11. Objektorientierung

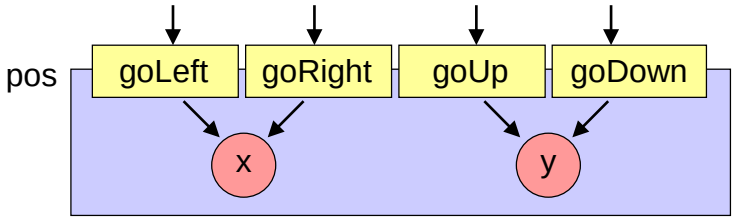
Klasse = Daten + Methoden

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
```

ruft *goRight*-Methode von *pos* auf

```
pos.goDown();          // pos.x == 1, pos.y == 1
pos.goDown();          // pos.x == 1, pos.y == 2
...
```

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*

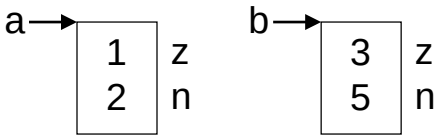
```
class Position {
    private int x;
    private int y;

    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)

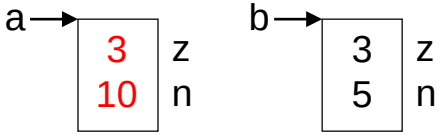
Beispiel: Bruchzahlenklasse

```
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;
    }
}
```

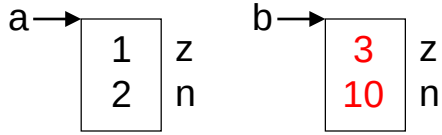
Fraction a = new Fraction(); a.z = 1; a.n = 2;
Fraction b = new Fraction(); b.z = 3; b.n = 5;



a.mult(b);



b.mult(a);



Es wird immer der Zustand des Empfängers verändert!

Grafische Notation für Klassen

UML-Notation (Unified Modeling Language)

Fraction
int z int n
void mult (Fraction f) void add (Fraction f)

Klassenname

Felder

Methoden

Vereinfachte Form

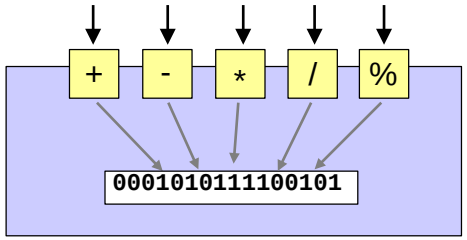
Fraction
z n
mult(f) add(f)

falls weniger Details gewünscht oder nötig

Klassen sind Abstrakte Datentypen

Konkreter Datentyp

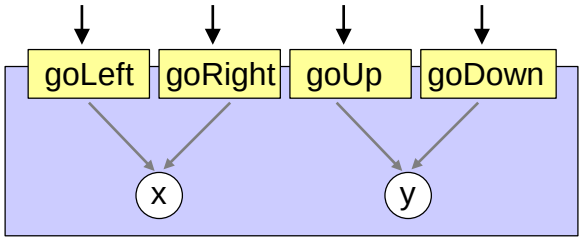
int



in die Sprache eingebaut

Abstrakter Datentyp (ADT)

Position



selbst implementiert

Programmiersprache kann mit ADTs (Klassen) beliebig erweitert werden.

Immer mehr Sprachkonzepte wandern heute in die Klassenbibliothek

Beispiel *String*

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

11. Objektorientierung

Konstruktoren

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) {...}
}
```

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

Andere Möglichkeit

