

# ESOP - Classes and Objects

Assoc. Prof. Dr. Mathias Lux  
ITEC / AAU

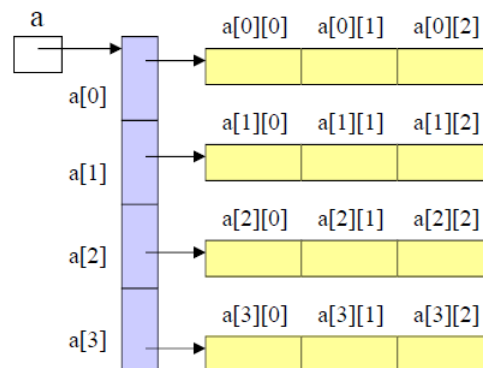
# Multidimensional Arrays



- 2-dimensional 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: arrays of arrays



Declaration and instantiation

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

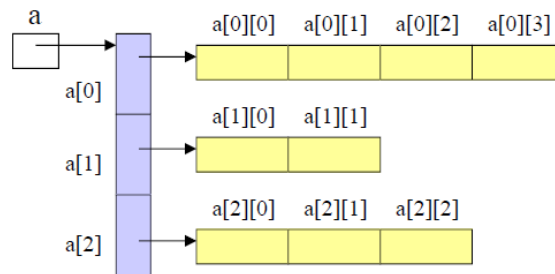
Access

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

# Multidimensional Arrays



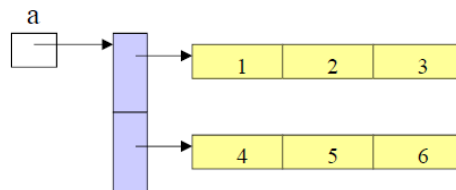
- Rows can be of arbitrary length



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

- Initialisation

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



# Looking back ..



- Scalar data types
  - „basic data types“ int, byte, short, int, long, float, double, boolean, char
  - Variable contains value
- Aggregated data types
  - More than a single basic data organized through a single name
  - cp. arrays ...

# Looking back ...



- Reference data type
  - variable stores reference / address
  - not a value
- In Java
  - basic data type -> by value
  - everything else -> by reference

# About „everything else“ ...



- Basically a combination ...
  - of fundamentalen Datentypen
  - in a (sometimes) complex structure
- Different concepts in different languages
  - Pascal: Record
  - C: struct
  - Java / Python: class

# Java Classes

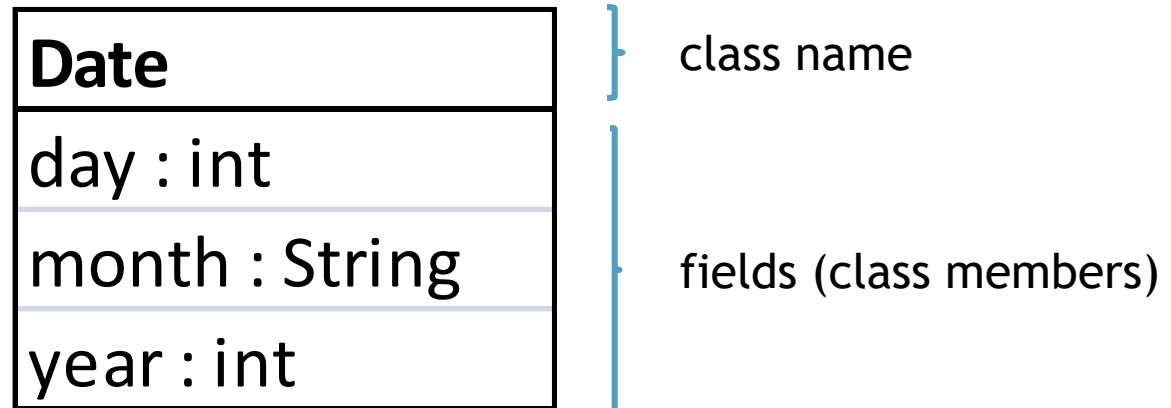


- Example: Store a data in a single structure.
  - day, month, year, ...
- Basic data types not practical ...
  - storing more than one
  - return values of functions
  - comparing to other dates

# Java Classes



- Combine all necessary variables in one structure:





# Data Type Class



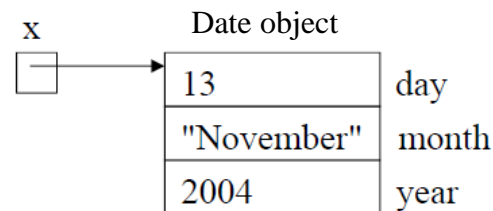
- Declaration
- Data type usage
- Access

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

fields of class date

```
Date x, y;
```

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



Date variables are references /  
addresses to objects.

