

# Einführung in die strukturierte und objektbasierte Programmierung (620.200, »ESOP«)

Assoc. Prof. Dr. Mathias Lux  
ITEC / AAU

# Modalitäten



- Wir sind hier in der Vorlesung
- Prüfung am Ende des Semesters
  - 04.02. 2015, 11:00 Uhr, HS A
  - 100 Minuten, 100 Punkte
  - An- bzw. Abmeldung nicht vergessen!

# Termine



- Donnerstags, 14-16 Uhr, HS B (c.t.)
  - Ausser 30.10. & 6.11. 2014
- Zusätzlich
  - Mo, 06.10. 2014, 11-13, HS B (s.t.)
  - Mo, 13.10. 2014, 11-13, HS B (s.t.)



# Übung und Tutorium



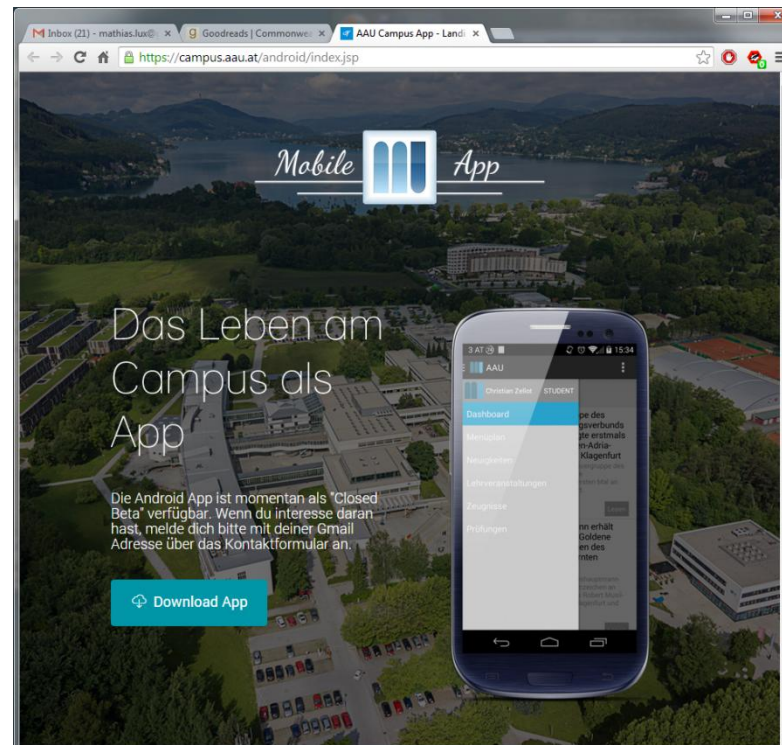
- Übung ab kommender Woche
  - Laptop mitbringen falls vorhanden!
- Tutorium
  - Bitte anmelden!
  - Je PR ein Tutor
  - Eine zusätzliche Einheit



# AAU Campus App



- <https://campus.aau.at/android/index.jsp>



# Umfrage im Moodle ...



# Literatur



Hanspeter Mössenböck, *Sprechen Sie Java?  
Eine Einführung in das systematische  
Programmieren*

5. Auflage, dpunkt.verlag, 2014

ISBN 978-3-86490-099-0

**Buchaktion!**



# Java Dokumentation



- Java API Doc
  - <http://docs.oracle.com/javase/8/docs/api/>
- Java Tutorials
  - <http://docs.oracle.com/javase/tutorial/>



# Wie soll ich Java lernen?



1. Spaß am Programmieren entwickeln
2. Java Tutorials durcharbeiten
3. VO und PR besuchen

# Motivation - Warum Lux?



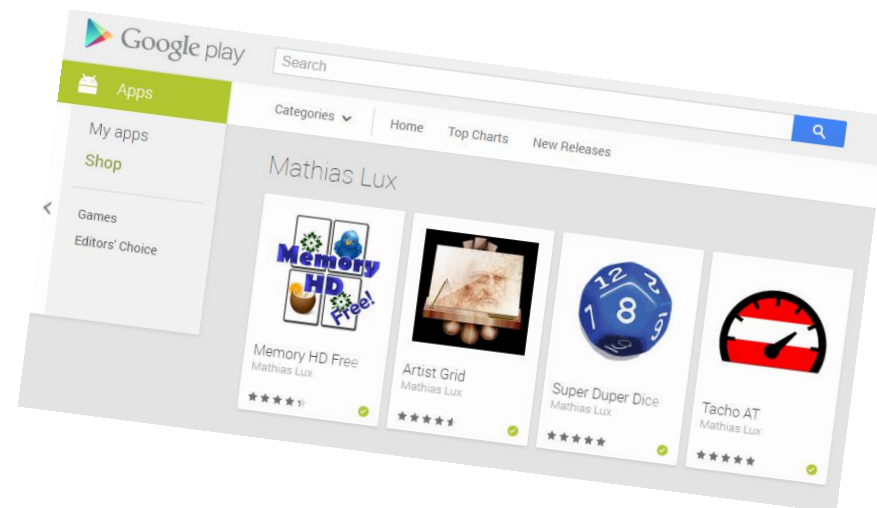
Paqo Fruit Masters  
Paqo International

★★★★★



Picalicious  
econob GmbH

★★★★★



# Motivation



- **Notwendigkeit in der Forschung**
  - Grand Challenge Projekte
- **Projekte aus Processing**
- **Games, usw.**

# Was ist Programmieren?



... die Lösung eines Problems so exakt beschreiben, dass es ein Computer lösen kann

Vgl. Kochrezept, Bedienungsanleitung.



# Programmieren ist ...



- eine kreative Tätigkeit
- eine Ingeieurstätigkeit
- schwierig, wenn man es gut machen will

# Programm

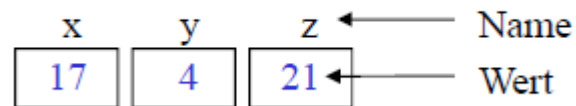


Programm = Daten + Befehle

# Daten



- Menge adressierbarer Speicherzellen



- Daten binär gespeichert (z.B.  $17 = 10001$ )
- Binärspeicherung ist universell
  - (Zahlen, Texte, Bilder, Ton, ...)
- 1 Byte = 8 Bit
- 1 Wort = 4 Byte (typischerweise)

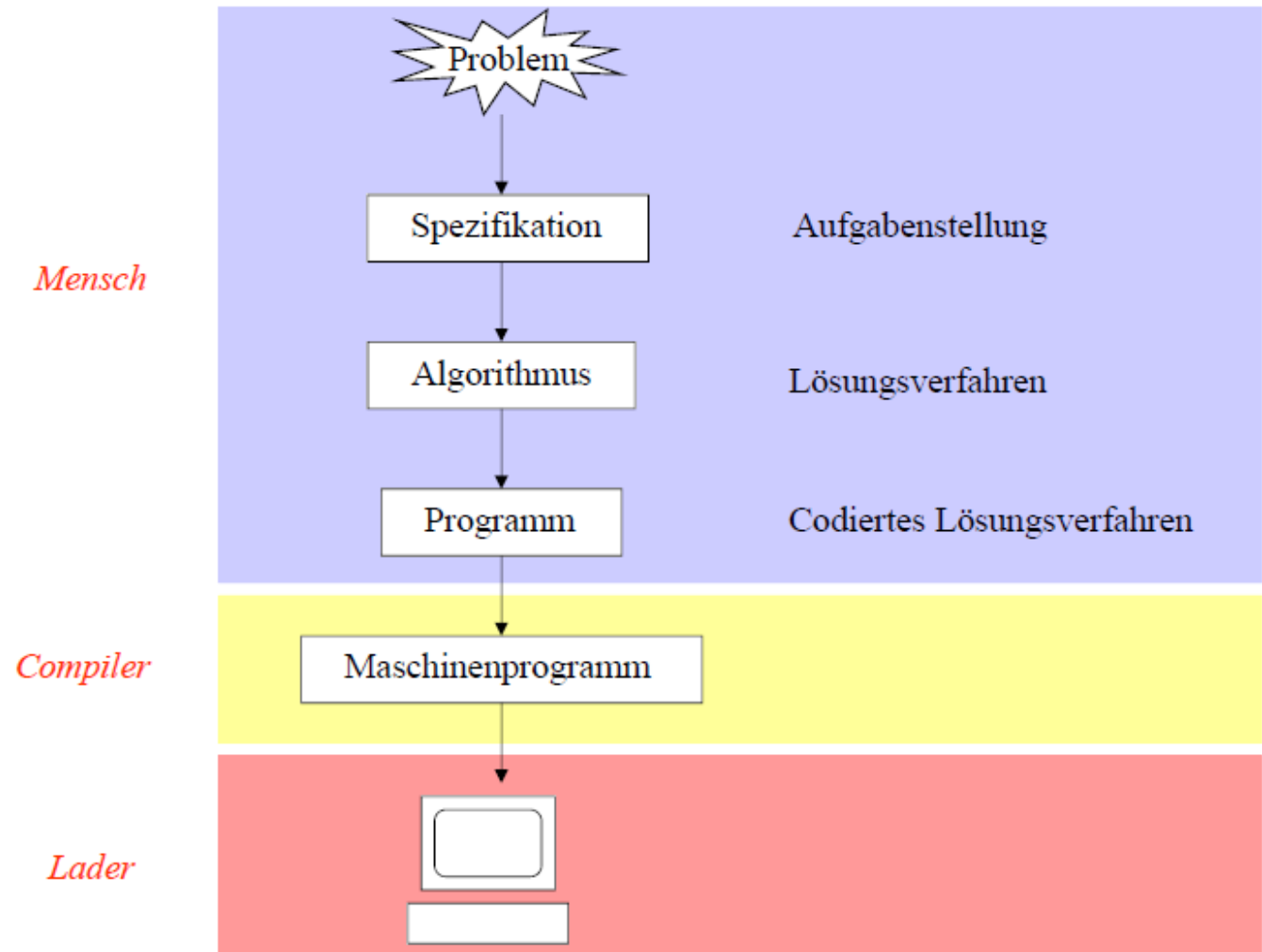
# Befehle



- Operationen mit den Speicherzellen

<i>Maschinensprache</i>	<i>Hochsprache</i>
$ACC \leftarrow x$ // Lade Zelle x	$z = x + y;$
$ACC \leftarrow ACC + y$ // Addiere Zelle y	
$z \leftarrow ACC$ // Speichere Ergebnis in Zelle z	

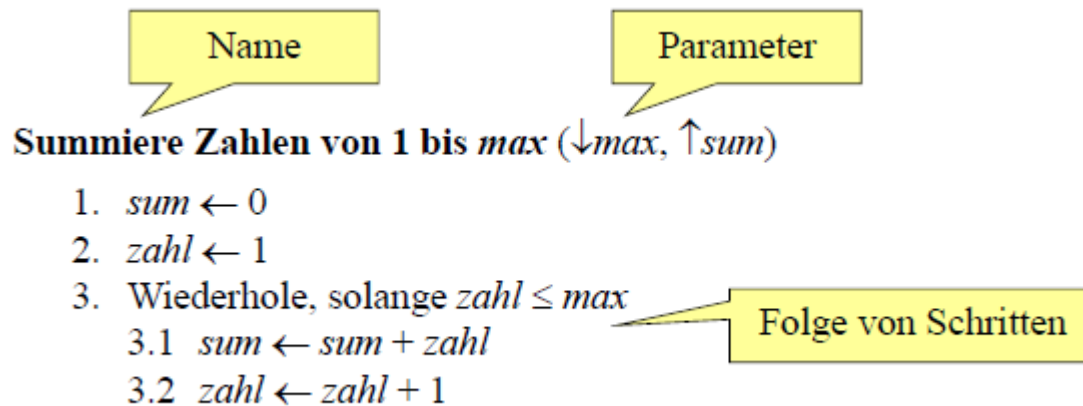
# Programmerstellung



# Algorithmus



- Schrittweises, präzises Verfahren zur Lösung eines Problems



- Programm = Beschreibung eines Algorithmus in einer Programmiersprache

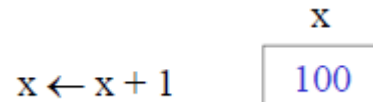
# Variablen



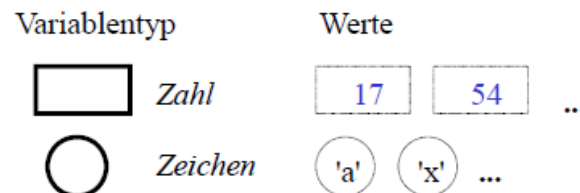
- Sind benannte Behälter für Werte



- Können ihren Wert ändern



- Haben einen Datentyp  
– = Menge erlaubter Werte



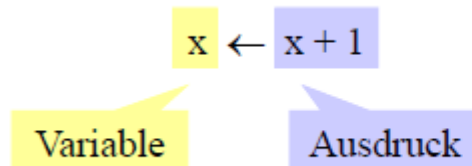
Typ  $\cong$  Form

- in eine Zahlenvariable passen nur Zahlen
- in eine Zeichenvariable passen nur Zeichen

# Anweisungen

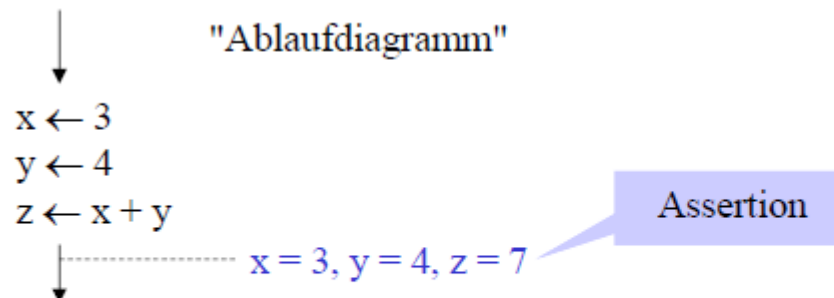


- Wertzuweisung



1. werte Ausdruck aus
2. weise seinen Wert der Variablen zu

- Anweisungsfolge (Sequenz)



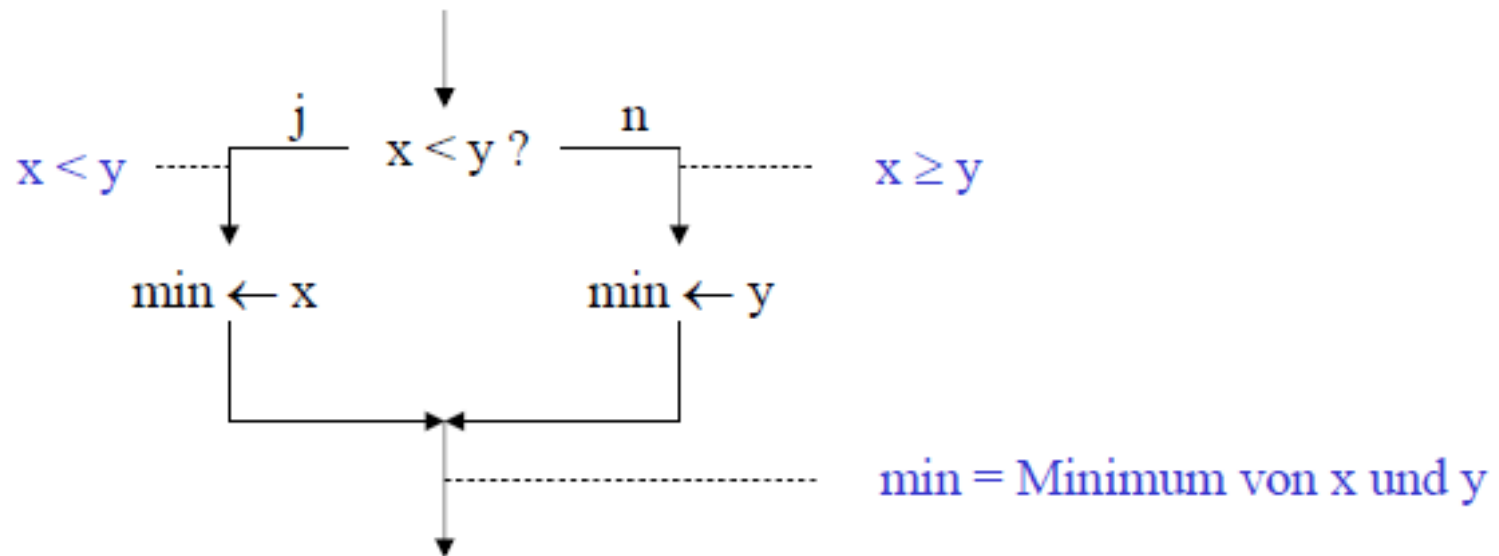
*Assertion (Zusicherung)*  
Aussage über den Zustand des Algorithmus  
an einer bestimmten Stelle



# Anweisungen



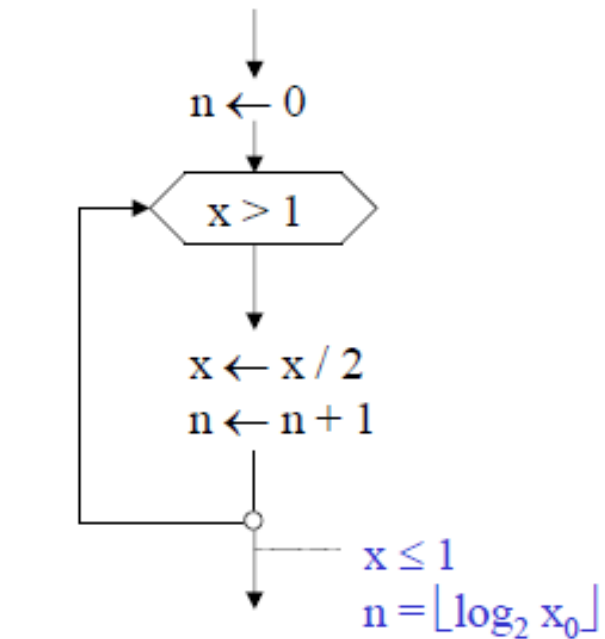
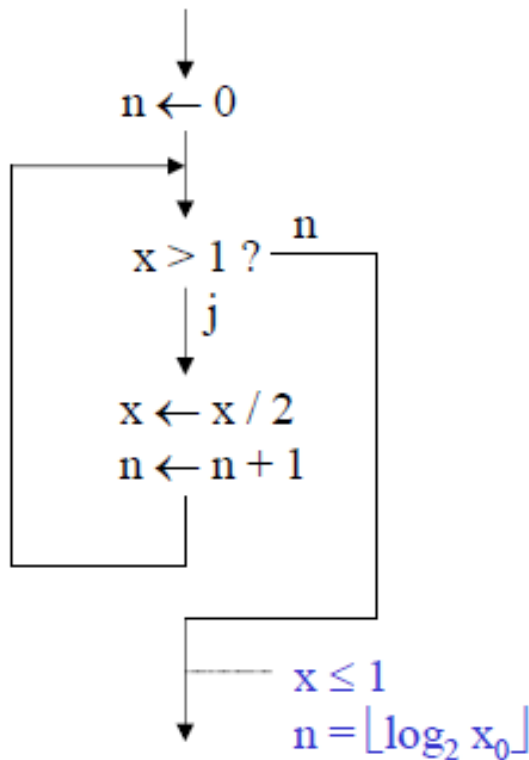
- Auswahl
  - auch Verzweigung, Abfrage, Selektion



# Anweisungen



- Wiederholung
  - Auch: Schleife, Iteration



Alternative Darstellung

# Beispiel: Vertauschen zweier Variablen



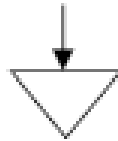
Swap ( $\uparrow x$ ,  $\uparrow y$ )



$h \leftarrow x$

$x \leftarrow y$

$y \leftarrow h$



*Schreibtischtest*

x	y	h
<del>3</del>	<del>2</del>	3
2	3	

# Beispiel: Vertauschen zweier Variablen



```
int x = 10;  
int y = -5;  
int h;
```

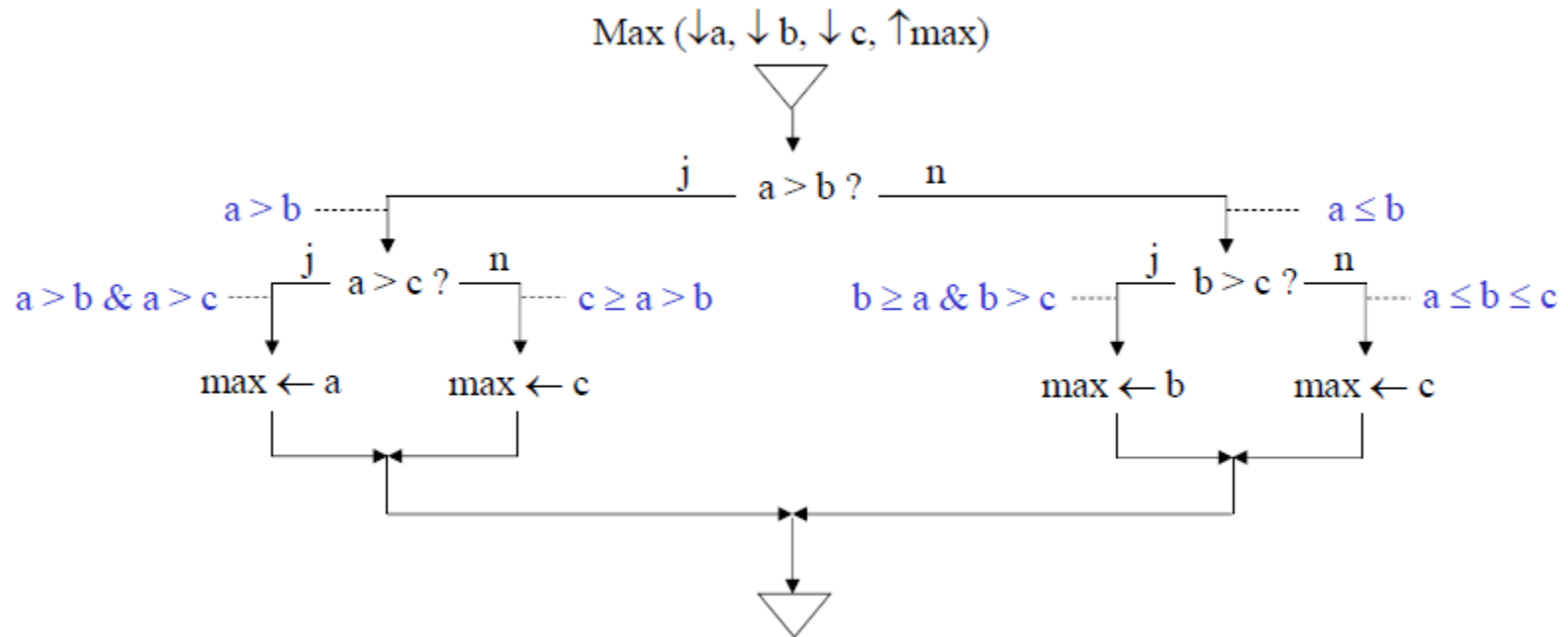
```
println(x);  
println(y);
```

```
h = x;  
x = y;  
y = h;
```

```
println(x);  
println(y);
```

- Source Code für Processing
- Processing ist „wie Java“
- int ... Datentyp
- ; ... beendet Anweisung
- println() ... Funktion zur Ausgabe

# Beispiel: Maximum dreier Zahlen bestimmen



# Beispiel: Maximum dreier Zahlen bestimmen



```
int a = 11;
int b = 12;
int c = 13;
int max;

if (a<b) {
    if (b<c) {
        max = c;
    } else {
        max = b;
    }
} else {
    if (a<c) {
        max = c;
    } else {
        max = a;
    }
}

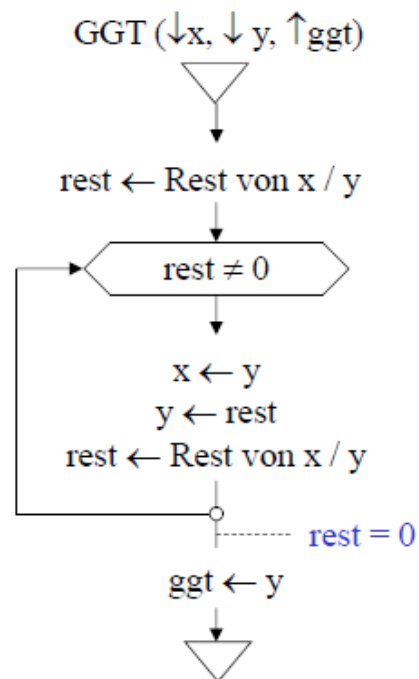
println(max);
```

- Source Code für Processing
- if (test) {...}
- else {...}

# Beispiel: Euklidischer Algorithmus



- Berechnet den größten gemeinsamen Teiler zweier Zahlen  $x$  und  $y$



*Schreibtischtest*

x	y	rest
<del>28</del>	<del>20</del>	8
<del>20</del>	<del>8</del>	4
8	4	0

Warum funktioniert dieser Algorithmus?

(ggt teilt  $x$ ) & (ggt teilt  $y$ )

$\Rightarrow x = i \cdot \text{ggt}, y = j \cdot \text{ggt}, (x - y) = (i - j) \cdot \text{ggt}$

$\Rightarrow \text{ggt teilt } (x - y)$

$\Rightarrow \text{ggt teilt } (x - q \cdot y)$

$\Rightarrow \text{ggt teilt rest}$

$\Rightarrow \text{GGT}(x, y) = \text{GGT}(y, \text{rest})$

# Beispiel: Euklidischer Algorithmus



```
int x = 21;
int y = 14;

int rest = x % y;

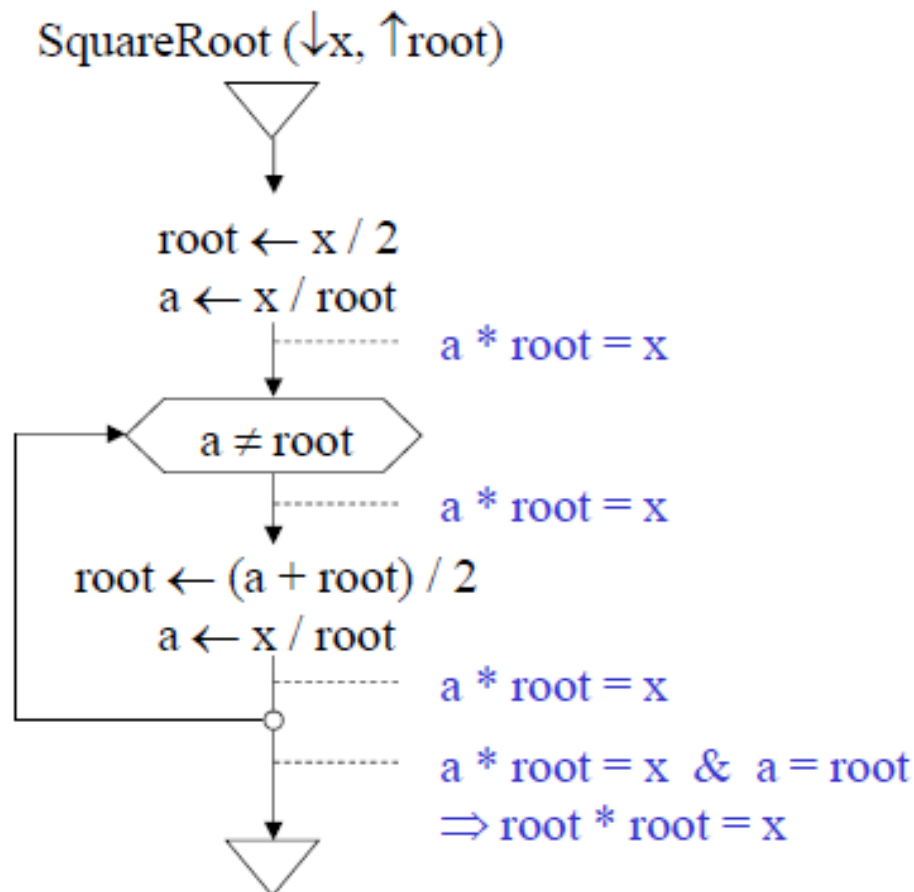
while (rest != 0) {
    x = y;
    y = rest;
    rest = x % y;
}

println(y);
```

- Source Code für Processing
- While (test) {..}
- % ... modulo



# Beispiel: Quadratwurzel



Schreibtischttest

x	root	a
10	<del>5</del>	<del>2</del>
	<del>3.5</del>	<del>2.85714</del>
	<del>3.17857</del>	<del>3.14607</del>
	<del>3.16232</del>	<del>3.16223</del>
	3.16228	3.16228

# Beispiel: Quadratwurzel



```
float x = 10;

float root = x / 2;
float a = x / root;

while (a != root) {
    root = (a + root) / 2;
    a = x / root;
}

println(root);
```

- Source Code für Processing
- float ... Datentyp
- / ... Division
- Tipp: float nicht auf Gleichheit prüfen!
  - $|a - \text{root}| < 0,00001$

# Beschreibung von Programmiersprachen



- Syntax
  - Regeln, nach denen Sätze gebaut werden dürfen
  - z.B.: Zuweisung = Variable „<-“ Ausdruck
- Semantik
  - Bedeutung der Sätze
  - z.B.: werte Ausdruck aus und weise ihn der Variablen zu

# Beschreibung von Programmiersprachen

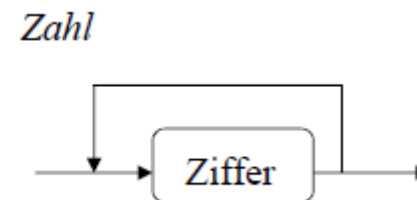
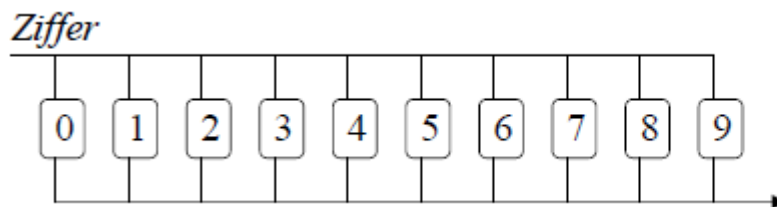


- Grammatik

- Menge von Syntaxregeln

- z.B. Grammatik der ganzen Zahlen

- Ziffer = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
    - Zahl = Ziffer {Ziffer}.



# EBNF (Erweiterte Backus-Naur-Form)



## Beispiele

- *Grammatik der Gleitkommazahlen*
  - Zahl = Ziffer {Ziffer}.
  - Gleitkommazahl = Zahl "." Zahl ["E" ["+" | "-"] Zahl].
- *Grammatik der If-Anweisung*
  - IfAnweisung = "if" "(" Ausdruck ")" Anweisung ["else" Anweisung].

Metazeichen	Bedeutung	Beispiel	beschreibt
=	trennt Regelseiten	A = x y z .	
.	schließt Regel ab		
	trennt Alternativen	x   y	x, y
()	klammert Alternativen	(x   y) z	xz, yz
[]	wahlweises Vorkommen	[x] y	xy, y
{}	0..n-maliges Vorkommen	{x} y	y, xy, xxy, xxxy, ...

# Programmiersprachen



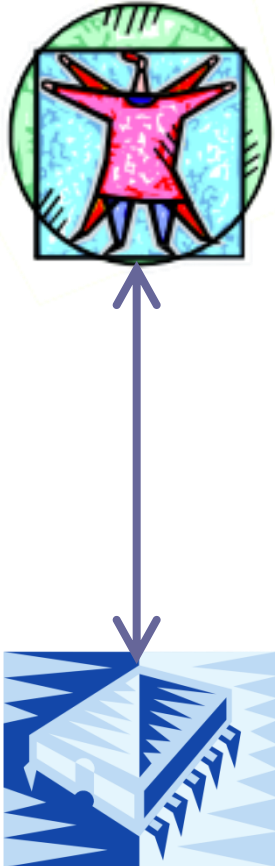
- Maschinell (durch ein Programm) übersetzbare formale Sprachen
  - Ein Programm ist ein »Text« in einer formalen Sprache
- Viele verschiedene formale Sprachen
  - Java, Python, C, C++, Objective C, Pascal, Modula, Perl, Basic, C#, JavaScript, Dart, Erlang, LUA uvm.

