

11. Objektorientierung

11.1 Methoden

11.2 Konstruktoren

11.3 static

11.4 Beispiele für Klassen

11.5 Vererbung

11.6 Dynamische Bindung

11.7 Klasse Object

11.8 final

11.9 Abstrakte Klassen

11.10 Interfaces

11.11 Anonyme Klassen

Klasse = Daten + Methoden

Beispiel: Positionsklasse

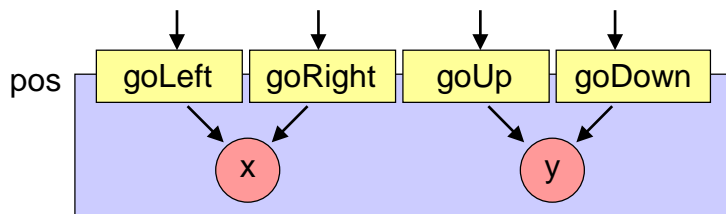
```
class Position {
    private int x;
    private int y;

    void goLeft()    { x = x - 1; }
    void goRight()   { x = x + 1; }
    void goUp()      { y = y - 1; }
    void goDown()    { y = y + 1; }
}
```

Methoden sind

- lokal zur Klasse *Position*
- ohne *static* deklariert (siehe später)

Position-Objekt ist eine **Black-Box**



Objekte haben einen *Zustand*,
der über Methoden manipuliert wird

Benutzung

```
Position pos = new Position();
pos.goRight(); // pos.x == 1, pos.y == 0
pos.goDown(); // pos.x == 1, pos.y == 1
pos.goDown(); // pos.x == 1, pos.y == 2
...
```

ruft *goDown*-Methode von *pos* auf

Jedes Objekt hat seinen eigenen Zustand

```
Position pos2 = new Position();
pos2.goUp(); // pos2.x == 0, pos2.y == -1
pos2.goLeft(); // pos2.x == -1, pos2.y == -1
...
```

pos.goRight();

Man sagt:

- *pos* bekommt die Nachricht (message) *goRight*
- *pos* ist der Empfänger der Nachricht *goRight*

Schlüsselwort *this*



Methoden können Parameter haben

```
class Position {  
    private int x;  
    private int y;  
  
    void goLeft(int n)    { x = x - n; }  
    ...  
}
```

```
Position pos = new Position();  
pos.goLeft(3);    // pos.x == -3, pos.y == 0  
...
```

Schlüsselwort *this*

```
class Position {  
    private int x;  
    private int y;  
  
    void goLeft(int x) { this.x = this.x - x; }  
    ...  
}
```

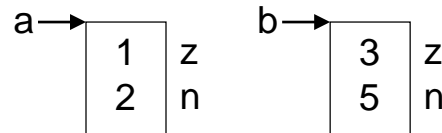
this bezeichnet das Objekt,
auf das *goLeft* angewendet wird
(hier nötig, um Feld *x* vom
Parameter *x* zu unterscheiden)

Beispiel: Bruchzahlenklasse

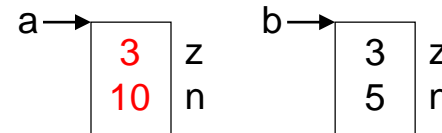


```
class Fraction {  
    int z; // Zähler  
    int n; // Nenner  
  
    void mult (Fraction f) {  
        // multipliziert this * f  
        z = z * f.z;  
        n = n * f.n;  
    }  
  
    void add (Fraction f) {  
        // addiert this + f  
        z = z * f.n + f.z * n;  
        n =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

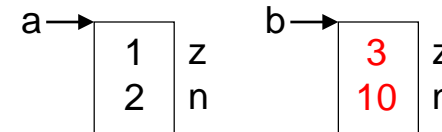
Fraction a = new Fraction(); a.z = 1; a.n = 2;
Fraction b = new Fraction(); b.z = 3; b.n = 5;



a.mult(b);



b.mult(a);



Es wird immer der Zustand des Empfängers verändert!

Grafische Notation für Klassen

UML-Notation (Unified Modeling Language)

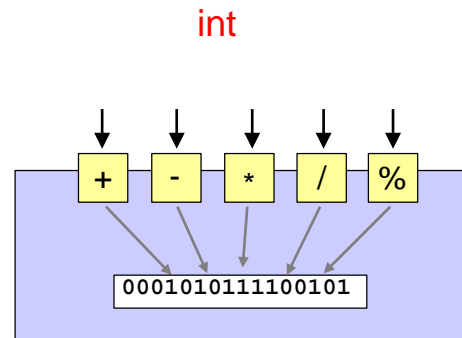
Fraction	<i>Klassename</i>
int z int n	<i>Felder</i>
void mult (Fraction f) void add (Fraction f)	<i>Methoden</i>

Vereinfachte Form

Fraction	<i>falls weniger Details gewünscht oder nötig</i>
z n	
mult(f) add(f)	

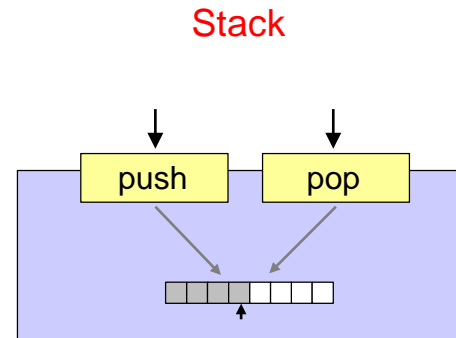
Klassen sind Abstrakte Datentypen

Konkreter Datentyp



in die Sprache eingebaut

Abstrakter Datentyp (ADT)



selbst implementiert

Programmiersprache kann mit ADTs (Klassen) beliebig erweitert werden.

Immer mehr Sprachkonzepte wandern heute in die Klassenbibliothek

Beispiel *String*

- in Pascal und C: Teil der Sprache
- in Java: Klasse

11. Objektorientierung

11.1 Methoden

11.2 Konstruktoren

11.3 static

11.4 Beispiele für Klassen

11.5 Vererbung

11.6 Dynamische Bindung

11.7 Klasse Object

11.8 final

11.9 Abstrakte Klassen

11.10 Interfaces

11.11 Anonyme Klassen

Konstrukturen



Spezielle Methoden, die beim Erzeugen eines Objekts automatisch aufgerufen werden

