

# 6.

# Objektorientierung

---

In Teil I der Vorlesung wurde (bzgl. C++) einiges bereits kurz angesprochen

Buch Mark Weiss „Data Structures & Problem Solving Using Java“ siehe:

- 105-109 (Objektorientierung)
- 145-204 (Vererbung)

# Lernziele Kapitel 6 Objektorientierung

- Prinzipien von Objektorientierung bei Modellierung und Softwareentwurf kennen
- Konzepthierarchie und Klassenhierarchie verstehen
- Mit Vererbung und Polymorphie umgehen können
- Generische Algorithmen entwickeln können

## Thema / Inhalt

Wir kommen in diesem Kapitel zum Eingemachten, der **Objektorientierung**. Dabei handelt es sich im Grunde genommen um eine Weltsicht, fast eine Art Ideologie: Wie soll die Welt (oder genauer gesagt, der relevante Ausschnitt davon) modelliert werden, damit ein grösseres Softwaresystem (das typischerweise ein Problem der realen Welt lösen soll) realitätsgerecht entworfen und gut strukturiert werden kann? Wenn beim Problembereich also beispielsweise Autos, Kunden, Strassenkreuzungen, Parkhäuser, Mietwagenfirmen etc. vorkommen, dann würde eine realitätstreue Modellierung von vielem Sonstigen in der Welt abstrahieren, aber eben Autos, Kunden,... explizit hervorheben und geeignet repräsentieren. Da es nicht nur ein einziges Auto und einen einzigen Kunden gibt, würde man viele verschiedene Entitäten der **Klassen** „Autos“ und „Kunden“ haben wollen, wobei diese individuellen **Objekte** vielleicht dynamisch entstehen und vergehen können. Vor allem aber wird es zu Wechselwirkungen

## Thema / Inhalt (2)

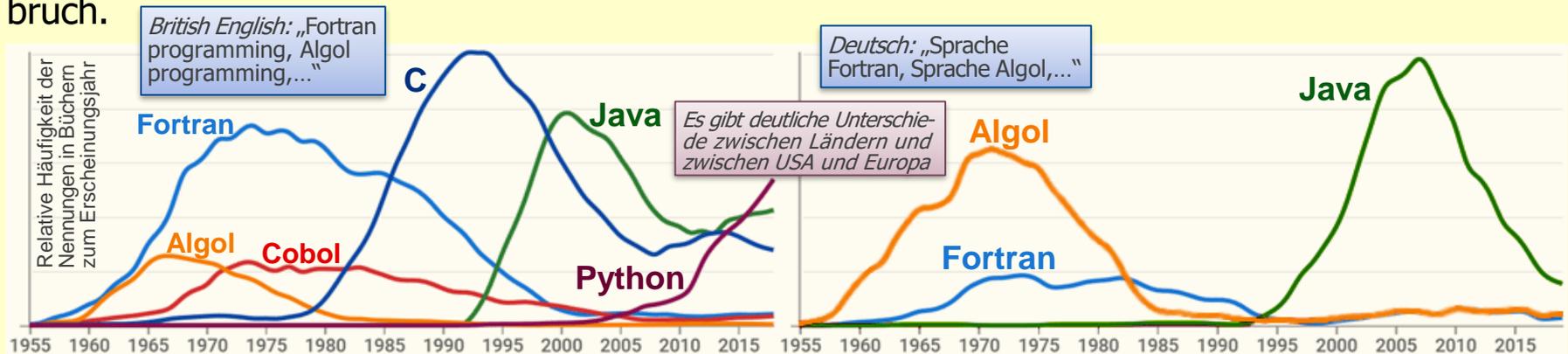
zwischen diversen Objekten kommen: Kunden mieten ein Auto; ein Auto kommt an einer Strassenkreuzung an; wenn zu viele Autos in ein Parkhaus fahren, wird es voll und neu ankommende Autos werden abgewiesen etc. Und vielleicht ist es zweckmässig, Autos in zwei Unterklassen zu separieren: Personenwagen und Lastwagen – zwar sollten beide Typen in vielen Fällen gleich behandelt werden, und dann sollte quasi unsichtbar sein, ob jetzt gerade ein Personenwagen oder Lastwagen ansteht, in anderen Fällen sollte aber differenziert werden, weil alle Personenwagen Eigenschaften haben, die nicht allen Autos zukommen.

Die Objektorientierung wurde für derartige Problembereiche konzipiert. Wenn man Computer im eher mathematischen Sinne als bessere Rechenmaschine einsetzt, und man beispielsweise Matrizen invertieren oder Nullstellen approximieren will, dann nützen die Konzepte der Objektorientierung wenig (und wären vielleicht sogar unangebracht), da die kompakte und stringente Sprache der Mathematik bereits recht gut mit den gängigen Paradigmen und Konzepten von Programmiersprachen abgedeckt wird. Anders ist es, wenn nicht-mathematische Objekte zum expliziten Gegenstand des Problems gehören.

Tatsächlich entstand die Objektorientierung recht früh, Mitte der 1960er-Jahre, als erstmalig komplexere, nicht-numerische Probleme mittels Computern bearbeitet werden sollten. Konkret ging es um Simulationsanwendungen, wo man sowieso explizit eine Modellierung des Gegenstandsbereichs vornehmen muss. Praktisch alle relevanten Konzepte der Objektorientierung, die man heute beispielsweise bei Java findet, wurden seinerzeit für die Sprache **Simula**, eine Weiterentwicklung der Sprache **Algol**, konzipiert: Klassen, Objekte, typisierte Referenzen auf Objekte, Klassenhierarchien, Vererbung, Datenkapselung, virtuelle Klassen etc. Im historischen Teil des Kapitels stellen wir kurz die beiden norwegischen Erfinder von Simula, Ole-Johan Dahl und Kristen Nygaard, vor.

# Thema / Inhalt (3)

Obwohl die Konzepte der Objektorientierung die akademische Welt überzeugten, sollte es noch Jahrzehnte dauern, bis dies Eingang in den Mainstream fand; zunächst dominierten in der Industrie klassische, nicht-objektorientierte Programmiersprachen wie Fortran und Cobol. Als mit **C++** Mitte der 1980er-Jahre Prinzipien der Objektorientierung an die Trägersprache C angeflanscht wurden, erfuhr die Objektorientierung langsam einen Bedeutungszuwachs in der Praxis, auch wenn die Realisierung bei C++ unvollständig blieb und aufgrund des Erbes von C mit vielen pragmatischen Kompromissen verbunden war. **Java** legte dann in dieser Hinsicht Mitte der 1990er-Jahre einen Neustart hin und machte die Objektorientierung in konsequenterer Weise zu einem Hauptpfeiler der Sprache – und verhalf ihr so zum endgültigen Durchbruch.

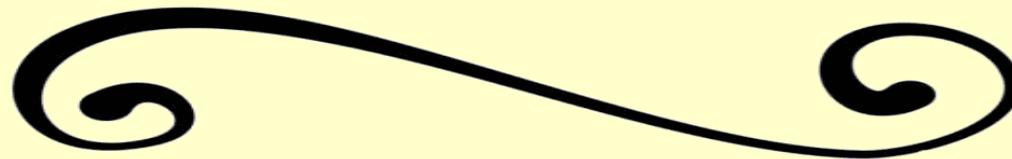


Zu Beginn des Kapitels besprechen wir zunächst kurz die „Philosophie“ der Objektorientierung, geben dann einige Beispiele für Konzept- und Klassenhierarchie an und kommen schliesslich zu abgeleiteten Klassen, Vererbung und Polymorphie in Verbindung mit abstrakten Klassen. Erst nach dieser konzeptionellen Diskussion gehen wir auf Java selbst ein und besprechen, wie die objektorientierten Konzepte bei Java konkret ausgestaltet sind und wie diese verwendet werden können.

# Thema / Inhalt (4)

Für ein konkretes Beispiel, bei dem die diversen Aspekte zusammenspielen, bilden wir zunächst eine **Klassenhierarchie geometrischer Objekte** und entwickeln sodann einen **typgenerischen Sortieralgorithmus**. Nachdem wir das Sortierverfahren an einfachen Verbunddatentypen mit je drei relevante Datenelementen „Name, Fach, Semesterzahl“ für eine Menge von Studi-Objekten getestet haben, bringen wir die beiden Dinge zusammen: Interaktiv können nun geometrische Objekte eines gewünschten Typs mit einer wählbaren Grösse erzeugt werden, anschliessend wird typspezifisch der jeweilige Flächenwert ermittelt und die geometrischen Objekte werden sortiert nach diesem Flächenwert ausgegeben.

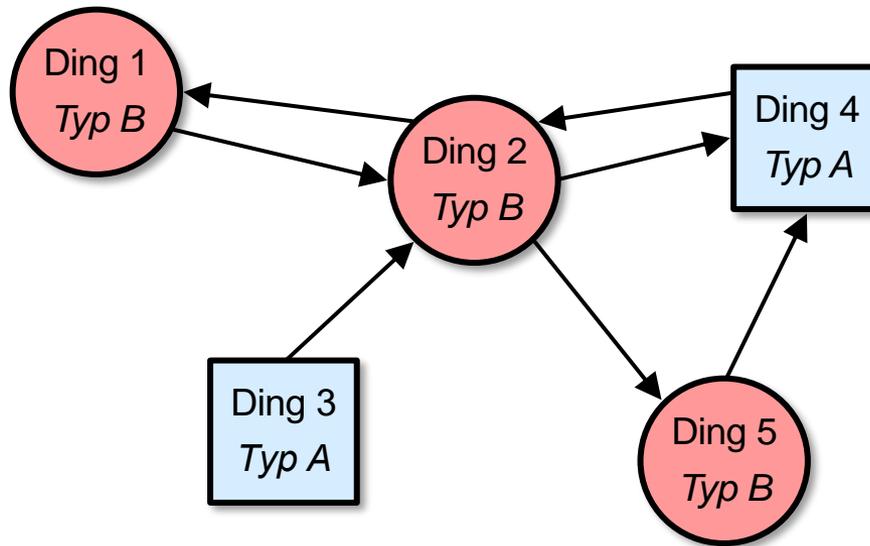
Auch wenn das Beispiel hoffentlich instruktiv und erhellend ist, so sollte man bedenken, dass Beispiele in der Vorlesung notgedrungen klein bleiben; ihren eigentlichen Wert und ihre volle Überzeugungskraft können objektorientierte Konzepte, Sprachen und Architekturen erst bei grossen Projekten mit vielen Entwicklern ausspielen.



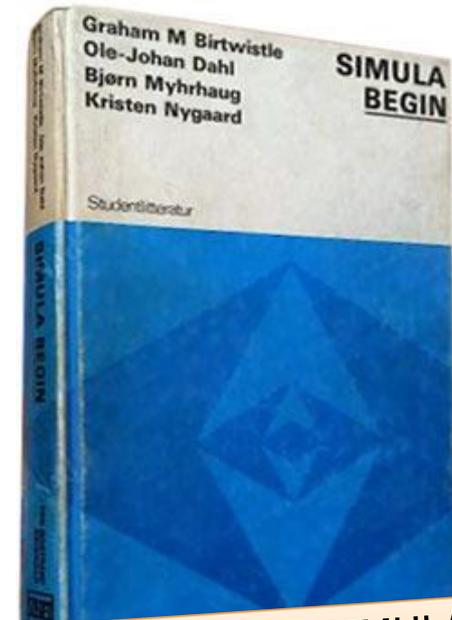
# Objektorientiertes Programmieren

Einige Konzepte der Objektorientierung (Polymorphie, dyn. Bindung, Vererbung, abstrakte Klassen) wurden bereits kurz am Ende von Informatik I angesprochen

- **Weltsicht:** Die Welt besteht aus verschiedenen **interagierenden „Dingen“**, die sich typbezogen **klassifizieren** lassen

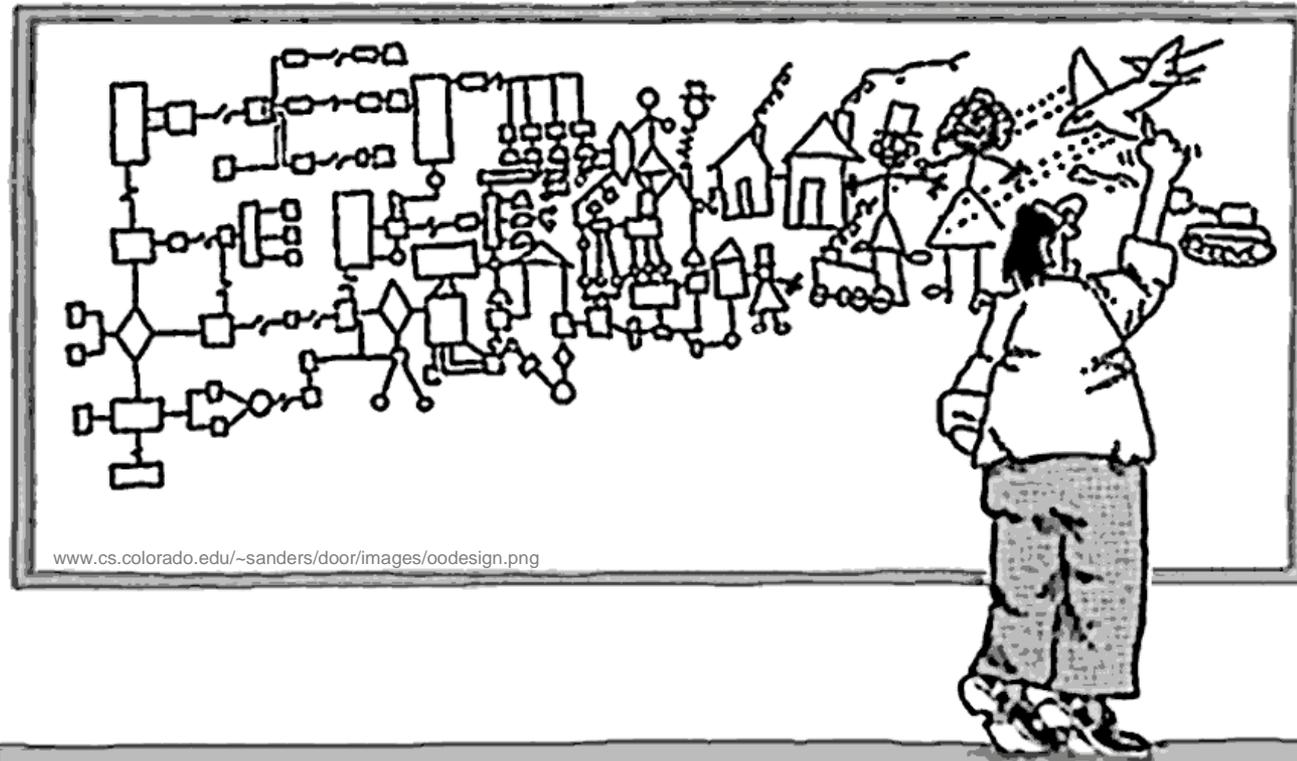


- **Ziel:** Betrachteten Weltausschnitt strukturkonform mit **interagierenden Objekten** abbilden und **modellieren**



