Erweiterung von Klassen: Ausgliedern von Gemeinsamkeiten

- Beispiel "Versichertenverwaltung"
  - Aufgabenstellung
  - Das Hauptprogramm
  - Der/die Versicherten
  - Die Verwaltung
  - Eine gemeinsame Oberklasse wird beerbt
- Zusammenfassung, Begriffe

Auftrag eines Versicherungsunternehmens

- Gesucht ist eine Software für die Verwaltung der Mitglieder
- Verwaltet werden sollen zwei verschiedene Arten von Versicherten:
  - Studierende
  - Angestellte
- Es soll möglich sein,
  - Versicherte anzulegen,
  - nach Versicherten anhand ihres Namens zu suchen und
  - die Daten einzelner Versicherter auf die Konsole auszugeben
- Je nach Art der Versicherten sind folgende Informationen relevant:

  Name, Vorname, Versicherungsnummer, Hochschule und Matrikelnummer, Name des Arbeitgebers und 1. Arbeitstag

Vorüberlegung: was brauchen wir?

- Eine Versicherungsverwaltung: eine Klasse, die zumindest Folgendes kann:
  - Versicherte anlegen, suchen
  - Die Versicherten können wir in einer linearen Liste verwalten
- Ein Versicherter/ein Mitglied der Versicherung: eine oder mehrere Klassen, die zumindest Folgendes kann/können:
  - Die eigenen Daten enthalten:
    - → Name, Vorname: String
    - → Mitgliedsnummer: int
    - → Hochschule: String, Matrikelnummer: int
    - → Arbeitgeber: String, 1. Arbeitstag: Datum (als Hilfsklasse)
  - Die eigenen Daten auf die Konsole ausgeben

So ungefähr könnte das Hauptprogramm aussehen

Die Klasse Versichertenverwaltung fehlt noch

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

    Versichertenverwaltung verwaltung = new Versichertenverwaltung();

    verwaltung.legeVersichertenAn(  ...  );

    ? mitglied = verwaltung.suche("Schmidt", "Harald");

    mitglied.druckeAufDieKonsole();
```

Welcher Typ / welche Klasse?

Methode der Klasse des Mitglieds

Hier wird ein Versicherter übergeben

Alles in einer Klasse: die Attribute

- Ein Versicherter/ein Mitglied der Versicherung: eine Klasse, die zumindest Folgendes kann:
  - …
  - Es sollte Felder geben für:
    - → Name, Vorname: String
    - → Mitgliedsnummer: int
    - → Hochschule: String, Matrikelnummer: int
    - → Arbeitgeber: String, 1. Arbeitstag: Datum (als Hilfsklasse)

```
public class Versicherter {

    private String name, vorname;
    private int mitgliedsnummer;
    // nur für den Studenten
    private String hochschule;
    private int matrikelnummer;
    // nur für den Angestellten
    private String nameDesArbeitgebers;
    private Datum ersterArbeitstag;
```

Alles in einer Klasse: das Hauptprogramm

```
public static void main(String[] args) {
    Versichertenverwaltung verwaltung = new Versichertenverwaltung();

    verwaltung.legeVersichertenAn(
            new Versicherter("Dampf", "Hans", 1, "HS MA", 1720001) );
    verwaltung.legeVersichertenAn(
            new Versicherter("Schmidt", "Harald", 2, "SAT 1", new Datum(1, 1, 2011)) );

    Versicherter mitglied = verwaltung.suche("Dampf", "Hans");
    mitglied.druckeAufDieKonsole();

    mitglied = verwaltung.suche("Schmidt", "Harald");
    mitglied.druckeAufDieKonsole();
}
```

Methode der Verwaltungsklasse

ein Konstruktor für einen Studenten

ein Konstruktor für einen Angestellten

Methode der Verwaltungsklasse

Alles in einer Klasse: die Verwaltungsklasse

```
public class Versichertenverwaltung {

    private List versicherte = new List();

    public void legeVersichertenAn( Versicherter versicherter ) {
        versicherte.addFirst(versicherter);
    }

    public Versicherter suche( String nachname, String vorname ) {
        int i = 0;
        boolean found = false;
        while (i < versicherte.size() && !found) {
            found = versicherte.getAt(i).heißt(nachname, vorname);
            i++;
        }

        if (found)
            return versicherte.getAt(i - 1);
        else
            throw new PRException("Versicherter " + nachname + ", "
                    + vorname + " nicht gefunden");
    }
}
```

die Schleife könnte auch in die Klasse *List* ausgelagert werden

Alles in einer Klasse: die Konstruktoren

- Die Klasse *Versicherter* bekommt zwei Konstruktoren:
  - einer für Studenten
  - einer für Angestellte

