OO

Adrian Schüpbach
adrian_laurent.schuepbach@alumni.ethz.ch

Worum geht es?

- Objektorientiertes Programmieren
- Ableitung von Klassen
- Vererbung
- Überschreiben von Methoden
- Abstrakte Methoden
- Interfaces

Klassen ableiten

Klassen ableiten

- Gesehen
  - Klassen = Typ → Objekte
  - Viele Objekte → “Objektorientiertes Programmieren”

Klassen ableiten

- Gesehen
  - Klassen = Typ → Objekte
  - Viele Objekte → “Objektorientiertes Programmieren”
- Weiterer Schritt: Klassen leiten von Klassen ab
  - ... und erben von Klassen

Klassen ableiten

- Deklaration der Klasse → hinschreiben, wovon sie ableitet
  - `extends`

Klassen ableiten

- Deklaration der Klasse → hinschreiben, wovon sie ableitet
  - `extends`
- Methoden und Felder werden vererbt
  - Neue Klasse hat alle Methoden und Felder der “Super”-Klasse
  - Kann diese benutzen, wie eigene Methoden und Felder
- Klasse kann zusätzliche eigene Methoden und Felder definieren
  - Kann geerbte und eigene benutzen

Klassen ableiten

Ableitung: Basisklasse Person + zwei abgeleitete Klassen

```
public class Person {
  String vor,nach;
  public String getName() {
    return (nach + ", " + vor);
  }
}
public class Student extends Person {
  int matrikel;
  public int getMatrikel() { return(matrikel); }
}
public class Dozent extends Person {
  int personalNr;
  public int getPersonalNummer() { return(personalNr); }
}
```

Klassen ableiten

- Studenten sind auch Personen

Klassen ableiten

- Studenten sind auch Personen
- Auch Dozenten sind Personen

Klassen ableiten

- Studenten sind auch Personen
- Auch Dozenten sind Personen

- Gemeinsamkeiten
  - Vorname
  - Nachname
- Unterschiede
  - Studenten haben Matrikelnummer
  - Dozenten haben Personalnummer
  - → Spezialisierung

Klassen ableiten

Ableitung: Benutzung der Klassen

```
public class Programm61 {
  public static void main(String[] args) {
    Student s = new Student();
    s.vor = "Marili"; s.nach = "Mortadella";
    s.matrikel=1981276;

    Dozent d = new Dozent();
    d.vor = "Kurtli"; d.nach = "Maccaroni";
    d.personalNr = 17;
    System.out.println("Studentin: " + s.getName()
            + ", Dozent: " + d.getName());
  }
}
```

Klassen ableiten

- Klasse besitzt auch *immer* Typ der Basisklasse
- Kann immer implizit hinuntergecastet werden
  - Hat dann Zugriff auf Felder und Methoden der Basisklasse

Klassen ableiten

- Klasse besitzt auch *immer* Typ der Basisklasse
- Kann immer implizit hinuntergecastet werden
  - Hat dann Zugriff auf Felder und Methoden der Basisklasse
- Bsp.:
  - `Person p = s;`

Klassen ableiten

Ableitung und Datenstrukturen

```
public class Programm61 {
  public static void main(String[] args) {
    Student s = new Student();
        s.vor = "Marili"; s.nach = "Mortadella";
        s.matrikel=1981276;
    Dozent d = new Dozent();
        d.vor = "Kurtli"; d.nach = "Maccaroni";
        d.personalNr = 17;
    // Datenstruktur fuer Person. Student und
    // Dozent runtergecastet
    TreeMap<String,Person> tm =
        new TreeMap<String,Person>();
    // beide werden konvertiert in Person
    tm.put(s.nach, s);
    tm.put(d.nach, d);
  }
```

Klassen ableiten

OO-Design

Klassen ableiten

OO-Design

- Warum macht man das?