# Programmiersprachen



- **Compiler:** Programmtext wird
  - von einem Übersetzungsprogramm
  - in Maschinensprache übersetzt
  - Bsp. C, C++
- **Interpreter:**
  - Programmtext wird unmittelbar, schrittweise ausgeführt
  - Bsp. Python, Ruby

# Algorithmennotation



Grafische Notationen	Verbale Notationen
Höhere Programmiersprachen (wie Java)	
Assemblersprachen	
Maschinencode (binär)	
Hardware (elektrische Signale)	



# Verbale Darstellung

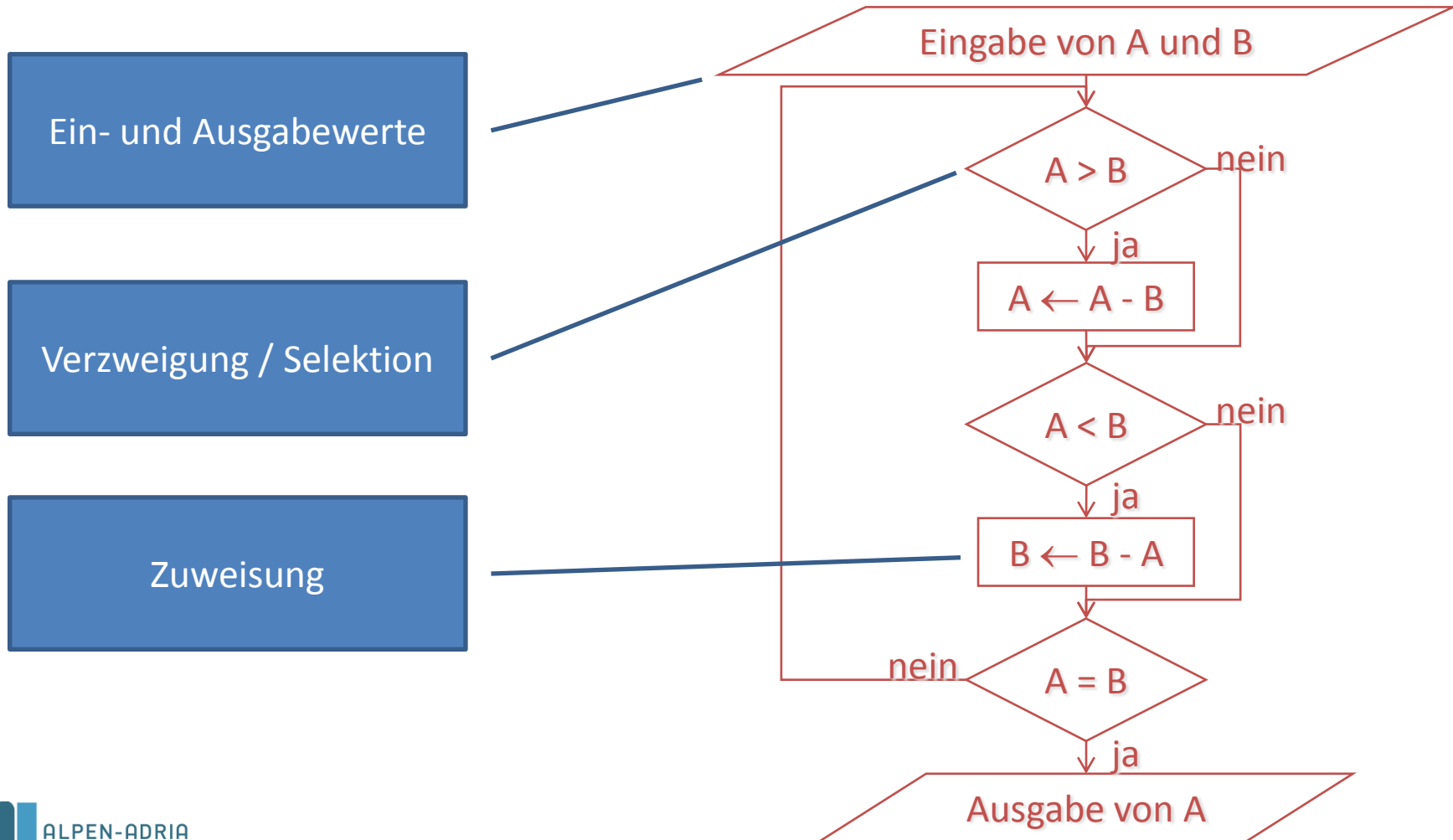


- Beschreibung in natürlicher Sprache

Euklidischer Algorithmus  $\text{ggT}(A, B)$

0. Eingabe von A und B
1. Wenn A größer B, dann subtrahiere B von A und weise das Ergebnis A zu
2. Wenn A kleiner B, dann subtrahiere A von B und weise das Ergebnis B zu
3. Wenn A ungleich B, weiter bei Schritt 1.
4. Das Ergebnis ist A (oder B)

# Flussdiagramm



# Flussdiagramm



## Nachteile eines Flussdiagramms

- Oft unstrukturiert, keine formalen Vorgaben
- Für fremde Leser nicht verständlich, nicht teamfähig
- Nicht wartbar, nicht erweiterbar

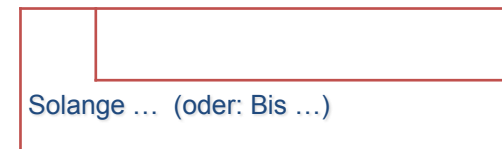
# Nassi-Shneiderman-Diagramm



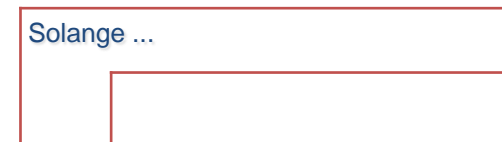
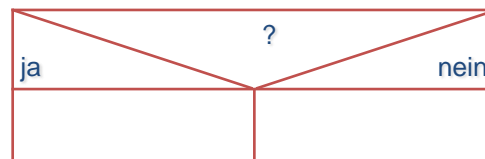
- Einschränkung der Darstellungsmöglichkeiten, führt zu strukturierteren Graphen
- Sequenz



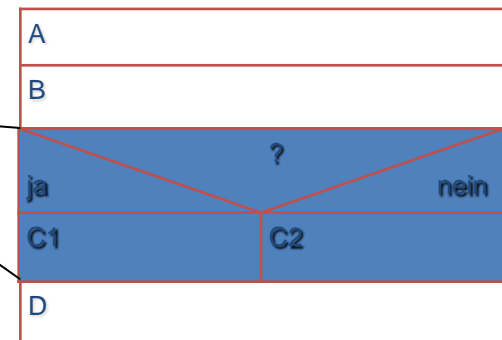
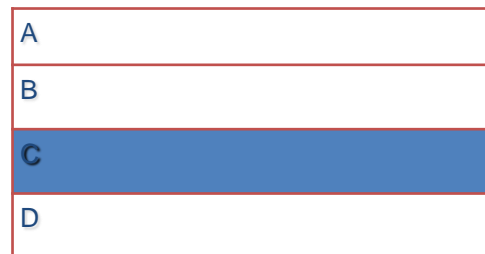
Wiederholung / Schleife



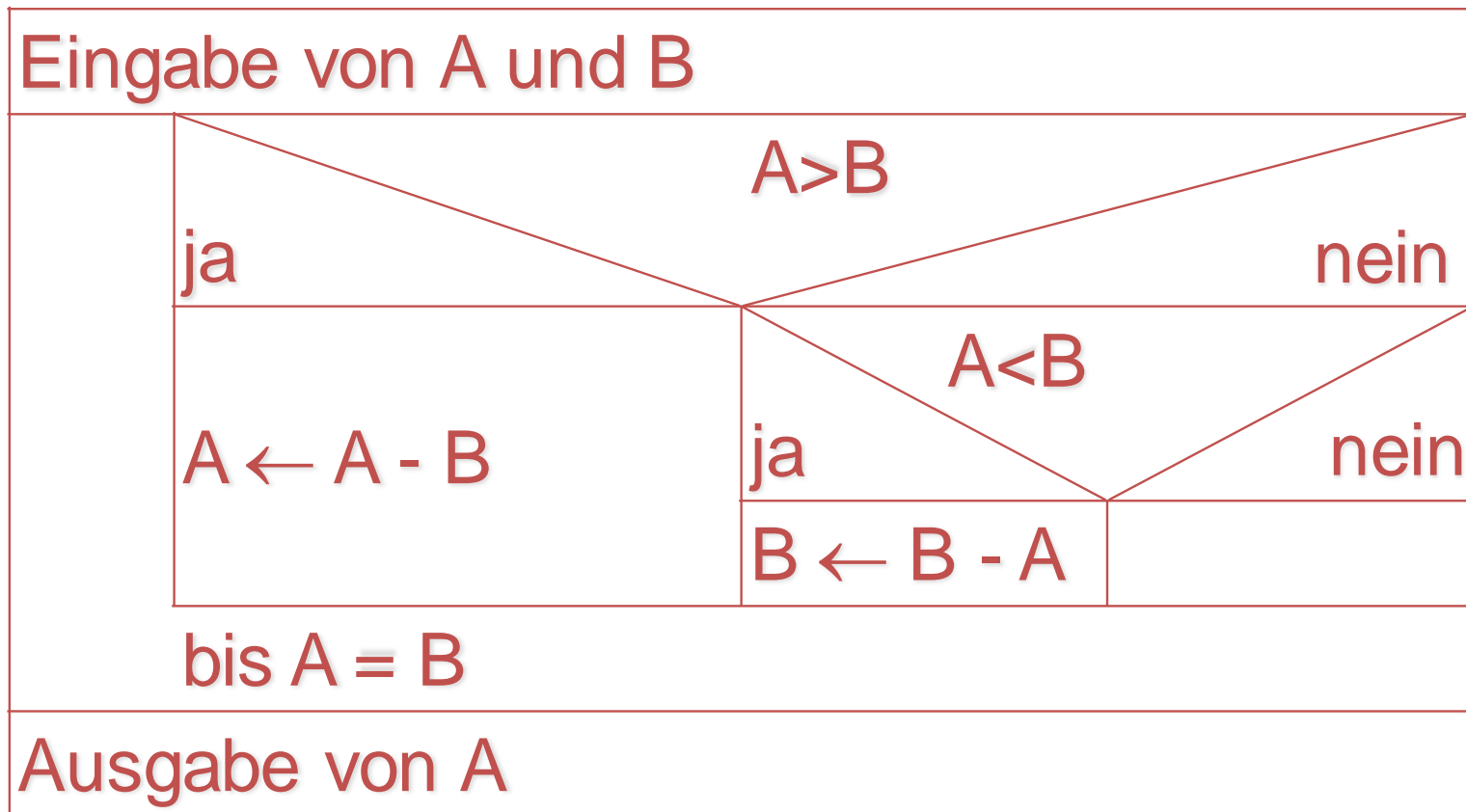
- Fallunterscheidung



- + Schachtelung!



# Nassi-Shneiderman-Diagramm: Euklidischer Algorithmus



# Pseudocode



- Semiformale Sprachen
- Beispiel:

```
WHILE  A ungleich B
    IF A > B
    THEN subtrahiere B von A
    ELSE
        subtrahiere A von B
    ENDIF
ENDWHILE
ggT := A
```

# ESOP - Einfache Programme

Assoc. Prof. Dr. Mathias Lux  
ITEC / AAU

# Agenda



- Grundsymbole
- Variablen, Konstanten
- Zuweisungen
- Operatoren



# Grundsymbole: Namen



Bezeichnen Variablen, Typen, ... in einem Programm

- bestehen aus Buchstaben, Ziffern und „\_“
- beginnen mit Buchstaben
- beliebig lang
- Groß-/Kleinschreibung signifikant
  
- Beispiele
  - x, x17, my\_Var, myVar

# Grundsymbole: Schlüsselwörter



- Heben Programmteile hervor
- Dürfen nicht als Namen verwendet werden
- Beispiele:
  - if, while, for, enum, class, static, ...

# Grundsymbole: Zahlen



- Ganze Zahlen
  - (dezimal oder hexadezimal)
- Gleitkommazahlen
- Beispiele
  - 376 ... dezimal
  - 0x1A5 ... hexadezimal
  - 3.14 ... Gleitkommazahl

# Grundsymbole: Zeichenketten



- Beliebige Zeichenfolgen zwischen Hochkommas
- Dürfen nicht über Zeilengrenzen gehen
- " in der Zeichenkette wird als \" geschrieben
  
- Beispiele
  - "a simple string"
  - "sie sagte \"Hallo\""

# Grundsymbole: Zeichenketten



- String ... Zeichenkette
  - Eigentlich kein Basisdatentyp, sondern ein Objekt!
- char ... ein einzelnes Unicode Zeichen
  - 2 Bytes
  - unter einfachem Hochkomma, z.B. 'L', ')', ...

# Variablendeklaration



- Jede Variable muss vor ihrer Verwendung deklariert werden
  - macht den Namen und den Typ der Variablen bekannt
  - Compiler reserviert Speicherplatz für die Variable
- Beispiele:
  - `int x; ...` deklariert eine Variable `x` vom Typ `int` (integer)
  - `short a, b; ...` deklariert 2 Variablen `a` und `b` vom Typ `short` (short integer)

# Ganzzahlige Typen



<b>byte</b>	8 bit	$-2^7 .. 2^7-1$	(-128 .. 127)
<b>short</b>	16 bit	$-2^{15} .. 2^{15}-1$	(-32.768 .. 32.767)
<b>int</b>	32 bit	$-2^{31} .. 2^{31}-1$	(-2.147.483.648 .. )
<b>long</b>	64 bit	...	...

- **Initialisierungen**

- `int x = 100;`

- deklariert int-Variable x; weist ihr den Anfangswert 100 zu

- `short a = 0, b = 1;`

- deklariert 2 short-Variablen a und b mit Anfangswerten

# Konstantendeklaration



- Initialisierte "Variablen", deren Wert man nicht mehr ändern kann
  - `static final` `int max = 100;`
- Zweck
  - bessere Lesbarkeit
    - `max` ist lesbarer als `100`
  - bessere Wartbarkeit
    - wenn die Konstante mehrmals vorkommt und geändert werden muss, dann muss das nur an 1 Stelle erfolgen
- Konstantendeklaration muss auf Klassenebene stehen (s. später)



# Kommentare



- Zeilenendekommentare
  - Beginnen mit `//` .. reichen bis zum Zeilenende
- Klammerkommentare
  - durch `/* ... */` begrenzt, können über mehrere Zeilen gehen
  
- Kommentare & Lesbarkeit
  - alles kommentieren, was Erklärung bedarf
  - nicht kommentieren, was schon da ist;

```
// Hier ist ein Zeilenkommentar

int x = 15; // Initialisierung an dieser Stelle erforderlich!
short y = -12;

/* *****
   Dieses Programm wurde von Mathias Lux geschrieben
   ***** */
```

# Kommentare



- Vgl. Open-Source-Projekte
  - <https://github.com/protyposis/ITEC-MediaPlayer/blob/master/MediaPlayer/src/main/java/at/aau/itec/android/mediaplayer/AudioPlayback.java>
  - <https://github.com/trifacta/vega/blob/master/src/core/Bounds.js>

# Sprache für Kommentare & Namen



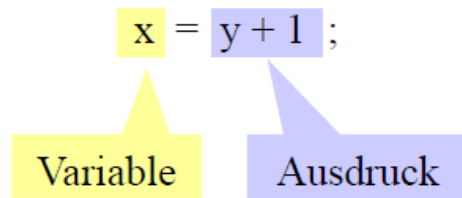
- Am “potentiellen Team” orientieren
  - Englisch besser als Deutsch
- Auf keinen Fall mischen
  
- Achtung bei
  - Schimpfworten, Emailadressen, Namen, Lizenzen!

# Namenswahl für Variablen & Konstanten



- Coding Conventions existieren für
  - Lesbarkeit über Teams hinweg
  - Wartbarkeit & Preservation
- Naming Conventions siehe:  
<http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-135099.html#367>
- Tipps:
  - Bedeutungsvolle Namen (vgl. Kommentare)
  - Eher kurz als lang, aber IDE unterstützt.

# Zuweisungen



1. berechne den Ausdruck
2. speichere seinen Wert in der Variablen

- **Bedingung: linke und rechte Seite müssen zuweisungskompatibel sein**
  - müssen dieselben Typen haben, oder
  - Typ links  $\supseteq$  Typ rechts
- **Hierarchie der ganzzahligen Typen**
  - long  $\supseteq$  int  $\supseteq$  short  $\supseteq$  byte

# Zuweisungen



- Beispiele

```
int i, j; short s; byte b;
```

```
i = j;           // ok: derselbe Typ
```

```
i = 300;        // ok (Zahlkonstanten sind int)
```

```
b = 300;        // falsch: 300 passt nicht in byte
```

```
i = s;         // ok
```

```
s = i;         // falsch
```

# Statische Typenprüfung



- Compiler prüft:
  - dass Variablen nur erlaubte Werte enthalten
  - dass auf Werte nur erlaubte Operationen ausgeführt werden

# Arithmetische Ausdrücke



- Vereinfachte Grammatik

Expr = Operand {BinaryOperator Operand}.

Operand = [UnaryOperator] ( identifier | number | "(" Expr ")" ).

- z.B.:  $-x + 3 * (y + 1)$



# Arithmetische Ausdrücke



## Binäre Operatoren

+	Addition				
-	Subtraktion				
*	Multiplikation				
/	Division, Ergebnis ganzzahlig	$5/3 = 1$	$(-5)/3 = -1$	$5/(-3) = -1$	$(-5)/(-3) = 1$
%	Modulo (Divisionsrest)	$5\%3 = 2$	$(-5)\%3 = -2$	$5\%(-3) = 2$	$(-5)\%(-3) = -2$

## Unäre Operatoren

+	Identität ( $+x = x$ )
-	Vorzeichenumkehr

# Typregeln in arithmetischen Ausdrücken



- Vorrangregeln
  - Punktrechnung ( $*$ ,  $/$ ,  $\%$ ) vor Strichrechnung ( $+$ ,  $-$ )
    - z.B.  $2 + 3 * 4 = 14$
  - Linksassoziativität
    - z.B.  $7 - 3 - 2 = 2$
  - Unäre Operatoren binden stärker als binäre
    - z.B.:  $-2 * 4 + 3$  ergibt  $-5$
- Typregeln
  - Operandentypen - byte, short, int, long
  - Ergebnistyp - wenn mindestens 1 Operand long  $\rightarrow$  long, sonst  $\rightarrow$  int

# Beispiele

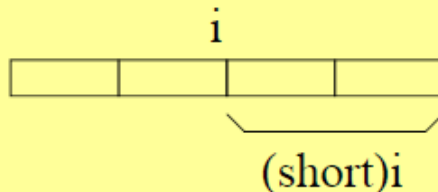


```
short s; int i; long x;  
x = x + i;           // long  
i = s + 1;          // int (1 ist vom Typ int)  
s = s + 1;          // falsch!  
s = (short)(s + 1); // Typumwandlung nötig
```

## Typumwandlung (type cast)

*(type)expression*

- wandelt Typ von *expression* in *type* um
- dabei kann etwas abgeschnitten werden



# Increment / Decrement



- Variablenzugriff kombiniert mit Addition/Subtraktion
  - `x++` ... nimmt den Wert von `x` und erhöht `x` anschließend um 1
  - `++x` ... erhöht `x` um 1 und nimmt anschließend den erhöhten Wert
  - `x--`, `--x` ... entsprechend
- Kann auch als eigenständige Anweisung verwendet werden
  - `x = 1; x++;` // `x = 2` entspricht: `x = x + 1;`
- Beispiele
  - `x = 1; y = x++ * 3;` // `x = 2, y = 3` entspricht: `y = x * 3; x = x + 1;`
  - `x = 1; y = ++x * 3;` // `x = 2, y = 6` entspricht: `x = x + 1; y = x * 3;`
- Darf nur auf Variablen angewendet werden (nicht auf Ausdrücke)
  - `y = (x + 1)++;` // falsch!

# Multiplikation/Division mit Zweierpotenzen



Mit Shift-Operationen effizient implementierbar

## *Multiplikation*

$x * 2$	$x \ll 1$
$x * 4$	$x \ll 2$
$x * 8$	$x \ll 3$
$x * 16$	$x \ll 4$
...	...

## *Division*

$x / 2$	$x \gg 1$
$x / 4$	$x \gg 2$
$x / 8$	$x \gg 3$
$x / 16$	$x \gg 4$
...	...

Division nur bei positiven Zahlen durch Shift ersetzbar

# Multiplikation/Division mit Zweierpotenzen



## Beispiele

$x = 3;$

0000 0011

$x = x \ll 2; // 12$

0000 1100

$x = -3;$

1111 1101

$x = x \ll 1; // -6$

1111 1010

$x = 15;$

0000 1111

$x = x \gg 2; // 3$

0000 0011

# Zuweisungsoperatoren



- Arithmetischen Operationen lassen sich mit Zuweisung kombinieren

	<i>Kurzform</i>	<i>Langform</i>
<code>+=</code>	<code>x += y;</code>	<code>x = x + y;</code>
<code>-=</code>	<code>x -= y;</code>	<code>x = x - y;</code>
<code>*=</code>	<code>x *= y;</code>	<code>x = x * y;</code>
<code>/=</code>	<code>x /= y;</code>	<code>x = x / y;</code>
<code>%=</code>	<code>x %= y;</code>	<code>x = x % y;</code>

# String-Operatoren



- Strings können mit ‘+’ verknüpft werden
  - “Mathias” + “ “ + “Lux”
- Andere Operatoren gelten nicht für Strings
  - Vor allem nicht Prüfung auf Gleichheit
  - “Mathias” != “Lux” ... vergleicht Adressen!



# Bit-Operatoren



- Die Bits der Operanden werden miteinander verknüpft.
  - Beispiel (Java nutzt Zweierkomplement)
    - `byte a = 17; // 00010001`
    - `byte b = 7; // 00000111`
- Eine Eins steht im Ergebnis genau dort, wo...
  - Disjunktion: ... einer der Operanden eine Eins aufweist
    - `byte or = a | b; // 23`
  - Konjunktion: ... beide Operanden eine Eins aufweisen
    - `byte and = a & b; // 1`
  - Antivalenz: ... die Operanden unterschiedlich sind
    - `byte xor = a ^ b; // 22`
  - Komplement: ... der Operand eine Null aufweist
    - `byte notB = ~b; // -8`

# Grundstruktur von Java-Programmen



```
class ProgramName {  
    public static void main (String[] arg) {  
        ... // Deklarationen  
        ... // Anweisungen  
    }  
}
```

Text muss in einer Datei  
namens  
*ProgramName.java*  
stehen

// Beispiel:

```
class Sample {  
    public static void main (String[] arg) {  
        int a = 23;  
        int b = 100;  
        System.out.print("Summe = ");  
        System.out.println(a + b);  
    }  
}
```

```
C:\Windows\system32\cmd.exe  
E:\Temp>javac Sample.java  
E:\Temp>java Sample  
Summe = 123  
E:\Temp>
```

# Übersetzen und Ausführen mit JDK



- Übersetzen

- C:\> cd MySamples

- wechselt zu Quelldatei

- C:\MySamples> javac Sample.java

- erzeugt Datei Sample.class

- Ausführen

- C:\MySamples> java Sample

- ruft main-Methode der Klasse Sample auf

- Summe = 123

# Beispiel: IDEA IDE



- Beispiele für Kommentare
  - Rechtschreibprüfung
- Live Templates
  - psvm + <tab>
- Automatische Benennung von Variablen
  - <Strg>-<Leertaste>

# ESOP - Verzweigungen & Schleifen

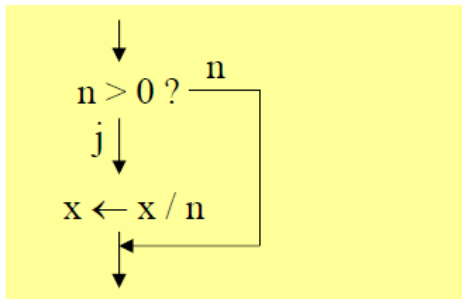
Assoc. Prof. Dr. Mathias Lux  
ITEC / AAU

# Agenda



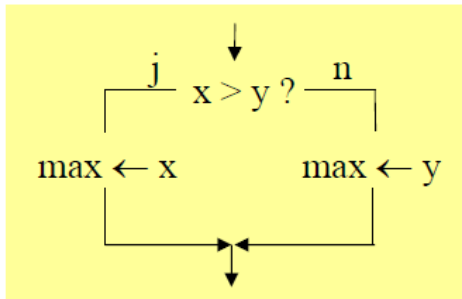
- Verzweigungen
  - If – Else, Switch
- Schleifen
  - While, Do-While, For

# If-Anweisung



```
if (n > 0) x = x / n;
```

ohne else-Zweig



```
if (x > y)  
    max = x;  
else  
    max = y;
```

mit else-Zweig

## Syntax

```
IfStatement = "if" "(" Expression ")" Statement ["else" Statement].
```

# Anweisungsblöcke



Wenn if-Zweig oder else-Zweig aus mehr als 1 Anweisung bestehen, müssen sie durch { ... } geklammert werden.

Statement = Assignment | IfStatement | Block | ... .  
Block = "{ **Statement** }".



# Anweisungsblöcke



- Beispiel

```
if (x < 0) {  
    <negNumbers++;  
    System.out.print(-x);  
} else {  
    posNumbers++;  
    System.out.print(x);  
}
```

Einrückung

Best Practice:  
{...} auch bei einzelnen  
Statements

# Einrückungen



- Erhöhen die Lesbarkeit
  - machen Programmstruktur besser sichtbar
- Einrückungstiefe
  - 1 Tabulator oder 2 Leerzeichen
- Kurze If-Anw. auch in einer Zeile:
  - `if (n != 0) x = x / n;`
  - `if (x > y) max = x; else max = y;`

# Dangling Else



```
if (a > b)
  if (a != 0) max = a;
else
  max = b;
```

```
if (a > b)
  if (a != 0) max = a; else max = b;
```

- Zu welchem if gehört das else?
- In Java: else gehört immer zum unmittelbar vorausgegangenen if
- Alternative: Anweisungsblöcke verwenden!

# Kurze If-Anweisungen



- `(Expression) ? Statement : Statement`

```
int x = 3;  
int y = 4;  
int max = (x < y) ? y : x;
```

```
println(max);
```

# Vergleichsoperatoren



- Vergleich zweier Werte
- Liefert *true* oder *false*

	<i>Bedeutung</i>	<i>Beispiel</i>
<code>==</code>	gleich	<code>x == 3</code>
<code>!=</code>	ungleich	<code>x != y</code>
<code>&gt;</code>	größer	<code>4 &gt; 3</code>
<code>&lt;</code>	kleiner	<code>x+1 &lt; 0</code>
<code>&gt;=</code>	größer oder gleich	<code>x &gt;= y</code>
<code>&lt;=</code>	kleiner oder gleich	<code>x &lt;= y</code>

# Zusammengesetzte Vergleiche

## Boolesche Operatoren



**&&** Und-Verknüpfung

x	y	x && y
true	true	true
true	false	false
false	true	false
false	false	false

**||** Oder-Verknüpfung

x	y	x    y
true	true	true
true	false	true
false	true	true
false	false	false

**!** Nicht-Verknüpfung

x	!x
true	false
false	true

- **Beispiel**

– `if (a >= 0 && a <= 10 || a >= 100 && a <= 110) b = a;`

# Boolesche Operatoren

## Bindung



- ! bindet stärker als && bzw ||
- && bindet stärker als ||
- Klammerung möglich
  - `if (a > 0 && (b==1 || b==7)) ...`

# Datentyp boolean



- Datentyp (wie z.B. int)
  - mit den beiden Werten *true* und *false*
- Beispiel

```
boolean p, q;  
p = false;  
q = x > 0;  
p = p || q && x < 10;
```



# DeMorgan'sche Regeln



- $!(a \ \&\& \ b) \Leftrightarrow !a \ || \ !b$
- $!(a \ || \ b) \Leftrightarrow !a \ \&\& \ !b$

```
if (x >= 0 && x < 10) {
```

```
  ...
```

```
} else { // !(x >= 0 && x < 10)
```

```
  ...
```

```
}
```

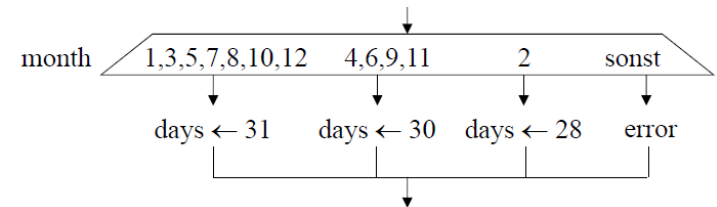
⇒  $!(x \geq 0) \ || \ !(x < 10)$

⇒  $x < 0 \ || \ x \geq 10$

# Switch-Anweisung



- Mehrfachverzweigung
- In Java



```
switch (month) {  
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:  
        days = 31; break;  
    case 4: case 6: case 9: case 11:  
        days= 30; break;  
    case 2:  
        days = 28; break;  
    default:  
        System.out.println("error");  
}
```

# Switch-Anweisung



- **Bedingungen**

- Ausdruck ganzzahlig, char oder String
- Case-Marken sind Konstante
- Case-Marken Typ muss zu Ausdruck passen
- Case-Marken müssen verschieden sein

- **Break-Anweisung**

- Spring ans Ende der Switch-Anweisung
- Fehlt break, wird alles danach ausgeführt  
=> häufiger Fehler!

Switch-  
Ausdruck

```
switch (month) {  
  case 1: case 3: case 5: case 7: case 8: case 10: case 12:  
    days = 31; break;  
  case 4: case 6: case 9: case 11:  
    days = 30; break;  
  case 2:  
    days = 28; break;  
  default:  
    System.out.println("error");  
}
```

# Switch-Syntax

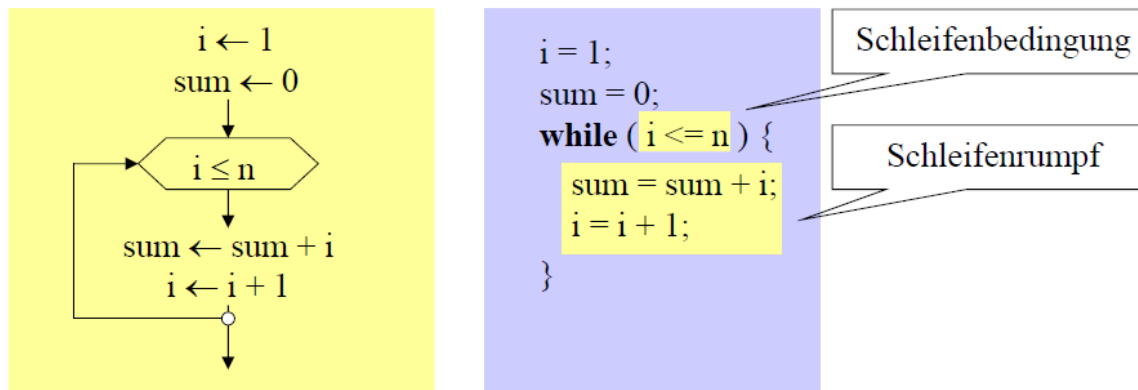


```
Statement = Assignment | IfStatement | SwitchStatement | ... | Block.  
SwitchStatement = "switch" "(" Expression ")" "{" {LabelSeq StatementSeq} }".  
LabelSeq = Label {Label}.  
StatementSeq = Statement {Statement}.  
Label = "case" ConstantExpression ":" | "default" ":".
```

# While-Schleife



- Führt eine Anweisungsfolge aus
- Solange eine bestimmte Bedingung gilt



Statement = Assignment | IfStatement | SwitchStatement | WhileStatement | ... | Block.  
WhileStatement = **"while" "(" Expression ")" Statement .**

# While-Schleife



```
class Pyramid {  
    public static void main (String[] arg) {  
        int i = 10;  
        while (i-->0) {  
            int j = 0;  
            while (j++<i) {  
                System.out.print("*");  
            }  
            System.out.println();  
        }  
    }  
}
```

```
C:\Windows\system32\cmd.exe  
E:\Temp>javac Pyramid.java  
E:\Temp>java Pyramid  
*****  
*****  
*****  
*****  
*****  
****  
***  
**  
*  
E:\Temp>
```

# Termination

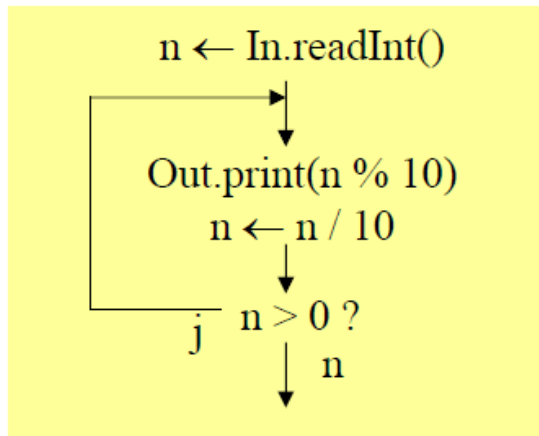


- Schleifen sollten ein Ende haben
  - kein `while (true) { ... }`
- Problem: Endlosschleifen
  - Abgefragte Variable wird nicht verändert
  - Abbruchbedingung wird nicht erreicht
    - z.B. `while (x!=0) { x -= 5; }`
- Lösung: Modellierung möglicher Probleme

# Do-While-Schleife



- Abbruchbedingung wird am Ende der Schleife geprüft
- Schleife wird mind. 1x durchlaufen



```
int n = In.readInt();  
do {  
    Out.print(n % 10);  
    n = n / 10;  
} while ( n > 0 );
```

*Schreibtischtest*

n	n % 10
<del>123</del>	3
<del>12</del>	2
<del>1</del>	1
0	

Statement = Assignment | IfStatement | WhileStatement |  
DoWhileStatement | ... | Block.

DoWhileStatement = **"do" Statement "while" "(" Expression ")" ";"**.