```
public Versicherter( String name, String vorname, int mitgliedsnummer,
        String hochschule, int matrikelnummer ) {
    this.name = name;
    this.vorname = vorname;
    this.hochschule = hochschule;
    this.matrikelnummer = matrikelnummer;
    this.mitgliedsnummer = mitgliedsnummer;
}

public Versicherter( String name, String vorname, int mitgliedsnummer,
        String nameDesArbeitgebers, Datum ersterArbeitstag ) {
    this.name = name;
    this.vorname = vorname;
    this.nameDesArbeitgebers = nameDesArbeitgebers;
    this.ersterArbeitstag = ersterArbeitstag;
    this.mitgliedsnummer = mitgliedsnummer;
}
```

- Das ist *kein Problem*, weil beim Aufruf eines dieser Konstruktoren eindeutig festzustellen ist, welcher gemeint ist!

Alles in einer Klasse: die Methoden des *Versicherten*

```
public boolean heißt( String gesuchterName, String gesuchterVorname ) {
    return name.equals(gesuchterName) && vorname.equals(gesuchterVorname);
}

public void druckeAufDieKonsole() {
    println(vorname + " " + name);
    println("  Mitgliedsnummer: " + mitgliedsnummer);

    if (hochschule != null)
        println("  studiert an der: " + hochschule + ", Matrikelnummer: " + matrikelnummer);
    if (nameDesArbeitgebers != null) {
        print("  arbeitet bei: " + nameDesArbeitgebers + " seit ");
        ersterArbeitstag.druckeAufDieKonsole();
    }
}
```

Alles in einer Klasse: Bewertung

Man *kann* beide Versichertenarten in eine Klasse packen

- Das Programm (die beiden Klassen) sind überschaubar
- Das Programm funktioniert

Aber

- Was passiert, wenn wir eine neue Art von *Versicherten* hinzufügen wollen?
  Wir müssen die Klasse *Versicherter* erweitern:
  - Neue Attribute
  - Ein neuer Konstruktor (hoffentlich unterscheidbar) 👎
  - Neue Fallunterscheidung in der Methode „druckeAufDieKonsole()“ 👎

Alles in einer Klasse: Bewertung

```
public class Versicherter {

    private String name, vorname;
    private int mitgliedsnummer;
    // nur für den Studenten
    private String hochschule;
    private int matrikelnummer;
    // nur für den Angestellten
    private String nameDesArbeitgebers;
    private Datum ersterArbeitstag;

    public Versicherter( String name, String vorname, int mitgliedsnummer,
            String hochschule, int matrikelnummer ) {
        this.name = name;
        this.vorname = vorname;
        this.hochschule = hochschule;
        this.matrikelnummer = matrikelnummer;
        this.mitgliedsnummer = mitgliedsnummer;
    }

    public Versicherter( String name, String vorname, int mitgliedsnummer,
            String nameDesArbeitgebers, Datum ersterArbeitstag ) {
        this.name = name;
        this.vorname = vorname;
        this.nameDesArbeitgebers = nameDesArbeitgebers;
        this.ersterArbeitstag = ersterArbeitstag;
        this.mitgliedsnummer = mitgliedsnummer;
    }

    public boolean heißt( String gesuchterName, String gesuchterVorname ) {
        return name.equals(gesuchterName) && vorname.equals(gesuchterVorname);
    }

    public void druckeAufDieKonsole() {
        println(vorname + " " + name);
        println("  Mitgliedsnummer: " + mitgliedsnummer);

        if (hochschule != null)
            println("  studiert an der: " + hochschule + ", Matrikelnummer: " + matrikelnummer);
        if (nameDesArbeitgebers != null) {
            print("  arbeitet bei: " + nameDesArbeitgebers + " seit ");
            ersterArbeitstag.druckeAufDieKonsole();
        }
    }

}
```

neue Attribute

neuer Konstruktor

neue Fallunterscheidung

Hinzufügen einer neuen Art von *Versicherten*: Rentner

```
// nur für den Rentner
private float jahreAktiverMitgliedschaft;
```

```
public Versicherter( String name, String vorname, int mitgliedsnummer,
        float jahreAktiverMitgliedschaft ) {
    this.name = name;
    this.vorname = vorname;
    this.jahreAktiverMitgliedschaft = jahreAktiverMitgliedschaft;
    this.mitgliedsnummer = mitgliedsnummer;
}
```

```
if (jahreAktiverMitgliedschaft > 0)
    println("  aktive Jahre: " + jahreAktiverMitgliedschaft);
```

Hinzufügen einer neuen Art von *Versicherten*: Rentner

```
// nur für den Rentner
private float jahreAktiverMitgliedschaft;
```

Ändern an existierendem Code ist immer schlecht!

