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)

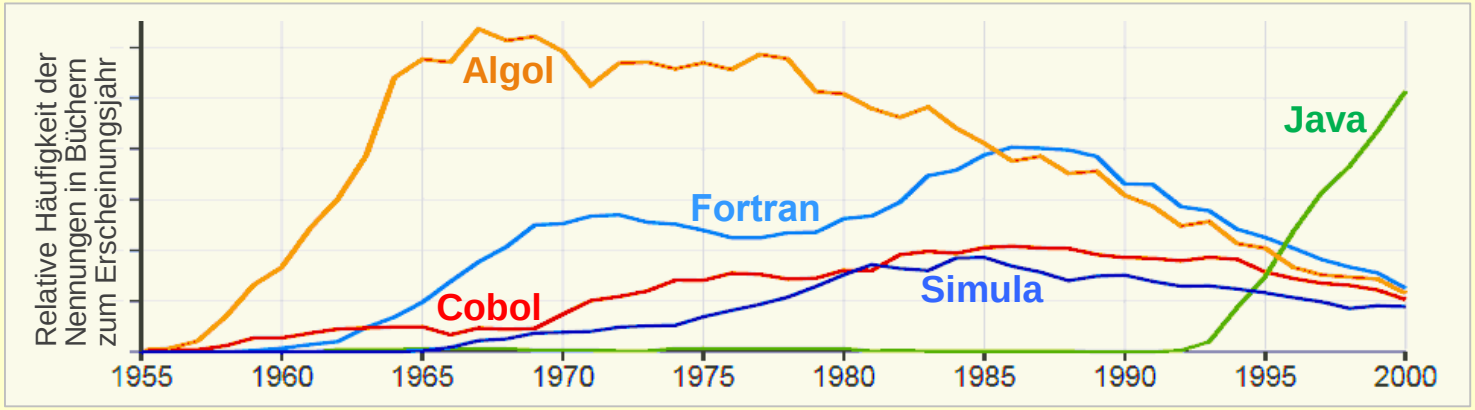
zwischen diversen Objekten kommen: Kunden mieten ein Auto; ein Auto kommt an einer Strassenkreuzung an; wenn zu viele Autos in ein Parkhaus fahren, wird es voll und neu ankommende Autos werden abgewiesen etc. Und vielleicht ist es zweckmässig, Autos in zwei Unterklassen zu separieren: Personenwagen und 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 Gegenstandbereichs 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 überzeugte, 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 Prinzipen 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.

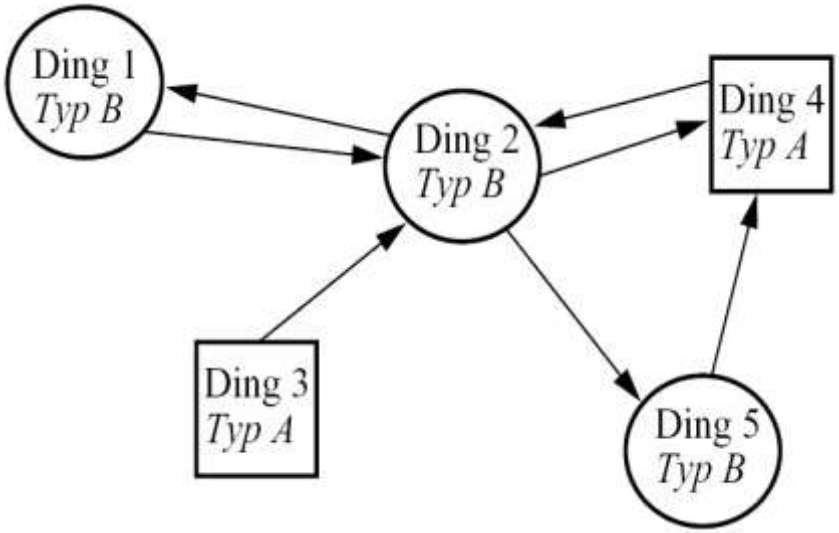
Thema / Inhalt (4)

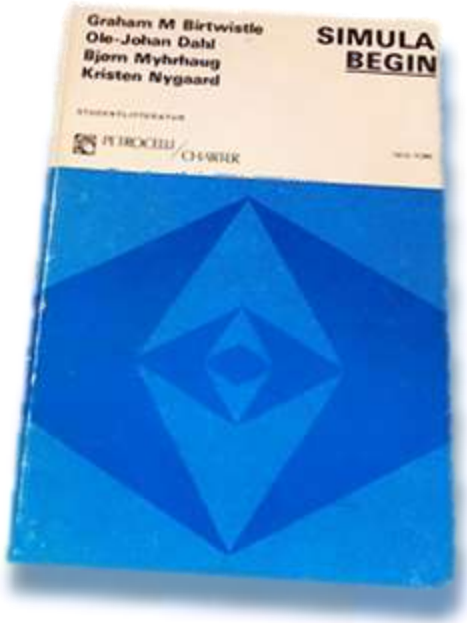
Für ein konkretes Beispiel, bei dem die diversen Aspekte zusammenspielen, bilden wir zunächst eine **Klassenhierarchie geometrischer Objekte** und entwickeln sodann einen **typgenerischen Sortieralgorithmus**. Nachdem wir das Sortierverfahren an einfachen Verbunddatentypen mit je drei relevante Datenelementen „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

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

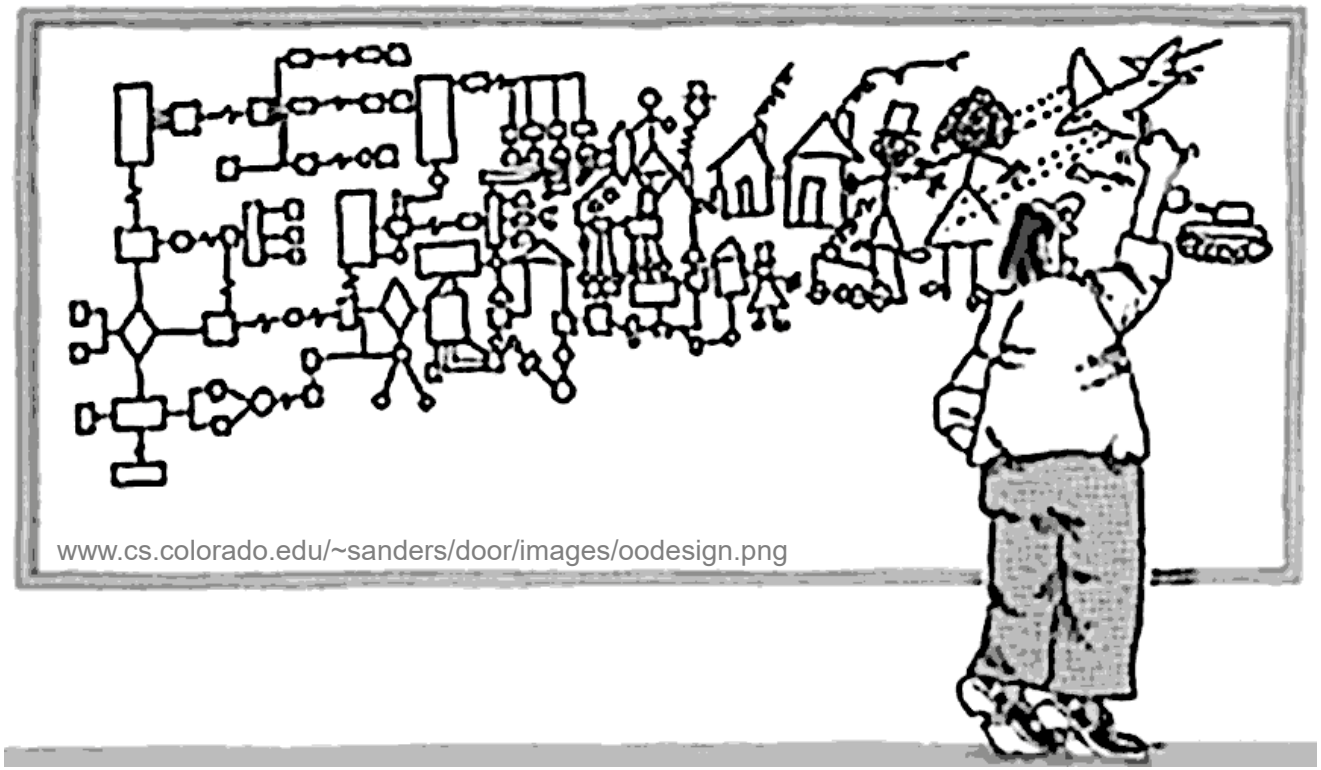




Simulationssprache SIMULA war die erste objektorientierte Programmiersprache (1967)