# For-Schleife (Zählschleife)



- Falls Anzahl im Vorhinein bekannt ist

```
sum = 0;  
for (i = 1 ; i <= n ; i++)  
    sum = sum + i;
```

- 1) Initialisierung der Laufvariablen
- 2) Schleifenabbruchbedingung
- 3) Ändern der Laufvariablen

Kurzform für

```
sum = 0;  
i = 1;  
while (i <= n) {  
    sum = sum + i;  
    i++;  
}
```

# For-Schleife: Beispiele



<code>for (i = 0; i &lt; n; i++)</code>	<code>i: 0, 1, 2, 3, ..., n-1</code>
<code>for (i = 10; i &gt; 0; i--)</code>	<code>i: 10, 9, 8, 7, ..., 1</code>
<code>for (int i = 0; i &lt;= n; i = i + 1)</code>	<code>i: 0, 1, 2, 3, ..., n</code>
<code>for (int i = 0, j = 0; i &lt; n &amp;&amp; j &lt; m; i = i + 1, j = j + 2)</code>	<code>i: 0, 1, 2, 3, ...</code> <code>j: 0, 2, 4, 6, ...</code>
<code>for (;;) ...</code>	Endlosschleife

# For-Schleife: Definition



`ForStatement = "for" "(" [ForInit] ";" [Expression] ";"  
[ForUpdate] ")" Statement.`

`ForInit = Assignment {",", " Assignment} | Type VarDecl {",",  
VarDecl}.`

`ForUpdate = Assignment {",", " Assignment}.`

# For-Schleife: Beispiel



```
class PrintMulTab {  
    public static void main (String[] arg) {  
        int n = 5;  
        for (int i = 1; i <= n; i++) {  
            for (int j = 1; j <= n; j++) {  
                System.out.print(i * j + "\t");  
            }  
            System.out.println();  
        }  
    }  
}
```

```
C:\Windows\system32\cmd.exe  
E:\Temp>javac PrintMulTab.java  
E:\Temp>java PrintMulTab  
1      2      3      4      5  
2      4      6      8      10  
3      6      9      12     15  
4      8      12     16     20  
5      10     15     20     25  
E:\Temp>_
```

# Schleifenabbrüche



- Abbruch mit Keyword *break*

```
while (In.done()) {
    sum = sum + x;
    if (sum > 1000) {
        Out.println("zu gross");
        break;
    }
    x = In.nextNumber();
}
```

- Besser als Schleifenbedingung ...

```
while (In.done() && sum < 1000) {
    sum = sum + x;
    x = In.nextNumber();
}
if (sum > 1000)
    Out.println("zu gross");
```

# Abbruch äußerer Schleifen



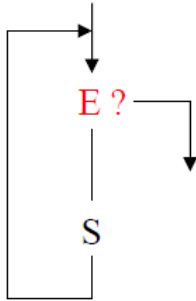
```
outer: // Marke!  
for (;;) { // Endlosschleife!  
    for (;;) {  
        ...  
        if (...) break; // verlässt innere Schleife  
        else break outer; // verlässt äußere Schleife  
        ...  
    }  
}
```

# Schleifenabbrüche



- Wann ist ein Schleifenabbruch mit `break` typischerweise vertretbar?
  - bei Abbruch wegen Fehlern (Performance!)
  - bei mehreren Ausprägungen an verschiedenen Stellen der Schleife
  - bei echten Endlosschleifen (z.B. in Echtzeitsystemen)

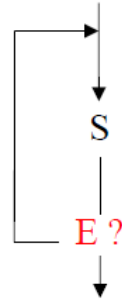
# Vergleich der Schleifenarten



**Abweisschleife**

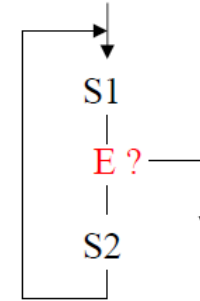
```
while (E)  
  S
```

```
for (I; E; U)  
  S
```



**Durchlaufschleife**

```
do  
  S  
while (E)
```



**allgemeine Schleife**

```
for (;;) {  
  S1;  
  if (E) break;  
  S2;  
}
```



# Welche Schleife Wann?



- Auswahl nach “Convenience”
- Auswahl nach Performance
  - (s.u. für Javascript, <http://jsperf.com/fun-with-for-loops/8>)

## Test runner

Done. Ready to run again.

Run again

Testing in Chrome 37.0.2062.124 32-bit on Windows Server 2008 R2 / 7 64-bit

	Test	Ops/sec
FOR standard	<pre>for (var i; i &lt; a.length; i++) {   n++; }</pre>	329,591,795 ±0.23% fastest
FOR optimized	<pre>for (var i, imax = a.length; i &lt; imax; i++) {   n++; }</pre>	329,708,498 ±0.43% 0.16% slower
While Counting Down	<pre>var i = a.length + 1; while(--i) {   n++; }</pre>	29,620,863 ±19.14% 92% slower

# ESOP - Gleitkommazahlen, Methoden und Arrays

Assoc. Prof. Dr. Mathias Lux  
ITEC / AAU

# Wiederholung ..



```
/**
 * Check for primes, simple version ...
 */
public class Primes {
    public static void main(String[] args) {
        int maxPrime = 1000;
        // iterate candidates
        for (int candidate = 3; candidate <= maxPrime; candidate++) {
            boolean isPrime = true;
            // iterate potential dividers
            for (int divider = 2; divider < candidate; divider++) {
                // check for division without rest
                if (candidate % divider == 0) {
                    isPrime = false;
                }
            }
            if (isPrime)
                System.out.println("prime = " + candidate);
        }
    }
}
```

- Finde Primzahlen < maxPrime

# Gleitkommazahlen



- Zwei Datentypen
  - float ... 32 Bit Genauigkeit (24/8 in Java 8)
  - double ... 64 bit Genauigkeit (53/11 in Java 8)

- Syntax

FloatConstant = [Digits] "." [Digits] [Exponent]  
[FloatSuffix].

Digits = Digit {Digit}.

Exponent = ("e" | "E") ["+" | "-"] Digits.

FloatSuffix = "f" | "F" | "d" | "D".

# Gleitkommazahlen



- **Variablen**
  - float x, y;
  - double z;
- **Konstanten**
  - 3.14 // Typ double
  - 3.14f // Typ float
  - 3.14E0 //  $3.14 * 10^0$
  - 0.314E1 //  $0.314 * 10^1$
  - 31.4E-1 //  $31.4 * 10^{-1}$
  - .23
  - 1.E2 // 100

# Harmonische Reihe



```
public class HarmonicSequence {
    public static void main (String[] arg) {
        float sum = 0;
        int n = 10;
        for (int i = n; i > 0; i--)
            sum += 1.0f / i;
        System.out.println("sum = " + sum);
    }
}
```

- Was würden statt  $1.0f / i$  folgende Ausdrücke liefern?
  - $1 / i$  ... 0 (weil ganzzahlige Division)
  - $1.0 / i$  ... einen double-Wert

# Float vs. Double



```
public class HarmonicSequence {
    public static void main (String[] arg) {
        float sum = 0;
        int n = 10;
        for (int i = n; i > 0; i--)
            sum += 1.0f / i;
        System.out.println("sum = " + sum);
    }
}
```

D:\Java\JDK\jdk1.6.0\_45\bin\java ...  
sum = 2.9289684

Process finished with exit code 0

```
public class HarmonicSequence {
    public static void main (String[] arg) {
        double sum = 0;
        int n = 10;
        for (int i = n; i > 0; i--)
            sum += 1.0d / i;
        System.out.println("sum = " + sum);
    }
}
```

D:\Java\JDK\jdk1.6.0\_45\bin\java ...  
sum = 2.9289682539682538

Process finished with exit code 0

# Zuweisungen und Operationen



- Zuweisungskompatibilität
  - `double`  $\supseteq$  `float`  $\supseteq$  `long`  $\supseteq$  `int`  $\supseteq$  `short`  $\supseteq$  `byte`
- Erlaubte Operationen
  - Arithmetische Operationen (+, -, \*, /)
  - Vergleiche (==, !=, <, <=, >, >=)  
Achtung! Gleitkommazahlen nicht auf Gleichheit prüfen!



# Zuweisungen und Casts



```
float f; int i;
f = i;      // erlaubt
i = f;      // verboten
i = (int) f; // erlaubt: schneidet Nachkommastellen ab;
             // falls zu groß oder zu klein:
             // Integer.MAX_VALUE, Integer.MIN_VALUE
f = 1.0;    // verboten, weil 1.0 vom Typ double ist
```

# Methoden

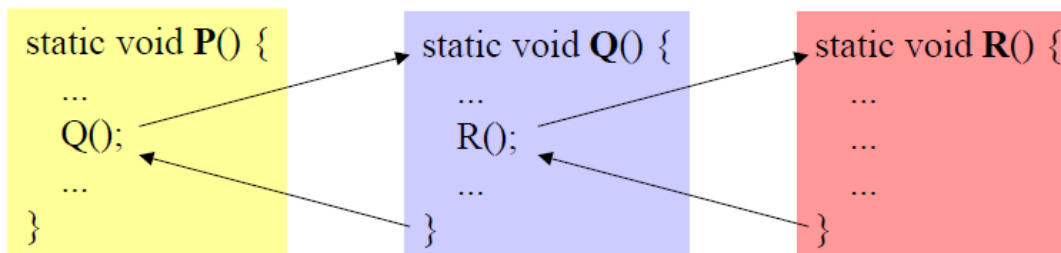


- Vgl. funktionalen Sprachen
  - Unterprogramme, Funktionen, ...
- Ziel ist Code wiederzuverwenden
  - Oft genutzte Funktionen / Operationen
- Weniger Zeilen Code
  - weniger Arbeit, weniger Fehler
  - leichter zu warten

# Methoden in Java



- Wir betrachten erst Spezialfall von Methoden
  - .. und nutzen sie als Unterprogramme
- Namenskonventionen für Methoden
  - Beginnen mit Verb und Kleinbuchstaben
  - Beispiele:
    - printHeader, findMaximum, traverseList, ...



# Methoden in Java



```
public class SubroutineExample {
    private static void printRule() {           // Methodenkopf
        System.out.println("-----");       // Methodenrumpf
    }

    public static void main(String[] args) {
        printRule();                           // Aufruf
        System.out.println("Header 1");
        printRule();
    }
}
```

D:\Java\JDK\jdk1.6.0\_45\bin\java ...

```
-----
Header 1
-----
```

Process finished with exit code 0

# Parameter



- Es können Werte an die Methode übergeben werden.

```
class Sample {  
  
    static void printMax (int x, int y) {  
        if (x > y) Out.print(x); else Out.print(y);  
    }  
  
    public static void main (String[] arg) {  
        ...  
        printMax(100, 2 * i);  
    }  
}
```

## formale Parameter

- im Methodenkopf (hier x, y)
- sind Variablen der Methode

## aktuelle Parameter

- an der Aufrufstelle (hier 100, 2\*i)
- können Ausdrücke sein

# Parameter



- Aktuelle Parameter werden den entsprechenden formalen Parametern zugewiesen
- $x = 100; y = 2 * i;$ 
  - aktuelle Parameter müssen mit formalen zuweisungskompatibel sein
  - formale Parameter enthalten Kopien der aktuellen Parameter

```
class Sample {  
    static void printMax (int x, int y) {  
        if (x > y) Out.print(x); else Out.print(y);  
    }  
  
    public static void main (String[] arg) {  
        ...  
        printMax(100, 2 * i);  
    }  
}
```

# Funktionen



- Funktionen sind Methoden, die einen Ergebniswert an den Aufrufenden zurückliefern

```
class Sample {  
    static int max (int x, int y) {  
        if (x > y) return x; else return y;  
    }  
  
    public static void main (String[] arg) {  
        ...  
        int result = 3 * max(100, i + j) + 1;  
        ...  
    }  
}
```

- haben Funktionsstyp (z.B. *int*) statt *void* (= kein Typ)
- liefern Ergebnis mittels *return*-Anweisung an den Rufer zurück (*x* muss zuweisungskompatibel mit *int* sein)
- Werden wie Operanden in einem Ausdruck benutzt

# Funktionen vs. Prozeduren



- Funktionen
  - Methoden mit Rückgabewert
  - static `int` max (int x, int y) {...}
- Prozeduren
  - Methoden ohne Rückgabewert
  - static `void` printMax (int x, int y) {...}



# Beispiel



```
public class BinomialCoefficient {
    public static void main(String[] args) {
        int n = 5, k = 3;
        int result = factorial(n) /
            (factorial(k) * factorial(n - k));
        System.out.println("result = " + result);
    }

    public static int factorial(int k) {
        int result = 1;
        for (int i = 2; i <= k; i++) {
            result *= i;
        }
        return result;
    }
}
```

$$\binom{n}{k} = \frac{n!}{k! \cdot (n - k)!}$$

# Return & Rekursion



```
public class BinomialCoefficient {
    static int n = 5, k = 3;

    public static void main(String[] args) {
        int result = factorial(n) /
            (factorial(k) * factorial(n - k));
        System.out.println("result = " + result);
    }

    public static int factorial(int k) {
        if (k>1) {
            return factorial(k-1)*k;
        }
        else {
            return 1;
        }
    }
}
```

- Return leitet Rücksprung ein
- Kann an beliebiger Stelle in der Methode stehen
- Methode, die sich selbst aufruft -> Rekursion

# Primzahlen



```
/**
 * Primes based on function.
 */
public class PrimesWithMethod {
    public static void main(String[] args) {
        int maxPrime = 1000;
        // iterate candidates
        for (int candidate = 3; candidate <= maxPrime;
candidate++) {
            if (isPrime(candidate))
                System.out.println("prime = " + candidate);
        }
    }
}
```

```
public static boolean isPrime(int candidate) {
    boolean isPrime = true;
    // iterate potential dividers
    for (int divider = 2; divider < candidate; divider++) {
        // check for division without rest
        if (candidate % divider == 0) {
            isPrime = false;
        }
    }
    return isPrime;
}
}
```

```
/**
 * Check for primes, simple version ...
 */
public class Primes {
    public static void main(String[] args) {
        int maxPrime = 1000;
        // iterate candidates
        for (int candidate = 3; candidate <= maxPrime;
candidate++) {
            boolean isPrime = true;
            // iterate potential dividers
            for (int divider = 2; divider < candidate; divider++) {
                // check for division without rest
                if (candidate % divider == 0) {
                    isPrime = false;
                }
            }
            if (isPrime)
                System.out.println("prime = " + candidate);
        }
    }
}
```

# Gültigkeitsbereiche (Scope) von Variablen



- Innerhalb eines Blocks gültig
  - { ... },
  - for (int i; ...) {...}
- Außerhalb ist Variable nicht bekannt!

# Beispiel



```
public class BinomialCoefficient {
    public static void main(String[] args) {
        int n = 5, k = 3;
        int result = factorial(n) /
            (factorial(k) * factorial(n - k));
        System.out.println("result = " + result);
    }

    public static int factorial(int k) {
        int result = 1;
        for (int i = 2; i <= k; i++) {
            result *= i;
        }
        return result;
    }
}
```

Unterschiedliche  
Variablen

# Beispiel: Scope



```
public class BinomialCoefficient {
    static int n = 5, k = 3;

    public static void main(String[] args) {
        int result = factorial(n) /
            (factorial(k) * factorial(n - k));
        System.out.println("result = " + result);
    }

    public static int factorial(int k) {
        int result = 1;
        for (int i = 2; i <= k; i++) {
            result *= i;
        }
        return result;
    }
}
```

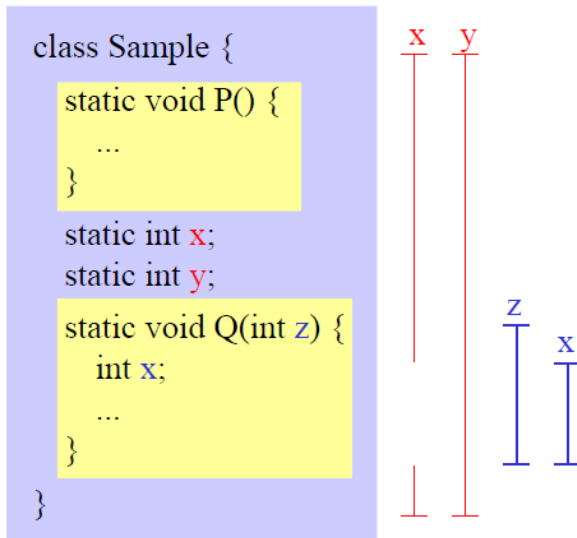
Kleinsten  
Gültigkeitsbereich  
entscheidend!

# Sichtbarkeitsbereich von Namen



## Regeln

1. Ein Name darf in einem Block nicht mehrmals deklariert werden (auch nicht in geschachtelten Anweisungsblöcken).
2. Lokale Namen verdecken Namen, die auf Klassenebene deklariert sind.
3. Der Sichtbarkeitsbereich eines lokalen Namens beginnt bei seiner Deklaration und geht bis zum Ende der Methode.
4. Auf Klassenebene deklarierte Namen sind in allen Methoden der Klasse sichtbar.



# Lokale & statische Variablen

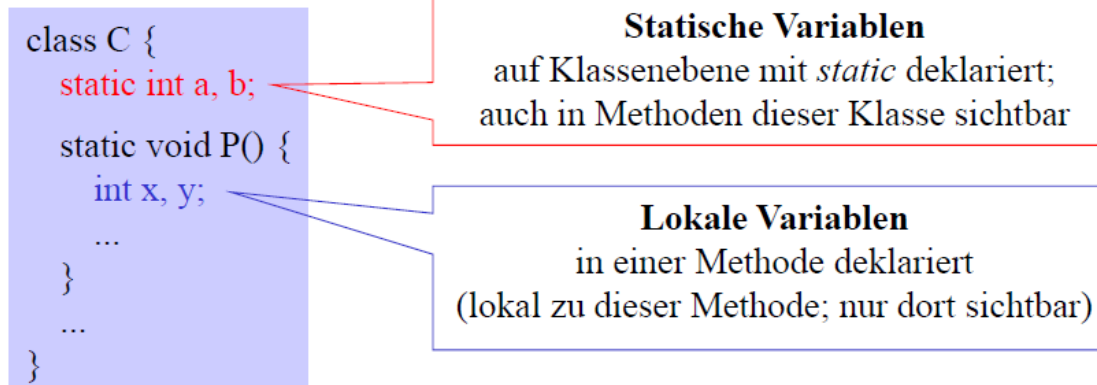


## Statische Variablen

- Am Programmbeginn angelegt
- Zu Programmende wieder freigegeben

## Lokale Variablen

- Bei jedem Aufruf der Methode angelegt
- Am Ende der Methode wieder freigegeben





# Lokalität



Best Practice: Variablen möglichst lokal deklarieren, nicht als statische Variablen.

## Vorteile:

- Übersichtlichkeit: Deklaration und Benutzung nahe beisammen
- Sicherheit: Lokale Variablen können nicht durch andere Methoden zerstört werden
- Effizienz: Zugriff auf lokale Variable ist oft schneller als auf statische Variable

# Überladen von Methoden



- Methoden mit gleichem Namen aber verschiedenen Parameterlisten können in derselben Klasse deklariert werden

```
static void write (int i) {...}  
static void write (float f) {...}  
static void write (int i, int width) {...}
```

- Beim Aufruf wird diejenige Methode gewählt, die am besten zu den aktuellen Parametern passt

```
write(100);      ⇒ write (int i)  
write(3.14f);   ⇒ write (float f)  
write(100, 5);  ⇒ write (int i, int width)  
short s = 17;  
write(s);       ⇒ write (int i);
```

# Varargs



- In Java können Methoden mit beliebiger Anzahl an Parametern definiert werden.

```
public class VarargExample {  
    public static void main(String[] args) {  
        printList("one", "two", "three");  
    }  
  
    public static void printList(String... list) {  
        System.out.println("list[0] = " + list[0]);  
        System.out.println("list[1] = " + list[1]);  
        System.out.println("list[2] = " + list[2]);  
    }  
}
```

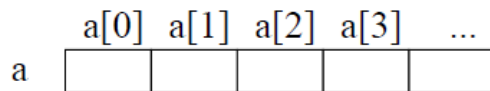


# Arrays



- Zusammenfassung Daten gleichen Typs
- Arrays haben fixe Länge
  - Bei Erzeugung festgelegt
- Array Variablen sind Referenz-Variablen
  - In Java! Vgl. int, float, etc. -> Basistypen
- Zugriff erfolgt über Index
  - Erstes Element hat Indexzahl 0.

# Eindimensionale Arrays



- Name *a* bezeichnet das gesamte Array
- Elemente werden über Indizes angesprochen (z.B. *a[3]*)
- Indizierung beginnt bei 0
- Elemente sind "namenlose" Variablen

## Deklaration

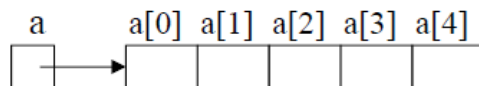
```
int[] a;  
float[] b;
```

- deklariert ein Array namens *a* (bzw. *b*)
- seine Elemente sind vom Typ *int* (bzw. *float*)
- seine Länge ist noch unbekannt

## Erzeugung

```
a = new int[5];  
b = new float[10];
```

- legt ein neues *int*-Array mit 5 Elementen an (aus dem Heap-Speicher)
- weist seine Adresse *a* zu



Array-Variablen enthalten in Java Zeiger auf Arrays!  
(Zeiger = Speicheradresse)

# Zugriff auf Arrays



- Arrayelemente werden wie Variablen benutzt
- Index kann ein ganzzahliger Ausdruck sein
- Laufzeitfehler, falls Array noch nicht erzeugt wurde
- Laufzeitfehler, falls Index  $< 0$  oder  $\geq$  Arraylänge
- *length* ist ein Standardoperator
- Liefert Anzahl der Elemente

```
a[3] = 0;  
a[2*i+1] = a[i] * 3;
```

```
int len = a.length;
```

# Beispiel



```
public class ArrayExample {
    public static void main(String[] args) {
        int[] myArray = new int[5];
        // initialisiere Werte in Array: {1, 2, 3, 4, 5}
        for (int i = 0; i < myArray.length; i++) {
            myArray[i] = i+1;
        }
        // Berechne Durchschnitt:
        float sum = 0;
        for (int i = 0; i < myArray.length; i++) {
            sum += myArray[i];
        }
        System.out.println(sum/myArray.length);
    }
}
```

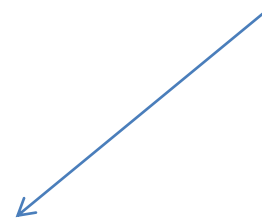
- Berechnet Durchschnitt
- Beachte impliziten Cast auf float!

# Beispiel: While, For Each



```
public class ArrayExample {
    public static void main(String[] args) {
        int[] myArray = new int[5];
        // initialisiere Werte in Array: {1, 2, 3, 4, 5}
        int i = 0;
        while (i < myArray.length) { // while
            myArray[i] = i+1;
            i++;
        }
        // Berechne Durchschnitt:
        float sum = 0;
        for (int myInt : myArray) { // for each
            sum += myInt;
        }
        System.out.println(sum/myArray.length);
    }
}
```

- Andere Schleifen
- Beachte „for each“





# Beispiel: Initialisierung



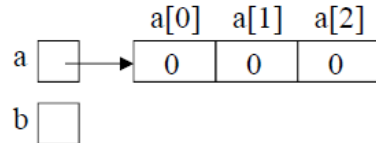
```
public class ArrayExample {
    public static void main(String[] args) {
        // initialisiere Werte in Array: {1, 2, 3, 4, 5}
        int[] myArray = {1, 2, 3, 4, 5};
        // Berechne Durchschnitt:
        float sum = 0;
        for (int myInt : myArray) { // for each
            sum += myInt;
        }
        System.out.println(sum/myArray.length);
    }
}
```

- Andere Initialisierung!

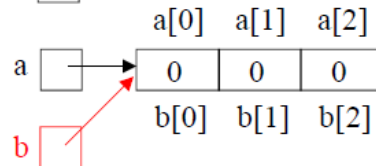
# Arrayzuweisung



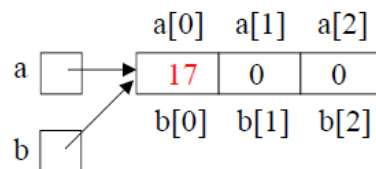
```
int[] a, b;  
a = new int[3];
```



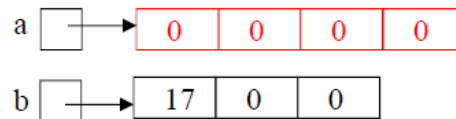
```
b = a;
```



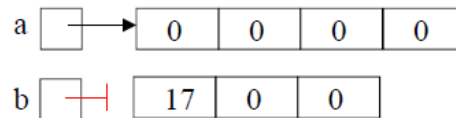
```
a[0] = 17;
```



```
a = new int[4];
```



```
b = null;
```



Arrayelemente werden in Java standardmäßig mit 0 initialisiert

*b* bekommt denselben Wert wie *a*.  
Arrayzuweisung ist in Java **Zeigerzuweisung!**

ändert in diesem Fall auch *b*[0]

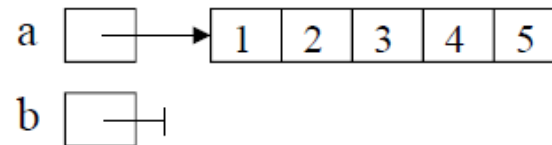
*a* zeigt jetzt auf neues Array.

*null*: Spezialwert, der auf kein Objekt zeigt;  
kann jeder Arrayvariablen zugewiesen werden

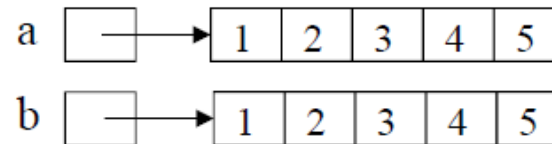
# Kopieren von Arrays



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



```
b = (int[]) a.clone();
```



- Typumwandlung nötig, da `a.clone()` Typ `Object[]` liefert

# Kommandozeilenparameter



- Programmaufruf mit Parametern
  - `java Programmname par1 par2 par3 ...`
- Parameter als String-Array
  - `main(String[] args)` Methode des Programms

# Kommandozeilenparameter



```
public class ArrayExample {  
    public static void main(String[] args) {  
        for (int i = 0; i < args.length; i++) {  
            String arg = args[i];  
            System.out.println(arg);  
        }  
    }  
}
```

```
$> java ArrayExample one two three  
one  
two  
three
```

# Beispiel: Sequentielle Suche



```
public class ArrayExample {
    public static void main(String[] args) {
        int[] myArray = {12, 2, 32, 74, 26, 42, 53, 22};
        int query = 22;
        for (int i = 0; i < myArray.length; i++) {
            if (query == myArray[i]) {
                System.out.println("Found at position " + i);
            }
        }
    }
}
```

- Jedes Element wird untersucht  
-> sequentiell
- Braucht  $n$  Schritte - Wie groß ist  $n$ ?

# Beispiel: Sortierung



- Wie sortiert man ein Array  $a$ ?
- Einfacher Ansatz:
  1. Erzeuge ein gleich großes Array  $b$
  2. Verschiebe Minimum von  $a$  nach  $b$
  3. Falls  $a$  nicht leer gehe zu Schritt 2.

# Beispiel: Sortierung



```
public class ArrayExample {
    public static void main(String[] args) {
        // o.b.d.A. a[k] > 0 & a[k] < 100
        int[] a = {12, 2, 32, 74, 26, 42, 53, 22};
        // create result array
        int[] b = new int[a.length];
        for (int i = 0; i < b.length; i++) { // set each item of b
            int minimum = 100;
            int pos = 0;
            for (int j = 0; j < a.length; j++) { // find minimum
                if (a[j] < minimum) {
                    minimum = a[j];
                    pos = j;
                }
            }
            b[i] = minimum;
            a[pos] = 100; // set visited.
        }

        for (int i = 0; i < b.length; i++) {
            System.out.print(b[i] + ", ");
        }
    }
}
```

- Lösbar auf viele Arten
- Vgl. AlgoDat!



# Beispiel: Sieb des Eratosthenes



```
public class Sieve {
    public static void main(String[] args) {
        int maxPrime = 200000;
        boolean[] sieve = new boolean[maxPrime];
        // init array
        for (int i = 0; i < sieve.length; i++) {
            sieve[i] = true;
        }

        // mark the non-primes
        for (int i = 2; i < Math.sqrt(sieve.length); i++) {
            if (sieve[i] == true) { // if it is a prime
                int k = 2;
                while (k*i < sieve.length) {
                    sieve[k*i] = false;
                    k++;
                }
            }
        }

        // print results
        for (int i = 2; i < sieve.length; i++) {
            if (sieve[i]) System.out.println(i);
        }
    }
}
```



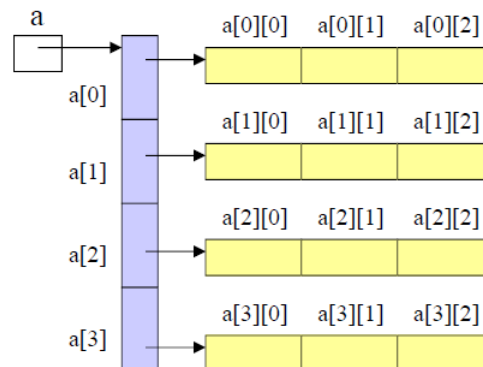
# Mehrdimensionale Arrays



- Zweidimensionales Arrays == Matrix

	0	1	2
0	a[0][0]	a[0][1]	a[0][2]
1	a[1][0]	a[1][1]	a[1][2]
2	a[2][0]	a[2][1]	a[2][2]
3	a[3][0]	a[3][1]	a[3][2]

- In Java: Array von Arrays



Deklaration und Erzeugung

```
int[][] a;  
a = new int[4][3];
```

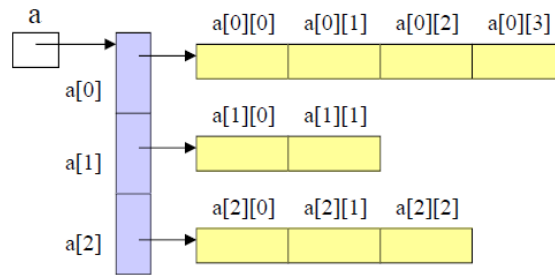
Zugriff

```
a[i][j] = a[i][j+1];
```

# Mehrdimensionale Arrays



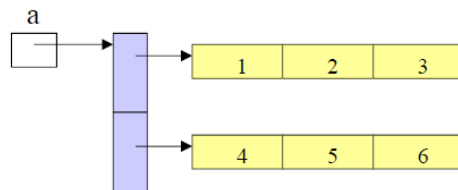
- Zeilen können unterschiedlich lang sein



```
int[][] a = new int[3][];  
a[0] = new int[4];  
a[1] = new int[2];  
a[2] = new int[3];
```

- Initialisierung

```
int[][] a = {{1, 2, 3}, {4, 5, 6}};
```



# ESOP - Klassen und Objekte

Assoc. Prof. Dr. Mathias Lux  
ITEC / AAU

# Retrospektiv ...



- Skalare Datentypen
  - „basic data types“ int, byte, short, int, long, float, double, boolean, char
  - Variable enthält Wert
- Aggregierte Datentypen
  - Mehrere Datenelemente über einen Namen verwaltbar
  - siehe Arrays ...

# Retrospektiv ...



- Referenzdatentypen
  - Variable speichert Referenz (nicht Wert)
- In Java
  - fundamentale Typen -> call by value
  - alles andere -> call by reference

# Über „Alles Andere“ ...



- Zusammenfassung
  - von fundamentalen Datentypen
  - in eine komplexe Struktur
- In jeder Sprache ein bisschen anders ...
  - Pascal: Record
  - C: struct
  - Java / Python: class

# Java-Klassen



- Beispiel: Speichern eines Datums
  - Tag, Monat, Jahr
- Unbequem ..
  - wenn man mehrere speichern will
  - als Rückgabewert einer Funktion
  - im Vergleich mit anderen Datums-Elementen



# Java-Klassen



- Idee: Fasse diese 3 Variablen zu einer Struktur (Klasse) zusammen:

<b>Date</b>
day : int
month : String
year : int

# Datentype Klasse



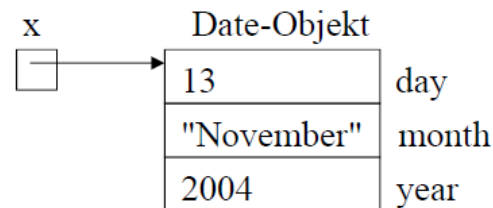
- Deklaration
- Verwendung als Typ
- Zugriff

```
class Date {  
    int day;  
    String month;  
    int year;  
}
```

Felder der Klasse *Date*

```
Date x, y;
```

```
x.day = 13;  
x.month = "November";  
x.year = 2004;
```



Date-Variablen sind Zeiger auf Objekte



# Objekte



- Objekte (Instanzen) einer Klasse müssen vor Verwendung erzeugt werden!

Date x, y;

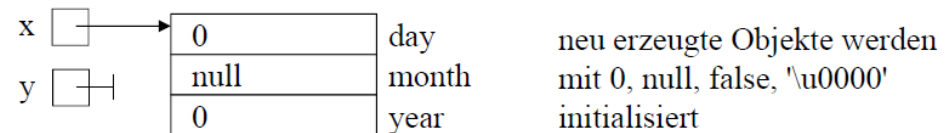
reserviert nur Speicher für die Zeigervariablen

x  y  haben anfangs den Wert *null*

## *Erzeugung*

x = **new** Date();

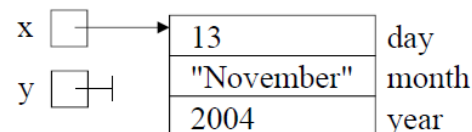
erzeugt ein *Date*-Objekt und weist seine Adresse *x* zu



Eine Klasse ist wie eine Schablone, von der beliebig viele Objekte erzeugt werden können.

## *Benutzung*

x.day = 13;  
x.month = "November";  
x.year = 2004;



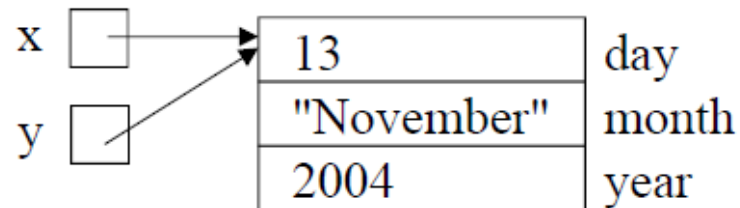
## *Freigabe von Objekten*

durch den Garbage Collector

# Zuweisungen

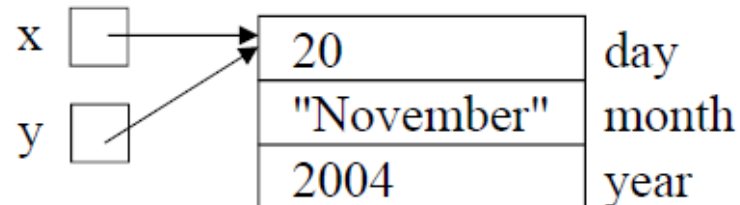


`y = x;`



*Zeigerzuweisung!*

`y.day = 20;`



ändert auch x.day!