Klassen ableiten

OO-Design

- Warum macht man das?
- Gemeinsame Funktionalität/Attribute in Basisklasse(n) gliedern
  - Weniger duplizierter Code
  - Weniger fehleranfällig

Klassen ableiten

OO-Design

- Warum macht man das?
- Gemeinsame Funktionalität/Attribute in Basisklasse(n) gliedern
  - Weniger duplizierter Code
  - Weniger fehleranfällig
- Starten mit der allgemeinsten Klasse

Klassen ableiten

OO-Design

- Warum macht man das?
- Gemeinsame Funktionalität/Attribute in Basisklasse(n) gliedern
  - Weniger duplizierter Code
  - Weniger fehleranfällig
- Starten mit der allgemeinsten Klasse
- → Spezialisierung in abeleiteten Klassen

Methoden überschreiben

OO-Design

- Klasse erbt Methoden, wenn sie von anderer Klasse ableitet
- Kann geerbte Methoden benutzen

Methoden überschreiben

OO-Design

- Klasse erbt Methoden, wenn sie von anderer Klasse ableitet
- Kann geerbte Methoden benutzen
- Klasse kann neue Methoden hinzufügen und benutzen

Methoden überschreiben

OO-Design

- Klasse erbt Methoden, wenn sie von anderer Klasse ableitet
- Kann geerbte Methoden benutzen
- Klasse kann neue Methoden hinzufügen und benutzen

- Klasse kann geerbte Methode überschreiben
  - Gleiche Signatur
  - Anderer Code

Methoden überschreiben

OO-Design

- Bsp.: Wetterobjekte
  - Basisklasse `WetterObjekt`
  - Hat Methode `String zeigWetter()`
- Klasse `Sonne` leitet von `WetterObjekt` ab
  - `String zeigWetter()` soll “Sonne” ausgeben
- Klasse `Regen` leitet von `WetterObjekt` ab
  - `String zeigWetter()` soll “Regen” ausgeben
- Klasse `Nieselregen` leitet von `Regen` ab
  - `String zeigWetter()` soll “Nieselregen” ausgeben

Methoden überschreiben

OO-Design

Wetter

```
class WetterObjekt {
  String zeigWetter() { return ("Wetter"); }
}
class Sonne extends WetterObjekt {
  String zeigWetter() { return ("Sonne"); }
}
class Regen extends WetterObjekt {
  String zeigWetter() { return ("Regen"); }
}
class Nieselregen extends Regen {
  String zeigWetter() { return ("Nieselregen"); }
}
```

Methoden überschreiben

OO-Design

Wetter

```
public class Programm62 {
  public static void main(String[] args) {
    WetterObjekt wo = new WetterObjekt();
    System.out.println("wo: " + wo.zeigWetter();

    Sonne s = new Sonne();
    System.out.println("s: " + s.zeigWetter();

    Regen r = new Regen();
    System.out.println("r: " + r.zeigWetter();

    Nieselregen nr = new Nieselregen();
    System.out.println("nr: " + nr.zeigWetter();

  }
```

Methoden überschreiben

Wetter: Runtercasten: Überschriebene Methode

```
public class Programm63 {
 public static void main(String[] args) {
  WetterObjekt wetter;

  Sonne s = new Sonne();
  wetter = s;
  System.out.println("Wetter: "+wetter.zeigWetter());

  Regen r = new Regen();
  wetter = r;
  System.out.println("Wetter: "+wetter.zeigWetter());

  Nieselregen nr = new Nieselregen();
  wetter = nr;
  System.out.println("Wetter: "+wetter.zeigWetter());
 }
```

Methoden überschreiben

OO-Design

- Aufruf der überschriebenen Methode in der Basisklasse
  - Objekt auf Basisklasse runtercasten
  - Aufruf der Methode
  - Überschriebene Methode wird aufgerufen

Methoden überschreiben

OO-Design

- Aufruf der überschriebenen Methode in der Basisklasse
  - Objekt auf Basisklasse runtercasten
  - Aufruf der Methode
  - Überschriebene Methode wird aufgerufen
- Vorteil:
  - Algorithmus arbeitet auf Basisklasse
  - Kennt die verfügbaren Methoden
  - Klassenableitungen spezialisieren und überschreiben Methoden
  - Algorithmus benutzt überschriebene Methoden pro Objekt

