

# ESOP - Klassen und Objekte

Assoc. Prof. Dr. Mathias Lux  
ITEC / AAU

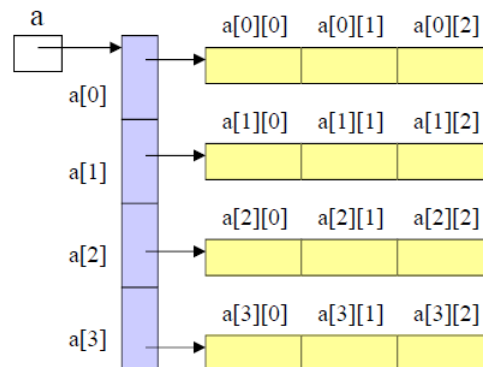
# Mehrdimensionale Arrays



- Zweidimensionales Array == 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];
```

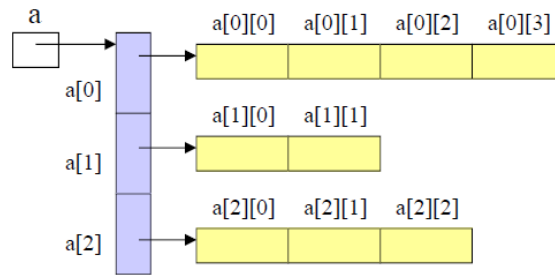
Zugriff

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

# Mehrdimensionale Arrays



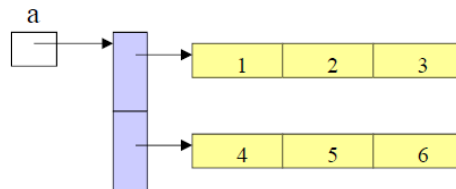
- Zeilen können unterschiedlich lang sein



```
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: Name Generator



- Vokale & Konsonanten in Arrays
- Zufallszahlen via Systemzeit
- Funktionalität ausgelagert in Methoden

# 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 -> by value
  - alles andere -> by reference

# Über „Alles Andere“ ...



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

# Java-Klassen



- Beispiel: Speichern eines Datums in einer einzelnen Struktur.
  - Tag, Monat, Jahr
- Einzelne Werte unbequem ..
  - wenn man mehrere speichern will
  - als Rückgabewert einer Funktion
  - im Vergleich mit anderen Datums-Elementen



# Java-Klassen



- Idee: Fasse die notwendigen Variablen zu einer Struktur (Klasse) zusammen:

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



Klassenname



Felder (fields, class members)

# Datentype Klasse



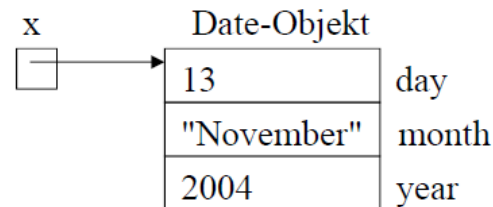
- Deklaration
- Verwendung als Typ
- Zugriff

```
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





- Klasse ist wie eine Schablone
  - Nach deren Vorlage Objekte erstellt werden
- Objekte (Instanzen) einer Klasse müssen vor Verwendung erzeugt werden!
  - Variablen haben sonst den Wert null

# Objekte



Date x, y;

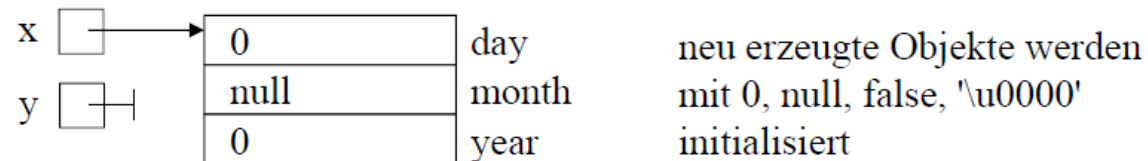
reserviert nur Speicher für die Zeigervariablen

x  y  haben anfangs den Wert *null*

## Erzeugung

x = **new** Date();

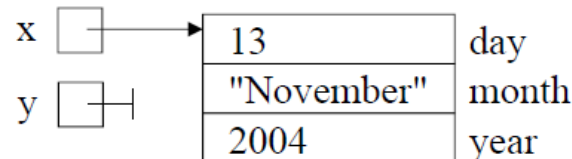
erzeugt ein *Date*-Objekt und weist seine Adresse *x* zu



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

## Benutzung

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



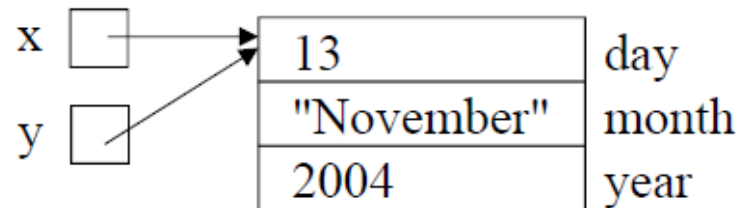
## Freigabe von Objekten

durch den Garbage Collector

# Zuweisungen

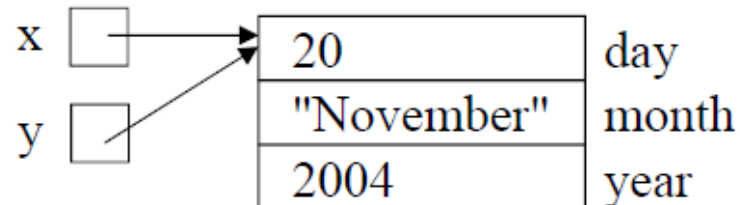


`y = x;`



*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();
```