# Zuweisungen



```
class Date {  
    int day;  
    String month;  
    int year;  
}
```

```
class Address {  
    int number;  
    String street;  
    int zipCode;  
}
```

```
Date d1, d2 = new Date();  
Address a1, a2 = new Address();
```

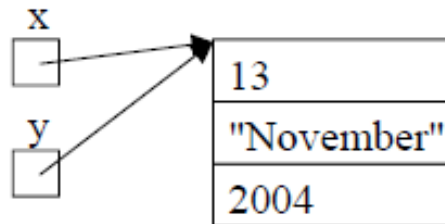
```
d1 = d2;           // ok, gleiche Typen  
a1 = a2;           // ok, gleiche Typen  
d1 = a2;           // verboten: verschiedene Typen trotz gleicher Struktur!
```

# Referenzvergleich

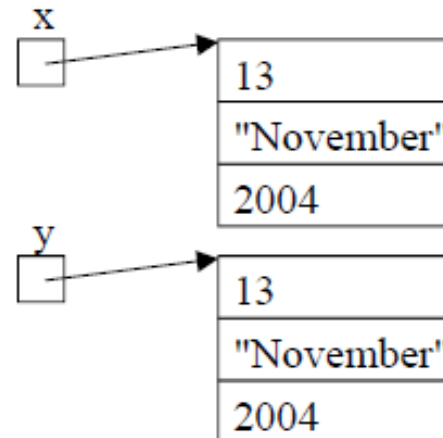


- $x == y$  und  $x != y$  ... vergleicht Referenzen
- $<$ ,  $<=$ ,  $>$ ,  $>=$  ... nicht erlaubt

$x == y$  liefert true



$x == y$  liefert false



# Wertevergleich



- Muss durch Methode implementiert werden.

```
static boolean equalDate (Date x, y) {  
    return x.day == y.day &&  
        x.month.equals(y.month) &&  
        x.year == y.year;  
}
```

# Wo werden Klassen deklariert?



## Einzelne Datei

```
class C1 {  
    ...  
}  
class C2 {  
    ...  
}  
class MainProgram {  
    public static void  
        main (String[] arg) {  
        ...  
    }  
}
```

MainProgram.java

Übersetzung

\$> javac MainProgram.java

## Getrennte Dateien

```
class C1 {  
    ...  
}  
class C2 {  
    ...  
}  
class MainProgram {  
    public static void  
        main (String[] arg) {  
        ...  
    }  
}
```

C1.java

C2.java

MainProgram.java

Übersetzung

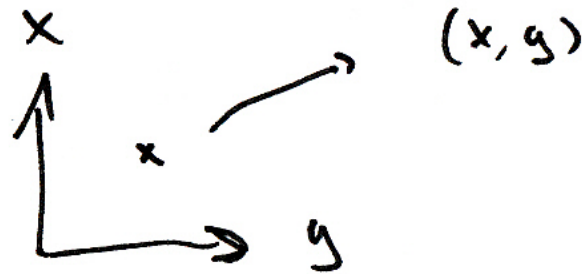
\$> javac MainProgram.java C1.java C2.java



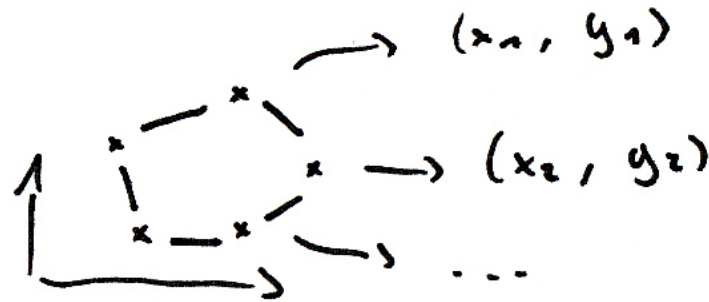
# Was kann man mit Klassen tun?



Point



Polygon



# Was kann man mit Klassen tun?



- Klassen können andere Klassen beinhalten:

```
class Point {  
    int x, y;  
}  
class Polygon {  
    Point[] pt;  
    int color;  
}
```



# Was kann man mit Klassen tun?



- Implementierung von Methoden mit mehreren Rückgabewerten:

```
class Time {  
    int h, m, s;  
}  
class Program {  
    static Time convert (int sec) {  
        Time t = new Time();  
        t.h = sec / 3600; t.m = (sec % 3600) / 60; t.s = sec % 60;  
        return t;  
    }  
    public static void main (String[] arg) {  
        Time t = convert(10000);  
        System.out.println(t.h + ":" + t.m + ":" + t.s);  
    }  
}
```

# Was kann man mit Klassen tun?



- Kombination von Klassen mit Arrays:

```
class Person {  
    String name, phoneNumber;  
}
```

```
class Phonebook {  
    Person[] entries;  
}
```

```
class Program {  
    public static void main (String[] arg) {  
        Phonebook phonebook = new Phonebook();  
        phonebook.entries = new Person[10];  
        phonebook.entries[0].name = "Mathias Lux"  
        phonebook.entries[0].phoneNumber = "+43 463 2700 3615"  
        // ...  
    }  
}
```

# Objektorientierung



- Bisher erläutert ...
  - Klasse fasst Datentypen zusammen
  - Funktioniert mit Methoden, Arrays und anderen Klassen
- Objektorientierung
  - Klassen = Daten + Methoden

# Beispiel: Positionsklasse



```
class Position {  
    private int x;  
    private int y;  
  
    void goLeft() { x = x - 1; }  
    void goRight() { x = x + 1; }  
}
```

// ... Benutzung

```
Position pos1 = new Position();  
pos1.goLeft();  
Position pos2 = new Position();  
pos2.goRight();
```

- Methoden sind lokal definiert
  - ohne Keyword *static*
- Jedes Objekt hat seinen eigenen Zustand
  - pos1 = new Position()
  - pos2 = new Position()
  - ...

# Beispiel: Positionsklasse



```
class Position {  
    private int x;  
    private int y;  
  
    // Methoden mit Parametern  
    void goLeft(int n) {  
        x = x - n;  
    }  
  
    // [...]  
}
```

- Nutzung von Parametern in Methoden.
- Dasselbe gilt für Rückgabewerte.

# Beispiel: Positionsklasse



```
class Position {  
    private int x;  
    private int y;  
  
    // Keyword "this"  
    void goLeft(int x) {  
        this.x = this.x - x;  
    }  
  
    // [...]  
}
```

- *this* notwendig wenn auf Klassenweiten Scope zugegriffen wird
- Bei Nichtnutzung würde lokales *x* verwendet werden.

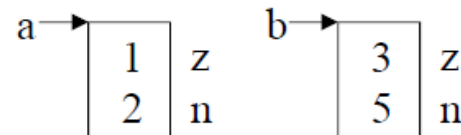


# Beispiel: Bruchzahlenklasse

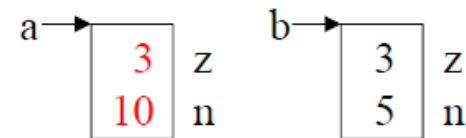


```
class Fraction {  
    int z; // Zähler  
    int n; // Nenner  
  
    void mult (Fraction f) {  
        z = z * f.z;  
        n = n * f.n;  
    }  
  
    void add (Fraction f) {  
        z = z * f.n + f.z * n;  
        n = n * f.n;  
    }  
}
```

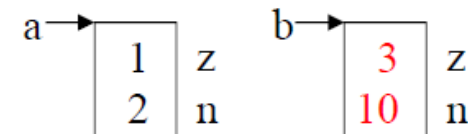
Fraction a = new Fraction(); a.z = 1; a.n = 2;  
Fraction b = new Fraction(); b.z = 3; b.n = 5;



**a.mult(b);**



**b.mult(a);**



Es wird immer der Zustand des Empfängers verändert!

# UML Notation



<b>Fraction</b>	<i>Klassenname</i>
int z int n	<i>Felder</i>
void mult(Fraction f) void add(Fraction f)	<i>Methoden</i>

<b>Fraction</b>	<i>Vereinfachte Form</i>
z n	
mult(f) add(f)	

# Konstruktoren



- Spezielle Methoden
  - bei der Instanziierung aufgerufen
  - dienen zur Initialisierung eines Objekts
  - heißen wie die Klasse
  - ohne Funktionstyp und ohne *void*
  - können Parameter haben
  - können überladen werden

# Konstrukturen



```
class Fraction {  
    int z, n;  
  
    Fraction (int z, int n) {  
        this.z = z; this.n = n;  
    }  
  
    Fraction () {  
        z = 0; n = 1;  
    }  
  
    void mult (Fraction f) {...}  
    void add (Fraction f) {...}  
}
```

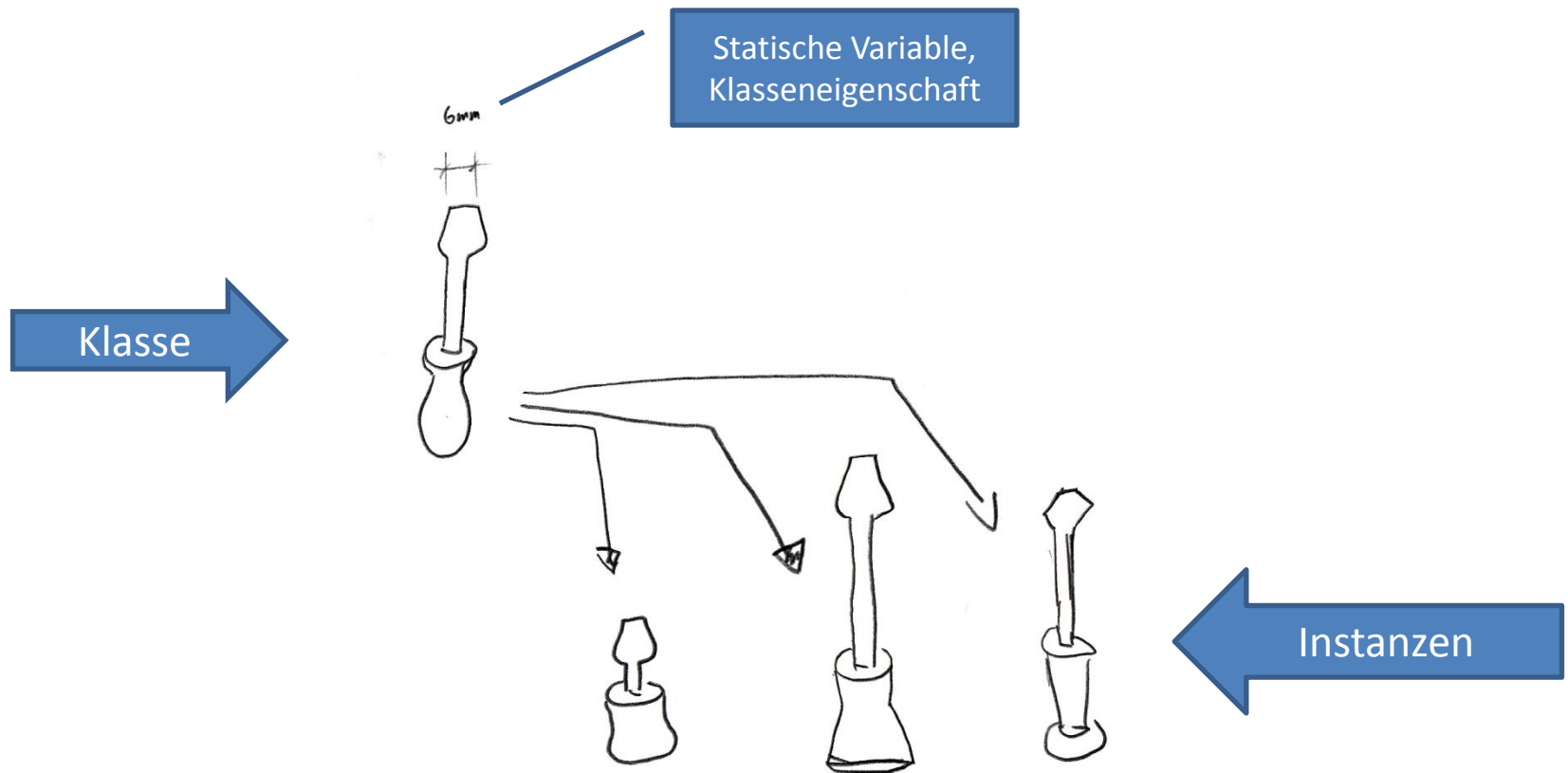
- Fraction f = new Fraction();
- Fraction g = new Fraction(3, 5);
  
- ruft entsprechende Konstrukturen auf ...

# Konstruktoren ...



- Beispiel: Time-Klasse
- Beispiel: Position-Klasse

# static



# static



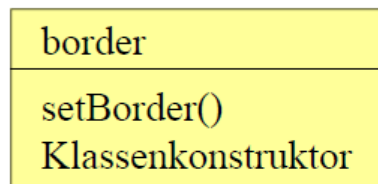
```
class Window {  
    int x, y, w, h; // Objektfelder (in jedem Window-Objekt vorhanden)  
    static int border; // Klassenfeld (nur einmal pro Klasse vorhanden)  
  
    Window(int x, int y, int w, int h) {...} // Objektkonstruktor (zur Initialisierung von Objekten)  
    → static { // Klassenkonstruktor (zur Initialisierung der Klasse)  
        border = 3;  
    }  
  
    void redraw () {...} // Objektmethode (auf Objekte anwendbar)  
    → static void setBorder (int n) {border = n;} // Klassenmethode (auf Klasse Window anwendbar)  
}
```

# static

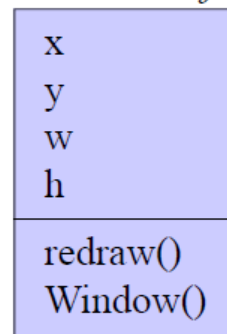


- Objektmethoden haben Zugriff auf Klassenfelder
  - `redraw()` kann auf `border` zugreifen
- Klassenmethoden haben keinen direkten Zugriff auf Objektfelder
  - `setBorder()` kann nicht auf `x` zugreifen

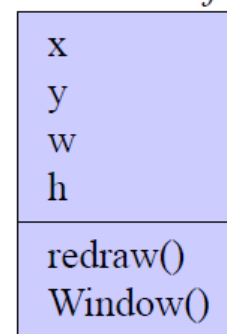
*Klasse Window*



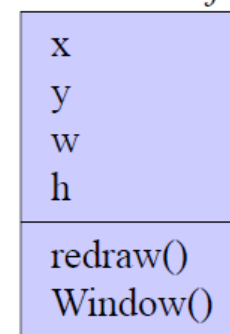
*Window-Objekt*



*Window-Objekt*



*Window-Objekt*





# static



## Was geschieht wann?

- Beim Laden der Klasse Window
  - Klassenfelder werden angelegt (border)
  - Klassenkonstruktor wird aufgerufen
- Beim Erzeugen eines Window-Objekts (`new Window(...)`)
  - Objektfelder werden angelegt ( $x, y, w, h$ )
  - Objektkonstruktor wird aufgerufen

# static



- Zugriff static-Elemente: Klassennamen
  - `Window.border = ...; Window.setBorder(3);`
  - Methoden der Klasse `Window` können Klassennamen weglassen (`border = ...; setBorder(3);`)
- Zugriff nonstatic-Elemente: Objektnamen
  - `Window win = new Window(100, 50);`  
`win.x = ...; win.redraw();`
  - Methoden der Klasse `Window` können auf eigene Elemente direkt zugreifen (`x = ...; redraw();`)

# static



- **Achtung: Statische Felder leben während der gesamten Programmausführung!**
- **Entsprechend: Lokalitätsprinzip anwenden!**
- **Vgl. auch SE und weiterführende VO/PRs**

# Zusätzliches Tutorium



Wann? 19.11.2014

Um? 17:00 - ca. 19:00

Wo? HS2

# Beispiel: Stack & Queue



- Stack (Stapel, Kellerspeicher)
  - push(x) ... legt x auf den Stapel
  - pop() ... entfernt/liefert oberstes Element
  - LIFO-Datenstruktur == last in first out
- Queue (Puffer, Schlange)
  - put(x) ... stellt x hinten an
  - get() ... entfernt/liefert erstes Element
  - FIFO-Datenstruktur == first in first out

# Queue



```
class Queue {
    int[] data;
    int head, tail, n;

    Queue(int size) {
        data = new int[size];
        head = 0;
        tail = 0;
        n = 0;
    }

    void put(int x) {
        if (n == data.length) System.out.println("-- overflow");
        else {
            data[tail] = x;
            n++;
            tail = (tail + 1) % data.length;
        }
    }

    int get() {
        int x;
        if (n <= 0) {
            System.out.println("-- underflow");
            return 0;
        } else x = data[head];
        n--;
        head = (head + 1) % data.length;
        return x;
    }
}
```

## Nutzung:

```
Queue q = new Queue(10);
q.put(3);
q.put(6);
int x = q.get(); // x == 3
int y = q.get(); // y == 6
```

# Stack ...



```
public class Stack {
    int[] data;
    int top;

    Stack(int size) {
        data = new int[size];
        top = -1;
    }

    void push(int x) {
        if (top == data.length - 1)
            System.out.println("-- overflow");
        else
            data[++top] = x;
    }

    int pop() {
        if (top < 0) {
            System.out.println("-- underflow");
            return 0;
        } else
            return data[top--];
    }
}
```

## Nutzung:

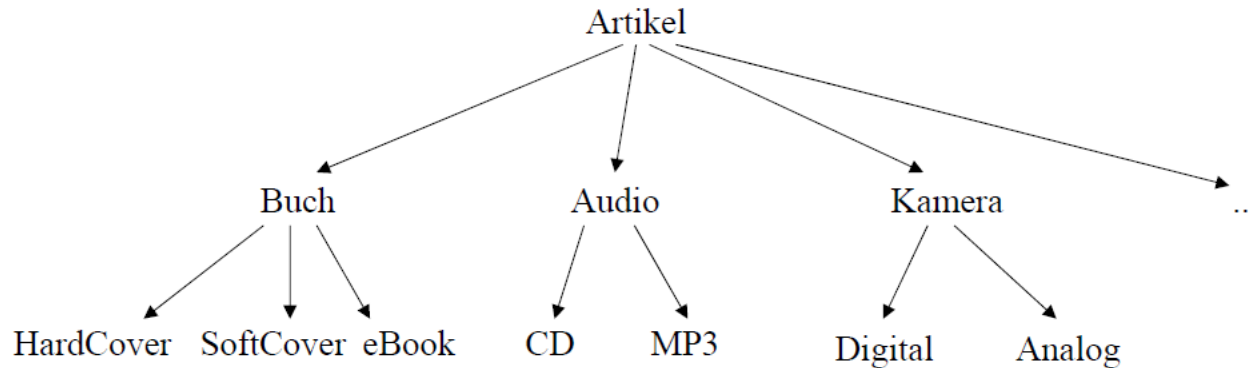
```
Stack s = new Stack(10);
s.push(3);
s.push(5);
int x = s.pop() - s.pop();
// x == 2 (d.h. 5 - 3)
```

# Klassifikation



## Dinge der realen Welt lassen sich oft klassifizieren

z.B. Artikel eines Web-Shops



## Man beachte

- Ein *eBook* hat alle Eigenschaften eines *Buchs*; zusätzlich hat es ...  
Ein *Buch* hat alle Eigenschaften eines *Artikels*; zusätzlich hat es ...
- *CD* und *MP3* lassen sich gleichermaßen als *Audio* behandeln  
*Buch*, *Audio* und *Kamera* lassen sich gleichermaßen als *Artikel* behandeln

Vererbung



# Vererbung



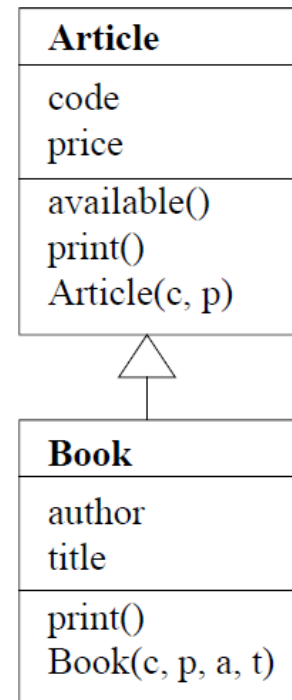
```
class Article {  
    int code;  
    int price;  
  
    boolean available() {...}  
    void print() {...}  
  
    Article(int c, int p) {...}  
}
```

```
class Book extends Article {  
    String author;  
    String title;  
  
    void print() {...}  
  
    Book(int c, int p,  
        String a, String t) {...}  
}
```

**Oberklasse**  
**Basisklasse**

**Unterklasse**

erbt: *code, price, available, print*  
ergänzt: *author, title, Konstruktor*  
überschreibt: *print*



Wenn keine Oberklasse angegeben wird, ist sie *Object*

# Überschreiben von Methoden



```
class Article {  
    ...  
    void print() {  
        Out.print(code + " " + price);  
    }  
    Article(int c, int p) {  
        code = c; price = p;  
    }  
}
```

```
class Book extends Article {  
    ...  
    void print() {  
        super.print();  
        Out.print(" " + author + ": " + title);  
    }  
    Book(int c, int p, String a, String t) {  
        super(c, p);  
        author = a; title = t;  
    }  
}
```

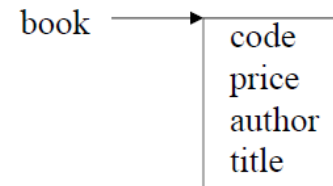
## Benutzung

```
Book book = new Book(code, price, author, title);
```

⇒ erzeugt *Book*-Objekt

⇒ *Book*-Konstruktor

⇒ *Article*-Konstruktor (code = c; price = p;)  
author = a; title = t;



```
book.print();
```

⇒ *print* aus *Book*

⇒ *print* aus *Article*

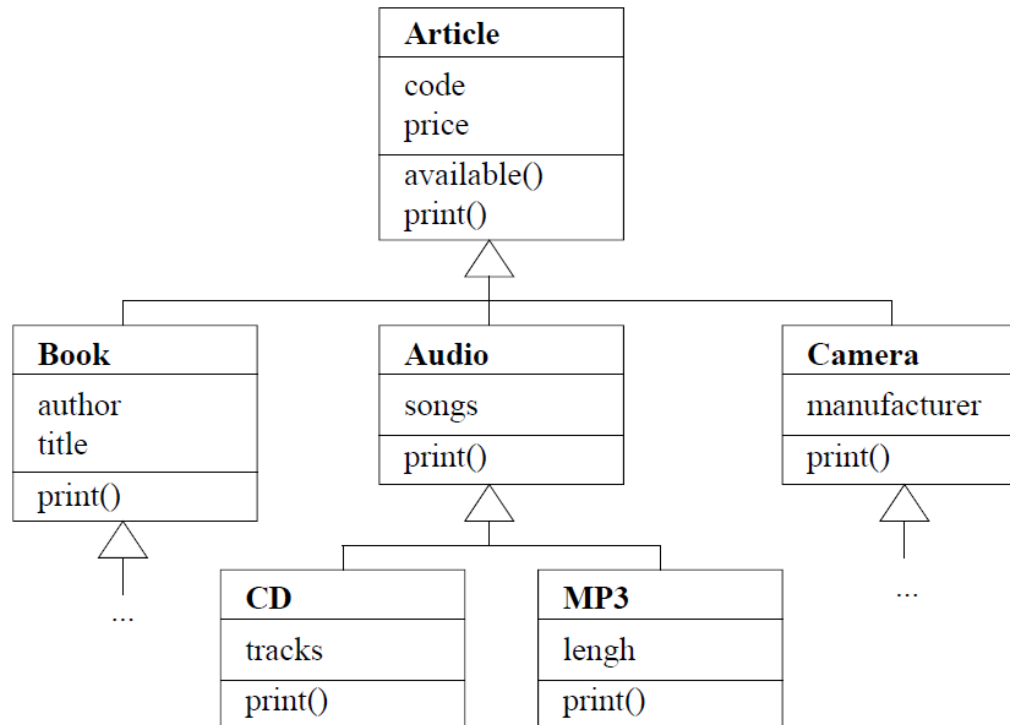
⇒ Out.print(...);

code price

author: title

Ausgabe: code price author: title

# Klassenhierarchien



Jedes Buch ist ein Artikel  
Aber: nicht jeder Artikel ist ein Buch

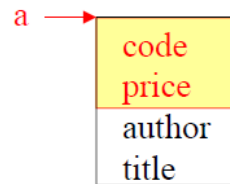
# Kompatibilität zwischen Klassen



**Unterklassen sind Spezialisierungen ihrer Oberklassen**

***Book*-Objekte können *Article*-Variablen zugewiesen werden**

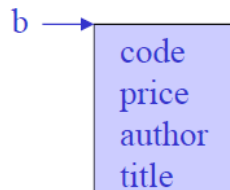
```
Article a = new Book(code, price, author, title);
```



nur *Article*-Felder sind über *a* zugreifbar

a.code  
a.price

```
if (a instanceof Book) // Laufzeittypstest  
    Book b = (Book) a; // Typumwandlung mit Laufzeittypprüfung
```



alle *Book*-Felder sind über *b* zugreifbar

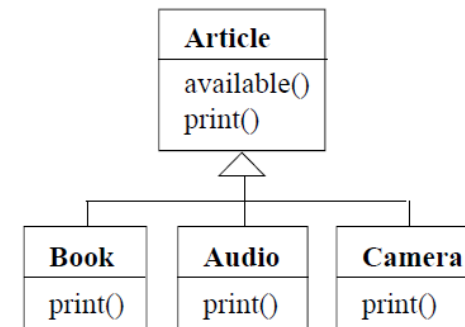
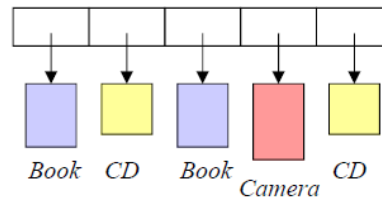
b.code  
b.price  
b.author  
b.title

# Dynamische Bindung



## Heterogene Datenstruktur

Article[] a;



Alle Varianten können als Artikel behandelt werden

```
void printArticles() {
    for (int i = 0; i < a.length; i++) {
        if (a[i].available()) {
            a[i].print();
        }
    }
}
```

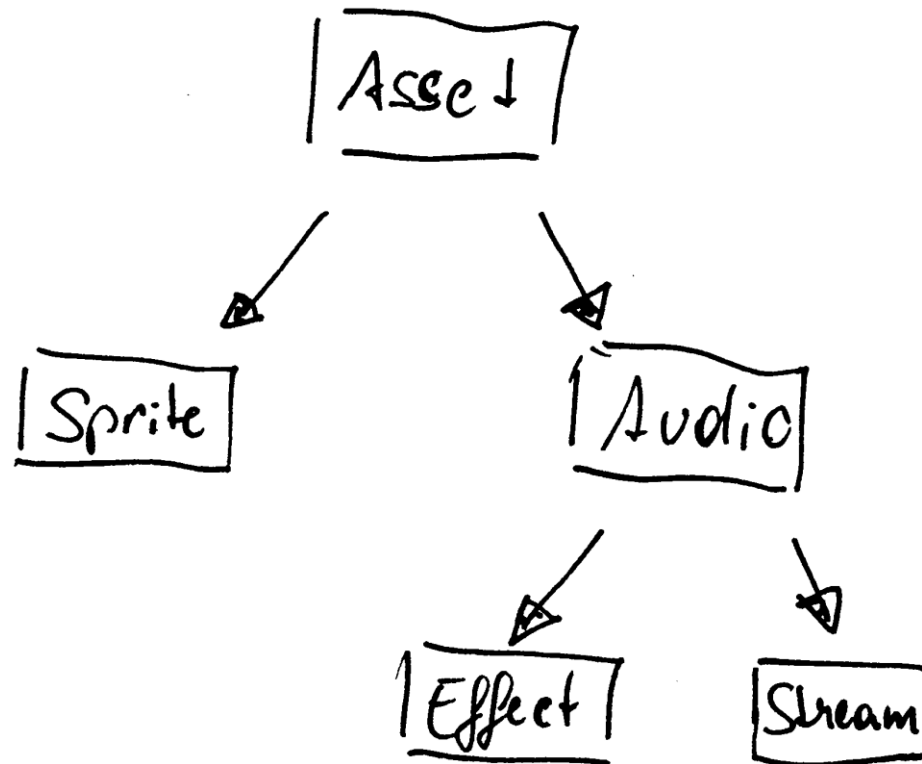
ruft *available()* aus *Article* auf

ruft je nach Artikelart das *print()* aus *Book*, *CD* oder *Camera* auf

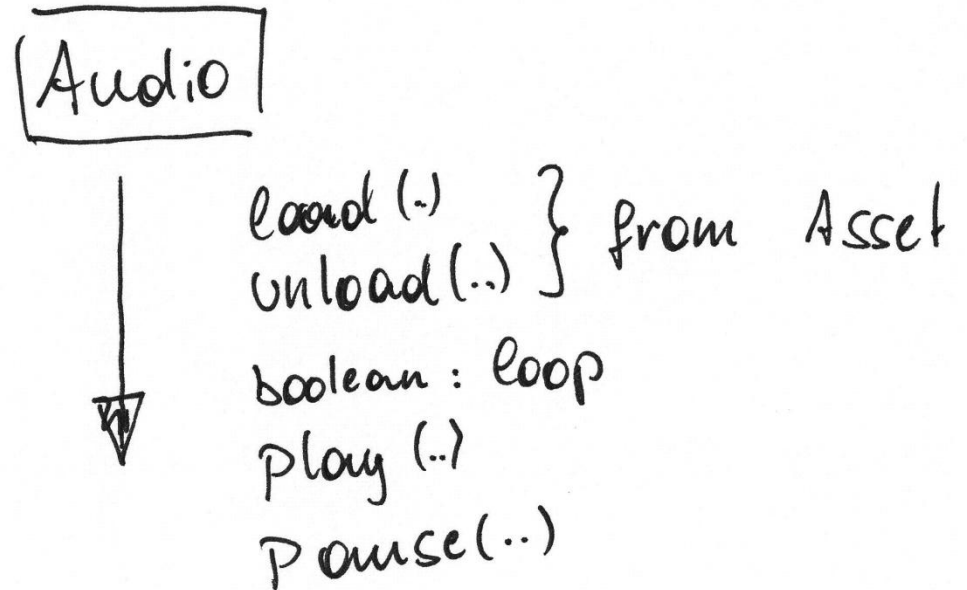
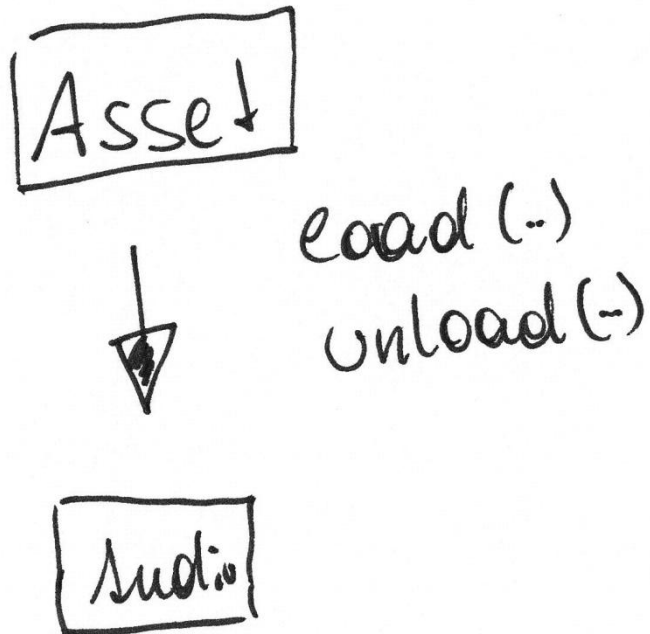
## Dynamische Bindung

*obj.print()* ruft die *print*-Methode des Objekts auf, auf das *obj* gerade zeigt

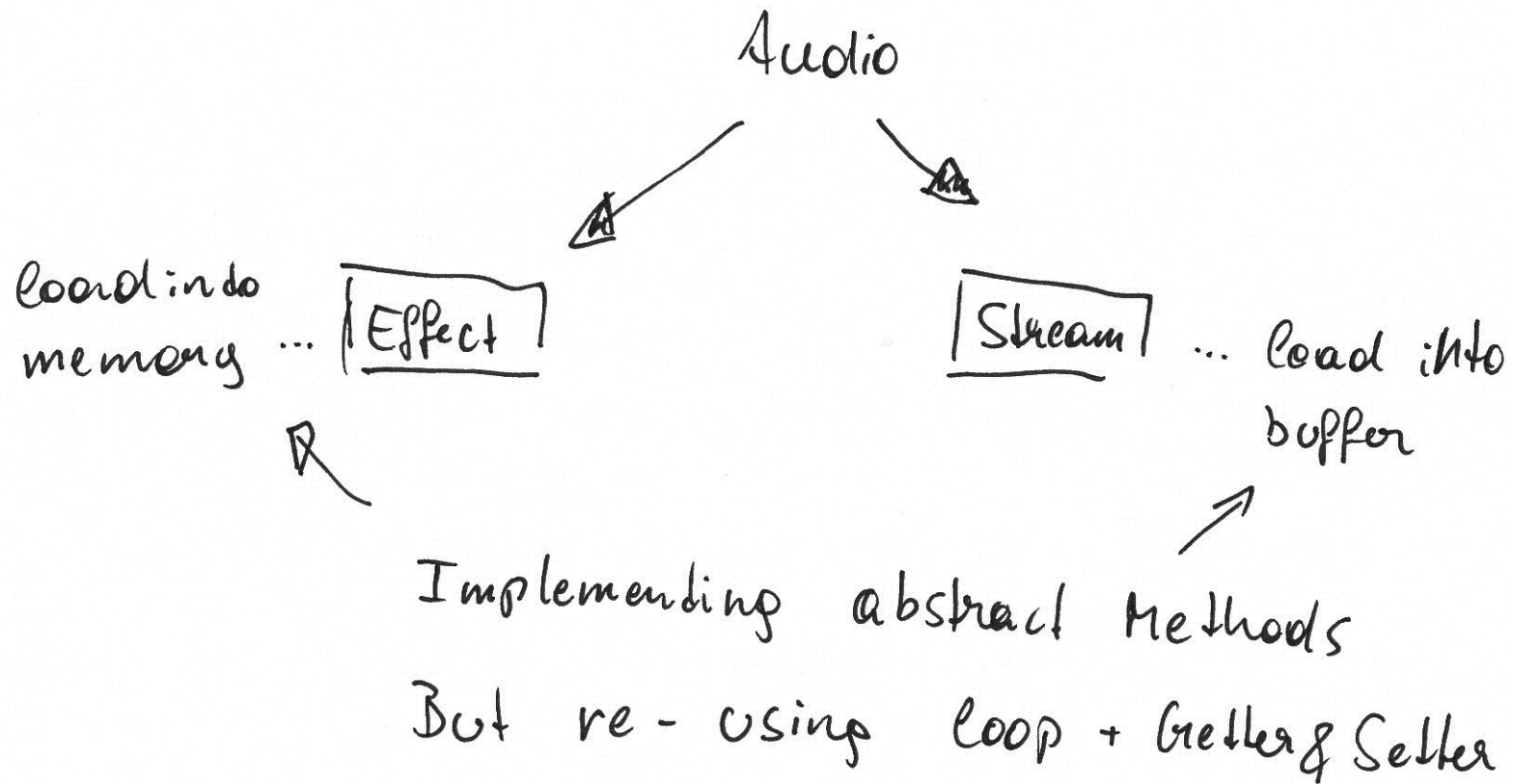
# Beispiel ...



