Herzlich willkommen!

Dozent: Dipl.-Ing. Jürgen Wemheuer

Mail: wemheuer@ewla.de
Online: http://cpp.ewla.de/

Disclaimer

- Diese Vorlesungs-/Unterrichtsfolien wurden durch den Dozenten ausschließlich für die Gestaltung seines Unterrichts / seiner Vorlesung zusammengestellt bzw. verfasst und sind nicht als Referenz einer Programmiersprache gedacht.
- Das Skript verwendet teilweise Materialien meiner geschätzten Fachkollegen Prof. Dr. Dietrich Kuhn († 2010) und Prof. Dr. Stefan Enderle der Naturwissen-schaftlich-Technischen Akademie Dr. Grübler (nta) in Isny/Allgäu. Vielen Dank!
- Dem Vorlesungsskript mangelt es an jeglichem Kontext.
  Dieser ist vielmehr der bestimmende Lehrinhalt in den Vorlesungen.
- Nicht alle Inhalte des Vorlesungsskripts sind prüfungsrelevant.
- Nicht alle prüfungsrelevanten Fakten sind im Vorlesungsskript enthalten.
- Ausschlaggebend für Prüfungen sind deshalb allein die im Unterricht bzw. in den Übungen und/oder Projektbeispielen vorgebrachten Inhalte.
- Aktuelle Änderungen des Vorlesungsskripts sind jederzeit vorbehalten.
- Mit allen auftretenden Fragen zum Fachgebiet und dem Vorlesungsskript sollten sich die SchülerInnen und StudentInnen stets an den Dozenten wenden.
- Das Vorlesungsskript wurde mit bestem Wissen und Gewissen und sorgfältig erarbeitet, jedoch können Irrtümer und Fehler nicht ausgeschlossen werden.
- Jegliche Haftung und Gewährleistung ist ausgeschlossen.

Inhalt:

UML-Klassendiagramme

Klassenbeziehungen und Vererbungen

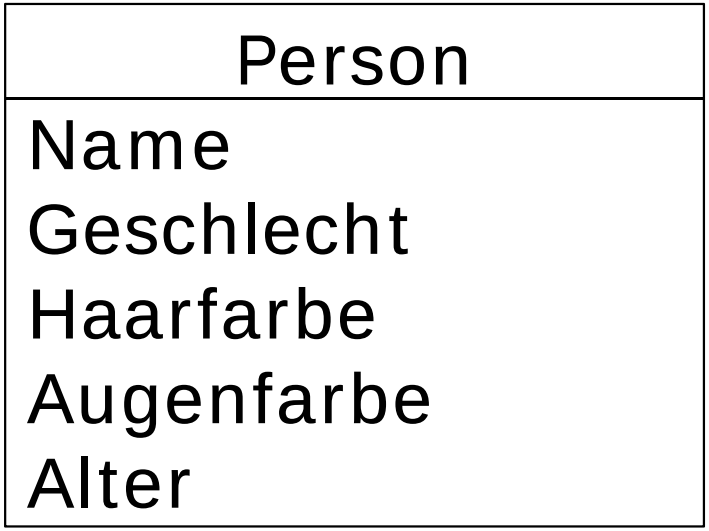
UML-Klassendiagramm

- **Unified Modeling Language**
  (Vereinheitlichte Modellierungssprache)
- ***Standardisierte,*** grafische Modellierungs-sprache zur Spezifikation, Konstruktion und Dokumentation von Software(-Teilen) (und anderen Systemen)

- Einfachster Fall: nur Klassenname

UML-Klassendiagramm: Attribute

- „Gedächtnis“ eines Objekts
- Sammlung von Daten, Eigenschaften
- Alle Attribute-Werte zusammen: ***Zustand***
- Attribute, die z.B. in einer Datenbank gespeichert werden, um das Objekt später zu rekonstruieren, heißen ***persistent***.