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

Objektorientiertes Programmieren



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

Objekte

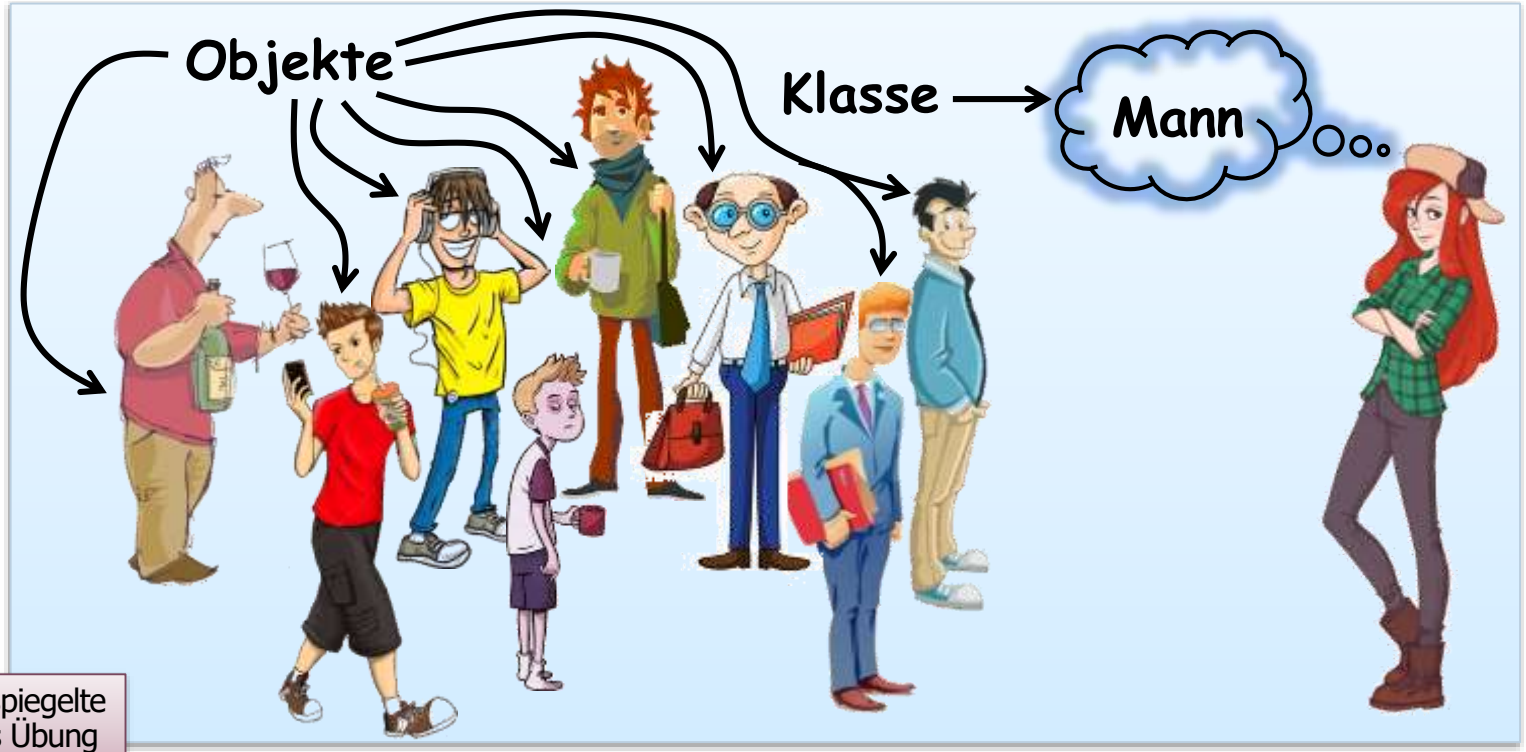
- Sind autonome, gekapselte Einheiten eines bestimmten Typs
- Haben einen eigenen Zustand
  (= lokale Variablen)
- Besitzen ein Verhalten
  (wirksam, wenn sie bzw. ihre Methoden aktiviert werden)
- Bieten anderen Objekten Dienstleistungen an
  - Durchführung von Berechnungen
  - Änderungen des lokalen Zustandes
  - Zurückliefern von Variablenwerten oder Berechnungsergebnissen
  - Allgemein: „Reaktion“ auf Aufruf einer Methode

Objektorientierter Softwareentwurf

Think big!

Strukturierung der Problemlösung als eine Menge kooperierender Objekte

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



Gendergespiegelte Version als Übung

Objektorientierter Softwareentwurf

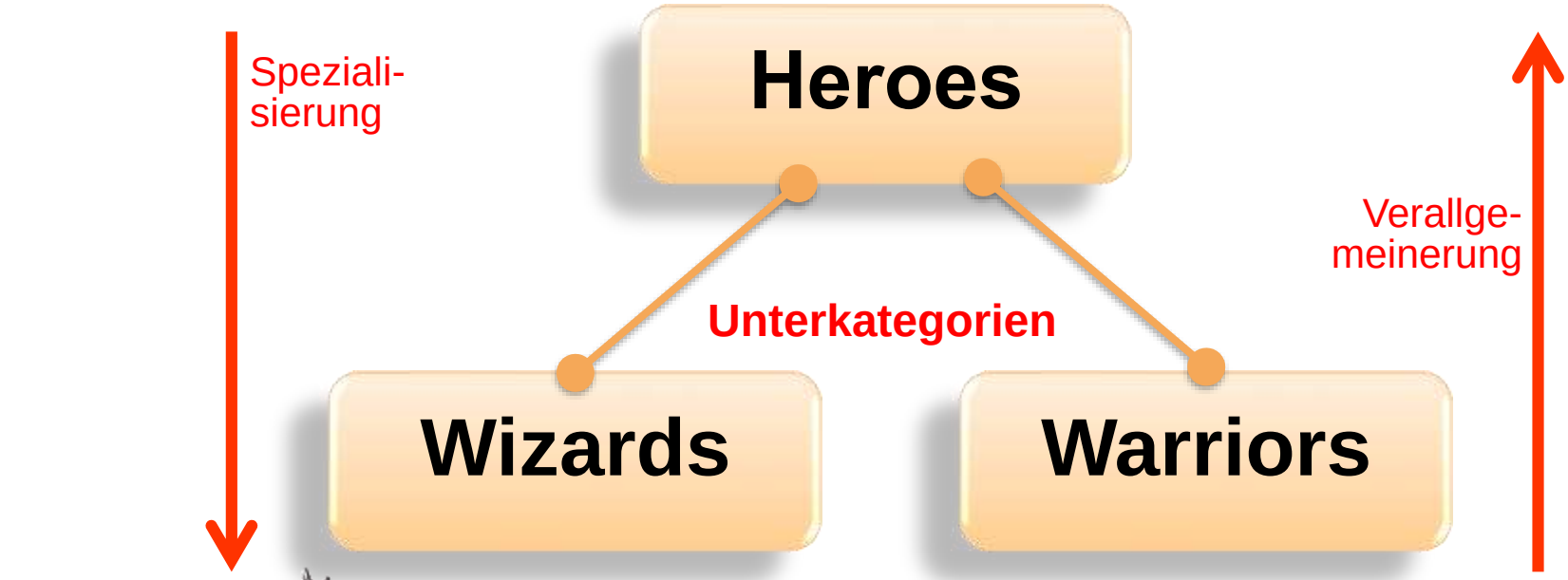
Think big!

Strukturierung der Problemlösung als eine Menge kooperierender Objekte

Top-Down-Entwurf

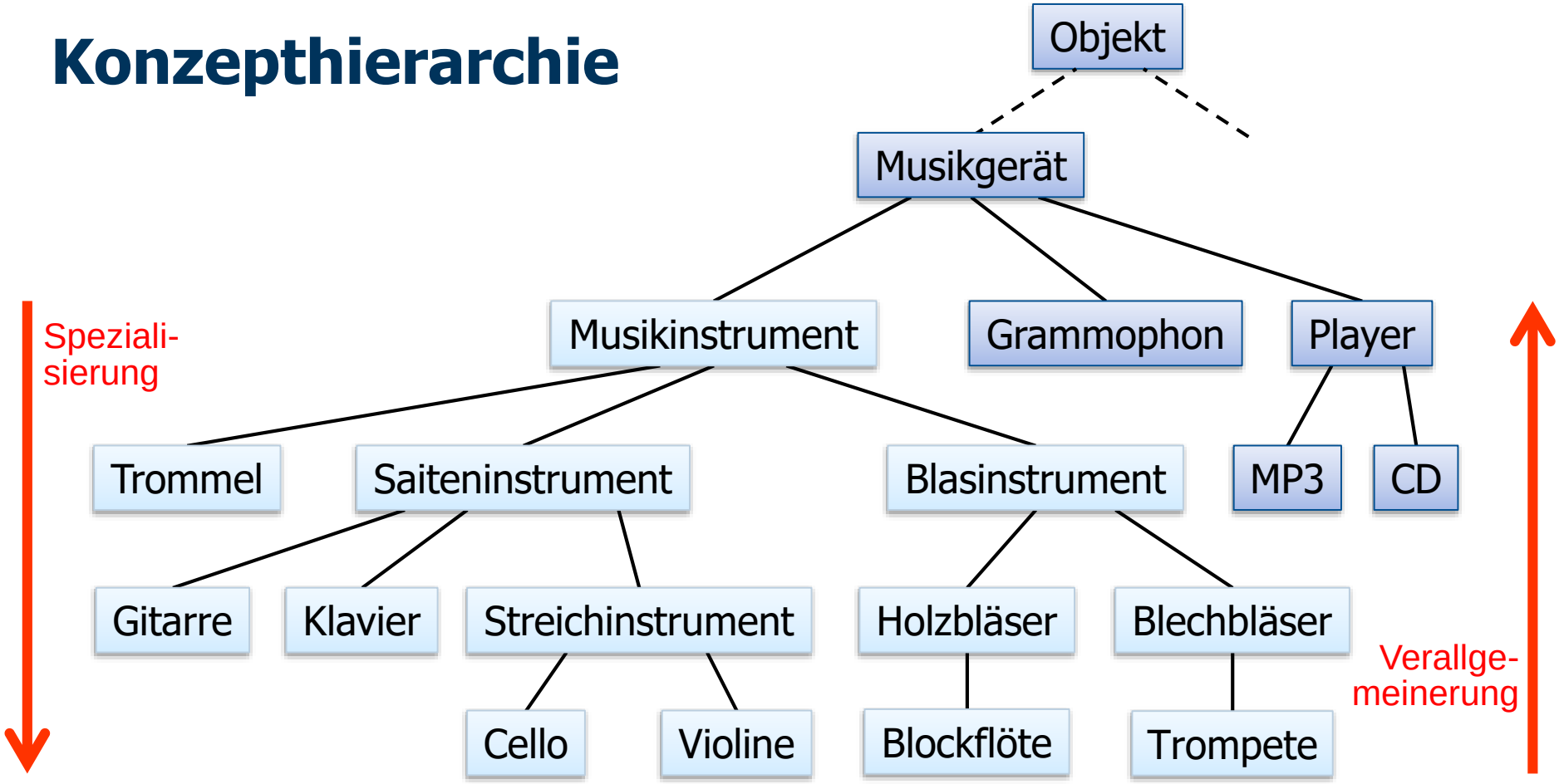
1) Entwurf der Objekttypen, dabei ähnliche Objekte zu Klassen zusammenfassen
2) Herausfaktorisierung gemeinsamer Aspekte verschiedener Klassen ⇒ Konzepthierarchie festlegen
3) Festlegung einzelner Dienstleistungen als Methoden
4) Entwurf der Objektbeziehungen
5) Feinplanung der einzelnen Methoden, Festlegung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

Konzepthierarchie (für Fantasy-Fans)