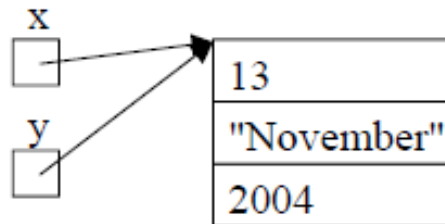
```
d1 = d2;           // ok, gleiche Typen  
a1 = a2;           // ok, gleiche Typen  
d1 = a2;           // verboten: verschiedene Typen trotz gleicher Struktur!
```

# Referenzvergleich

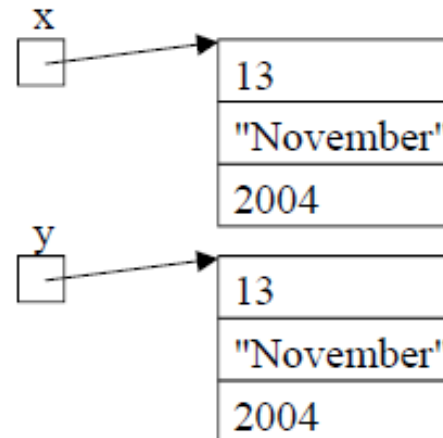


- $x == y$  und  $x != y$  ... vergleicht Referenzen
- $<$ ,  $<=$ ,  $>$ ,  $>=$  ... nicht erlaubt

$x == y$  liefert true



$x == y$  liefert false



# Wertevergleich



- Muss durch Methode implementiert werden.

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



# Wo werden Klassen deklariert?



## Eine 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

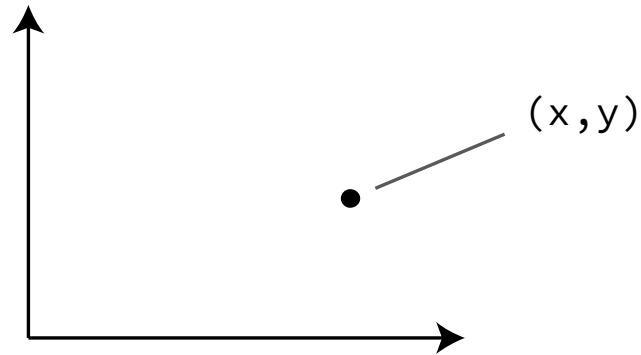
Übersetzung

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

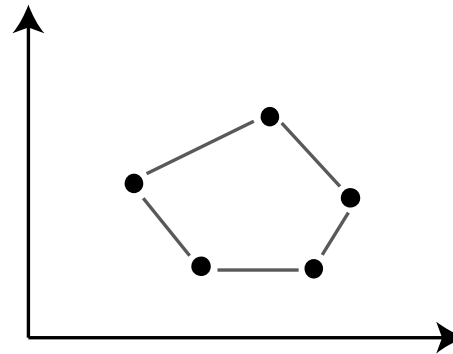
# Was kann man mit Klassen tun?



```
class Point {  
    double x,y;  
}
```



```
class Polygon {  
    Point[] points;  
}
```



# Was kann man mit Klassen tun?



- Klassen können andere Klassen beinhalten
  - und darauf aufbauen

```
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 Basisdatentypen, 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
- .. und Rückgabewerte



# Beispiel: Positionsklasse



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

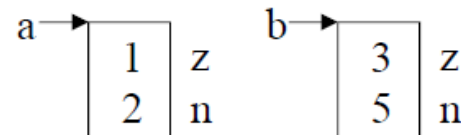
- this notwendig wenn auf Objektweiten Scope zugegriffen wird
- Bei Nichtnutzung würde lokales x verwendet werden.

# Beispiel: Bruchzahlenklasse

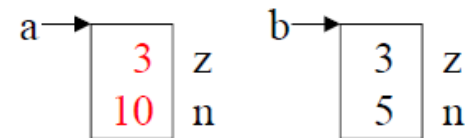