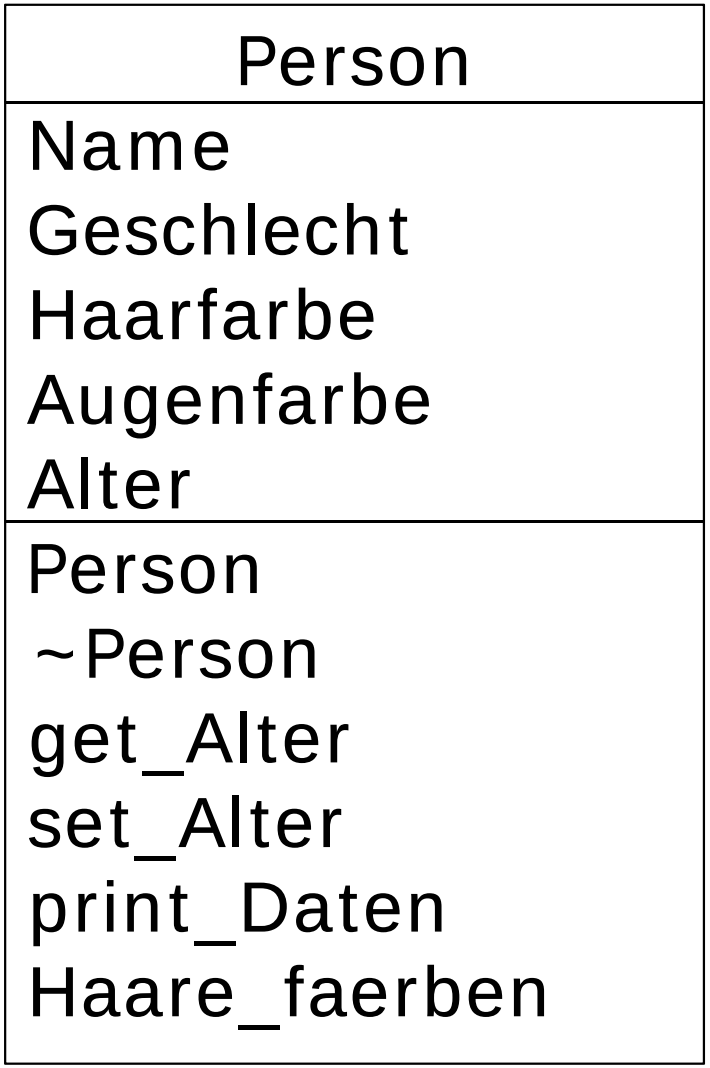


UML-Klassendiagramm: Methoden

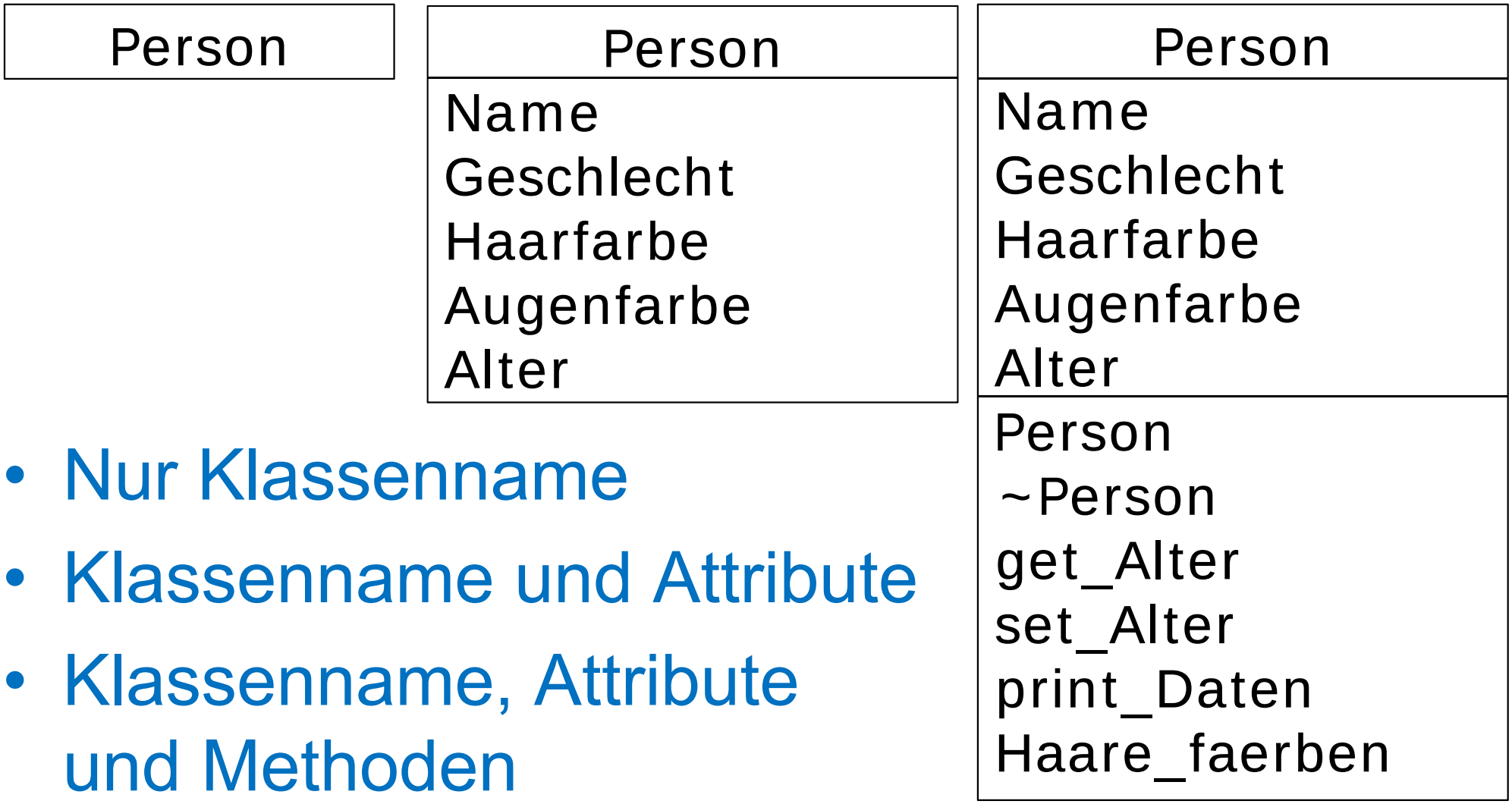
- „Fähigkeiten“ eines Objekts
- Unterscheidungsmöglichkeit:
  - Konstruktoren und Destruktoren
  - Speichern und Laden von Objektzuständen
  - Einzelne Attribute oder Gesamtzustand ändern
  - Einzelne Attribute oder Gesamtzustand auslesen
  - Berechnung ausführen, basierend auf aktuellem Zustand

UML-Klassendiagramm: Methoden

- Beispiel:



UML-Klassendiagramm: Detailgrade



- Nur Klassenname
- Klassenname und Attribute
- Klassenname, Attribute und Methoden