```
if (jahreAktiverMitgliedschaft > 0)
    println("  aktive Jahre: " + jahreAktiverMitgliedschaft);
```

Andere Lösung: anstelle einer einzigen Klasse…

- Ein Versicherter/ein Mitglied der Versicherung: eine Klasse, die zumindest Folgendes kann:
  - …
  - Es sollte Felder geben für:
    - → Name, Vorname: String
    - → Mitgliedsnummer: int
    - → Hochschule: String, Matrikelnummer: int
    - → Arbeitgeber: String, 1. Arbeitstag: Datum (als Hilfsklasse)
    - → Jahre aktiver Mitgliedschaft: float

```
public class Student {

    private String name, vorname;
    private int mitgliedsnummer;
    private String hochschule;
    private int matrikelnummer;
```

```
public class Rentner {

    private String name, vorname;
    private int mitgliedsnummer;
    private float jahreAktiverMitgliedschaft;
```

```
public class Angestellter {

    private String name, vorname;
    private int mitgliedsnummer;
    private String nameDesArbeitgebers;
    private Datum ersterArbeitstag;
```

Probleme! Lösungen?

- Wie verwalten wir die verschiedenen Versicherten?
  -> Bitte *nicht* in drei verschiedenen Listen…!
- Was (= welchen Typ) bekommen wir bei einer Suche zurück?

```
? mitglied = verwaltung.suche("Schmidt", "Harald");
```

→ Wir führen ein Interface „Versicherter“ ein, das von allen drei Versichertenarten implementiert wird

```
public interface Versicherter {

    boolean heißt( String gesuchterName, String gesuchterVorname );

    void druckeAufDieKonsole();

}
```

→ Die Suche liefert ein Objekt, das dieses Interface implementiert (wir müssen nicht wissen, ob Student, Angestellter oder Rentner)

```
Versicherter mitglied = verwaltung.suche("Schmidt", "Harald");
```

Konsequenz

- Die drei Versichertenklassen implementieren das Interface und bieten entsprechende Konstruktoren

```
public class Angestellter implements Versicherter {

    private String name, vorname;
    private int mitgliedsnummer;
    private String nameDesArbeitgebers;
    private Datum ersterArbeitstag;

    public Angestellter( String name, String vorname, int mitgliedsnummer,
            String nameDesArbeitgebers, Datum ersterArbeitstag ) {
        this.name = name;
        this.vorname = vorname;
```

```
public class Rentner implements Versicherter {

    private String name, vorname;
    private int mitgliedsnummer;
    private float jahreAktiverMitgliedschaft;

    public Rentner( String name, String vorname, int mitgliedsnummer,
            float jahreAktiverMitgliedschaft ) {
        this.name = name;
        this.vorname = vorname;
```

```
public class Student implements Versicherter {

    private String name, vorname;
    private int mitgliedsnummer;
    private String hochschule;
    private int matrikelnummer;

    public Student( String name, String vorname, int mitgliedsnummer,
            String hochschule, int matrikelnummer ) {
        this.name = name;
```

So könnte das vollständige Hauptprogramm aussehen

```
public static void main(String[] args) {
    Versichertenverwaltung verwaltung = new Versichertenverwaltung();

    verwaltung.legeVersichertenAn(
            new Student("Dampf", "Hans", 1, "HS MA", 1720001) );
    verwaltung.legeVersichertenAn(
            new Angestellter("Schmidt", "Harald", 2, "SAT 1", new Datum(1, 1, 2011)) );
    verwaltung.legeVersichertenAn(
            new Rentner("Gottschalk", "Thomas", 3, 22.5f) );

    Versicherter mitglied = verwaltung.suche("Dampf", "Hans");
    mitglied.druckeAufDieKonsole();

    mitglied = verwaltung.suche("Schmidt", "Harald");
    mitglied.druckeAufDieKonsole();
}
```

drei unterscheidbare und lesbare Konstruktoren für drei verschiedene Klassen

Die Implementierung des Studenten

```
public class Student implements Versicherter {

    private String name, vorname;
    private int mitgliedsnummer;
    private String hochschule;
    private int matrikelnummer;

    public Student( String name, String vorname, int mitgliedsnummer,
            String hochschule, int matrikelnummer ) {
        this.name = name;
        this.vorname = vorname;
        this.hochschule = hochschule;
        this.matrikelnummer = matrikelnummer;
        this.mitgliedsnummer = mitgliedsnummer;
    }

    public boolean heißt( String gesuchterName, String gesuchterVorname ) {
        return name.equals(gesuchterName) && vorname.equals(gesuchterVorname);
    }

    public void druckeAufDieKonsole() {
        println(vorname + " " + name);
        println("  Mitgliedsnummer: " + mitgliedsnummer);
        println("  studiert an der: " + hochschule + ", Matrikelnummer: " + matrikelnummer);
    }