**Simulationssprache SIMULA** war die erste objektorientierte Programmiersprache (1967)

# Objektorientiertes Programmieren



- **Ziel:** Betrachteten Weltausschnitt strukturkonform mit interagierenden Objekten abbilden und modellieren

# Objekte

- Sind autonome, gekapselte Einheiten eines bestimmten **Typs**
- Haben einen eigenen **Zustand**
  - = Lokale Variablen
- Besitzen ein **Verhalten**
  - Wird wirksam, wenn sie über ihre Methoden aktiviert werden
- Bieten anderen Objekten **Dienstleistungen** an
  - Ausführen von lokalen Algorithmen
  - Änderungen des eigenen Zustandes
  - Zurückliefern von Berechnungsergebnissen
  - Allgemein: „Reaktion“ auf Aufruf einer Methode

# Objektorientierter Softwareentwurf

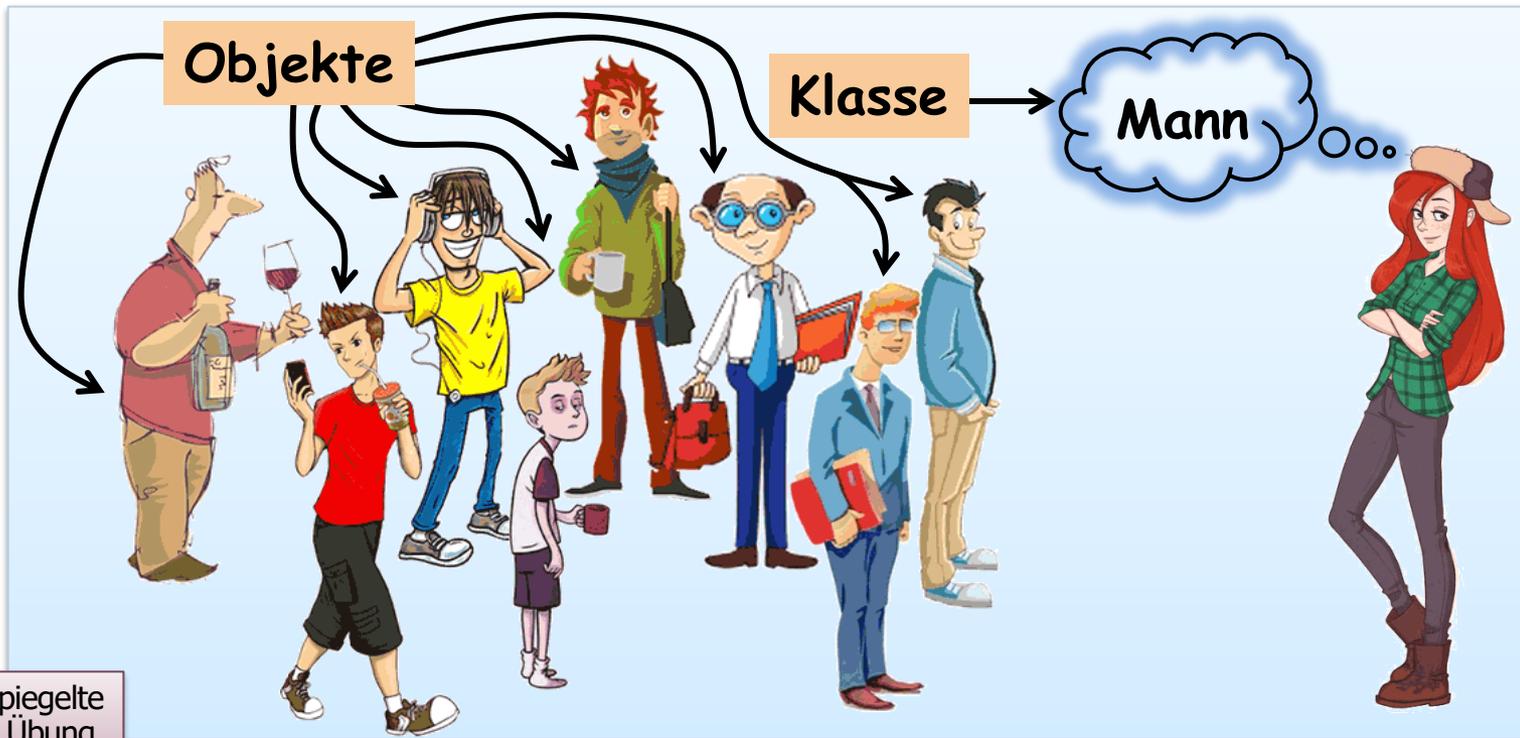
*Think big!*

- Grundlegend für den Softwareentwurf ist die **Modellierung**, bei der ein Ausschnitt der Wirklichkeit auf abstrakte Konzepte abgebildet wird. Dazu wird **analysiert**, was für die Aufgabenstellung relevant ist, um daraus ein **implementierungsfähiges Modell** zu erstellen.
  - Ziel der **Analyse** (zu Beginn des Softwareentwurfs) ist es herauszufinden, was alles zu dem zu modellierenden Ausschnitt der Realität gehören soll, um das Problem zu lösen und den Anforderungen an eine Lösung zu genügen.
  - Ziel der **Implementierung** (am Ende des Softwareentwurfs) ist es, den modellierten Wirklichkeitsausschnitt in konkrete Software abzubilden.
- Bereits in frühen Phasen des Softwareentwurfs wird dabei das **Objektparadigma** verwendet.
  - Relevante **Begriffe** der Wirklichkeit herausarbeiten und operationalisieren.
  - Geeignete **Objekte / Typen der Anwendungsdomäne** identifizieren sowie die Beziehungen der Objekte untereinander analysieren.
  - Den betrachteten Realitätsausschnitt dann möglichst **strukturkonform mittels interagierender Objekte modellieren**.

# Objektorientierter Softwareentwurf *Think big!*

Strukturierung der Softwareanwendung als eine Menge kooperierender Objekte

- 1) Entwurf der Objekttypen, dabei ähnliche Objekte zu **Klassen** zusammenfassen



Gendergespiegelte  
Version als Übung

# Objektorientierter Softwareentwurf

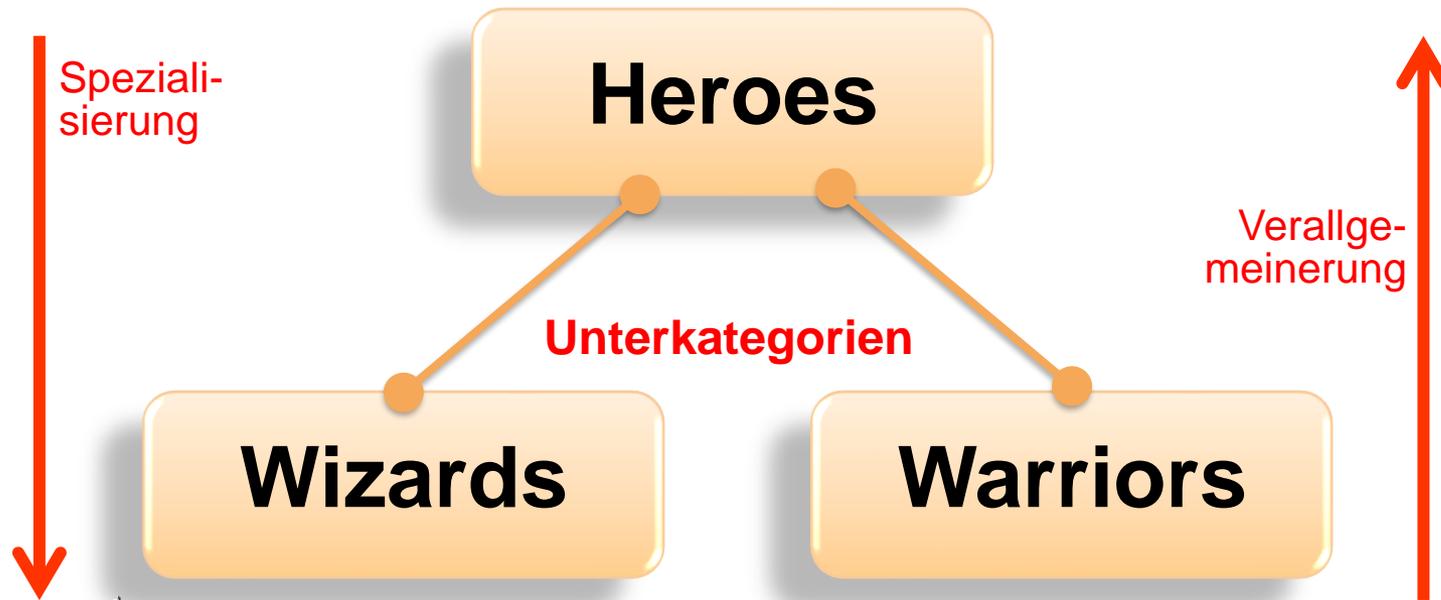
*Think big!*

Strukturierung der Softwareanwendung als eine Menge kooperierender Objekte

- 
- 1) Entwurf der Objekttypen, dabei ähnliche Objekte zu **Klassen** zusammenfassen
  - 2) Herausfaktorisierung gemeinsamer Aspekte verschiedener Klassen  $\Rightarrow$  **Konzepthierarchie** festlegen
  - 3) Festlegung einzelner **Dienstleistungen** als Methoden
  - 4) Entwurf der **Objektbeziehungen**
  - 5) **Feinplanung** der einzelnen Methoden, Festlegung der Klassenattribute etc.
  - 6) **Implementierung** der Methoden (d.h. klassisches Programmieren im Kleinen)

„Die objektorientierte Analyse beschäftigt sich mit dem Extrahieren von Objektklassen und Objektbeziehungen aus einer informellen Problembeschreibung. Das Ziel ist es, ein Modell der realen Welt zu gewinnen, das dann im anschließenden Entwurf zum Grundstock eines Modells der Software wird. Dadurch ähneln sich die Struktur der Realität und die Struktur der Software; es steigt die Wahrscheinlichkeit, dass die Software die Realität widerspiegelt und durchschaubar bleibt.“  
-- Wolfgang Küchlin, Andreas Weber

# Konzepthierarchie (für Fantasy-Fans)



Instanzen



# Konzepthierarchie

Begriffserweiterung  
durch neuen Oberbegriff

Credo der  
Objektorientierung:  
**Alles** ist ein Objekt!

Objekt

Musikgerät

Was auch immer

„Nicht-automatische Musikgeräte“

Musikinstrument

Grammophon

Player

Speziali-  
sierung

Trommel

Saiteninstrument

Blasinstrument

MP3

CD

Laute

Klavier

Streichinstrument

Holzbläser

Blechbläser

Verallge-  
meinerung

Cello

Violine

Blockflöte

Trompete

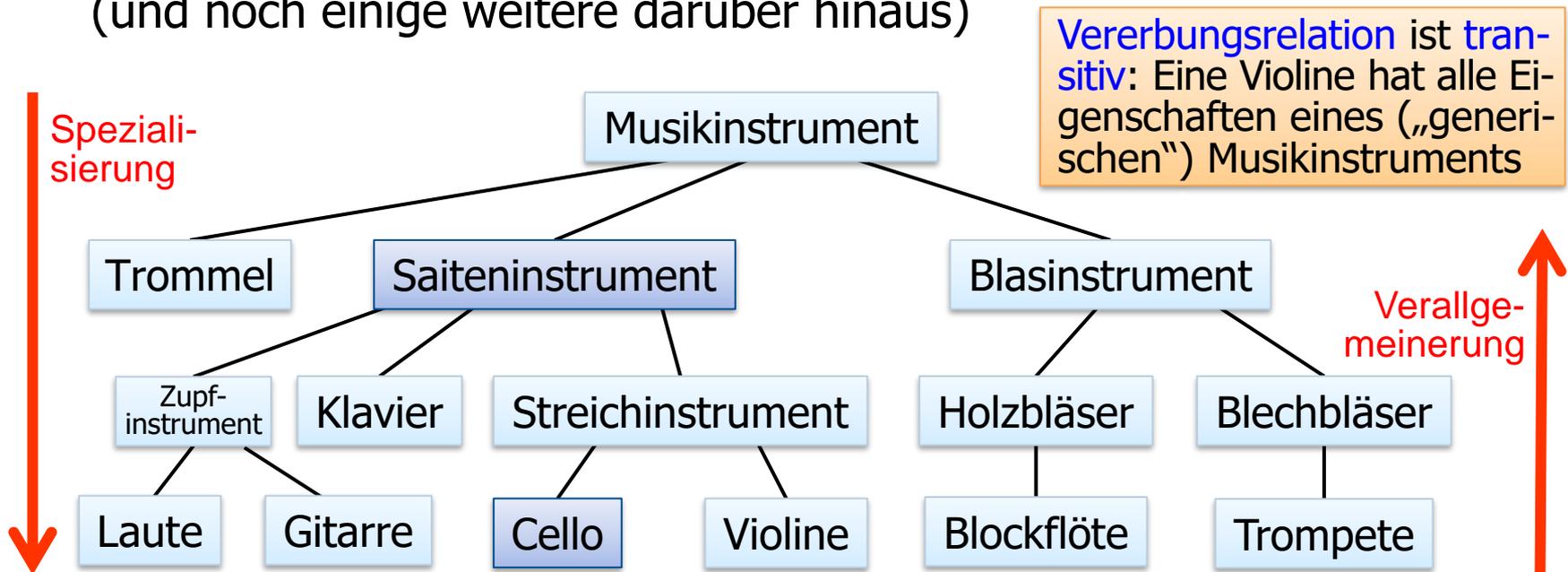
- Spezialisierung, Verallgemeinerung
  - Violinen sind (spezielle) Musikinstrumente
  - Blasinstrumente umfassen u.a. Blockflöten



Codex Manesse (fol. 423v)

# Vererbung („Inheritance“) und Polymorphie

- Alle **Merkmale, Eigenschaften...** des umfassenderen Begriffs werden an den Unterbegriff vererbt
  - Ein **Cello** hat **alle Eigenschaften** eines allgemeinen **Saiteninstrument** (und noch einige weitere darüber hinaus)



- Ein Objekt (als „Begriffsinstanz“) ist oft polymorph
  - Eine **Violine** kann je nach Zweckmässigkeit als **Streichinstrument** oder einfach als **Musikinstrument** betrachtet werden

# Haben Verallgemeinerungen eine reale Existenz?

## Ein philosophischer Exkurs

Ob es ein Allgemeines wirklich gibt oder ob Allgemeinbegriffe nur menschliche bzw. verstandesmäßige Konstruktionen sind, ist ein **zentrales Thema der Philosophie**. Es ist als „**Universalienproblem**“ bekannt, da Allgemeinbegriffe wie z.B. „Mensch“ und „Menschheit“, aber auch die typischen mathematischen Abstraktionen (oder eben doch „Entitäten“?) wie „Zahl“ oder „Relation“ als Universalien bezeichnet werden. Universalien umfassen dabei Oberbegriffe konkreter Individuen, die etwas gemeinsam haben (wie etwa „Lebewesen“), aber auch Merkmale, die mehreren Entitäten zukommen (wie z.B. „männlich“ oder „schwarz“ – substantivisch dann etwa „Mannheit“ oder „Männlichkeit“ bzw. „Schwärze“ oder „Schwarzheit“ etc.).