```
class Fraction {  
    int z; // Zähler  
    int n; // Nenner  
  
    void mult (Fraction f) {  
        z = z * f.z;  
        n = n * f.n;  
    }  
  
    void add (Fraction f) {  
        z = z * f.n + f.z * n;  
        n = n * f.n;  
    }  
}
```

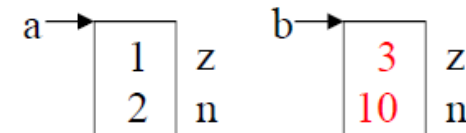
Fraction a = new Fraction(); a.z = 1; a.n = 2;  
Fraction b = new Fraction(); b.z = 3; b.n = 5;



**a.mult(b);**



**b.mult(a);**



Es wird immer der Zustand des Empfängers verändert!

# UML Notation



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

<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



```
public class ExtendedFraction {
    int n; // numerator
    int d; // denominator

    /**
     * Constructor for the fraction class.
     * @param n
     * @param d
     */
    public ExtendedFraction(int n, int d) {
        this.n = n;
        this.d = d;
    }

    public ExtendedFraction() {
        n = 0;
        d = 1; // make sure denominator is not 0.
    }

    /**
     * Multiply this fraction with another one.
     *
     * @param f the second factor
     */
    void mult(ExtendedFraction f) {
        ...
    }
}
```

```
ExtendedFraction f = new ExtendedFraction();
ExtendedFraction g = new ExtendedFraction(3, 5);
```

- ruft entsprechende Konstrukturen auf.

# Konstrukturen ...



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

# Beispiel für eine Klasse: `java.lang.String`



- Char-Array vs. Strings
  - `char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };`
  - `String helloString = new String(helloArray);`
  - `System.out.println(helloString);`
- Länge eines String-Objekts
  - `helloString.length()`
- Aus String chars lesen
  - `helloString.charAt(2) // result: 'l',`
  - `helloString.getChars(...)`
  - `helloString.toCharArray()`

# Beispiel: Reverse String



```
public class ReverseString {  
    public static void main(String[] args) {  
        // input String  
        String myString = new String("FTW");  
        // data structures for reversing  
        char[] tmpCharsIn = new char[myString.length()];  
        char[] tmpCharsOut = new char[myString.length()];  
        // getting the input data to an array:  
        myString.getChars(0, myString.length(), tmpCharsIn, 0);  
        // iterating output and setting chars:  
        for (int i = 0; i < tmpCharsOut.length; i++) {  
            tmpCharsOut[i] = tmpCharsIn[myString.length()-1-i];  
        }  
        // print result:  
        System.out.println(new String(tmpCharsOut));  
    }  
}
```



# Java String



- String aneinanderhängen
  - `string1.concat(string2)`
  - `"Hello ".concat("World!")`
  - `"Hello " + "World!"`
- Achtung: Die String-Klasse ist immutable

# Strings $\rightleftharpoons$ Numbers



- String zu Zahl
  - `float a = (Float.valueOf("3.14")).floatValue();`
  - `float a = Float.parseFloat("3.14");`
  - Entsprechend für die anderen numerischen Typen
- Zahl zu String
  - `String s = Double.toString(42.0);`

# String - Manipulation



- **Teilstring**
  - `String substring(int beginIndex, int endIndex)`
  - `String substring(int beginIndex)`
- **Groß- und Kleinschreibung**
  - `String toLowerCase()`
  - `String toUpperCase()`
- **Leerzeichen am Ende entfernen**
  - `String trim()`

# String - Suche



- Suche nach char oder String in Strings
  - `int indexOf(char ch)`
  - `int lastIndexOf(char ch)`
  - `int indexOf(char ch, int fromIndex)`
  - `int lastIndexOf(char ch, int fromIndex)`
- Auch mit String als argument
  - `int indexOf(String str)`
  - ...

# Beispiel



