II.4.1a Unterklassen und Vererbung (Teil 1)

Dienstag, 21. November 2017 09:30

1. Generelles Konzept der Ober- und Unterklassen
2. Erzeugung von Objekten in Klassenhierarchien
3. Verdecken v. Attributen und Überschreiben v. Methoden

1. Generelles Konzept v. Ober- und Unterklassen

Bsp: Klassen Student, Angestellter

Klassen haben viele gleiche Attribute, aber jeweils auch eigene Attribute.

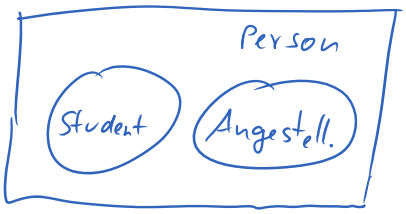
Methode toString wäre in beiden Klassen gleich implementiert => nicht elegant.

Lösung: fasse Gemeinsamkeiten von Student u. Angestellter in neuer Klasse Person zusammen.

Dann deklariere Student und Angestellter als Unterklassen von Person.

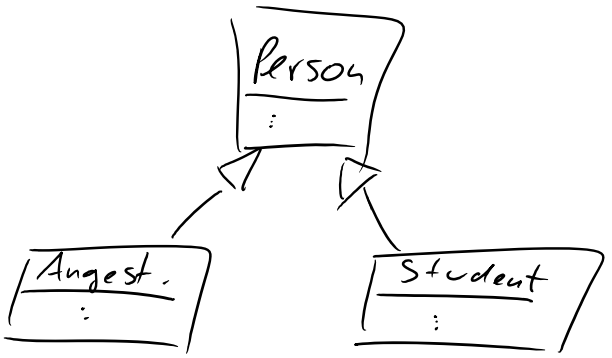
d.h: U-Klasse hat alle Eigen-

schaften der Oberklasse.
Darüber hinaus kann sie
zusätzliche weitere Eigenschaften
haben (z.B. matrikelnr oder
stellung).
U-Klasse erweitert (extends)
die Oberklasse ggf. um weitere
Eigenschaften.



Objekte d. U-Klasse
erben alle Eigenschaften
d. Oberklasse.

Klassendiagramm:



- Hiermit kann man bel. tiefe Klassenhierarchien aufbauen. Attribute + Methoden d. Oberklassen vererben sich auch an U-Klassen der U-Klassen etc. ...
- Java hat nur Einfachvererbung: Jede Klasse hat nur eine direkte Oberklasse. Auf diese Weise gibt es keine Konflikte, falls

in mehreren Oberklassen Methoden/Attribute d. gleichen Namens vorhanden sind. (Simulation v. Mehrfachvererbung: nächster Abschnitt).

- Methoden + Attribute können in d. allgemeinsten Klasse implementiert werden u. sie vererben sich an U-Klassen.
- Vererbung auch bei statischen Methoden + Attributen.
  z.B. Wenn Person eine statische Methode f() enthält, dann kann man auf sie nicht nur mit Person.f() sondern auch mit Student.f() oder Angestellter.f() zugreifen.
- Methoden d. U-Klasse können auch Attribute ändern, die in Oberklasse def. wurden.
  z.B. Student könnte folgende Methode enthalten:

```
void setzeKeyZurueck () {
    key = 0;
}
```

Datentypanpassung + Zugriff auf Objekte in Klassenhierarchien

- Zuweisung v. Objekten v. Unter- zu Oberklasse erlaubt, das Objekt d. U-Klasse wird dabei nicht verändert, es geht keine Inform. verloren
  (im Bsp: matrikelnr von s ist weiterhin vorhanden)
- Zuweisung zwischen 2 U-Klassen verboten, denn die beiden Klassen stehen in keiner Beziehung zueinander
- Zugriff auf Eigenschaften eines Objekts d. U-Klasse

Man darf nicht über Var. der Oberklasse auf Eigenschaften zugreifen, die in der U-Klasse deklariert wurden.
Grund: Man kann i.A. nicht sicherstellen, ob Var. der O-Klasse wirklich auf ein Objekt dieser U-Klasse zeigt.

- Zuweisung von Oser- zu U-Klasse
  im Prinzip verboten, denn man weiß nicht, ob Var. der O-Klasse wirklich auf ein Objekt dieser U-Klasse zeigt.
  Aber: explizite Typanpassung v. O-Klasse zu U-Klasse ist möglich

Analog zu: `double d = 3;` ← implizite Typanpassg. v. int nach double

`int i = (int) d;` ← explizite Typanpassung v. double nach int

Wenn Objekt d. Oberklasse nicht einem Objekt der U-Klasse entspricht, dann scheitert die explizite Typanpassung ⇒ ggf. Prog-Abbruch.
Man kann mit "instanceof" überprüfen, zu welchen Klassen ein Objekt gehört. (Meist kann man aber ohne "instanceof" auskommen)

Konzeptionelles Modell f. Objekte in Klassenhierarchien

P "sieht" im Bsp nur den Teil des Student-Objekts, der einem Person-Objekt entspricht.

2. Erzeugung von Objekten in Klassenhierarchien

- Generell: mit new und Aufruf eines Konstruktors
- Es ex. Klasse Object, die automatisch Oberklasse v. allen Java-Klassen ist. ⇒ Auch bisher haben wir Klassen in Klassenhierarchien geschrieben.

Wie arbeiten Konstruktoren in Klassehierarchien?

- Bsp: Konstruktor Student() soll alle Attribute setzen, indem Benutzer gefragt wird. Dazu kann man im Konstruktor der U-Klasse den Konstruktor der O-Klasse aufrufen.

  super() ruft den Konstruktor ohne Arg. der Oberklasse auf

  Warum ruft man in Student() nicht

  Person p = new Person();

  auf? Weil dadurch nicht die Attribute des gerade erzeugten Student-Objekts gesetzt werden.

- super(...) kann auch für Konstruktoren mit Argumenten benutzt werden.
- Man kann auch aus einem Konstruktor einen anderen Konstruktor der eigenen Klasse aufrufen: this(...)

  ⇒ this ist das eigene Objekt (in der eigenen Klasse)

  super ist das eigene Objekt (aus der

Sicht der Oberklasse)

this. ... : greift auf Eigenschaften des aktuellen Objekts zu
this (...) : ruft Konstruktor (d. eig. Klasse) f. aktuelles Objekt auf.

Analog dazu: super. .... und super (...).
↑ später

- Aufruf eines anderen Konstruktors mit this (...) oder super (...) muss die erste Anweisung des Konstruktors sein.

Wenn man keine solche Anweisung schreibt, dann ergänzt Java automatisch

super ();

als 1. Anweisung des Konstruktors.
(Bei Klassen ohne "... extends ..." wird dann also

Object () ausgeführt.)

- Potentielle Fehlerquelle: Wenn O-Klasse keinen Konstruktor ohne Arg. besitzt, müssen alle Konstruktoren d. U-Klasse explizit mit geeignetem this(..) oder super(...) - Aufruf starten, da super() zum Fehler führen würde.
- Vor Ausführg. v. Konstruktoren werden alle Attribute auf ihre Initialwerte gesetzt (falls diese bei Deklaration d. Attributs angegeben werden — sonst wird der Initialwert des Typs genommen)

↑ z.B. 0 bei int, null bei Referenztypen, ...