1.6 Ein wenig Swing

Das Swing-Hauptfenster `JFrame` besteht aus

- einer transparenten, die gesamte Fläche füllende `GlassPane` (von `Container` abgeleitet), die normalerweise keine darzustellenden Elemente enthält, und
- der parallel dazu liegenden (also ebenfalls die gesamte Fläche füllende) `LayeredPane` (von `JLayeredPane` abgeleitet) mit den Teilen
  - Menüleiste (von `JMenuBar` abgeleitet) und
  - `ContentPane` (von `Container` abgeleitet), in dem die GUI-Elemente des Fensters angeordnet werden:

```

1 import javax.swing.*;
2
3 public class MyJFrame extends JFrame
4 {
5
6     // ein String als Parameter fuer den Titel:
7     public MyJFrame(String t) {super(t);}
8     // und dann muss auch der Standardkonstruktor neu definiert werden!

```

```

9  public MyJFrame() {}
10
11  public static void main( String[] args )
12  {
13      JFrame f = new MyJFrame( "Das Hallo-Welt-Fenster" );
14      f.setDefaultCloseOperation( EXIT_ON_CLOSE );
15      // f.setDefaultCloseOperation( DISPOSE_ON_CLOSE );
16      f.setSize( 300, 200 );
17
18      f.getContentPane().add(new JLabel("Alles Quatsch!"));
19      // oder auch nur: f.add(new JLabel("Alles Quatsch!"));
20      // der Bezug auf die ContentPane erfolgt implizit
21      f.setVisible( true );
22  }
23  }

```

Mit dem Festlegen der `DefaultCloseOperation` wird die Reaktion des implizit vorhandenen `WindowListener` festgelegt! Ein extra zu programmierendes Eventhandling wie bei AWT ist hier nicht nötig.

## 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 `EventObject` 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 AWT-Frames, welches ein `MouseListener`-Interface implementiert und damit auch die Methoden des `MouseListener` implementieren muß (nach einem Applet aus dem Java-Tutorial [3]):

```

1  import java.awt.*;
2  import java.awt.event.*;
3

```

```

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

```

... und der Spot

```

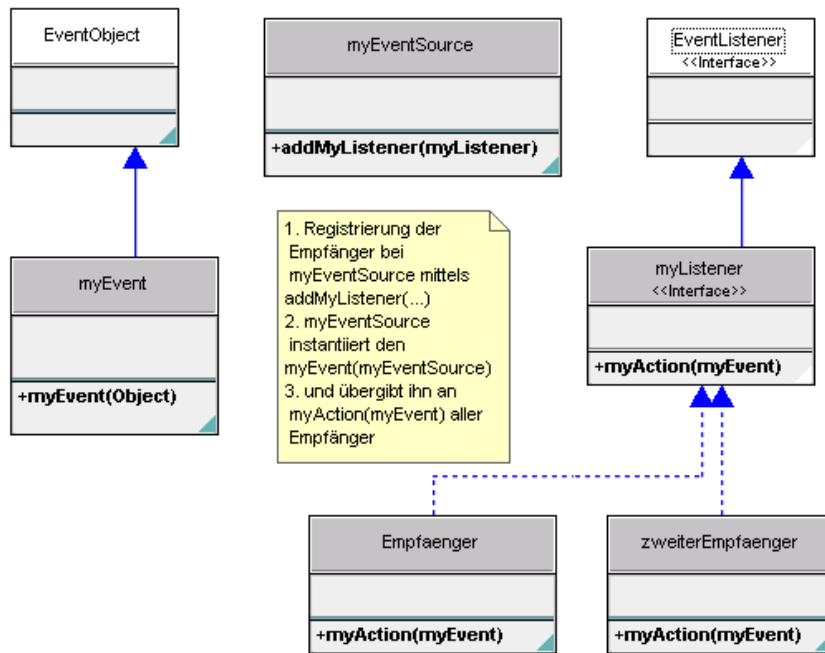
1 public class Spot {
2     public int size;
3     public int x, y;
4
5     public Spot(int intSize) {
6         size = intSize;
7         x = -1;
8         y = -1;
9     }
10 }

```

**Übung 21** Bauen Sie die Anwendung nach folgenden Vorgaben um:

- Modifizierung des Spots, die Farbe soll im Spot gehalten werden, benutzen der entsprechenden 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)`

Das Java AWT Delegation Model mit den beteiligten Klassen / Schnittstellen



#### 4.2.3 Beauftragung einer inneren Klasse

**xxxAdapter** sind Implementationen von **xxxListnern** mit leeren Methodenkörpern, durch ihre Nutzung erspart man sich Schreibaarbeit, 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!

```

1  import java.awt.*;
2  import java.awt.event.*;
3
4  public class EmptyFrameWithInnerAnonymClass extends Frame {
5
6      public EmptyFrameWithInnerAnonymClass () {
7          super("Ein leeres Fenster");
8
9          addWindowListener(
10             new WindowAdapter() {
11                 public void windowClosing(WindowEvent event) {
12                     System.out.println("WindowClosing...");
13                     setVisible(false);
14                     dispose();
15                 }
16                 public void windowClosed(WindowEvent event)
17                 {
18                     System.out.println("terminating program...");
19                     System.exit(0);
20                 }
21             }
22         );
23     }
24 }

```

```

23     setSize(500,300);
24     setLocation(100,100);
25     setVisible(true);
26 }
27
28
29 public static void main (String [] a) {
30     EmptyFrameWithInnerAnonymClass cm = new EmptyFrameWithInnerAnonymClass();
31 }
32
33 }

```

Von Zeile 10 bis 21 (das ist innerhalb der Parameterklammern von `addWindowListener`) wird ein Objekt inszenziert, das zu einer Klasse gehört, die genau an dieser Stelle von der Klasse `WindowAdapter` abgeleitet wird. Diese Klasse erhält keinen expliziten Namen, da sie aber die erste innere anonyme Klasse innerhalb der Klasse `EmptyFrameWithInnerAnonymClass` ist, lautet ihr Name `EmptyFrameWithInnerAnonymClass$1` und ihr Bytecode liegt in der Datei `EmptyFrameWithInnerAnonymClass$1.class`.