Seit der Antike wird eine grundlegende Diskussion darüber geführt („Nominalismusstreit“), ob Universalien eine ontologische Existenz beigemessen werden kann (**Realismus**) oder ob es sich dabei um rein gedankliche Begriffsbildungen handelt (**Nominalismus**); nach letzterer Auffassung kommt Realität nur den Einzeldingen zu. Im Mittelalter war die Frage sogar von theologischer Relevanz, etwa als es darum ging, den Monotheismus zu rechtfertigen und gleichzeitig die Dreieinigkeit als „reale“ Einheit anzuerkennen, statt in dieser nur eine (leichter wegzudiskutierende) Gedankenkonstruktion zu sehen – der sogen. „Tritheismus“ galt schliesslich als Häresie.

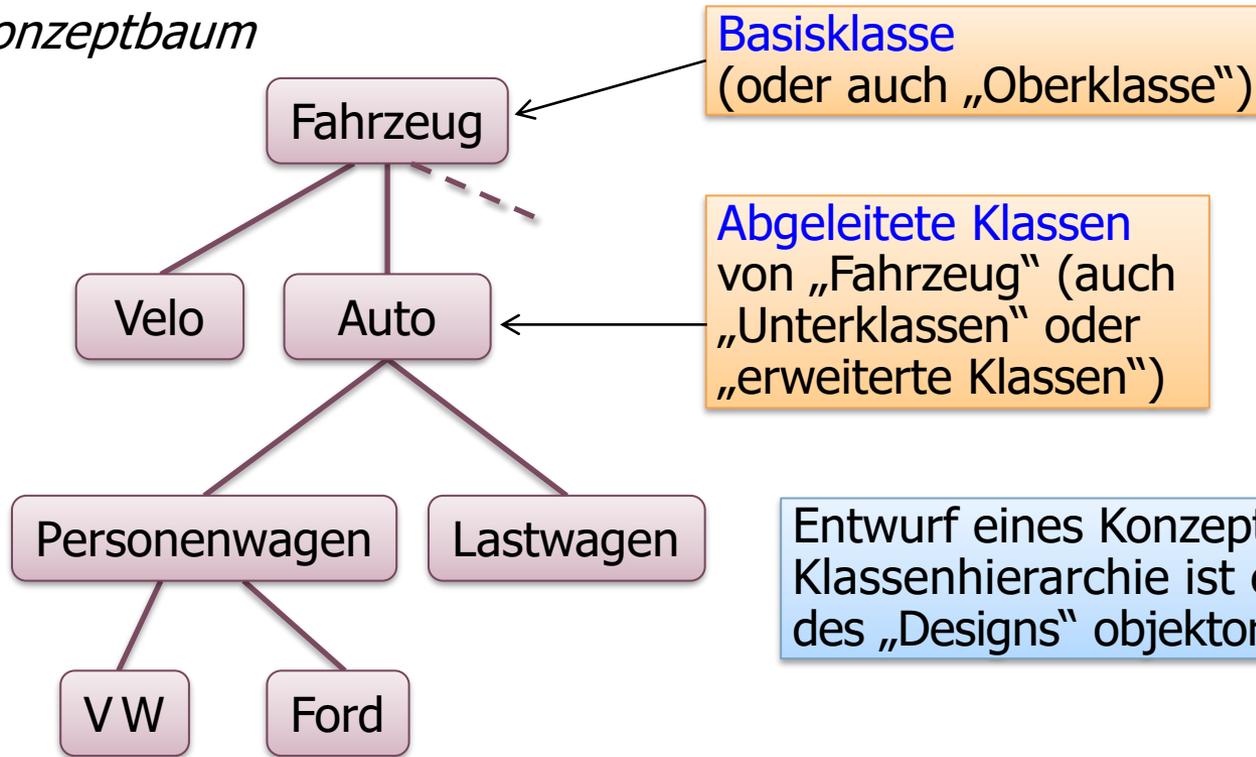
← Wird nun nicht mehr wie früher von powerpoint ungefragt korrigiert zu „Dreieinigkeit“!

In der **Mathematik** war und bleibt umstritten, welche Objekte „konkret“ existieren und welche nur als (mehr oder weniger) nützliche Hirngespinnste aufgefasst werden sollten. Ein Ausdruck davon ist auch die strittige Frage, ob mathematische Objekte und Gesetze entdeckt oder erfunden werden. Von Leopold Kronecker (1823 - 1891) ist der Spruch „die ganzen Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk“ überliefert – er liess nur mathematische Gegenstände gelten, deren Existenz durch explizite Konstruktionen gesichert werden konnte, und wurde so zu einem Vorläufer des mathematischen Konstruktivismus.

Die **Objektorientierung** geht mit der Klassenbildung pragmatisch um – in der Modellwelt „existiert“, was nützlich ist. Allerdings dienen dabei normale Klassen als Schablonen für Objekte, während abstrakte Klassen eher als Konzepte im Sinne von allgemeineren Oberbegriffen fungieren.

# Klassenhierarchie als Konzeptbaum

Konzeptbaum



Entwurf eines Konzeptbaums und einer Klassenhierarchie ist ein wichtiger Teil des „Designs“ objektorientierter Systeme

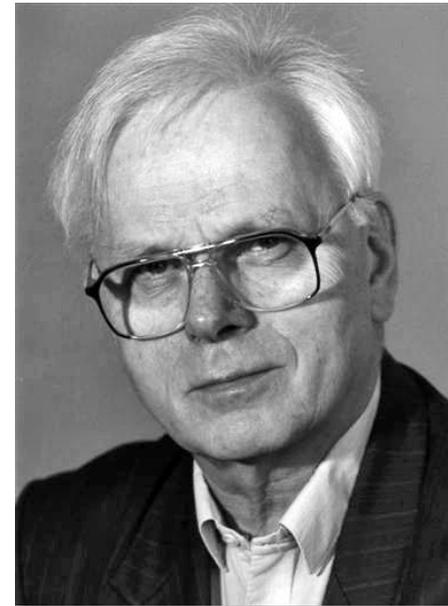
„is-a“-Relation

- Ein VW *ist ein* Personenwagen *ist ein* Auto *ist ein* Fahrzeug
- Eine Violine *ist ein* Streichinstrument *ist ein* Musikinstrument
- Ein Auto hat Räder ⇒ ein VW hat Räder

**Umkehrung gilt i.A. nicht!**

# Kristen Nygaard erinnert sich: Klassen und Vererbung

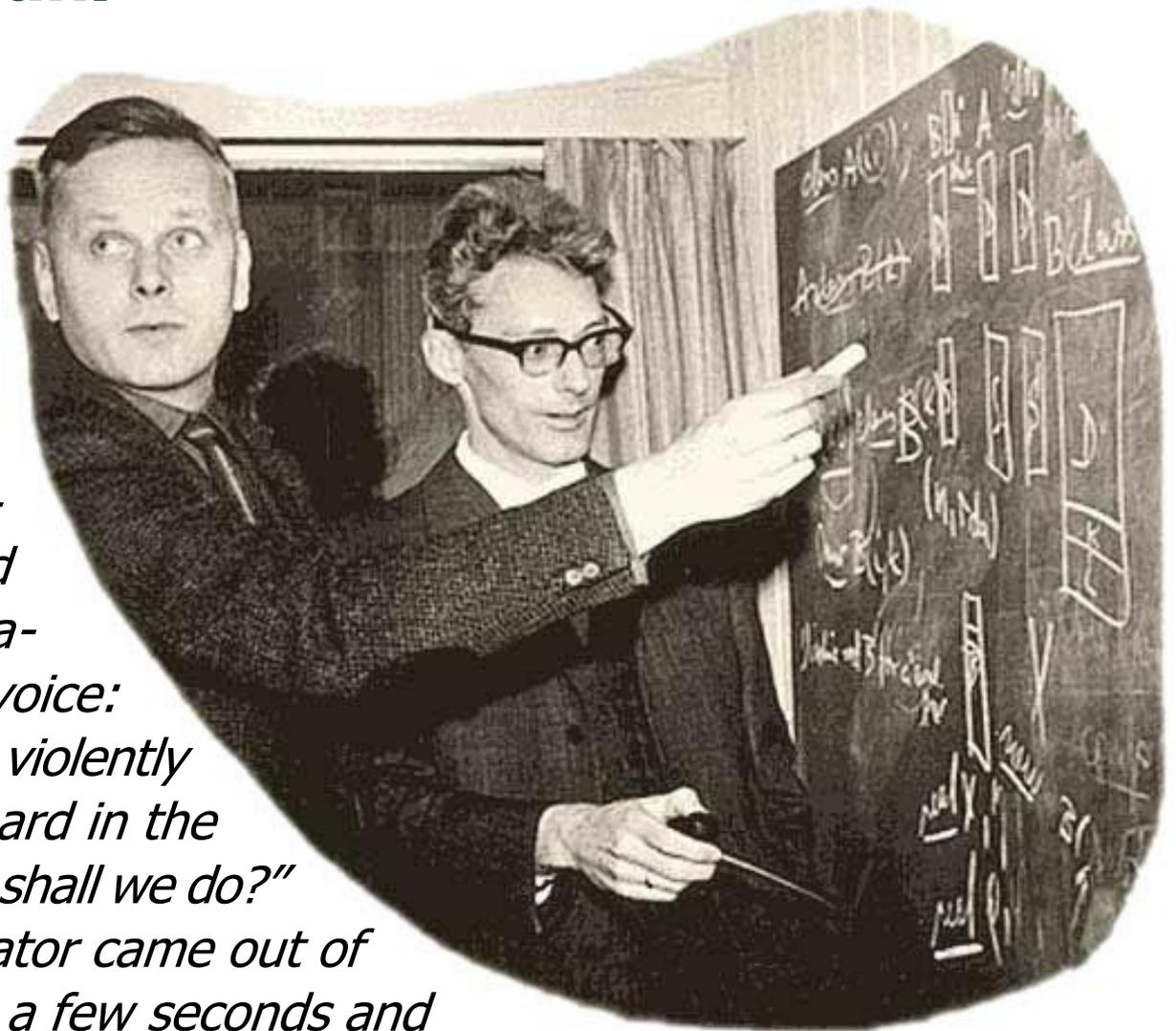
Kristen Nygaard (1926–2002) entwickelte gemeinsam mit Ole-Johan Dahl bereits in den 1960er-Jahren die erste objektorientierte Programmiersprache („Simula“, aufbauend auf Algol 60, mit Klassenkonzept, Vererbung sowie Schlüsselwörtern wie *class*, *new*, *this* etc., analog zu Java heute). Er erhielt dafür zusammen mit Dahl im Jahr 2001 den Turing Award.



I remember very, very clearly the exact moment, around two o'clock in the night at the desk in the bedroom at Nesodden, **January 1967**, when the **concept of "inheritance" (or classes and subclasses) had been created**. I realized immediately that this was the solution to a very important problem Ole-Johan Dahl and I had been struggling with for months and weeks. I also realized that the solution introduced for the first time in a programming language **a strong and flexible version of the notions of generalization and specialization**, with all the power embedded in those concepts. And sure enough, inheritance has become a key concept in object oriented programming, and thus in programming in general.

# Nygaard und Dahl

*In the spring of 1967 a new employee at the Norwegian Computing Center, where Dahl and Nygaard were working together, rushed into her colleague's office and told the switchboard operator in a very shocked voice: "Two men are fighting violently in front of the blackboard in the upstairs corridor. What shall we do?" The switchboard operator came out of her office, listened for a few seconds and then said: "Relax, it's only **Dahl and Nygaard discussing Simula.**"*



# Nygaard und Dahl: Objektorientierung

Jeg er forsker i informasjonsteknologi (IT) og har sammen med Ole-Johan Dahl oppfunnet OBJEKT-ORIENTERT PROGRAMMERING.  
-- Kristen Nygaard auf seiner Homepage

The class related concepts in Simula were clearly **ahead of their time**; it took some 20 years until they gained understanding and popularity. Languages such as Smalltalk, Beta, C++, Eiffel, Java, and C#, have directly adopted Simula's fundamental concepts about objects and classes.

How could Ole-Johan Dahl and Kristen, at such an early stage, design a language with all the mechanisms that today form the "object-oriented" paradigm for system development? [...] It was probably very fortunate that they first designed a language for simulation (Simula 1), and later generalized the concepts in a general purpose language (Simula 67).

Ole-Johan has expressed it this way: "A reason for this may be that in developing a complicated simulation model it is useful to decompose it in terms of 'objects', and to have an explicit mapping from external objects to program constructs. A natural way of developing real systems is not much different." Kristen emphasized that an essential motivation behind Simula was system description, and the need for **a language to model real world concepts**.

Auszug aus: O. Owe, S. Krogdahl, and T. Lyche: A Biography of Ole-Johan Dahl. In: O. Owe et al. (Eds.): From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl, Springer LNCS 2635, pp. 1–7, 2004.