```
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) {...}  
}
```

- dienen zur Initialisierung eines Objekts
- heißen wie die Klasse
- ohne Funktionstyp und ohne void
- können Parameter haben
- können überladen werden

Aufruf

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

1. legt neues *Fraction*-Objekt an
2. ruft für dieses Objekt den passenden Konstruktor auf

Feld-Zuweisungen im Konstruktor überschreiben
Feld-Zuweisungen bei der Deklaration

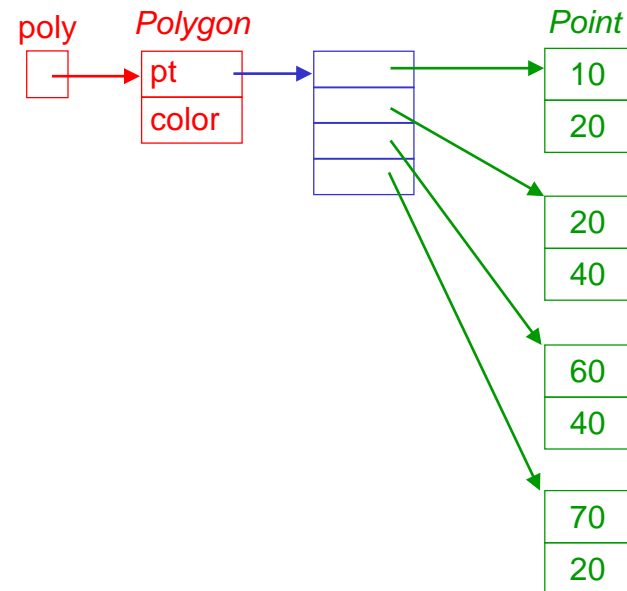
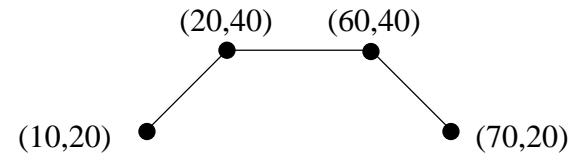
Bsp: Polygon-Aufbau mit Konstruktoren



```
class Point {  
    int x, y;  
    Point (int x, int y) { this.x = x; this.y = y; }  
}
```

```
class Polygon {  
    Point[] pt;  
    int color;  
    Polygon (Point[] pt, int color) {  
        this.pt = pt; this.color = color;  
    }  
}
```

```
class Program {  
    ...  
    Polygon poly = new Polygon(  
        new Point[] {  
            new Point(10, 20),  
            new Point(20, 40),  
            new Point(60, 40),  
            new Point(70, 20)  
        },  
        RED  
    );  
    ...  
}
```



11. Objektorientierung

11.1 Methoden

11.2 Konstruktoren

11.3 static

11.4 Beispiele für Klassen

11.5 Vererbung

11.6 Dynamische Bindung

11.7 Klasse Object

11.8 final

11.9 Abstrakte Klassen

11.10 Interfaces

11.11 Anonyme Klassen

static

Felder, Methoden und Konstruktoren können *static* oder *nicht static* sein

```
class Window {
    int x, y, w, h;
    static int border;

    void redraw () {...}
    static void setBorder (int n) {border = n;}

    Window(int x, int y, int w, int h) {...}
    static { ... }
}
```

Objektfelder (in jedem *Window*-Objekt vorhanden)
Klassenfeld (nur einmal pro Klasse vorhanden)

Objektmethode (auf Objekte anwendbar)
Klassenmethode (auf Klasse *Window* anwendbar)

Objektkonstruktor (zur Initialisierung von Objekten)
Klassenkonstruktor (zur Initialisierung der Klasse)

Klasse Window

border
setBorder() Klassenkonstruktor

Window-Objekt

x y w h
redraw() Window()

Window-Objekt

x y w h
redraw() Window()

Window-Objekt

x y w h
redraw() Window()

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

static (Fortsetzung)

Was geschieht wann?

Beim Laden der Klasse *Window* (i.a. am Programmbeginn)

- Klassenfelder werden angelegt (*border*)
- Klassenkonstruktor wird aufgerufen

Beim Erzeugen eines *Window*-Objekts (`new Window(...)`)

- Objektfelder werden angelegt (*x, y, w, h*)
- Objektkonstruktor wird aufgerufen

Zugriffe

Zugriff auf *static*-Elemente über den Klassennamen

- `Window.border = ...; Window.setBorder(3);`
- Methoden der Klasse *Window* können Klassennamen weglassen (`border = ...; setBorder(3);`)

Zugriff auf *non-static*-Elemente über einen Objektnamen

- `Window w = new Window(0, 0, 200, 400);`
`w.x = ...; w.redraw();`
- Methoden der Klasse *Window* können auf eigene Elemente direkt zugreifen (`x = ...; redraw();`)

static (Fortsetzung)

```
class Window {
```

```
    static int border;
```

```
    static {...}
```

```
    static void setBorder(int n) {...}
```

```
    int x, y, w, h;
```

```
    Window(...) {...}
```

```
    void redraw() {...}
```

```
    void foo() {
```

```
        x = 0; redraw();
```

```
        border = 1; setBorder(1);
```

```
    }
```

```
}
```

statische Programmelemente

← angelegt am Programmanfang (wenn die Klasse geladen wird)

← aufgerufen am Programmanfang (wenn die Klasse geladen wird)

nichtstatische Programmelemente

← angelegt, wenn das *Window*-Objekt erzeugt wird

← aufgerufen, wenn das *Window*-Objekt erzeugt wird

} Zugriff auf eigene Elemente ohne Qualifikation

```
...
```

```
Window w = new Window(0, 0, 100, 50);
```

```
w.x = 0; w.redraw();
```

```
Window.border = 1; Window.setBorder(1);
```

```
...
```

} Zugriff auf fremde Elemente mit Qualifikation

Statische Felder leben während der gesamten Programmausführung!

11. Objektorientierung

11.1 Methoden

11.2 Konstruktoren

11.3 static

11.4 Beispiele für Klassen

11.5 Vererbung

11.6 Dynamische Bindung

11.7 Klasse Object

11.8 final

11.9 Abstrakte Klassen

11.10 Interfaces

11.11 Anonyme Klassen

Beispiel: Stack und Queue

Stack (Stapel, Kellerspeicher)

push(x);
x = pop();

fügt x hinten an den Stack an
entfernt und liefert hinterstes Stackelement

push(3);
push(4);
x = pop();
y = pop();



// x == 4

// y == 3

LIFO-Datenstruktur
(last in first out)

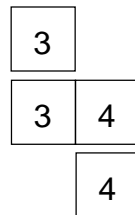


Queue (Puffer, Schlange)

put(x);
x = get();

fügt x hinten an die Queue an
entfernt und liefert vorderstes Queueelement

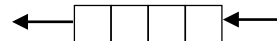
put(3);
put(4);
x = get();
y = get();



// x == 3

// y == 4

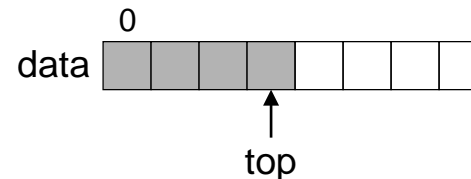
FIFO-Datenstruktur
(first in first out)



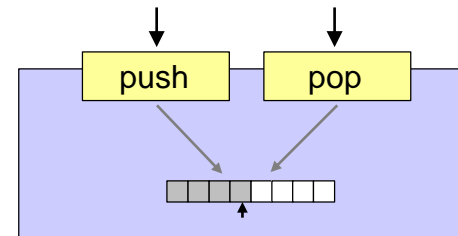
Klasse Stack