Dateien nach dem Kompilieren:

```

EmptyFrameWithInnerAnonymClass.java
EmptyFrameWithInnerAnonymClass.class
EmptyFrameWithInnerAnonymClass$1.class

```

### innere benannte Klasse

```

1  import java.awt.*;
2  import java.awt.event.*;
3
4  public class EmptyFrameWithInnerClass extends Frame {
5
6
7      public EmptyFrameWithInnerClass () {
8          super("Ein leeres Fenster");
9
10         addWindowListener( new MyWindowCloser() );
11
12         setSize(500,300);
13         setLocation(100,100);
14         show();
15     }
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 22** *Modifizieren Sie die Klasse aus Übung 20 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-Menü 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 Aufklappmenüs (**Menu**) zugeordnet, diese wiederum erhalten mittels **add()** Menuelemente (**MenuItem**) zugeordnet.

Man beachte, daß es hier zwei **ActionListener** gibt, einer ist der Frame selbst, der andere ein anonym, der auf der Menuelemente **mbItemExit** angemeldet wird (immer zu finden über die **add...Listener()**-Methoden:

```
28 // das AWT - Menü ~~~~~
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     // Möglichkeit 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)

```

... 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 23** Bauen Sie zu Aufgabe 16 (S.31) 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 }

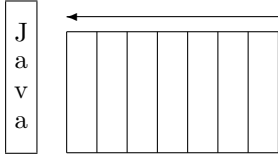
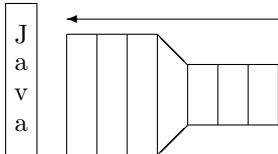
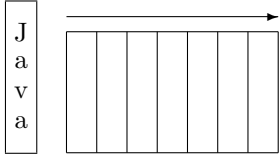
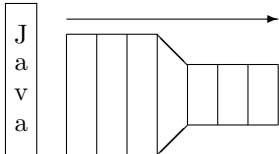
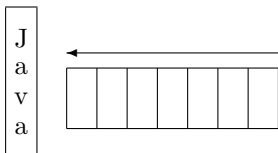
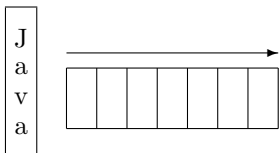
```

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

## Übung 24 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?

### Prinzipielle Einteilung der Streamklassen:

	JVM ←	JVM →
<b>Character Streams</b>  1 char : (2 byte)	<b>Reader:</b>    <b>InputStreamReader:</b>  	<b>Writer:</b>    <b>OutputStreamWriter:</b>  
<b>Byte Streams</b>  1 byte:	<b>InputStream:</b>  	<b>OutputStream:</b>  

### Klassenhierarchie der Streamklassen

#### Character Streams

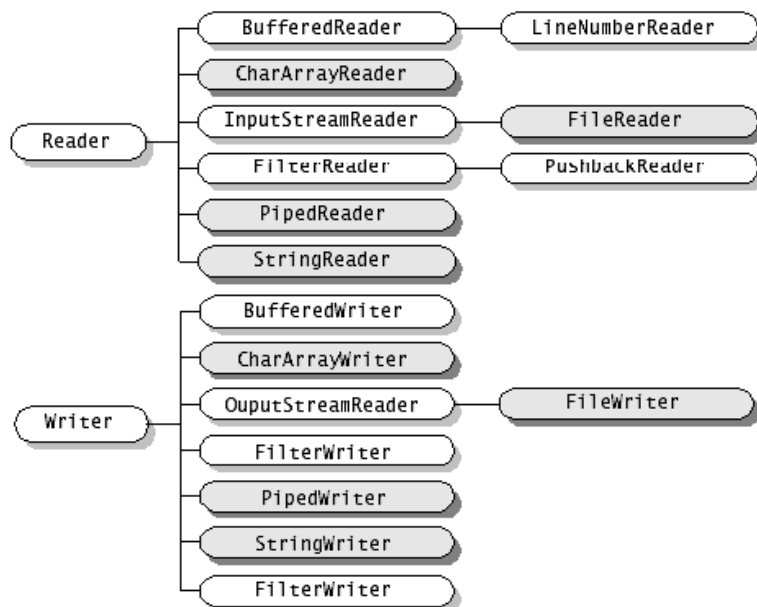


Abb. aus dem Java-Tutorial (java.sun.com)

## Byte Streams

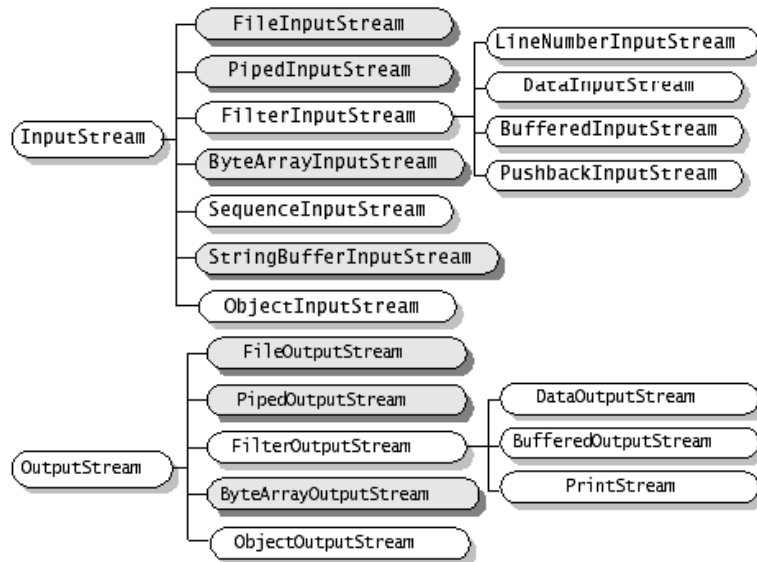


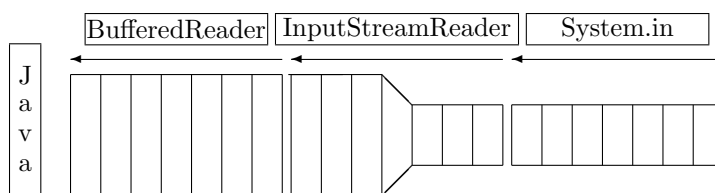
Abb. aus dem Java-Tutorial ([java.sun.com](http://java.sun.com))