# Objects



- Class is like a template
  - from which instances (objects) are created
- Objects (instances) of a class have to be created explicitly before use.
  - variable otherwise have the value `null`

# Objects



`Date x, y;`

reserves memory for the address

`x, y` have value null



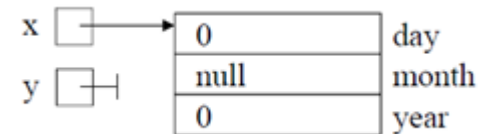
## Instantiation

`x = new Date();`

creates a new Date object and assigns its address to `x`.

Initial values are

0, null, false or `'\u0000'`

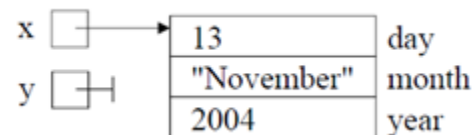


## Usage

`x.day = 13;`

`x.month = „November“`

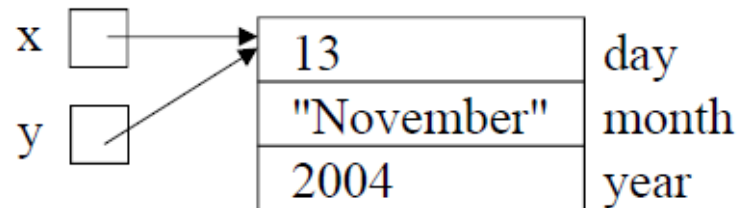
`x.year = 2004;`



# Assignments

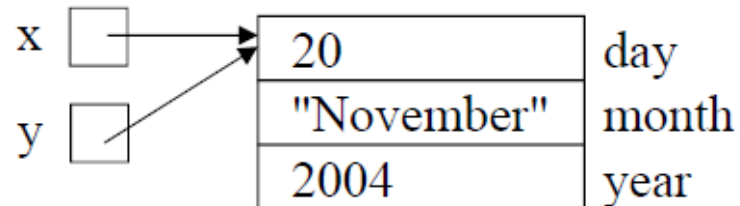


`y = x;`



Reference / address  
assignment

`y.day = 20;`



changes `x.day` too!

# Assignments



```
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, same type

a1 = a2; // ok, same type

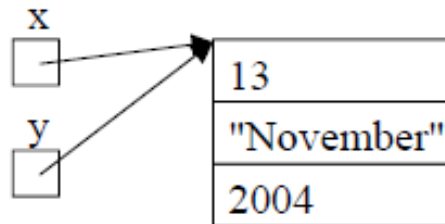
d1 = a2; // not ok, different type (although structure is the same)

# Comparing references

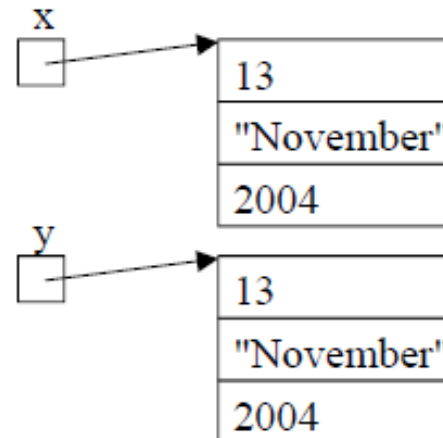


- $x == y$  und  $x != y$  ... compares references
- $<$ ,  $<=$ ,  $>$ ,  $>=$  ... not applicable

$x == y$  returns true



$x == y$  returns false



# Compares actual values



- Has to be implemented by method.

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

# Declaration of Classes



## Single file

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

MainProgram.java

Compile

\$> javac MainProgram.java

## Multiple files

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

C1.java

C2.java

MainProgram.java

Compile

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

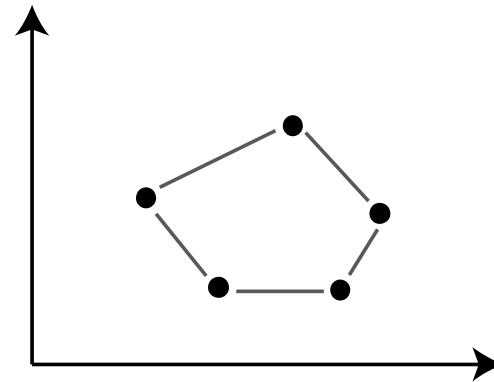
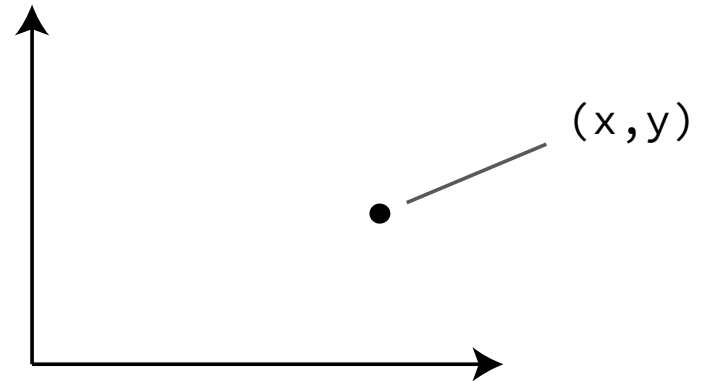


# What can we do with classes?



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

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



# What can we do with classes?



- Classes can use other classes
  - and extend on that

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

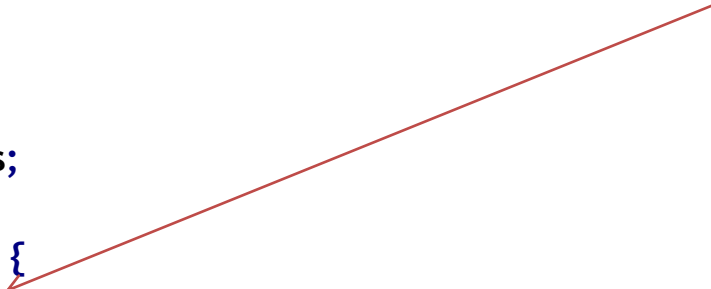


# What can we do with classes?



- Implement methods with multiple return values