# Nygaard und Dahl: Objektorientierung

„Es ist sicherlich kein Zufall, dass die objektorientierte Programmierung in Norwegen erfunden wurde – einem Land, das sich aus einer Gesellschaft freier Bauern entwickelte und weder Feudalherrschaft noch Leibeigenschaft kannte. Die **politische Symbolik selbstständiger Software-Objekte**, die über eigene Mittel verfügen, um mit anderen Objekten in Beziehung zu treten und zu interagieren, liegt auf der Hand. Sicherlich war sie auch Nygaard bewusst, der zeitlebens ein politischer Mensch war.“

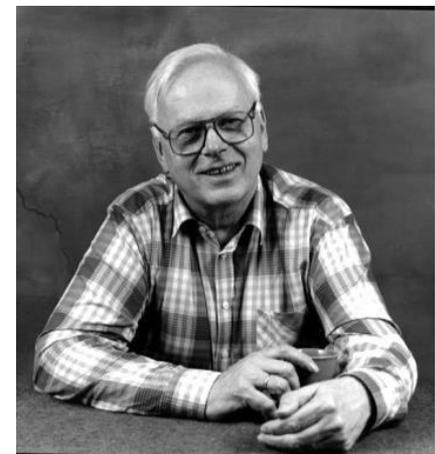
-- Berthold Daum: *Happy Birthday! Simula 67 – Pionier der Objektorientierung*. Java-Spektrum 02/2007, 31 – 34



Kristen Nygaard (li.), zusammen mit Ole-Johan Dahl (re.), mit dem Malteserkreuz des Sankt-Olav-Ordens (Aug. 2000)

# Kristen Nygaard

Einige kurze Auszüge aus: *Drude Berntse, Knut Elgsaa, Håvard Hegna: The Many Dimensions of Kristen Nygaard, Creator of Object-Oriented Programming and the Scandinavian School of System Development. A. Tatnall (Ed.): HC 2010, 38–49, 2010.*



Nygaard showed exceptional and diverse talents from an early age. Initially his main interests were in the natural sciences and in mathematics. While he was in secondary school, he followed university-level lectures and won a national mathematics award before finishing grammar school (age 16-19). His insistence on knowing everything about the topics that caught his interest was well-known among his friends and colleagues. During his years as a student, his broad music interest paid off by giving him extra financing from writing reviews of recordings of classical music in a leading Norwegian newspaper. Nygaard's many talents and interests were aided by an immense memory capacity.

Nygaard's studies at the University of Oslo led to bachelor degrees in astronomy and physics and to a master in mathematics in 1956 based on his thesis "Theoretical Aspects of Monte Carlo Methods". The thesis theme resulted from his work at the Norwegian Defence Research Establishment (NDRE) where he had been working full time since he started his military service in 1948. His first six years at NDRE were spent in the Mathematics Section, initially doing Numerical Analysis and Computer Programming. It was here that he first met Ole Johan Dahl.

In 1952 he was asked to join the Operations Research (OR) Group at NDRE, soon he became the head of the group. Nygaard was central in founding the Norwegian OR Society (NORS) in 1959. Between 1959 and 1963 Nygaard's professional life changed completely. He was asked by representatives of Norwegian industry to establish a group for civilian OR at the Norwegian Computing Center (NCC). In the autumn of 1963, Ole-Johan Dahl also joined Nygaard at NCC.

For Nygaard, compilers and program execution were important and necessary, but after a while object-orientation for him was not about programming, it was a tool for modelling and understanding. For a year in 1974 he was guest professor at the University of Aarhus in Denmark. In 1984 Nygaard became a full-time professor at the University of Oslo.

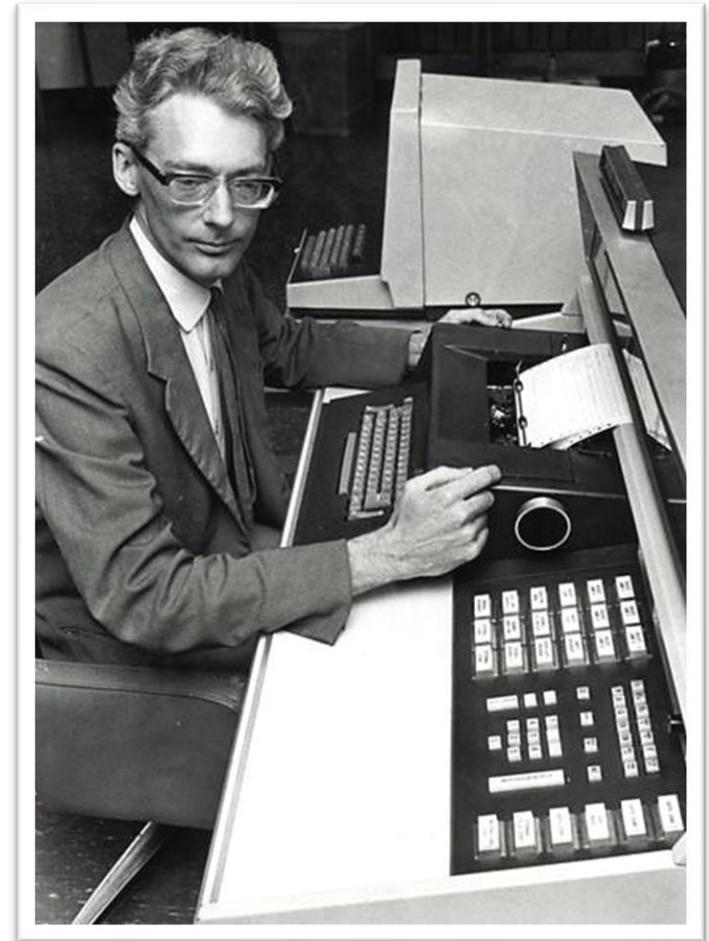
# Ole-Johan Dahl

Ole-Johan Dahl (1931–2002) wurde 1968 der erste Professor für Informatik in Norwegen, im Jahr 2000 erhielt er die Ehrendoktorwürde der ETH Zürich sowie zusammen mit Kristen Nygaard im Jahr 2001 den Turing Award.



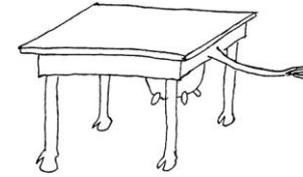
Bild: <https://ititan.uio.no/node/2485>

Kristen Nygaard (li) und Ole-Johan Dahl (re) 1974 während eines Workshops in einer Hütte auf Røros; dazwischen der dänische Informatik-Pionier Peter Naur.



Dahl in den 1960er-Jahren.

# Ursprünge des OO-Paradigmas



“Can't an object be both a cow and a table?” fragte Adriaan van Wijn-gaarden mit dieser Skizze bei einem Workshop im Oktober 1965.

Tony Hoare, der bereits mehrfach erwähnt wurde, spielte eine wichtige Rolle bei der Genese von zentralen Konzepten der Objektorientierung.

Hier zwei kurze Auszüge aus einem Artikel „[Record Handling](#)“ von ihm, veröffentlicht im ACM Algol Bulletin 21 (pp 39–69) aus dem Jahr 1965.

Es taucht hier zwar schon der Begriff „[object](#)“ auf, allerdings ist damit ein „object of the real world“ gemeint, repräsentiert wird es im Computer durch einen “record”, der dynamisch erzeugt werden kann. Auch der Begriff „[class](#)“ wird verwendet, hier allerdings eher noch als Kurzform von „record class“.

Schon bald wird man in der Fachgemeinschaft einfach von „object“ und „class“ sprechen – bis heute fundamentale Begriffe des objektorientierten Programmierparadigmas. Insbesondere griffen Dahl und Nygaard die Konzepte und Begriffe für ihre Sprache „[Simula 67](#)“ auf, erweiterten das Klassenkonzept aber noch, z.B. um ausführbaren Code.

## 2. Summary

### 2.1 Records and Record Classes

The proposal envisages the existence inside the computer during the execution of the program, of an arbitrary number of records, each of which represents some object which is of past, present or future interest to the programmer. The program keeps dynamic control of the number of records in existence, and can create new records or destroy existing ones in accordance with the requirements of the task in hand.

Each record in the computer must belong to one of a limited number of disjoint record classes; the programmer may declare as many record classes as he requires, and he associates with each class an identifier to name it. A record class name may be thought of as a common generic term like "cow", "table" "house", and the records which belong to these classes represent the individual cows, tables and houses with which a particular activation of a program is concerned. It is obvious that no object can be both a cow and a table, so it is quite natural to insist that every record shall belong to one and only one class.

### 4.2 Simulation Studies.

In the simulation of complex situations in the real world, it is necessary to construct in the computer analogues of the objects of the real world, so that procedures representing types of <sup>events?</sup> even may operate upon them in a realistic fashion. For example, tankers belonging to an oil company can be represented by records whose fields give the relevant operational characteristics of the ships; similarly, the ports of arrival, and departure can be represented by records of a different class. The ship records will probably contain references to the current port of destination, the port of departure, etc.

Ende der historischen Notiz

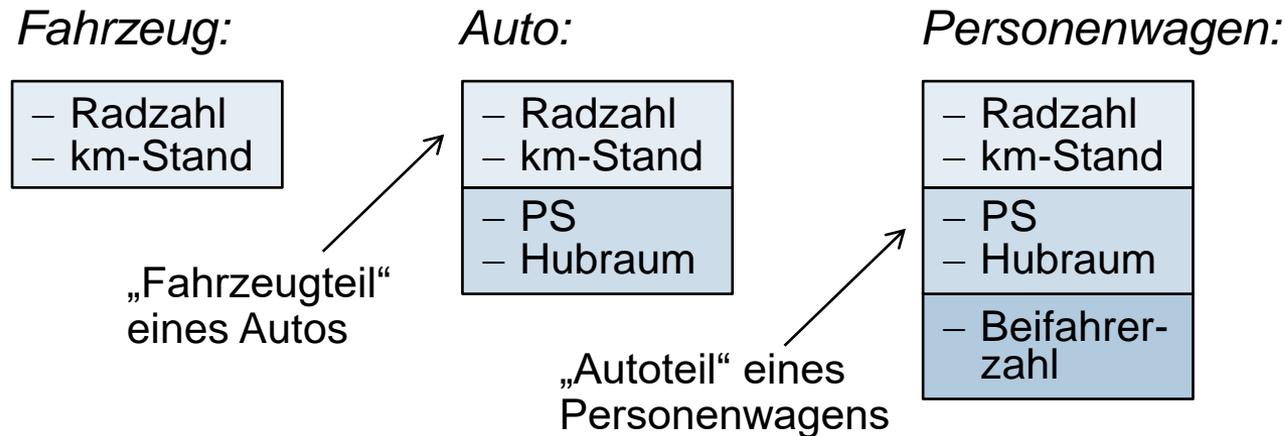
# Abgeleitete Klassen, Redefinition

- Eine abgeleitete Klasse besitzt automatisch alle Eigenschaften der zugehörigen Basisklassen
  - Konkret: besitzt alle **Attribute** und **Methoden** der Basisklassen
- Ausser: Es werden explizit einige davon **redefiniert**
  - Redefiniert: heissen noch genauso, verhalten sich aber etwas anders
  - Beachte: **Sichtbarkeit** (public/private etc.) darf nicht reduziert werden

**Java Compiler:** Cannot reduce the visibility of the inherited method.

- Eine abgeleitete Klasse wird in der Regel **zusätzliche** Attribute und Methoden definieren
  - → **Spezialisierung** durch **Erweiterung**:  
Ein Facharzt ist ein um Spezialkenntnisse „erweiterter“ Arzt

# Abgeleitete Klassen, Redefinition – Beispiel



- Eine Methode „**berechneSteuer**“ ist im Allgemeinen nicht für alle Fahrzeugtypen genau gleich definiert
  - Man würde z.B. in „Auto“ eine Standardmethode vorsehen (Benutzung von „Hubraum“), jedoch für **spezielle** Fahrzeuge (z.B. Elektroautos) diese Methode **anders** definieren

# PS, Hubraum, VW, BMW,...



Glücksspiel oder Strategiespiel? Zwar spielt Zufall (oder „Glück“?) eine wichtige Rolle, aber die jeweils „richtige“ Wahl der Kenngröße (und damit auch das Wissen über die technischen Daten der Autos auf den Spielkarten) ist entscheidend. Früher (also bevor es mobiles Internet gab, quasi in der Steinzeit) war dies ein beliebtes Spiel bei Schülern.



**Autoquartett:** Die gemischten Karten werden gleichmässig unter den Spielern verteilt. Jeder Spieler hält seine Karten so in der Hand, dass nur er das oberste Blatt sehen kann. Der Spieler, der zuletzt gewonnen hat, nennt eine beliebige Kenngröße seiner Karte, wie zum Beispiel die Anzahl der Zylinder, Motorleistung (PS), Höchstgeschwindigkeit oder Baujahr. Die Mitspieler nennen daraufhin die entsprechenden Daten auf ihrer obersten Karte, und der Spieler, dessen Karte den besten Wert hat, gewinnt die obersten Karten aller Mitspieler. Meist wird der höchste Wert als der beste angesehen, bei einigen Kenngrößen auch der niedrigste – so gewinnt etwa beim Baujahr typischerweise der älteste Wagentyp. Besitzen zwei oder mehr Spieler Karten mit demselben besten Wert, so spielen diese mit einem anderen gewählten Kennwert eine Entscheidungsrunde.

# Ein Beispiel in Java

```
class Fahrzeug{
    public int Radzahl;
    public int kmStand;
}

class Auto extends Fahrzeug {
    public int PS;
    public float Hubraum;
}

class PW extends Auto {
    public int Beifahrerzahl;
}

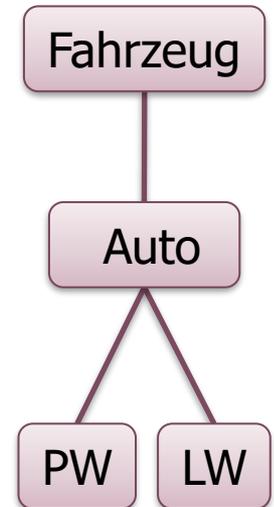
void print() {
    System.out.println("Radzahl: " + Radzahl +
        ", Beifahrerzahl: " +
        Beifahrerzahl);
}

class LW extends Auto {
    public float Zuladung;
}
```

Vorfahre weiss nichts von den Erben; Erbe muss sich seinen passenden Vorfahren selbst aussuchen!

