- Vererbung -

Überblick Gesamt

- Einführung in das Programmieren (S. 1-140, S. 181-226, = 185 Seiten)
  - Grundlagen (Kapitel 1, im Schnelldurchlauf!)
  - Datentypen und Operatoren (Kapitel 2)
  - Kontrollanweisungen (Kapitel 3)
  - Arrays (Kapitel 5)
- Objektorientierte Programmierung (S. 141-180, S. 227-388, = 200 Seiten)
  - Klassen, Objekte Methoden (Kapitel 4)
  - Genauere Betrachtung der Methoden und Klassen (Kapitel 6)
  - Vererbung und Polymorphie (Kapitel 7)
  - Pakete und Schnittstellen (Kapitel 8)
  - Ausnahmebehandlung (Kapitel 9)

Überblick Heute

- Vererbung in Java: Hard Facts
- Konstruktion von Klassen in Klassenhierarchien
- Beispiel für mehrstufige Klassenhierarchien
- Vererbung als Aufweichung des Typzwangs in Java?
- Anwendungsfälle
- Übungsbeispiel
- ClassCasting
- Zusammenfassung
- java.lang.Object und ArrayList

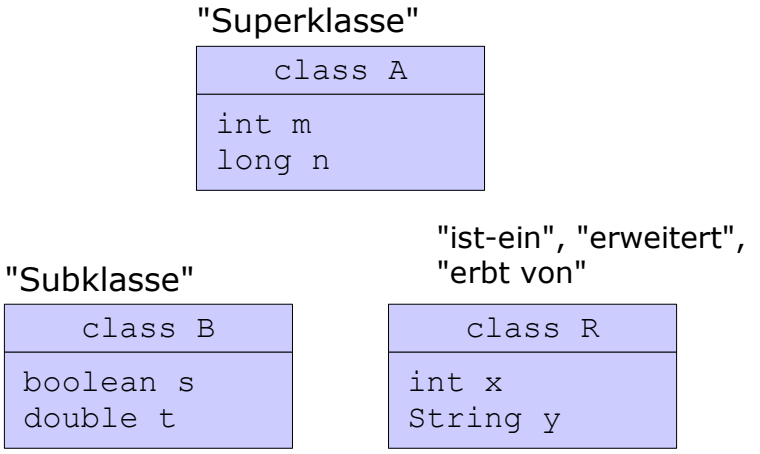
Vererbung in Java: Hard Facts (1)

- Vererbung als weiterer Eckpfeiler der OOP
- ermöglicht Einrichtung hierarchischen Klassifikationen

- mehrstufige Vererbung möglich und üblich
  - einige Klassen schlüpfen in 2 Rollen

- Einschränkung:
  - pro Klasse max. eine Superklasse
  - ... aber beliebig viele Subklassen

- die Superklasse weiss nichts von ihren Subklassen




```
class A { ... } // Superklasse (keine Angabe von Subklassen!)
class B extends A { ... } // Subklasse erweitert Superklasse
```

Vererbung in Java: Hard Facts (2)

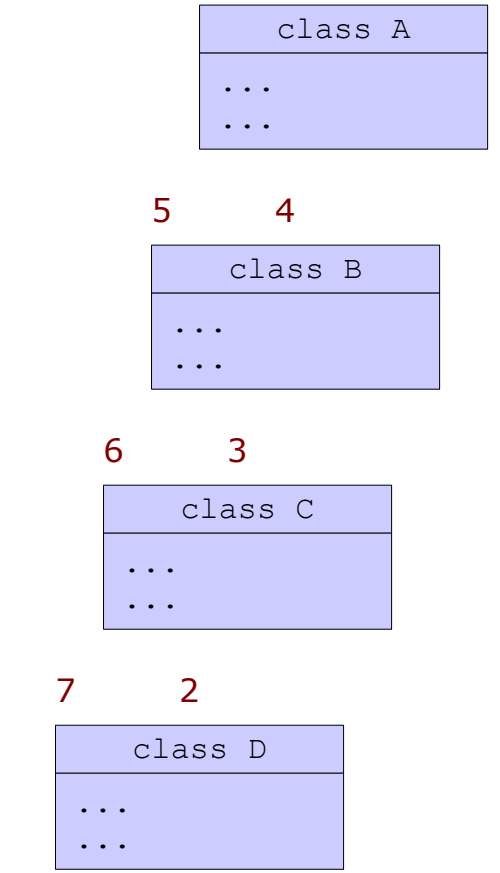
- Zugriffsbeschränkungen gelten weiter
  - in Subklasse *kein* Zugriff auf private Elemente der Superklasse
  - private Elemente sind *da*, aber in der Subklasse nicht sichtbar

```
class U {
    private int x;
    public int setX(int newX) {
        x = newX;
    }
}

class V extends U {
    void someMethod() {
        x = 12; // geht nicht, x ist private in U
        setX(12); // geht okay, setX() ist public in U
    }
}
```