# Beispiel



# Beispiel





# Zusätzliche Konzepte



## Keyword **abstract**

- spezifiziert, dass alle Subklassen eine solche Methode haben,
- aber bietet sie nicht an
  - Im Gegensatz, sie wird verlangt.

# ESOP - Information Hiding

Assoc. Prof. Dr. Mathias Lux  
ITEC / AAU

# Addendum



super kann nur auf die direkte Superklasse zugreifen.

- Sonst würden Vererbungsprinzipien verletzt werden
  - Überspringen/Ignorieren der Superklasse

# Geheimnisprinzip



- In großen Softwaresystemen muss der globale Namensraum strukturiert und eingeschränkt werden: information hiding
- Unterscheidung zwischen öffentlichen und geheimen Bezeichnern.

# Beispiel



```
public class ShipExample {  
    // actual position of the ship  
    private int positionX, positionY;  
    // maximum number for x and y  
    private int maxX = 320, maxY = 640;
```



```
    public ShipExample() {  
        this.positionX = maxX/2;  
        this.positionY = maxY/2;  
    }  
  
    public void moveShip(int offSetX, int offsetY) {  
        positionX += offSetX;  
        positionY += offsetY;  
        // check for violation of maximum  
        if (positionX > maxX)  
            positionX = maxX;  
        if (positionY > maxY)  
            positionY = maxY;  
    }  
}
```

# Geheimnisprinzip



- Klienten können ausschließlich auf die spezifizierten Operationen
- Eine geprüfte Komponente kann durch einen fehlerhaften externen Zugriff nicht zerstört werden

# Geheimnisprinzip



- Bezeichner, die in der Spezifikation eines abstrakten Datentyps vorkommen, sollten öffentlich sein
- Bezeichner, die nur für die Implementierung notwendig sind sollten für Klienten verborgen bleiben

# Grundsätzlich ...



- Nie mehr öffentlich preisgeben als notwendig!



# Beispiel: Zu Öffentlich



```
Stack armerStack = new Stack();  
armerStack.push(1);  
armerStack.push(2);  
armerStack.push(3);  
armerStack.top = 0; // 2 und 3 werden "gelöscht"  
int drei = armerStack.pop();
```

# Dynamische Datenstrukturen



- Elemente werden zur Laufzeit angelegt
  - mit *new*, sozusagen „dynamisch“
- Datenstruktur kann dynamisch
  - wachsen: Speicher wird belegt
  - schrumpfen: Speicher wird frei

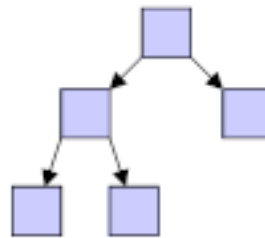
# Dynamische Datenstrukturen



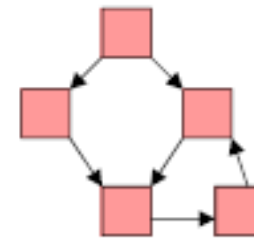
- Wichtige dynamische Datenstrukturen



*Liste*



*Baum*



*Graph*

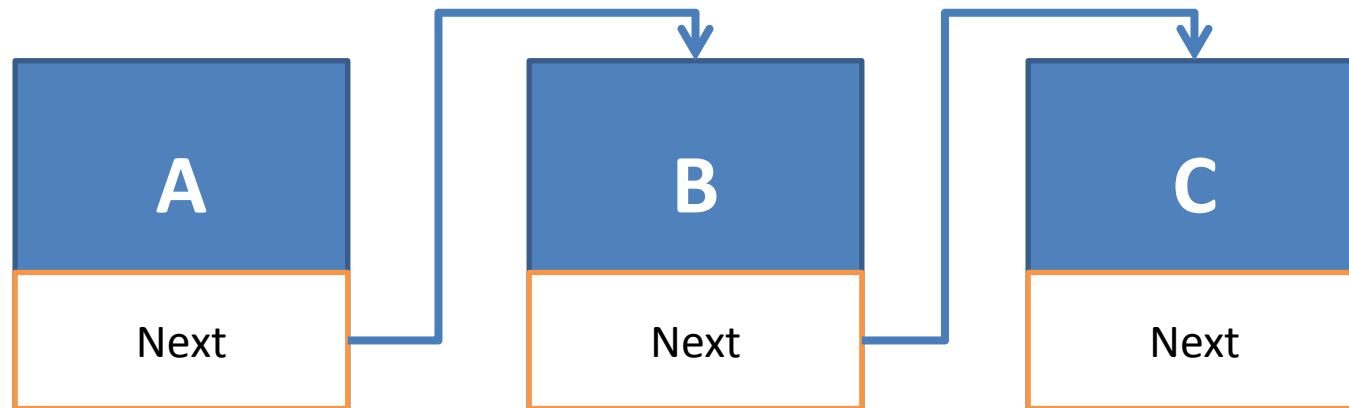
# Dynamische Datenstrukturen



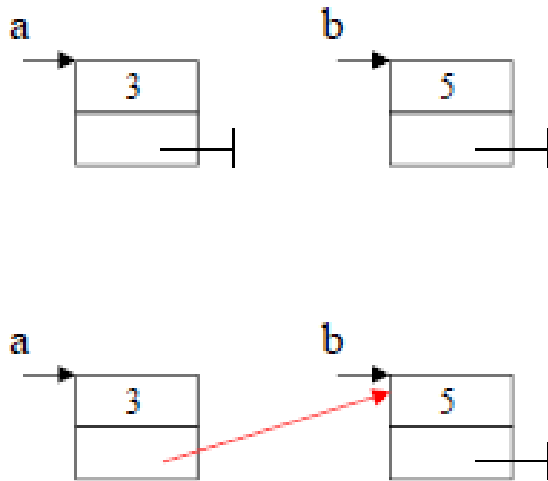
Bestehen aus Knoten die über Kanten miteinander verbunden sind.

- Knoten ... Objekte
- Kanten ... Zeiger / Referenzen

# Verknüpfen von Knoten

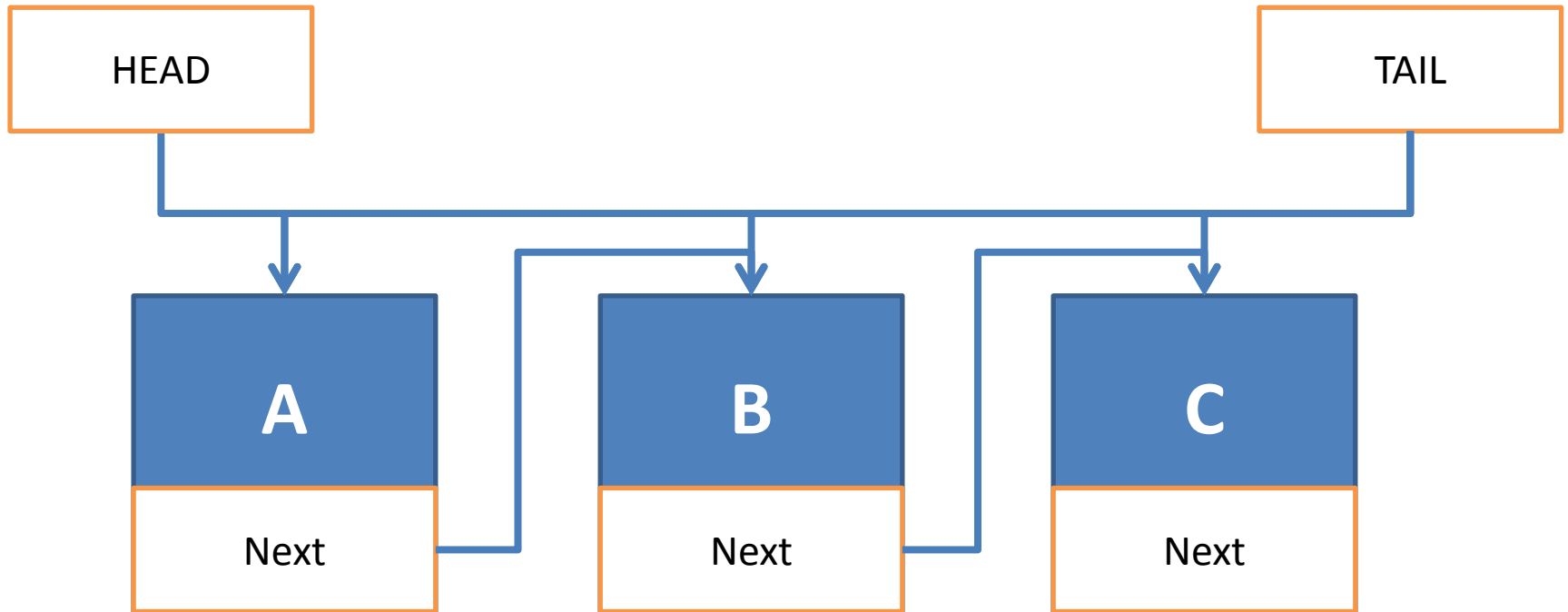


# Verknüpfen von Knoten.

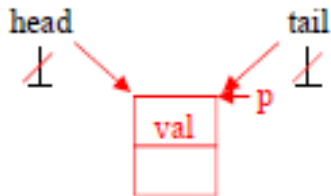
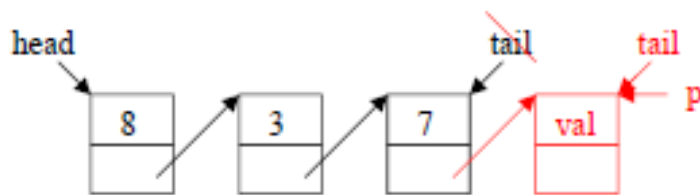


```
public class Node {  
    int value;  
    Node nextNode;  
  
    public Node(int value) {  
        this.value = value;  
    }  
}  
  
// ...  
Node a = new Node(3);  
Node b = new Node(5);  
a.nextNode = b;
```

# Einfügen am Listenende



# Unsortierte Liste: Einfügen am Listende

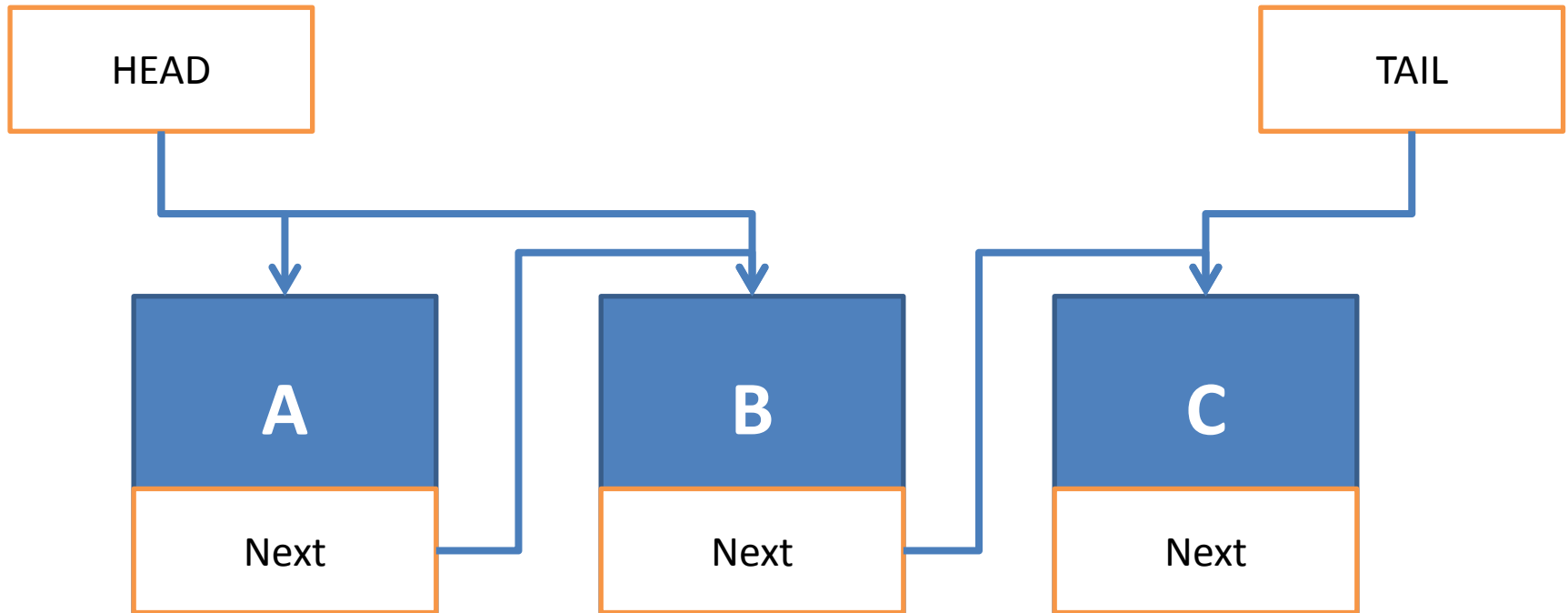


```
public class List {
    private Node head, tail;

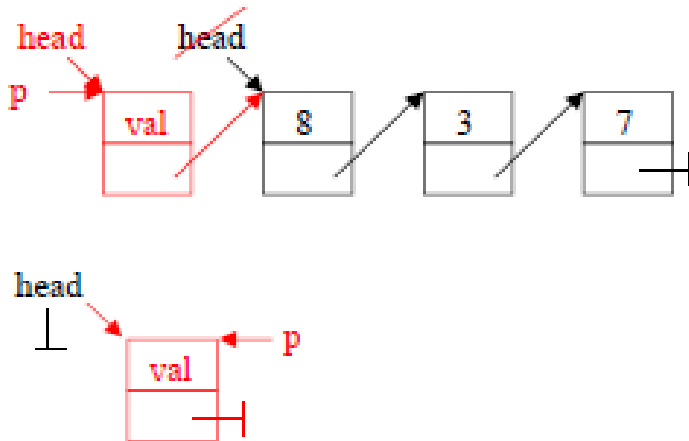
    public void append(int val) {
        Node p = new Node(val);
        if (head == null)
            head = p;
        else
            tail.nextNode = p;
        tail = p;
    }
    //...
}
// ...
List l = new List();
l.append(3);
l.append(4);
```



# Einfügen am Listenanfang



# Unsortierte Liste: Einfügen am Listenanfang



```
public class List {  
    private Node head, tail;  
  
    public void prepend(int val) {  
        Node p = new Node(val);  
        p.nextNode = head;  
        head = p;  
    }  
    // ...  
  
    // ...  
List l = new List();  
l.prepend(3);  
l.prepend(4);
```

# Unsortierte Liste: Eintrag suchen



```
public class List {
    private Node head, tail;

    public boolean contains(int val) {
        Node p = head;
        boolean result = false;
        while (p!=null) {
            if (p.value == val) result = true;
            p = p.nextNode;
        }
        return result;
    }
    // ...
}
// ...
List l = new List();
l.append(3);
l.append(14);
l.append(-1);
System.out.println(l.contains(3));
```

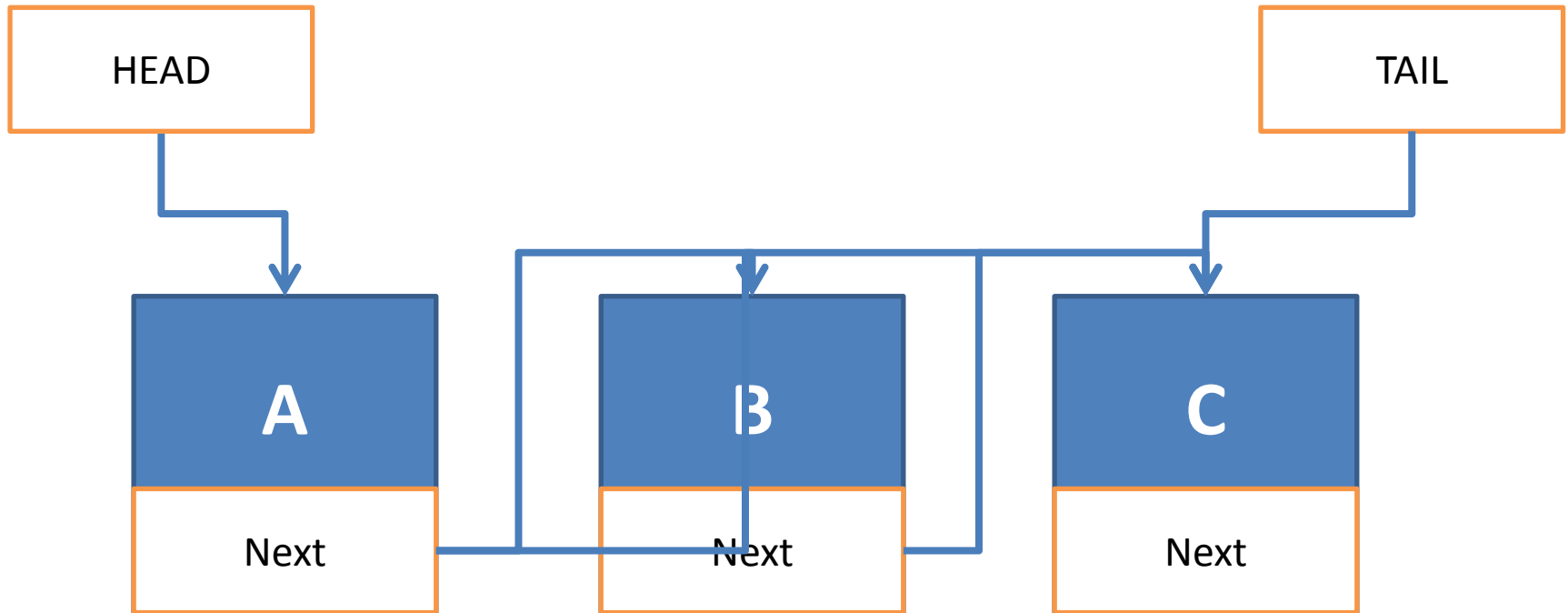
# Unsortierte Liste: Eintrag suchen



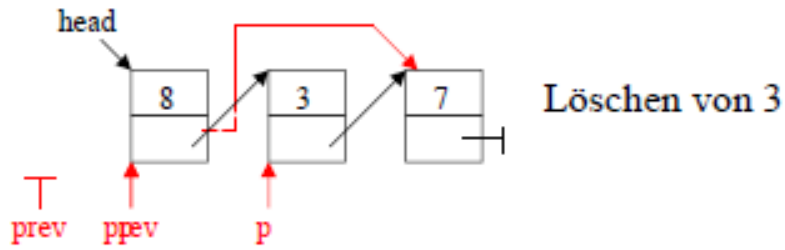
```
public class List {
    private Node head, tail;

    public boolean contains(int val) {
        Node p = head;
        while (p!=null && p.value!=val) {
            p = p.nextNode;
        }
        return p != null;
    }
    // ...
}
// ...
List l = new List();
l.append(3);
l.append(14);
l.append(-1);
System.out.println(l.contains(3));
```

# Unsortierte Liste: Eintrag löschen



# Unsortierte Liste: Eintrag löschen



```
public class List {
    private Node head, tail;

    public void delete(int val) {
        Node p = head, prev = null;
        while (p!=null && p.value!=val) {
            prev = p;
            p = p.nextNode;
        }
        if (p != null) {
            if (p == head)
                head = p.nextNode;
            else
                prev.nextNode = p.nextNode;
            if (tail == p)
                tail = prev;
        }
        // ...
    }
}
```

# Live Beispiel ...



- Methode `index(int val)`
- Stack & Queue mit dynamischen Datenstrukturen.

# ESOP - Rekursion / Interface / Math

Assoc. Prof. Dr. Mathias Lux  
ITEC / AAU



# Rekursion



- Eine Methode  $m()$  heißt *rekursiv*, wenn sie sich selbst aufruft
  - $m() \rightarrow m() \rightarrow m()$       direkt rekursiv
  - $m() \rightarrow n() \rightarrow m()$       indirekt rekursiv

# Rekursion: Fakultät n!



- Definition Fakultät

- $n! = (n-1)! * n$

- $1! = 1$

- Beispiel

- $4! = 4*3! = 4*3*2! = 4*3*2*1! = 4*3*2*1$

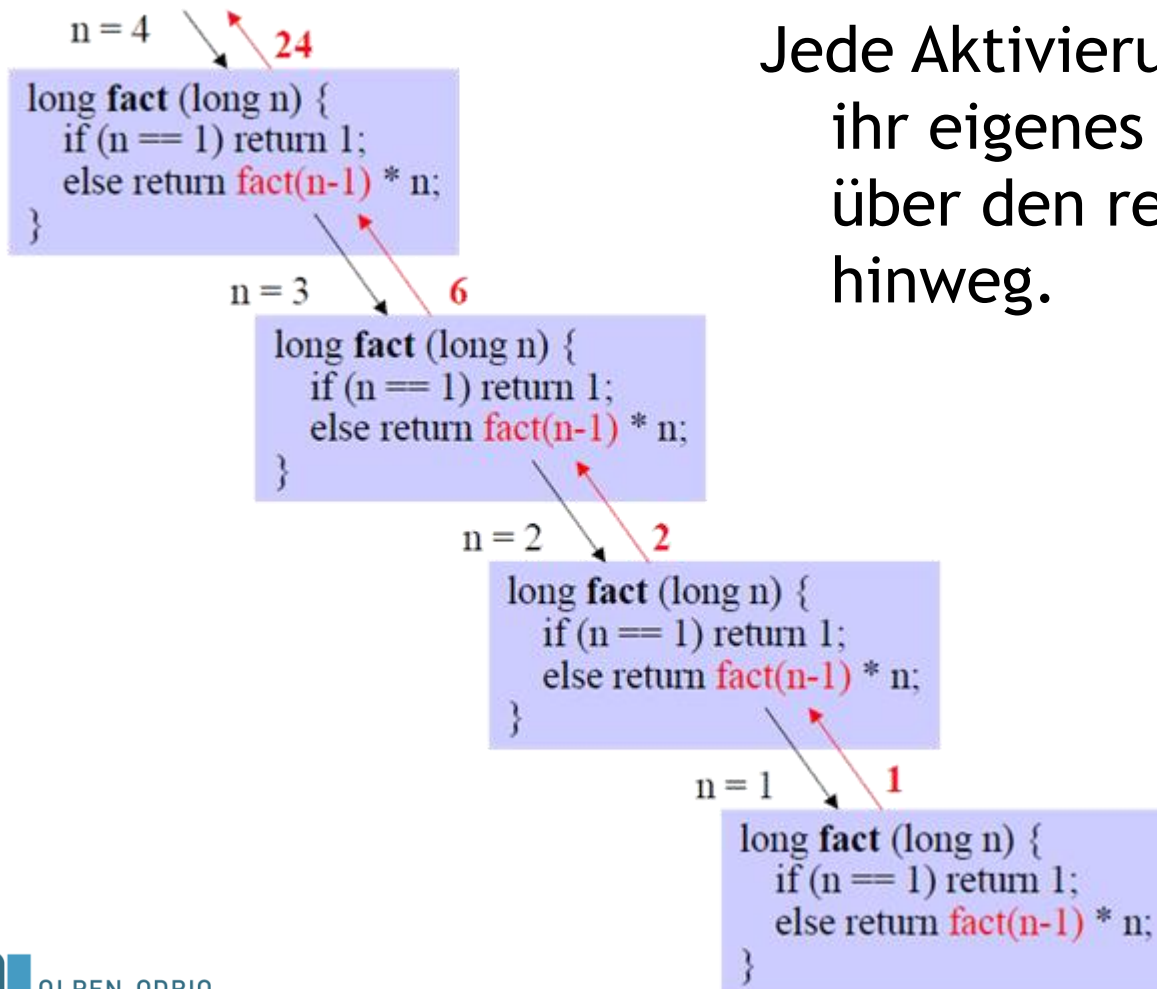
# Rekursion: Fakultät n!



```
long fact (long n) {  
    if (n == 1)  
        return 1;  
    else  
        return fact(n-1) * n;  
}
```

Ende der Rekursion  
bei Erreichen von  
1!

# Ablauf einer rekursiven Methode

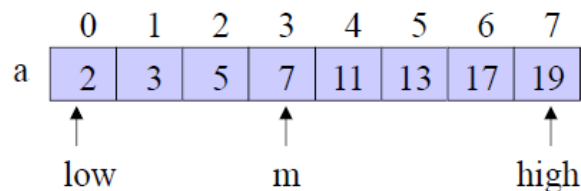


Jede Aktivierung von `fact` hat ihr eigenes `n` und rettet es über den rekursiven Aufruf hinweg.

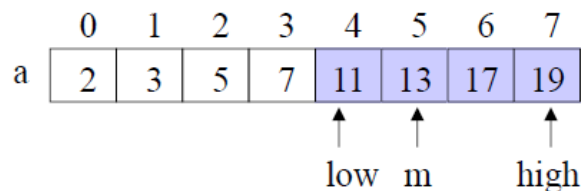
# Beispiel: Binäre Suche Rekursiv



z.B. Suche von 17 (Array muss sortiert sein)



- Index  $m$  des mittleren Element bestimmen
- $17 > a[m] \Rightarrow$  in rechter Hälfte weitersuchen



```
static int search (int elem, int[] a, int low, int high) {  
    if (low > high) return -1; // empty  
    int m = (low + high) / 2;  
    if (elem == a[m]) return m;  
    if (elem < a[m]) return search(elem, a, low, m-1);  
    return search(elem, a, m+1, high);  
}
```

} nichtrekursiver Zweig

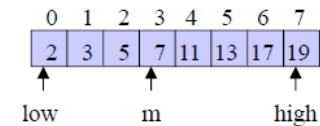
} rekursiver Zweig

# Interfaces



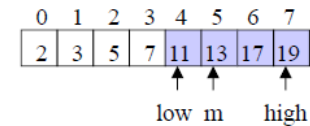
```
elem = 17, low = 0, high = 7    ↑ 6  
static int search (int elem, int[] a, int low, int high) {  
  if (low > high) return -1;  
  int m = (low + high) / 2;  
  if (elem == a[m]) return m;  
  if (elem < a[m]) return search(elem, a, low, m-1);  
  return search(elem, a, m+1, high);  
}
```

m = 3



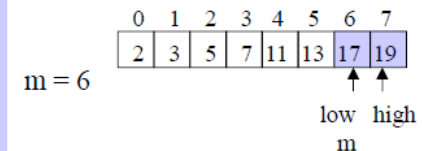
```
low = 4, high = 7    ↓ ↑ 6  
static int search (int elem, int[] a, int low, int high) {  
  if (low > high) return -1;  
  int m = (low + high) / 2;  
  if (elem == a[m]) return m;  
  if (elem < a[m]) return search(elem, a, low, m-1);  
  return search(elem, a, m+1, high);  
}
```

m = 5



```
low = 6, high = 7    ↓ ↑ 6  
static int search (int elem, int[] a, int low, int high) {  
  if (low > high) return -1;  
  int m = (low + high) / 2;  
  if (elem == a[m]) return m;  
  if (elem < a[m]) return search(elem, a, low, m-1);  
  return search(elem, a, m+1, high);  
}
```

m = 6

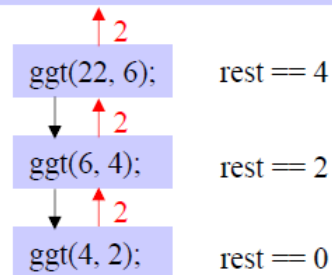


# Beispiel: GGT



*rekursiv*

```
static int ggt (int x, int y) {  
    int rest = x % y;  
    if (rest == 0) return y;  
    else return ggt(y, rest);  
}
```



*iterativ*

```
static int ggt (int x, int y) {  
    int rest = x % y;  
    while (rest != 0){  
        x = y; y = rest;  
        rest = x % y;  
    }  
    return y;  
}
```

Jeder rekursive Algorithmus kann auch iterativ programmiert werden

- rekursiv: meist kürzerer Quellcode
- iterativ: meist kürzere Laufzeit

Rekursion v.a. bei rekursiven Datenstrukturen nützlich (Bäume, Graphen, ...)

# Beispiel: Fibonacci Zahlen



- $F_n = F_{n-1} + F_{n-2}$

```
public static int get(int number) {  
    if (number <= 2)  
        return 1;  
    return get(number-1) + get(number-2);  
}
```



# Interfaces



- Klassenähnlicher Mechanismus
  - zur reinen Verhaltensspezifikation.
- Erlaubt die Definition eines benutzerdefinierten Datentyps von seiner Realisierung zu trennen
  - abstrakter Datentyp.

# Interfaces



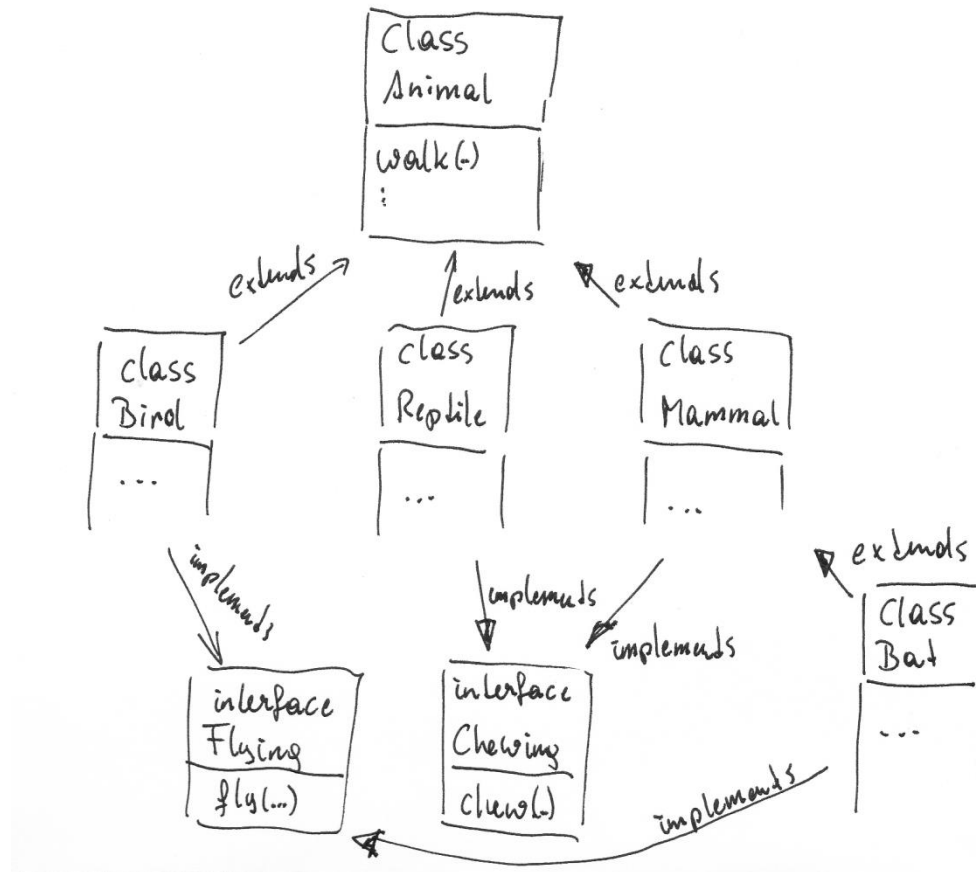
- Spezifikation via interface
- Methoden-Spezifikationen
  - beschreiben, auf welche Nachrichten ein Objekt reagiert
  - ohne Rumpf, also ohne Implementierung.
- Keine Instanzvariablen
  - Aber evt. Konstante

# Interfaces



- Der interface-Name ist in Java als Datentyp verwendbar
- Implementierung via `class`
- Vollständige Methoden
- Instanzvariablen

# Interface Example I



# Interface Example II



[Overview](#) [Package](#) **[Class](#)** [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#) *Java™ 2 Platform  
Standard Ed. 5.0*

[PREV CLASS](#) [NEXT CLASS](#) [FRAMES](#) [NO FRAMES](#) [All Classes](#)  
SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)      DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

java.lang

## Interface `Iterable<T>`

All Known Subinterfaces:

[BeanContext](#), [BeanContextServices](#), [BlockingQueue<E>](#), [Collection<E>](#), [List<E>](#), [Queue<E>](#), [Set<E>](#), [SortedSet<E>](#)

All Known Implementing Classes:

[AbstractCollection](#), [AbstractList](#), [AbstractQueue](#), [AbstractSequentialList](#), [AbstractSet](#), [ArrayBlockingQueue](#), [ArrayList](#), [AttributeList](#), [BeanContextServicesSupport](#), [BeanContextSupport](#), [ConcurrentLinkedQueue](#), [CopyOnWriteArrayList](#), [CopyOnWriteArraySet](#), [DelayQueue](#), [EnumSet](#), [HashSet](#), [JobStateReasons](#), [LinkedBlockingQueue](#), [LinkedHashSet](#), [LinkedList](#), [PriorityBlockingQueue](#), [PriorityQueue](#), [RoleList](#), [RoleUnresolvedList](#), [Stack](#), [SynchronousQueue](#), [TreeSet](#), [Vector](#)

```
public interface Iterable<T>
```

Implementing this interface allows an object to be the target of the "foreach" statement.