Erweiterung der Klasse „Fahrzeug“: Alles, was in „Fahrzeug“ deklariert ist, gehört damit auch zur „Auto“ (sowohl Attribute als auch Methoden) – mit kleinen Einschränkungen (→ später)

Auf „weiter oben“ definierte Attribute kann ohne weiteres zugegriffen werden – diese sind Teil der abgeleiteten Klasse!



# Ein Beispiel in Java (2)

```
class Beispiel{
    public static void main(String args[]) {
        Fahrzeug f = new Fahrzeug();
        Auto a = new Auto();
        PW p = new PW();
        LW l = new LW();

        p.Beifahrerzahl = 5;
        p.PS = 70;
        p.Hubraum = 1794;
        p.Radzahl = 4;
        p.print();
    }
}
```

Hier werden Instanzen (also Objekte) der verschiedenen Hierarchiestufen erzeugt

Zugriff auf Variablen und Methoden des mit 'p' bezeichneten PW-Objektes

p.Zuladung geht natürlich nicht!

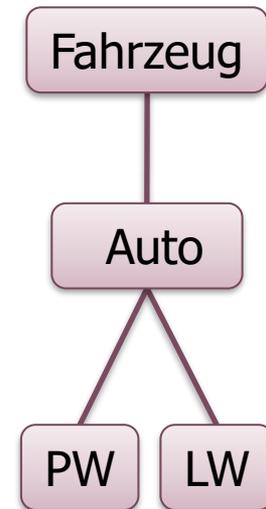
Idee: Gemeinsame Aspekte *herausfaktorisieren* und in eine übergeordnete Klasse einbringen

*Personenwagen:*

- |                 |
|-----------------|
| – Radzahl       |
| – km-Stand      |
| – PS            |
| – Hubraum       |
| – Beifahrerzahl |

# Zuweisungskompatibilität

- Objekte von abgeleiteten Klassen können an Variablen vom Typ der Basisklasse zugewiesen werden
  - Fahrzeug f; Auto a; ... f = a;
  - Variable f kann Fahrzeugobjekte speichern
  - Ein Auto ist ein Fahrzeug
  - Daher kann f auch Autoobjekte speichern
- Die Umkehrung gilt jedoch nicht!
  - d.h. a = f; ist verboten, da „unmöglich“!
  - Variable a kann Autoobjekte speichern
  - Ein Fahrzeug ist aber kein Auto (jedenfalls nicht immer)!
- „Gleichnis“ zur Zuweisungskompatibilität: Auf einem Parkplatz für Fahrzeuge dürfen Autos, Personenwagen, Velos,... abgestellt werden; auf einem Parkplatz für Velos jedoch keine beliebigen Fahrzeuge!



# Zuweisungskompatibilität

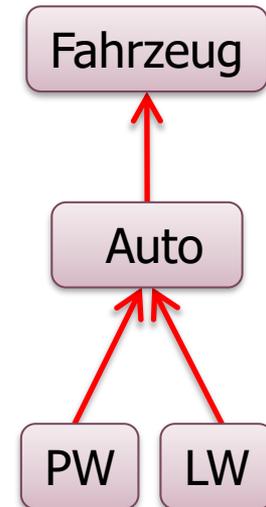
- Ein Auto ist ein Fahrzeug

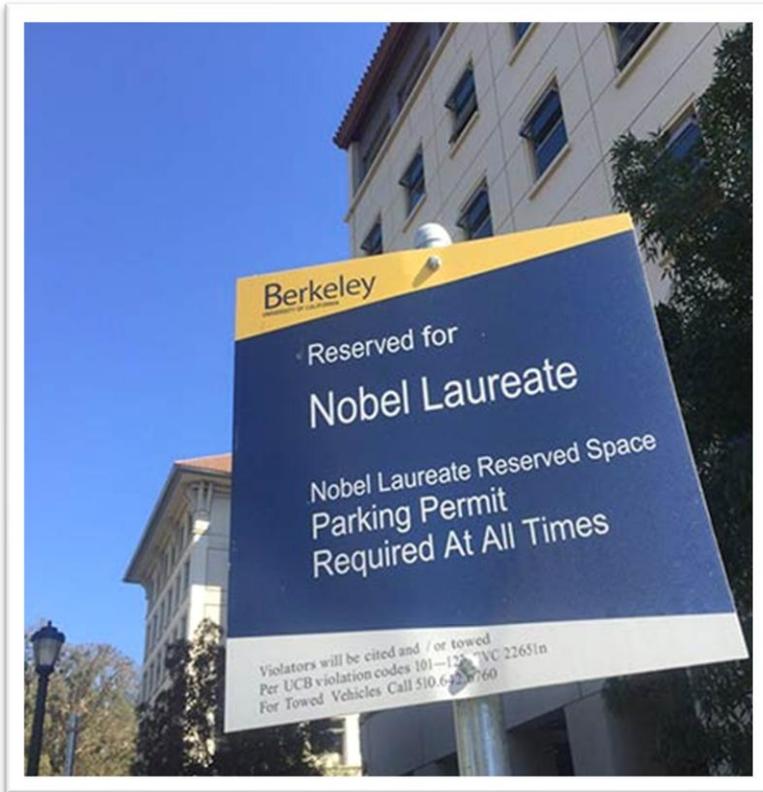
„is-a“-Relation ist asymmetrisch!

- Ein Fahrzeug ist aber kein Auto

Wieso ist die Welt so kompliziert, dass es Parkplätze gibt, wo nur manche Fahrzeugtypen zugelassen sind?

Wenn es dafür wirklich gute Gründe gibt, → gelten diese Gründe auch für Software?





(Univ. of California, Berkeley, seit 1980)

<https://pbs.twimg.com/media/CT2xngWcAAzUo?format=jpg>

(Ca. 1982 an der Universität Bonn, medizinische Fakultät Venusberg)



Wieso ist die **Welt so kompliziert**, dass es Parkplätze gibt, wo nur manche Fahrzeugtypen zugelassen sind?



<https://i.redd.it/jhixjvn796e01.jpg>

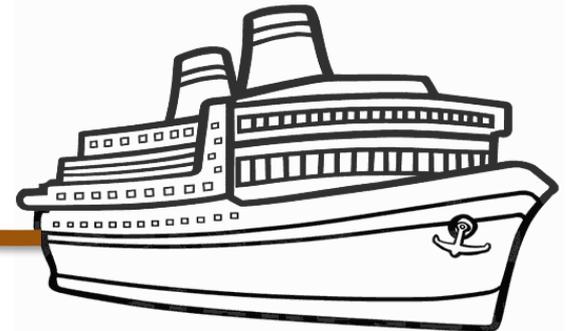
Auch in der Schweiz gibt es seit einiger Zeit reservierte Parkplätze für Nobelpreis-träger. Links am Institut für Molekularbiologie und Biophysik (IMBB) der ETH Zürich für **Kurt Wüthrich** (Nobelpreis für Chemie 2002) und rechts der Veloparkplatz von **Jacques Dubochet** an der Universität Lausanne (Nobelpreis für Chemie 2017).

# Zuweisungskompatibilität (2)

- Merke also: *Eine Variable vom Typ „Basisklasse“ darf auch ein Objekt einer abgeleiteten Klasse enthalten*
- Man nennt diese Eigenschaft auch **Polymorphie**, da eine Referenz auf Objekte *verschiedenen Typs* zeigen kann (bzw. eine Variable Werte unterschiedlichen Typs haben kann)

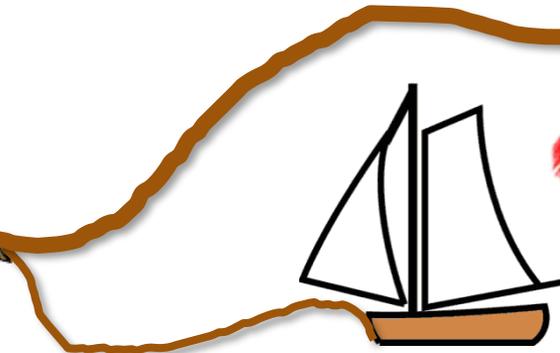
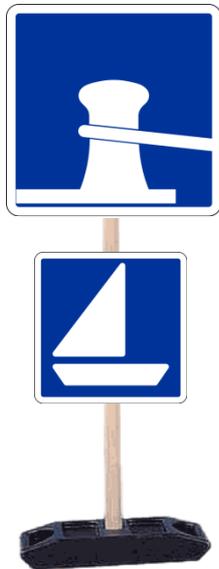


Für Schiffe



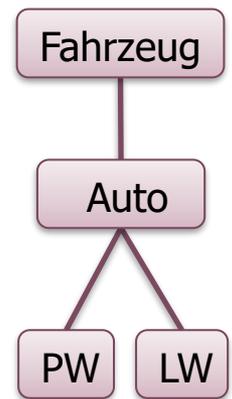
# Zuweisungskompatibilität (3)

- Merke also: *Eine Variable vom Typ „Basisklasse“ darf auch ein Objekt einer abgeleiteten Klasse enthalten*
- Man nennt diese Eigenschaft auch **Polymorphie**, da eine Referenz auf Objekte *verschiedenen Typs* zeigen kann (bzw. eine Variable Werte unterschiedlichen Typs haben kann)



# Zuweisungskompatibilität (4)

- Merke also: *Eine Variable vom Typ „Basisklasse“ darf auch ein Objekt einer abgeleiteten Klasse enthalten*
- Man nennt diese Eigenschaft auch **Polymorphie**, da eine Referenz auf Objekte *verschiedenen Typs* zeigen kann (bzw. eine Variable Werte unterschiedlichen Typs haben kann)
- Beispiel:* Eine Variable vom Typ „Referenz auf Fahrzeug“ kann zur Laufzeit zeitweise sowohl auf **PW-Objekte**, als auch auf **LW-Objekte** zeigen
  - Das ist sehr nützlich, wie wir noch sehen werden!



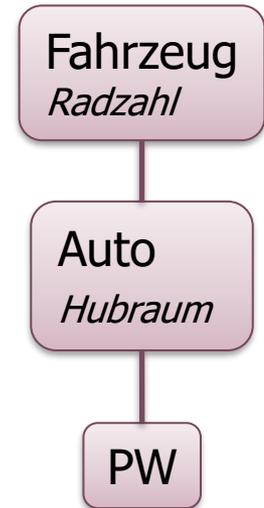
# Ein Java-Beispiel zur Zuweisungskompatibilität

```
class Fahrzeug {... int Radzahl;}  
class Auto extends Fahrzeug {... float Hubraum;}  
class PW extends Auto ...  
Fahrzeug f; Auto a; PW p; LW l;  
... new ...  
p.Hubraum = 1702;  
p.Radzah1 = 4;  
  
a = p;  
f = p;  
f = a;  
/* a = f; */  
/* ERROR: Incompatible types */  
a.Hubraum = 1100;  
f = a;  
System.out.println(f.Radzah1);  
System.out.println(f.Hubraum);  
ERROR: No variable Hubraum defined in Fahrzeug
```

Ein PW ist ein Auto  
und ein Fahrzeug

Ein Fahrzeug-Variable darf PW-  
Objekte oder Auto-Objekte speichern

Es wurde zwar Hubraum und Rad-  
zahl zugewiesen; Hubraum ist aber  
*über f* nicht zugreifbar; f hat nicht  
die dafür notwendige Qualifikation



# Typkonversion („type cast“)

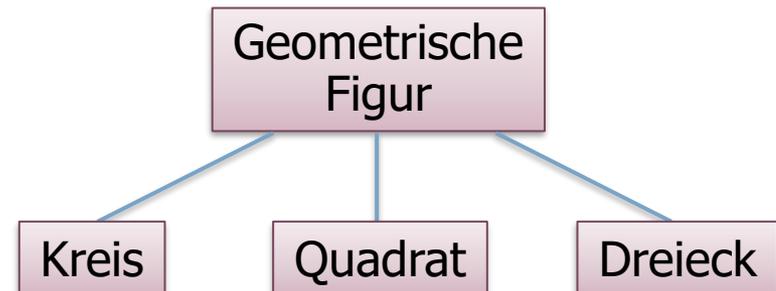
- Aus gutem Grund ist **f.Hubraum** verboten: Auf f könnte sich ja zufällig ein Fahrrad (ohne Hubraum!) befinden!
- Durch **explizite Typkonversion** kann man aber notfalls (wenn man überzeugt ist, dass konkret das Fahrzeug ein Auto ist) auch über f an den Hubraum des Auto-Objektes kommen:  
`System.out.println( ((Auto)f).Hubraum );`
- Aber wenn dort derzeit doch kein Auto (sondern z.B. ein Velo) parkt? Das gibt einen **Laufzeitfehler** „ClassCastException“!
- Dem kann man wiederum so vorbeugen:  
`if (f instanceof Auto)  
 System.out.println( ((Auto)f).Hubraum );  
else System.out.println("kein Auto, kein Hubraum!");`

# Polymorphie („Vielgestaltigkeit“)

- Eine Variable kann **Werte unterschiedlichen Typs** annehmen
  - Genauer: eine Referenz kann auf Objekte unterschiedlichen (aber „verwandten“) Typs verweisen
- Gleichnamige Methoden in Objekten **verwandten Typs**

## *Beispiel:*

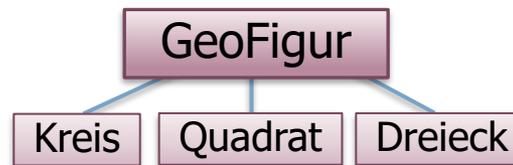
Methode „**Flächenwert**“ ist für alle Objekttypen realisiert, aber je nach Typ auf unterschiedliche Weise!



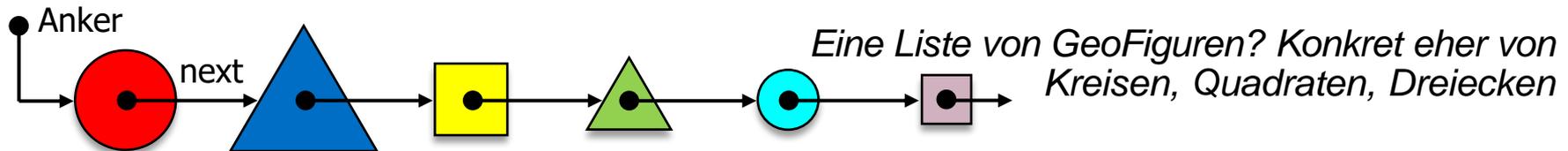
In gewisser Weise erlaubt es Polymorphie, „Äpfel und Birnen“ zu „addieren“: 2 Äpfel  $\cup$  3 Birnen ergibt dann 5 *Früchte* in einer polymorphen Menge

# Abstrakte Methoden und abstrakte Klassen

```
abstract class GeoFigur {  
    GeoFigur next;  
    public abstract  
        float flaechenwert();  
}
```



Diese Methode muss für jede von GeoFigur **abgeleitete** Klasse adäquat implementiert werden!