Wetter-“Stream”

- Verschiedene Wetterobjekte
- Queue, die Wetter ausgibt
- Jemand füllt Wetterdaten ein

Wetter-“Stream”

- Verschiedene Wetterobjekte
- Queue, die Wetter ausgibt
- Jemand füllt Wetterdaten ein

- Queue + Wetterausgabealgorithmus
  - Müssen Wetterobjekte nicht kennen
  - Algorithmus weiss nur, dass Methode `zeigWetter()` gibt
  - Ruft diese auf Basisklasse auf
  - Wetter wird richtig angezeigt, da Methode überschrieben ist

Wetter-“Stream”

- → Demo

Interfaces

Interfaces

- Algorithmus muss nur wissen, welche Methoden es gibt
  - Zur Entwicklungszeit ev. Klassen nicht nicht bestimmt
  - Klassen werden ev. von “Kunden” meiner Algorithmenklasse erstellt
  - Algorithmus muss trotzdem wissen, welche Methoden es geben wird

Interfaces

- Algorithmus muss nur wissen, welche Methoden es gibt
  - Zur Entwicklungszeit ev. Klassen nicht nicht bestimmt
  - Klassen werden ev. von “Kunden” meiner Algorithmenklasse erstellt
  - Algorithmus muss trotzdem wissen, welche Methoden es geben wird
- Interface
  - Definiert nur Signaturen
  - Keine Implementation
  - Algorithmus benutzt Interface

Interfaces

Interface: Definiert Methoden, die in Klassen sein *müssen*

```
public interface Fahrzeug {
    int gewicht = 0;
    int anzahlAchsen = 0;

    public void setzeAnzahlAngetriebenerAchsen(
        int anzahl);
    public void setzeAnzahlAchsen(int anzahl);
    public void setzeGewicht(int gewicht);
    public void setzeLaengeUeberAlles(int l);
    public double gibAchsLast();
    public int gibLaengeUeberAlles();
}
```

Interfaces

- Interface
  - Kein Code
  - $\Rightarrow$ Muss Klasse implementieren, um Code zu haben
- Klasse
  - **Implementiert** Interface
  - *Muss* alle Methoden implementieren/“überschreiben”

Interfaces

Klasse *implementiert* Interface → `implements`

```
public class Dreirad implements Fahrzeug {
    public void setzeAnzahlAngetriebenerAchsen(
        int anzahl) {
    }
    public void setzeAnzahlAchsen(int anzahl) {
    }
    public void setzeGewicht(int gewicht) {
    }
    public void setzeLaengeUeberAlles(int l) {
    }
    public void gibAchsLast() {
    }
    public int gibLaengeUeberAlles() {
        return 0;
    }
}
```

Interfaces

- Interface
  - Können bestehende Interfaces erweitern
  - $\rightarrow$ Leiten von bestehenden Interfaces ab
- Kann sich ganze Hierarchie bauen

Interfaces

Interface erweitern

```
public interface Schienenfahrzeug extends Fahrzeug {
    public void setzeSpurWeite(int spurWeite);
    public int gibSpurWeite();
    public boolean istStellwerkTauglich();
}
```

Abstrakte Klassen

Abstrakte Klassen

- Abstrakte Klassen
  - Können Methoden ohne Code haben
  - Können Interface implementieren, müssen aber nicht alle Methoden implementieren
  - Können nicht instanziiert werden

Abstrakte Klassen

- Abstrakte Klassen
  - Können Methoden ohne Code haben
  - Können Interface implementieren, müssen aber nicht alle Methoden implementieren
  - Können nicht instanziiert werden
- Gemeinsame Funktionalität kann in abstrakten Klassen implementiert werden
- Methoden, die spezialisiert werden *sollen/müssen*, leer lassen