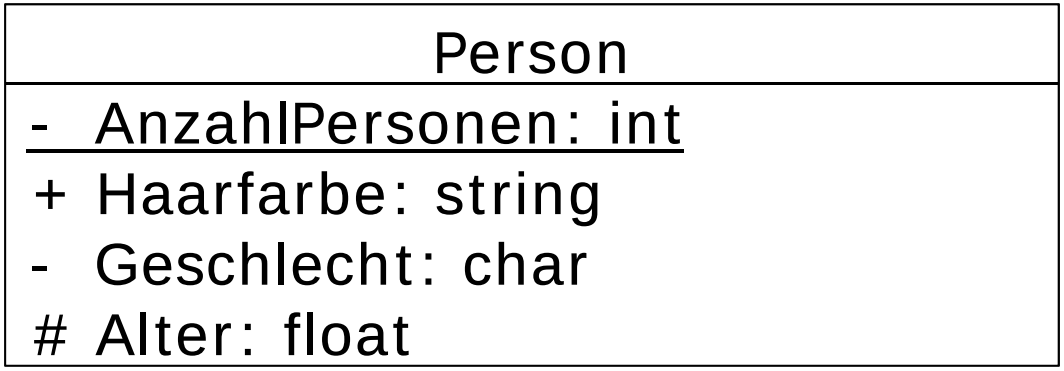
Erweiterungen (1)

- Sichtbarkeit („Scope“) der Attribute:
  Angaben durch Zeichen vor dem Attributnamen:
  - + = public (systemweiter Zugriff)
  - - = private (Zugriff nur durch Instanzen der Klasse)
  - # = protected (Instanzen der Klasse und Unterklassen)
  - <u>unterstrichen</u> = class-scope (static, „Klassenattribut“)

Person
- <u>AnzahlPersonen</u>
+ Haarfarbe
- Geschlecht
Alter

Erweiterungen (2)

- Typen der Attribute:
  Angabe des Datentyps nach „:“ hinter dem Attributnamen



- Datentyp kann sein:
  - elementarer Basisdatentyp (int, float etc.)
  - eigener Datentyp (struct, enum, typedef)
  - andere Klasse (siehe „Aggregation“)

Erweiterungen (3)

- Initialisierung der Attribute:
  Angabe der Werte nach „=“ hinter dem Typ
  - Natürlich nur, wenn nötig und sinnvoll
  - Erst beim Instanziieren einer Klasse!
    (Anlegen eines Objekts dieser Klasse, im Konstruktor)
  - Nur bei statischen Klassenattributen ist sofortige Initialisierung (ohne Objektinstanz) möglich

Person
- AnzahlPersonen: int = 0
+ Haarfarbe: string
- Geschlecht: char = ‘m‘
Alter: float = 0.0

Erweiterungen (4)

- Sichtbarkeit der Methoden:
  Angabe von + , - , # , _ wie bei den Attributen

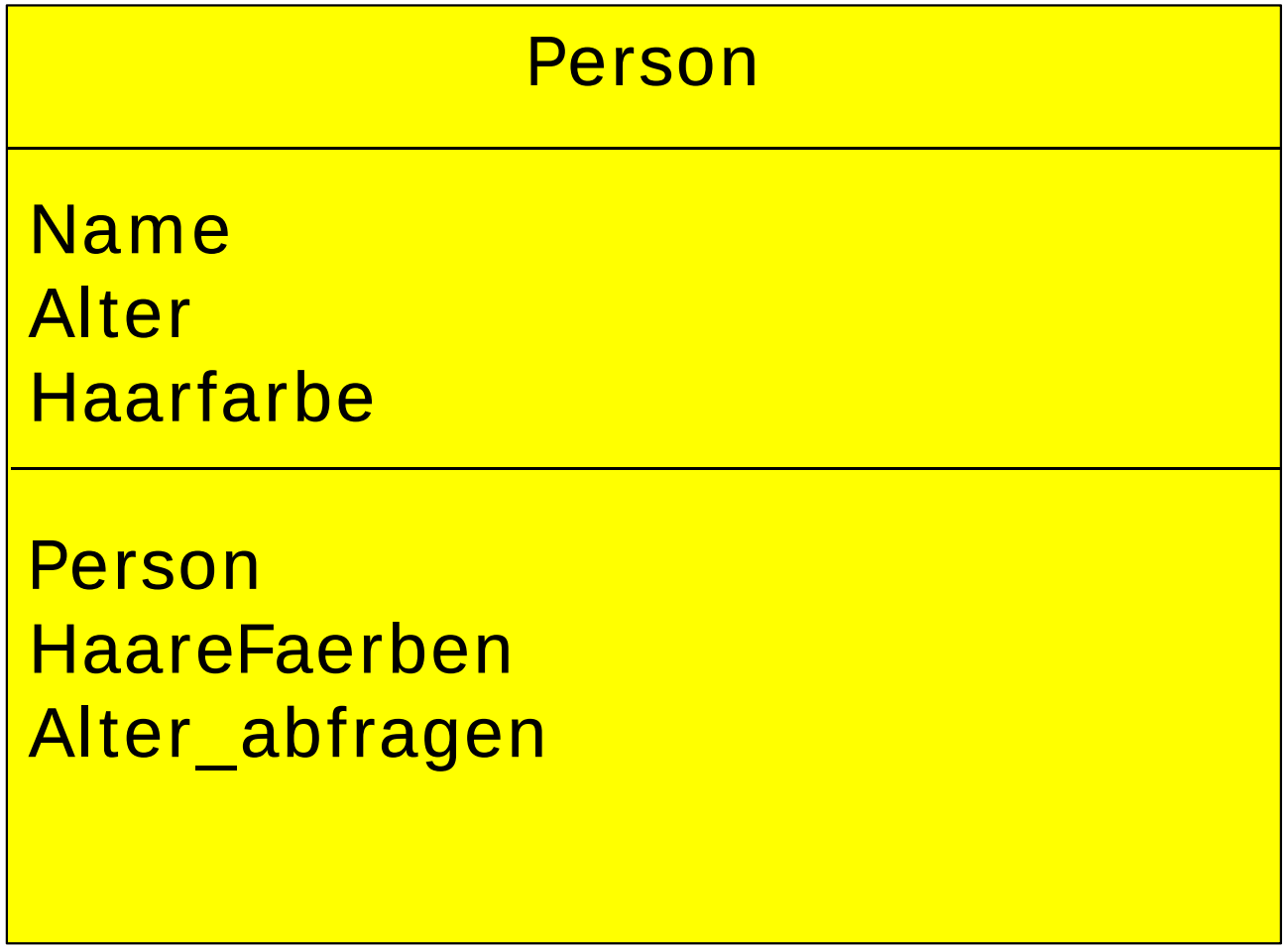
Person
- AnzahlPersonen: int = 0 + Haarfarbe: string - Geschlecht: char = ‘m‘ # Alter: float = 0.0
+ Person + Haare_faerben + get_Alter + set_Alter + print_Daten # ~Person