}
```

Die Implementierung des Angestellten

```
public class Angestellter implements Versicherter {

    private String name, vorname;
    private int mitgliedsnummer;
    private String nameDesArbeitgebers;
    private Datum ersterArbeitstag;

    public Angestellter( String name, String vorname, int mitgliedsnummer,
            String nameDesArbeitgebers, Datum ersterArbeitstag ) {
        this.name = name;
        this.vorname = vorname;
        this.nameDesArbeitgebers = nameDesArbeitgebers;
        this.ersterArbeitstag = ersterArbeitstag;
        this.mitgliedsnummer = mitgliedsnummer;
    }

    public boolean heißt( String gesuchterName, String gesuchterVorname ) {
        return name.equals(gesuchterName) && vorname.equals(gesuchterVorname);
    }

    public void druckeAufDieKonsole() {
        println(vorname + " " + name);
        println("  Mitgliedsnummer: " + mitgliedsnummer);
        print("  arbeitet bei: " + nameDesArbeitgebers + " seit ");
        ersterArbeitstag.druckeAufDieKonsole();
        println();
    }

}
```

Methode der Datumsklasse

Die Implementierung des Rentners

```
public class Rentner implements Versicherter {

    private String name, vorname;
    private int mitgliedsnummer;
    private float jahreAktiverMitgliedschaft;

    public Rentner( String name, String vorname, int mitgliedsnummer,
            float jahreAktiverMitgliedschaft ) {
        this.name = name;
        this.vorname = vorname;
        this.jahreAktiverMitgliedschaft = jahreAktiverMitgliedschaft;
        this.mitgliedsnummer = mitgliedsnummer;
    }

    public boolean heißt( String gesuchterName, String gesuchterVorname ) {
        return name.equals(gesuchterName) && vorname.equals(gesuchterVorname);
    }

    public void druckeAufDieKonsole() {
        println(vorname + " " + name);
        println("  Mitgliedsnummer: " + mitgliedsnummer);
        println("  aktive Jahre: " + jahreAktiverMitgliedschaft);
    }
}
```

Nutzen von drei Klassen: Bewertung

Man *kann* die drei Arten von *Versicherten* in drei verschiedene Klassen packen

- Das Programm (die drei Klassen) sind gut überschaubar
- Das Programm funktioniert

Aber

- Was passiert, wenn wir eine neue Art von *Versicherten* hinzufügen wollen?
  Wir fügen eine neue Klasse nach dem Muster der bisherigen hinzu:
  - Keine Änderung von existierendem Code 👍
  - Ein lesbarer anderer Konstruktor 👍
- Was passiert, wenn wir Versicherten weitere Daten hinzufügen wollen?
  - Wir müssen das *in allen Klassen* tun 👎

Nutzen von drei Klassen: Bewertung

```java
public class Student implements Versicherter {

    private String name, vorname;
    private int mitgliedsnummer;
    private String hochschule;
    private int matrikelnummer;

    public Student( String name, String vorname, int mitgliedsnummer,
            String hochschule, int matrikelnummer ) {
        this.name = name;
        this.vorname = vorname;
        this.hochschule = hochschule;
        this.matrikelnummer = matrikelnummer;
        this.mitgliedsnummer = mitgliedsnummer;
    }

    public boolean heißt( String gesuchterName, String gesuchterVorname ) {
        return name.equals(gesuchterName) && vorname.equals(gesuchterVorname);
    }

    public void druckeAufDieKonsole() {
        println(vorname + " " + name);
        println("  Mitgliedsnummer: " + mitgliedsnummer);
        println("  studiert an der: " + hochschule + ", Matrikelnummer: " + matrikelnu
    }

}
```

```java
public class Angestellter implements Versicherter {

    private String name, vorname;
    private int mitgliedsnummer;
    private String nameDesArbeitgebers;
    private Datum ersterArbeitstag;

    public Angestellter( String name, String vorname, int mitgliedsnummer,
            String nameDesArbeitgebers, Datum ersterArbeitstag ) {
        this.name = name;
        this.vorname = vorname;
        this.nameDesArbeitgebers = nameDesArbeitgebers;
        this.ersterArbeitstag = ersterArbeitstag;
        this.mitgliedsnummer = mitgliedsnummer;
    }

    public boolean heißt( String gesuchterName, String gesuchterVorname ) {
        return name.equals(gesuchterName) && vorname.equals(gesuchterVorname);
    }

    public void druckeAufDieKonsole() {
        println(vorname + " " + name);
        println("  Mitgliedsnummer: " + mitgliedsnummer);
        print("  arbeitet bei: " + nameDesArbeitgebers + " seit ");
        ersterArbeitstag.druckeAufDieKonsole();
        println();
    }
```