### Method Summary

<a href="#">Iterable&lt;I&gt;</a>	<a href="#">iterator()</a> Returns an iterator over a set of elements of type T.
-----------------------------------	---

### Method Detail

**iterator**

[Iterable<I>](#) `iterator()`

Returns an iterator over a set of elements of type T.

# Verwendung Interfaces?



- Freigabe minimaler Funktionalität eines abstrakten Datentyps
- Mehrfachvererbung
  - Graph, nicht Baum

# Math



- `Math.random()`
  - liefert Zufallszahl in  $[0,1)$
- `Math.floor()`
  - Rundet Zahl ab
- `Math.round()`
  - Rundet Zahl
- `Math.sin()` ...

# Beispiel: ASCII Sinuswelle



```
public static void main(String[] args) {  
    for (double d = 0d; d < 10; d+=0.1) {  
        double x = 60*(Math.sin(d) + 1);  
        x = Math.round(x);  
        for (int i = 0; i < x; i++) System.out.print(' ');  
        System.out.println('*');  
    }  
}
```



# Beispiel: Zufallsnamen



```
public class SimpleNameGenerator {
    public static void main(String[] args) {
        char[] v = new char[]{'a', 'e', 'i', 'o', 'u', 'y'};
        char[] c = new String("bcdfghjklmnpqrstvwxz").toCharArray();
        System.out.print(getRandomChar(v));
        System.out.print(getRandomChar(c));
        System.out.print(getRandomChar(v));
        System.out.print(getRandomChar(c));
        System.out.print(getRandomChar(c));
        System.out.print(getRandomChar(c));
        System.out.print(getRandomChar(v));
        System.out.print(getRandomChar(c));
    }

    public static char getRandomChar(char[] c) {
        int randomIndex = (int) Math.floor(c.length * Math.random());
        return c[randomIndex];
    }
}
```

# NameGenerator



- Jetzt mit Interfaces / Vererbung ...

# ESOP - Exceptions

Assoc. Prof. Dr. Mathias Lux  
ITEC / AAU

# Fehlerbehandlung & Qualitätssicherung



- Ausnahmebehandlung
  - für den Fall, dass etwas schief geht
- Zusicherungen
  - ein Mechanismus, um strukturiert über die Korrektheit eines Programms nachzudenken und Fehler frühzeitig zu entdecken

# Klassische Muster bei Fehlererkennung



- Fehlerbehandlung überfrachtet den Code
- Algorithmus vs. Fehlerbehandlung?
- Immer tiefer in Vorbedingungen.

```
Aktion1;  
if (Probleme_sind_aufgetreten1)  
    Fehlerbehandlung1;  
else {  
    Aktion2;  
    if (Probleme_sind_aufgetreten2)  
        Fehlerbehandlung2;  
    else {  
        Aktion3;  
        if (Probleme_sind_aufgetreten3)  
            Fehlerbehandlung3;  
        else { ....
```

# Klassische Muster bei Fehlerbehandlung



- Fehlerbehandlung wird nach oben gereicht.
- Aufrufer hat Verantwortung

Vgl. `System.exit(int status)`

- `status != 0` => abnormal termination
- Status wird vom Aufrufer ausgewertet

```
int methode3 () { // Ergebnis = Fehlercode Aktion;
    if (Probleme_sind_aufgetreten)
        return 1;
    return 0;
}
int methode2 () { // Ergebnis = Fehlercode
    Aktion;
    int fehler = methode3();
    if (fehler > 0) return 1;
    return 0;
}
void methode1 () {
    Aktion;
    int fehler = methode2();
    if (fehler > 0)
        System.err.println("...nicht hingehauen");
    else ...;
}
```

# Exceptions



Java erlaubt die Trennung von normalem Algorithmus und der Behandlung von Ausnahmesituationen (Exceptions):

- Programm wird einfacher zu verstehen.
- Convenience: Fehlerbehandlung wird einfacher
- Die Fehlerbehandlung wird aufgeschoben
  - Wird an „sinnvoller Stelle“ abgearbeitet
- Aufrufer haben meist erhöhtes Kontextwissen
  - können daher eher sinnvoll auf Fehler reagieren.

# Exceptions



- Eine Exception erzeugt viele Aufrufe in der Java Virtual Machine
  - Erzeugen stacktrace, usw.
- Entsprechend: Exceptions sind Ausnahme
  - kein Ersatz für Kontrollflusssteuerung



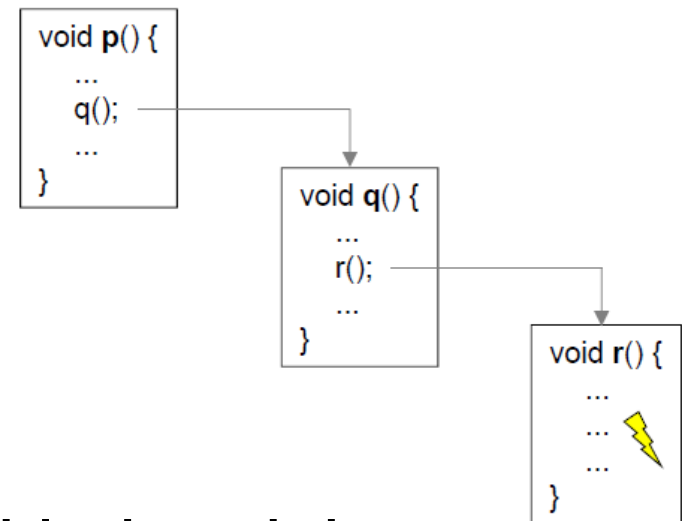
# Optimalfall für Exceptions



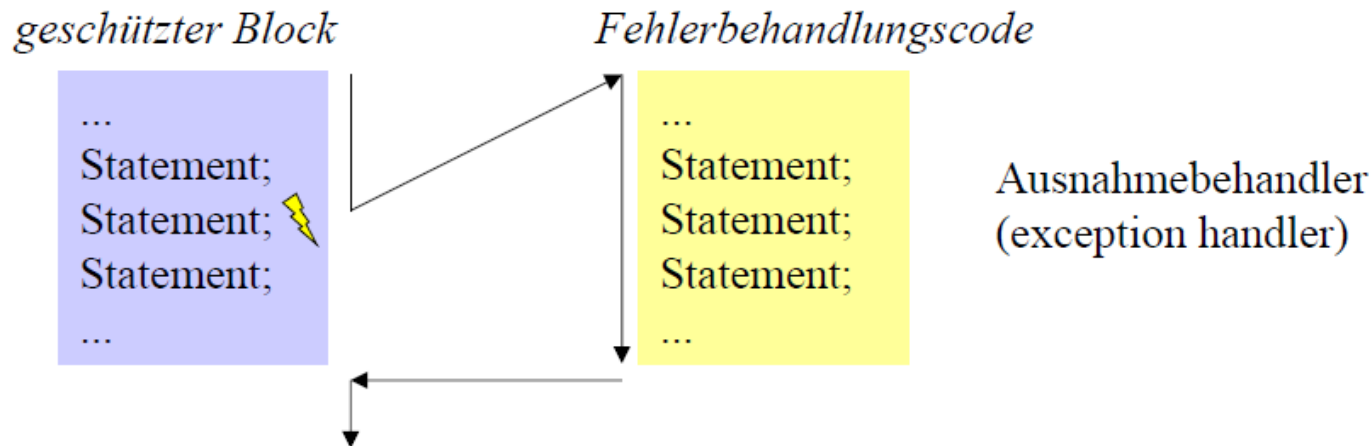
- In `r()` geht etwas schief
  - Was soll passieren?

- Lösung

- `r()` meldet Fehler an `q()`
- `q()` meldet an `p()`
- ... solange bis ihn jemand behandelt.



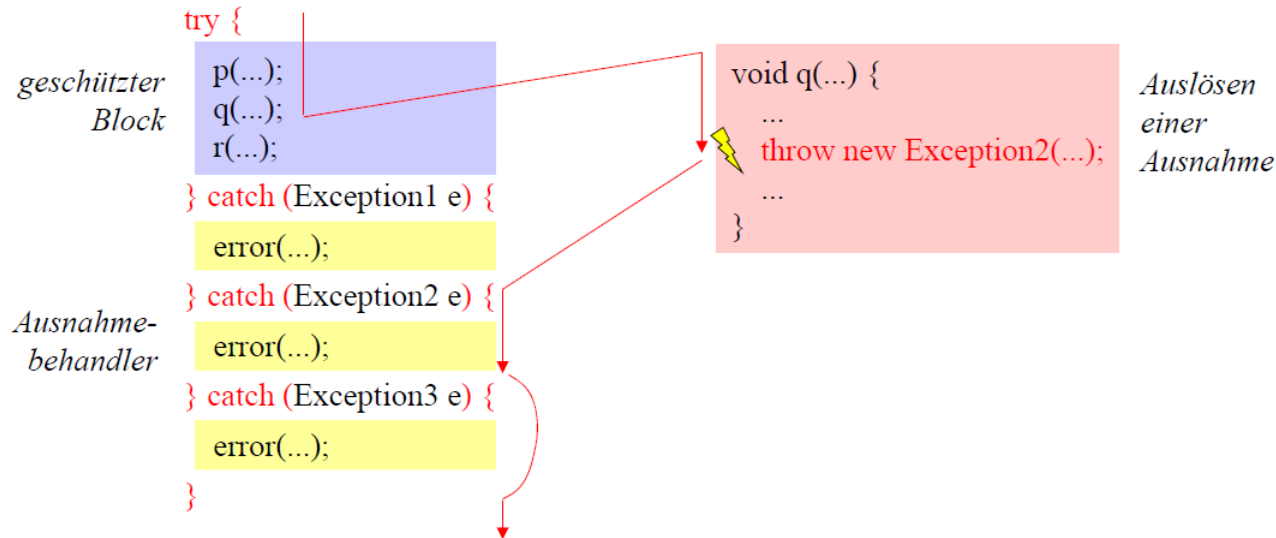
# Fehlerbehandlung in Java



Wenn im geschützten Block ein Fehler (eine Ausnahme) auftritt:

- Ausführung des geschützten Blocks wird abgebrochen
- Fehlerbehandlungscode wird ausgeführt
- Programm setzt nach dem geschützten Block fort

# Try-Anweisungen in Java



- Fehlerfreier Fall und Fehlerfälle sind sauberer getrennt
- Man kann nicht vergessen, einen Fehler zu behandeln
  - Compiler prüft, ob es zu jeder möglichen Ausnahme einen Behandler gibt

# Arten von Ausnahmen



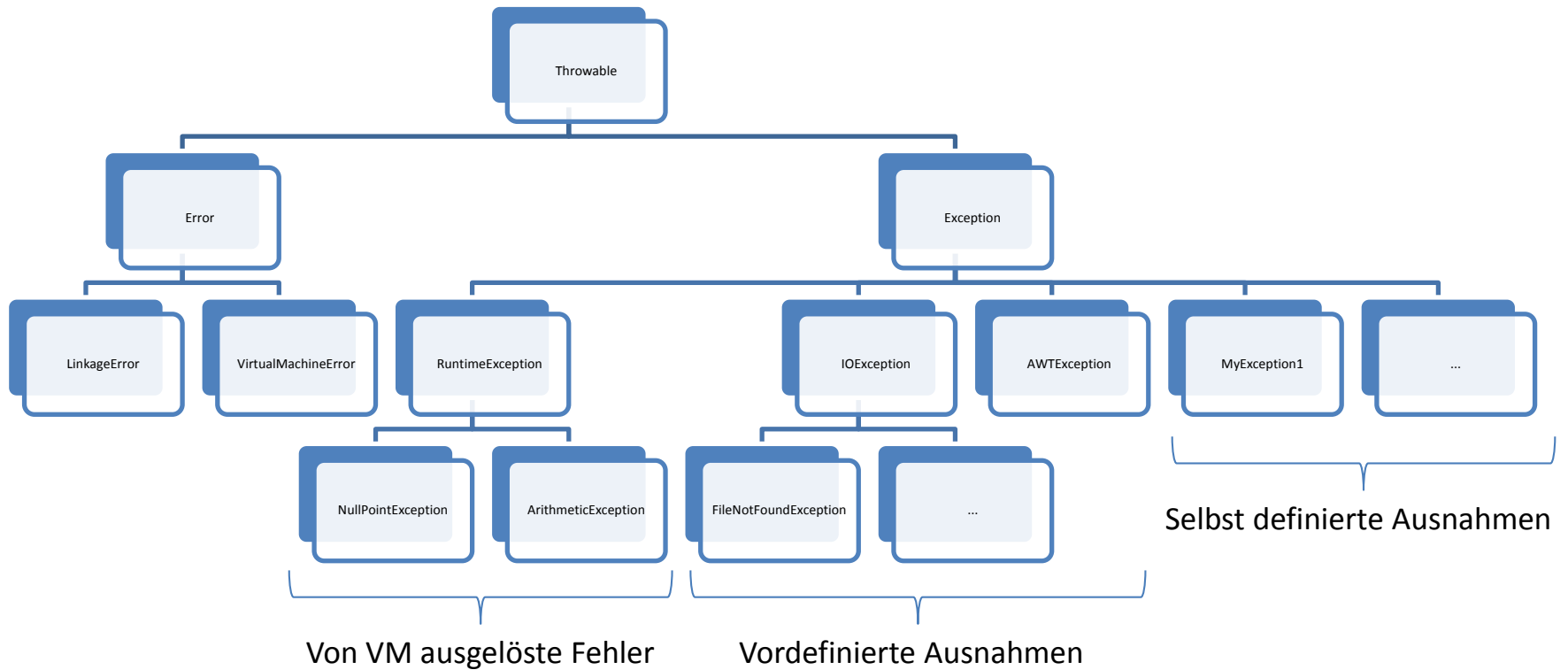
- Laufzeitfehler (Runtime Exceptions) werden von der Java-VM ausgelöst
  - Division durch 0 *ArithmeticException*
  - Zugriff über null-Zeiger *NullPointerException*
  - Indexüberschreitung *ArrayIndexOutOfBoundsException*
- Müssen nicht behandelt werden
- Bei Nichtbehandlung stürzt Programm ab
  - mit einer Fehlermeldung ...

# Arten von Ausnahmen



- Geprüfte Ausnahmen (Checked Exceptions) werden vom Benutzercode ausgelöst (throw-Anweisung)
  - vordefinierte Ausnahmen z.B. `FileNotFoundException`
  - selbst definierte Ausnahmen z.B. `MyException`
- Müssen behandelt werden.
- Compiler prüft, ob sie abgefangen werden.

# Hierarchie der Ausnahmeklassen



# Ausnahmen sind als Klassen umgesetzt.



## Fehlerinformationen stehen in einem Ausnahme-Objekt

```
class Exception extends Throwable {  
    Exception(String msg) {...} // erzeugt neues Ausnahmeobjekt mit Fehlermeldung  
    String getMessage() {...} // liefert gespeicherte Fehlermeldung  
    String toString() {...} // liefert Art der Ausnahme und gespeichert Fehlermeldung  
    void printStackTrace() {...} // gibt Methodenaufruflkette aus  
    ...  
}
```

## Eigene Ausnahmeklasse (speichert Informationen über speziellen Fehler)

```
class MyException extends Exception {  
    private int errorCode;  
    MyException(String msg, int errorCode) { super(msg); this.errorCode = errorCode; }  
    int getErrorCode() {...}  
    // toString(), printStackTrace(), ... von Exception geerbt  
}
```

# Throw-Anweisung



- Löst eine Ausnahme aus

```
throw new MyException("invalid operation", 42);
```

- "Wirft" ein Ausnahmeobjekt mit entsprechenden Fehlerinformationen
  - bricht normale Programmausführung ab
  - sucht passenden Ausnahmebehandler (catch-Block)
  - führt Ausnahmebehandler aus und übergibt ihm Ausnahmeobjekt als Parameter
  - setzt nach try-Anweisung fort, zu der der catch-Block gehört



# catch-Blöcke und finally-Block




```
try {  
    ...  
} catch (MyException e) {  
    Out.println(e.getMessage() + ", error code = ", + e.getErrorCode());  
} catch (NullPointerException e) {  
    ...  
} catch (Exception e) {  
    ...  
} finally {  
    ...  
}
```

- Passender catch-Block wird an Hand des Ausnahme-Typs ausgewählt
- catch-Blöcke werden sequentiell abgesucht
- Achtung: speziellere Ausnahme-Typen müssen vor allgemeineren stehen
- Am Ende wird (optionaler) finally-Block ausgeführt
- egal, ob im geschützten Block ein Fehler auftrat oder nicht


# Beispiel finally-Block



```
try {  
    In.open("myfile.txt");  
    ...  
    ...   
    ...  
    In.close();  
} catch (...) {  
    ...  
}
```

**falsch**

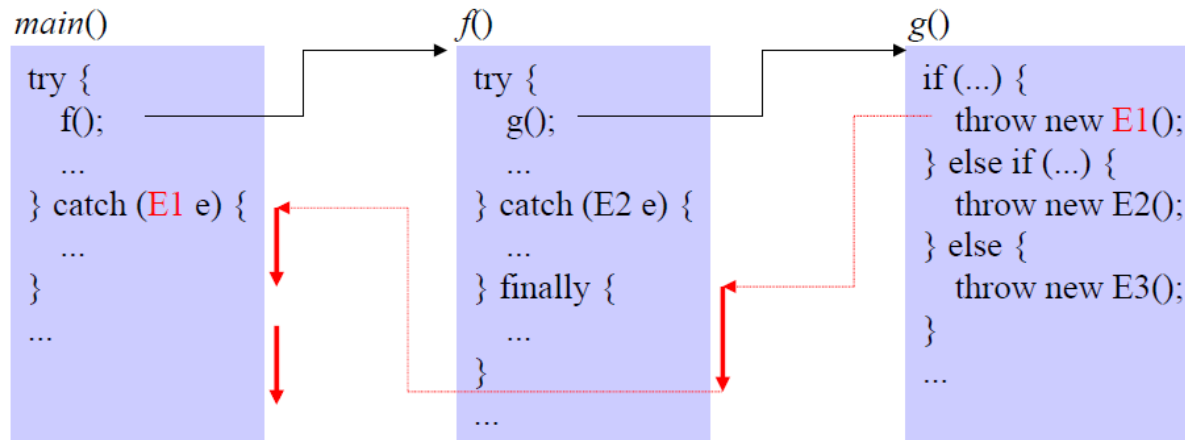
Datei wird im Fehlerfall nicht geschlossen

```
try {  
    In.open("myfile.txt");  
    ...  
    ...   
    ...  
} catch (...) {  
    ...  
} finally {  
    In.close();  
}
```

**richtig**

Datei wird auf jeden Fall geschlossen

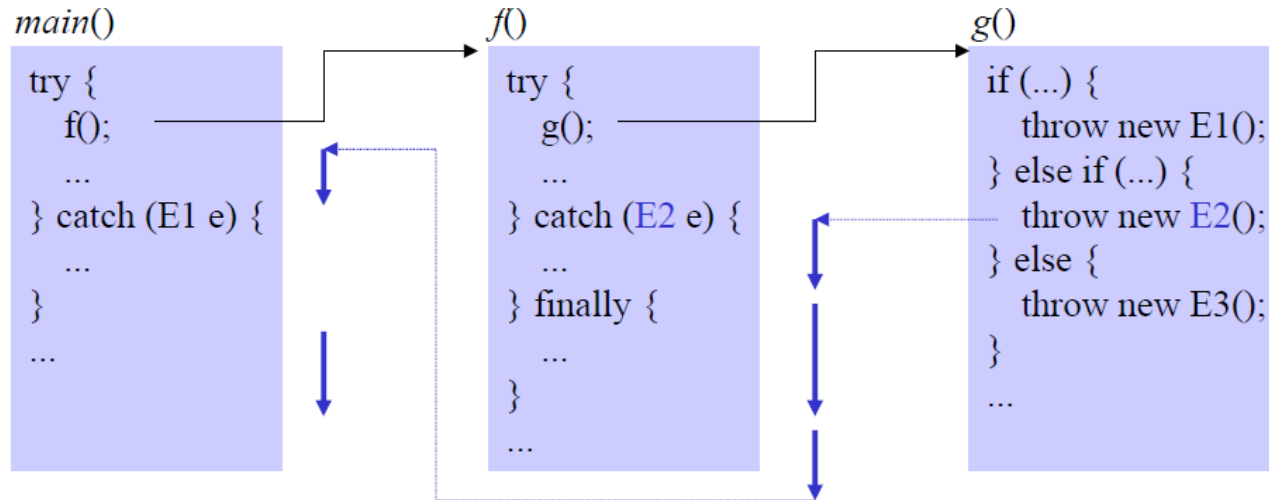
# Ablauflogik bei Ausnahmen



`throw new E1();`

- keine try-Anweisung in `g()` => bricht `g()` ab
- kein passender catch-Block in `f()` => führt finally-Block in `f()` aus und bricht `f()` dann ab
- führt catch-Block für `E1` in `main()` aus
- setzt nach try-Anweisung in `main()` fort

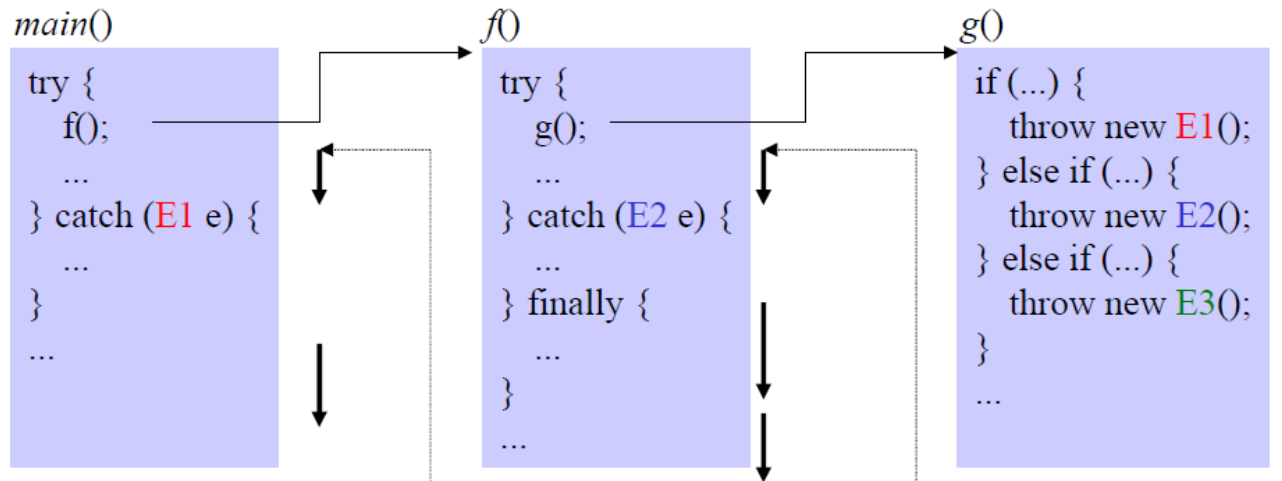
# Ablauflogik bei Ausnahmen



`throw new E2();`

- keine try-Anweisung in `g()` => bricht `g()` ab
- führt catch-Block für `E2` in `f()` aus
- führt finally-Block in `f()` aus
- setzt nach try-Anweisung in `f()` fort

# Ablauflogik bei Ausnahmen



`throw new E3();`

- Compiler meldet einen Fehler, weil E3 nirgendwo in der Ruferkette abgefangen wird
- fehlerfreier Fall
- führt `g()` zu Ende aus
- führt `try`-Block in `f()` zu Ende aus
- führt `finally`-Block in `f()` aus
- setzt nach `finally`-Block in `f()` fort

# Spezifikation von Ausnahmen im Methodenkopf



Wenn eine Methode eine Ausnahme an den Rufer weiterleitet, muss sie das in ihrem Methodenkopf mit einer *throws-Klausel* spezifizieren

# Spezifikation von Ausnahmen im Methodenkopf



```
void f() {  
    try {  
        ...  
        g();  
        ...  
    } catch (E2 e) {  
        ...  
    }  
}
```

```
void g() throws E2 {  
    try {  
        ...  
        throw new E1();  
        ...  
        throw new E2();  
        ...  
    } catch (E1 e) {  
        ...  
    }  
}
```

# Spezifikation von Ausnahmen im Methodenkopf



- Compiler weiß dadurch, dass `g()` eine E2-Ausnahme auslösen kann.
- Wer `g()` aufruft, muss daher
  - entweder E2 abfangen
  - oder E2 im eigenen Methodenkopf mit einer *throws-Klausel* spezifizieren
- **Man kann nicht vergessen, eine Ausnahme zu behandeln!**



# Daher ...



*void main()* throws E3

```
try {  
    f();  
    ...  
} catch (E1 e) {  
    ...  
}  
...
```

*void f()* throws E1, E3

```
try {  
    g();  
    ...  
} catch (E2 e) {  
    ...  
} finally {  
    ...  
}  
...
```

*void g()* throws E1, E2, E3

```
if (...) {  
    throw new E1();  
} else if (...) {  
    throw new E2();  
} else if (...) {  
    throw new E3();  
}  
...
```

# java.io.File



- Klasse für den Umgang mit Dateien
  - Verzeichnisse sind auch Dateien
- Methoden erlauben
  - Existenzprüfung, Rechteprüfung
  - Auflistung der Kind-Dateien
  - Verzeichnisse anlegen
  - uvm.

# java.io.FileReader



- Erweitert Reader (abstrakte Klasse)
- Liest Zeichen (char) aus File
  - Vgl. FileInputStream -> byte
  - Unterschied zwischen char & byte!
- Nutzung benötigt einen Buffer
  - char[] für Reader, byte[] für InputStream

# java.io.BufferedReader



- Stellt Buffer zur Verfügung
- Liefert ganze Zeilen zurück
- **ACHTUNG:** Ist abgeleitet von Reader -> char[]!

# Schreiben in Dateien?



- `FileWriter`, `BufferedWriter`
  - Nicht auf `close()` bzw. `flush()` vergessen!

Und:

- `ObjectStreamWriter`, `ObjectStreamReader`
  - Schreibt und liest Objekte von einem `InputStream`
- `GZipInputStream`, `GZipOutputStream`
  - Komprimiert I/O Streams

# ESOP - Pakete & Collections

Assoc. Prof. Dr. Mathias Lux  
ITEC / AAU

# Pakete



## Sammlung zusammengehöriger Klassen

- Bring mehr Ordnung in Programme
- Vermeidung von Namenskonflikten
- bessere Zugriffskontrolle



# Beispiele für Pakete



- `java.lang`
  - `System`, `String`, `Integer`, `Character`, `Object`, `Math`, ...
- `java.io`
  - `File`, `InputStream`, `OutputStream`, `Reader`, `Writer`, ...
- `java.awt`
  - `Button`, `CheckBox`, `Frame`, `Color`, `Cursor`, `Event`, ...
- `java.util`
  - `ArrayList`, `Hashtable`, `BitSet`, `Stack`, `Vector`, ...



# Anlegen von Paketen



- Anweisung in der ersten Zeile der Datei
- Fehlt die package-Zeile dann gehört die Klasse zu namenlosen Standardpaket.

Datei *Circle.java*

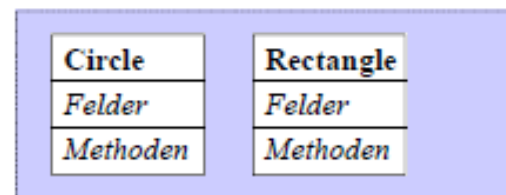
```
package graphics;  
class Circle {  
    ...  
}
```

Datei *Rectangle.java*

```
package graphics;  
class Rectangle {  
    ...  
}
```

← 1. Zeile der Datei

Paket *graphics* enthält die Klassen *Circle* und *Rectangle*

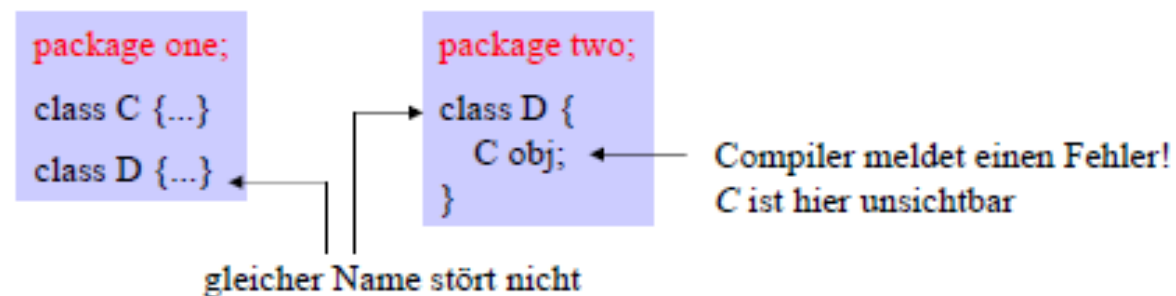


— Paket *graphics*

# Pakete als Sichtbarkeitsgrenzen



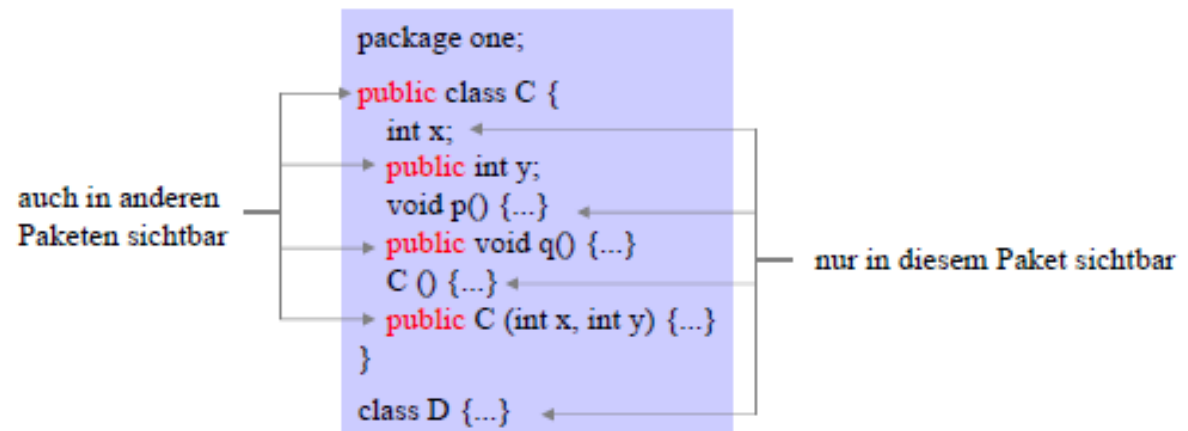
- Was in einem Paket deklariert ist, ist in einem anderen unsichtbar.
  - Verschiedenen Paketen & gleiche Namen möglich
  - Name muss nicht global eindeutig sein.



# Export von Namen



- Namen können mit dem Zusatz `public` exportiert werden
  - sie sind dann in anderen Paketen sichtbar
  - `public`-Felder und -Methoden werden nur dann exportiert, wenn die Klasse selbst `public` ist.
  - Lokale Variablen und Parameter können nicht exportiert werden.