```
Fraction () {
   this(0, 1);
}
```

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

Zweck: Erzeugung und Initialisierung sind zu einer einzigen Operation zusammengefasst

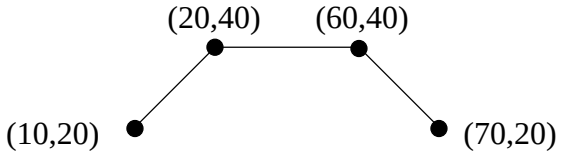
Feld-Zuweisungen im Konstruktor überschreiben Feld-Initialisierungen bei der Deklaration

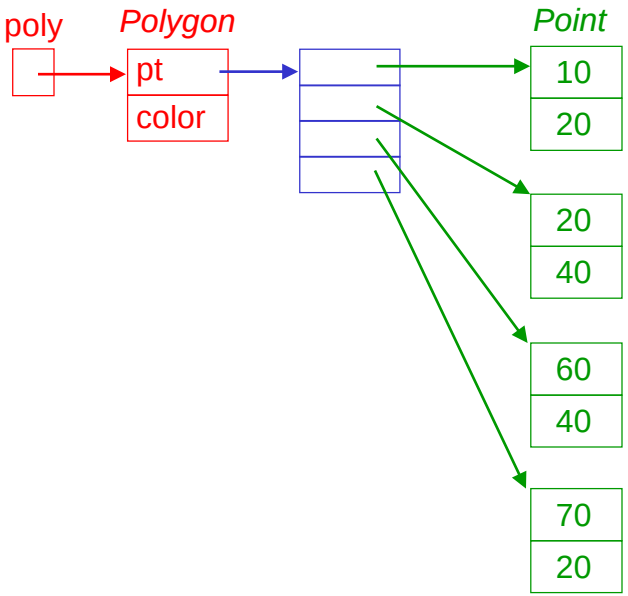
Bsp: Polygon-Aufbau mit Konstruktoren

```
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

static

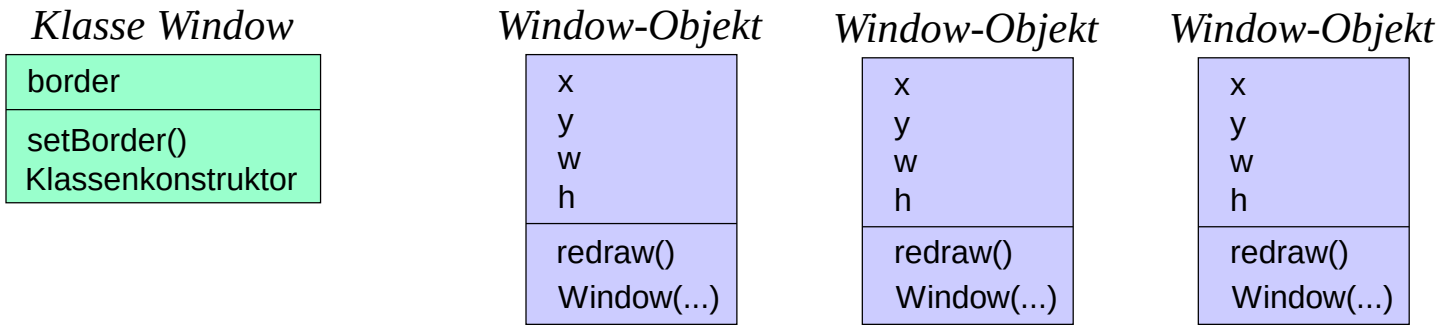
Felder, Methoden und Konstruktoren können *static* oder *nicht static* sein

```
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)



- Objektmethoden haben Zugriff auf Klassenfelder (*redraw* kann auf *border* zugreifen)
- Klassenmethoden haben keinen direkten Zugriff auf Objektfelder
  (*setBorder* kann nicht direkt auf *x* zugreifen, wohl aber z.B. auf *win.x*)

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 erfolgen über den Klassennamen

- Window.border = ...; Window.setBorder(3);
- Methoden der Klasse *Window* können auf eigene Elemente direkt zugreifen (border = ...; setBorder(3);)

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

- Window w = new Window(0, 0, 200, 400);
  w.x = ...; w.redraw();
- Objektmethoden 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

```
void foo() {
   this.x = 0;  this.redraw();
   Window.border = 1;
   Window.setBorder(1);
}
```

```
...
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

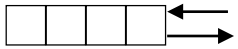
Beispiel: Stack und Queue

Stack (Stapel, Kellerspeicher)

push(x);	fügt *x* hinten an den Stack an
x = pop();	entfernt und liefert hinterstes Stackelement

```
push(3);   [3]
push(4);   [3][4]
x = pop();  [3]      // x == 4
y = pop();           // y == 3
```

LIFO-Datenstruktur
(last in first out)

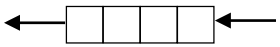


Queue (Puffer, Schlange)

put(x);	fügt *x* hinten an die Queue an
x = get();	entfernt und liefert vorderstes Queueelement

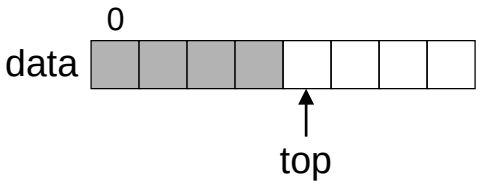
```
put(3);    [3]
put(4);    [3][4]
x = get();  [4]      // x == 3
y = get();           // y == 4
```