```
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);  
    }  
}
```



# What can we do with classes?



- Combination of classes and 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"  
        // ...  
    }  
}
```

# Object Oriented Programming



- What we assumed up to now
  - classes combine data types to structures
  - works with base data types, arrays and other classes.
- Object oriented programming
  - class = data + methods

# Example: Position Class



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

// ... Usage

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

- Methods are defined locally
  - without static
- Each object has its own state
  - pos1 = new Position()
  - pos2 = new Position()
  - ...

# Example: Position Class



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

- Usage of Parameters in methods ..
- .. and return values

# Example: Position Class



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

- `this` is used to access fields of the object (object scope)
- Without `this` the local variable would be used.



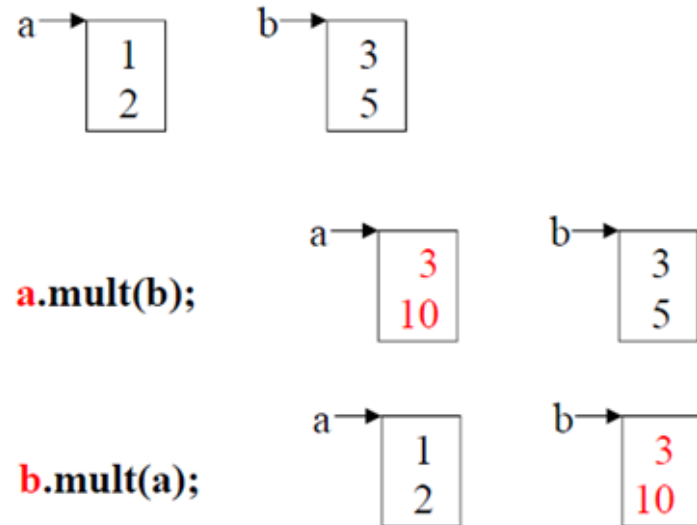
# Example: Fraction Class



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

    /**
     * Multiply this fraction with another one.
     *
     * @param f the second factor
     */
    void mult(Fraction f) {
        n = f.n * n;
        d = f.d * d;
    }

    /**
     * Add a fraction to this one.
     *
     * @param f the fraction to add to this one.
     */
    void add(Fraction f) {
        d = f.d * d; // bring to same denominator
        n = f.n * d + n * f.d;
    }
}
```



Only one object changes!

# UML Notation



<b>Fraction</b>	<i>class name</i>
int z	<i>fields</i>
int n	
void mult(Fraction f)	<i>methods</i>
void add(Fraction f)	

<b>Fraction</b>	<i>simple form</i>
z	
n	
mult(f)	
add(f)	

# Constructors



- Special methods
  - are called upon object creation
  - used for initialisation of values
  - have the same name as the class
  - without function type or `void`
  - can have parameters
  - can be overloaded

# Constructors



```
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);
```

- calls matching constructors

# Constructors...



- Example: time class
- Example: position class

# Class example: java.lang.String



- Char-Array vs. Strings
  - `char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };`
  - `String helloString = new String(helloArray);`
  - `System.out.println(helloString);`
- Length of a String-Object
  - `helloString.length()`
- Reading chars from Strings
  - `helloString.charAt(2) // result: 'l',`
  - `helloString.getChars(...)`
  - `helloString.toCharArray()`

# Example: 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 concatenation
  - `string1.concat(string2)`
  - `"Hello ".concat("World!")`
  - `"Hello " + "World!,,`
- Note: The String class is immutable



# Strings $\rightleftharpoons$ Numbers



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

# String - Manipulation



- Substring
  - `String substring(int beginIndex, int endIndex)`
  - `String substring(int beginIndex)`
- Lower and upper case
  - `String toLowerCase()`
  - `String toUpperCase()`
- trim white space at the end of a String
  - `String trim()`

# String - Search



- Search for char or String in Strings
  - `int indexOf(int ch)`
  - `int lastIndexOf(int ch)`
  - `int indexOf(int ch, int fromIndex)`
  - `int lastIndexOf(int ch, int fromIndex)`
- With String as argument
  - `int indexOf(String str)`
  - ...

# Example