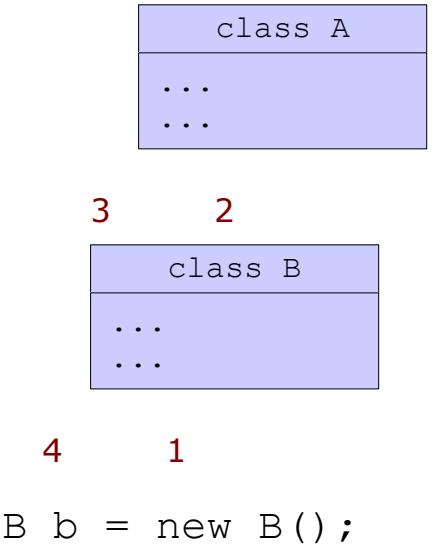
Konstruktion von Klassen in Klassenhierarchien (1)

- vordergründig unauffällig
  - aus Benutzersicht Konstruktion wie gehabt
  - `B b = new B();`
- intern allerdings nicht unproblematisch
  - Objekt b: Attribute m, n, s, t (vgl. vorherige Folie)
  - Initialisierung aller Attribute nicht immer möglich
  - zum Beispiel: m, n private in A, und keine Setter?
  - Codeduplizierung in Konstruktoren von A und B?
- Lösung
  - kommt der Konstr. einer Klasse zur Ausführung, ruft er als *erstes* den Konstruktor der Superklasse auf.
  - dadurch schrittweise Initialisierung der Instanz
  - keine Codeduplizierung



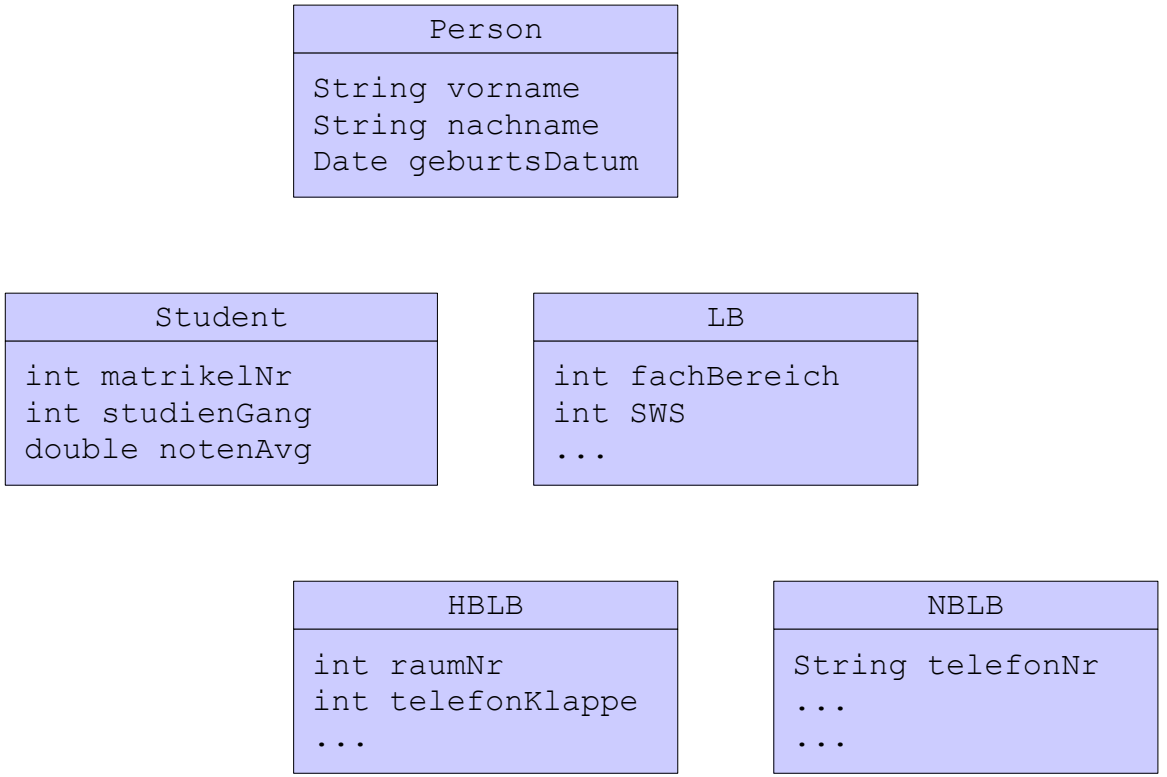
Konstruktion von Klassen in Klassenhierarchien (2)