FIFO-Datenstruktur
(first in first out)

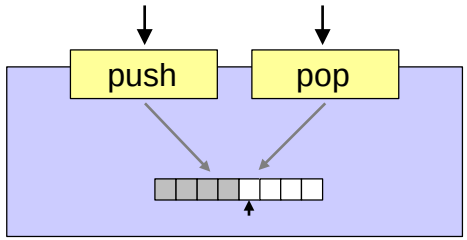


Klasse Stack

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

    Stack (int size) {
        data = new int[size]; top = 0;
    }

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

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



Baustein



Schnittstelle

Zustand

Benutzung

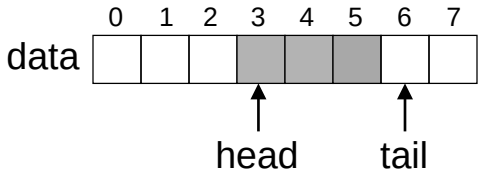
```
Stack s = new Stack(10);
s.push(3);
s.push(5);
int x = s.pop() - s.pop();  // x == 2 (d.h. 5 - 3)
```

Funktionen können auch wie Prozeduren aufgerufen werden

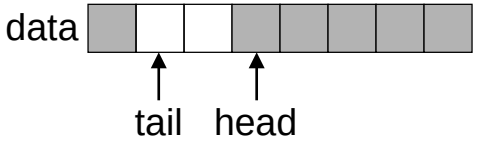
```
s.push(3);
s.pop();  // Rückgabewert wird verworfen
```

Klasse Queue

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

   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;
   }
}
```

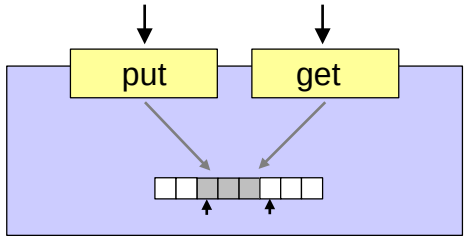


n == 3



n == 6

Baustein



Schnittstelle

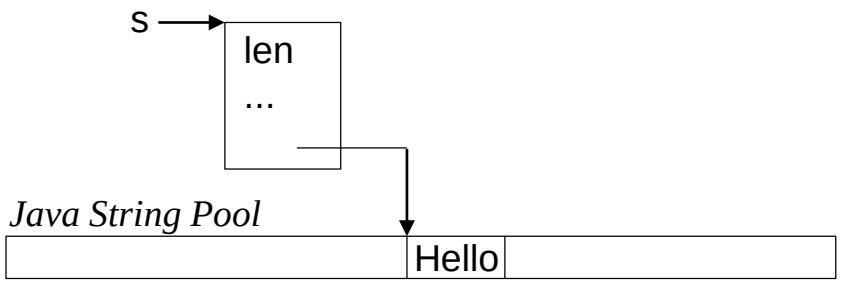
Zustand

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        indexOf(String s) {...}
   int        lastIndexOf(String s) {...}
   ...
}
```



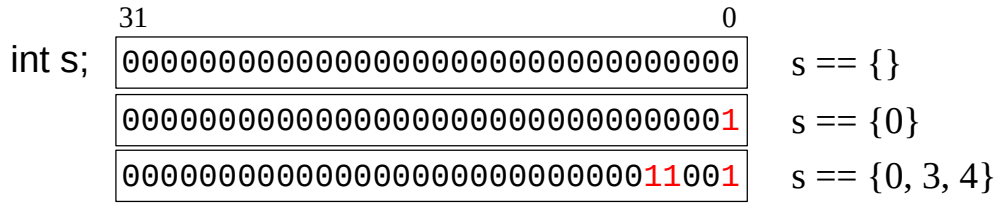
String ist teilweise auch Teil der Sprache Java

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

Ähnlich ist auch *StringBuilder* eine Klasse

Beispiel: Klasse für Zahlenmengen

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



Bit *i* gesetzt $\Leftrightarrow$ $i \in s$

```
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;
   }
}
```

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

00000001	1
00001000	1 << 3
00010011	data (Beispiel)
00011011	data | (1 << 3)

$i \in s$?