```
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 - Add. Methods



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

more information:

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

# CharSequence

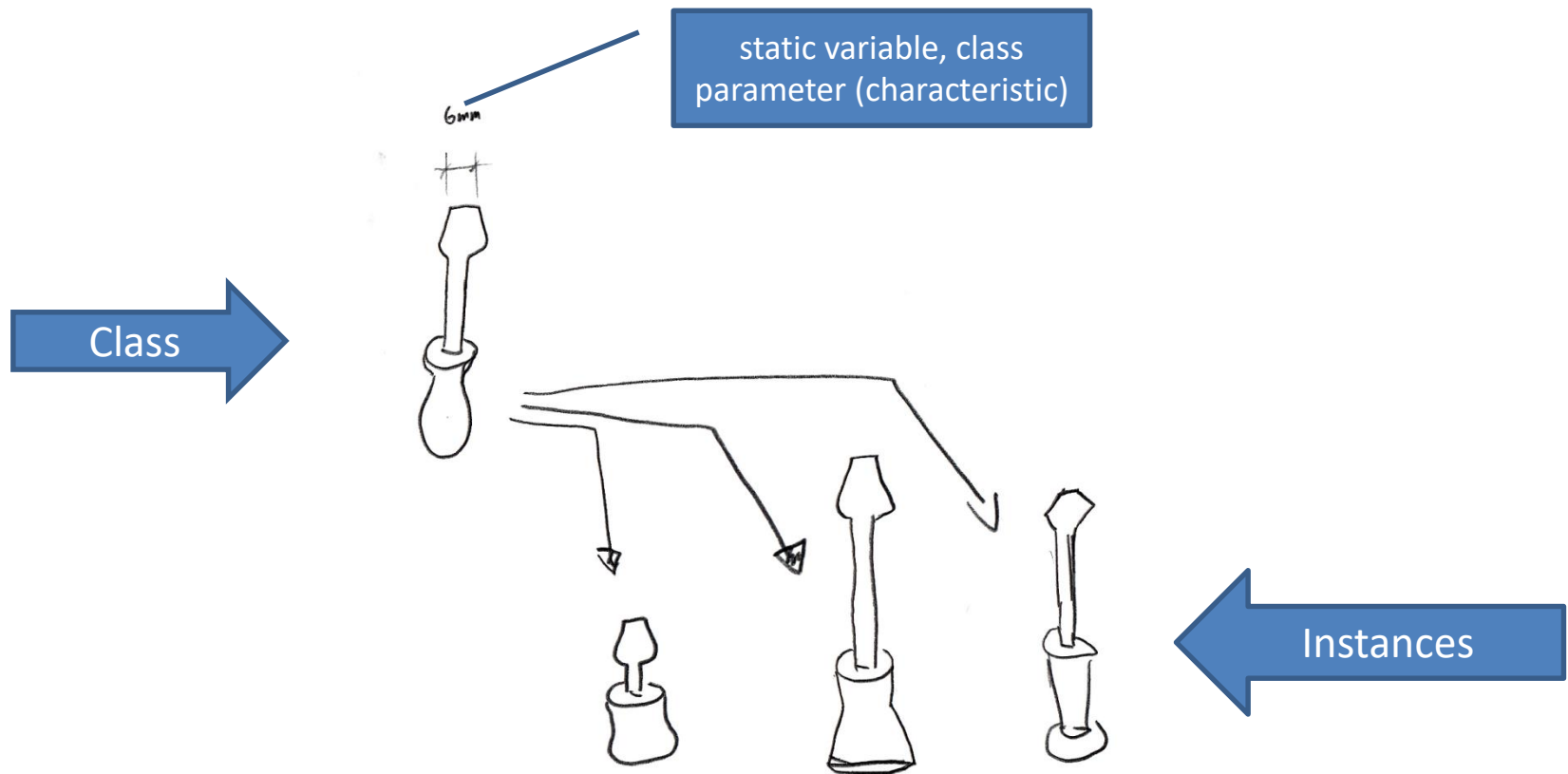


- String is immutable
  - Manipulations are expensive
- CharSequence is Interface String-like classes
  - StringBuilder
  - StringBuffer

more Information:

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

# static



# static



```
class Window {  
    int x, y, w, h;           // object fields (in each object different)  
    static int border;        // static (class) field (only once per class)  
  
    // constructor (initialisation of the object)  
    Window(int x, int y, int w, int h) {...}  
  
    // class constructor (initialisation of the class)  
    → static {  
        border = 3;  
    }  
  
    // method of the object (instance)  
    void redraw () {...}  
  
    // static (class) method, operates on class level, not object  
    → static void setBorder (int n) {border = n;}  
}
```



# static



- Object methods can access static (class) fields
  - `redraw()` can access `border`
- Static (class) methods can't access object fields
  - `setBorder()` can't access `x`

class Window

<code>border</code>
<code>setBorder()</code>
<code>class constructor</code>

Window instances ...