# Import von Klassennamen



- Exportierte Klassennamen können in anderen Paketen importiert werden
  - gezielten Import der Klasse
  - Import aller public-Klassen eines Pakets
  - Qualifikation mit dem Paketnamen

```
package myPack;  
import graphics.Circle;  
import one.C;  
class MyClass {  
    Circle c;  
    ...  
}
```

```
package myPack;  
import graphics.*;  
class MyClass {  
    Circle c;  
    Rectangle r;  
    ...  
}
```

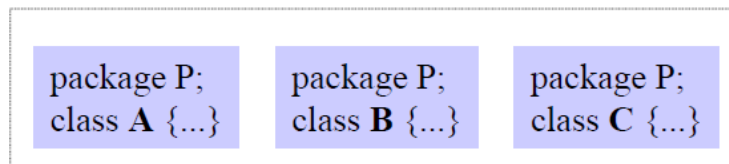
```
package myPack;  
class MyClass {  
    graphics.Circle c1;  
    java.awt.Circle c2;  
    ...  
}
```

# Pakete und Verzeichnisse

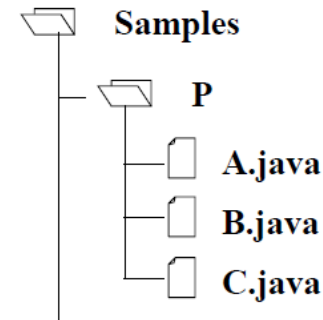


**Pakete werden auf Verzeichnisse abgebildet, Klassen auf Dateien**

Klasse C  $\Rightarrow$  Datei C.java  
Paket P  $\Rightarrow$  Verzeichnis P



*Paket P*



**Übersetzung und Ausführung mit dem JDK**

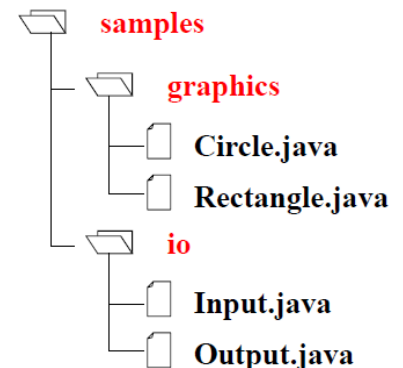
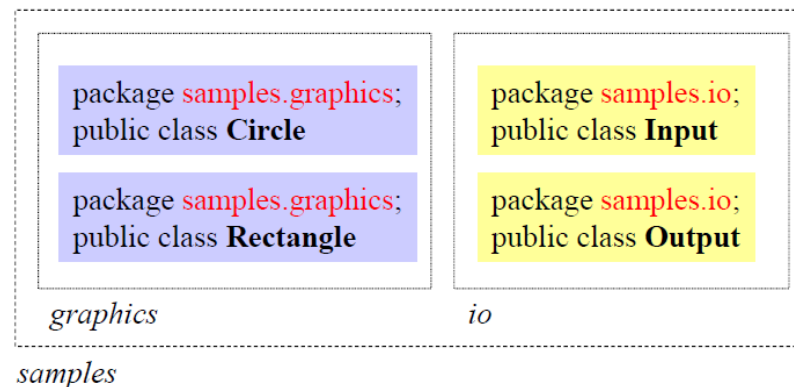
```
cd C:\Samples
javac P/A.java

java P/A } beides möglich
java P.A }
```

# Geschachtelte Pakete



- Pakete können zu größeren Paketen zusammengefasst werden.
  - `import samples.graphics.Circle;` importiert die Klasse *Circle*
  - `import samples.graphics.*` importiert alle public-Klassen aus *samples.graphics*
  - `import samples.*;` importiert alle public-Klassen aus *samples* (nicht aus *samples.graphics*)



# Information Hiding (Erweiterung)



- Sichtbarkeitsattribute für Felder und Methoden

```
private    int a; // nur in Klasse sichtbar, in der Element deklariert wurde
           int b; // nur im Paket sichtbar, in dem Element deklariert wurde
protected int c; // sichtbar:
            // - in der deklarierenden Klasse
            // - in deren Unterklassen (auch in anderen Paketen!)
            // - im deklarierenden Paket
public     int d; // auch in anderen Paketen sichtbar, wenn importiert
```

# Sichtbarkeitsattribute



```
private    int a; // nur in Klasse sichtbar, in der Element deklariert wurde
           int b; // nur im Paket sichtbar, in dem Element deklariert wurde
protected int c; // sichtbar:
           // - in der deklarierenden Klasse
           // - in deren Unterklassen (auch in anderen Paketen!)
           // - im deklarierenden Paket
public     int d; // auch in anderen Paketen sichtbar, wenn importiert
```

```
package one;
public class C {
    private int a;
    int b;
    protected int c;
    public int d;
}
public class D {
    ...
}
```

```
package two;
import one.C;
public class E extends C {
    C x = new C();
    x.a = ...;
    x.b = ...;
    x.c = ...;
    x.d = ...;
}
public class F {
    ...
}
```



# Java Collections Framework



- Collection == Objekt, das andere Objekte gruppiert
- Collections werden benutzt um mit Daten umzugehen
  - speichern, suchen, ändern, verteilen, ...
- Ein Framework von
  - Interfaces
  - Implementierungen
  - Algorithmen



# Collection Interfaces



- Set ... Menge
  - kann ein Element nur einmal enthalten
- List ... Liste, Sequenz
  - hat Ordnung, kann Duplikate enthalten
- Queue ... Warteschlange
  - zum Abarbeiten
- Map ... Zuordnung Name -> Wert
  - Namen sind in einer Menge (s.o.)

# Interface Set



- Hinzufügen und Löschen
  - add(..), addAll(..), remove(..), removeAll(..)
- Überprüfen
  - contains(..), containsAll(..)
- Und mehr ...
  - size(), clear()

# Addendum: Autoboxing



- Collections können nur mit Objekten umgehen:
  - `mySet.add(new Integer(5));`
- Basisdatentypen werden automatisch in Objekte verpackt:
  - `mySet.add(5); // gleich w.o.`

# Collections: Set



- Beispiele
  - Set intersection - Schnittmenge
  - Set union - Vereinigungsmenge

# Generics



- Vorgehensweise bei Collections:

```
List list = new ArrayList();  
list.add("hello");  
String s = (String) list.get(0);
```

- Besser mit Generics

```
List<String> list = new ArrayList<String>();  
list.add("hello");  
String s = list.get(0); // no cast
```

# Traversing Collections



- for-each

```
for (Object o : collection)
    System.out.println(o);
```

- Iterator

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

# Beispiel: Sets



- Jetzt mit Generics & Iteratoren ...



# Listen



- ArrayList ... Array als Datenbasis
  - random access, langsame Vergrößerung
- LinkedList ... doppelte verkettete Liste
  - Iterator schnell, keine Re-Allokationen

# Beispiele mit Listen



- Sortierung einer Liste
- Sortierung mit Comparator

# Listenoperationen



- **sort**
  - sorts a List using a merge sort algorithm, which provides a fast, stable sort. (A *stable sort* is one that does not reorder equal elements.)
- **shuffle**
  - randomly permutes the elements in a List.
- **reverse**
  - reverses the order of the elements in a List.
- **rotate**
  - rotates all the elements in a List by a specified distance.
- **swap**
  - swaps the elements at specified positions in a List.
- **replaceAll**
  - replaces all occurrences of one specified value with another.
- **fill**
  - overwrites every element in a List with the specified value.
- **copy**
  - copies the source List into the destination List.
- **binarySearch**
  - searches for an element in an ordered List using the binary search algorithm.
- **indexOfSubList**
  - returns the index of the first sublist of one List that is equal to another.
- **lastIndexOfSubList**
  - returns the index of the last sublist of one List that is equal to another.

# Maps



- HashMap
  - als wichtiger Vertreter und Beispiel
  - HashSet als Basis (schneller Zugriff)
  - keine zugesicherte Ordnung
- TreeMap
  - TreeSet als Basis -> geordnet
- LinkedHashMap
  - LinkedHashMapSet -> hash + geordnet

# HashMap

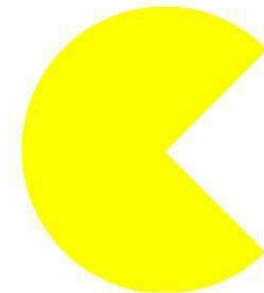


- Beispiel
  - `HashMap<String, List<String>> student2course`

# Organisatorisches



- Wir haben noch 5 Einheiten
  - 1 x Tutoren, 1x Threading,
  - 1x Skriptsprache, 1x Beispielprojekt, 1x Wiederholung
- Umfragen
  - Eine Skriptsprache: Python oder JavaScript?
  - Wiederholung: Welche/s Themenbereich/e
  - Beispielprojekt: Datenanalyse, Datenmodell, Algorithmus, Spiel, ...



# ESOP - Innere Klassen & Threads

Assoc. Prof. Dr. Mathias Lux  
ITEC / AAU

# Innere Klassen



- Drei Typen von verschachtelten Klassen
  - Innere Klassen
  - Lokale Klassen
  - Anonyme Klassen

```
public class OuterClass {  
    private int number;  
  
    class InnerClass {  
        int innerNumber = 42;  
    }  
}
```

} Innere Klasse



# Warum Innere Klassen?



- Gruppierung von Elementen
  - Klassen, die nur an einer Stelle benutzt werden.
- Lesbarer Code
  - Code ist da, wo er benutzt wird.

# Warum Innere Klassen?




## Erweiterung Geheimnisprinzip

- Ist B eine innere Klasse von A, dann kann B auf *private members* von A zugreifen.
- A muss weniger preisgeben.


```
// with inner class
public class A {
    private int x, y;

    class B {
        int add() {
            return x + y;
        }
    }
}
```



```
// without inner class
public class A {
    protected int x, y;
}

class B {
    A a;
    int add() {
        return a.x + a.y;
    }
}
```



# Shadowing



```
public class ShadowTest {
    public int x = 0;
    class FirstLevel {
        public int x = 1;
        void methodInFirstLevel(int x) {
            System.out.println("x = " + x);
            System.out.println("this.x = " + this.x);
            System.out.println("ShadowTest.this.x = " +
                ShadowTest.this.x);
        }
    }
    public static void main(String... args) {
        ShadowTest st = new ShadowTest();
        ShadowTest.FirstLevel fl = st.new FirstLevel();
        fl.methodInFirstLevel(23);
    }
}
```

```
// output:
x = 23
this.x = 1
ShadowTest.this.x = 0
```



- Innere Klassen sind an Instanzen gebunden, nicht Klassen.
- Instanzen innerer Klassen benötigen eine Instanz der äußeren Klasse.

*Ausnahme: static nested classes*

# Lokale Klassen



- Lokale Klassen sind innere Klassen
  - die in einer Methode definiert werden
- Lokale Klassen können auf Variablen und Methoden der äußeren Klasse zugreifen
  - lokale Variablen müssen aber *final* sein

# Lokale Klassen



```
public class Greet {  
  
    interface HelloThere {  
        public void greet();  
    }  
  
    public void greetInEnglish() {  
        final String name = "Mathias";  
        class EnglishHelloThere implements HelloThere {  
            public void greet() {  
                System.out.println("Hello " + name);  
            }  
        }  
        HelloThere myGreeting = new EnglishHelloThere();  
        myGreeting.greet();  
    }  
  
    public static void main(String[] args) {  
        Greet g = new Greet();  
        g.greetInEnglish();  
    }  
}
```

- Interface wird in Klasse definiert.
- Lokale Klasse weil in der Methode.

# Anonyme Klassen



- Anonyme Klassen sind lokale Klassen ohne Namen
- Instanziierung und Deklaration fallen zusammen
  - Code wird kompakter.

# Anonyme Klassen



```
public class GreetAnon {  
  
    interface HelloThere {  
        public void greet();  
    }  
  
    public void greetInEnglish() {  
        final String name = "Mathias";  
        HelloThere myGreeting = new HelloThere() {  
            public void greet() {  
                System.out.println("Hello " + name);  
            }  
        };  
        myGreeting.greet();  
    }  
  
    public static void main(String[] args) {  
        GreetAnon g = new GreetAnon();  
        g.greetInEnglish();  
    }  
}
```

- Lokale Klasse anonym
- Oft in Verwendung oft um Interfaces dynamisch zu implementieren

# Beispiel KeyListener.



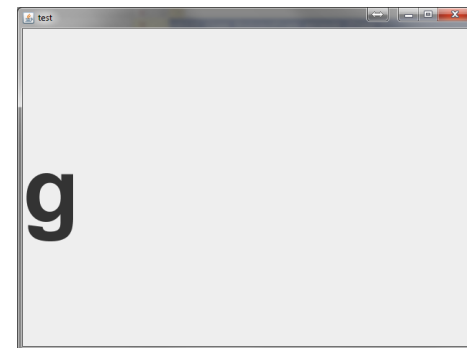
```
public class KeyboardFrame extends JFrame {
    JLabel label = new JLabel();

    public KeyboardFrame() throws HeadlessException {
        super("test");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(640, 480);
        label.setFont(label.getFont().deriveFont(128f));
        add(label, BorderLayout.CENTER);

        addKeyListener(new KeyAdapter() {
            public void keyReleased(KeyEvent keyEvent) {
                if (!keyEvent.isActionKey())
                    label.setText(Character.toString(
                        keyEvent.getKeyChar()));
            }
        });
    }

    public static void main(String[] args) {
        KeyboardFrame kf = new KeyboardFrame();
        kf.setVisible(true);
    }
}
```

- Listener-Klassen als Beispiel
  - Behandeln Events von außen
  - Sind implementierungsspezifisch
- Anonyme Klasse für
  - genau diesen Frame
  - genau diese Aktion





# Threads

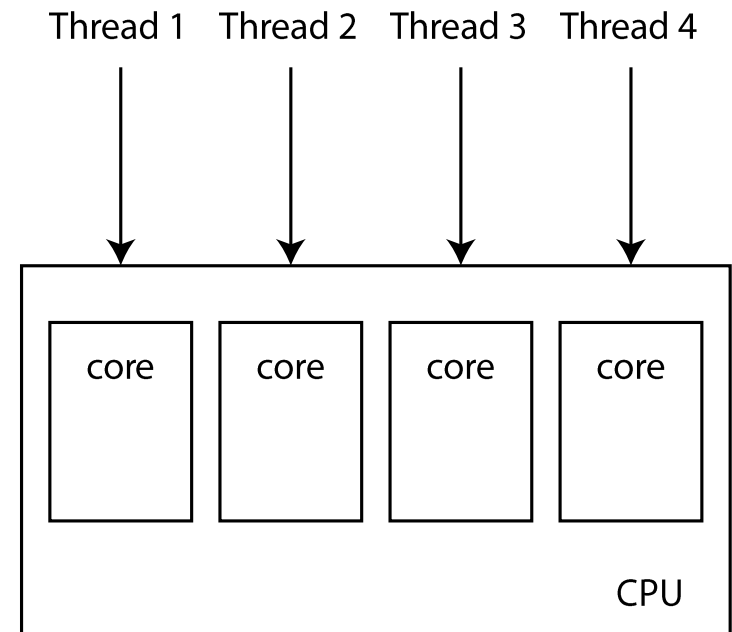
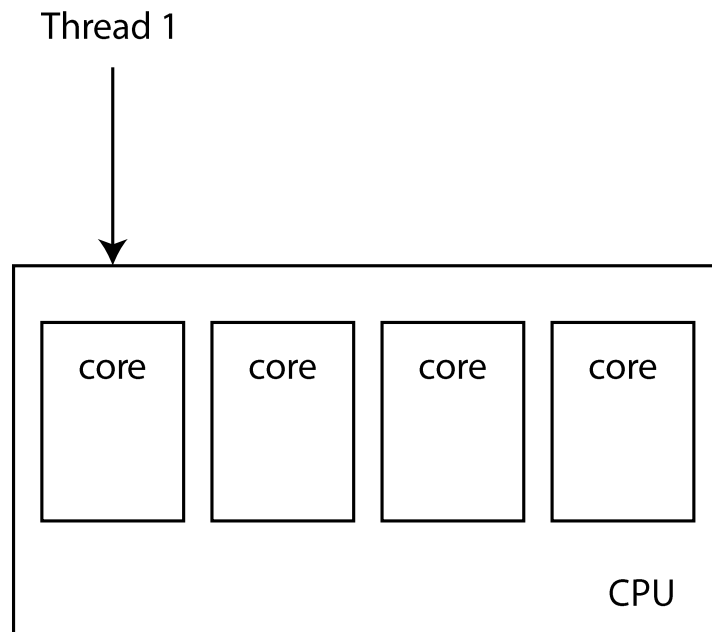


- Prozesse
  - Eigene Umgebung,
  - Stichwort Inter-Process Communication
- Thread
  - Teilen sich eine Umgebung (Variablen, etc.)
  - „light weight processes“

# Threads



- Single threaded vs. multi threaded



# Beispiele



- Datenverarbeitung
  - Es gibt  $n$  Datenpakete und  $k$  Threads
  - Jeder Thread bearbeitet  $n/k$  Datenpakete
- Threads in Computerspielenspielen
  - Zeichnen der Oberfläche
  - Berechnung der KI
  - Netzwerkkommunikation

# Threads in Java



- Jeder Thread
  - ist in einem Thread-Objekt gekapselt.
  - wird mit `start()` gestartet.
  - danach wird asynchron die `run()` Methode ausgeführt.

# Threads in Java



- Implementierung des Runnable Interface
  - Nutzung als Thread durch Konstruktor `new Thread(Runnable)`
- Ableitung der Thread Klasse
  - Überschreiben von `run()`



# Beispiel



```
public class ThreadExample implements Runnable {
    String text;
    int count = 5000;

    public ThreadExample(String text) {
        this.text = text;
    }

    public static void main(String[] args) {
        new Thread(new ThreadExample("-")).start();
        new Thread(new ThreadExample("O")).start();
        new Thread(new ThreadExample("/")).start();
        new Thread(new ThreadExample("|")).run();
    }

    @Override
    public void run() {
        while (count-- > 0) System.out.print(text);
    }
}
```

- Vier Threads
- Drei werden mit `start()` gestartet
  - Warum?
- Ergebnis?



# Problem: Blockierung



- Im Fall von geteilten Ressourcen
  - Ergebnisliste, Festplatte, Variable, ...
- Fall A: Ressource blockiert
  - Festplatte kann immer nur von einem Thread gelesen werden.
- Fall B: Nicht Thread-Safe
  - Es kann gleichzeitig auf Ressource zugegriffen werden.



# Beispiel: Threads



```
public class ThreadConcurrencyExample implements Runnable {
    boolean up; // increment or decrement
    int count = 1000; // each runs # times
    static int sum; // shared variable

    public ThreadConcurrencyExample(boolean up) {
        this.up = up;
    }

    public void run() {
        while (count-- > 0)
            if (up)
                sum++;
            else
                sum--;
    }

    public static void main(String[] args) {
        boolean upMe = true;
        LinkedList<Thread> t = new LinkedList<Thread>();
        for (int i = 0; i < 10; i++) { // creating 10 threads
            Thread thread = new Thread(new ThreadConcurrencyExample(upMe));
            upMe = !upMe;
            thread.start();
            t.add(thread);
        }
        // making sure to wait for them to end:
        for (Iterator<Thread> iterator = t.iterator(); iterator.hasNext(); ) {
            try {
                iterator.next().join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("sum = " + sum);
    }
}
```

- Threads greifen gleichzeitig auf Variable zu
  - n/2 Threads inkrementieren
  - n/2 Threads dekrementieren
- Das Ergebnis sollte 0 sein.

# Threads kontrollieren



- `Thread.sleep(ms)`
  - Lässt aktuellen Thread *ms* schlafen
- `t.join()`
  - Stellt aktuellen Thread an Thread *t* an.

# Threads synchronisieren



- Keyword *synchronized*
  - Schützt Code-Teil
  - Auf Basis eines Monitor-Objekts

```
boolean up;           // increment or decrement
int count = 1000;    // each runs # times
static int sum;      // shared variable
static Object monitor = new Object();

public ThreadConcurrencyExample (boolean up) {
    this.up = up;
}

public void run() {
    while (count-- > 0)
        synchronized (monitor) {
            if (up)
                sum++;
            else
                sum--;
        }
}
```

# Threads synchronisieren



- Blockierende Objekte
  - Sind thread-safe
  - Blockieren durch interne Mechanismen
- Beispiele
  - Vector, HashTable, StringBuffer
  - BlockingQueue

	<i>Throws exception</i>	<i>Special value</i>	<i>Blocks</i>	<i>Times out</i>
<b>Insert</b>	<a href="#">add(e)</a>	<a href="#">offer(e)</a>	<a href="#">put(e)</a>	<a href="#">offer(e, time, unit)</a>
<b>Remove</b>	<a href="#">remove()</a>	<a href="#">poll()</a>	<a href="#">take()</a>	<a href="#">poll(time, unit)</a>
<b>Examine</b>	<a href="#">element()</a>	<a href="#">peek()</a>	<i>not applicable</i>	<i>not applicable</i>

# Organisatorisches



- Skriptsprache
  - JavaScript
- Beispielprojekt
  - Spiel
- Wiederholung
  - Arrays, Interfaces und Vererbung, java.io

# ESOP - JavaScript

Assoc. Prof. Dr. Mathias Lux  
ITEC / AAU

# Organisatorisches



- Fragenkatalog
  - 2 Kapitel online (Moodle)
  - .. Rest in Arbeit

# JavaScript: Anweisungen



- JavaScript verwendet ähnliche Anweisungen wie Java:

```
var myNumber = 42;
```



# JavaScript Interpreter



- Programmtext wird vom Interpreter gelesen.
  - Schritt für Schritt ausgeführt
- Interpreter verwaltet Speicher, Daten, Verbindungen, I/O, etc.
- Es gibt keinen Schritt zwischen Verfassen des Programms und der Ausführung.

# JavaScript ausführen



- Jeder Browser hat einen JavaScript-Interpreter eingebaut.
- JavaScript im WWW wird
  - im Rahmen einer HTML Datei definiert,
  - im Kopf einer HTML Datei verlinkt und
  - Auf „Events“ hin ausgeführt.
- Jeder Browser hat eine JavaScript-Konsole
  - zB. F12 in Chrome, `console.log(...)` als Befehl

# Variablen



- Deklaration

```
var myNumber;
```

- Zuweisung

```
myNumber = 42;
```

- Gleichzeitig ..

```
var myNumber = 42;
```

```
var myString = "Hello World!"
```

# Dynamische Typisierung



```
var myNumber = 42;
```

```
var myString = "Hello World!"
```

```
myString = myString + 10
```

- Typ einer Variable wird zur Laufzeit ermittelt.
  - <http://jsfiddle.net/qstzejwr/>

# Assignments and Operators



- Grundsätzliche Struktur

```
VarName = Expression;
```

- Dieselben Operatoren wie in Java

```
var myNumber = 42 * 3;
```

- Bool'sche Variablen

```
myBool = yes | no
```

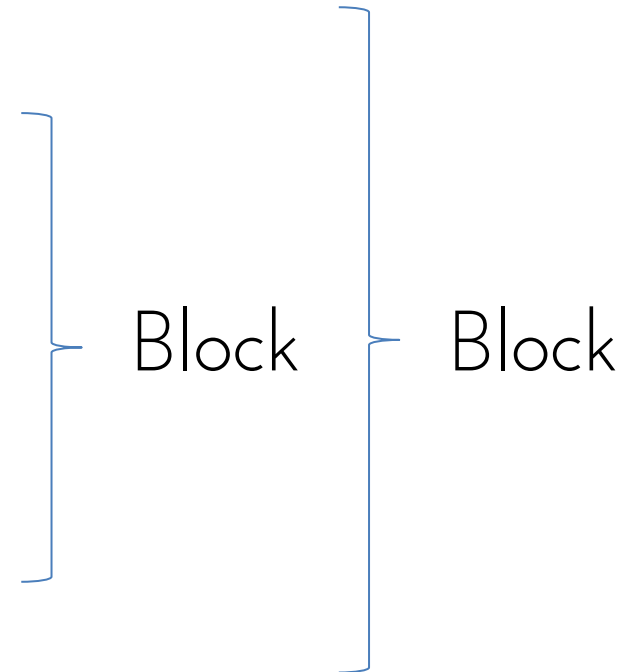
- Strings

```
myString = "Hello " + "World!"
```

# Blocks



```
var myNumber = 42;  
function squareIt() {  
    var num = 8  
    num = num*num;  
    return num;  
}  
myNumber = squareIt();
```



# Scope



```
var myNumber = 42;
```

```
function squareIt() {
```

```
  var num = 8
```

```
  num = num*num;
```

```
  return num;
```

```
}
```

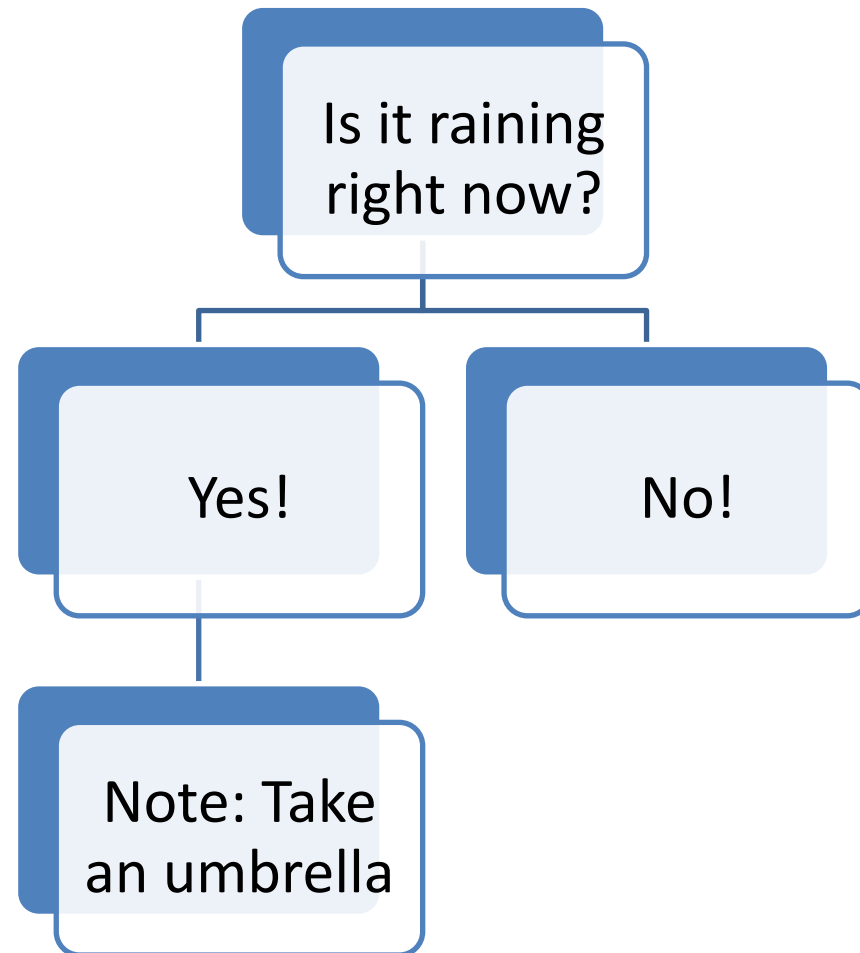
```
myNumber = num;
```



num is not defined  
in this scope!

<http://jsfiddle.net/oLwsaz2L/>

# Bedingungen





# Bedingungen



```
var myNumber = 12;  
if (myNumber < 10) {  
    console.log("It's small!");  
} else {  
    console.log("It's big!");  
}
```

<http://jsfiddle.net/j3sdhx08/>

# Bedingungen: Switch ...



```
var myNumber = 2
switch(myNumber)
{
case 1:
  console.log("It's case 1!");
  break;
case 2:
  console.log("It's case 1!");
  break;
default:
  console.log("It's neither case 1 nor case 2!");
  break;
}
```

# Bedingungen: “Short IF”



```
var myNumber = 1;
```

```
var myString =
```

```
    myNumber > 0 ? "positive" : "negative";
```

```
console.log(myString);
```

# Control Structures: Do ... While



```
var myNumber = 10;  
do {  
    myNumber -= 1;  
} while (myNumber > 0)  
console.log(myNumber);  
// output is "0"
```

# Control Structures: While ...



```
var myNumber = 10;  
while (myNumber > 0) {  
    myNumber -= 1;  
}  
console.log(myNumber);  
// output is "0"
```

# Control Structures: FOR ...



```
var myNumber = 10;  
for (var i = 0; i < 10; i++) {  
    myNumber -= 1;  
}  
console.log(myNumber);  
// output is "0"
```

# Funktionen



```
function doubleIt(snippet) {  
    return snippet + " " + snippet;  
}
```

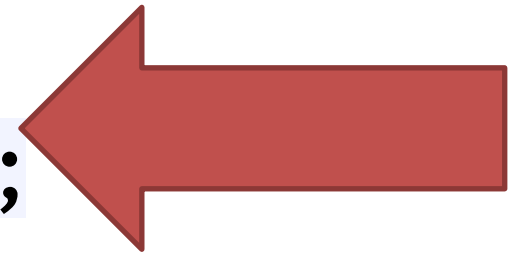
```
var myText = "Ding!";  
myText = doubleIt(myText)  
console.log(myText);
```

<http://jsfiddle.net/LbLh54xp/>

# Functions



```
function doubleIt(snippet) {  
    return snippet + " " + snippet;  
}  
var myText = "Ding!";  
console.log(doubleIt(myText));
```





# Functions



```
function doubleIt(snippet) {  
    return snippet + " " + snippet;  
}  
var myText = "Ding!";  
console.log(doubleIt(doubleIt(myText)));  
// What is the output?
```



# Funktionen: Rekursion



```
// Recursive function:  
function half(param) {  
  console.log(param);  
  if (param > 1)  
    return half(param/2);  
  else  
    return param;  
}  
var myNum = 20;  
console.log("Result = " + half(myNum));
```

# Datentypen und Strukturen



- JavaScript hat folgenden Datentypen
  - Number
  - String
  - Boolean
  - Null
  - Undefined
  - Object

# Datentypen und Strukturen: Arrays



```
arr = new Array(3);
```

```
arr[0] = 3;
```

```
arr[2] = 2;
```

```
arr[1] = arr[0] - arr[2];
```

```
console.log(arr[1]);
```

# Objekte in JavaScript



```
var person1 = {  
  firstName: "John",  
  lastName: "Doe",  
  age: 50,  
  eyeColor: "blue"  
};
```

# Objekte in JavaScript



```
var person2 = new Object();  
person.firstName = "John";  
person.lastName = "Doe";  
person.age = 50;  
person.eyeColor = "blue";
```

# Objekte in JavaScript: Konstruktor



```
function person(first, last, age, eyecolor) {  
  this.firstName = first;  
  this.lastName = last;  
  this.age = age;  
  this.eyeColor = eyecolor;  
}  
var myFather = new person("John", "Doe", 50, "blue");  
var myMother = new person("Sally", "Rally", 48, "green");
```

# Objekte in JavaScript



```
var person = {fname:"John", lname:"Doe", age:25};  
  
person.firstname + " is " + person.age + " years old."  
  
person["firstname"] + " is " + person["age"] + " years old."  
  
for (x in person) {  
    txt += person[x];  
}
```



# JavaScript im Browser



- Mit JavaScript schreibt man Programme, die in einem WWW Browser laufen.
  - Alle bekannten Browser unterstützen JavaScript
  - Sollte Interaktivität von Webseiten ermöglichen

# JavaScript



- Typische Szenarien
  - HTML Dokumente ändern und erweitern
  - Auf Events reagieren
  - Eingaben validieren
  - Eigenschaften der Zielplattform auslesen
- JavaScript-Bibliotheken
  - Hauptsächlich werden zZ. Bibliotheken benutzt.

# JavaScript: Geschichte



- 1995: developed for Netscape 2
- 1996: Microsoft dialect JScript for Internet Explorer 3
- 1997: ECMAScript (specification handed in for standardization)
- 1999: ECMA-262 fourth edition

# JavaScript: Erstes Beispiel



```
<html>
```

```
<body>
```

```
  <script type="text/javascript">
```

```
    document.write("Hello World!");
```

```
    alert("Hello World Alert");
```

```
  </script>
```

```
</body>
```

```
</html>
```

# JavaScript Code in HTML



- HTML Head und Body
  - Code wird ausgeführt, wenn genug für die Interpretation geladen ist.
  - Kann vor Abschluss des Ladens ausgeführt werden.
- JavaScript Funktionen
  - werden bei Aufruf ausgeführt.
- JavaScript kann an “Events” angehängt werden
  - wird dann bei Auftreten des Events ausgeführt.

# JavaScript: Second Example



```
<!DOCTYPE html>
<html>
<head lang="en">
  <title>Test</title>
  <script language="JavaScript">
    function logDate() {
      var node = document.getElementById("currentDate");
      node.innerHTML = " " + new Date().toDateString();
      console.log(new Date().toDateString())
    }
  </script>
</head>
<body onload="logDate()">
  <h1>And here is our test</h1>
  <p id="currentDate">...</p>
</body>
</html>
```

# Beispiel: Password-Generator



- siehe Dateien ...

# Mehr über JavaScript?



- Sehr gutes Tutorial:  
<http://www.w3schools.com/js/>
- JavaScript-Bibliotheken
  - <http://jquery.com/>
  - <http://prototypejs.org/>
- IDE: JetBrains WebStorm
- Ohne Browser: <http://nodejs.org/>



# Python



- Bessere Wahl für „nicht-Web-Scripting“
- Bibliotheken für viele Szenarien
  - NumPy, Pillow, SciPy, ...
- Stabil und schnell
- Tutorials
  - JetBrains PyCharm Educational Edition
  - <https://docs.python.org/2/tutorial/>

# ESOP - Wiederholung: Arrays, Interfaces und Vererbung, java.io

Assoc. Prof. Dr. Mathias Lux  
ITEC / AAU

# Organisatorisches



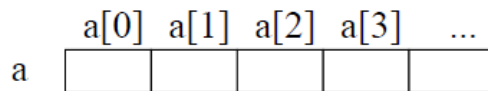
- Fragenkatalog
  - weitere Kapitel online (Moodle)
  - .. Rest in Arbeit

# Arrays



- Zusammenfassung Daten gleichen Typs
- Arrays haben fixe Länge
  - Bei Erzeugung festgelegt
- Array Variablen sind Referenz-Variablen
  - In Java! Vgl. int, float, etc. -> Basistypen
- Zugriff erfolgt über Index
  - Erstes Element hat Indexzahl 0.