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 }
  
```

Zeile 5 bewirkt das Aneinanderketten der Datenströme nach folgendem Muster:



...Seit dem JDK 1.5 liefert die Klasse **Scanner** eine bequeme Alternative:

```

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 in diesem Beispiel 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** oder aber global und nicht auf dem Stack 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) {
55         s = i.getLineNumber() + ":" + s;
56         System.out.println(s);
57         o.write(s,0,s.length());
58         o.newLine();
59         g += s + "\n";
60     }
61     i.close();
62     o.close();
63     return g;
64     //Experimentieren ohne close, und mit flush
65 }
66 }

```

### 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:

```

1 FileOutputStream fs = new FileOutputStream("test.serial");
2 ObjectOutputStream os = new ObjectOutputStream(fs);
3 os.writeObject("Hallo");
4 os.writeObject(new Time(10, 30));
5 os.writeObject(new Time(11, 25));
6 os.close();
7
8 FileInputStream fs = new FileInputStream("test2.serial");
9 ObjectInputStream is = new ObjectInputStream(fs);
10 Time time = (Time)is.readObject();
11 System.out.println(time.toString());
12 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:

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import java.io.*;
4
5 public class DefaultFrameSerial extends Frame implements Serializable {
6
7     public DefaultFrameSerial () {
8         super("Kopiertest");
9         addWindowListener(
10             // als WindowListener arbeitet ein Objekt der Klasse MyWindowSerializer
11             new MyWindowSerializer()
12         );
13     }
14
15     public void serialisiere() {
16         try {
17             System.out.println("Serialisierungsmethode");
18             FileOutputStream fos = new FileOutputStream("dfs.ser");
19             ObjectOutputStream os = new ObjectOutputStream(fos);
20             os.writeObject(this);
21             os.close();
22         } catch (Exception e) {}
23     }
24
25     public static void main (String [] a) {
26         DefaultFrameSerial dfs;
27         FileInputStream fis;
28         ObjectInputStream ois;
29         try {
30             fis = new FileInputStream("dfs.ser");
31             ois = new ObjectInputStream(fis);
32             dfs = (DefaultFrameSerial) ois.readObject();
33             dfs.setVisible(true);
34         } catch (FileNotFoundException e) {
35             System.out.println(e);
36             dfs = new DefaultFrameSerial();
37             dfs.setSize(300,100);
38             dfs.setLocation(100,100);
39             dfs.setVisible(true);
40         } catch (Exception e) {}
41     }
42 }
```

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

```
43 class MyWindowSerializer extends WindowAdapter implements Serializable {
44     // Serializable bewirkt, daß die Objekte der inneren Klasse mit serialisiert werden,
45     // sonst fehlt der MyWindowSerializer beim 2. Lauf!!
46     public void windowClosing(WindowEvent event) {
47         System.out.println("Begin windowClosing");
48         serialisiere();
49         setVisible(false);
50         dispose();
51         System.out.println("Ende windowClosing");
52     }
53     public void windowClosed(WindowEvent event) {
```

```
54     System.out.println("terminating program...");
55     System.exit(0);
56 }
57 }
58 }
```



## 4.4 Containerklassen und Stringhandling

### 4.4.1 Containerklassen - Java Collections Framework

**Motivation:** Wir suchen variable Container, um nicht mehr mit unflexiblen Feldern, deren Länge nicht geändert werden kann, arbeiten zu müssen.

```
static int MAXANZAHL = 100;
Mitarbeiter[] mitglieder = new Mitarbeiter[MAXANZAHL];
```

Mit den sogenannten **Generics** werden typisierte Container eingeführt, hier ein **Vector**, der auf Objekte der Klasse/ des Typs E typisiert ist und entsprechende Hinzufüge- und Lese-Funktionen hat:

```
java.util.Vector<E>
    public boolean add(E o)
    public E get(int index)
```

**Vector** ist ein Container, der null bis beliebig viele Elemente enthalten kann, diese Elemente sind, wie bei jeder Objektvariablen in Java, Referenzen (hier auf Objekte der Klasse E). Ein Casten beim Auslesen von Inhalten nicht mehr nötig, der Typ wird beim Hinzufügen (mit `add(...)`) registriert und beim Auslesen dem referenzierten Objekt zugewiesen:

```
1 // veraltete Nutzung ohne Typisierung
2 Vector v = new Vector();
3 E o = new E();
4 v.add(o); // bringt unchecked-Warnungen
5           // gemischtes Speichern verschiedener Typen moeglich
6 E n = v.get(...);
7
8 // und mit checked option fuer die Typsicherheit:
9 Vector<E> v = new Vector<E>();
10 ... // keine Warnungen
11     // nur ein Typ E im Vector
12     // wobei der festgelegte Typ E eben auch Object sein kann,
13     // dann passen ALLE Klassen in den Vector!
```

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

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:

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

## Java Collections Framework (JCF)

einige grundlegende Interfaces in ihren Hierarchien

```
Iterable ... hat Elemente und einen Iterator zur Rückgabe dieser
Collection ... eine Sammlung von Elementen
    Set ... ohne Doppel und Reihenfolge (Menge)
    List ... geordnete Liste

Map ... Menge von Schlüssel-/Werte-Paaren
```

...und mit ihren wesentlichen Methoden

```
Iterable<E>
    Iterator<E> iterator()

Collection<E> ... eine Sammlung
    boolean add(E o)
    boolean contains(Object o)
    boolean remove(Object o)
    int size()
    ...

Iterator<E>
    boolean hasNext()
    E next()
    void remove()
```

Etwas, was `Iterable` implementiert, kann anstatt eines Feldes in einer enhanced for-Schleife genutzt werden:

```
Syntax: for (Typ elem: Typ[] oder Iterable<Typ>) ...