```
public static void main(String[] args) {  
    // input  
    String myFileName = "paper.pdf";  
    // find the position of the last dot  
    int dotIndex = myFileName.lastIndexOf('.');  
    // take substring and add new suffix  
    String newFileName = myFileName.substring(0, dotIndex) + ".doc";  
    // print result:  
    System.out.println("newFileName = " + newFileName);  
}
```

# String - Andere Methoden



- `boolean endsWith(String suffix)`
- `boolean startsWith(String prefix)`
- `int compareTo(String anotherString)`
- `boolean equals(Object anObject)`
- ...

mehr Information:

<https://docs.oracle.com/javase/8/docs/api/java/lang/String.html>

# CharSequence

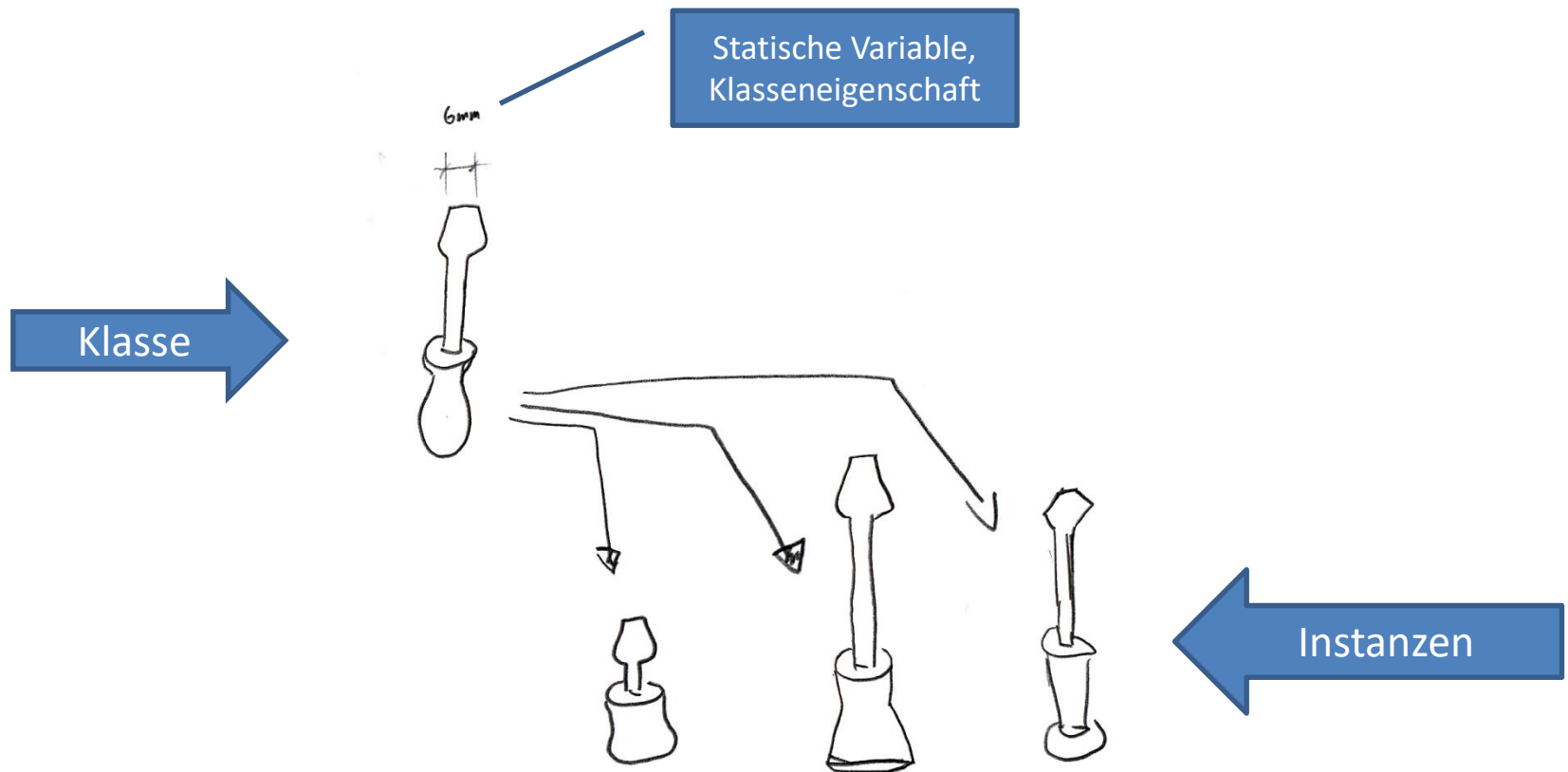


- String ist immutable
  - Manipulationen sind teuer in der Ausführung
- CharSequence ist Interface für String-ähnliche Klassen
  - StringBuilder
  - StringBuffer

mehr Informationen:

<https://docs.oracle.com/javase/8/docs/api/java/lang/CharSequence.html>

# 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



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

*Klasse Window*

<code>border</code>
<code>setBorder()</code> Klassenkonstruktor

*Window-Objekt*

<code>x</code> <code>y</code> <code>w</code> <code>h</code>
<code>redraw()</code> <code>Window()</code>

*Window-Objekt*

<code>x</code> <code>y</code> <code>w</code> <code>h</code>
<code>redraw()</code> <code>Window()</code>

*Window-Objekt*

<code>x</code> <code>y</code> <code>w</code> <code>h</code>
<code>redraw()</code> <code>Window()</code>

# 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 OOP, SE und weiterführende VO/PRs

# Beispiel für static: java.lang.Math



- Java stellt erweiterte mathematische Funktionen in der Klasse Math bereit
- Jede Methode in Math ist static
  - Optionaler statischer Import
  - `import static java.lang.Math.*;`
  - dann Methode wie Funktionsaufrufe, zB. `cos(x)`

# Java Math Konstante



- Math.E
  - Eulersche Zahl  $e$
- Math.PI
  - Kreiszahl  $\pi$

# Java Math Basics



- Absolutwerte
  - `int Math.abs(int value)`
  - auch für `double`, `long`, `float`
- Auf- und Abrundung
  - `double Math.ceil(double value)`
  - `double Math.floor(double value)`
- Rundung
  - `long Math.round(double value)`
  - `int Math.round(float value)`



# Java Math Basics



- Minimum zweier Zahlen
  - `double Math.min(double arg1, double arg2)`
  - auch für `float`, `long`, `int`
- Maximum zweier Zahlen
  - `double Math.max(double arg1, double arg2)`
  - auch für `float`, `long`, `int`

# Java Math Exp & Log



- Exponentialfunction und Logarithmus
  - `double Math.log(double value)`
  - `double Math.exp(double value)`
- Potenzieren und Wurzel
  - `double Math.pow(double base, double exp)`
  - `double Math.sqrt(double value)`

# Java Math Trigonometrie



- Winkelfunktionen
  - `double Math.sin(double value)`
  - auch für `cos`, `tan`, `asin`, `acos`, `atan`
- Winkel eines Vektors
  - `double Math.atan2(double x, double y)`

# 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('*');  
    }  
}
```