  - Klassen, die von abstrakten Klassen ableiten, sind gezwungen, Spezialisierung zu geben

Abstrakte Klassen

Abstrakte Schienenfahrzeug-Klasse

```
public abstract class AbstractSchienenFahrzeug
  implements Schienenfahrzeug {

    public double gibAchsLast() {
        return ((double)gewicht
                / (double)anzahlAchsen);
    }
    public boolean istStellwerkTauglich() {
        return (gibAchsLast() >= 2500);
    }
}
```

Konkrete Klassen/Implementation

Konkrete Klassen/Implementation

- Konkrete Klasse
  - Leitet von abstrakter Klasse ab
  - Und/oder implementiert Interface(s)
- Regeln:
  1. Darf nur von *einer* Klasse ableiten
  2. Darf mehrere Interfaces implementieren
- Konkrete Klasse kann instanziiert werden (→ Objekt)

Konkrete Klassen

Klasse “Krokodil”

```
public class Krokodil
    extends AbstractSchienenFahrzeug {

    public void setzeSpurWeite(int spurWeite) {
    }
    public int gibSpurWeite() {
        return 0;
    }
    public void setzeAnzahlAngetriebenerAchsen(
        int anzahl) {
    }
    public void setzeAnzahlAchsen(int anzahl) {
    }
    public void setzeGewicht(int gewicht) {
    }
    public void setzeLaengeUeberAlles() {
```

Was gehört in Basisklasse, was überschreiben?

Was gehört in Basisklasse, was überschreiben?

- Ist das nun gutes Design?

Was gehört in Basisklasse, was überschreiben?

- Ist das nun gutes Design?
- Die meisten Methoden könnten in Basisklasse sein
- Anzahl Achsen, Gewicht, ... für alle Fahrzeugtypen gleiche Methode benutzen

Was gehört in Basisklasse, was überschreiben?

- Ist das nun gutes Design?
- Die meisten Methoden könnten in Basisklasse sein
- Anzahl Achsen, Gewicht, ... für alle Fahrzeugtypen gleiche Methode benutzen
- $\rightarrow$ Sollte Methoden überschreiben, die *wirklich* spezialisiert sein müssen

Was gehört in Basisklasse, was überschreiben?

- Annahme: Simulationsframework
  - Framework kann Fahrzeuge “fahren lassen”
  - Benutzt Methoden `beschleunigen()` und `bremsen()` der Fahrzeuge

Was gehört in Basisklasse, was überschreiben?

- Annahme: Simulationsframework
  - Framework kann Fahrzeuge “fahren lassen”
  - Benutzt Methoden `beschleunigen()` und `bremsen()` der Fahrzeuge
- Beschleunigen

Was gehört in Basisklasse, was überschreiben?

- Annahme: Simulationsframework
  - Framework kann Fahrzeuge “fahren lassen”
  - Benutzt Methoden `beschleunigen()` und `bremsen()` der Fahrzeuge
- Beschleunigen
  - Dreirad: Schneller trampeln

Was gehört in Basisklasse, was überschreiben?

- Annahme: Simulationsframework
  - Framework kann Fahrzeuge “fahren lassen”
  - Benutzt Methoden `beschleunigen()` und `bremsen()` der Fahrzeuge
- Beschleunigen
  - Dreirad: Schneller trampeln
  - Auto: Auf das Gaspedal drücken, schalten, ...

Was gehört in Basisklasse, was überschreiben?

- Annahme: Simulationsframework
  - Framework kann Fahrzeuge “fahren lassen”
  - Benutzt Methoden `beschleunigen()` und `bremsen()` der Fahrzeuge
- Beschleunigen
  - Dreirad: Schneller trampeln
  - Auto: Auf das Gaspedal drücken, schalten, ...
  - Krokodillok: (Manueller) Stufenschalter um Anzahl Stufen erhöhen

Was gehört in Basisklasse, was überschreiben?