Erweiterungen (5)

- Typen der Übergabeparameter und Rückgabewerte der Methoden:
  Angabe Name : Typ wie bei den Attributen

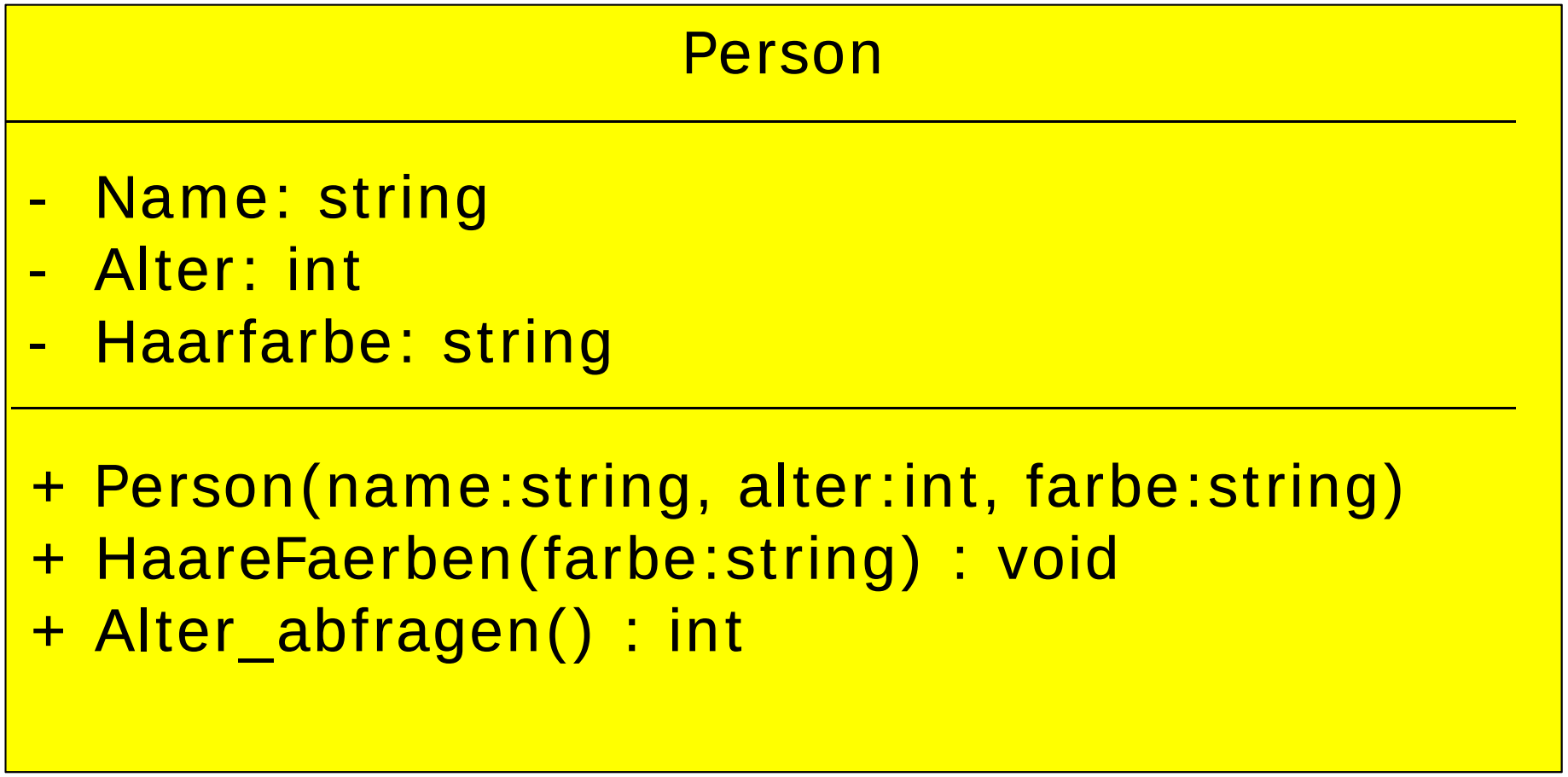
Person
- AnzahlPersonen: int = 0 + Haarfarbe: string - Geschlecht: char = ‘m‘ # Alter: float = 0.0
+ Person(F:string, A:float, ...) + Haare_faerben(Farbe: string) : void + get_Alter(void) : float + set_Alter(A:float) : void + print_Daten() : void # ~Person