Instanzen

Konzepthierarchie



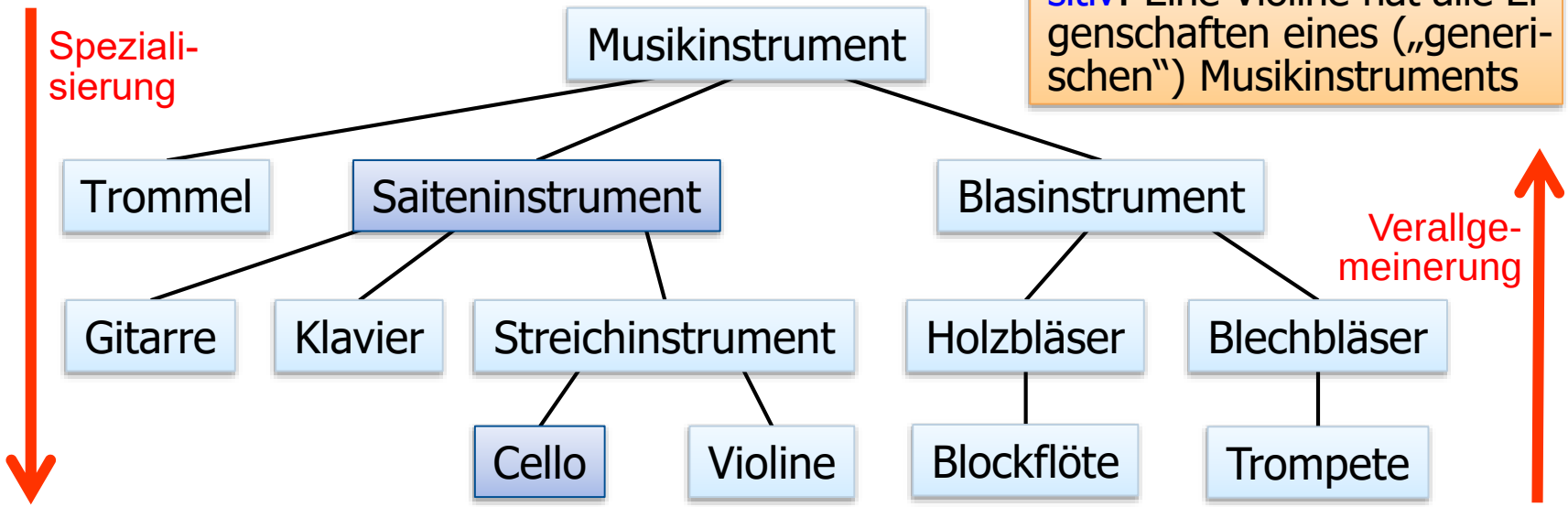
- Spezialisierung, Verallgemeinerung
  - Blockflöten sind spezielle Blasinstrumente
  - Violinen sind Musikinstrumente

Codex Manesse (fol. 423v)

Vererbung und Polymorphie

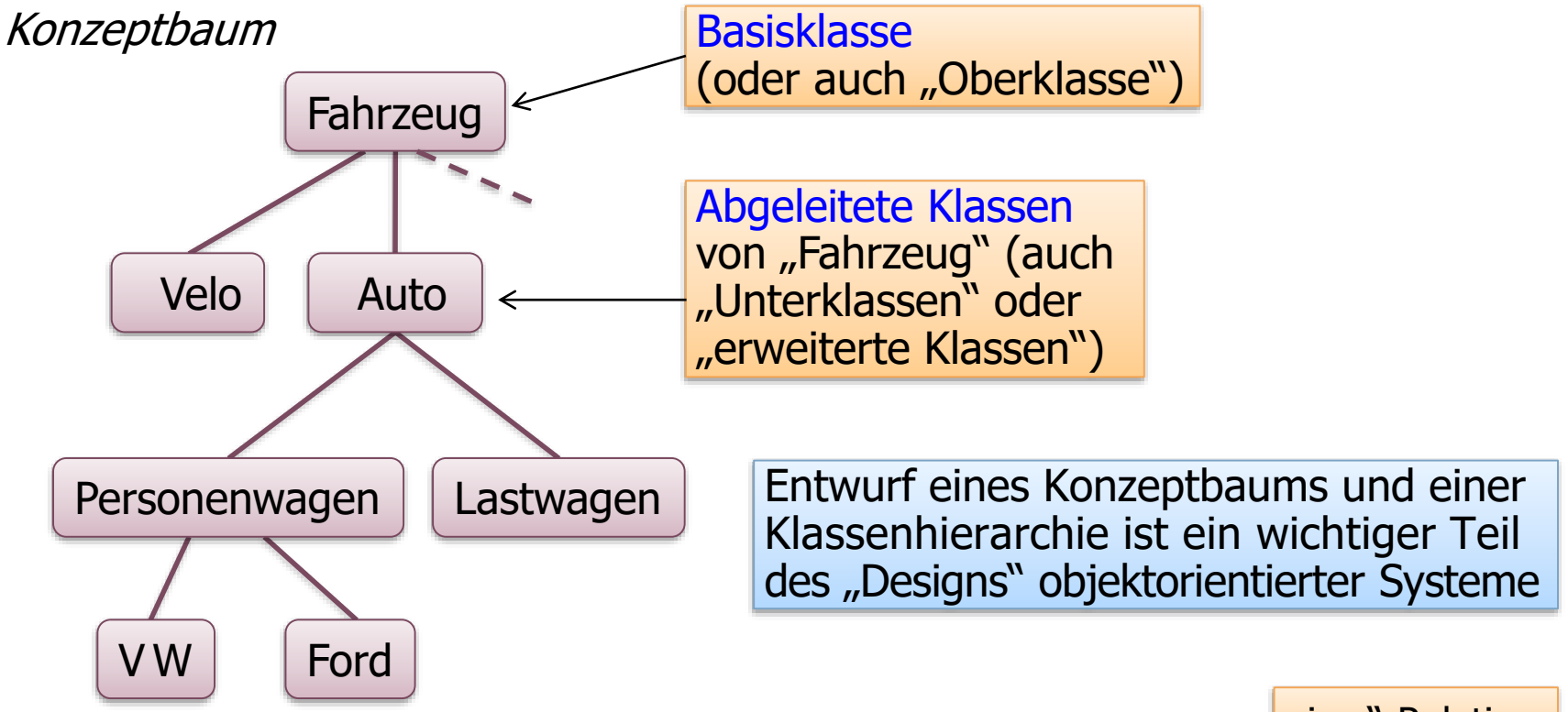
- Alle Merkmale, Eigenschaften... des umfassenderen Begriffs werden an den Unterbegriff vererbt
  - Ein Cello hat alle Eigenschaften eines allgemeinen Saiteninstruments (und noch einige weitere darüber hinaus)

Vererbungsrelation ist transitiv: Eine Violine hat alle Eigenschaften eines („generischen") Musikinstruments



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

Klassenhierarchie als Konzeptbaum



- 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!

Historische Notiz

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 und 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."

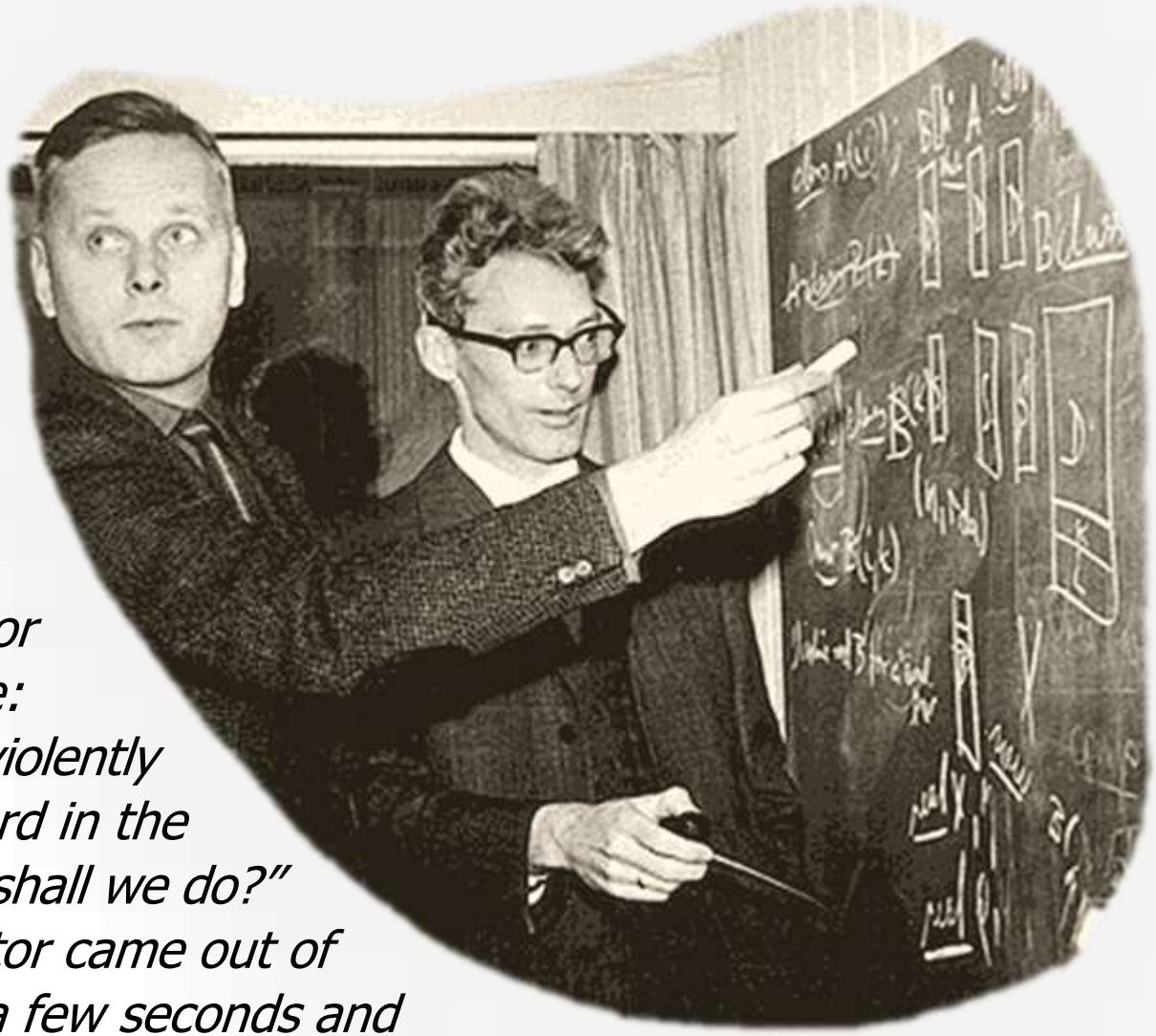


Bild: http://heim.ifi.uio.no/~gisle/in_memoriam_kristen/images/kristen_simulatavlen.html

Nygaard und Dahl: Objektorientierung

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.

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 erster Professor für Informatik in Norwegen, er erhielt im Jahr 2000 die Ehrendoktorwürde der ETH Zürich und zusammen mit Kristen Nygaard im Jahr 2001 den Turing Award.

Bild: https://titan.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.