```
class Stack {  
    int[] data;  
    int top;  
  
    Stack (int size) {  
        data = new int[size]; top = -1;  
    }  
  
    void push (int x) {  
        if (top == data.length - 1) {  
            Out.println("-- overflow");  
        } else {  
            top++; data[top] = x;  
        }  
    }  
  
    int pop () {  
        if (top < 0) {  
            Out.println("-- underflow"); return 0;  
        } else {  
            return data[top--];  
        }  
    }  
}
```



Baustein



Schnittstelle

Zustand

Benutzung

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

Funktionen können auch wie Prozeduren
aufgerufen werden

```
s.push(3);  
s.pop(); // Rückgabewert wird verworfen
```


Klasse Queue



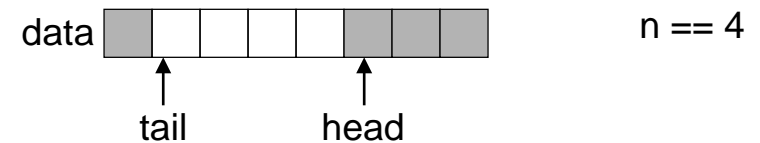
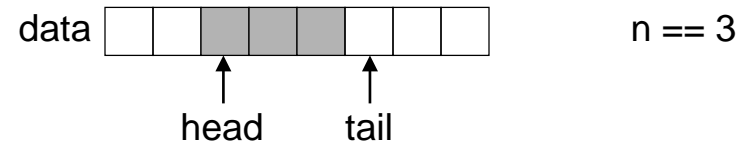
```
class Queue {  
    int[] data;  
    int head, tail, n;
```

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

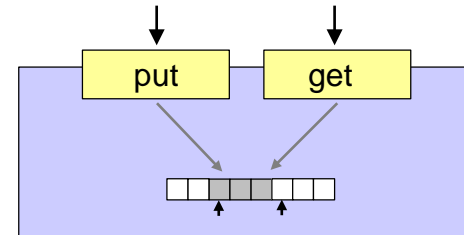
```
    void put (int x) {  
        if (n == data.length)  
            Out.println("-- overflow");  
        else {  
            data[tail] = x; n++;  
            tail = (tail+1) % data.length;  
        }  
    }
```

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

```
}
```



Baustein



Schnittstelle

Zustand

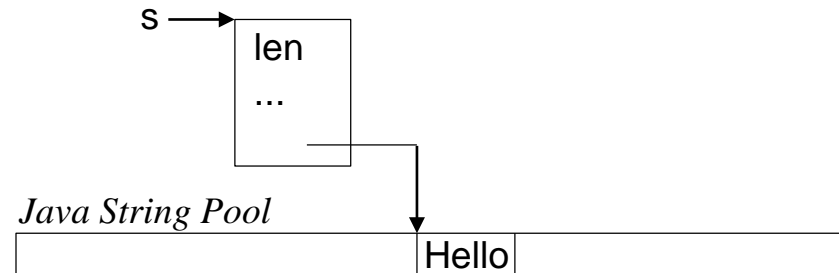
Benutzung

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

Klasse String



```
class String {  
  
    // Konstruktoren  
    String() {...}  
    String(String source) {...}  
    String(char[] a) {...}  
    ...  
    // Methoden  
    int    length() {...}  
    char   charAt(int i) {...}  
    boolean equals(String s) {...}  
    int    compareTo(String s) {...}  
    int    indexOf(String s) {...}  
    int    lastIndexOf(String s) {...}  
    ...  
}
```



String ist teilweise auch Teil der Sprache Java

- Stringkonstanten: "Hello"
- Stringverkettung: s1 + s2

Ähnlich ist auch *StringBuilder* eine Klasse

Beispiel: Klasse für Zahlenmengen

Als Bit-Folge in *int* darstellbar (Bereich 0 .. 31)

	31	0	
int s;	00000000000000000000000000000000	s == {}	
	000000000000000000000000000000001	s == {1}	Bit <i>i</i> gesetzt $\hat{=} i \in s$
	0000000000000000000000000000000011001	s == {1, 4, 5}	

```
class BitSet {
    private int data; // bit string
    void set (int i) {
        if (i >= 0 && i < 32) {
            data = data | (1 << i);
        }
    }
    boolean get (int i) {
        if (i >= 0 && i < 32) {
            return (data & (1 << i)) != 0;
        } else {
            return false;
        }
    }
    void clear() {
        data = 0;
    }
}
```

$s \leftarrow s \dot{\cup} \{i\}$

$i \in s?$

$s \leftarrow \{\}$

Bit-Operationen auf *long*, *int*, *short*, *byte*

$x \mid y$... bitweise Oder-Verknüpfung
 $x \& y$... bitweise Und-Verknüpfung
 $x \wedge y$... bitweise Xor-Verknüpfung

Logische Operationen auf *boolean*

$a \parallel b$... logische Oder-Verknüpfung
 $a \&\& b$... logisch Und-Verknüpfung

Benutzung