- **Abstrakte Methoden** „existieren“ nur dem Namen nach; sie werden in **abgeleiteten Klassen** jeweils **spezifisch implementiert**
- Klassen mit abstrakten Methoden müssen selbst „abstract“ sein
- Von **abstrakten Klassen** kann man **keine Objekte** mit new erzeugen
- **Abstrakte Klassen** dürfen aber auch **nicht-abstrakte Methoden** haben
  - Hier z.B.: farbe\_aendern()

*Informatisch gebildeter Händler:  
„Gnädige Frau,  
von abstrakten  
Klassen gibt es  
keine Objekte.“*



# Eine polymorphe Liste

```
class Dreieck extends GeoFigur {  
    // Koordinaten etc. als Attribute  
    public float flaechenwert() {  
        ... // bekannte Formel anwenden  
        System.out.print("Dreiecksfläche:"...);  
    }  
}
```

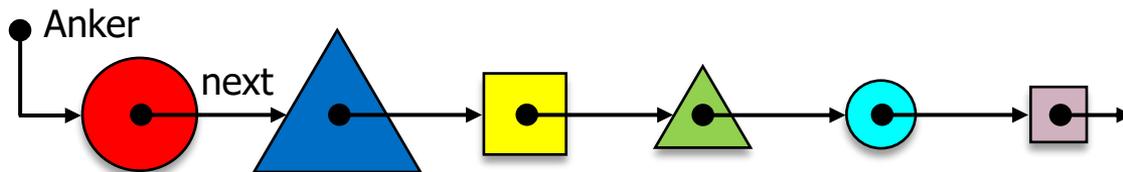
```
class Kreis extends GeoFigur {  
    ... ..flaechenwert() ...  
}
```

```
abstract class GeoFigur {  
    GeoFigur next;  
    public abstract  
        float flaechenwert();  
}
```

„Late binding“: Entscheidung  
dynamisch zur Laufzeit

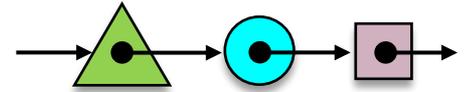
- Eine polymorphe Liste kann man dann z.B. so durchlaufen:

```
for (GeoFigur g = Anker; g != null; g = g.next)  
    f = f + g.flaechenwert();
```

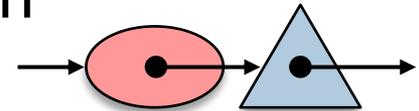


# Verarbeitung polymorpher Objekte

- **Analoge Verarbeitung** der einzelnen Objekte trotz jeweils **spezifischem (Unter-)Typ**
  - In der Praxis wird eine solche Verarbeitung i.a. deutlich komplexer sein, als hier mit „Flächenwert“ angedeutet



- **Ohne den Verarbeitungsalgorithmus anzupassen** können **neue Objekttypen** als Unterklassen von „GeoFigur“ eingeführt werden (z.B. „Ellipse“)
  - Nur **hinzufügen**, aber (im Idealfall) **nichts verändern**, vermindert den Software-Wartungsaufwand erheblich!



- **Late binding**: Es wird erst zur Laufzeit ermittelt, welche konkrete Methode jeweils „angesprungen“ wird; dies steht zur Übersetzungszeit (→ „early binding“) noch nicht fest!

# Generische Methoden und abstrakte Klassen

## Sort

Ich sortiere alles, was  
„kleiner“ entscheiden kann!

**Wähle mich dazu als Vorfahre!**

## Wörter

alphabetisch!

## Kunden

Vermögen!

## Rekruten

Körpergrösse!

## Partner

Schönheit!

## Schüler

Note!

Schüler **extends** Sort

**kleiner**: Meine Note  
< Note des anderen

# Generische Methoden und abstrakte Klassen (2)

```
abstract class Sort {  
    abstract boolean kleiner (Sort y);  
    static void sortiere(Sort[] Tab) {  
        // Sortiermethode: "exchange sort"  
        for (int i=0; i<Tab.length; i++)  
            for (int j=i+1; j<Tab.length; j++)  
                if (Tab[i].kleiner(Tab[j]))  
                    { Sort swap = Tab[i];  
                      Tab[i] = Tab[j];  
                      Tab[j] = swap;  
                    }  
    }  
}
```

Sortiert wird ein array von Werten

Dieses einfache Sortierverfahren („exchange sort“, verwandt mit „selection sort“ und nur oberflächlich ähnlich zu „Bubblesort“) ist allerdings ziemlich ineffizient!

Bei ernstern Anwendungen: Nicht vergessen, das Nötige (Sort? sortiere? kleiner?) mit **public** zu qualifizieren, wenn man ein Paket schnürt!

Achtung: Es wird absteigend sortiert!

- Wir fordern, dass die Klassen der zu sortierenden Objekte von **Sort** abgeleitet sind
- Dort muss dann die Methode „kleiner“ implementiert werden
  - als **totale Ordnungsrelation** auf den Objekten

# Generische Methoden und abstrakte Klassen (3)

- Unabhängig davon, wie die Relation „kleiner“ konkret definiert ist, funktioniert unser Sortierverfahren!
  - Das Sortierverfahren kann also bereits implementiert (und getestet) werden, bevor überhaupt die Daten selbst bekannt sind
- Einmal entwickelt, kann man den Algorithmus zum **Sortieren verschiedener Datentypen** / Dinge verwenden
  - int, float, Brüche, Zeichenketten, Rankingwerte von Unis,...



# Generische Methoden und abstrakte Klassen (4)

...oder, um die Substantive der Vorlesungssides nach der Länge zu sortieren:

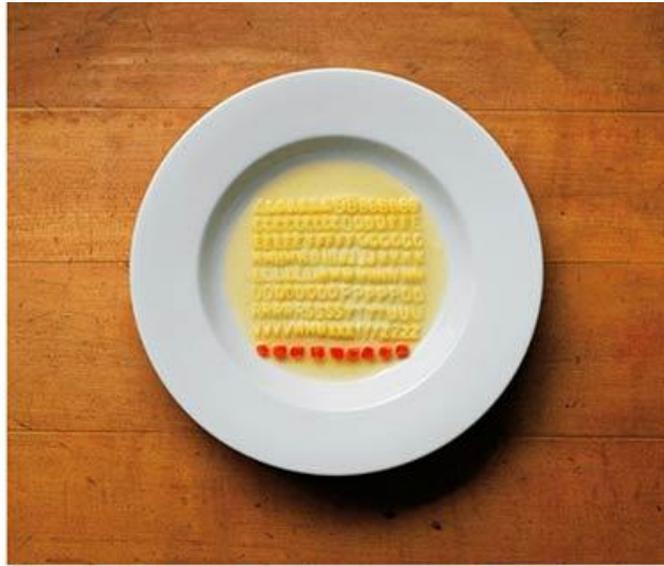


Selbstbaumöbelliebhaberbinärsuchanleitung  
Tendenzkorrelationskoeffizienten  
Panzerabwehrwaffenbeschaffungen  
Wahrscheinlichkeitsüberlegungen  
Kommunikationswissenschaftlerin  
Eintreffenswahrscheinlichkeit  
Implementierungsmöglichkeiten  
Differentialgleichungssystem  
Verteidigungsangelegenheiten  
Hardwarebeschreibungssprache  
Tabellenkalkulationsprogramm  
Dezimalklassifikationssystem  
Taschenrechnerfunktionalität  
Datenübertragungsfunktionen  
Demonstrationseinrichtungen  
Versicherungsgesellschaften  
Wahrscheinlichkeitsannahmen  
Hochgeschwindigkeitsrechnen  
Unterbrechungsanforderungen  
Volkswirtschaftsverwaltung  
Geschwindigkeitssteigerung



Komplexitätsgrößenordnung  
Kommunikationsverbindungen  
Funktionstabelleneinheiten  
Präsentationseigenschaften  
Rationalisierungspotential  
Hochleistungsrechenzentrum  
Gleichgewichtsgesellschaft  
Antidiskriminierungsgesetz  
Rechenmaschinenfabrikanten  
Multiplikationsalgorithmus  
Maschineninstruktionsebene  
Rüstungsforschungsinstitut  
Grossstadttelephonzentrale  
Zeitintegrationsverfahren  
Anforderungsspezifikation  
Terminplanimplementierung  
Rückkoppelungsmechanismen  
Gewinnauszahlungsfunktion  
Parameterübergabesemantik  
Gefechtssimulationsmodell  
Verschlüsselungsmaschinen

# Eine Sortieranwendung: Finden statt suchen



Der Schweizer Aktionskünstler [Ursus Wehrli](#) wurde 2002 durch das Buch „Kunst aufräumen“ bekannt; er arrangierte die Elemente klassischer Bilder neu in geometrisch geordneter Weise.

**Ist Q in der Buchstabensuppe?**



**Ist ein grünes Handtuch vorhanden?**

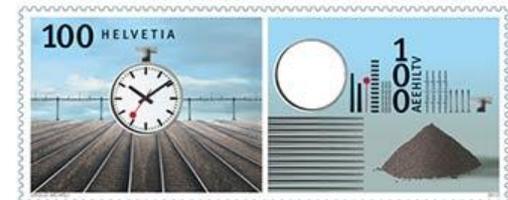
2011 erschien „Die Kunst, aufzuräumen“: Der in Zürich lebende Ursus Wehrli räumt nun mit allem auf – Buchstabensuppen, Badis,... nichts entgeht seiner ordnenden Hand.

# Sortieren = Ordnen = Aufräumen?

„Wehrli kommt nämlich aus der Schweiz, und dort ist die Ordnung so etwas wie die vierte Macht im Staat.“  
– Wiener Zeitung, 9.6.12



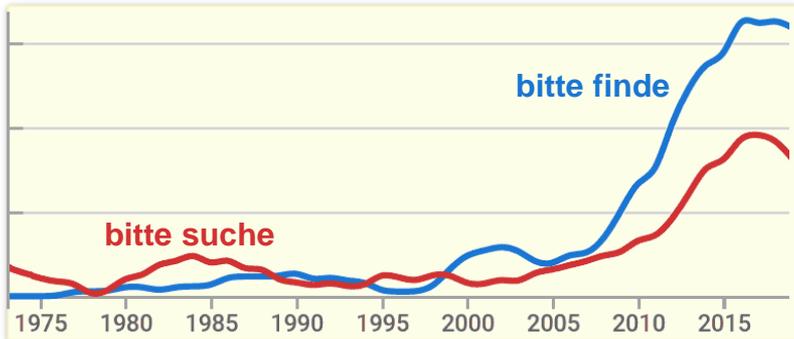
Wir konsultieren den **Duden**: Das Verb »**ordnen**« ist aus lateinisch *ordinare* »in Reihen zusammenstellen« entlehnt. Dies ist von lateinisch *ordo (ordinis)* »Reihe; Ordnung; Rang, Stand« abgeleitet. Beispiele: Akten, Bücher ordnen; etwas nach der Größe, die Stichwörter nach dem Alphabet, die Belege chronologisch / der Reihe nach ordnen; etwas in Mappen ordnen; Blumen zu einem Strauß ordnen; seine Gedanken ordnen; hier fehlt eine ordnende Hand; seinen Nachlass ordnen, sie lebt in geordneten Verhältnissen; ein geordneter Rückzug; seinen Anzug ordnen, nach dem Mittagsschlaf seine Haare, Kleider ordnen; sich zum Festzug ordnen; alles hatte sich sinnvoll geordnet.



**Sortieren**: »in (Güte)klassen einteilen, auslesen, sondern, ordnen« (16. Jahrhundert; aus gleichbedeutend italienisch *sortire*, das seinerseits lateinisch *sortiri* »[er]losen; auswählen« fortsetzt; vgl. franz. *sort* »Los, Schicksal«). Beispiele: Waren, Papiere sortieren; etwas nach der Größe, nach Qualität sortieren; etwas alphabetisch, maschinell sortieren; die Mutter sortiert die Wäsche in den Schrank; ein gut sortiertes Lager; das Geschäft ist in Weinen gut sortiert.

**Aufräumen**: Ordnung schaffen, an seinen Platz stellen; aus dem Weg räumen, wegmachen, beseitigen, aus der Welt schaffen, zum Verschwinden bringen, eliminieren, ausradieren, Schluss machen.

# Finden statt suchen



Früher konnte man im Deutschen zwar das Suchen befehlen („Suche!“ oder „Bitte suchen Sie!“) und dann auf das Finden als Erfolg der Suche hoffen, aber man konnte natürlich nicht das Finden als solches anordnen.

Es muss dann um 1975 gewesen sein, als sich jemand gedankenlos die nette englische Floskel „please find enclosed“ in einem Brief (heute heisst es bei E-Mail ja „please find *attached*“ stattdessen) auf Deutsch zu eigen machte und wörtlich-analog „bitte finden Sie“ bzw. „bitte finde“ formulierte. Wahrscheinlich kreierten das mehrere Deutschschreiber unabhängig voneinander, jedenfalls setzte sich die Redewendung im Verlauf der Jahre immer mehr durch und fällt Vielen gar nicht mehr negativ auf. Dabei gäbe es auch mit dem Wort „finden“ korrekte Formulierungen auf Deutsch: „Im Anhang finden Sie“ oder kürzer: „Anbei finden Sie“. Besser ist aber vielleicht „Anbei erhalten Sie“ oder „... ist beigefügt“; je nach Kontext (z.B. bei „please find details...“) auch Umschreibungen wie „Einzelheiten entnehmen Sie...“ etc. Aber Sprache verändert sich laufend – vielleicht setzt sich der flachere englische Stil gegenüber dem gestelzten deutschen Stil langfristig durch?

\*) Bzw. „In der Anlage“ bei Briefen statt E-Mail

When I was new to the job market and mailing out resumes [...] I sent my carefully crafted cover letters with a note that read: *Enclosed please find my resume*. One such mailing resulted in an interview. There I was in the wood-paneled office of an immaculately groomed lawyer. While I waited anxiously in an oversized leather wingback chair, he sat at his desk clicking his pen top and scanning my resume and cover letter. He looked up suddenly and grinned, pointing at the letter. “I love it when people write ‘Enclosed *please find* my resume.’ I didn’t even know your resume was lost!”