# Java Math - Zufall



- `double Math.random()`
  - liefert Pseudo-Zufallszahl  $0 \leq x < 1$
  - funktioniert ausreichend gut für einzelne Zufallszahlen
- Andere Zahlenbereiche
  - z.B. `Math.random() * 10.0`

# 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("bcdfghjklmnpqrstvwxyz").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(v));
        System.out.print(getRandomChar(c));
    }

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



- JavaDoc
  - <https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html>
- BigInteger
  - Für beliebig große ganze Zahlen
- BigDecimal
  - Für beliebig genaue Dezimalzahlen

# 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



# 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--];
    }
}
```

## Usage:

```
public static void main(String[] args) {
    Stack s = new Stack(10);
    s.push(3);
    s.push(5);
    int x = s.pop() - s.pop();
    System.out.println("x = " + x);
}
```

# Queue



## Nutzung:

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

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

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

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

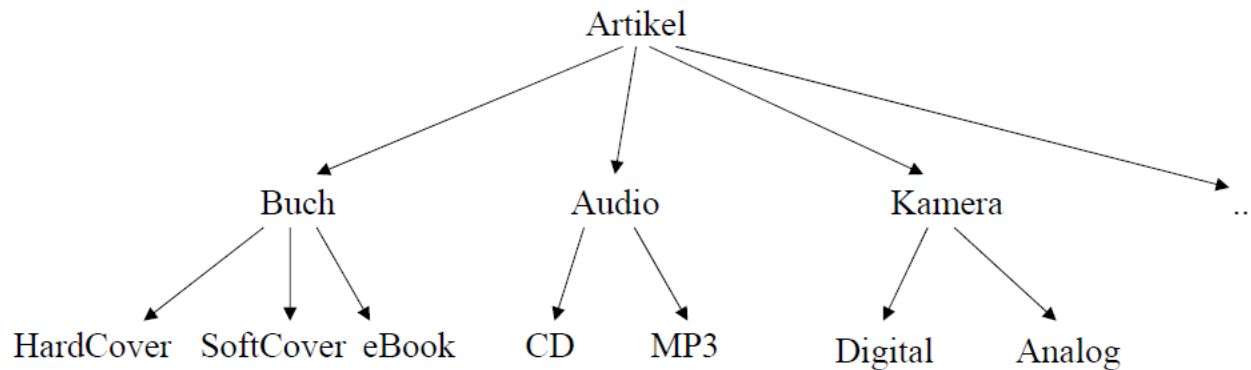
```
Queue q = new Queue(10);
q.put(3);
q.put(6);
int x = q.get(); // x == 3
int y = q.get(); // y == 6
```

# 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



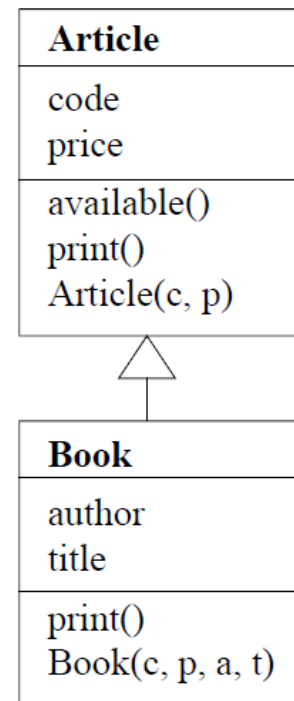
```
class Article {  
    int code;  
    int price;  
  