Ende der historische Notiz

Abgeleitete Klassen, Redefinition

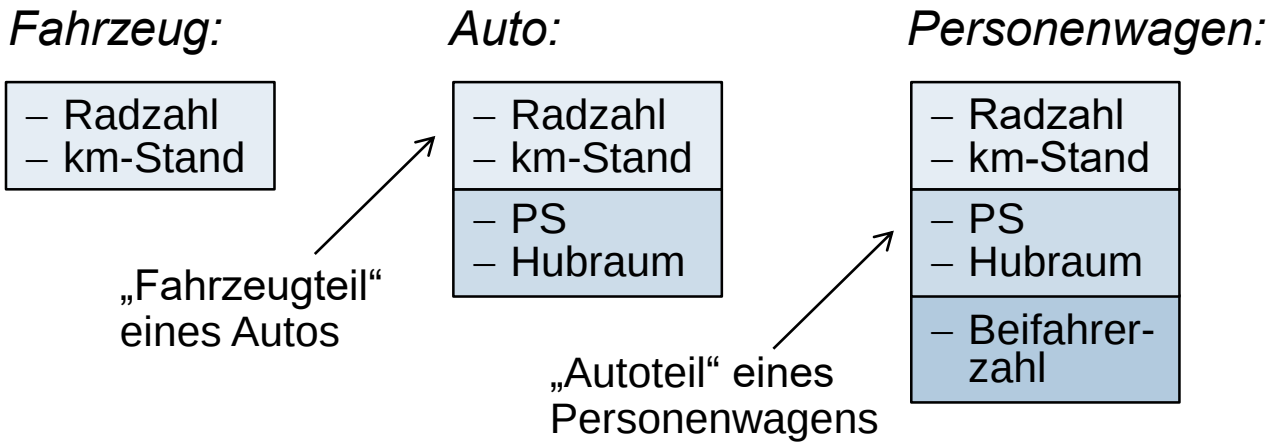
- Eine abgeleitete Klasse besitzt automatisch alle Eigenschaften der zugehörigen Basisklassen
  - Konkret: besitzt alle Attribute und Methoden der Basisklassen

- 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

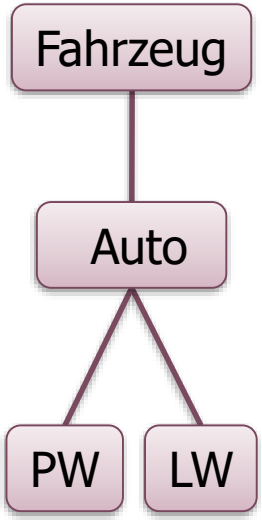
Ein Beispiel in Java

```
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!

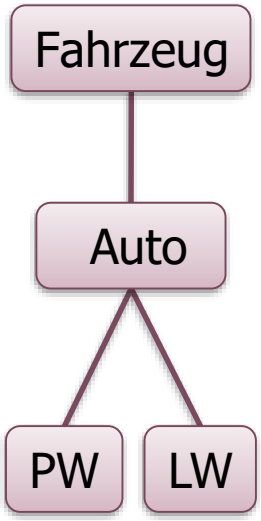
Idee: Gemeinsame Aspekte *herausfaktorisieren* und in eine übergeordnete Klasse einbringen

Personenwagen:

- Radzahl
- km-Stand
- PS
- Hubraum
- Beifahrer-zahl

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

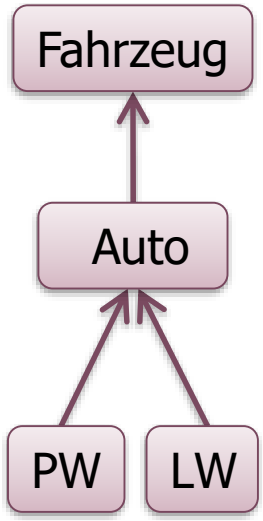


- Die Umkehrung gilt jedoch nicht!
  - d.h. `a = f;` ist verboten, da „unmöglich"!
  - Variable a kann Autoobjekte speichern
  - Ein Fahrzeug ist aber kein Auto (jedenfalls nicht immer)!

- „Gleichnis" zur Zuweisungskompatibilität: Auf einem Parkplatz für Fahrzeuge dürfen Autos, Personenwagen, Velos... abgestellt werden; auf einem Parkplatz für Velos jedoch keine beliebige 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?

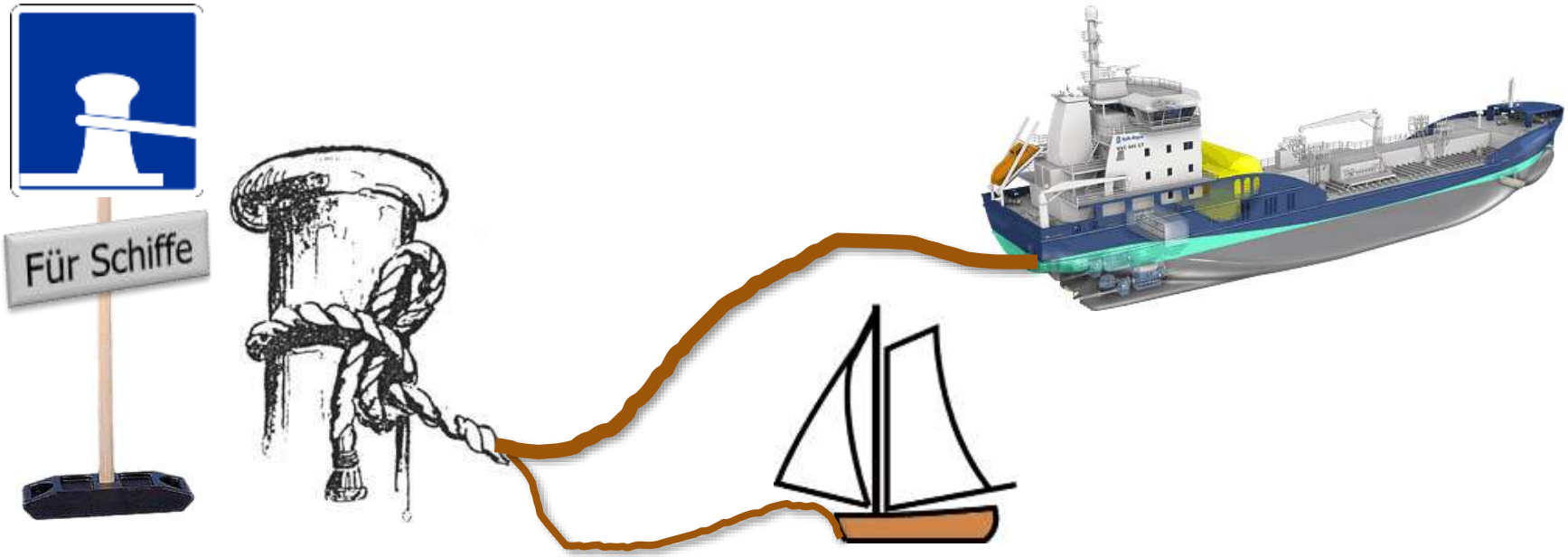
(Ca. 1999, University of California, Berkeley)

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

Zuweisungskompatibilität (2)

- Merke also: *Eine Variable vom Typ „Basisklasse" darf auch ein Objekt einer abgeleiteten Klasse enthalten*

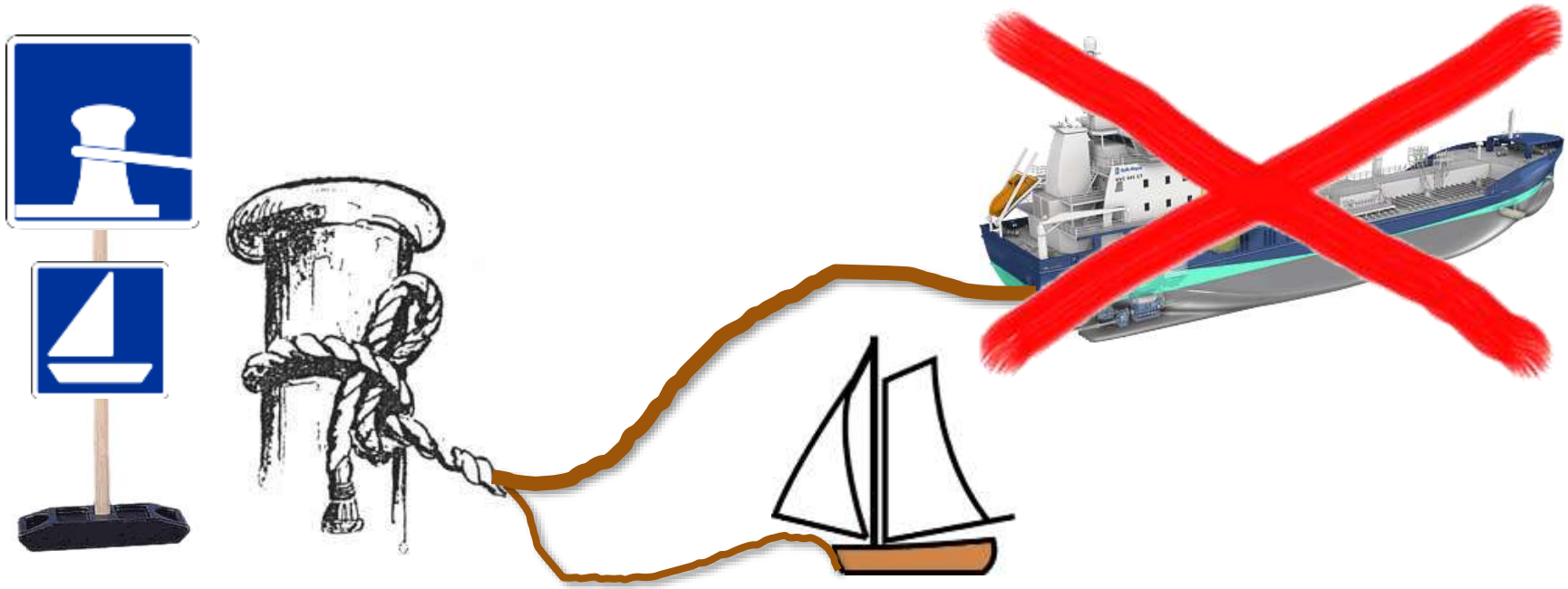
- Man nennt diese Eigenschaft auch Polymorphie, da eine Referenz auf Objekte *verschiedenen Typs* zeigen kann
  (bzw. eine Variable Werte unterschiedlichen Typs haben kann)



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)

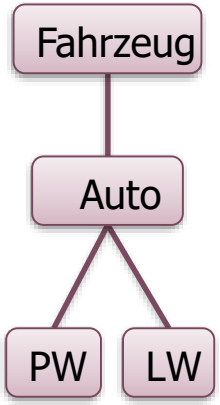


Zuweisungskompatibilität (4)

- 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!



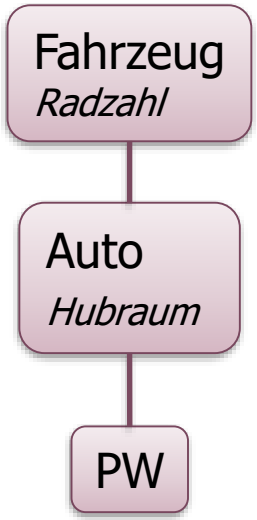
Ein Java-Beispiel zur Zuweisungskompatibilität