00000001	1
00001000	1 << 3
00011011	data (Beispiel)
00001000	data & (1 << 3)

$s \leftarrow \{\}$

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

x | y	...	bitweise Oder-Verknüpfung
x & y	...	bitweise Und-Verknüpfung
x ^ y	...	bitweise Xor-Verknüpfung

Logische Operationen auf *boolean*

a || b	...	logische Oder-Verknüpfung
a && b	...	logische Und-Verknüpfung

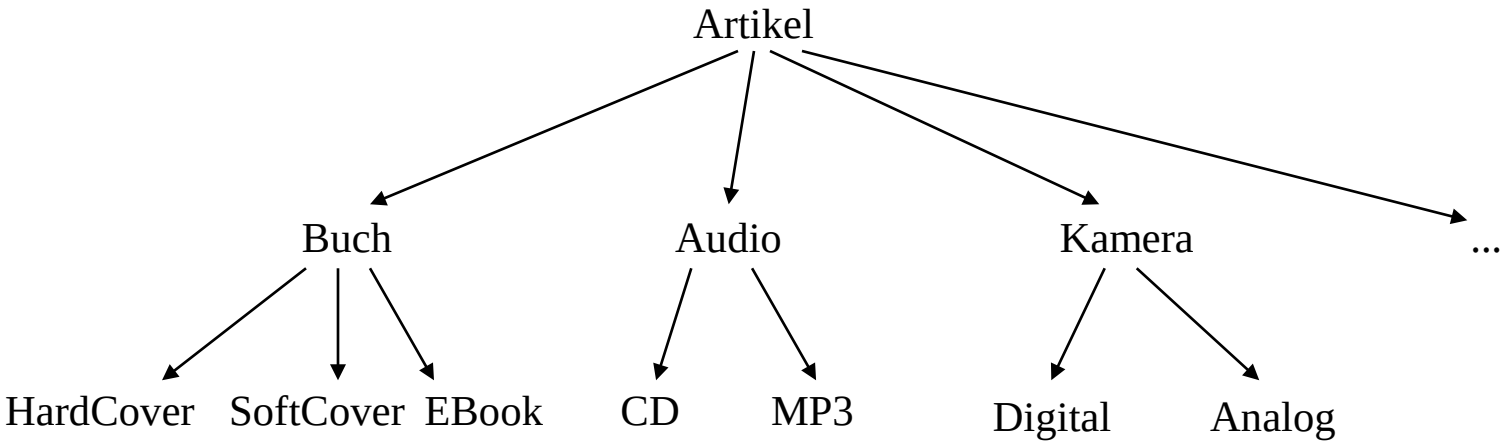
Benutzung

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

11. Objektorientierung

Klassifikation

Dinge der realen Welt lassen sich oft klassifizieren
z.B. Artikel eines Web-Shops