Übung:



Ergänzen Sie Angaben zu Sichtbarkeit (Scope) und Datentyp

Übung:



Setzen Sie das UML-Klassendiagramm in die Sprache C++ um

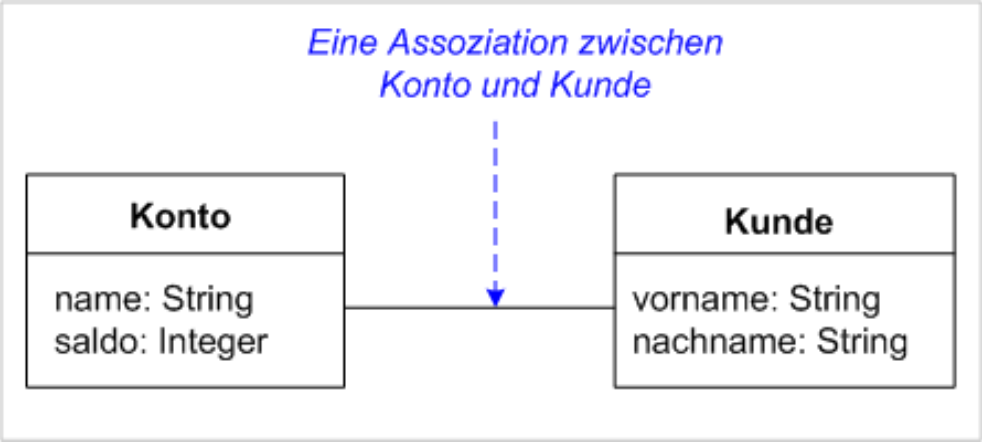
Übung:

Person
-Name: string -Alter: int -Haarfarbe: string
+ Person(name:string, alter:int, farbe:string) + HaareFaerben(farbe:string) : void + Alter_abfragen() : int

```
class Person {
private:
 string Name;
 int Alter;
 string Haarfarbe;
public:
 Person(string n, int a,
        string f);
 void HaareFaerben
                  (string n);
 int Alter_abfragen();
};
```

Klassenbeziehungen

Oft haben die Objekte verschiedener Klassen irgendetwas miteinander zu tun, sie sind „assoziiert“ (und werden zunächst durch einfache Striche miteinander verbunden):



Eine einfache („binäre“) Assoziation…

„ist“- und „hat“-Beziehungen

Beispiel:

Ein Schüler ***ist*** eine Person → **Ableitung / Vererbung**:

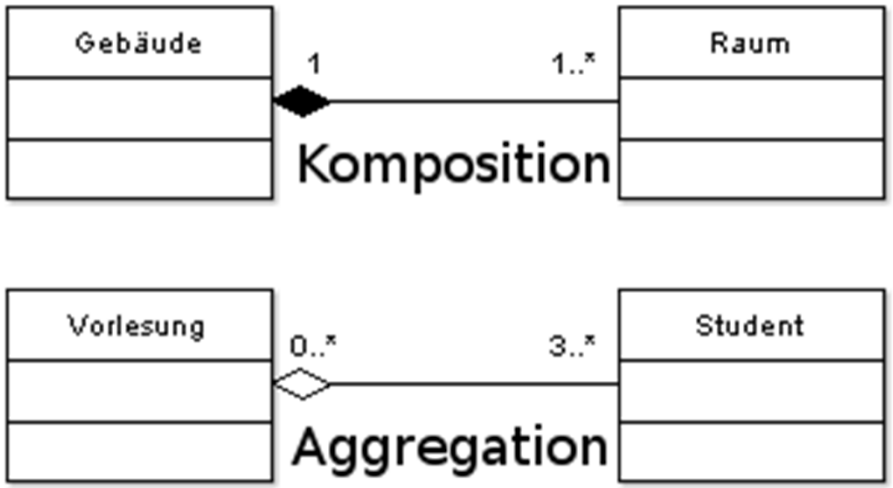
```
class Person ...
class Schueler : public Person {
…
};
```

Eine Person ***hat*** ein Geburtsdatum → **Aggregation**:

```
class Datum ...
class Person {
private:
  Datum Geburtsdatum;
...
};
```

Aggregation (hat / besteht aus)