```
BitSet s = new BitSet();
s.set(5);
s.set(17);
if (s.get(5)) Out.println("5 in s");
```

11. Objektorientierung

11.1 Methoden

11.2 Konstruktoren

11.3 static

11.4 Beispiele für Klassen

11.5 Vererbung

11.6 Dynamische Bindung

11.7 Klasse Object

11.8 final

11.9 Abstrakte Klassen

11.10 Interfaces

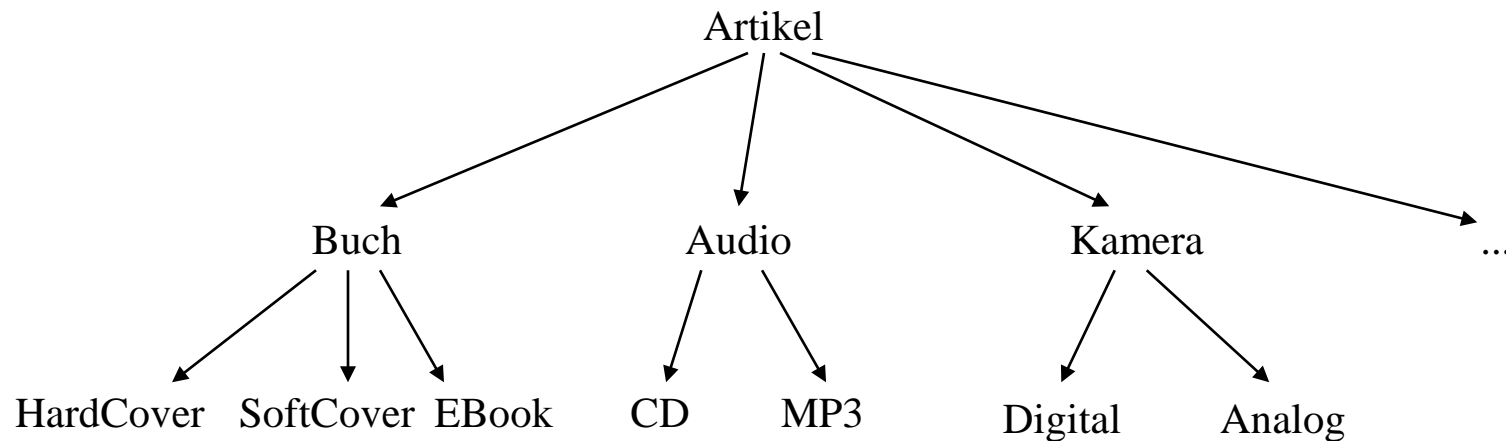
11.11 Anonyme Klassen

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) {...}  
}
```

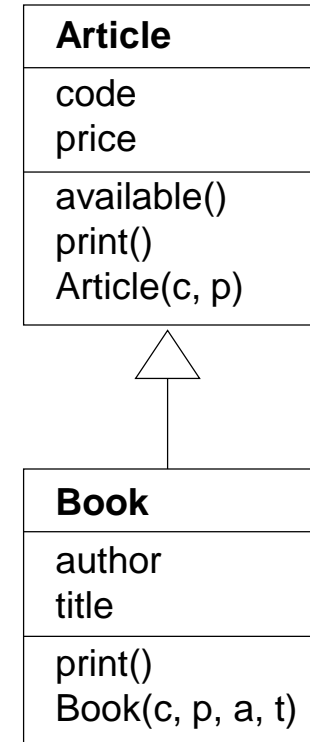
Oberklasse
Basisklasse

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

Unterklasse

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

Konstruktor wird nicht vererbt
Warum?



Java unterstützt nur *einfache Vererbung*, C++ auch *mehrfache Vererbung* (mehrere Oberklassen)

Überschreiben von Methoden



Neudeklaration einer Methode mit *gleicher Signatur* in der Unterklasse

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

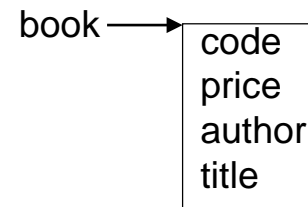
```
class Book extends Article {  
    ...  
    void print() {  
        super.print();  
        Out.print(" " + author + ": " + title);  
    }  
    Book(int c, int p, String a, String t) {  
        super(c, p);  
        author = a; title = t;  
    }  
}
```

Super-Call

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* code price
 - ⊢ Out.print(...); author: title

Ausgabe: code price author: title

Überschreiben ¹ Überladen



```
class Article {  
    ...  
    void print() {  
        Out.print(code + " " + price);  
    }  
    ...  
}
```

```
class Book extends Article {  
    ...  
    void print() {  
        super.print();  
        Out.print(" " + author + ": " + title);  
    }  
    void print(String s) {  
        Out.println(s);  
    }  
}
```

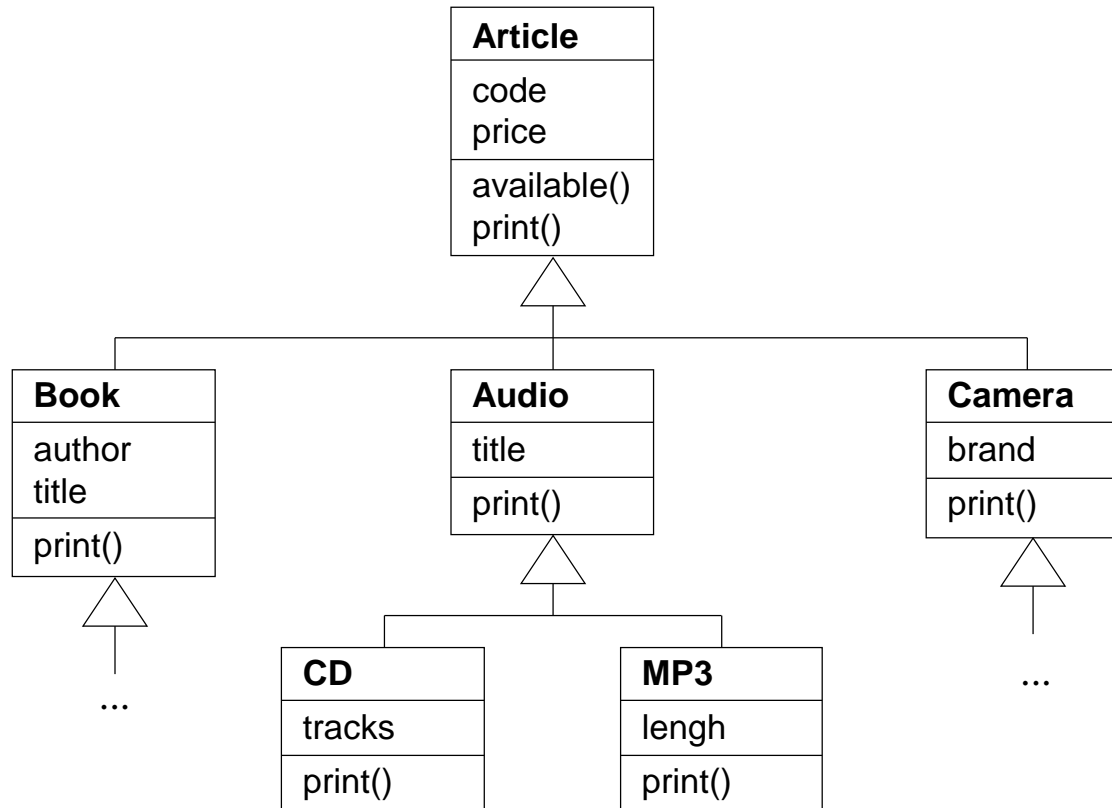
Überschreiben

Neudeklaration mit derselben Signatur

Überladen

Neudeklaration mit unterschiedlicher Signatur

Klassenhierarchien



Vererbung ist transitiv

CD ist Unterklasse von *Audio*

Audio ist Unterklasse von *Article*

=>

CD ist Unterklasse von *Article*

erbt: *code*, *price*, *title*, *available*, *print*

Vererbung ist eine Ist-Beziehung

Jedes Buch ist ein Artikel

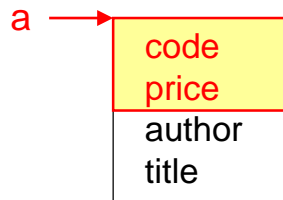
Aber: nicht jeder Artikel ist ein Buch

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

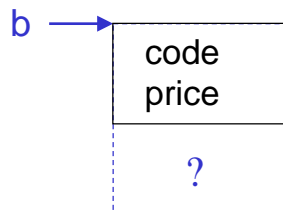
a.price

Statischer Typ von *a*: *Article* (Typ mit dem *a* deklariert ist)