It was an embarrassing moment. [...] I never used *enclosed please find* again. [...] There’s no need to boss around the other person to go about finding things. [...] In fact, it would be just plain weird to put this into another request form like *Could you please find the document attached?* or *I would be very grateful if you would find the document attached*. -- Karen Hertzberg, [www.grammarly.com/blog/please-find-attached/](http://www.grammarly.com/blog/please-find-attached/)

# Anwendung der Sort-Klasse für int-Werte

- Es sollen hier einfache **int-Werte** sortiert werden, und (ohnehin unter Benutzung der existierenden generischen Sortieroutine e diese kennen zu müssen!):

```
class IntSort extends Sort {  
    int w;  
    IntSort(int i) {  
        w = i;  
    }  
    boolean kleiner(Sort y) {  
        return w < ((IntSort)y.w);  
    }  
}
```

Hier wird die Relation „kleiner“ definiert und implementiert

Konstruktor

Dies ist die vom Anwender bereitgestellte Klasse (aufbauend auf der Basisklasse Sort)

Hier darf leider nicht IntSort stehen; „kleiner“ erwartet einen Parameter vom Typ „Sort“

Typkonversion von y (von „Sort“ nach „IntSort“)

# Anwendung der Sort-Klasse für int-Werte (2)

```
class ... {  
    ...  
    IntSort[] Tabelle = new IntSort[12];  
    for (int i=0; i<Tabelle.length; i++) {  
        Tabelle[i] = new IntSort((int)(Math.random()*20.0));  
        System.out.print(" " + Tabelle[i].w);  
    }  
    IntSort.sortiere(Tabelle);  
    System.out.println();  
    for (int i=0; i<Tabelle.length; i++)  
        System.out.print(" " + Tabelle[i].w);  
    System.out.println();  
}
```

Diese Klasse verwendet beispielhaft IntSort

Füllen mit Zufallszahlen

Name der Klasse, nicht einer Referenz!

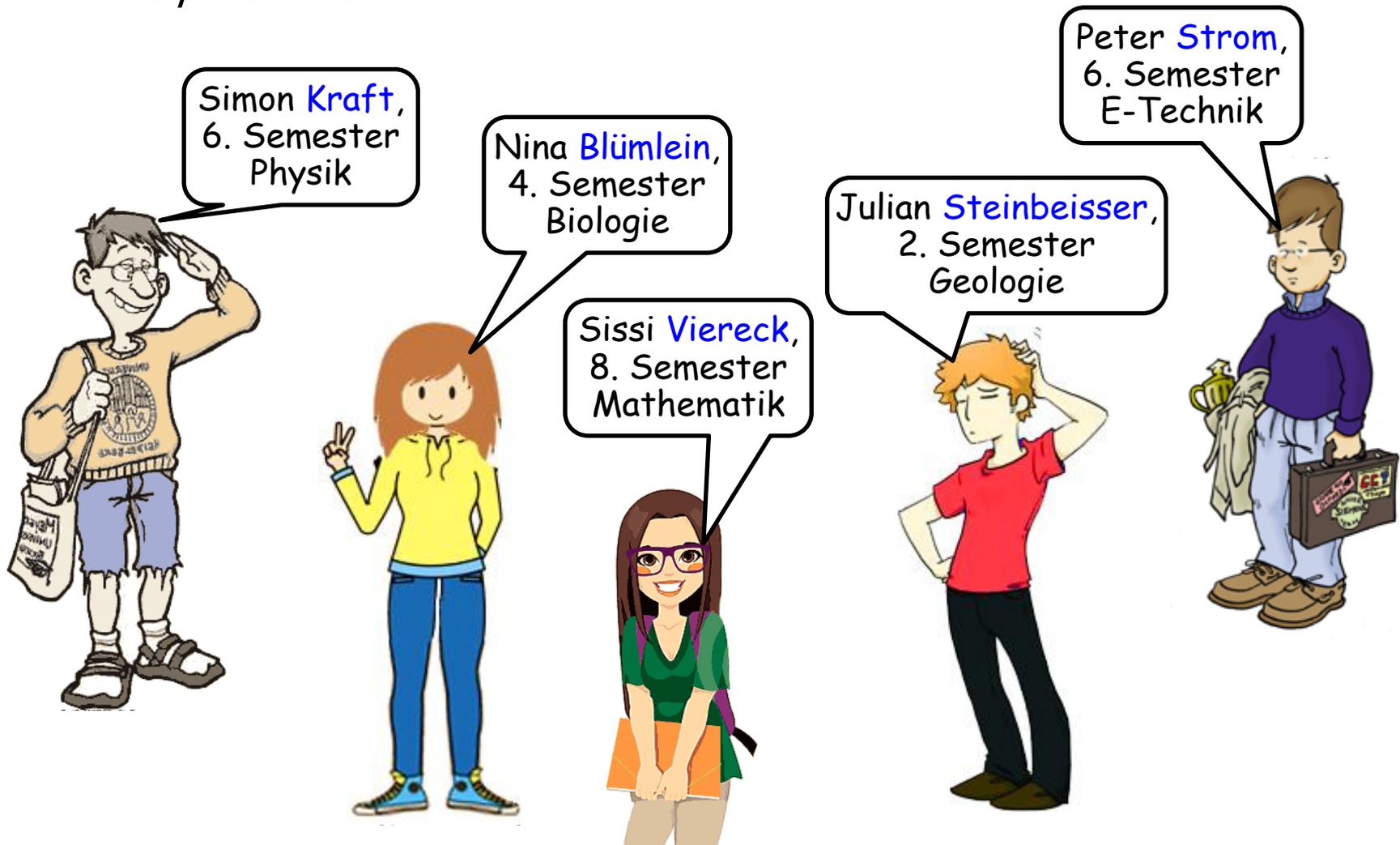
IntSort, von Sort abgeleitet, erbt die Methode „sortiere“

Zum Sortieren einfacher Zahlen ist das Prinzip einer generischen Oberklasse zum Sortieren ein Overkill → Beispiel mit komplexeren Daten folgt

# Eine Sortieranwendung – Studi-Daten

Name – Semester – Fach

*Dramatis personae:*



# Eine Sortieranwendung – Studi-Daten

```
class Studi extends Sort {
```

```
    int Semester;  
    String Name, Fach;
```

```
    Studi(int S, String F, String N) {  
        Semester = S; Konstruktor  
        Fach = F;  
        Name = N;  
    }
```

```
    boolean kleiner(Sort y) {  
        Studi s = (Studi)y;
```

```
        return ((Semester < s.Semester)
```

Hier erfüllen wir die Implementierungspflicht von „kleiner“

```
        || ((Semester == s.Semester) && (Fach.compareTo(s.Fach) < 0))  
        || ((Semester == s.Semester) && (Fach.compareTo(s.Fach) == 0)  
        && (Name.compareTo(s.Name) < 0)));  
    }
```

```
}
```

Es wird primär nach Semesterzahl (absteigend) sortiert, nur bei Gleichheit dann weiter bezüglich Fach und Name

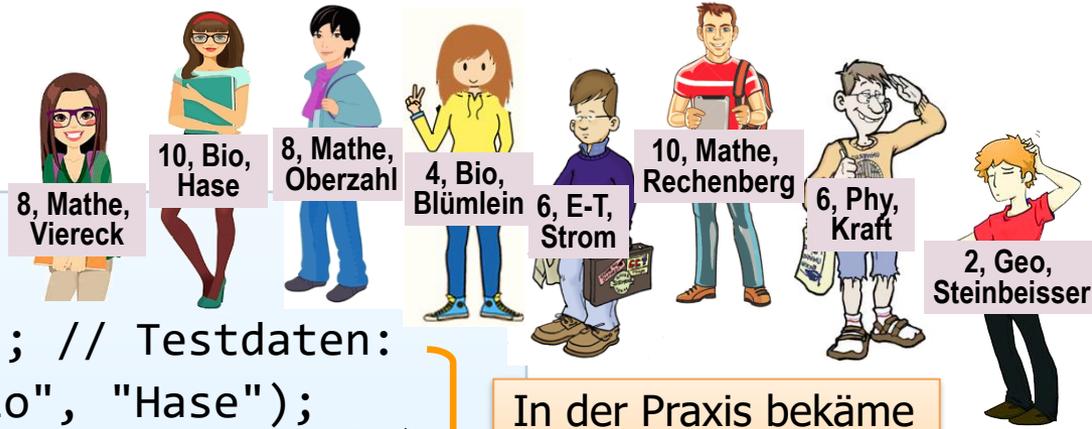
```
abstract class Sort {  
    abstract boolean kleiner (Sort y);  
    static void sortiere  
    {  
        ... if ... kleiner ...  
    }  
}
```

Hier darf leider nicht Studi stehen!

Typkonversion von Sort nach Studi

*compareTo* vergleicht zwei Strings lexikographisch:  
Resultat: < 0, = 0, oder > 0

# i Studi.sortiere !



```
class ...
...
Studi[] Tab = new Studi[8]; // Testdaten:
Tab[0] = new Studi(10, "Bio", "Hase");
Tab[1] = new Studi(8, "Mathe", "Oberzahl");
...
Tab[7] = new Studi(6, "Physik", "Kraft");

Studi.sortiere(Tab);

for (int i=0; i<Tab.length; i++) // Ausgabe
    System.out.println(Tab[i].Name + ", " +
                       Tab[i].Fach + ", " +
                       Tab[i].Semester);
}
```

In der Praxis bekäme man diese Daten von einer Datenbank

Es wird ein Array vom Typ Sort erwartet, aber ein Array vom Typ Studi übergeben – das ist OK

Übergabe (per „value“) einer Referenz auf das Array

Die Klasse Studi, von Sort abgeleitet, kennt auch die Methode „sortiere“

Beachte: „Studi“ ist der Name einer Klasse, nicht einer Referenz → „sortiere“ ist eine klassenbezogene Funktionalität („static“)

# Das Sortierergebnis

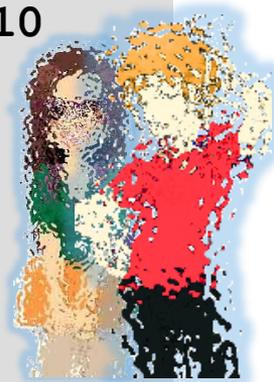
## Die Eingabedaten

```
10 Bio Hase
8 Mathe Oberzahl
8 Mathe Viereck
2 Geologie Steinbeisser
4 Bio Blümlein
6 E-Technik Strom
10 Mathe Rechenberg
6 Physik Kraft
```



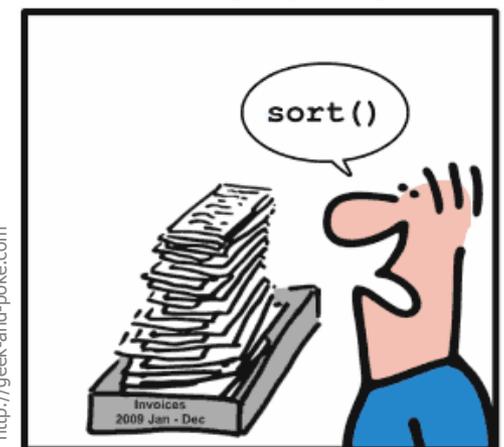
Das Sortierergebnis (absteigend sortiert entsprechend dem festgelegten Sortierkriterium „Semester-Fach-Name“)

```
Rechenberg, Mathe, 10
Hase, Bio, 10
Viereck, Mathe, 8
Oberzahl, Mathe, 8
Kraft, Physik, 6
Strom, E-Technik, 6
Blümlein, Bio, 4
Steinbeisser, Geologie, 2
```



- Um **eigene Datentypen** sortieren zu können, muss man die entsprechende Klasse also nur als von „Sort“ abgeleitet deklarieren  
`class ... extends Sort`
- und in seiner Klasse die „kleiner“-Methode geeignet festlegen  
`boolean kleiner(Sort ...) ...`

SIMPLY EXPLAINED

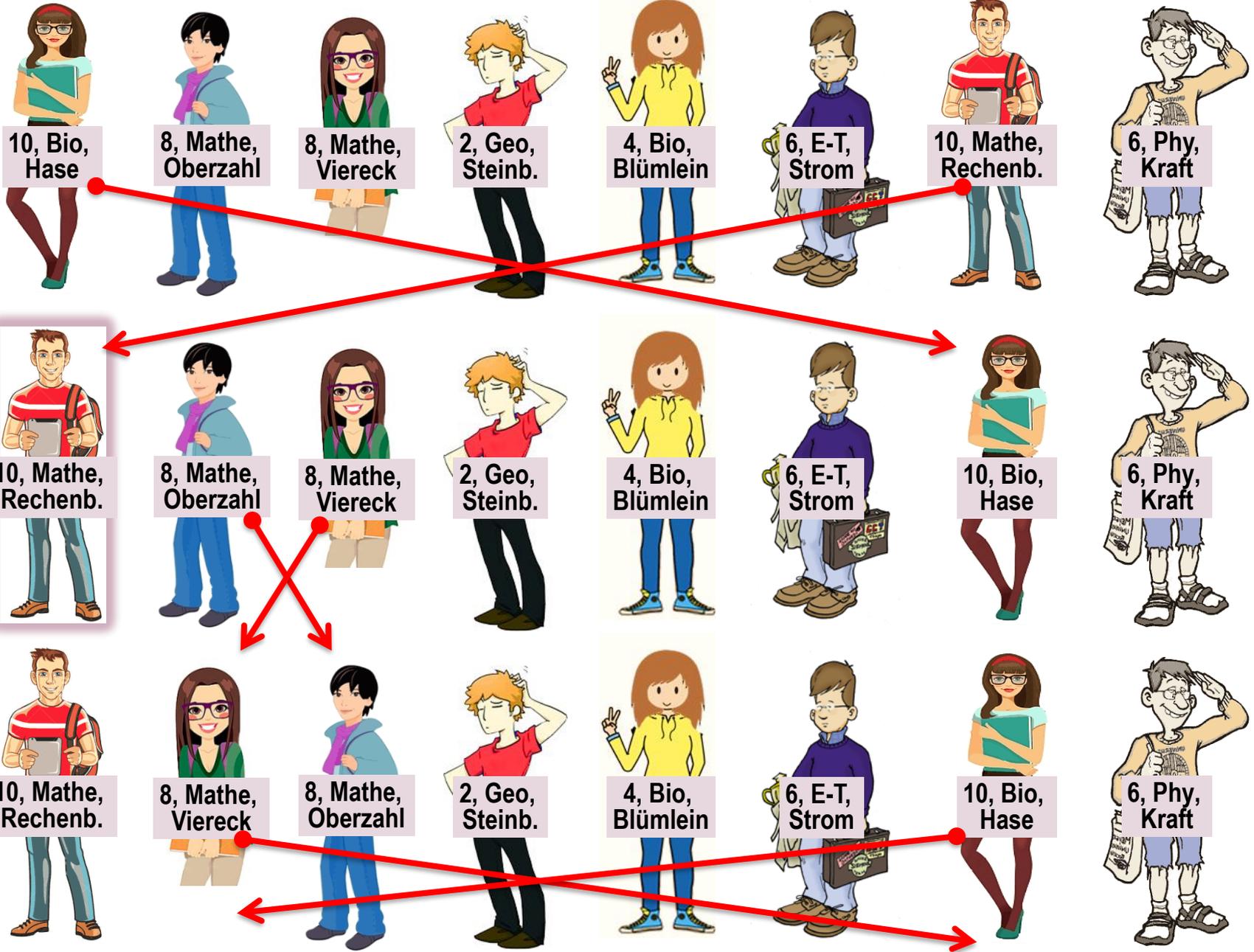


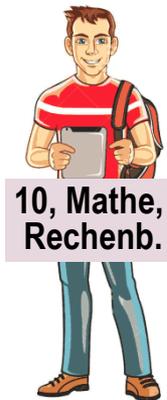
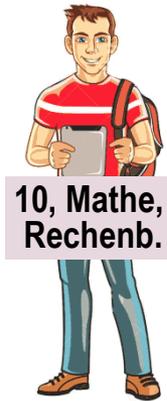
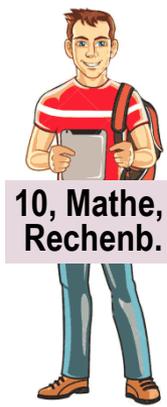
OBJECT ORIENTATION