```
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.Radzahl = 4;

a = p;
f = p;
f = a;
/* a = f; */
/* ERROR: Incompatible types */

a.Hubraum = 1100;
f = a;
System.out.println(f.Radzahl);
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 Radzahl zugewiesen; Hubraum ist aber *über f* nicht zugreifbar; f hat nicht die dafür notwendige Qualifikation



Typkonversion („type cast“)

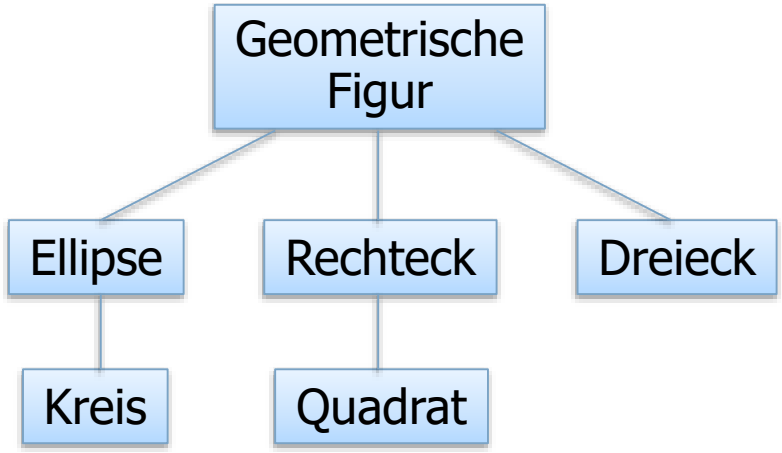
- Aus gutem Grund ist f.Hubraum verboten: Auf f könnte ja zufällig ein Fahrrad (ohne Hubraum!) „parken“!
- Durch explizite Typkonversion kommt man aber notfalls (wenn man überzeugt ist, dass konkret das Fahrzeug ein Auto ist) auch über f an den Hubraum des Auto-Objektes:
  `System.out.println(((Auto)f).Hubraum);`
- 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

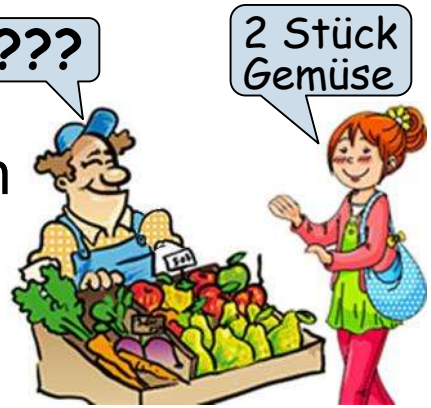
Beispiel:
Methode „Flächenwert“ ist für alle Objekttypen realisiert, aber je nach Typ auf unterschiedliche Weise!



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

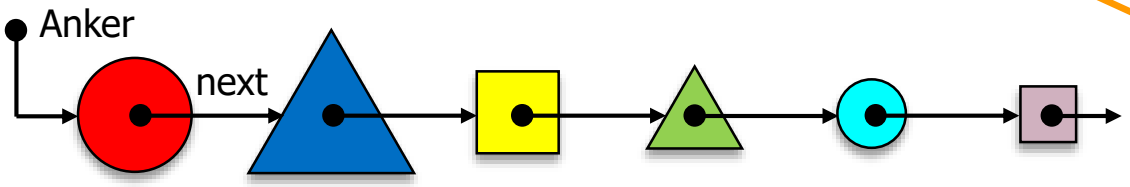
Abstrakte Methoden und abstrakte Klassen

- Abstrakte Methoden „existieren" nur dem Namen nach
  - Sie werden in abgeleiteten Klassen jeweils spezifisch implementiert
- Klassen mit abstrakten Methoden müssen selbst als „abstract" deklariert werden; von abstrakten Klassen kann man keine Objekte mit new erzeugen



Beispiel

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



Diese Methode muss für jede von GeoFigur abgeleitete Klasse mit sinnvoller Semantik implementiert werden!

- Abstrakte Klassen dürfen aber auch nicht-abstrakte Methoden haben

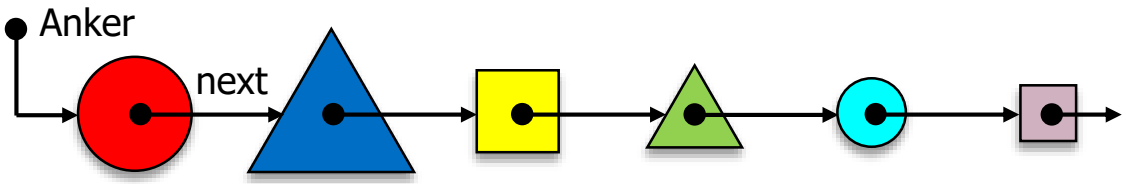
Eine polymorphe Liste

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

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

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

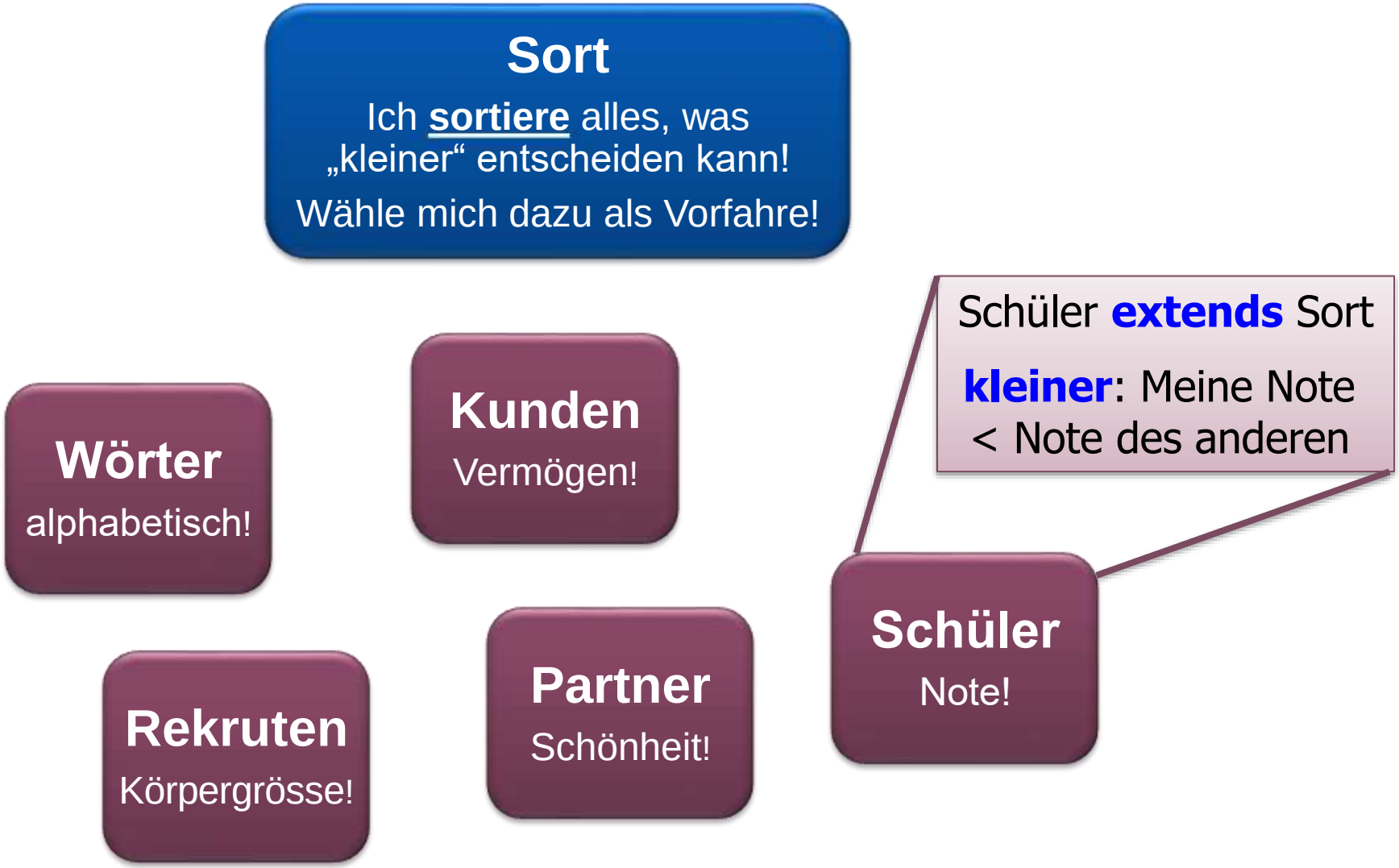
```
for (GeoFigur g = Anker; g != null; g = g.next)
  f = f + g.flaechenwert();
```



Verarbeitung polymorpher Objekte

- Die Verarbeitung der einzelnen Objekte kann unabhängig von ihrem eigentlichen Typ geschehen
  - In der Praxis wird eine solche „Verarbeitung" i.Allg. wesentlich komplexer sein, als in obigem Beispiel angedeutet
- Es können, *ohne den Verarbeitungsalgorithmus anzupassen*, neue Objekttypen als Unterklassen von „GeoFigur" eingeführt werden (z.B. „Quadrat")
  - Nur *hinzufügen*, aber (im Idealfall) *nichts verändern,* vermindert den Wartungsaufwand beträchtlich!
- Late binding: Es wird zur Laufzeit berechnet, welche konkrete Methode jeweils „angesprungen" wird; dies steht zur Übersetzungszeit (→ „early binding") noch nicht fest!