# Eindimensionale Arrays



- Name *a* bezeichnet das gesamte Array
- Elemente werden über Indizes angesprochen (z.B. *a[3]*)
- Indizierung beginnt bei 0
- Elemente sind "namenlose" Variablen

## Deklaration

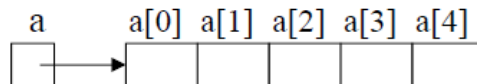
```
int[] a;  
float[] b;
```

- deklariert ein Array namens *a* (bzw. *b*)
- seine Elemente sind vom Typ *int* (bzw. *float*)
- seine Länge ist noch unbekannt

## Erzeugung

```
a = new int[5];  
b = new float[10];
```

- legt ein neues *int*-Array mit 5 Elementen an (aus dem Heap-Speicher)
- weist seine Adresse *a* zu



Array-Variablen enthalten in Java Zeiger auf Arrays!  
(Zeiger = Speicheradresse)

# Zugriff auf Arrays



- Arrayelemente werden wie Variablen benutzt
- Index kann ein ganzzahliger Ausdruck sein
- Laufzeitfehler, falls Array noch nicht erzeugt wurde
- Laufzeitfehler, falls Index  $< 0$  oder  $\geq$  Arraylänge
- *length* ist ein Standardoperator
- Liefert Anzahl der Elemente

```
a[3] = 0;  
a[2*i+1] = a[i] * 3;
```

```
int len = a.length;
```

# Beispiel



```
public class ArrayExample {
    public static void main(String[] args) {
        int[] myArray = new int[5];
        // initialisiere Werte in Array: {1, 2, 3, 4, 5}
        for (int i = 0; i < myArray.length; i++) {
            myArray[i] = i+1;
        }
        // Berechne Durchschnitt:
        float sum = 0;
        for (int i = 0; i < myArray.length; i++) {
            sum += myArray[i];
        }
        System.out.println(sum/myArray.length);
    }
}
```

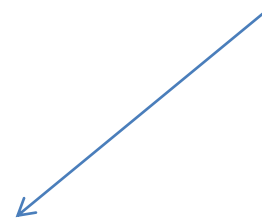
- Berechnet Durchschnitt
- Beachte impliziten Cast auf float!

# Beispiel: While, For Each



```
public class ArrayExample {
    public static void main(String[] args) {
        int[] myArray = new int[5];
        // initialisiere Werte in Array: {1, 2, 3, 4, 5}
        int i = 0;
        while (i < myArray.length) { // while
            myArray[i] = i+1;
            i++;
        }
        // Berechne Durchschnitt:
        float sum = 0;
        for (int myInt : myArray) { // for each
            sum += myInt;
        }
        System.out.println(sum/myArray.length);
    }
}
```

- Andere Schleifen
- Beachte „for each“





# Beispiel: Initialisierung



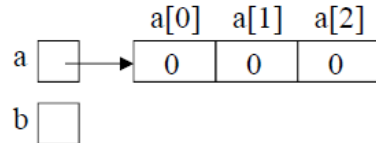
```
public class ArrayExample {  
    public static void main(String[] args) {  
        // initialisiere Werte in Array: {1, 2, 3, 4, 5}  
        int[] myArray = {1, 2, 3, 4, 5};  
        // Berechne Durchschnitt:  
        float sum = 0;  
        for (int myInt : myArray) { // for each  
            sum += myInt;  
        }  
        System.out.println(sum/myArray.length);  
    }  
}
```

- Andere Initialisierung!

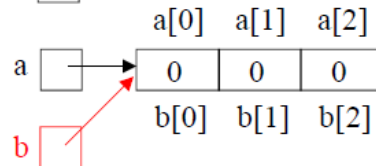
# Arrayzuweisung



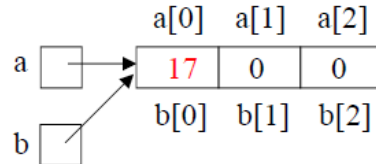
```
int[] a, b;  
a = new int[3];
```



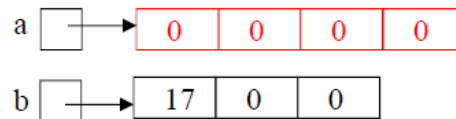
```
b = a;
```



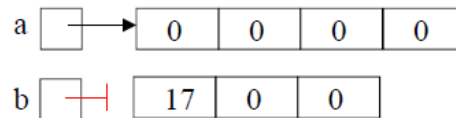
```
a[0] = 17;
```



```
a = new int[4];
```



```
b = null;
```



Arrayelemente werden in Java standardmäßig mit 0 initialisiert

*b* bekommt denselben Wert wie *a*.  
Arrayzuweisung ist in Java **Zeigerzuweisung!**

ändert in diesem Fall auch *b*[0]

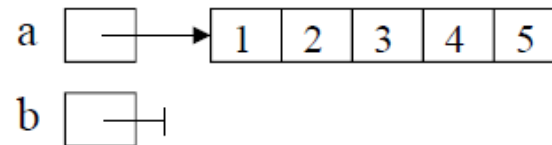
*a* zeigt jetzt auf neues Array.

*null*: Spezialwert, der auf kein Objekt zeigt;  
kann jeder Arrayvariablen zugewiesen werden

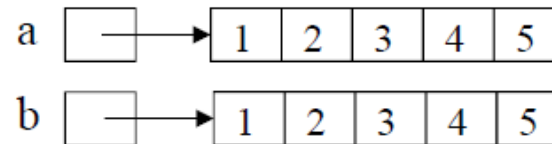
# Kopieren von Arrays



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



```
b = (int[]) a.clone();
```



- Typumwandlung nötig, da `a.clone()` Typ `Object[]` liefert

# Beispiel: Sequentielle Suche



```
public class ArrayExample {
    public static void main(String[] args) {
        int[] myArray = {12, 2, 32, 74, 26, 42, 53, 22};
        int query = 22;
        for (int i = 0; i < myArray.length; i++) {
            if (query == myArray[i]) {
                System.out.println("Found at position " + i);
            }
        }
    }
}
```

- Jedes Element wird untersucht  
-> sequentiell
- Braucht  $n$  Schritte - Wie groß ist  $n$ ?

# Beispiel: Sortierung



- Wie sortiert man ein Array  $a$ ?
- Einfacher Ansatz:
  1. Erzeuge ein gleich großes Array  $b$
  2. Verschiebe Minimum von  $a$  nach  $b$
  3. Falls  $a$  nicht leer gehe zu Schritt 2.

# Beispiel: Sortierung



```
public class ArrayExample {
    public static void main(String[] args) {
        // o.b.d.A. a[k] > 0 & a[k] < 100
        int[] a = {12, 2, 32, 74, 26, 42, 53, 22};
        // create result array
        int[] b = new int[a.length];
        for (int i = 0; i < b.length; i++) { // set each item of b
            int minimum = 100;
            int pos = 0;
            for (int j = 0; j < a.length; j++) { // find minimum
                if (a[j] < minimum) {
                    minimum = a[j];
                    pos = j;
                }
            }
            b[i] = minimum;
            a[pos] = 100; // set visited.
        }

        for (int i = 0; i < b.length; i++) {
            System.out.print(b[i] + ", ");
        }
    }
}
```

- Lösbar auf viele Arten
- Vgl. AlgoDat!

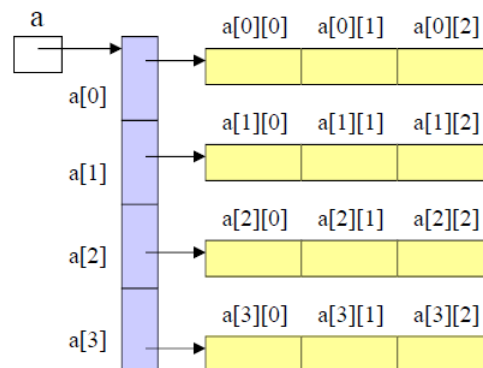
# Mehrdimensionale Arrays



- Zweidimensionales Arrays == Matrix

	0	1	2
0	a[0][0]	a[0][1]	a[0][2]
1	a[1][0]	a[1][1]	a[1][2]
2	a[2][0]	a[2][1]	a[2][2]
3	a[3][0]	a[3][1]	a[3][2]

- In Java: Array von Arrays



Deklaration und Erzeugung

```
int[][] a;  
a = new int[4][3];
```

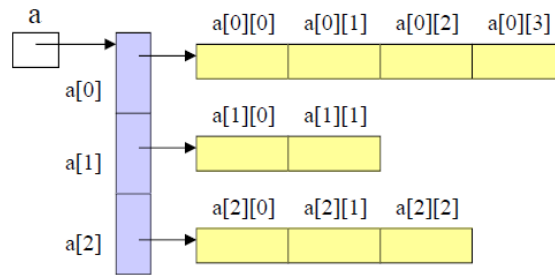
Zugriff

```
a[i][j] = a[i][j+1];
```

# Mehrdimensionale Arrays



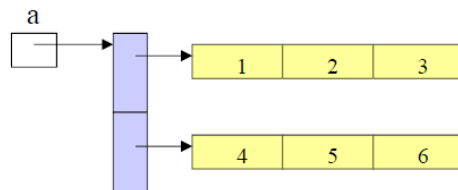
- Zeilen können unterschiedlich lang sein



```
int[][] a = new int[3][];  
a[0] = new int[4];  
a[1] = new int[2];  
a[2] = new int[3];
```

- Initialisierung

```
int[][] a = {{1, 2, 3}, {4, 5, 6}};
```





# Beispiel: Mehrdimensionale Arrays



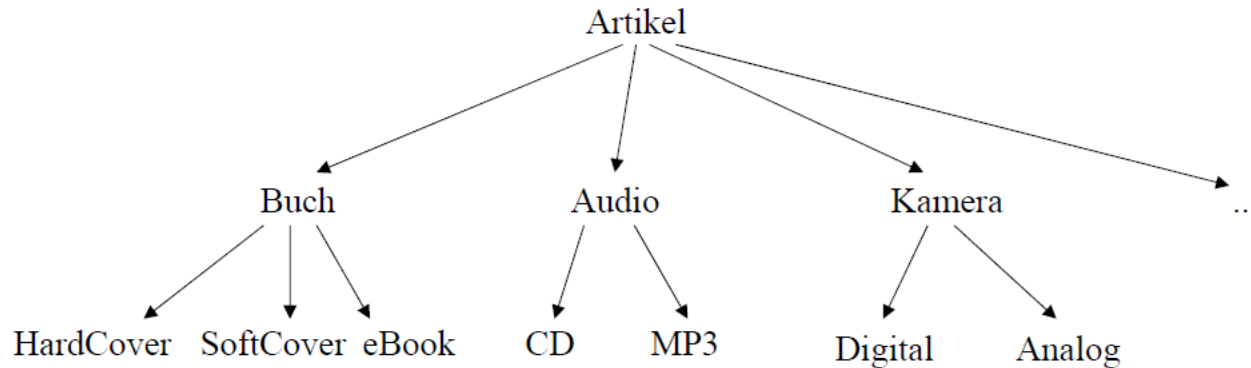
- 2D als Modell für Schiffe versenken.

# Klassifikation



## Dinge der realen Welt lassen sich oft klassifizieren

z.B. Artikel eines Web-Shops



## Man beachte

- Ein *eBook* hat alle Eigenschaften eines *Buchs*; zusätzlich hat es ...  
Ein *Buch* hat alle Eigenschaften eines *Artikels*; zusätzlich hat es ...
- *CD* und *MP3* lassen sich gleichermaßen als *Audio* behandeln  
*Buch*, *Audio* und *Kamera* lassen sich gleichermaßen als *Artikel* behandeln

Vererbung

# Vererbung



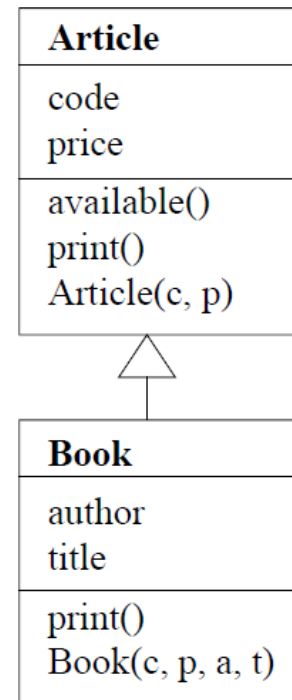
```
class Article {  
    int code;  
    int price;  
  
    boolean available() {...}  
    void print() {...}  
  
    Article(int c, int p) {...}  
}
```

```
class Book extends Article {  
    String author;  
    String title;  
  
    void print() {...}  
  
    Book(int c, int p,  
        String a, String t) {...}  
}
```

**Oberklasse**  
**Basisklasse**

**Unterklasse**

erbt: *code, price, available, print*  
ergänzt: *author, title, Konstruktor*  
überschreibt: *print*



Wenn keine Oberklasse angegeben wird, ist sie *Object*

# Überschreiben von Methoden



```
class Article {  
    ...  
    void print() {  
        Out.print(code + " " + price);  
    }  
    Article(int c, int p) {  
        code = c; price = p;  
    }  
}
```

```
class Book extends Article {  
    ...  
    void print() {  
        super.print();  
        Out.print(" " + author + ": " + title);  
    }  
    Book(int c, int p, String a, String t) {  
        super(c, p);  
        author = a; title = t;  
    }  
}
```

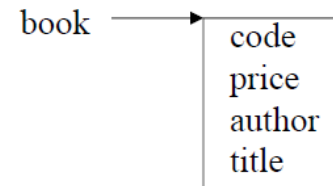
## Benutzung

```
Book book = new Book(code, price, author, title);
```

⇒ erzeugt *Book*-Objekt

⇒ *Book*-Konstruktor

⇒ *Article*-Konstruktor (code = c; price = p; author = a; title = t;



```
book.print();
```

⇒ *print* aus *Book*

⇒ *print* aus *Article*

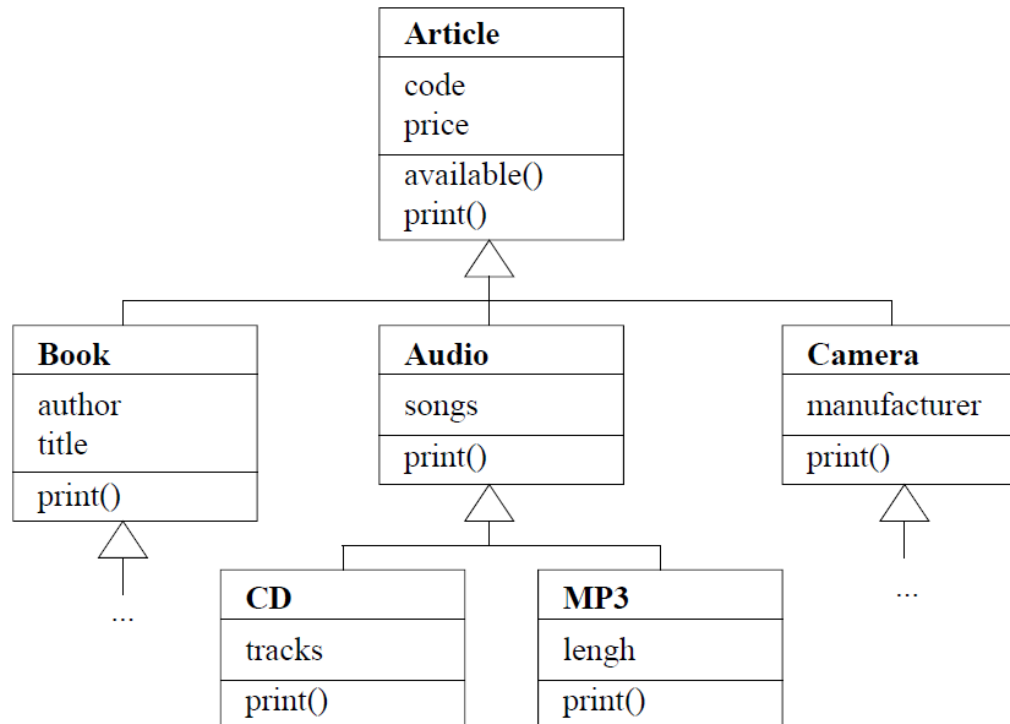
⇒ Out.print(...);

code price

author: title

Ausgabe: code price author: title

# Klassenhierarchien



Jedes Buch ist ein Artikel  
Aber: nicht jeder Artikel ist ein Buch

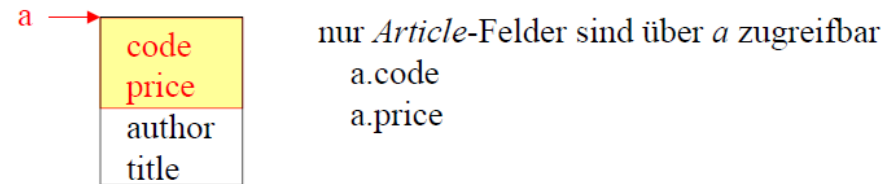
# Kompatibilität zwischen Klassen



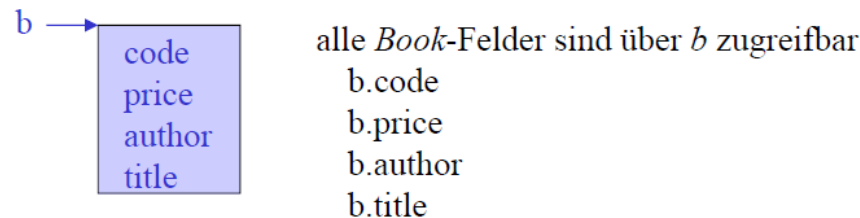
**Unterklassen sind Spezialisierungen ihrer Oberklassen**

***Book*-Objekte können *Article*-Variablen zugewiesen werden**

```
Article a = new Book(code, price, author, title);
```



```
if (a instanceof Book) // Laufzeittypstest  
    Book b = (Book) a; // Typumwandlung mit Laufzeittypprüfung
```

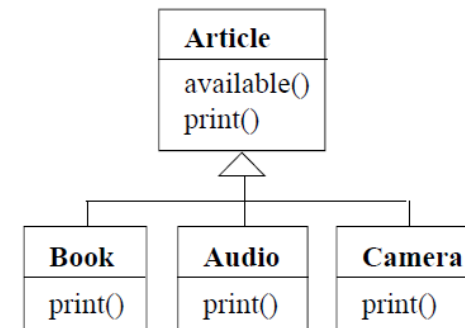
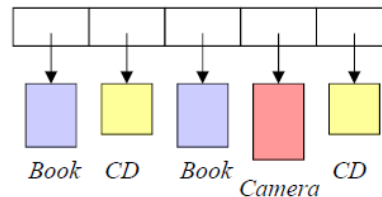


# Dynamische Bindung



## Heterogene Datenstruktur

Article[] a;



Alle Varianten können als Artikel behandelt werden

```
void printArticles() {  
    for (int i = 0; i < a.length; i++) {  
        if (a[i].available()) {  
            a[i].print();  
        }  
    }  
}
```

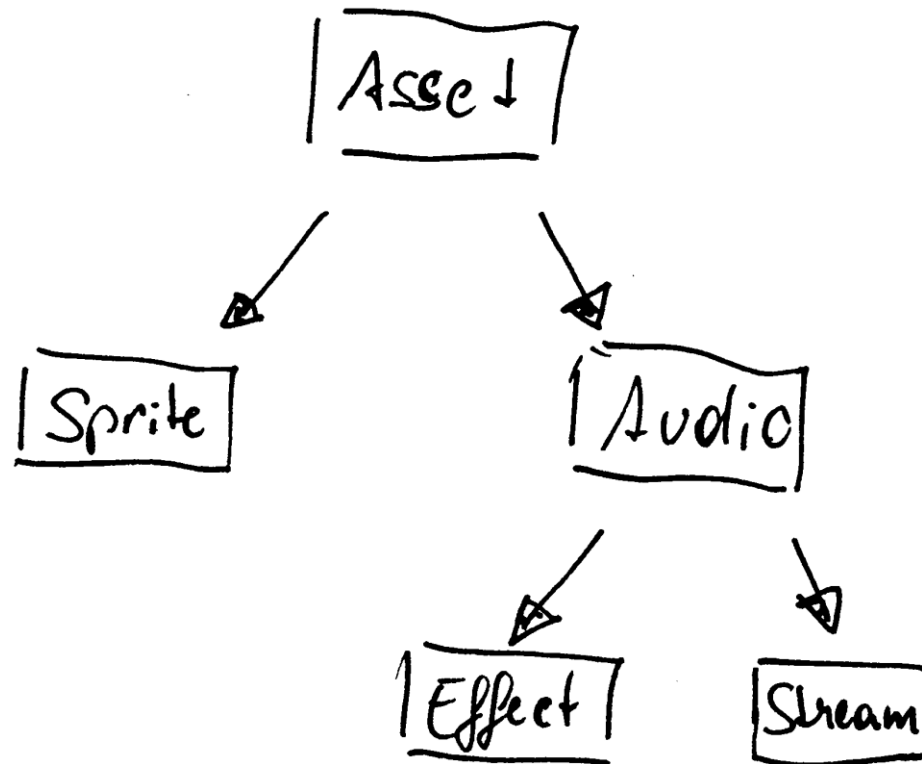
ruft *available()* aus *Article* auf

ruft je nach Artikelart das *print()* aus *Book*, *CD* oder *Camera* auf

## Dynamische Bindung

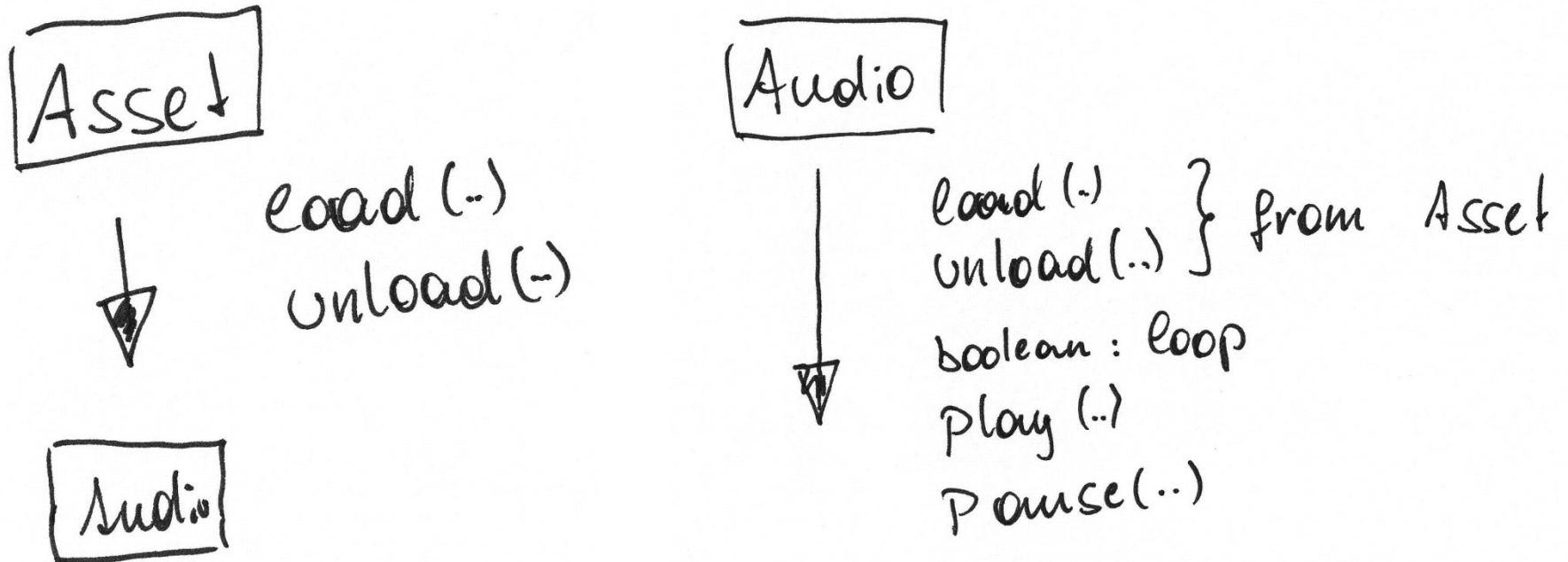
*obj.print()* ruft die *print*-Methode des Objekts auf, auf das *obj* gerade zeigt

# Beispiel ...

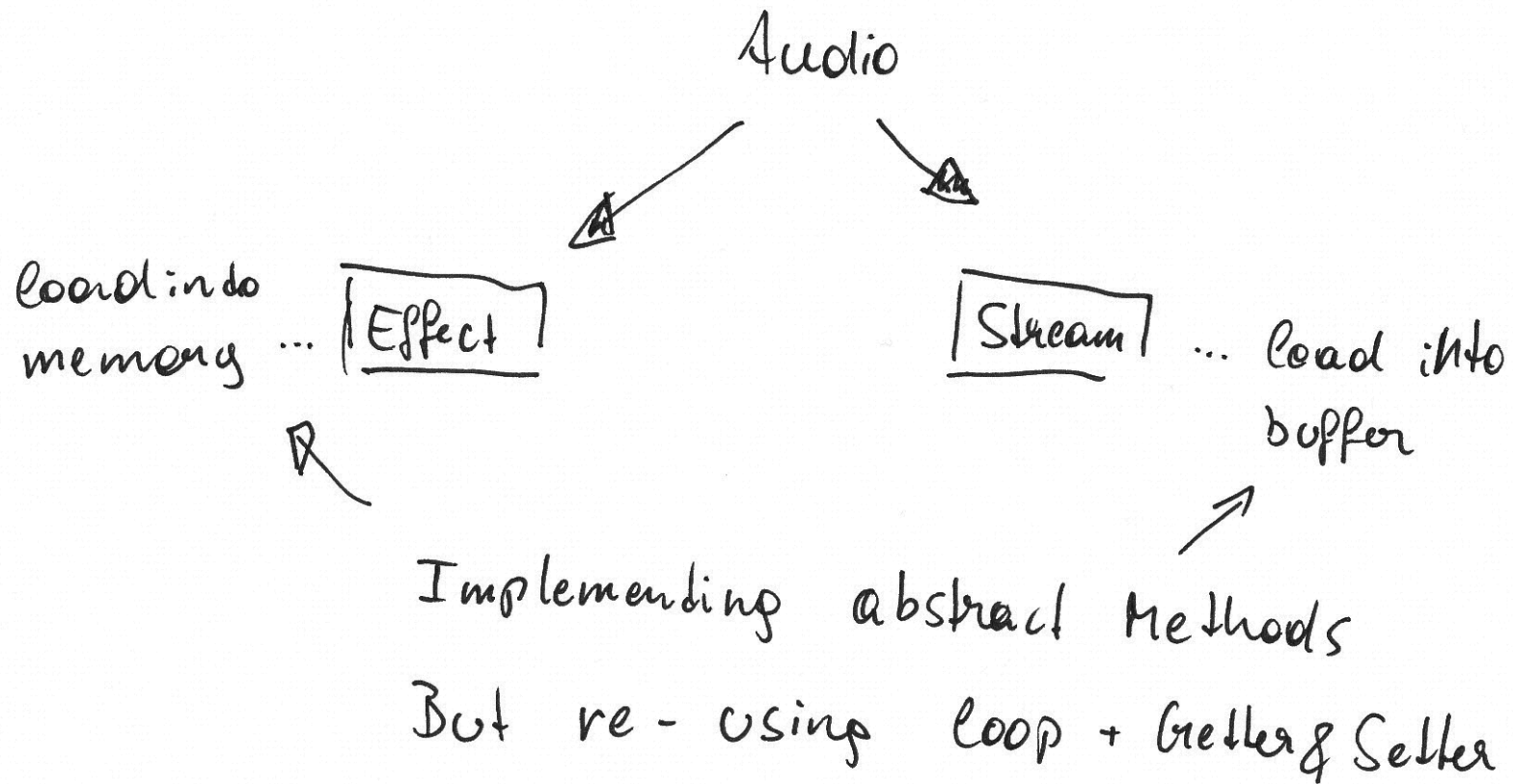




# Beispiel



# Beispiel



# Zusätzliche Konzepte



## Keyword **abstract**

- spezifiziert, dass alle Subklassen eine solche Methode haben,
- aber bietet sie nicht an
  - Im Gegensatz, sie wird verlangt.

# Interfaces



- Klassenähnlicher Mechanismus
  - zur reinen Verhaltensspezifikation.
- Erlaubt die Definition eines benutzerdefinierten Datentyps von seiner Realisierung zu trennen
  - abstrakter Datentyp.

# Interfaces



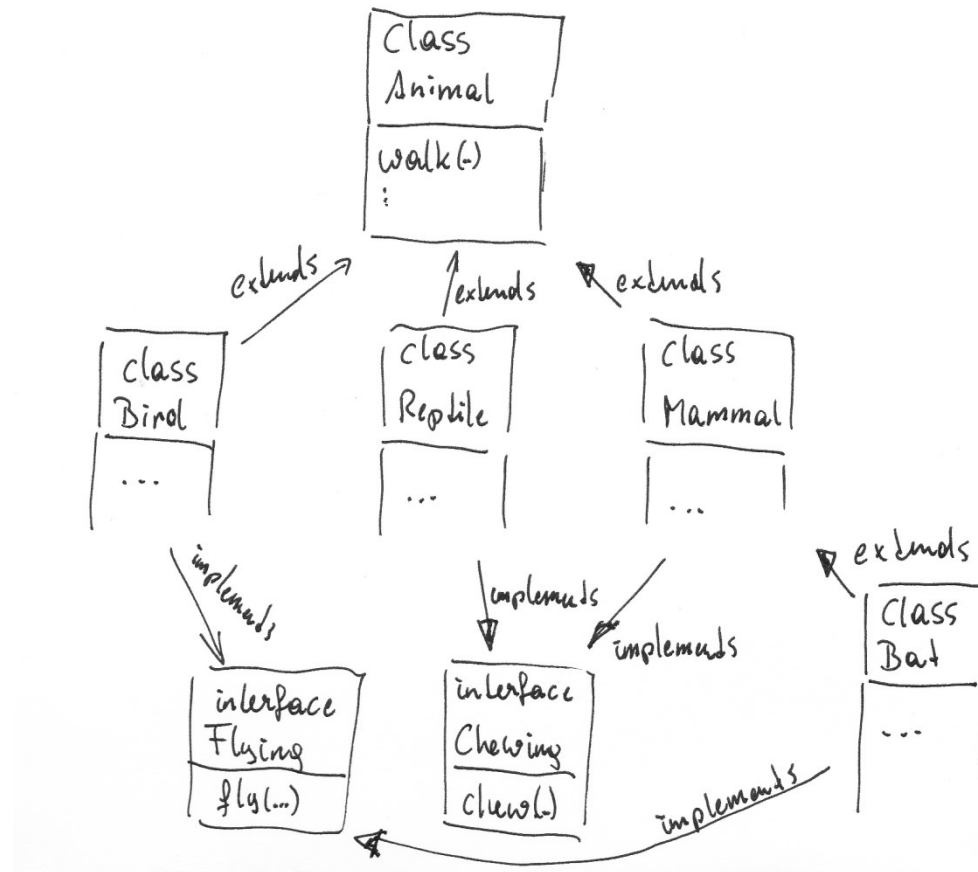
- Spezifikation via interface
- Methoden-Spezifikationen
  - beschreiben, auf welche Nachrichten ein Objekt reagiert
  - ohne Rumpf, also ohne Implementierung.
- Keine Instanzvariablen
  - Aber evt. Konstante

# Interfaces



- Der interface-Name ist in Java als Datentyp verwendbar
- Implementierung via `class`
- Vollständige Methoden
- Instanzvariablen

# Interface Example I



# Interface Example II



[Overview](#) [Package](#) **[Class](#)** [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

Java™ 2 Platform  
Standard Ed. 5.0

[PREV CLASS](#) [NEXT CLASS](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

java.lang

## Interface `Iterable<T>`

### All Known Subinterfaces:

[BeanContext](#), [BeanContextServices](#), [BlockingQueue<E>](#), [Collection<E>](#), [List<E>](#), [Queue<E>](#), [Set<E>](#), [SortedSet<E>](#)

### All Known Implementing Classes:

[AbstractCollection](#), [AbstractList](#), [AbstractQueue](#), [AbstractSequentialList](#), [AbstractSet](#), [ArrayBlockingQueue](#), [ArrayList](#), [AttributeList](#), [BeanContextServicesSupport](#), [BeanContextSupport](#), [ConcurrentLinkedQueue](#), [CopyOnWriteArrayList](#), [CopyOnWriteArraySet](#), [DelayQueue](#), [EnumSet](#), [HashSet](#), [JobStateReasons](#), [LinkedBlockingQueue](#), [LinkedHashSet](#), [LinkedList](#), [PriorityBlockingQueue](#), [PriorityQueue](#), [RoleList](#), [RoleUnresolvedList](#), [Stack](#), [SynchronousQueue](#), [TreeSet](#), [Vector](#)

```
public interface Iterable<T>
```

Implementing this interface allows an object to be the target of the "foreach" statement.

## Method Summary

<a href="#">Iterable&lt;I&gt;</a>	<a href="#">iterator()</a> Returns an iterator over a set of elements of type T.
-----------------------------------	---

## Method Detail

### iterator

[Iterable<I>](#) `iterator()`

Returns an iterator over a set of elements of type T.



# Verwendung Interfaces?



- Freigabe minimaler Funktionalität eines abstrakten Datentyps
- Mehrfachvererbung
  - Graph, nicht Baum

# java.io.File



- Klasse für den Umgang mit Dateien
  - Verzeichnisse sind auch Dateien
- Methoden erlauben
  - Existenzprüfung, Rechteprüfung
  - Auflistung der Kind-Dateien
  - Verzeichnisse anlegen
  - uvm.

# java.io.FileReader



- Erweitert Reader (abstrakte Klasse)
- Liest Zeichen (char) aus File
  - Vgl. FileInputStream -> byte
  - Unterschied zwischen char & byte!
- Nutzung benötigt einen Buffer
  - char[] für Reader, byte[] für InputStream

# java.io.BufferedReader



- Stellt Buffer zur Verfügung
- Liefert ganze Zeilen zurück
- **ACHTUNG:** Ist abgeleitet von Reader -> char[]!

# Schreiben in Dateien?



- `FileWriter`, `BufferedWriter`
  - Nicht auf `close()` bzw. `flush()` vergessen!

Und:

- `ObjectStreamWriter`, `ObjectStreamReader`
  - Schreibt und liest Objekte von einem `InputStream`
- `GZipInputStream`, `GZipOutputStream`
  - Komprimiert I/O Streams