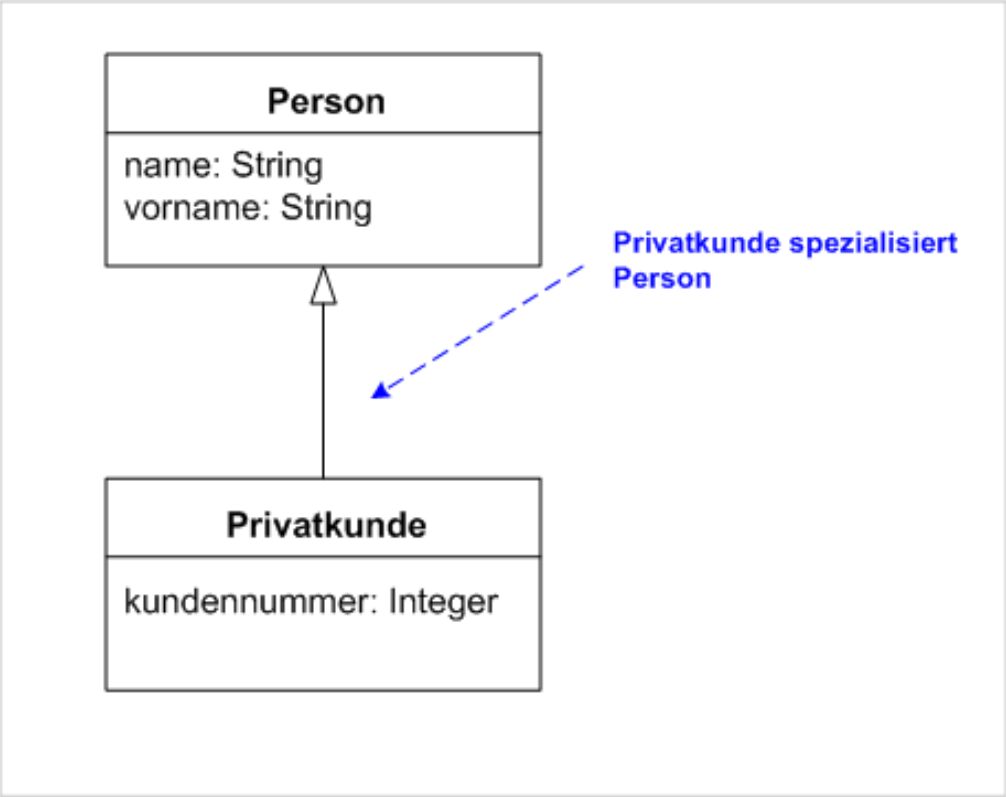
Eine recht oft vorkommende Beziehung zwischen Klassen ist die Beziehung zwischen einem ***Ganzen*** und seinen ***Teilen***.



Die Komposition ist ein Spezialfall der Aggregation und bildet den Fall ab, bei dem die *Teile* nicht ohne das *Ganze* existieren können (Existenzabhängigkeit).

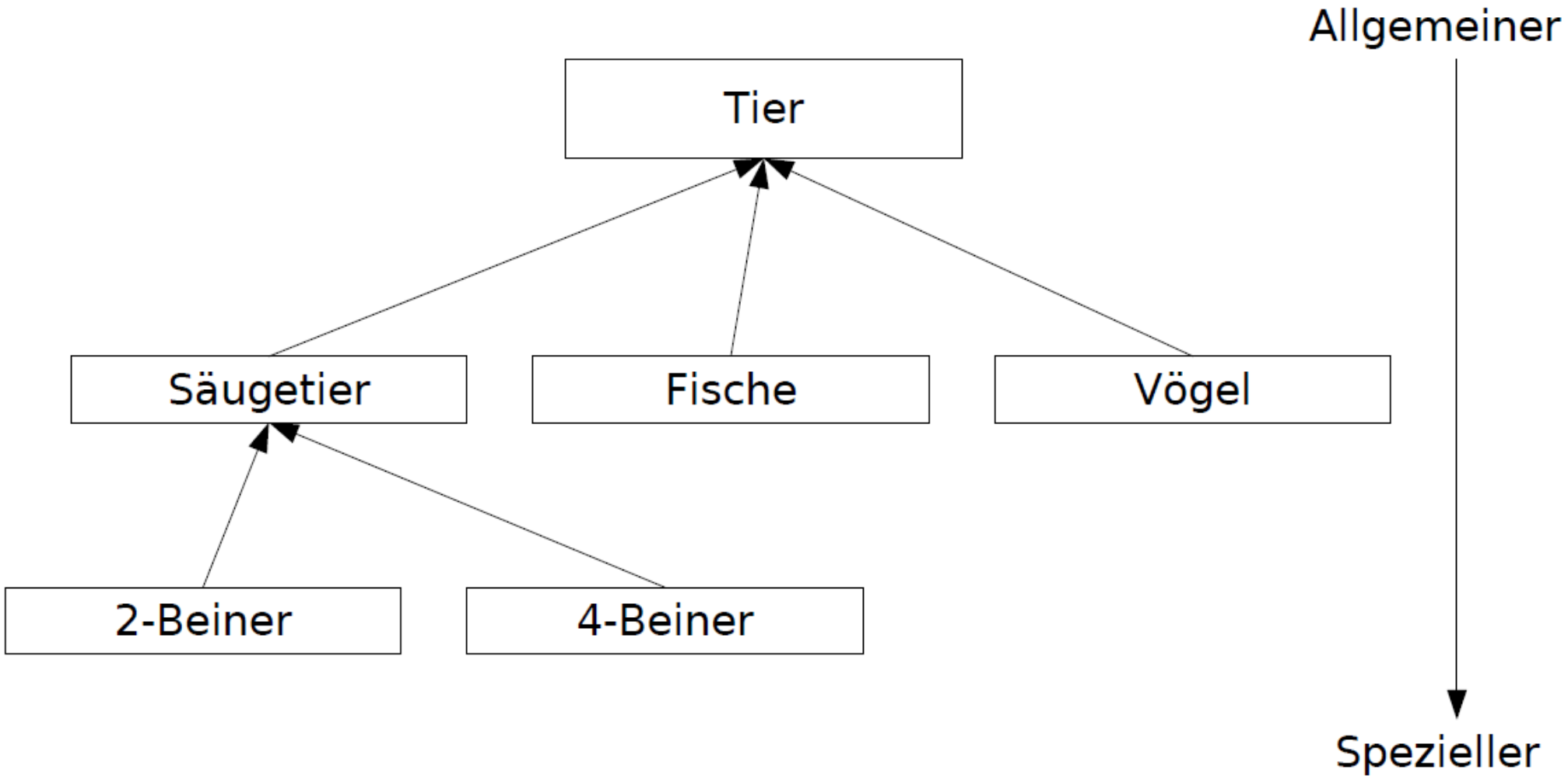
Generalisierung / Vererbung (ist)