Generische Methoden und abstrakte Klassen



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;
        }
  }
}
```

Achtung: Es wird absteigend sortiert!

Sort ist eine abstrakte Klasse; von solchen können keine Objekte mit „new" erzeugt werden, sie dienen nur dazu, hiervon abgeleitete Klassen zu definieren

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

- Wir fordern, dass die zu sortierenden Objekte vom Typ einer von Sort abgeleiteten Klasse sind
- In der abgeleiteten Klasse muss dann die Methode „kleiner" (als totale Ordnungsrelation auf den Objekten) implementiert werden

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 vieler verschiedener Datentypen verwenden
  - int, float, Brüche als rationale Zahlen, Zeichenketten...

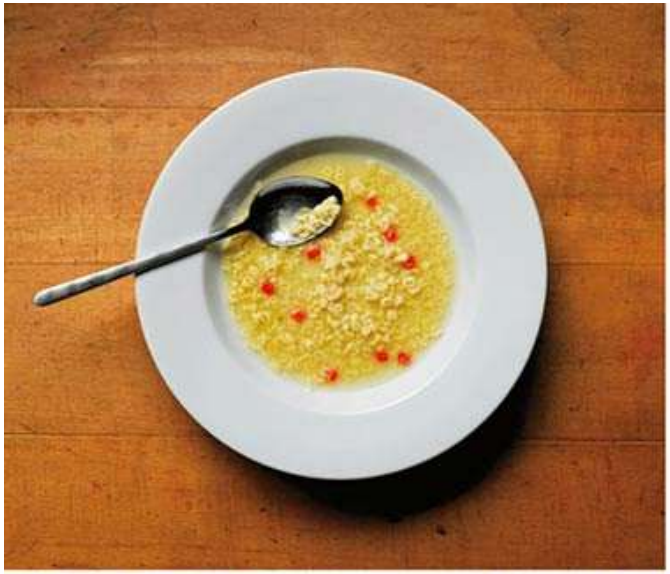
Generische Methoden und abstrakte Klassen (4)

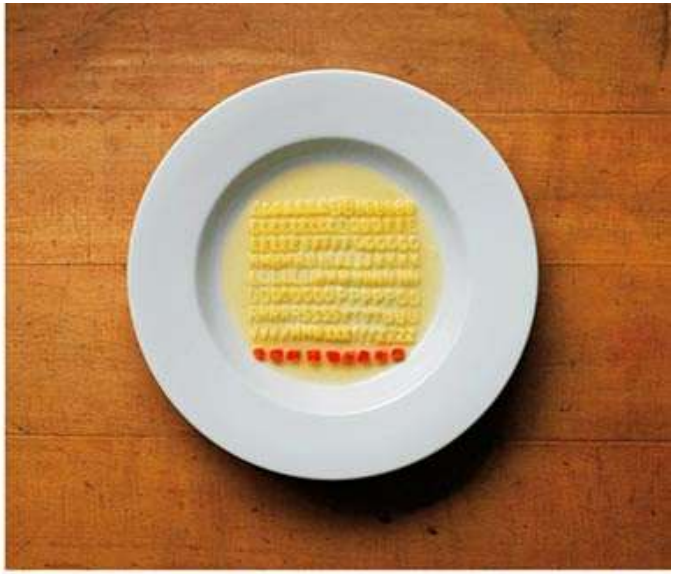
…oder um die Substantive der Vorlesungsslides der Länge nach zu sortieren:

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

https://o.quizlet.com/nW2M3sN.7lppaaSDV8.8gg_b.jpg

Anwendung der Sort-Klasse für int-Werte

Als Übung

- Es sollen hier einfache int-Werte sortiert werden, und zwar unter Benutzung der existierenden generischen Sortierroutine (ohne diese kennen zu müssen!):

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

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"

Hier wird die Relation „kleiner" definiert und implementiert

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

Anwendung der Sort-Klasse für int-Werte

Als Übung

```
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

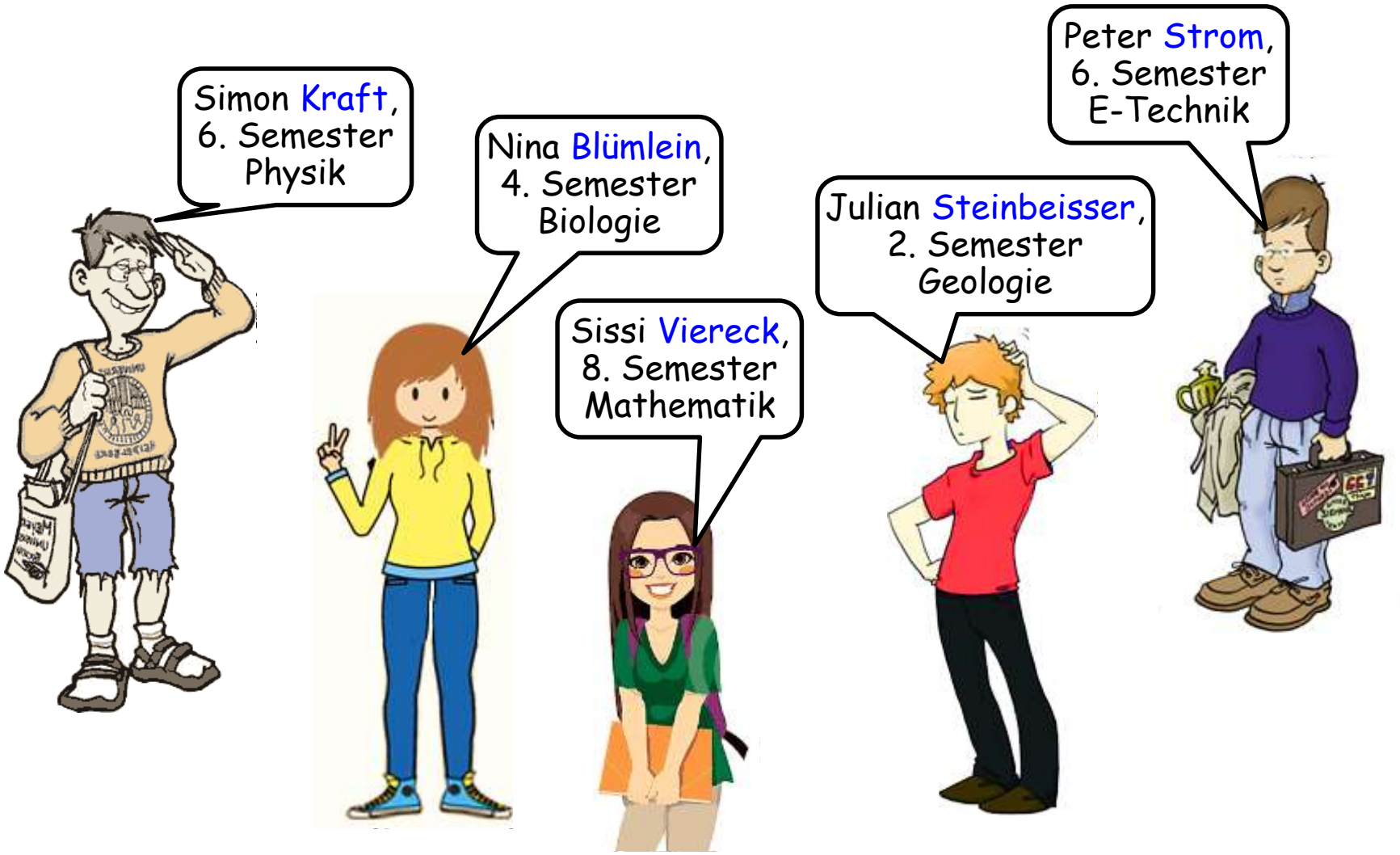
Name der Klasse, nicht einer Referenz!

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



Eine Sortieranwendung – Studi-Daten

```
class Studi extends Sort {
  int Semester;
  String Name, Fach;

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

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

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

  }
}
```

```
abstract class Sort {

  abstract boolean
    kleiner (Sort y);

  static void sortiere
        (Sort[] Tab) {
     … if … kleiner …
  }
}
```

Konstruktor

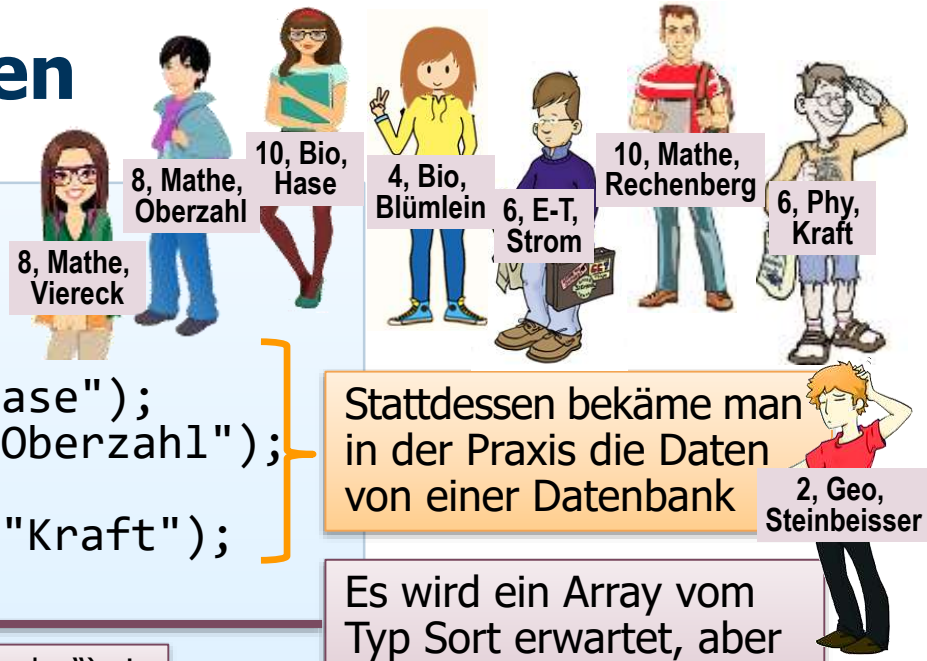
Hier darf leider nicht Studi stehen!