    boolean available() {...}  
    void print() {...}  
  
    Article(int c, int p) {...}  
}
```

```
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;  
    }  
}
```

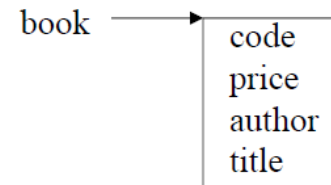
## Benutzung

```
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

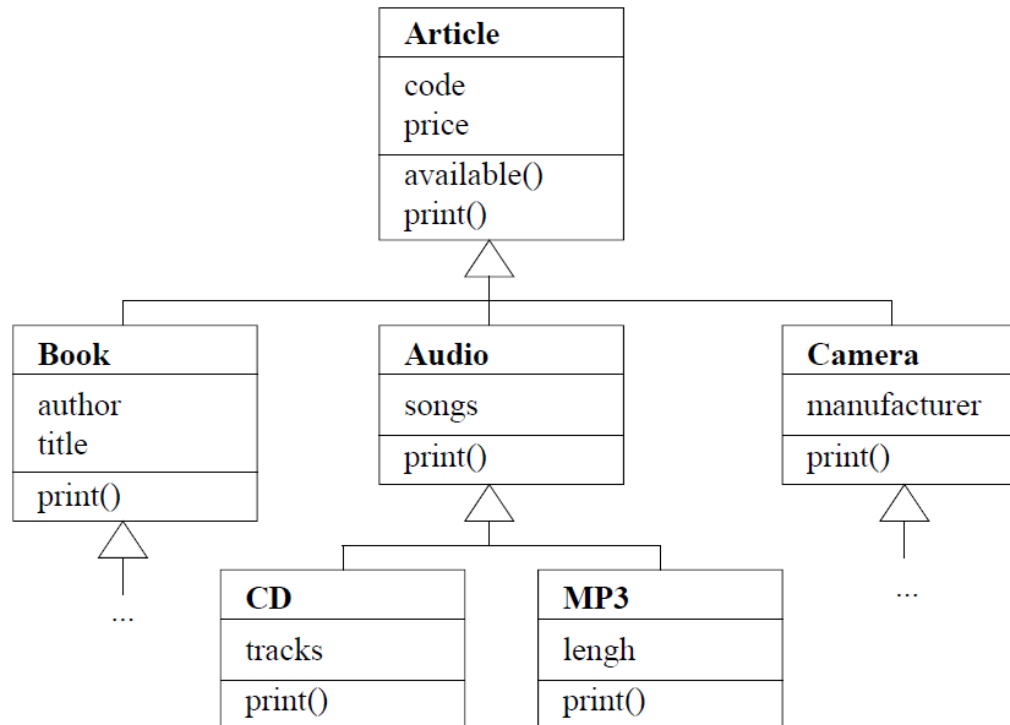
Ausgabe: code price author: title

# Addendum



- super kann nur auf die direkte Superklasse zugreifen.
- Sonst würden Vererbungsprinzipien verletzt werden
    - Überspringen/Ignorieren der Superklasse