String [] sf = { ... };
for (String s: sf) ...println(s);

Iterable<String> v = new ArrayList<>(); v.add(...); ...;
for (String s: v) ...println(s);
```

Neben dem `Iterator` wird auch noch das ähnliche `Enumeration`-Interface (z.B. als return-Typ von `Vector.elements()` oder `Hashtable.keys()` ) genutzt:

```
Enumeration<E>
    boolean hasMoreElements()
    E nextElement()
```

Interfaces und implementierende Klassen:

```
List<E> -> Vector<E>, ArrayList<E>
    ... eine geordnete Liste
    boolean add(E o)
    E get(int index)
    int size()

Map<K,V> -> HashMap<K,V>, Hashtable<K,V>
    ... Schlüssel/Werte-Paare mit eindeut. Schlüssel
    V put(K key, V value)
    V get(Object key)
    Set<K> keySet()
    // Hashtable hat zusätzlich: Enumeration<K> keys()
    Collection<V> values()
```

Utilities:

```
java.util.Collections
    static void sort(List<T>) // hier muss T Comparable implementieren,
                             // d.h., eine natuerliche Ordnung haben
    static void sort(List<T> list, Comparator<? super T> c)

Comparator<T>
    int compare(T o1, T o2)
    // Compares its two arguments for order.
    boolean equals(Object obj)
Comparable<T>
    int compareTo(T o)
    // Compares this object with the specified object o for order.
```

**Übung 25** 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 oder aus einem File) in die Anwendung gelangen, dort können sie in einem Container (Interface `List`: Klasse `Vector` oder `ArrayList`) 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` oder `HashMap`, die jeweils das `Map<K,V>`-Interface implementieren), in deren Ergebnis die Worte mit der entsprechenden Anzahl ausgegeben werden.

**Übung 26** Halten Sie die Menge der Punkte ( $x/y$ -Wertepaare in der Klasse `Punkt`) aus Aufgabe 16 (S.31) bzw. Aufgabe 23 (S.49) nicht mehr in einem Array, sondern in einem von `Vector` abgeleiteten 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 27** Aufbauend auf Aufgabe 26 (S.59) 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` ist eine Klasse zur Verwaltung von Zeichenketten. Der `String` beinhaltet ein finales (also unveränderliches) `char`-Array mit der entsprechenden Zeichenkette (ist also immutable (engl.)). `String` verfügt über eine `split(...)`-Methode, die, in Erweiterung des `StringTokenizer`s, als Begrenzer reguläre Ausdrücke (Abschnitt 4.4.3, S.60) verarbeiten kann. Soll eine Zeichenkette verändert werden, ohne dass ein neues Objekt instantiiert werden muss, sollten Klassen wie `StringBuilder` oder `StringBuffer` (beide mutable) verwenden!

**Übung 28** Nutzen Sie eine der genannten Möglichkeiten (mittels `String` oder `StringTokenizer`), um einen Text durch Auftrennung in seine Worte für Übung 25 aufzubereiten!

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

#### 4.4.3 Reguläre Ausdrücke (Regular Expressions ... RE)

Hilfsmittel zum Üben parallel zum Lesen: <http://www.hft-leipzig.de/~krause/tools/index.html>  
Richtig gut und ausgefeilt ist das hier: <http://regexr.com/>

**Def.** (nach [1, S. 407]): Reguläre Ausdrücke sind Wörter über einem um technische Zeichen erweitertem Alphabet, die sich aus folgenden Teilen zusammensetzen können:

- Zeichenkette (gewisse Zeichen stehen nach \ , Quoting)
- Zeichenklasse: [...] oder das Komplement [^...]
- Wiederholung, Option: ( ...\* , ...+ , ...? , ...{n,m})
- Alternative: ...|...
- Gruppierung: (...)

Ein regulärer Ausdruck bildet nach einem formalen Schema ein **Muster (Pattern)**, dessen Vorkommen in einer Zeichenkette (o.ä.) gesucht wird (**Mustersuche**). Dabei kann entweder nur das Auftreten getestet werden (boolesches Ergebnis oder Anzahl) oder aber auch der gefundene, den reg. Ausdruck matchende String weiterverarbeitet oder gegen einen anderen ausgetauscht werden.

So kann man zum Beispiel eine Textdatei (z.B. das Log-File eines Firewalls oder Webserver), 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, die genauen Wertebereiche der Zahlen (max. 255) wollen wir hier nicht testen:

`\d+\.\d+\.\d+\.\d+` oder `(\d+\.){3}\d+` oder `(?:\d+\.){3}\d+`

Folgende Klassen werden im Java die Formulierung der Muster und das Matching eines Strings nach diesem Muster übernehmen:

- **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 konkreten Text sucht

**Die Gestaltung der Muster** ist zusammenfassend in der Java-Dokumentation [2] der entsprechenden Klasse **Pattern** beschrieben, im Folgenden einige wichtige Komponenten:

```
Characters
x      The character x
\\     The backslash character
\t     The tab character ('\u0009')
\n     The newline (line feed) character ('\u000A')

Logical operators
XY     X followed by Y
X|Y    Either X or Y
(X)    X (can be any composition of RE elements), as a capturing group
```

Jedes Paar runder Klammern (aber nicht (? .... )) fängt ihren gematchten Teilstring ein (**to capture**) und macht ihn im nachhinein verfügbar. Die Nummer der Gruppe entspricht der Nummer der öffnenden Klammer (von links). Da das Verfügbarmachen Rechenzeit kostet, gibt es auch **non capturing groups** (?:....) (siehe weiter unten).

Bsp.: `E(ll|rn)a` matcht `Erna` oder `Ella`

```
Character classes
[abc]    a, b, or c (simple class)
[^abc]   Any character except a, b, or c (negation, complementary class)
[a-zA-Z] a through z or A through Z, inclusive (range)