Typkonversion von Sort nach Studi

Hier wird „kleiner“ implementiert

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

Sortieren von Studi-Daten




```
class ...
...

  Studi[] Tab = new Studi[8];
  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);
}
```

Stattdessen bekäme man in der Praxis die 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 Sortier-kriterium „Semester-Fach-Name“)

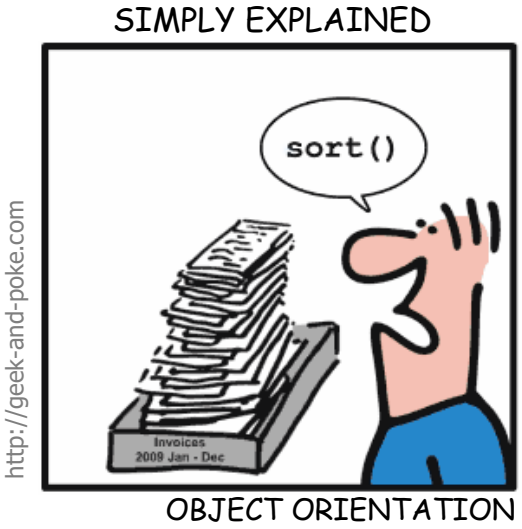
```
Rechenberg Mathe 10
Hase Bio 10
Viereck Mathe 8
Oberzahl Mathe 8
Kraft Physik 6
Strom E-Technik 6
Blümlein Bio 4
Steinbeisser Geologie 2
```

- 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

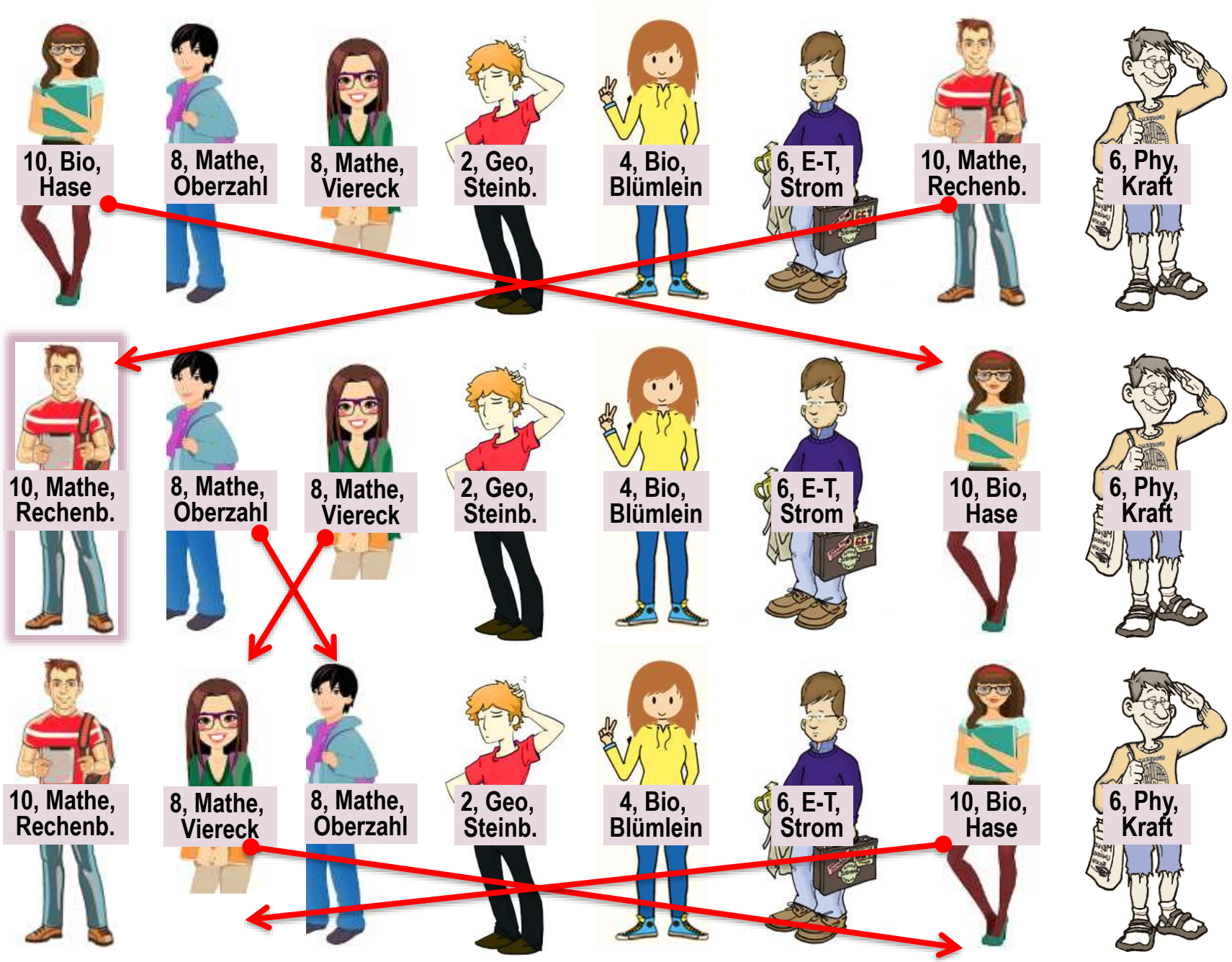
  ```
  boolean kleiner(Sort ...) ...
  ```

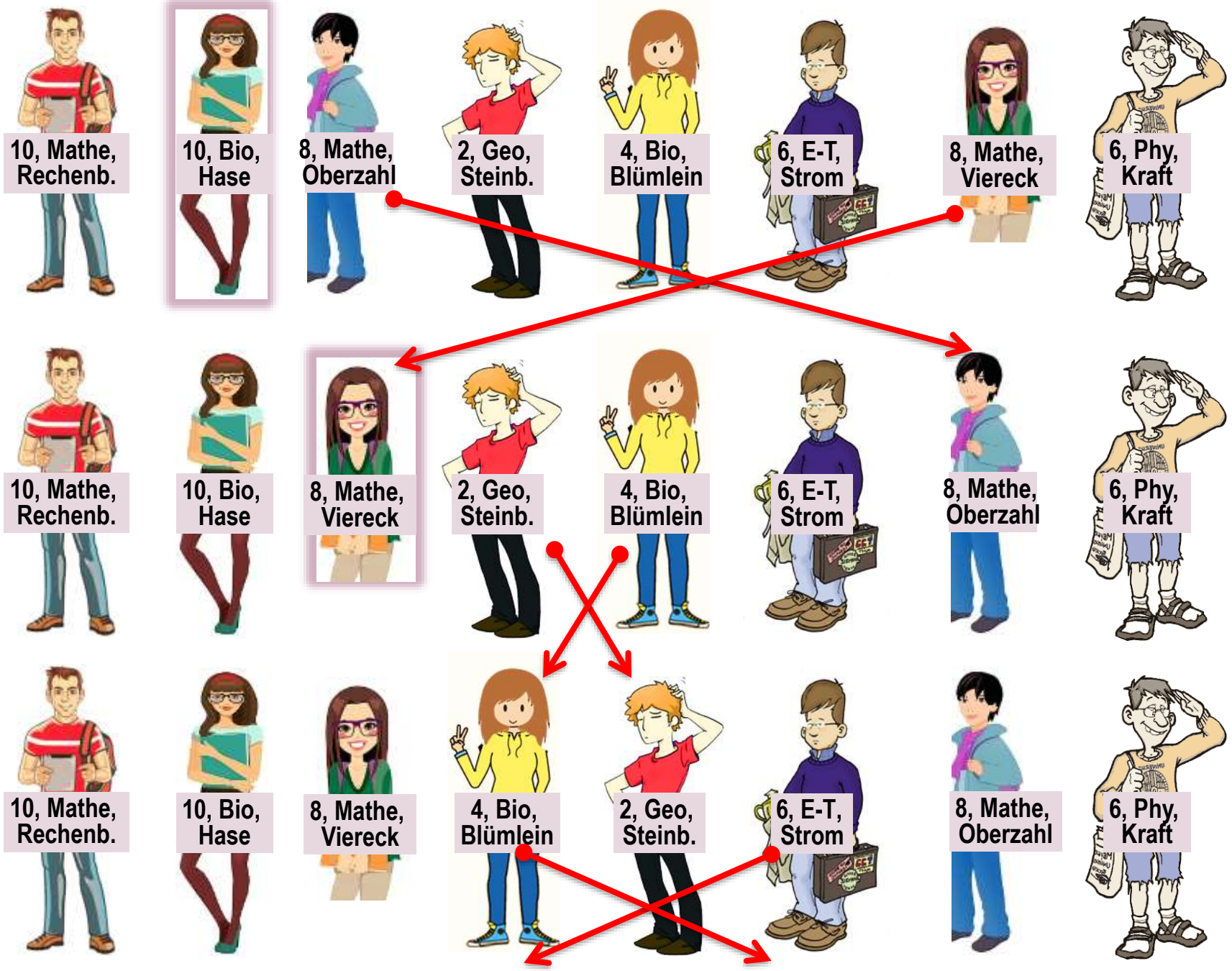


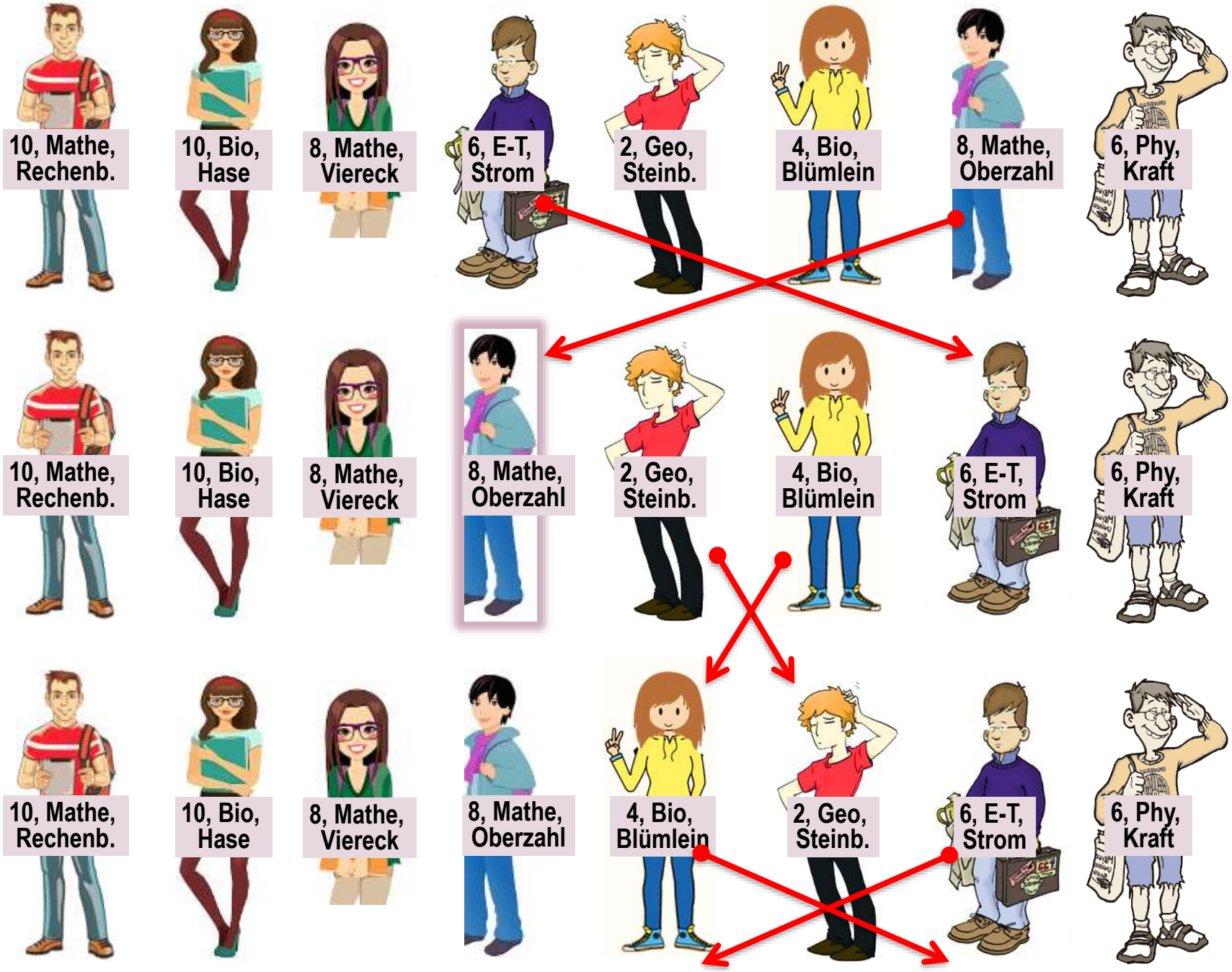
Zum Sortierverfahren („exchange sort“)