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

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

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

# static



## Order of execution

- Loading of class Window
  - class fields are created - border
  - class constructor is called
- At instantiation time - `new Window(...)`
  - object fields are created - `x`, `y`, `w`, `h`
  - object constructor is called

# static



- Accessing static members by class name
  - `Window.border = ...; Window.setBorder(3);`
  - Static methods can access them directly  
`border = ...; setBorder(3);`
- Non static members: instance variable
  - `Window win = new Window(100, 50);`  
`win.x = ...; win.redraw();`
  - Non static methods can access object variables directly  
`x = ...; redraw();`

# static



- Note: static fields will not be collected by the garbage collection.
- Therefore, prioritize locality of data!
- Cp. later lessons (object oriented programming, software engineering)

# Example for static: `java.lang.ath`



- Java provides additional mathematical support in the class `Math`
- Each method in `Math` is static
  - optional static import
  - `import static java.lang.Math.*;`
  - method calls like global functions, eg. `cos(x)`

# Java Math Constants



- Math.E
  - Euler's number  $e$
- Math.PI
  - $\pi$

# Java Math Basics



- absolute values
  - `int Math.abs(int value)`
  - also for `double`, `long`, `float`
- rounding up and down
  - `double Math.ceil(double value)`
  - `double Math.floor(double value)`
- rounding
  - `long Math.round(double value)`
  - `int Math.round(float value)`

# Java Math Basics



- Minimum of two values
  - `double Math.min(double arg1, double arg2)`
  - also for `float`, `long`, `int`
- Maximum of two values
  - `double Math.max(double arg1, double arg2)`
  - also for `float`, `long`, `int`



# Java Math Exp & Log



- Exponential function and logarithm
  - `double Math.log(double value)`
  - `double Math.exp(double value)`
- Power and root
  - `double Math.pow(double base, double exp)`
  - `double Math.sqrt(double value)`

# Java Math Trigonometrie



- trigonometric functions
  - `double Math.sin(double value)`
  - auch für `cos`, `tan`, `asin`, `acos`, `atan`
- angle of a vector (polar coordinates)
  - `double Math.atan2(double x, double y)`

# Example: ASCII sine wave



```
public static void main(String[] args) {  
    for (double d = 0; 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 - Random



- `double Math.random()`
  - generates pseudo random number  $0 \leq x < 1$
  - sufficient for single numbers, not sequences
- Other value ranges
  - eg. `Math.random() * 10.0`

# Example: Random Names



```
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(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];
    }
}
```

# More Math



- JavaDoc
  - <https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html>
- BigInteger
  - for arbitrarily big integers
- BigDecimal
  - for arbitrarily precise decimal numbers

# Example: Stack & Queue



- Stack
  - push(x) ... puts on top of the stack
  - pop() ... removes and returns topmost element
  - LIFO data structure == last in first out
- Queue (buffer)
  - put(x) ... adds x at the end of the queue
  - get() ... removes and returns first element
  - FIFO data structure == 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



## Usage:

```
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
```

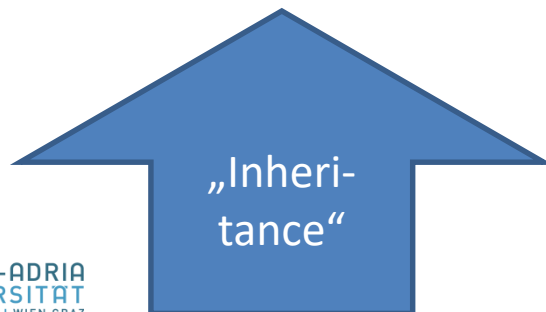
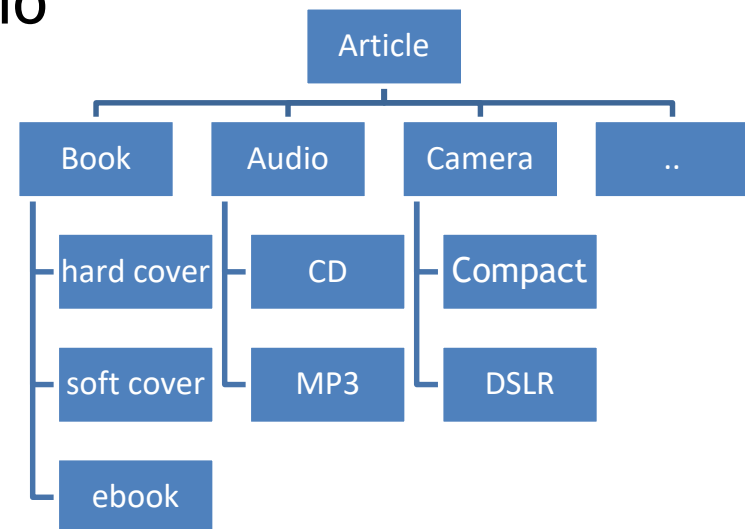
# Classification



Real world concepts can often be ordered in a hierarchy

Example:

- ebook has all characteristics of a book  
ebook has all characteristics of an article
- CD and MP3 both are of type Audio
- Book, Audio and Camera are of type Article



# Inheritance



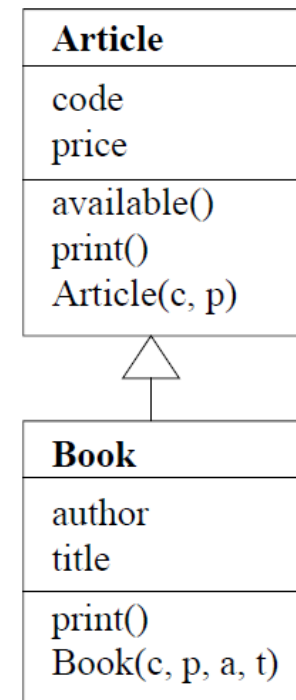
```
class Article {  
    int code;  
    int price;  
  