Predefined character classes
.        Any character (may or may not match line terminators)
```

\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [ \t\n\r\f]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z_0-9]
\W	A non-word character: [^\w]

Jedes Zeichen im Muster (z.B. X, \d, [0-9]) wird im untersuchten String genau einmal erwartet, es sei denn, danach steht ein **Quantifier** (z.B. X+, \d{2,}, [0-9]{4}):

Greedy quantifiers	
X?	X, once or not at all
X*	X, zero or more times
X+	X, one or more times
X{n}	X, exactly n times
X{n,}	X, at least n times
X{n,m}	X, at least n but not more than m times

Bsp.: \w\* matcht 0123456789abcdefghi

Bsp.: \w{4} matcht 0123456789abcdefghi

Bsp.: bei mehrfacher Anwendung nacheinander matcht \w{4}:

1. 0123456789abcdefghi
2. 0123456789abcdefghi
3. 0123456789abcdefghi
4. 0123456789abcdefghi

Bsp.: \w{4,7} matcht 0123456789abcdefghi

Bsp.: \w{4,7} matcht 01234==789abcdefghi

Bsp.: \w{4,7} matcht 012==789abcdefghi

Bsp.: bei mehrfacher Anwendung nacheinander matcht \w+ auf "Peter ist lieb!":

1. Peter ist lieb!
2. Peter ist lieb!
3. Peter ist lieb!

Reluctant quantifiers	
X??	X, once or not at all
...?	always an additional question mark ?
Possessive quantifiers	
X?+	X, once or not at all
...+	always an additional +

Das prinzipielle Verhalten der verschiedenen Quantifiergruppen wird wie folgt an einem regulären Ausdruck verdeutlicht, der auf den String 123abc1234567abc angewandt wird. Der Ausdruck besteht aus zwei Teilen (\w\*abc), der erste Teil (\w\*) kann alle Wortzeichen (also Buchstaben und Zahlen) treffen, der zweite (abc) braucht genau die Zeichenfolge abc. Damit ist der erste Teil in der Lage, auch das abc zu matchen, ob und wie er das tut, zeigt folgende Tabelle:

Verhalten von \w*	Gesamtausdruck	Matching	Gesamtausdruck matcht?
greedy (gierig)	\w*abc	123abc1234567abc	ja
possessive (besitzergreifend)	\w*+abc	123abc1234567abc	nein
reluctant (genügsam)	\w*?abc	123abc1234567abc	ja

Boundary matchers	
^	The beginning of a line
\$	The end of a line
\b	A word boundary

```

Special constructs (non-capturing)
(?:X)      X, as a non-capturing group
(?:idmsux-idmsux)  Nothing, but turns match flags idmsux on - off
(?:idmsux-idmsux:X) X, as a non-capturing group
                with the given flags idmsux on - off
(?:=X)     X, via zero-width positive lookahead
(?:!X)     X, via zero-width negative lookahead
(?:<=X)    X, via zero-width positive lookbehind
(?:<!X)    X, via zero-width negative lookbehind
(?:>X)     X, as an independent, non-capturing group

to the flags (RTFJavadoc!)
i  case insensitive
d  unix line mode
m  multi line mode
s  single line mode

```

**Übung 29** Bauen Sie einen RE, der auf eine achtstellige Binärzahl, aber nicht auf 00000000 matcht!

**Übung 30** Bauen Sie einen RE für einen Passworttest mit folgenden Bedingungen:

- min. 8 und max.32 bliebig Zeichen, aber keine Whitespaces,
- darunter mindestens 2 Kleinbuchstaben (a-z),
- mindestens 2 Großbuchstaben (A-Z),
- und mindestens 2 Ziffern (0-9).

Die Durchführung des Parsens obliegt einem Objekt der Klasse `Matcher`:

```

1 import java.util.regex.*;
2 ...
3 Pattern p = Pattern.compile("a+b"); // ein regulärer 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 `Matcher`-Objekt (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 String group(int n)`: liefert den der n-ten Gruppe des 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 31** *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 25)*

*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 ;-)*

...oder Sie schauen zum Vergnügen mal unter <http://regexcrossword.com/>.

## 4.5 Threadprogrammierung

### 4.5.1 Einfache Threads

Threads sind “Ablaufäden“ 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

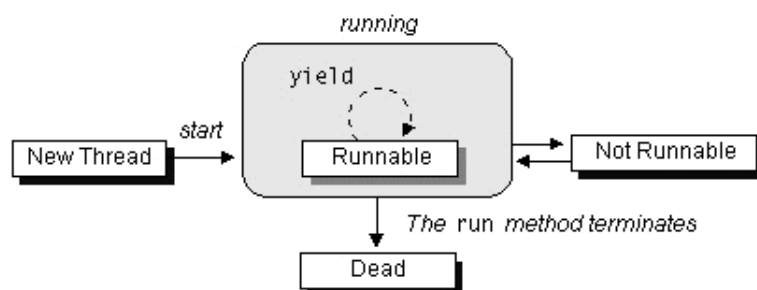
- Implementierung von Runnable, hier wird ein entsprechendes Objekt einem Thread-Konstruktor als Parameter übergeben, das entstehende Thread-Objekt erhält alle Fähigkeiten des Ausgangsobjektes,

```
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 [3])

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. Kompletterung 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	
<code>public void start()</code>	startet den Thread, die VM ruft <code>run()</code>
<code>public void stop()</code>	(deprecated)
<code>public void run()</code>	
<code>public void interrupt()</code>	setzt den Interruptstatus des Threads, bewirkt nur das Werfen einer <code>InterruptedException</code> aus <code>sleep</code> und <code>wait</code> (wodurch der Status zurückgesetzt wird)
<code>public boolean isInterrupted()</code>	liefert den Interruptstatus
<code>public static boolean interrupted()</code>	liefert den Interruptstatus des current Thread und setzt ihn zurück (damit kann die Anzahl der Interrupts erfaßt werden)
<code>static void sleep(int millis)</code>	der Thread pausiert, das Werfen von <code>InterruptedException</code> setzt den Interruptstatus des Threads zurück
<code>public final boolean isAlive()</code>	liefert true, falls der Thread noch lebt
<code>public final void join()</code>	wartet auf das Ende dieses Threads
<code>public void yield()</code>	gibt aus purer Freundlichkeit die Aktivität ab

Da die `stop()`-Methode deprecated ist, sollte man einen Thread immer durch das Erreichen des Endes seiner `run()`-Methode beenden.

Es folgt ein Beispiel, in dem durch die Methode `interrupt()` das Signal zum Beenden geschickt wird. Würde die Variable `ende` von außen auf `true` gesetzt, würde es bis zum Threadende etwas länger dauern, da `sleep` oder `wait` nicht beendet werden würden.

Listing 1: Beenden eines Threads durch Signal (Interrupt) von aussen