- Fall 1: kein programmierter Konstruktor in B
  Standard-Konstruktor in B wird ausgeführt, dieser ruft als erstes den Standard-Konstruktor der Superklasse A auf
- Fall 2: Konstruktor in B, aber nicht in A
  eigener Konstruktor in B kann benutzt werden, auch hier wird automatisch der Std.-Konstruktor von A aufgerufen
- Fall 3: Konstruktoren in B und A (üblich)
  soll in A ein eigener Konstruktor ausgeführt werden, muss er zu Beginn des Konstruktors in B aufgerufen werden:




```
B(int m, int n, int s, int t) {
    super(m, n); // Kst. aus A; muss an 1. Stelle stehen!
    this.s = s; // weitere Initialisierung der Attr. aus B
    this.t = t;
}
```

Beispiel für mehrstufige Klassenhierarchie



Vererbung als Aufweichung des Typzwangs?

- bisher galt:
  - eine Objektvariable kann ausschliesslich Instanzen von *derjenigen* Klasse aufnehmen, für die sie deklariert ist
- tatsächlich stimmt aber:
  - auch Instanzen von Subklassen der deklarierten Klasse können aufgenommen werden
- Beispiel
  - `Person p = new Student();`
  - ein Student "ist-ein" Person!
- statischer vs. dynamischer Typ:
  - *statischer Typ*: deklarierter Typ
  - *dynamischer Typ*: Typ der aktuell zugewiesenen Instanz

Eine Objektvariable speichert eine Referenz auf eine Instanz derjenigen Klasse, für die sie deklariert wurde, *oder von irgendeiner Subklasse davon*.

(... denn auch ein Student "ist-ein" Person)

3. Axiom von Grabo

Beispiele zur Klassenhierarchie von Folie 8

- was geht? was geht nicht?

```
// Zugriffe (Annahme: alle Attribute sind public)
HBLB b = new HBLB();
b.raumNr = 12;
b.vorname = "Fritz";

Person p = new Student();
p.vorname = "Fritz";
p.matrikelNr = 1027;

// Zuweisungen
Person p = new Person();
Student s = p;
p = new NBLB();
LB b = p;
b = new LB();
HBLB h = b;
```

Anwendungsfälle

- Wozu Variablen, deren statischer Typ != dynamischer Typ?
  - Vorteile: Programmteile werden allgemein/generisch gehalten und sind dadurch universeller einsetzbar und leichter wiederverwendbar. D.R.Y.!
  - Beispiel: `printPersonName(...)` kann mit Instanzen der Klasse Person oder irgendeiner Subklasse davon aufgerufen werden.

```
void printPersonName(Person p) {
    System.out.println(p.getName());
}

Person a = new Person("Anton");
Student b = new Student("Babsi", 1027);
LB c = new LB("Connie");

printPersonName(b);
```