- Annahme: Simulationsframework
  - Framework kann Fahrzeuge “fahren lassen”
  - Benutzt Methoden `beschleunigen()` und `bremsen()` der Fahrzeuge
- Beschleunigen
  - Dreirad: Schneller trampeln
  - Auto: Auf das Gaspedal drücken, schalten, ...
  - Krokodillok: (Manueller) Stufenschalter um Anzahl Stufen erhöhen
  - $\rightarrow$ wirklich unterschiedliche Methoden, wie man beschleunigt

Was gehört in Basisklasse, was überschreiben?

- Annahme: Simulationsframework
  - Framework kann Fahrzeuge “fahren lassen”
  - Benutzt Methoden `beschleunigen()` und `bremsen()` der Fahrzeuge
- Beschleunigen
  - Dreirad: Schneller trampeln
  - Auto: Auf das Gaspedal drücken, schalten, ...
  - Krokodillok: (Manueller) Stufenschalter um Anzahl Stufen erhöhen
  - $\rightarrow$ wirklich unterschiedliche Methoden, wie man beschleunigt
- Bremsen

Was gehört in Basisklasse, was überschreiben?

- Annahme: Simulationsframework
  - Framework kann Fahrzeuge “fahren lassen”
  - Benutzt Methoden `beschleunigen()` und `bremsen()` der Fahrzeuge
- Beschleunigen
  - Dreirad: Schneller trampeln
  - Auto: Auf das Gaspedal drücken, schalten, ...
  - Krokodillok: (Manueller) Stufenschalter um Anzahl Stufen erhöhen
  - $\rightarrow$ wirklich unterschiedliche Methoden, wie man beschleunigt
- Bremsen
  - Dreirad: Langsamer trampeln (mit Füssen bremsen)

Was gehört in Basisklasse, was überschreiben?

- Annahme: Simulationsframework
  - Framework kann Fahrzeuge “fahren lassen”
  - Benutzt Methoden `beschleunigen()` und `bremsen()` der Fahrzeuge
- Beschleunigen
  - Dreirad: Schneller trampeln
  - Auto: Auf das Gaspedal drücken, schalten, ...
  - Krokodillok: (Manueller) Stufenschalter um Anzahl Stufen erhöhen
  - $\rightarrow$ wirklich unterschiedliche Methoden, wie man beschleunigt
- Bremsen
  - Dreirad: Langsamer trampeln (mit Füssen bremsen)
  - Auto: Bremse und schliesslich Kupplung drücken

Was gehört in Basisklasse, was überschreiben?

- Annahme: Simulationsframework
  - Framework kann Fahrzeuge “fahren lassen”
  - Benutzt Methoden `beschleunigen()` und `bremsen()` der Fahrzeuge
- Beschleunigen
  - Dreirad: Schneller trampeln
  - Auto: Auf das Gaspedal drücken, schalten, ...
  - Krokodillok: (Manueller) Stufenschalter um Anzahl Stufen erhöhen
  - $\rightarrow$ wirklich unterschiedliche Methoden, wie man beschleunigt
- Bremsen
  - Dreirad: Langsamer trampeln (mit Füssen bremsen)
  - Auto: Bremse und schliesslich Kupplung drücken
  - Krokodillok: Stufenschalter hinunterschalten, Vakuum zerstören

Was gehört in Basisklasse, was überschreiben?

- Annahme: Simulationsframework
  - Framework kann Fahrzeuge “fahren lassen”
  - Benutzt Methoden `beschleunigen()` und `bremsen()` der Fahrzeuge
- Beschleunigen
  - Dreirad: Schneller trampeln
  - Auto: Auf das Gaspedal drücken, schalten, ...
  - Krokodillok: (Manueller) Stufenschalter um Anzahl Stufen erhöhen
  - → wirklich unterschiedliche Methoden, wie man beschleunigt
- Bremsen
  - Dreirad: Langsamer trampeln (mit Füssen bremsen)
  - Auto: Bremse und schliesslich Kupplung drücken
  - Krokodillok: Stufenschalter hinunterschalten, Vakuum zerstören
  - → wirklich unterschiedliche Methoden, wie man bremst

Was gehört in Basisklasse, was überschreiben?