```java
public class Rentner implements Versicherter {

    private String name, vorname;
    private int mitgliedsnummer;
    private float jahreAktiverMitgliedschaft;

    public Rentner( String name, String vorname, int mitgliedsnummer,
            float jahreAktiverMitgliedschaft ) {
        this.name = name;
        this.vorname = vorname;
        this.jahreAktiverMitgliedschaft = jahreAktiverMitgliedschaft;
        this.mitgliedsnummer = mitgliedsnummer;
    }

    public boolean heißt( String gesuchterName, String gesuchterVorname ) {
        return name.equals(gesuchterName) && vorname.equals(gesuchterVorname);
    }

    public void druckeAufDieKonsole() {
        println(vorname + " " + name);
        println("  Mitgliedsnummer: " + mitgliedsnummer);
        println("  aktive Jahre: " + jahreAktiverMitgliedschaft);
    }

}
```

Die markierten Stellen sind gleich für alle drei Klassen

→ Redundanter Programmcode ist immer schlecht!

Ein neues Attribut für die Versicherten: Beispiel *Geburtsdatum*

```
public class Student implements Versicherter {

    private String name, vorname;
    private int mitgliedsnummer;
    private Datum geburtsdatum;
    private String hochschule;
    private int matrikelnummer;

    public Student( String name, String vorname, int mitgliedsnummer,
            Datum geburtsdatum,
            String hochschule, int matrikelnummer ) {
        this.name = name;
        this.vorname = vorname;
        this.geburtsdatum = geburtsdatum;
        this.hochschule = hochschule;
        this.matrikelnummer = matrikelnummer;
        this.mitgliedsnummer = mitgliedsnummer;
    }

    public boolean heißt( String gesuchterName, String gesuchterVorname ) {
        return name.equals(gesuchterName) && vorname.equals(gesuchterVorname);
    }

    public void druckeAufDieKonsole() {
        println(vorname + " " + name);
        println("  Mitgliedsnummer: " + mitgliedsnummer);
        println("  ist geboren am ");
        geburtsdatum.druckeAufDieKonsole();
        println("  studiert an der: " + hochschule + ", Matrikelnummer: " + matrikelnummer);
    }

}
```

Attribut
(in allen 3 Klassen einfügen)

Zuweisung
(in allen 3 Klassen einfügen)

Ausgabe
(in allen 3 Klassen einfügen)

Ein neues Attribut für die Versicherten: Beispiel *Geburtsdatum*

```
public class Student implements Versicherter {

    private String name, vorname;
    private int mitgliedsnummer;
    private Datum geburtsdatum;
    private String hochschule;
    private int matrikelnummer;

    public Student( String name, String vorname, int mitgliedsnummer,
                    Datum geburtsdatum, String hochschule, int matrikelnummer ) {
        this.name = name;
        this.vorname = vorname;
        this.mitgliedsnummer = mitgliedsnummer;
        this.geburtsdatum = geburtsdatum;
        this.hochschule = hochschule;
        this.matrikelnummer = matrikelnummer;
    }

    public boolean heißt( String gesuchterName, String gesuchterVorname ) {
        return name.equals(gesuchterName) && vorname.equals(gesuchterVorname);
    }

    public void druckeAufDieKonsole() {
        println(vorname + " " + name);
        println("  Mitgliedsnummer: " + mitgliedsnummer);
        println("  ist geboren am ");
        geburtsdatum.druckeAufDieKonsole();
        println("  studiert an der: " + hochschule + ", Matrikelnummer: " + matrikelnummer);
    }

}
```

Attribut
(in allen 3 Klassen einfügen)

Redundanter Code ist immer schlecht!

einfügen)

Ausgabe
(in allen 3 Klassen einfügen)

Fazit

Eine Lösung mit *Redundanz* ist sehr unbefriedigend:

- Man schreibt (kopiert) oft den gleichen Code
- Wenn sich ein Fehler eingeschlichen hat, muss er mehrfach korrigiert werden: in allen Klassen!
- Wenn etwas ergänzt werden muss, muss es mehrfach ergänzt werden: in allen Klassen!

Bessere Lösung:

Man erstellt *eine Klasse* mit allen *gemeinsamen* Eigenschaften (statt eines Interface') und erstellt dann *spezielle Klassen* (*Student*, *Rentner*, *Angestellter*), die diese Gemeinsamkeiten dann um die "Spezialitäten" *ergänzen* / *erweitern*

In Java benötigen wir dazu die Schlüsselwörter

- extends
- super

Die neue Klasse mit den Gemeinsamkeiten der bisherigen drei Klassen

```
public class Versicherter {

    private String name, vorname;
    private int mitgliedsnummer;

    public Versicherter( String name, String vorname, int mitgliedsnummer ) {
        this.name = name;
        this.vorname = vorname;
        this.mitgliedsnummer = mitgliedsnummer;
    }

    public boolean heißt( String gesuchterName, String gesuchterVorname ) {
        return name.equals(gesuchterName) && vorname.equals(gesuchterVorname);
    }

    public void druckeAufDieKonsole() {
        println(vorname + " " + name);
        println("  Mitgliedsnummer: " + mitgliedsnummer);
    }