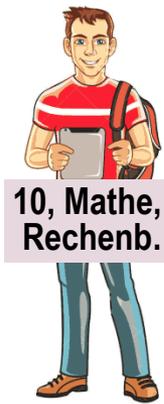
# Zum Sortierverfahren („exchange sort“)

- Prinzip: Angenommen, auf den Plätzen  $0, \dots, i-1$  stehen schon die richtigen Elemente. Auf Platz  $i$  soll nun das **grösste der restlichen Elemente** von den Plätzen weiter rechts ( $i, i+1, \dots$ ) kommen.
- Immer dann, wenn die innere  $j$ -Schleife weiter rechts ein „neues grösseres“ Element (als das aktuelle auf Platz  $i$ ) gefunden hat, wird es mit diesem **vertauscht**.
- Es wird dann aber weitergesucht, denn danach könnte ja noch weiter rechts ein noch grösseres kommen ( $\rightarrow$  „**Verlobungsprinzip**“: Festhalten und weitersuchen nach besseren Kandidaten!).
- Hat man so schliesslich auf Platz  $i$  das grösste unter den Restelementen  $i, i+1, \dots$  gesetzt, dann macht die äussere  $i$ -Schleife einen Schritt und wiederholt das Ganze mit dem „neuen“, um 1 grösseren,  $i$ .
- **Exchange sort** ist ein einfaches, aber bzgl. Zeitbedarf vergleichsweise ineffizientes Sortierverfahren; es ähnelt „**selection sort**“.
  - Bei „selection sort“ wird aber für ein festes  $i$  der äusseren  $i$ -Schleife nie mehrfach vertauscht, sondern nur der jeweilige Index des neuen Kandidaten gemerkt und nach Ende der inneren  $j$ -Schleife ein einziges Mal vertauscht; siehe auch <http://de.wikipedia.org/wiki/Selectionsort>.

# Illustration des Sortierverfahrens







10, Mathe,  
Rechenb.



10, Bio,  
Hase



8, Mathe,  
Viereck



6, E-T,  
Strom



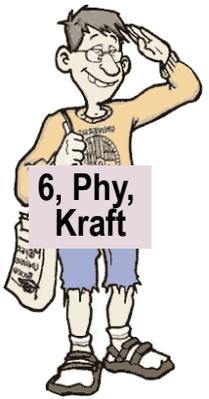
2, Geo,  
Steinb.



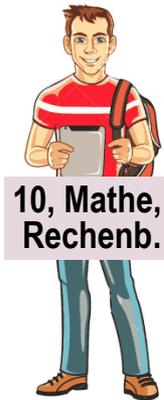
4, Bio,  
Blümlein



8, Mathe,  
Oberzahl



6, Phy,  
Kraft



10, Mathe,  
Rechenb.



10, Bio,  
Hase



8, Mathe,  
Viereck



8, Mathe,  
Oberzahl



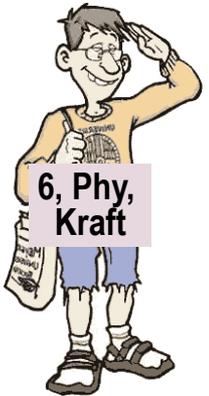
2, Geo,  
Steinb.



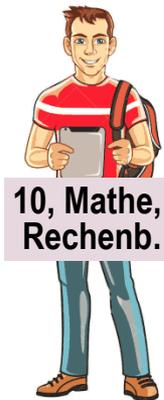
4, Bio,  
Blümlein



6, E-T,  
Strom



6, Phy,  
Kraft



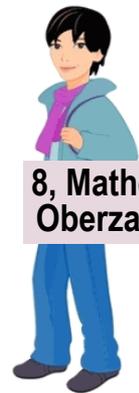
10, Mathe,  
Rechenb.



10, Bio,  
Hase



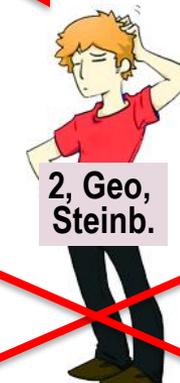
8, Mathe,  
Viereck



8, Mathe,  
Oberzahl



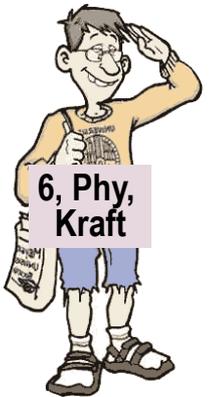
4, Bio,  
Blümlein



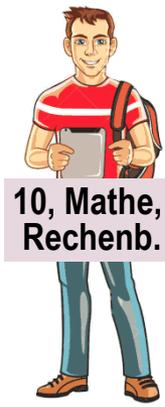
2, Geo,  
Steinb.



6, E-T,  
Strom



6, Phy,  
Kraft



10, Mathe, Rechenb.



10, Bio, Hase



8, Mathe, Viereck



8, Mathe, Oberzahl



6, E-T, Strom



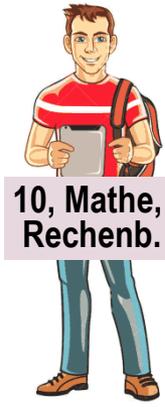
2, Geo, Steinb.



4, Bio, Blümlein



6, Phy, Kraft



10, Mathe, Rechenb.



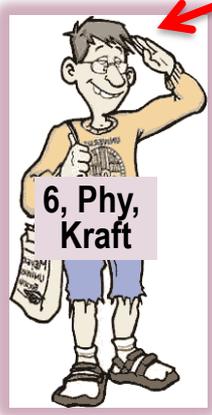
10, Bio, Hase



8, Mathe, Viereck



8, Mathe, Oberzahl



6, Phy, Kraft



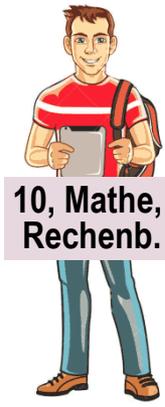
2, Geo, Steinb.



4, Bio, Blümlein



6, E-T, Strom



10, Mathe, Rechenb.



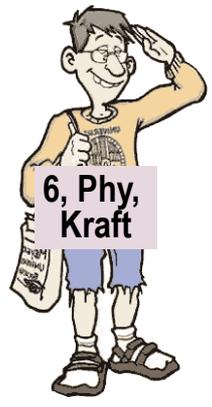
10, Bio, Hase



8, Mathe, Viereck



8, Mathe, Oberzahl



6, Phy, Kraft



4, Bio, Blümlein



2, Geo, Steinb.



6, E-T, Strom



10, Mathe,  
Rechenb.



10, Bio,  
Hase



8, Mathe,  
Viereck



8, Mathe,  
Oberzahl



6, Phy,  
Kraft



6, E-T,  
Strom



2, Geo,  
Steinb.



4, Bio,  
Blümlein



10, Mathe,  
Rechenb.



10, Bio,  
Hase



8, Mathe,  
Viereck



8, Mathe,  
Oberzahl



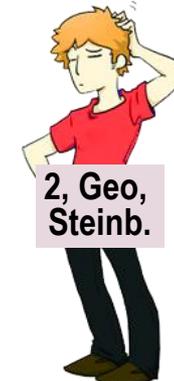
6, Phy,  
Kraft



6, E-T,  
Strom



4, Bio,  
Blümlein



2, Geo,  
Steinb.

**Happy End:  
Richtig sortiert!**



# Sortieren geometrischer Figuren nach ihrer Fläche

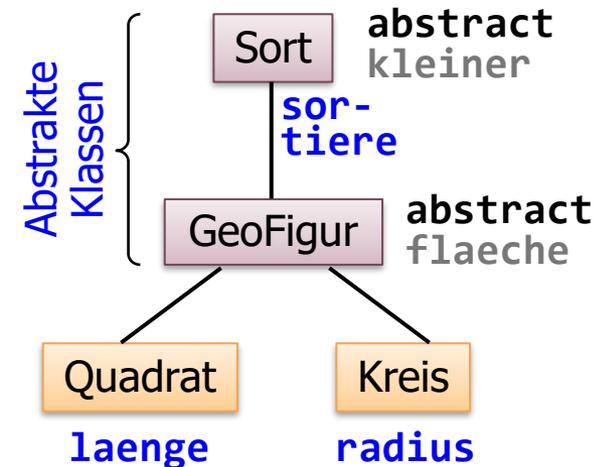
```
abstract class GeoFigur extends Sort {  
    public abstract float flaeche();  
    boolean kleiner (Sort y) {  
        return  
            flaeche() < (GeoFigur)y.flaeche();  
    }  
}
```

```
class Quadrat extends GeoFigur {  
    float laenge;  
    public float flaeche() {  
        return (laenge*laenge);  
    }  
    Quadrat() { Konstruktor  
        laenge = KbdInput.readInt("Länge? ");  
    }  
}
```

quick & dirty

Objekt in Gründung erfragt interaktive seine Grösse

```
class Kreis extends GeoFigur { Analog zu Quadrat  
    float radius;  
    public float flaeche() {  
        return (3.1415926536*radius*radius);  
    }  
    Kreis() {  
        radius = KbdInput.readInt("Radius? ");  
    }  
}
```



Die Fläche wird jeweils  
typspezifisch berechnet

Eine GeoFigur wird in der Praxis  
noch weitere Attribute haben –  
z.B. Koordinaten (x,y) zur Verortung  
auf der Zeichenfläche oder  
eine Füllfarbe, eine ID-Nr. etc.

# Das zugehörige Testprogramm

```

public static void main(String[] args) {
    GeoFigur[] Tabelle = new GeoFigur[3];
    for (int i=0; i<Tabelle.length; i++) {
        String s = KbdInput.readString
            ("Kreis (k) oder Quadrat (q)? ");
        if (s.compareTo("k") == 0)
            Tabelle[i] = new Kreis();
        else
            Tabelle[i] = new Quadrat();
    }
    GeoFigur.sortiere(Tabelle);

    System.out.println();

    for (int i=0; i<Tabelle.length; i++)
        System.out.print
            (" " + Tabelle[i].flaeche() );
    System.out.println();
}

```

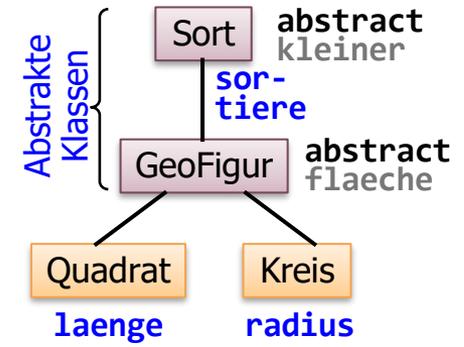
quick & dirty

Poly-  
morphie

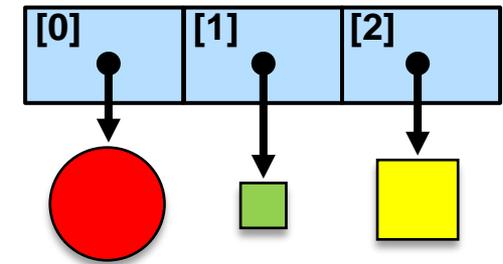
GeoFigur erbt  
die Methode  
„sortiere“

late binding

Hier wird jeweils die  
„richtige“ Methode für  
die Fläche verwendet



compareTo vergleicht zwei  
Strings lexikographisch:  
Resultat: <0, =0, oder >0



Kreis (k) oder Quadrat (q)? **k**  
 Radius? **5**  
 Kreis (k) oder Quadrat (q)? **q**  
 Länge? **4**  
 Kreis (k) oder Quadrat (q)? **q**  
 Länge? **7**

78.5397    49    16

# Resümee des Kapitels

- **Objektorientiertes Programmieren: Klassenkonzept**
  - Idee: Struktur- und konzepttreue Modellierung eines Realweltausschnitts
  - Konzepte: Begriffs- bzw. Klassenhierarchie, is-a-Relation, Vererbung
  - Praxis der OO-Programmierung:
    - Zuweisungskompatibilität von Variablen verschiedener Hierarchiestufe
    - Zugriff auf Attribute und Methoden abgeleiteter Klassen
    - Abstrakte Methoden / Klassen
- **Polymorphie**
  - Beispiel: polymorphe Listen
- **Generische (d.h. typunabhängige) Algorithmen**
  - **Sortierverfahren** für beliebige total geordnete Objekte, z.B.:
    - Studi-Objekte nach primären / sekundären Kriterien sortieren
    - Geo-Figuren verschiedenen Typs nach Fläche sortieren