- Simulationsframework kennt nicht alle Beschleunigungs- und Bremstechniken
  - V.a. werden neue Fahrzeuge entwickelt, die ev. andere Techniken haben
  - Bsps.:
    - Ge 4/4 II: Phasenanschnittssteuerung
    - ICN: Frequenzsteuerung
    - Mountainbike: Scheibenbremsen (statt mit Pedal bremsen)
- $\rightarrow$ Methoden `beschleunigen()` und `bremsen()` verbergen die tatsächliche Technik

Was gehört in Basisklasse, was überschreiben?

Interface: das müssen Fahrzeuge können

```
public interface Fahrzeug {
    int gewicht = 0;
    int anzahlAchsen = 0;

    public void setzeAnzahlAngetriebenerAchsen(
        int anzahl);
    public void setzeAnzahlAchsen(int anzahl);
    public void setzeGewicht(int gewicht);
    public void setzeLaengeUeberAlles(int l);
    public double gibAchsLast();
    public int gibLaengeUeberAlles();

    void beschleunigen(double prozent);
    void bremsen(double prozent);
    void anhalten();
}
```

Was gehört in Basisklasse, was überschreiben?

Abstraktes Fahrzeug: Kann scho etwas, aber noch nicht alles

```
public abstract class AbstractFahrzeug implements
  Fahrzeug {
    int anzahlA, antA, gewicht, lange;
    public void setzeAnzahlAngetriebenerAchsen(
      int anzahl) {
        this.antA = anzahl;
    }
    public void setzeAnzahlAchsen(int anzahl) {
        this.anzahlA = anzahl;
    }
    public void setzeGewicht(int gewicht) {
        this.gewicht = gewicht;
    }
    public double gibAchsLast() {
        return ((double)gewicht / (double)anzahlAchsen);
    }
```

Was gehört in Basisklasse, was überschreiben?

Abstraktes Schienenfahrzeug: Spezialisierte Methoden implementiert

```
public abstract class AbstractSchienenFahrzeug
  extends AbstractFahrzeug {
    int spurweite;
    public void setzeSpurWeite(int spurWeite) {
        this.spurweite = spurWeite;
    }
    public int gibSpurWeite() {
        return (spurweite);
    }
    public boolean istStellwerkTauglich() {
        return (gibAchsLast() >= 2500);
    }
}
```

Was gehört in Basisklasse, was überschreiben?

Krokodil: Kann fahren und bremsen

```
public class Krokodil
 extends AbstractSchienenFahrzeug {
  int stufe, vakuum;
  public void beschleunigen(double prozent) {
    vakuum = 56; //mmHg
    if (stufe == 0) { stufe = 1; }
    else {
      stufe+=(int)(stufe*(1+(prozent/(double)100.0)));
    }
  }
  public void bremsen(double prozent) {
    stufe = 0;
    if (vakuum > 0) {
      vakuum-=(int)(vakuum*(1+(prozent/(double)100.0)));
    }
  }
```

Was gehört in Basisklasse, was überschreiben?

Dreirad: Kann fahren und bremsen

```
public class Dreirad extends AbstractFahrzeug {
  int trampelFrequenz = 0;
  public void beschleunigen(double prozent) {
    trampelFrequenz+=(int)(trampelFrequenz
        *(1+prozent/100.0));
  }
  public void bremsen(double prozent) {
    trampelFrequenz-=(int)(trampelFrequenz
        *(1+prozent/100.0));
  }
  public void anhalten() {
    trampelFrequenz = 0;
  }
}
```

Was gehört in Basisklasse, was überschreiben?