```
1 public class Runner extends Thread {
2
3     public static void main(String[] args) {
4
5         Runner r = new Runner();
6         r.start();
7         ...
8         r.interrupt(); // setzt den Interrupt-Zustand des Threads
9         ...
10
11     }
12
13     private boolean ende = false;
14
15     public void run() {
16         while ( !ende ) {
17             ... // die Handlungen des Threads
18             ... // zu Beginn der Schleife
19
20             try { sleep(1000); }
21             catch (InterruptedException e) {
22                 // Werfen von InterruptedException setzt den Interrupt-Status zurueck
23                 interrupt(); // Neusetzen des Interruptstatus
24             }
25             if ( interrupted() ) // fragt den Interrupt-Zustand ab
26                 ende = true;
27         }
28     }
29 }
```

**Übung 32** Bauen Sie eine Graphikapplikation, die über einen typisierten Vector (oder andere das `List<E>`-Interface implementierende Klasse) 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, was dieser mit einer auf die Konsole geschriebenen Bemerkung bemerken soll.

Als Zusatzaufgabe sollen Sie über Synchronisation (wird im folgenden Abschnitt behandelt) verhindern, dass sich die Threads bei der Ausgabe auf die Konsole (`System.out`) gegenseitig stören!

#### 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 Mehrfachzugriffs. 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 das Objekt, das durch `varname` referenziert wird (keine elementaren Datentypen, Typ muß `Object` sein), 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 der gleichen 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`

#### **Ein Beispiel zur Thread-Synchronisation** (nach dem `CubbyHole`-Beispiel im Java-Tutorial [3]):

Eine Aufrufklasse:

```
1 public class ProducerConsumerTest {
2     public static void main(String[] args) {
3         int cycles = 10;
4         Postfach c = new Postfach();
5         Producer p1 = new Producer(c, 1, cycles);
6         Consumer c1 = new Consumer(c, 1, cycles);
7
8         p1.start();
9         c1.start();
10    }
11 }
```

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:

```
13 public class Postfach {
14     private int content;
15     private boolean available = false;
```

```

16
17 public synchronized int get() {
18     while (available == false) {
19         try {
20             wait();
21         } catch (InterruptedException e) { }
22     }
23     available = false;
24     System.out.println("Consumer got: " + value);
25     notifyAll();
26     return content;
27 }
28
29 public synchronized void put(int value) {
30     while (available == true) {
31         try {
32             wait();
33         } catch (InterruptedException e) { }
34     }
35     content = value;
36     available = true;
37     System.out.println("Producer put: " + value);
38     notifyAll();
39 }
40 }

```

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

```

42 public class Producer extends Thread {
43     private Postfach post;
44     private int number;
45     private int cycles;
46     public Producer(Postfach c, int number, int cycles) {
47         post = c;
48         this.number = number;
49         this.cycles = cycles;
50     }
51
52     public void run() {
53         for (int i = 0; i < cycles; i++) {
54             post.put(i);
55         }
56         try {
57             sleep((int)(Math.random() * 100));
58         } catch (InterruptedException e) { }
59     }
60 }
61 }

```

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

```

63 public class Consumer extends Thread {
64     private Postfach post;
65     private int number;
66     private int cycles;
67
68     public Consumer(Postfach c, int number, int cycles) {
69         post = c;
70         this.number = number;
71         this.cycles = cycles;

```

```

72     }
73
74     public void run() {
75         int value = 0;
76         for (int i = 0; i < cycles; i++) {
77             value = post.get();
78         }
79     }
80 }

```

**Übung 33** 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!

**Modifikation der Klasse Postfach** (siehe S.67, Abschn. 4.5.2):

Einrichten einer zweiten Information und eines zweiten Methodenpaares `get2/put2`.

Damit die zweite Information unabhängig von der ersten genutzt werden kann, sollte jetzt ein anderer, auch zum Postfach-Objekt gehörender Türsteher zum Einsatz kommen, der `synchronized`-Blöcke steuert. Die Methoden `wait()` und `notifyAll()` werden jetzt nicht mehr auf dem zu den `synchronized`-Methoden korrespondierenden `this` gerufen, sondern auf dem gewählten Türsteher. Damit gibt es für `content` und `content2` zwei unabhängige, sich bzgl. Parallelität nicht beeinflussende Bereiche.

Das sieht etwa so aus:

```

1  public class Postfach2 {
2  //////////////// Methodenpaar fuer content mit Tuersteher this:
3      private int content;
4      private boolean available = false;
5      public synchronized int get() { // this ist Tuersteher
6          ...
7          this.wait();           // this ist Tuersteher
8          ...
9          this.notifyAll();      // this ist Tuersteher
10         ...
11     }
12     public synchronized void put(int value) { // this ist Tuersteher
13         ... // wie bei get()
14     }
15 //////////////// Methodenpaar fuer content2 mit Tuersteher ts2:
16     private int content2;
17     private boolean available2 = false;
18     private Object ts2 = new Object(); // Tuersteher
19     public int get2() {
20         synchronized(ts2){     // ts2 ist Tuersteher
21             ...
22             ts2.wait();        // ts2 ist Tuersteher
23             ...
24             ts2.notifyAll();   // ts2 ist Tuersteher
25             ...
26         }
27     }
28     public void put2(int value) {
29         synchronized(ts2){     // ts2 ist Tuersteher
30             ... // wie bei get2()
31         }
32     }
33 }

```

Die Implementierung der Methoden entspricht der in der Klasse `Postfach` vorgestellten.

**Übung 34** Beschreiben und begründen Sie (unter der Bedingung der Sichtbarkeit aller Variablen und Methoden), wie die Abarbeitung zweier Threads (Thread-1 und Thread-2) in Betracht des folgenden Quelltextsnippets