}
```

Die neue Klasse mit den Gemeinsamkeiten der bisherigen drei Klassen

```
public class Versicherter {

    private String name, vorname;
    private int mitgliedsnummer;

    public Versicherter( String name, String vorname, int mitgliedsnummer ) {
        this.name = name;
        this.vorname = vorname;
        this.mitgliedsnummer = mitgliedsnummer;
    }

    public boolean heißt( String gesuchterNa
        return name.equals(gesuchterName) &&
    }

    public void druckeAufDieKonsole() {
        println(vorname + " " + name);
        println("  Mitgliedsnummer: " + mitg
    }

}
```

```
public class Rentner implements Versicherter {

    private String name, vorname;
    private int mitgliedsnummer;
    private float jahreAktiverMitgliedschaft;

    public Rentner( String name, String vorname, int mitgliedsnummer,
            float jahreAktiverMitgliedschaft ) {
        this.name = name;
        this.vorname = vorname;
        this.jahreAktiverMitgliedschaft = jahreAktiverMitgliedschaft;
        this.mitgliedsnummer = mitgliedsnummer;
    }

    public boolean heißt( String gesuchterName, String gesuchterVorname ) {
        return name.equals(gesuchterName) && vorname.equals(gesuchterVorname);
    }

    public void druckeAufDieKonsole() {
        println(vorname + " " + name);
        println("  Mitgliedsnummer: " + mitgliedsnummer);
        println("  aktive Jahre: " + jahreAktiverMitgliedschaft);
    }
}
```

Die neue Studenten-Klasse

Die Klasse *Student* "**erweitert**" die existierende Klasse *Versicherter*

```
public class Student extends Versicherter {

    private String hochschule;
    private int matrikelnummer;

    public Student( String name, String vorname, int mitgliedsnummer,
            Datum geburtsdatum,
            String hochschule, int matrikelnummer ) {
        super(name, vorname, mitgliedsnummer);
        this.hochschule = hochschule;
        this.matrikelnummer = matrikelnummer;
    }

    public void druckeAufDieKonsole() {
        super.druckeAufDieKonsole();
        println(" studiert an der: " + hochschule + ", Matrikelnummer: " + matrikelnummer);
    }

}
```

Ggü. dem *Versicherten* kommen 2 neue Felder *hinzu*

Aufgerufen wird zunächst der Konstruktor des *Versicherten* (dazu später mehr)

Aufgerufen wird zunächst die Methode des *Versicherten* (ein solcher Aufruf muss nicht erfolgen)

Ablauf beim Aufruf der „geerbten“ Methode „heißt“ 1/2

- Bei der *Übersetzung* der *main*-Methode prüft der Compiler für jeden Methodenaufruf, ob für den ***statischen Typ*** *Versicherter* der Variablen *mitglied* mindestens eine Methode auf die angegebenen aktuellen Parameter anwendbar ist, dazu
  1. bestimmt er die (statischen) Typen des *mitglied*s und aller Parameter
  2. stellt er fest, ob in der Klasse *Versicherter* des *mitglied*s (oder in einer von deren Oberklassen) eine Methode existiert, zu deren Parametertypen die Typen der aktuellen Parameter zuweisbar sind

```
public Versicherter suche( String nachname, String vorname ) {
    int i = 0;
    boolean found = false;
    while (i < versicherte.size() && !found) {
        Versicherter mitglied = versicherte.getAt(i);
        found = mitglied.heißt(nachname, vorname);
        i++;
    }
```

statischer Typ der Variable

passende Methode im statischen Typ

```
public class Versicherter {

    // ...

    public boolean heißt( String gesuchterName, String gesuchterVorname
        return name.equals(gesuchterName) && vorname.equals(gesuchterVo
    }
```

Falls nein: Compilerfehler
Falls ja: Bytecode erzeugen (*inkl. eventueller Casts* der Parameter)

Ablauf beim Aufruf der „geerbten“ Methode „heißt“ 2/2

- Bei der *Programmausführung* bestimmt die virtuelle Maschine anhand des ***dynamischen Typs*** (z.B. *Student*) der Variablen *mitglied* die passendste Methode, dazu
  1. bestimmt sie den dynamischen Typ (z.B. *Student*) der Variablen *mitglied* (die Parameter werden mit dem vom Compiler bestimmten/gecasteten Typ verwendet; dazu *später* mehr)
  2. wählt sie die speziellste passende Methode aus und führt diese aus

angenommen, es liegt ein *Student* vor