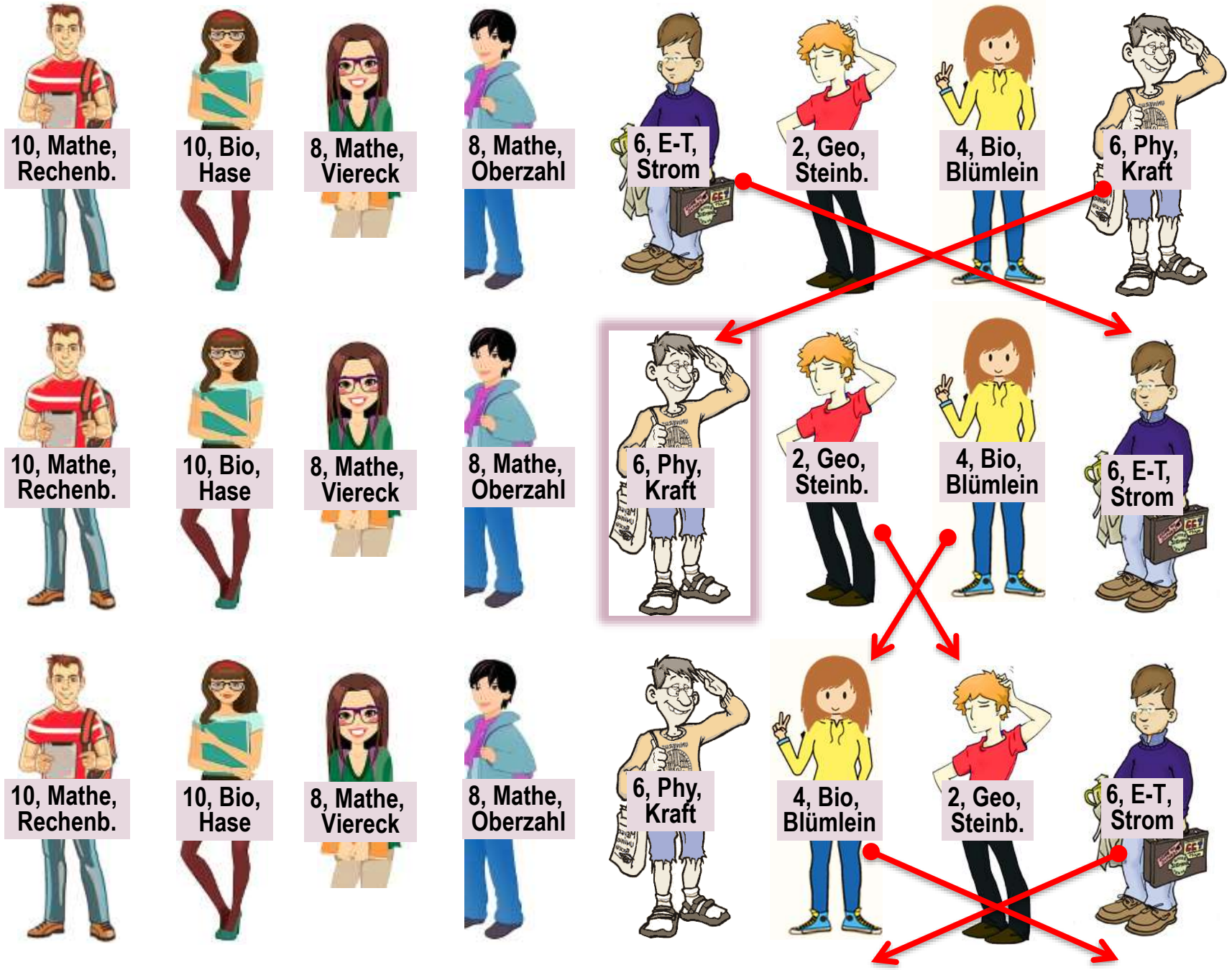
- Prinzip: Angenommen, auf den Plätzen 0, ..., i-1 stehen schon die richtigen Elemente. Auf Platz i soll nun das grösste der restlichen Elemente von den Plätzen weiter rechts (i, i+1,...) kommen.
- Immer dann, wenn die innere j-Schleife weiter rechts ein „neues grösseres“ Element (als das aktuelle auf 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 (→ „Verlobungsprinzip“: Festhalten und weitersuchen nach besseren Kandidaten).
- Hat man so schliesslich auf Platz i das grösste unter den Restelementen i, i+1,... 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.

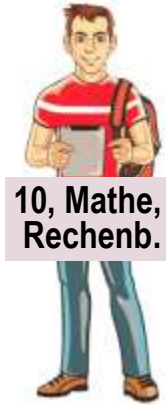
Illustration des Sortierverfahrens

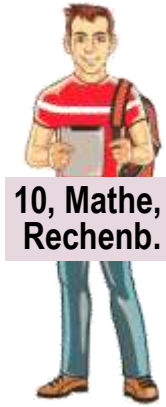



10, Mathe, Rechenb.
10, Bio, Hase
8, Mathe, Oberzahl
2, Geo, Steinb.
4, Bio, Blümlein
6, E-T, Strom
8, Mathe, Viereck
6, Phy, Kraft
10, Mathe, Rechenb.
10, Bio, Hase
8, Mathe, Viereck
2, Geo, Steinb.
4, Bio, Blümlein
6, E-T, Strom
8, Mathe, Oberzahl
6, Phy, Kraft
10, Mathe, Rechenb.
10, Bio, Hase
8, Mathe, Viereck
4, Bio, Blümlein
2, Geo, Steinb.
6, E-T, Strom
8, Mathe, Oberzahl
6, Phy, Kraft


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





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();}
}
```

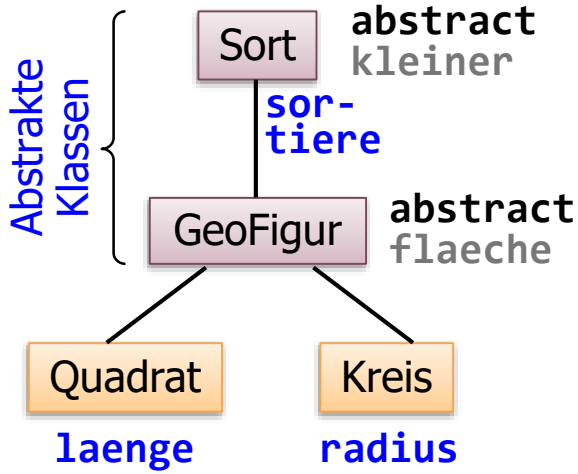
```
class Quadrat extends GeoFigur {
  float laenge;
  public float flaeche() {
    return (laenge*laenge);
  }
  Quadrat() {
    laenge = KbdInput.readInt("Länge? ");
  }
}
```

Konstruktor

Objekt in Gründung erfragt interaktive seine Grösse

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

Analog zu Quadrat



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

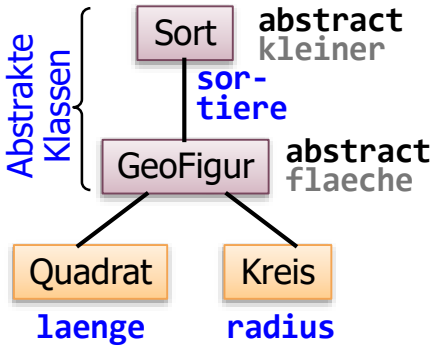
```
public static void main(String[] args) {

  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();
}
```

Poly-morphie

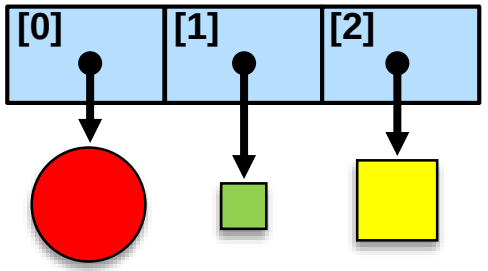
GeoFigur erbt die Methode „sortiere"

late binding

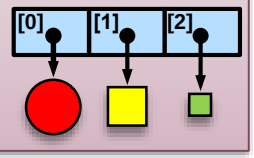
Hier wird jeweils die „richtige" Methode für die Fläche verwendet



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

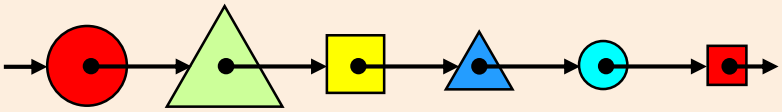



```
Kreis (k) oder Quadrat (q)? k
Radius?  5
Kreis (k) oder Quadrat (q)? q
Länge?  4
Kreis (k) oder Quadrat (q)? q
Länge?  7

78.5397    49    16
```



Resümee des Kapitels

- Objektorientiertes Programmieren: Klassenkonzept
  - Konzepthierarchie, is-a-Relation, abgeleitete Klassen, Vererbung
  - 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