- Je nach Betrachtungsweise spricht man bei „ist“-Beziehungen von *Generalisierung* oder *Vererbung*
- Beispiel:
  Eine Klasse Person
  ist die Generalisierung
  aller Arten von Personen
  (Schüler, Dozent,
  Kunde, Mann, Frau, …)



- Die „abgeleiteten“ Klassen
  (Schüler, Dozent,
  Kunde, Mann, Frau, …)
  „erben“ alle Eigenschaften der „Basisklasse“

Vererbung: Beispiel



Vererbung: Vorteile

Klassen vererben zu können hat zwei Vorteile:

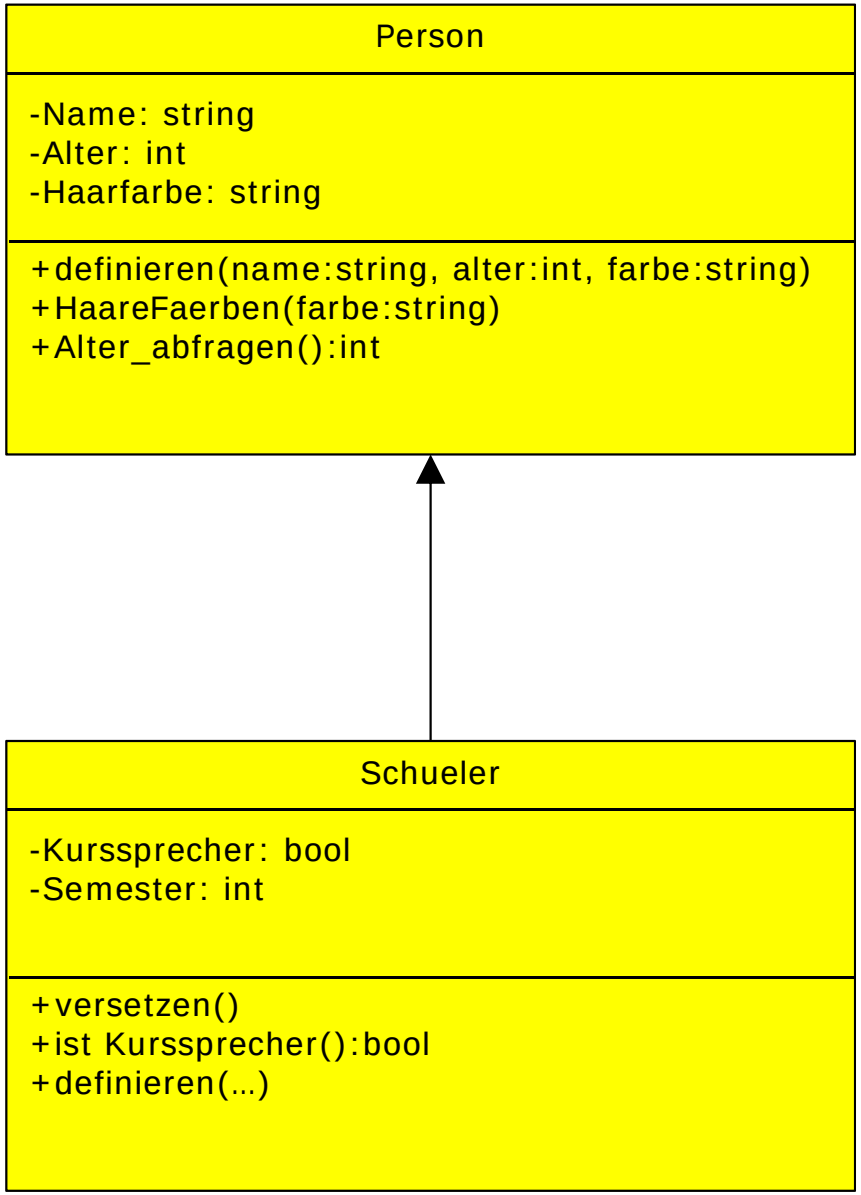
- *Datenabstraktion:*
  - Abbildung von hierarchischen Beziehungen, von allgemeineren Klassen zu spezielleren Klassen
- *Wiederverwendbarkeit:*
  - Bereits erstellte und getestete Klassen können weiterverwendet und angepasst werden. Man benötigt hierfür nur die *Schnittstelle* der Klasse!
  - Zusammenfassen von gemeinsamem Code (gemeinsamen Attributen und Methoden) in Basisklassen

Vererbung: Wiederholung

Durch Vererbung können aus bereits existierenden Klassen neue Klassen gebildet werden.

- Die *abgeleitete Klasse* „erbt“ sämtliche Attribute und Methoden von der *Basisklasse.*
- Die abgeleitete Klasse kann um weitere Attribute und Methoden erweitert werden. Sie stellt dadurch eine *Spezialisierung* der Basisklasse dar.
- Umgekehrt ist eine Basisklasse die *Generalisierung* einer oder mehrerer abgeleiteter Klassen.