```
public Versicherter suche( String nachname, String vorname ) {
    int i = 0;
    boolean found = false;
    while (i < versicherte.size() && !found) {
        Versicherter mitglied = versicherte.getAt(i);
        found = mitglied.heißt(nachname, vorname);
        i++;
    }
```

der dynamische Typ von *mitglied* ist dann *Student*

In der Klasse *Student* gibt es keine passende Methode

```
public class Student extends Versicherter {

    private String hochschule;
    private int matrikelnummer;

    public Student( String name, String vorname, int mitgliedsnummer,
                    String hochschule, int matrikelnummer ) {
        super(name, vorname, mitgliedsnummer);
        this.hochschule = hochschule;
        this.matrikelnummer = matrikelnummer;

    public void druckeAufDieKonsole() {
        super.druckeAufDieKonsole();
        println("  studiert an der: " + hochschule + ", Matrikelnummer: " + matrikelnummer);
    }

}
```

Die Suche wird in der Oberklasse von *Student* fortgesetzt

```
public class Versicherter {

    // ...

    public boolean heißt( String gesuchterName, String gesuchterVorname ) {
        return name.equals(gesuchterName) && vorname.equals(gesuchterVorname);
    }
```

Hier gibt es eine passende Methode

Feststellung 1

Wenn eine Klasse eine andere Klasse *erweitert*, dann bedeutet das, die erweiterte Klasse *kann alles, was die ursprüngliche kann* plus Einiges mehr!
Daher wird das Schlüsselwort *extends* benutzt

```
public class Student extends Versicherter {

    private String hochschule;
    private int matrikelnummer;
```

```
public class Angestellter extends Versicherter {

    private String nameDesArbeitgebers;
    private Datum ersterArbeitstag;
```

```
public class Rentner extends Versicherter {

    private float jahreAktiverMitgliedschaft;
```

Feststellung 2

Ein Objekt der erweiterten Klasse *ist* gleichzeitig *ein* spezialisiertes Objekt der ursprünglichen Klasse!

Die Methode „suche“ liefert in diesem Fall ein Studenten-Objekt als Ergebnis

```
Versicherter mitglied = verwaltung.suche("Dampf", "Hans");
mitglied.druckeAufDieKonsole();
```

Ein Objekt der erweiterten Klasse kann überall dort verwendet werden, wo ein Objekt der ursprünglichen Klasse verwendet werden kann!

Feststellung 3

In Java heißt das:
Man kann die Methoden der ursprünglichen Klasse auch auf die Objekte der neuen / spezialisierten Klasse(n) anwenden!

```
public class Versicherter {

    private String name, vorname;
    private int mitgliedsnummer;

    public Versicherter( String name, String vor
        this.name = name;
        this.vorname = vorname;
        this.mitgliedsnummer = mitgliedsnummer;
    }

    public boolean heißt( String gesuchterName, String gesuchterVorname ) {
        return name.equals(gesuchterName) && vorname.equals(gesuchterVorname);
    }

    public void druckeAufDieKonsole() {
        println(vorname + " " + name);
        println("  Mitgliedsnummer: " + mitgliedsnummer);
    }

}
```

```
public Versicherter suche( String nachname, String vorname ) {
    int i = 0;
    boolean found = false;
    while (i < versicherte.size() && !found) {
        found = versicherte.getAt(i).heißt(nachname, vorname);
        i++;
    }
```

In der Oberklasse schon

Die Methode “getAt“ liefert Objekte der Klassen Student, Angestellter, Rentner als Ergebnis.

Keine dieser Klassen besitzt eine *eigene* Methode „heißt“

Die Verwaltung bleibt unverändert…

Der Liste ist "egal", welche Art von Versicherten abgespeichert werden

Der Methode ist "egal", welche Art von Versicherten übergeben wird

```
public class Versichertenverwaltung {

    private List versicherte = new List();

    public void legeVersichertenAn( Versicherter versicherter ) {
        versicherte.addFirst(versicherter);
    }

    public Versicherter suche( String nachname, String vorname ) {
        int i = 0;
        boolean found = false;
        while (i < versicherte.size() && !found) {
            found = versicherte.getAt(i).heißt(nachname, vorname);
            i++;
        }

        if (found)
            return versicherte.getAt(i - 1);
        else
            throw new PRException("Versicherter
                    + vorname + " nicht gefunde
    }

}
```

Das können alle (speziellen) Versicherten

Die Oberklasse "Versicherter" kann verwendet werden wie zuvor das (gleichnamige) Interface

Die Angestellten-Klasse

```
public class Angestellter extends Versicherter {

    private String nameDesArbeitgebers;
    private Datum ersterArbeitstag;

    public Angestellter( String name, String vorname, int mitgliedsnummer,
            String nameDesArbeitgebers, Datum ersterArbeitstag ) {
        super(name, vorname, mitgliedsnummer);
        this.nameDesArbeitgebers = nameDesArbeitgebers;
        this.ersterArbeitstag = ersterArbeitstag;
    }

    public void druckeAufDieKonsole() {
        super.druckeAufDieKonsole();
        print("  arbeitet bei: " + nameDesArbeitgebers + " seit ");
        ersterArbeitstag.druckeAufDieKonsole();
        println();
    }