Übungsbeispiel

- wähle einen für dich interessanten Themenbereich, und definiere dafür für zwei unterschiedliche Applikationen eine mehrstufige Klassenhierarchie (Minimum 4 Stufen, 10 Klassen)
- erkläre und begründe jeweils kurz die Wahl der Kriterien für das Erstellen der Klassenhierarchie = die *Sicht* auf den Themenbereich
- die Klassen sollen Konstruktoren und Attribute definieren, Methoden sind für diese Übung nicht notwendig; Datentypen für Attribute können auch "erfunden" werden (z.B für LB: `Fach[] faecher`)
- überlege und kennzeichne, welche Klassen nur als Hilfe für die Konstruktion der Hierarchie dienen und nicht sinnvollerweise instanziiert werden (sollten)
- (Abgabe als PDF am eCampus)

ClassCasting

- ClassCasting ermöglicht *explizite* Typumwandlung
  - v.a. von allgemeinerem zu speziellerem Typ
  - hebelt den Schutz des strikten Typsystems teilweise aus
  - Programmierer muss genau wissen, was er/sie tut
  - allgemeine Form: `(Zieltyp) Ausdruck;`

- zu möglichen unerwünschten Folgen gehören
  - Informationsverlust, z.B. bei Cast von `double` zu `int`
  - Fehler zur Laufzeit, z.B. bei Cast einer `Person` zu einem `Student`

```
Person p = new Student("Peter", 1029);
Person f = new Person("Fritz");
Kfz k = new Kfz(...);

Student s = (Student) p; // ok: dyn. Typ von p ist zZ Student
Student t = (Student) f; // geht hier nicht: Laufzeitfehler!
Student u = (Student) k; // geht niemals: Compilerfehler!
```

Zusammenfassung

- 3. Axiom: "*Objektvariable speichert Referenz auf Instanz derjenigen Klasse, für die sie deklariert wurde, oder von irgendeiner Subklasse davon.*" Das ermöglicht unter anderem das Schreiben von universeller einsetzbaren Programmteilen.

- der Compiler betrachtet nur den *statischen* Typ eines Ausdrucks. Über den *dynamischen* Typ kann zur Zeit des Kompilierens keine Aussage gemacht werden.

- per ClassCast kann der Typ eines Ausdrucks vom Programmierer explizit in einen anderen (meist einen konkreteren) umgewandelt werden. Der Compiler kann hierbei nicht garantieren, dass zur Laufzeit des Programms der Ausdrucks tatsächlich in den gewünschten Typ umgewandelt werden kann.
  Der Programmierer übernimmt die Verantwortung.

Jede Klasse erbt von java.lang.Object

4. Axiom von Grabo

die Mutter aller Klassen: java.lang.Object

- im Extremfall möchten wir mit *allen* Objekttypen arbeiten können
  - in diesem Fall arbeiten wir mit Instanzen d. Klasse `java.lang.Object`
  - jede Klasse ist immer auch Subklasse von `java.lang.Object`

- Beispiel für Anwendung:
  - Sammeln und Verwalten von mehreren Objekten in Mengen/Listen
  - Der Typ (und damit der Inhalt) der Objekte ist dabei egal

- Klasse `java.util.ArrayList`
  - ermöglicht Arbeiten mit beliebig grossen Listen von Objekten
  - viele Vorteile im Vergleich zum herkömmlichen Array
  - siehe Klassendokumentation aus J2SE 1.4.2!
  - einfach gehaltene Schnittstelle:
    - `void add(Object o)`
    - `Object get(int index)`
    - `Object remove(int index)`
    - `int size()`
    - `...`

Anwendungsbeispiel für `ArrayList`

```
ArrayList list = new ArrayList();

list.add(new Student("Fritz", 1027));
list.add(new Student("Peter", 1028));
list.add(new Student("Roman", 1012));
...

System.out.println(list.size() + " Elemente i.d. Liste");

for (int i = 0; i < list.size(); i++) {
    Student s = (Student) list.get(i); // ClassCast notwendig!
    System.out.println(s.getMatrikelNr() + " " + s.getName());
}
```