Man beachte

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

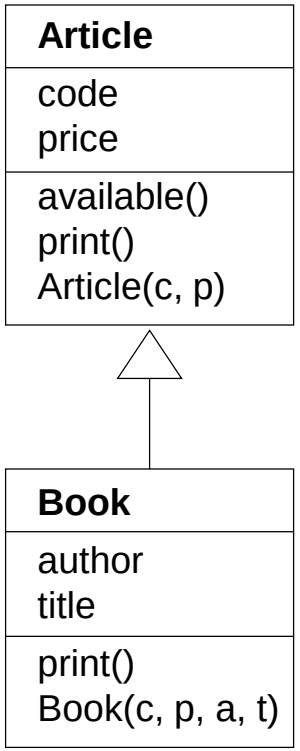
Vererbung

Dynamische Bindung

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*

Konstruktoren werden 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;
    }
}
```

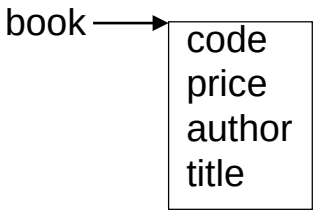
```
class Book extends Article {
    @Override
    void print() {
        super.print();
        Out.print(" " + author + ": " + title);
    }
    Book(int c, int p, String a, String t) {
        super(c, p);
        author = a; title = t;
    }
}
```

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 ≠ Überladen

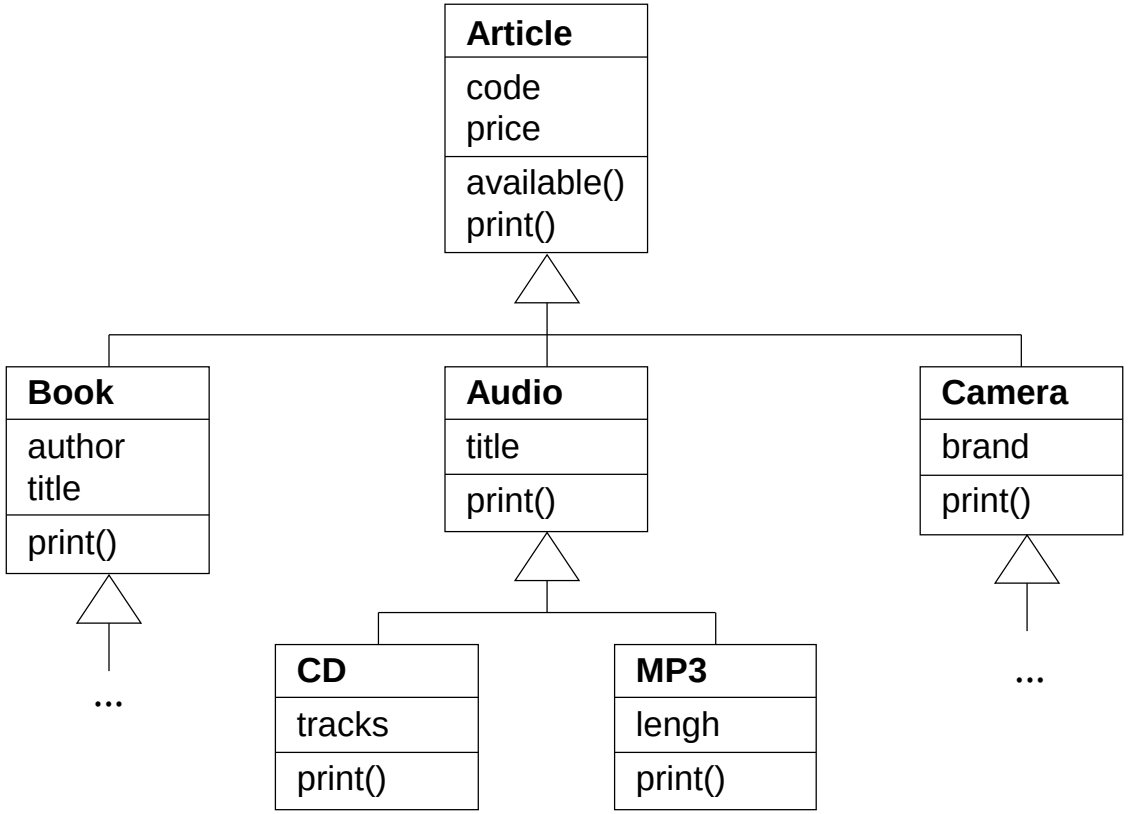

```
class Article {
    ...
    void print() {
        Out.print(code + " " + price);
    }

    ...
}
```

```
class Book extends Article {
    ...
    @Override
    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

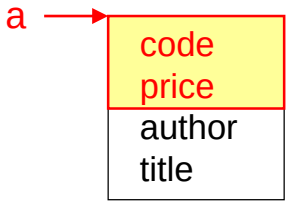
Jedes Buch ist ein Artikel
Aber: nicht jeder Artikel ist ein Buch

Kompatibilität zwischen Klassen

Unterklassen sind Spezialisierungen ihrer Oberklassen

Book-Objekte können _Article_-Variablen zugewiesen werden

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



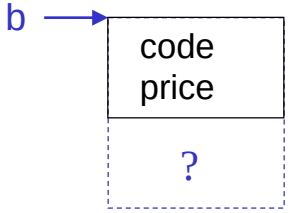
nur *Article*-Felder sind über *a* zugreifbar

```
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);
```

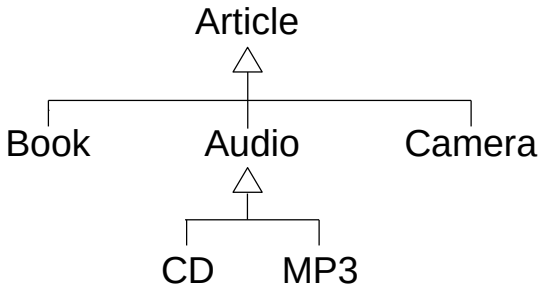


über *b* wären Felder zugreifbar,
die es nicht gibt

Jedes *Book* ist ein *Article*
aber nicht jeder *Article* ist ein *Book*

Beispiele für Zuweisungen

	statischer Typ	*dynamischer Typ*
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