# Klassenhierarchien



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

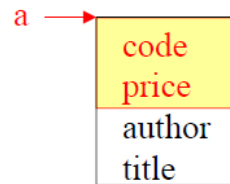
# Kompatibilität zwischen Klassen



**Unterklassen sind Spezialisierungen ihrer Oberklassen**

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

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

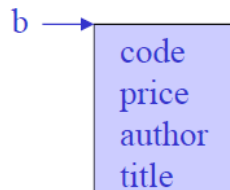


nur *Article*-Felder sind über *a* zugreifbar

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

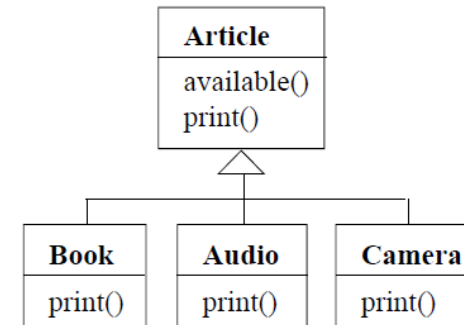
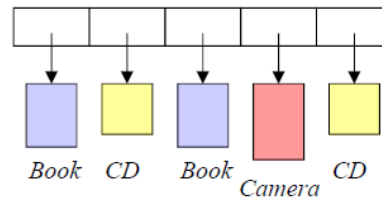