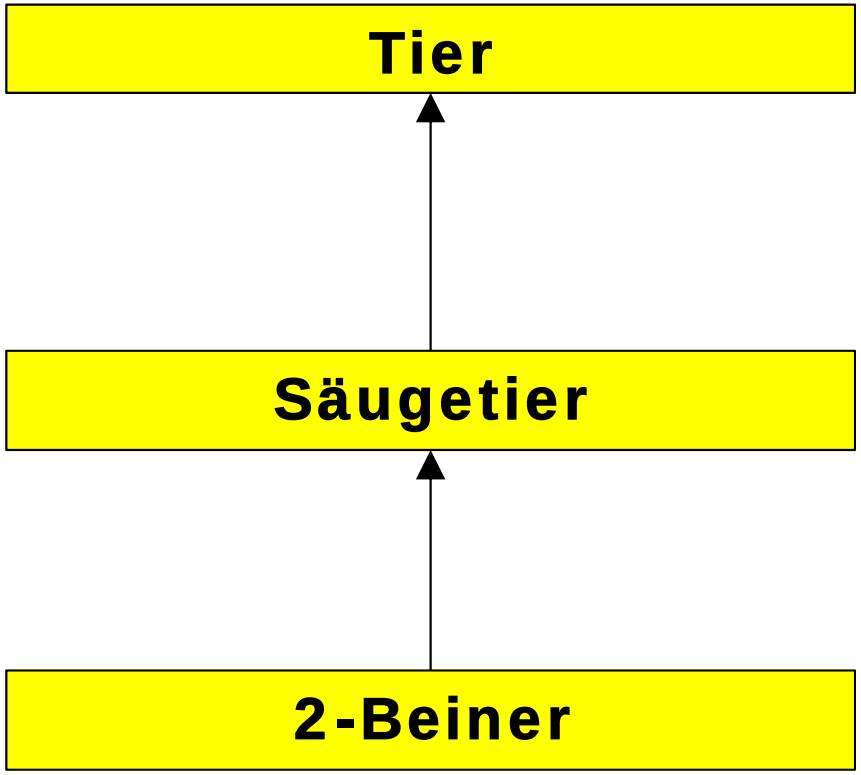
Vererbung in C++




```
class Person {
 string Name, Haarfarbe;
 int Alter;
public:
  Person();
  void definieren(string n, int a, string f);
  void HaareFaerben(string n);
  int Alter_abfragen();
};

class Schueler : public Person {
private:
// die in Person enthaltenen Attribute sind
// automatisch enthalten: Name, Haarfarbe,
// Alter werden von Person geerbt! ZUSÄTZLICH:
  bool Kurssprecher;
  int Semester;
private:
// die in Person enthaltenen Methoden werden
// geerbt, ZUSÄTZLICH ODER ÜBERSCHREIBEN:
  void versetzen();
  bool istKurssprecher();
  void definieren(…);
};
```

Mehrfach-Vererbung in C++




```
class Tier
{
...
};

class Saeuger : public Tier
{
...
};

class Zweibeiner : public Saeuger
{
...
};
```

Mehrmaliges Vererben
(Mehrfachvererbung)
ist natürlich
auch möglich!

Überschreiben („override“)

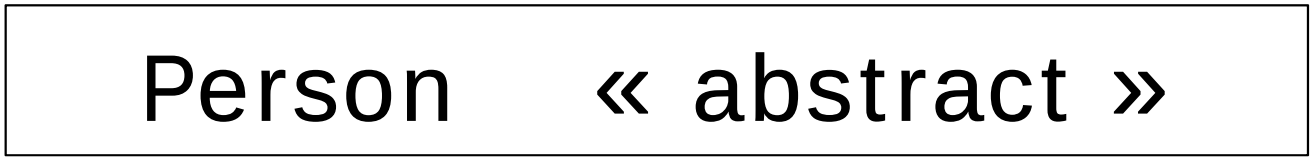
- Abgeleitete Klassen erben zwar automatisch alle Attribute und Methoden der Basisklassen
- Sie können aber auch „ein Erbe ausschlagen“ und geerbte Elemente mit eigenen, gleichnamigen Elementen überschreiben
- Der Compiler sucht
  - beginnend in der abgeleiteten Klasse
  - aufsteigend in der Klassenhierarchie

  nach passenden Elementen und nimmt das erste passende Element (und sollte spätestens in der obersten Basisklasse etwas passendes finden…)

Abstrakte Klassen

- Oft ist es sinnvoll, dass:
  - eine Basisklasse zwar den abgeleiteten Klassen viele gemeinsame Attribute und Methoden zur Verfügung stellt (Wiederverwendbarkeit, Software-Tests, Schreibarbeit…)
  - von dieser Basisklasse selbst aber keine Objekte erzeugt werden sollen

Eine solche Basisklasse nennt man „abstrakte Klasse“



class Person abstract {…}

Zugriffsmöglichkeit „protected“

`protected` (#) wirkt wie **`private`** (-) mit der Ausnahme, dass abgeleitete Klassen (und nur diese) ebenfalls auf diese Elemente zugreifen dürfen.

Wenn also abgeleitete Klassen direkt auf Attribute der Basisklasse zugreifen sollen, ist es sinnvoll, in der Basisklasse diese Attribute als **`protected`** zu deklarieren.