    boolean available() {...}  
    void print() {...}  
  
    Article(int c, int p) {...}  
}
```

superclass

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

subclass

**inherits:** code, price, available, print  
**adds:** author title, constructor  
**overrides:** print



All classes extend Object, even if no superclass is given.

# Overriding methods



```
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;  
    }  
}
```

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

→ creates Book object

→ Book constructor

→ Article constructor

→ set Book fields

book.print();

→ print() from Book object

→ print() from Article

→ Out.print(...)

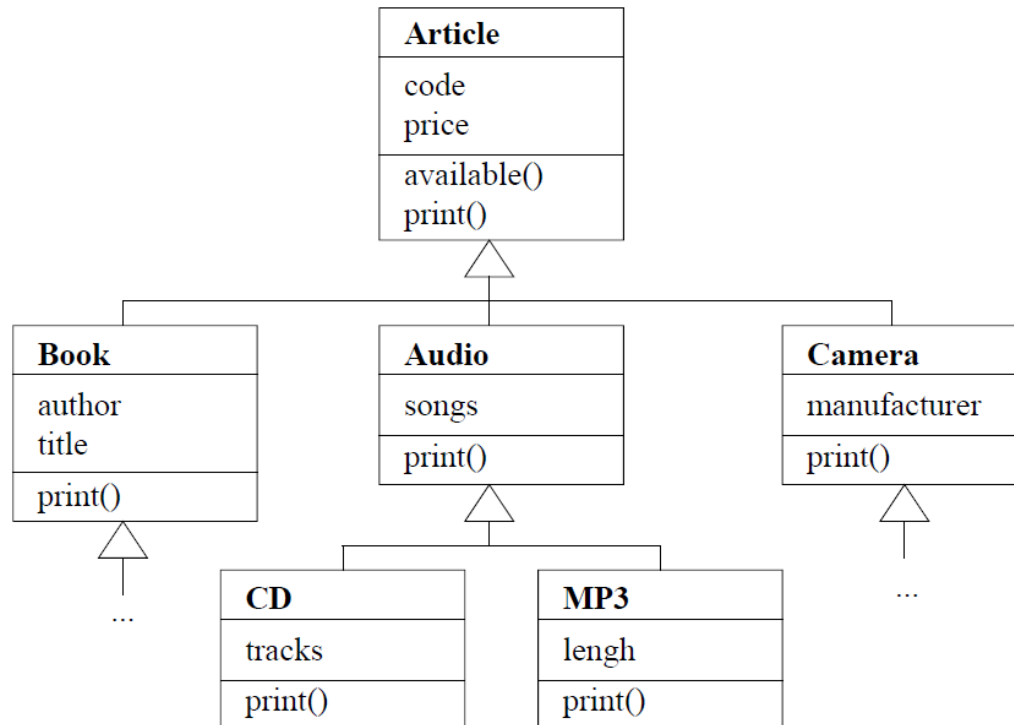
# Addendum



super can only access the direct super class.

- Otherwise the principle of inheritance is violated
  - by ignoring the super class.

# Class Hierarchies



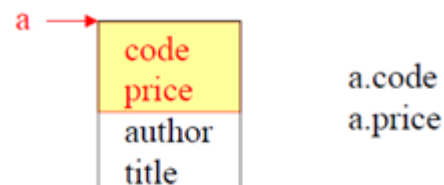
Each book is an Article, but not each Article is a book

# Inter-Class Compatibility



- Subclasses are specializations of superclasses
- Book objects can be assigned to Article variables

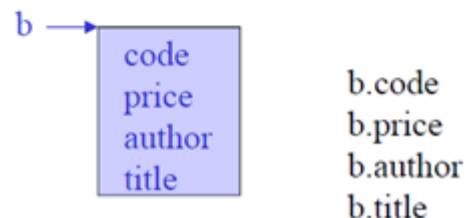
```
Article a = new Book(code, price, author, title);
```



Only Article fields are accessible now.