# Dynamische Bindung



## 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();  
        }  
    }  
}
```

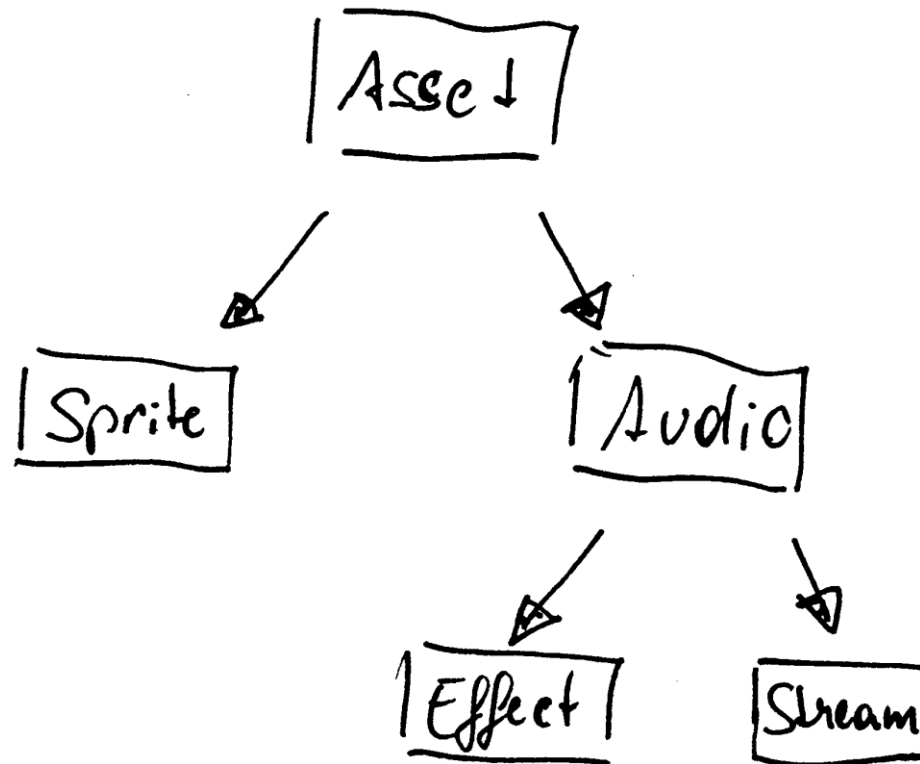
ruft *available()* aus *Article* auf

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

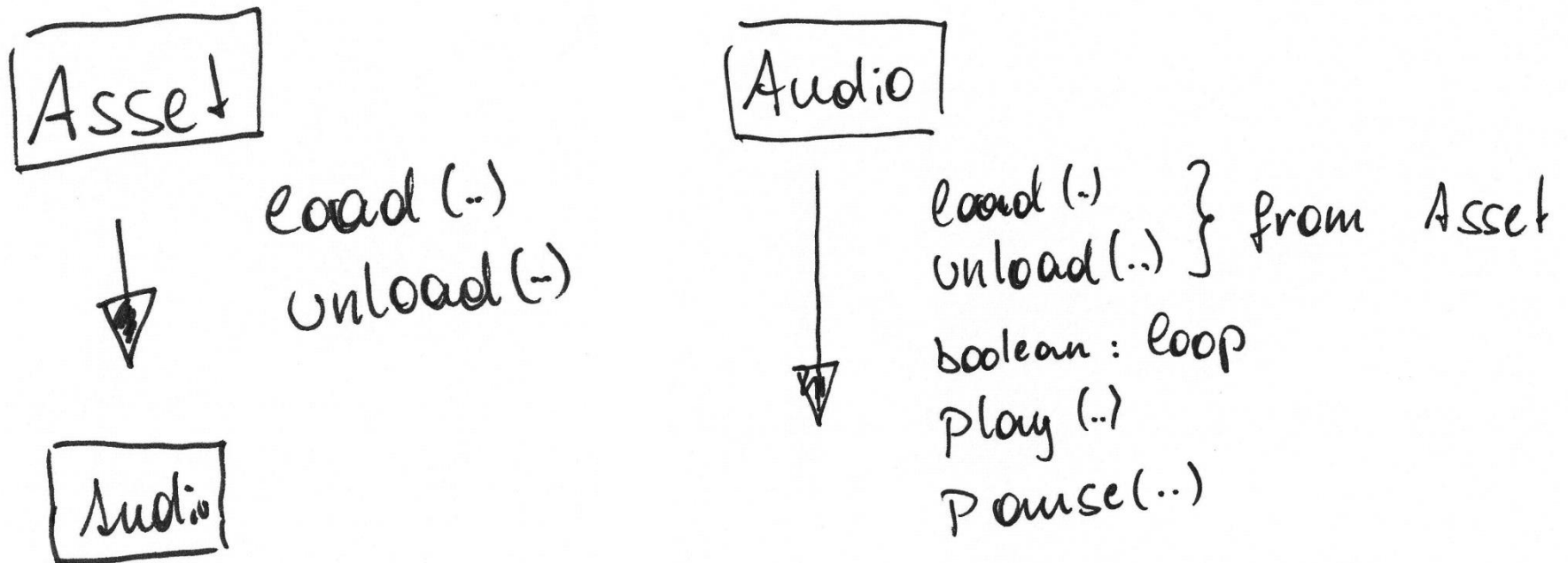
## Dynamische Bindung

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

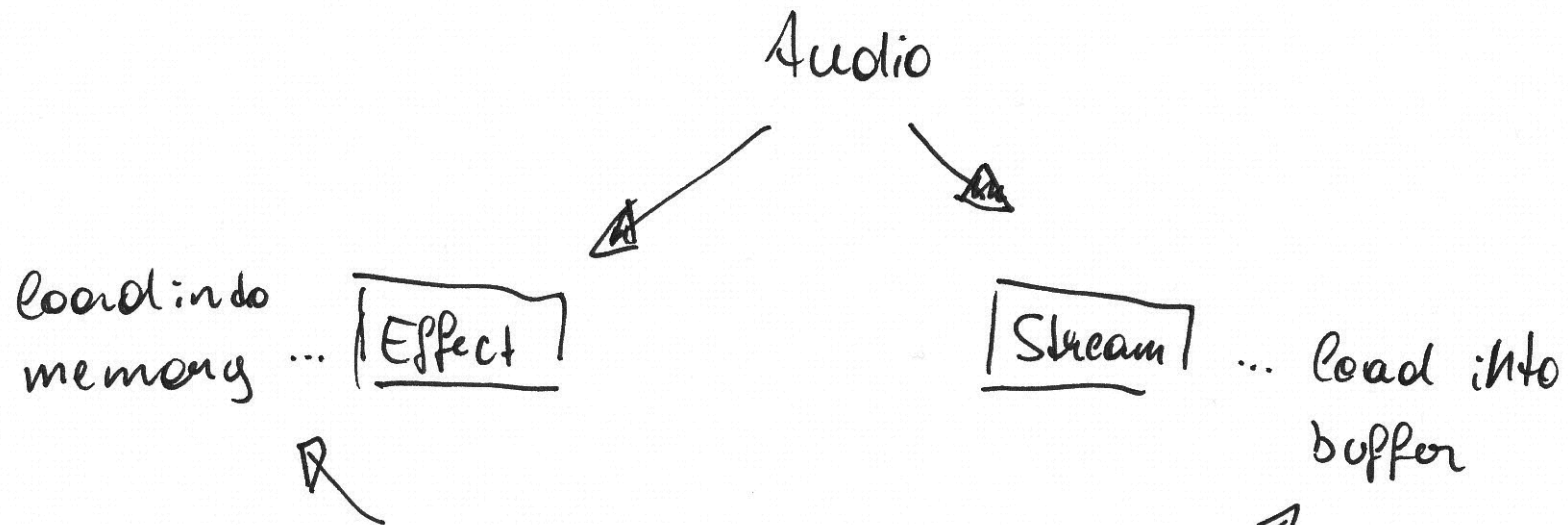
# Beispiel ...



# Beispiel



# Beispiel



Implementing abstract Methods  
But re-using loop + Getter & Setter

# Zusätzliche Konzepte



## Keyword **abstract**

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