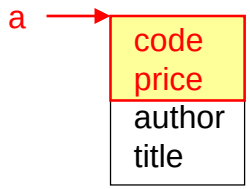
Welche der folgenden Zuweisungen sind korrekt?
Was ist der statische/dynamische Typ der linken Seite?

		stat. Typ	*dyn. Typ*
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*) ... *Article*
DT(*a*) ... *Book*



Laufzeit-Typtest

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

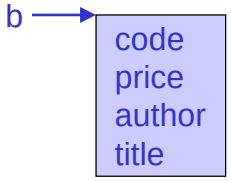
ist DT(*a*) zumindest *Book*?
(d.h. *Book* oder Unterklasse: *Hardcover*, *Softcover*, *EBook*)

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

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

Typumwandlung (Type Cast)

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



Geprüfte Typumwandlung

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

Nach Typumwandlung ist Zugriff auf *Book*-Elemente möglich

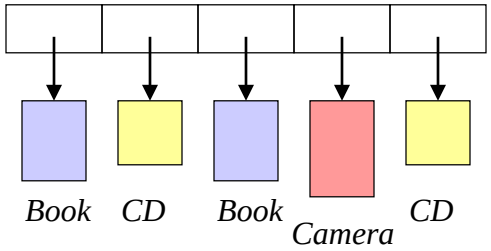
```
((Book)a).title = ...;
```

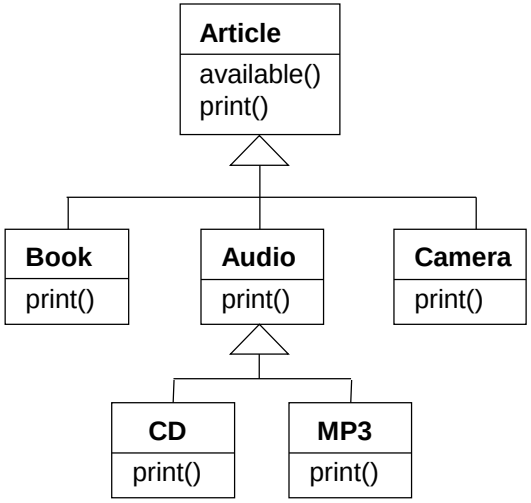
11. Objektorientierung

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

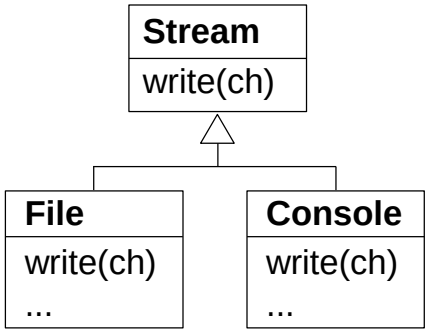
Dyn. Bindung fördert Wiederverwendung

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!