Fahren und bremsen lassen

```
public class FahrzeugSimulation {
    public static void main(String[] args) {
        LinkedList<Fahrzeug> fahrzeuge =
            new LinkedList<Fahrzeug>();

        fahrzeuge.add(new Krokodil());
        fahrzeuge.add(new Dreirad());
        fahrzeuge.getFirst().beschleunigen(10.0);
    }
}
```

Ableitung von Object

Ableitung von Object

- → Jede Klasse leitet von Object ab

Ableitung von Object

- $\rightarrow$ Jede Klasse leitet von Object ab
- Alle anderen Ableitungen explizit mit `extends`

Beispiele in Java

- `equals()` gehört zu Object
  - Ist Basisklasse für alle Klassen
  - Jede Klasse hat somit `equals()`
  - Datenstrukturen gehen davon aus, dass sie `equals()` benutzen können
- Spezialisierung
  - Object weiss nicht, welche Klasse erben wird
  - Object weiss nicht, wie “Gleichheit” in abgeleiteter Klasse definiert wird
  - → Methode überschreiben, um zu spezialisieren

Beispiele in Java

- `compareTo()` gehört nicht zu Object
- Gibt Interface `Comparable<T>` mit `compareTo()`

Beispiele in Java

- `compareTo()` gehört nicht zu Object
- Gibt Interface `Comparable<T>` mit `compareTo()`
- Eigene vergleichbare Klassen müssen `Comparable<T>` implementieren
  - Methode `compareTo()` implementieren
  - Datenstruktur erwartet `Comparable<T>`

Beispiele in Java

- Eigene Exception definieren:
  - Von der Klasse `Exception` ableiten
  - Methode `public String getMessage()` überschreiben

Beispiele in Java

Eigene Exception

```
public class MeineException extends Exception {
    String param;
    public String getMessage() {
        return("Hui, Fehler: " + param);
    }
    public MeineException(String param) {
        this.param = param;
    }
}
```

Sichtbarkeit der Felder und Methoden

- Gesehen: `public`

Sichtbarkeit der Felder und Methoden

- Gesehen: `public`
- Es gibt: `public`, `protected`, `private`
  - Diese Schlüsselwörter definieren Sichtbarkeit

Sichtbarkeit der Felder und Methoden

- Gesehen: `public`
- Es gibt: `public`, `protected`, `private`
  - Diese Schlüsselwörter definieren Sichtbarkeit

- `public`
  - Alle dürfen Methode aufrufen resp. auf Feld zugreifen
  - → auch andere Klassen
- `protected`
  - Nur abgeleitete Klassen dürfen diese Methode aufrufen
  - Nur abgeleitete Klassen können auf dieses Feld zugreifen
- `private`
  - Nur Klasse, die Methode beinhaltet, darf sie aufrufen
  - Nur Klasse kann auf dieses Feld zugreifen
  - → Nicht mal abgeleitete Klassen haben Zugriff darauf

Sichtbarkeit der Felder und Methoden

- Ziel: Sich überlegen, wie man Klasse schützen kann
  - Was ist interne Funktionalität?
  - → Muss für Benutzer nicht sichtbar sein
  - Was ist externe Schnittstelle?
  - → Für Benutzer sichtbar machen
  - Wie kann Benutzer meine Klasse erweitern?
  - → `protected`, damit Benutzer davon ableiten und es benutzen kann

Sichtbarkeit der Felder und Methoden

- Ziel: Sich überlegen, wie man Klasse schützen kann
  - Was ist interne Funktionalität?
  - → Muss für Benutzer nicht sichtbar sein
  - Was ist externe Schnittstelle?
  - → Für Benutzer sichtbar machen
  - Wie kann Benutzer meine Klasse erweitern?
  - → `protected`, damit Benutzer davon ableiten und es benutzen kann
- Felder schützen:
  - Nicht direkt zugreifen lassen
  - Methode, die zuerst übergebenen Wert prüft und dann erst Feld beschreibt

Sichtbarkeit der Felder und Methoden

Wert vor dem Schreiben prüfen

```
public class Alter {
  protected int alter;
  public void setzeAlter(int alter)
    throws NegativAlterException {
      if (alter < 0) {
        throw new NegativAlterException(alter);
      } else {
        this.alter = alter;
      }
  }
}
```