```
if (a instanceof Book)  
    Book b = (Book) a;
```

runtime type test and cast

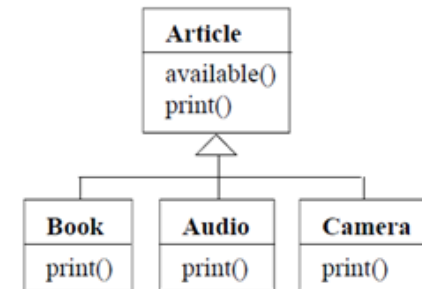
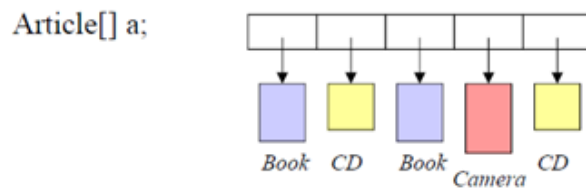


Now all fields are accessible.

# Dynamic Binding



- Heterogeneous data structure



- All instances are of type Article and can be used as such:

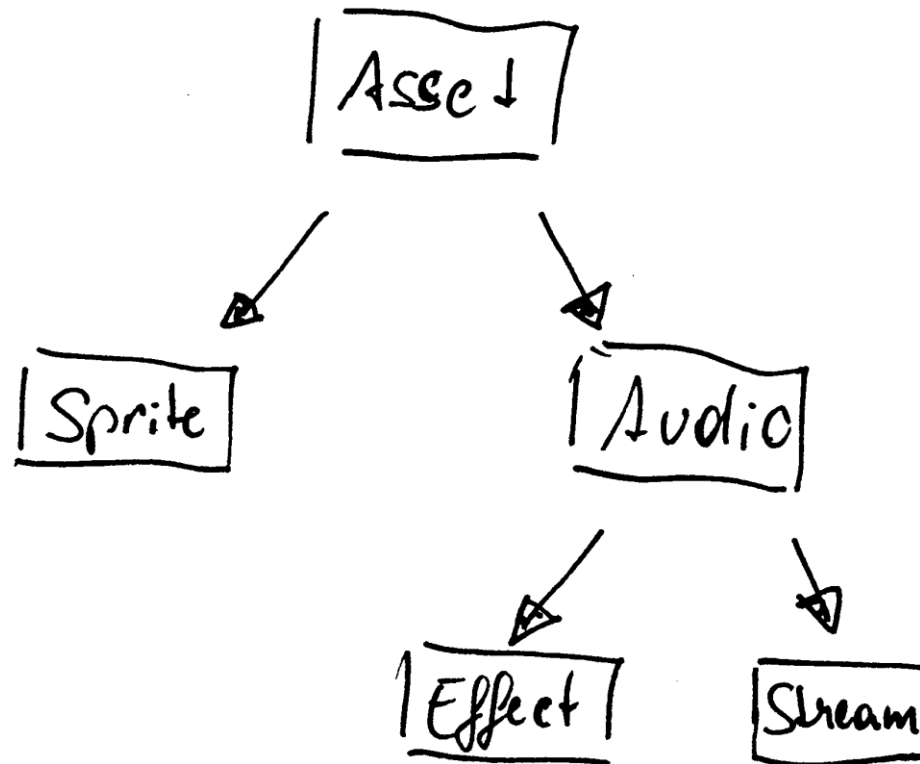
```
void printArticles() {  
    for (int i = 0; i < a.length; i++) {  
        if (a[i].available()) {  
            a[i].print();  
        }  
    }  
}