Dynamischer Typ von *a*: *Book* (Typ des Objekts, auf das *a* verweist)

Zuweisung in umgekehrte Richtung ist verboten!

```
Book b = new Article(code, price);
```



über *b* wären Felder zugreifbar,
die es nicht gibt

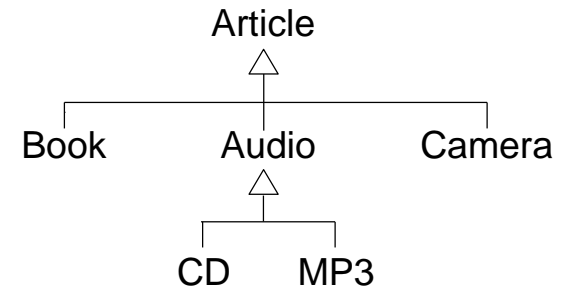
Jedes *Book* ist ein *Article*

aber nicht jeder *Article* ist ein *Book*

Beispiele für Zuweisungen



	<i>statischer</i>	<i>dynamischer</i>
	<i>Typ</i>	<i>Typ</i>
Article article = new Article();	Article	Article
Book book = new Book();	Book	Book
Audio audio = new Audio();	Audio	Audio
CD cd = new CD();	CD	CD
MP3 mp3 = new MP3();	MP3	MP3



Welche der folgenden Zuweisungen sind korrekt?

Was ist der statische/dynamische Typ der linken Seite?

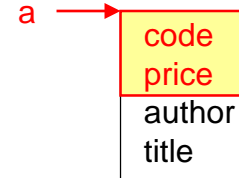
article = book;	✓	Article	Book
article = cd;	✓	Article	CD
mp3 = audio;	✗		
mp3 = cd;	✗		
book = cd;	✗		
audio = cd;	✓	Audio	CD

Typtest und Typumwandlung

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

$ST(a) \dots Article$

$DT(a) \dots Book$



Laufzeit-Typtest

```
if (a instanceof Book) ...
```

ist $DT(a)$ zumindest *Book*?

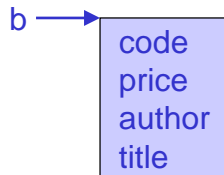
(d.h. *Book* oder Unterklasse: *Hardcover*, *EBook*)

```
a = null;
if (a instanceof Book) ...
```

liefert *false* (*null* verweist auf kein Objekt => kein DT)

Typumwandlung (Type Cast)

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



Gepriüfte Typumwandlung

```
if (a instanceof Book || a == null)
    ... betrachte ST(a) hier als Book ...
else
    Laufzeitfehler
```

Nach Typumwandlung ist Zugriff auf *Book*-Elemente möglich

```
((Book)a).title = ...;
```

11. Objektorientierung

11.1 Methoden

11.2 Konstruktoren

11.3 static

11.4 Beispiele für Klassen

11.5 Vererbung

11.6 Dynamische Bindung

11.7 Klasse Object

11.8 final

11.9 Abstrakte Klassen

11.10 Interfaces

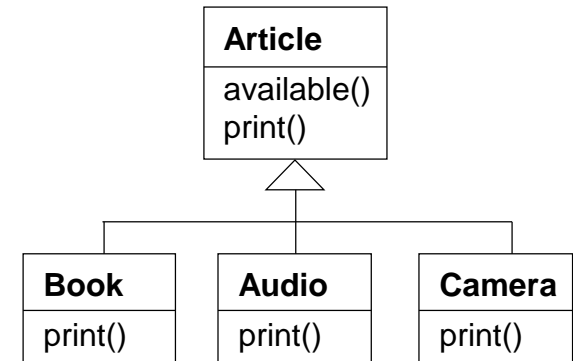
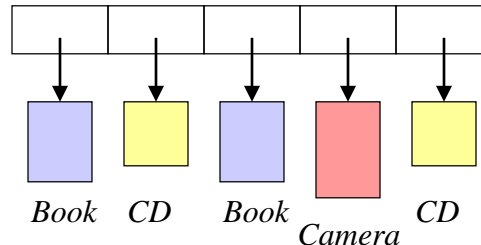
11.11 Anonyme Klassen

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 dynamischen Typs von *obj* auf

Man braucht sich nicht um die unterschiedlichen Varianten zu kümmern
Neue Varianten können hinzukommen, ohne dass man den Code ändern muss

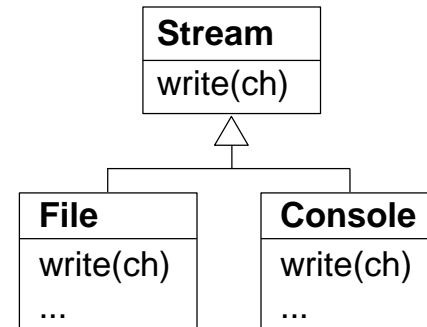
Dyn. Bindung fördert Wiederverwendung

Jeder Code, der mit Objekten einer Klasse *C* arbeiten kann,
kann automatisch auch mit Objekten der Unterklassen von *C* arbeiten

Beispiel: Datenströme

```
void writeText (Stream stream, String text) {  
    for (int i = 0; i < text.length(); i++) {  
        stream.write(text.charAt(i));  
    }  
}
```

↑ dynamische Bindung!