```
1 class MyCalculator {
2     static MyCalculator m1 = new MyCalculator();
3     static MyCalculator m2 = new MyCalculator();
4
5     static String st = new String("statische Variable!");
6     String ob1 = new String("Objektvariable!");
7     String ob2 = new String("Objektvariable 2!");
8
9     MyCalculator(){ ob2 = st; }
10
11    synchronized void calc1() { /* lange Rechnung */ }
12    void calc2() {
13        synchronized (this) { /* lange Rechnung */ }
14    }
15    void calc3() {
16        synchronized (st) { /* lange Rechnung */ }
17    }
18    void calc4() {
19        synchronized (ob1) { /* lange Rechnung */ }
20    }
21    void calc5() {
22        synchronized (ob2) { /* lange Rechnung */ }
23    }
24 }
```

zeitlich verläuft (parallel oder nacheinander), wenn

- Thread-1 m1.calc1() und Thread-2 m2.calc1() ruft?
- Thread-1 m1.calc2() und Thread-2 m2.calc2() ruft?
- Thread-1 m1.calc3() und Thread-2 m2.calc3() ruft?
- Thread-1 m1.calc4() und Thread-2 m2.calc4() ruft?
- Thread-1 m1.calc5() und Thread-2 m2.calc5() ruft?
- Thread-1 m1.calc1() und Thread-2 m1.calc2() ruft?
- Thread-1 m1.calc3() und Thread-2 m1.calc5() ruft?

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

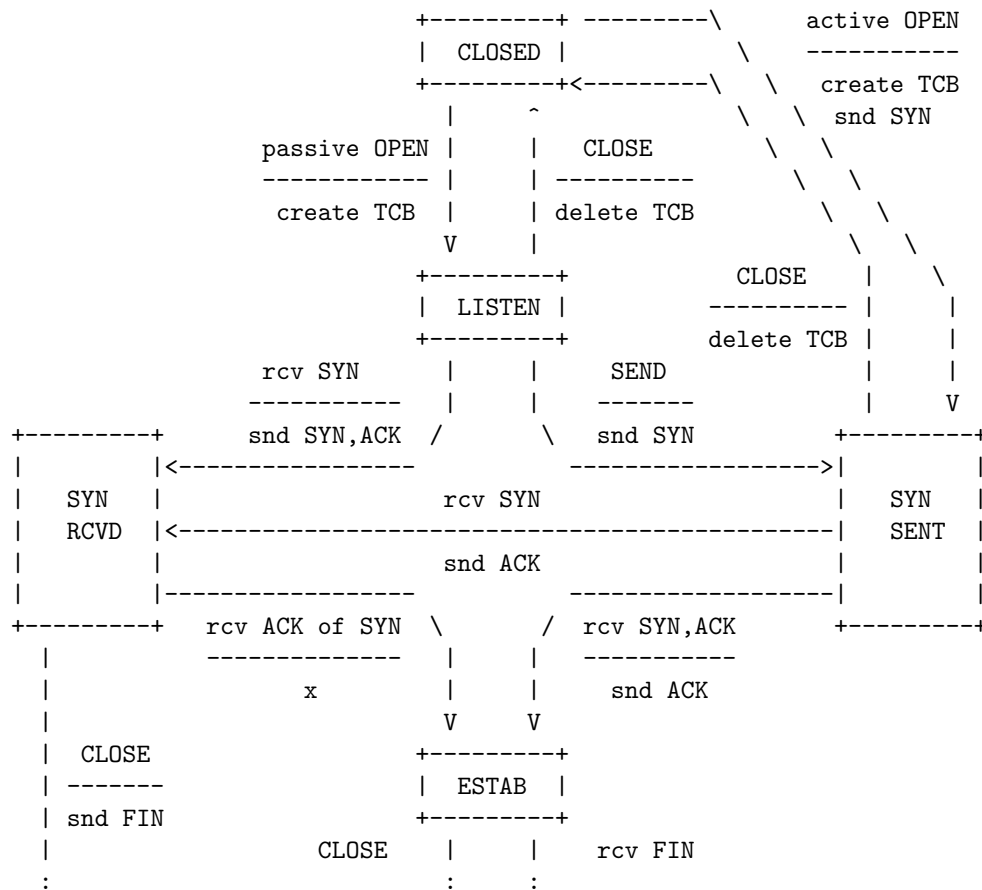
Vereinfachtes 4-Schichten-Modell / TCP/IP-Referenzmodell

Abb./Begriffe nach [5] , [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.*` (Netzwerkprogrammierung)  
`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.49) 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:

```

1 // Ein Echo-Server, der als Diener eines Herrn mit einem Client redet
2 // Abbruch mit ESC-Taste im Client
3 import java.net.*;
4 import java.io.*;
5 class ServerMono {
6     static void main (String [] a) throws java.io.IOException{
7         // fuer saubere Fehlerbehandlung:
8         // entfernen der throws-Klausel, compilieren,
9         // Fehler analysieren, try catch !!!
10        int port = 1234;
11        ServerSocket ss = new ServerSocket (port);
12        while (true) {
13            int i=0;
14            System.out.println("wartet auf Anfrage");
15            Socket con = ss.accept();
16            // accept horcht, bis jemand auf dem Port anklopft
17            System.out.println("neue Verbindung");
18            InputStream in = con.getInputStream();
19            OutputStream out = con.getOutputStream();
20            while (i!=27) { // ESC
21                char c= (char) (i=in.read());
22                System.out.println("empfangen :"+ c + " : , Wert = " + i);
23                out.write(i);
24            }
25            con.close();
26        }
27        //ss.close();
28        // Schließen des ServerSockets, wenn oben keine Endlos-Schleife existiert
29    }

```



30 }

und nun einen multithreaded Server für parallele Anfragen:

```
1 // Ein multithreaded Echo-Server,
2 // der mit mehreren Clients reden kann
3 import java.net.*;
4 import java.io.*;
5 class ServerMulti {
6     static void main (String [] a) throws Exception{
7         int port = 1234;
8         ServerSocket ss = new ServerSocket (port);
9         while (true) {
10             int i=0;
11             System.out.println("wartet auf Anfrage");
12             Socket con = ss.accept();
13             System.out.println("neue Verbindung");
14             // wird an den ServerThread uebergeben:
15             ServerThread sthr = new ServerThread(con);
16             sthr.start();
17             // und der ServerMulti kann wieder zum accept gehen
18         }
19         //ss.close();
20         // Schließen des ServerSockets, wenn oben keine Endlos-Schleife existiert
21     }
22 }
```

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

```
24 class ServerThread extends Thread {
25     // realisiert die konkrete Client-Verbindung
26     Socket con = null;
27     public ServerThread (Socket con){
28         this.con = con;
29     }
30     public void run() {
31         int i = 0;
32         System.out.println("neuer ServerThread");
33         try {
34             InputStream in = con.getInputStream();
35             OutputStream out = con.getOutputStream();
36             while (i!=27) { // ESC
37                 char c= (char) (i=in.read());
38                 System.out.println("empfangen :" + c + ": , Wert = " + i);
39                 out.write(i);
40             }
41             con.close();
42         }
43         catch (java.io.IOException ioe) {
44             System.out.println (ioe);
45         }
46     }
47 }
```

### 4.6.3 Ein variables Serversystem

Es soll ein Server entwickelt werden, der multicientfähig 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.33) passieren:

Die **Hauptklasse** des Servers:

```
1 import java.net.*;
2 import java.lang.reflect.*;
3
4 class ServerMulti {
5     static boolean STOP = false;
6     public static void main (String [] a) throws Exception {
7
8         int port = 2345;
9         String ThreadKlasse = a[0];
10        Class m = Class.forName(ThreadKlasse);
11
12        //                                {Class-Array für die KonstruktorparameterTypen}
13        Constructor c = m.getDeclaredConstructor (new Class[] {Socket.class});
14        //                                {Class.forName("java.net.Socket")}
15
16        // ein ObjektArray für die Konstruktorparameter:
17        Object [] po = new Object [1];
18
19        ServerSocket ss = new ServerSocket (port);
20        while (!STOP) {
21            int i=0;
22            System.out.println("wartet auf Anfrage");
23            Socket con = ss.accept();
24            System.out.println("neue Verbindung");
25            po[0] = con;
26            // wird an den ServerThread uebergeben:
27            ServerThread sthr = (ServerThread) c.newInstance(po);
28
29            sthr.start();
30
31        }
32        ss.close();
33    }
34 }
```

Die **abstrakte Oberklasse** aller Serverthreads:

```
1 import java.net.*;
2
3 abstract class ServerThread extends Thread {
4
5     Socket con = null;
6
7     public ServerThread (Socket con){
8         this.con = con;
9     }
10
11     abstract public void run();
12 }
```

12  
13 }

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

```
1 import java.net.*;
2 import java.io.*;
3
4 class ServerThreadCharEcho extends ServerThread {
5     // realisiert die konkrete Client-Verbindung
6
7     public ServerThreadCharEcho (Socket con){
8         super(con);
9     }
10
11     public void run() {
12         int i = 0;
13         System.out.println("neuer ServerThread");
14         try {
15             InputStream in = con.getInputStream();
16             OutputStream out = con.getOutputStream();
17             System.out.println("Klasse des InputStreams : "+in.getClass().getName());
18             System.out.println("Port (remote) : "+con.getPort());
19             System.out.println("Port (local) : "+con.getLocalPort());
20             while ((i=in.read())!=-1) { // oder 27 für ESC
21                 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 35** Bauen Sie den Server so um, daß er die Zeichen vom Client zeilenweise liest und zurückschickt! (siehe Abschnitt 4.3.1 S.49)

**Übung 36** Modifizieren Sie den Server aus Übung 35 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

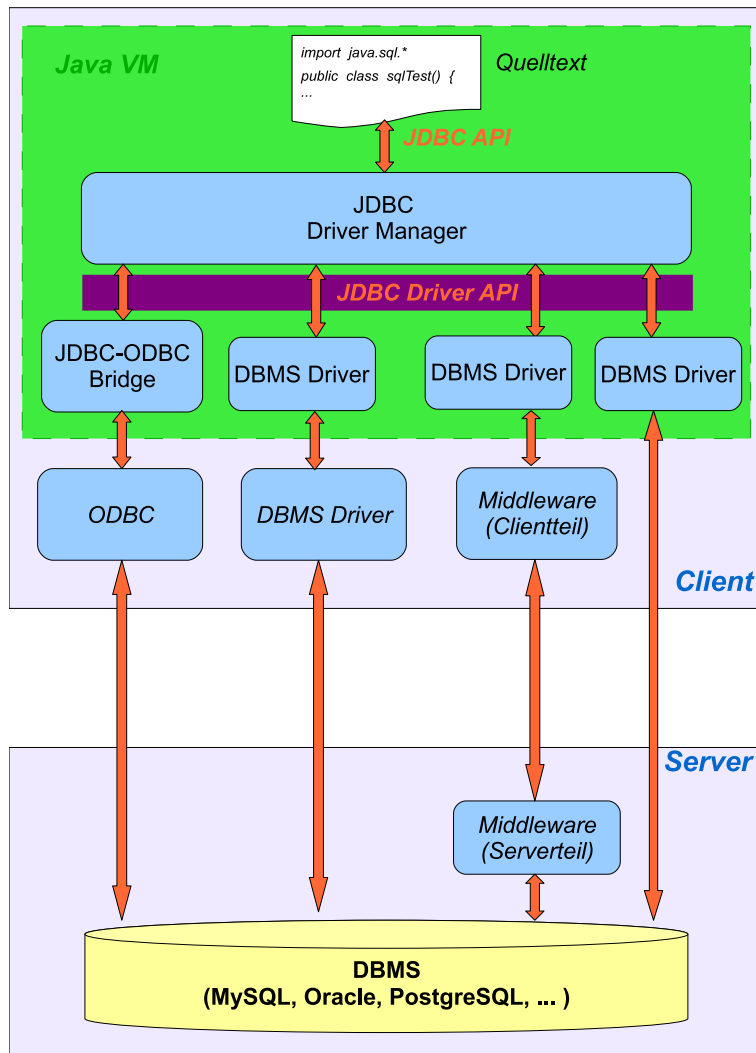
- anhand des Trails JDBC Database Access im Java-Tutorial [3]
- oder über das Kapitel Datenbankzugriffe mit JDBC in Handbuch der Javaprogrammierung [5]

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-Netzwerkcommunication mit Gegenstück auf dem Server) serverseitig auf die proprietären Datenbanktreiber zugreift
4. ein JDBC-Treiber, der direkt mit der Datenbank in deren Protokoll spricht



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

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

- PreparedStatement
- ResultSet
- DatabaseMetaData
- ...

### Ein simples Beispiel zur Illustration (teilweise aus der Java-Dokumentation [2])

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     // <subprotokoll> = 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 37** 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 (mysql.driver.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 38** 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] C. HORN, I. O. KERNER, AND P. FORBRIG, *Lehr- und Übungsbuch Informatik, Band 1*, Fachbuchverlag Leipzig, 2001.
- [2] *Jdk 8 documentation*. <http://docs.oracle.com/javase/8/>.
- [3] *The Java Tutorial*. <https://docs.oracle.com/javase/tutorial/index.html>.
- [4] F. JOBST, *Programmieren in Java*, Carl Hanser Verlag, 2001.
- [5] G. KRÜGER, *Handbuch der Java-Programmierung*, Addison Wesley, 2011.
- [6] P. NIEMEYER AND J. KNUDSEN, *Learning Java*, O'Reilly, 2000.
- [7] C. ULLENBOOM, *Java ist auch eine Insel*, Galileo Computing, 2007.