}
```

Die Rentner-Klasse

```
public class Rentner extends Versicherter {

    private float jahreAktiverMitgliedschaft;

    public Rentner( String name, String vorname, int mitgliedsnummer,
            float jahreAktiverMitgliedschaft ) {
        super(name, vorname, mitgliedsnummer);
        this.jahreAktiverMitgliedschaft = jahreAktiverMitgliedschaft;
    }

    public void druckeAufDieKonsole() {
        super.druckeAufDieKonsole();
        println("  aktive Jahre: " + jahreAktiverMitgliedschaft);
    }

}
```

Der Vollständigkeit halber: die Datum-Klasse

```
public class Datum {

    private int tag, monat, jahr;

    public Datum( int tag, int monat, int jahr ) {
        this.tag = tag;
        this.monat = monat;
        this.jahr = jahr;
    }

    public void druckeAufDieKonsole() {
        print(tag + "." + monat + "." + jahr);
    }

}
```

Zusammenfassung

- Die Aufgabenstellung fordert mehrere *Spezialfälle* von Versicherten mit *Gemeinsamkeiten* und *Unterschieden*.
- *Alle* irgendwo benötigten Felder *in eine* Klasse zu packen (= Vereinigungsmenge aller Spezialfälle) ist schlecht
  - bei Änderungen,
  - bei Ergänzungen
  - etc.
- *Getrennte* Klassen zu bilden, in denen auch alle gemeinsamen Felder wiederholt werden, ist auch schlecht, weil viel redundanter Programmcode entsteht, der bei Änderungen / Verbesserungen mehrfach zu behandeln ist.
- Die Lösung:
  Gemeinsame Anteile werden in eine gemeinsame ***Ober-Klasse*** ausgegliedert, die dann von speziellen ***Unterklassen*** um deren Unterschiede *erweitert* wird.

Begriffe der Objektorientierung 1/2

- Gemeinsame Felder mehrerer (spezieller) Klassen können in einer (allgemeinen) Klasse implementiert werden, die dann von den (speziellen) Klassen um deren jeweilige (spezielle) Felder ***erweitert*** wird
- Man nennt die allgemeinere Klasse mit den Gemeinsamkeiten ***Oberklasse*** (manchmal auch *Basisklasse*) der speziellen Klassen
  Im Englischen heißt „Oberklasse“ *super class*, daher das Schlüsselwort *super* in Java
  Beispiel:
  Versicherter ist Oberklasse von *Student*
- Man nennt die speziellen Klassen ***Unterklassen*** der allgemeinen Klasse
  Beispiel *Student* ist Unterklasse von *Versicherter*
  So wie ein *Hund* ein spezielles *Säugetier* ist…

Begriffe der Objektorientierung 2/2

- Die speziellen Klassen ***erweitern*** oder ***spezialisieren*** ihre Oberklasse; Beispiel:
  Die Klasse *Student* erweitert die Klasse *Versicherter* um den Namen seiner Hochschule und seine Matrikelnummer
- Die speziellen Klassen ***erben*** von ihrer Oberklasse oder sie ***beerben*** ihre Oberklasse; Beispiel:
  Die Klasse *Student* erbt alle Felder ihrer Oberklasse *Versicherter*: Name, Vorname und Mitgliedsnummer
- Im Englischen spricht man von einer ***is-a – Beziehung***:
  Ein Objekt der Unterklasse *ist* (auch) *ein* (englisch: *is a*) Objekt der Oberklasse und kann genau so verwendet werden
- Von unten nach oben gesehen handelt es sich um eine ***Abstraktion***:
  Die Oberklasse abstrahiert von ihren speziellen Unterklassen
- *Man sollte nicht sagen*: Die Oberklasse *vererbt* etwas an ihre Unterklassen, weil die Initiative *nicht* von der Oberklasse ausgeht!

Vorteil von *Erweiterung* (Vererbung)

Objekte einer Klasse können überall eingesetzt werden, wo auch Objekte ihrer Oberklasse(n) eingesetzt werden können [Liskovsches Substitutionsprinzip]

Im Beispiel:
Objekte der Klassen *Student*, *Angestellter* oder *Rentner* können überall eingesetzt werden, wo Objekte der Klasse *Versicherter* eingesetzt werden können

→ Alles, was ich mit einem Objekt der Klasse *Versicherter* tun kann, kann ich auch mit einem *Studenten*, einem *Angestellten*, oder einem *Rentner* tun.
Oder mit zukünftigen weiteren Unterklassen...