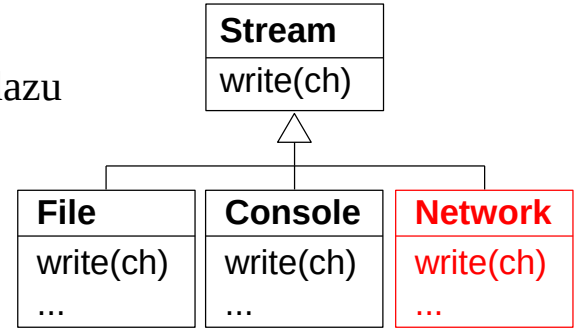


Benutzung

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

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

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



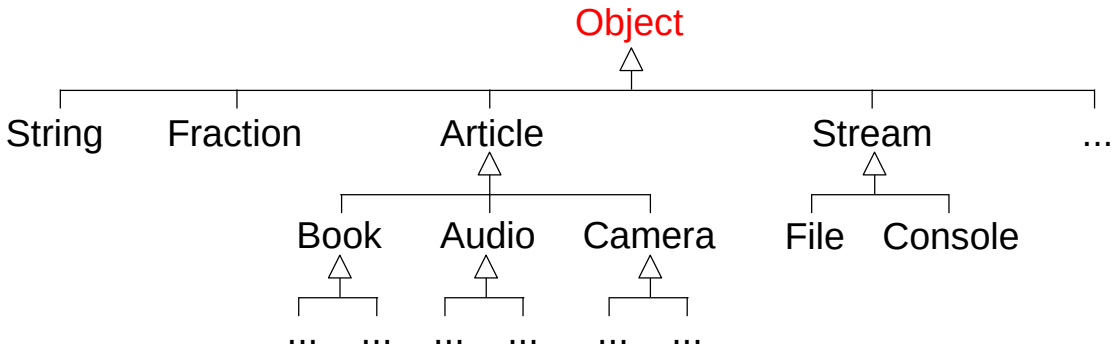
writeText kann mit *NetWork* arbeiten,
ohne geändert werden zu müssen

11. Objektorientierung

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

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

} sollten überschrieben werden

} werden unverändert vererbt

obj.toString()	soll den Wert von *obj* als String zurückgeben
obj.equals(obj2)	soll true liefern, wenn die Werte von *obj* und *obj2* gleich sind (*obj == obj2* bedeutet Zeigervergleich, nicht Wertvergleich)
obj.getClass()	liefert Information über den dynamischen Typ von *obj*
obj.clone()	liefert eine Kopie von *obj*

Überschreiben von equals und toString

```
class Fraction {
    int z;
    int n;

    Fraction(int z, int n) {...}

    void mult(Fraction f) {...}
    void add(Fraction f) {...}

    @Override
    public String toString() {
        return z + "/" + n;
    }

    @Override
    public 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)) ...
```

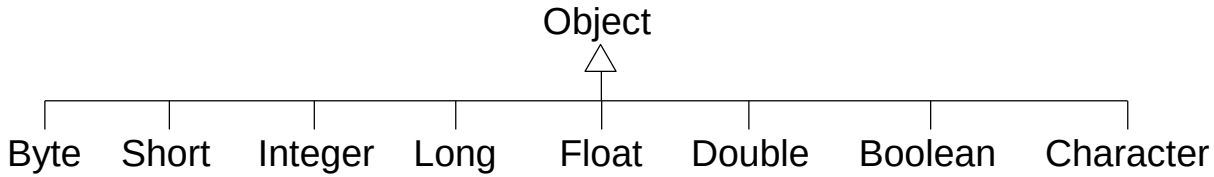
gibt "1/2" aus
gibt "1/2" aus (Compiler fügt .toString() ein)

liefert *false* (Zeigervergleich)
liefert *true* (Wertvergleich)

Wrapper-Klassen für Werttypen

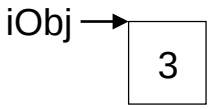
Problem: *int, long, double, ...* sind nicht direkt mit *Object* kompatibel

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

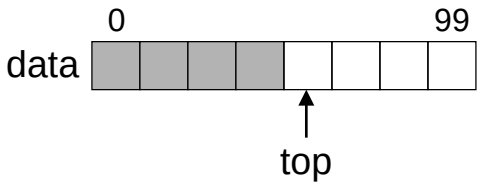
```
Integer iObj = 3;      // wickelt int-Wert 3 in ein Integer-Objekt ein
int i = (int) iObj;    // wickelt int-Wert aus iObj wieder aus
```

```
Object obj = 3;
int i = (int) obj;
```

Anwendungsbeispiel für Object

Allgemeiner Stack für Objekte beliebigen Typs

```
class GeneralStack {
    private Object[] data = new Object[100];
    private int top = 0;

    public void push (Object obj) {
        data[top] = obj; top++;
    }

    public Object pop () {
        top--;
        return data[top];
    }
}
```



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();
```

Bessere Lösung: generischer Datentyp (siehe Softwareentwicklung 2)

11. Objektorientierung

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 {
    ...
    final boolean login (String password) {
        ...
    }
    void print() {
        ...
    }
}
```

```
class SpecialAccount extends UserAccount {
    long someOtherMethod() {
        ...
    }
    boolean login (String password) {
        ...
    }
    void print() {
        super.print();
        ...
    }
}
```