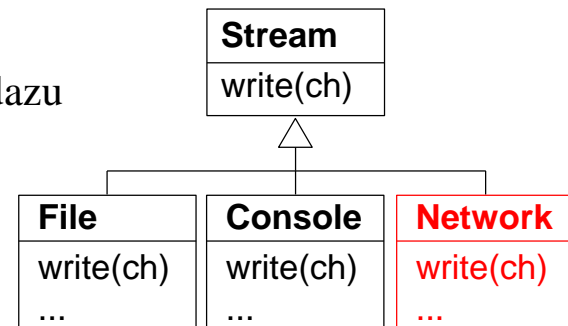
Benutzung

```
writeText(file, "Hello");  
writeText(console, "Hello");  
...
```

Angenommen, später kommt neue Stream-Art *Network* dazu

```
writeText(network, "Hello");  
...
```

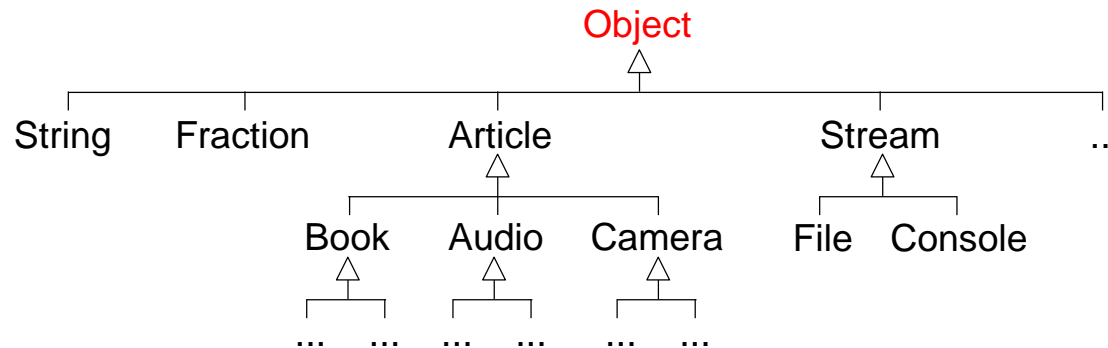
writeText kann mit *NetWork* arbeiten,
ohne geändert werden zu müssen



Oberste Basisklasse *Object*

```
class Article {
    int code;
    int price;
    ...
}
```

Wenn keine Basisklasse angegeben wird,
ist sie implizit *Object*



Alle Klassen sind mit *Object* kompatibel

z.B.: `Object obj = new Book(...);`

Alle Klassen erben von *Object*

Klasse Object



```
class Object {  
    Object() {...}  
    String toString() {...}  
    boolean equals(Object obj) {...}  
    int hashCode() {...}  
    Class getClass() {...}  
    Object clone() {...}  
    ...  
}
```

} sollten überschrieben werden

} werden unverändert vererbt

- | | |
|-------------------------------|--|
| <code>obj.toString()</code> | soll den Wert von <i>obj</i> als String zurückgeben |
| <code>obj.equals(obj2)</code> | soll true liefern, wenn die Werte von <i>obj</i> und <i>obj2</i> gleich sind
(<i>obj</i> == <i>obj2</i> bedeutet Zeigervergleich, nicht Wertvergleich) |
| <code>obj.hashCode()</code> | soll eine aus dem Wert von <i>obj</i> abgeleitete möglichst eindeutige Zahl liefern |
| <code>obj.getClass()</code> | liefert Information über den dynamischen Typ von <i>obj</i> |
| <code>obj.clone()</code> | liefert eine Kopie von <i>obj</i> |

Überschreiben von *equals* und *toString*



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

```
    String toString() {  
        return z + "/" + n;  
    }
```

```
    boolean equals(Object obj) {  
        Fraction f = (Fraction)obj;  
        return z == f.z && n == f.n;  
    }
```

```
}
```

erbt von *Object*

überschreibt *toString* und *equals*

Signatur darf beim Überschreiben nicht geändert werden

~~boolean **equals**(Fraction f) {~~

```
Fraction f1 = new Fraction(1, 2);  
Fraction f2 = new Fraction(1, 2);  
  
Out.println(f1.toString());  
Out.println(f1);  
  
if (f1 == f2) ...  
if (f1.equals(f2)) ...
```

gibt "1/2" aus

gibt "1/2" aus (Compiler fügt *.toString()* ein)

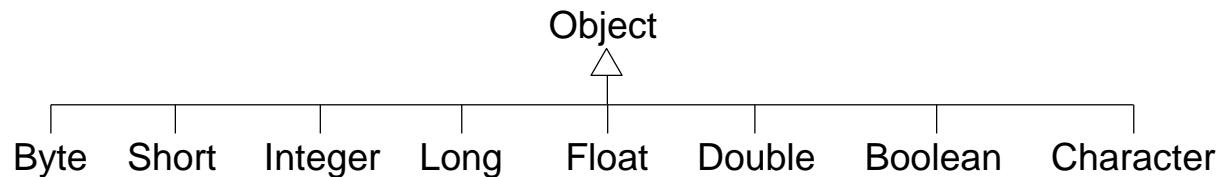
liefert *false* (Zeigervergleich)

liefert *true* (Wertvergleich)

Wrapper-Klassen für Werttypen

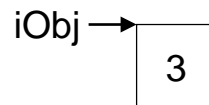
Problem: *int*, *long*, *double*, ... sind nicht direkt mit *Object* kompatibel

Wrapper-Klassen



Integer

```
Integer iObj = new Integer(3);
```



"wickelt" den Wert 3 in ein *Integer*-Objekt ein
das nun kompatibel zu *Object* ist:

```
Object obj = iObj;
```

```
int i = iObj.intValue();
```

"wickelt" den *int*-Wert wieder aus

Float

```
Float fObj = new Float(3.14f);
float f = fObj.floatValue();
```

Auto-Boxing/Unboxing

```
Integer iObj = 3; // wickelt int-Wert 3 in ein Integer-Objekt ein
```

```
Object obj = 3;
```

```
int x = (int) iObj; // wickelt int-Wert aus iObj wieder aus
```

```
int x = (int) obj;
```

Anwendungsbeispiel für Object



Allgemeiner Stack für Objekte beliebigen Typs

```
class GeneralStack {  
    private Object[] data = new Object[100];  
    private int top = -1;  
  
    public void push (Object obj) {  
        top++; data[top] = obj;  
    }  
  
    public Object pop () {  
        Object x = data[top]; top--;  
        return x;  
    }  
}
```

Benutzung

```
GeneralStack stack = new GeneralStack();
```

```
stack.push("a string");  
...  
String s = (String) stack.pop();
```

```
stack.push(article);  
...  
Article a = (Article) stack.pop();
```

```
stack.push(17);  
...  
int i = (int) stack.pop();
```

11. Objektorientierung

- 11.1 Methoden
- 11.2 Konstruktoren
- 11.3 static
- 11.4 Beispiele für Klassen
- 11.5 Vererbung
- 11.6 Dynamische Bindung
- 11.7 Klasse Object
- 11.8 final
- 11.9 Abstrakte Klassen
- 11.10 Interfaces
- 11.11 Anonyme Klassen

final-Klassen



Problem

```
class UserAccount {  
    ...  
    boolean login (String password) {  
        if (valid(password)) {  
            ...  
            return true;  
        } else {  
            return false;  
        }  
    }  
}
```

```
class FakeAccount extends UserAccount {  
    ...  
    boolean login (String password) {  
        ...  
        return true;  
    }  
}
```

```
UserAccount account = new FakeAccount();  
...  
account.login(...);
```

würde Passwort-Schutz umgehen

final als Schutz vor ungewollter Abänderung

```
final class UserAccount {  
    ...  
}
```