```

available() from the Article class  
print() from Book, Audio or Camera.

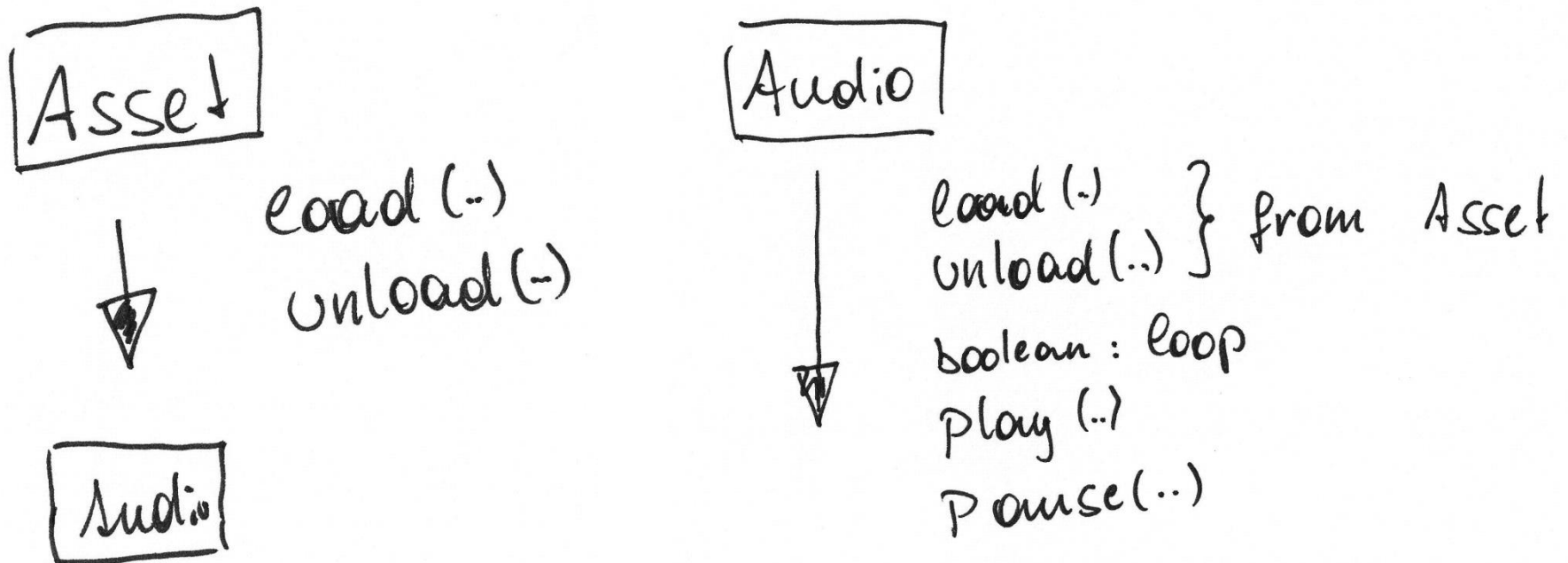
- Dynamic binding: obj.print() calls the method of the actual instance.



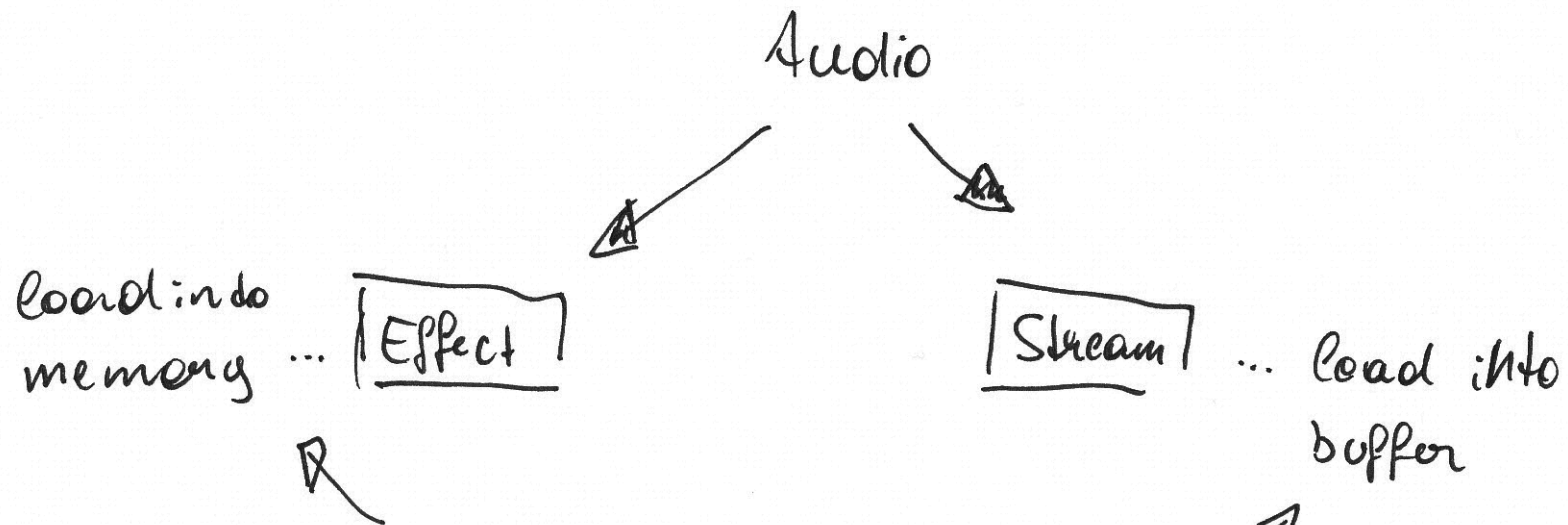
# Example ...



# Example



# Example



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

# Additional Concepts



## Keyword **abstract**

- defines that each subclass has such a member,
- but does not implement / provide it
  - it has to be implemented by the subclass