Unterklassen erlaubt

neue Methoden hinzufügbar

final-Methoden dürfen nicht über-schrieben werden

non-*final*-Methoden dürfen über-schrieben 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;
      ...
      userName = ...;     // compile error
      offset = 10;        // ok
      offset = ...;       // compile error
      a = ...;            // compile error
      a[0] = ...;         // ok
   }
}
```

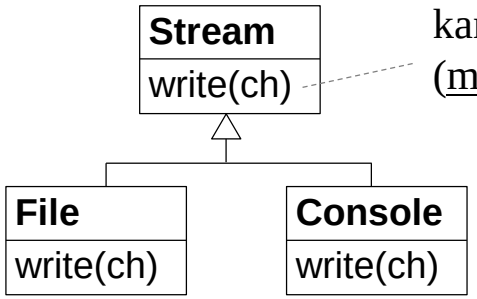
Konstante
- belegt keinen Speicherplatz

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
- *final*-Variablen dürfen nur ein einziges Mal einen Wert bekommen.

11. Objektorientierung

Abstrakte Klassen



kann noch nicht sinnvoll implementiert werden
(muss überschrieben werden)

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

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

Schlüssewort *abstract*, kein Code!

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

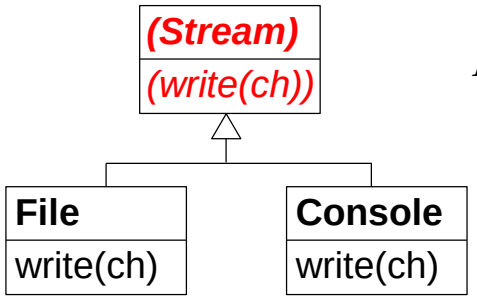
Es dürfen keine Objekte abstrakter Klassen erzeugt werden

```
Stream stream = new Stream();
```

Abstr. Klassen dürfen aber als Datentyp verwendet werden

```
Stream stream = new Console();
```

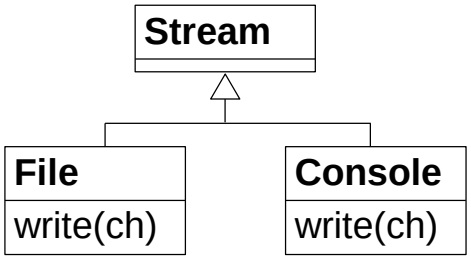
Abstrakte Klasse gibt Schnittstelle für zukünftige Unterklassen vor



Abstrakte Klasse/Methode: kursiv und in Klammern

Abstrakte Klassen (Fortsetzung)

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



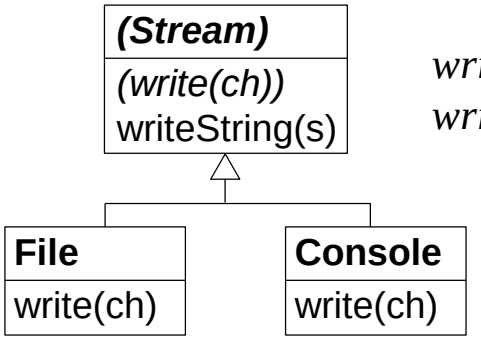
Weil sonst dynamische Bindung nicht funktionieren würde

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

Stream hat kein *write*
Compiler würde einen Fehler melden

Alle Methoden, die man dynamisch gebunden aufrufen will, müssen in *Stream* deklariert werden (wenn auch 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()) {
         this.write(ch);
      }
   }
}
```

Dynamische Bindung

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

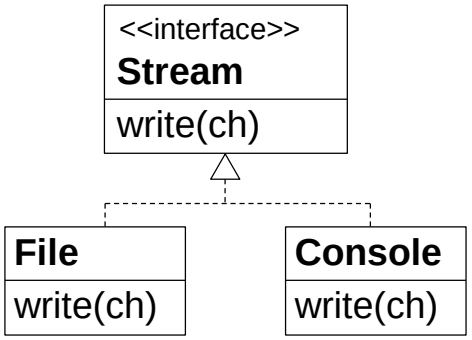
Benutzung

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

ruft *writeString* von *Stream* auf,
das mittels dyn. Bindung *write* von *Console* aufruft
(DT(*this*) == *Console*)

11. Objektorientierung

Interfaces



Interface entspricht vollständig abstrakter Klasse

- nur Methodensignaturen
- keine Datenfelder

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

Methoden sind automatisch *abstract*

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

implements statt *extends*

muss alle Methoden von *Stream* überschreiben (implementieren)

Klassen können beliebig viele Interfaces implementieren, aber nur von 1 Klasse erben

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

} a - b

Anwendungsbeispiel Bruchzahlen vergleichbar machen

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

this *f2*

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

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

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()*