- Es dürfen *keine Unterklassen* gebildet werden
=> Schutz vor Abänderung der Semantik
- Methoden werden *statisch gebunden*
=> schneller

Beispiel: Klasse *String* ist *final*

final-Methoden



Statt Klasse *final* zu machen, kann man einzelne Methoden *final* machen

```
class UserAccount {  
    ...  
    String userName;  
    long loginTime, logoutTime;  
    final boolean login (String password) {  
        loginTime = System.currentTimeMillis();  
        ...  
    }  
    final void logout() {  
        ...  
        logoutTime = System.currentTimeMillis();  
    }  
    void print() {  
        Out.print(userName);  
    }  
}
```

```
class SpecialAccount extends UserAccount {  
    long activeTime() {  
        return logoutTime - loginTime;  
    }  
    void print() {  
        super.print();  
        Out.print(" active: ");  
        Out.print(activeTime() / 1000);  
    }  
}
```

- Unterklassen erlaubt
- neue Methoden hinzufügar
- *final*-Methoden
dürfen nicht überschrieben werden
- non-*final*-Methoden
dürfen überschrieben werden

Erlaubt feinere Einschränkung, welche Semantik unabänderlich sein soll

final-Felder/Variablen/Parameter



Dürfen nach Initialisierung nicht mehr verändert werden

```
class UserAccount {  
    static final int MAX = 100;  
  
    final String connection = "http://ssw.jku.at";  
    final String userName;  
  
    UserAccount (String name) {  
        userName = name;  
    }  
  
    void write (final char[] a) {  
        final int offset = In.readInt();  
        ...  
        int x = 2 * MAX;    // ok: x = 2 * 100;  
        MAX = ...;         // compile error  
        userName = ...;    // compile error  
        a = ...; // compile error  
        a[0] = ...;        // ok  
        offset = ...;      // compile error  
    }  
}
```

Konstante

- belegt keinen Speicherplatz

Feld

- belegt Speicherplatz
- darf nur in Deklaration oder Konstruktor initialisiert werden

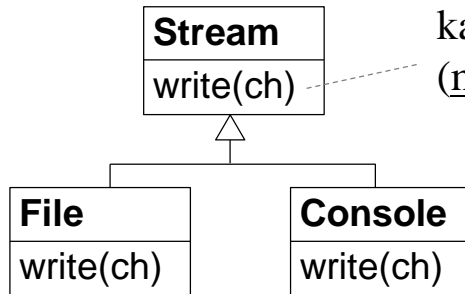
Variable/Parameter

- belegt Speicherplatz
- garantiert, dass Initialwert erhalten bleibt
- selten verwendet, kann aber u.U. Fehler vermeiden
- Parameter sollten generell nie verändert werden

11. Objektorientierung

- 11.1 Methoden
- 11.2 Konstruktoren
- 11.3 static
- 11.4 Beispiele für Klassen
- 11.5 Vererbung
- 11.6 Dynamische Bindung
- 11.7 Klasse Object
- 11.8 final
- 11.9 Abstrakte Klassen
- 11.10 Interfaces
- 11.11 Anonyme Klassen

Abstrakte Klassen



kann noch nicht sinnvoll implementiert werden
(muss überschrieben werden)

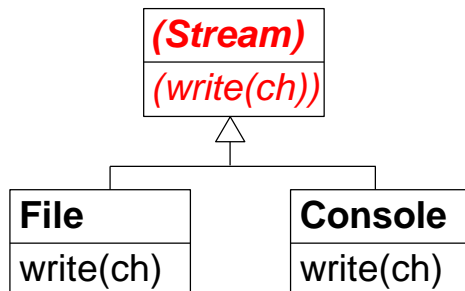
Wenn Klasse zumindest 1 abstrakte Methode hat,
muss sie selbst *abstract* sein

```
abstract class Stream {  
    abstract void write(char ch) ;  
}
```

Schlüsselwort *abstract*, kein Code!

```
class Console extends Stream {  
    void write(char ch) {  
        Out.print(ch);  
    }  
}
```

Abstrakte Klasse gibt Schnittstelle für zukünftige Unterklassen vor

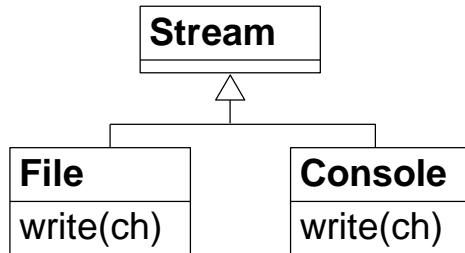


Abstrakte Klasse/Methode: kursiv und in Klammern

Abstrakte Klassen (Fortsetzung)



Warum kann man *write* in *Stream* nicht weglassen?



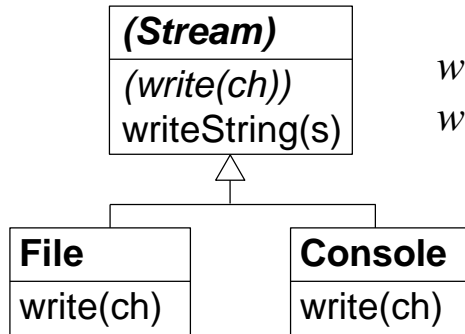
Weil sonst dynamische Bindung nicht funktionieren würde

```
Stream s = new Console();
s.write(ch);
```

← *Stream* hat kein *write*
Compiler würde einen Fehler melden

Alle dynamisch gebundenen Methoden müssen in *Stream* deklariert werden
(auch wenn nur als abstrakt)

Halbabstrakte Klassen



write abstrakt: muss überschrieben werden
writeString konkret: wird unverändert geerbt

```
abstract class Stream {
    abstract void write(char ch) ;
    void writeString(String s) {
        for (char ch: s.toCharArray()) {
            write(ch);
        }
    }
}
```

Dynamische
Bindung

```
class Console extends Stream {
    void write(char ch) {
        Out.print(ch);
    }
}
```

Benutzung

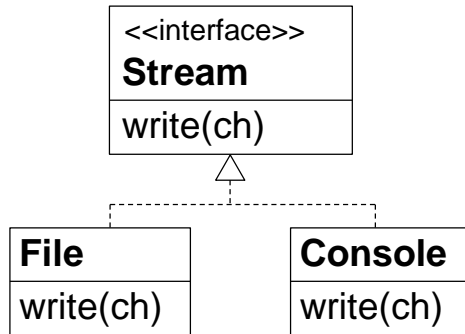
```
Stream s = new Console();
s.writeString("Hello");
```

ruft *writeString* von *Stream* auf,
das mittels dyn. Bindung *write* von *Console* aufruft

11. Objektorientierung

- 11.1 Methoden
- 11.2 Konstruktoren
- 11.3 static
- 11.4 Beispiele für Klassen
- 11.5 Vererbung
- 11.6 Dynamische Bindung
- 11.7 Klasse Object
- 11.8 final
- 11.9 Abstrakte Klassen
- 11.10 Interfaces
- 11.11 Anonyme Klassen

Interfaces



Interface entspricht vollständig abstrakter Klasse

- nur Methodensignaturen
- keine Datenfelder

```
interface Stream {
    void write(char ch);
}
```

Methoden sind automatisch *abstract*

```
class Console implements Stream {
    void write(char ch) {
        Out.print(ch);
    }
}
```

implements statt *extends*

muss alle Methoden von *Stream* überschreiben
(implementieren)

Klassen können beliebig viele Interfaces implementieren

Benutzung

```
Stream s = new Console();
s.write(ch);
```

dynamische Bindung wie bei Klassen

Interface Comparable



Zum Vergleichen von Objekten; Teil der Java-Bibliothek

```
interface Comparable {  
    int compareTo(Object other);  
}
```

a.compareTo(b)

< 0: wenn a < b

0: wenn a == b

> 0: wenn a > b

Anwendungsbeispiel Bruchzahlen vergleichbar machen

```
class Fraction implements Comparable {  
    int z;  
    int n;  
    ...  
    int compareTo(Object other) {  
        Fraction f2 = (Fraction)other;  
        // return this -f2 (nominator)  
        return z * f2.n - f2.z * n;  
    }  
}
```

```
Fraction a = new Fraction(1, 2); // 1/2  
Fraction b = new Fraction(2, 3); // 2/3  
Out.println(a.compareTo(b));    // -1
```

$$\frac{z}{n} - \frac{f2.z}{f2.n} = \frac{z * f2.n - f2.z * n}{n * f2.n}$$

$$1 * 3 - 2 * 2 = -1$$

Interface Comparable (Fortsetzung)



Sortieralgorithmen, die mit *Comparable* arbeiten, können auch mit *Fraction* arbeiten

Beispiel Bibliotheksklasse *Arrays* enthält *sort*-Methode

```
class Arrays {  
    static void sort(Object[] data) {...}  
    ...  
}
```

← nimmt an, dass Elemente *Comparable* unterstützen

```
Fraction[] data = new Fraction[] {  
    new Fraction(1, 2),           // 1/2  
    new Fraction(1, 3),           // 1/3  
    new Fraction(2, 3),           // 2/3  
};  
Arrays.sort(data);                // 1/3,  
1/2, 2/3
```

sortiert *Fraction*-Array mit *compareTo()*

11. Objektorientierung

- 11.1 Methoden
- 11.2 Konstruktoren
- 11.3 static
- 11.4 Beispiele für Klassen
- 11.5 Vererbung
- 11.6 Dynamische Bindung
- 11.7 Klasse Object
- 11.8 final
- 11.9 Abstrakte Klassen
- 11.10 Interfaces

11.11 Anonyme Klassen

Anonyme Klassen

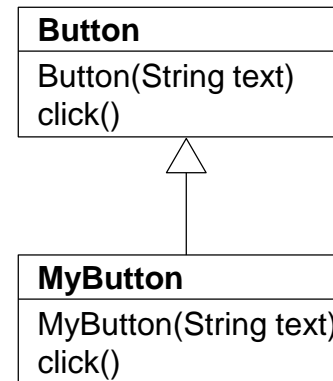


Beispiel mit konkreter Unterklasse

```
class Button {  
    Button(String text) {...}  
    void click() {}  
}
```

```
class MyButton extends Button {  
    MyButton(String text) { super(text); }  
    void click() {  
        Out.println("clicked");  
    }  
}
```

```
void buildGUI() {  
    ...  
    MyButton b = new MyButton("click me");  
    ... add b to GUI  
    ... when user clicks button  
    ... => click() from MyButton called  
}
```



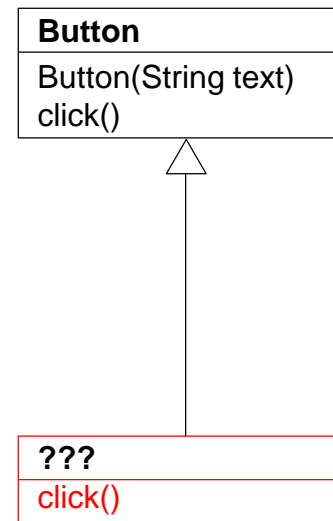
Anonyme Klassen (Fortsetzung)



Beispiel mit anonymer Unterklasse

```
class Button {  
    Button(String text) {...}  
    void click() {}  
}
```

```
void buildGUI() {  
    final int val = ...;  
    ...  
    MyButton b = new Button("click me") {  
        void click() {  
            Out.println("click, val = " + val);  
        }  
    };  
    ... add b to GUI ...  
}
```



Erzeugt anonyme Unterklasse von *Button* und überschreibt die Methode *click()*

Konstruktor der Oberklasse *Button* kann (mit Parametern) aufgerufen werden

click() kann auf lokale Variable *val* zugreifen. Diese muss aber *final* sein.

Vermeidet Deklaration einer expliziten Klasse *MyButton*
Für Fälle, in denen eine Klasse nur an *einer* Stelle benötigt wird

Anonyme Klassen (Fortsetzung)



Funktioniert auch mit Interfaces

```
interface Comparator {  
    int compare(Object a, Object b);  
}
```

```
class Arrays {  
    static void sort(Object[] data) {...}  
    static void sort(Object[] data, Comparator c) {...}  
    ...  
}
```

↑
Annahme: implementiert *Comparable* nicht!

```
interface Comparable {  
    int compareTo(Object other);  
}
```

```
c.compare(a, b)
```

< 0: wenn a < b
0: wenn a == b
> 0: wenn a > b

Anwendungsbeispiel: Sortieren von Personen nach Alter

```
class Person {  
    String name;  
    int age;  
}
```

implementiert *Comparable* nicht
=> Personen sind nicht automatisch vergleichbar

```
Person[] persons = ...;  
Arrays.sort(persons, new Comparator() {  
    int compare(Object a, Object b) {  
        Person p1 = (Person)a, p2 = (Person)b;  
        return p1.age - p2.age;  
    }  
});
```

übergibt *Comparator*-Objekt
zum Vergleichen von Personen