252-0027

Einführung in die Programmierung

7.0 Vererbung

Thomas R. Gross

Department Informatik
ETH Zürich

Übersicht

6.1 Einleitung – Datenstrukturen mit Verknüpfungen

6.2 Entwurf von (abgekapselten) Klassen

6.3 Hinweise (und Regeln) für verständliche Programme

6.4 Objektexemplare in Programmen

6.5 (Mehr) Sichtbarkeit

Packages und Sichtbarkeit

- **In Java gibt es die folgenden Zugriffsmodifizierer ("access modifiers"):**
  - **`public` : Sichtbar für alle anderen Klassen (nach Import).**
  - **`private` : Sichtbar nur in dieser Klasse (und ggf. in eingeschlossenen Klassen/Typen – später mehr).**
  - **`protected` : Sichtbar nur in dieser Klasse, allen Unterklassen der Klasse, und allen anderen Klassen/Typen die in dieser Package deklariert sind.**
  - **default (package): Sichtbar in dieser Klasse und allen anderen Klassen/Typen die in dieser Package deklariert sind.**

Packages und Sichtbarkeit

- In Java gibt es die folgenden Zugriffsmodifizierer ("access modifiers"):
  - `public`: Sichtbar für alle anderen Klassen (nach Import).
  - `private`: Sichtbar nur in dieser Klasse (und ggf. in eingeschlossenen Klassen/Typen – später mehr).
  - `protected`: Sichtbar nur in dieser Klasse, allen Unterklassen der Klasse, und allen anderen Klassen/Typen die in dieser Package deklariert sind.
  - default (package): Sichtbar in dieser Klasse und allen anderen Klassen/Typen die in dieser Package deklariert sind.

Packages und Sichtbarkeit

- **Damit ein Attribut/eine Methode die default Sichtbarkeit hat brauchen Sie *keinen* Modifizierer anzugeben.**

```
package pacman.model;
public class Sprite {
    int points;       // default: visible to pacman.model.*
    String name;      // default: visible to pacman.model.*
}
```

```
public class Xray {
   int direction;     // default: visible in default package
   String name;       // default: visible in default package
}
```

Packages und Sichtbarkeit

- In einer Datei Problem.java

```
public class Problem {
    public static void main(String[] args) {
        …
    }
}
class SubProblem1 { … }
class SubProblem2 { … }
```

Attribute, Methoden und (static) Variable

In einer Datei Problem.java

```
public class Problem {
 public static void main(
       String[] args) {
  SubProblem s = new SubProblem();
  int a = s.x;
  a = s.y;
  a = s.z;
  }
 }
```

Fortsetzung Datei Problem.java

```
class SubProblem {
  int x;
  public int y;
  protected int z;
}
```

Attribute, Methoden und (static) Variable

In einer Datei Problem.java

```
public class Problem {
 public static void main(
        String[] args) {
  SubProblem s = new SubProblem();
  int a = s.f();
  a = s.g();
  a = s.h();
  }
 }
```

Fortsetzung Datei Problem.java

```
class SubProblem {
  int f() { … }
  public int g() { … }
  protected int h() {
     …
  }
}
```

Attribute, Methoden und (static) Variable

In einer Datei Problem.java

```
public class Problem {
 public static void main(
        String[] args) {
  SubProblem s = new SubProblem();
  int a = s.v; // prohibited
 }
}
```

Fortsetzung Datei Problem.java

```
class SubProblem {
  private int v;
}
```

Attribute, Methoden und (static) Variable

In einer Datei Problem.java

```
public class Problem {
 public static void main(
        String[] args) {
  SubProblem s = new SubProblem();
  int a = s.f(); //prohibited
 }
}
```

Fortsetzung Datei Problem.java

```
class SubProblem {
  private int f() {
      …
  }
}
```

Attribute, Methoden und (static) Variable

In einer Datei Problem.java

```
public class Problem {
 static int a;
 public static void main(
        String[] args) {
  SubProblem s = new SubProblem();
  a = s.f();
  a = s.g();
  }
 }
```

Fortsetzung Datei Problem.java

```
class SubProblem {
  int f() { … }
  public int g() { … }
}
```

Ein paar Fragen

- Die Attribute von `ListNode` sind nicht `private`? Ist das guter oder schlechter Stil?

Antwort:

- Das kann man vertreten da `LinkedIntList` der einzige Klient von `ListNode` ist
  - Andere Programme manipulieren nicht die `ListNode` Objekte sondern arbeiten mit Methoden der `LinkedIntList` Klasse.
  - Es gibt in Java noch bessere Möglichkeiten, solche Klassen zu organisieren, aber die lernen wir erst später kennen.

ListNode und LinkedIntList

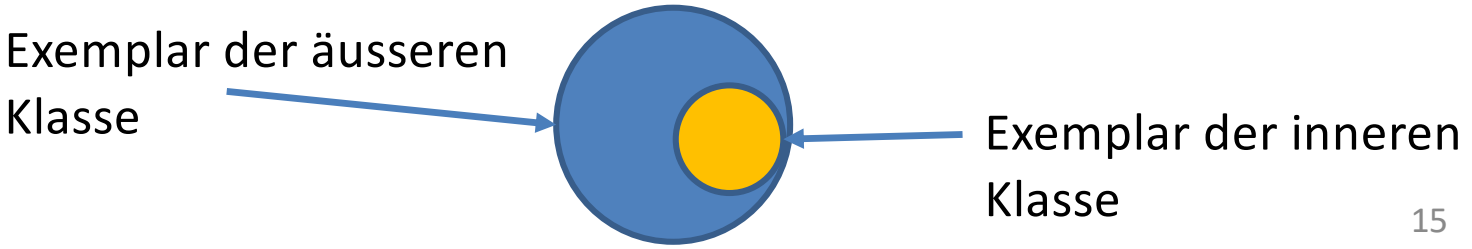
- *Eine* Lösung:
  - `LinkedIntList` ist eine `public` Klasse in einer Package
    - Kann nach Import verwendet werden
    - Package hat Namen
  - `ListNode` ist eine Klasse in der selben Package, aber nicht `public`
    - Kann von `LinkedIntList` verwendet werden, nicht von anderen Klienten
- Es gibt noch weitere Lösungen

Nested classes

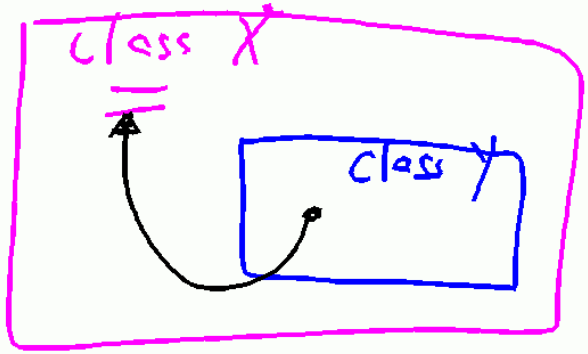
- **Geschachtelte Klasse (“nested class”): Eine Klasse die innerhalb einer anderen Klasse definiert ist.**
  - **Können als `static` oder non-static (default) Klassen definiert werden**
  - **non-static nested classes heissen *inner classes* (innere Klasse)**
- **Wir betrachten hier zuerst die non-static Klassen, d.h. die inneren Klassen ("inner class")**

Inner classes

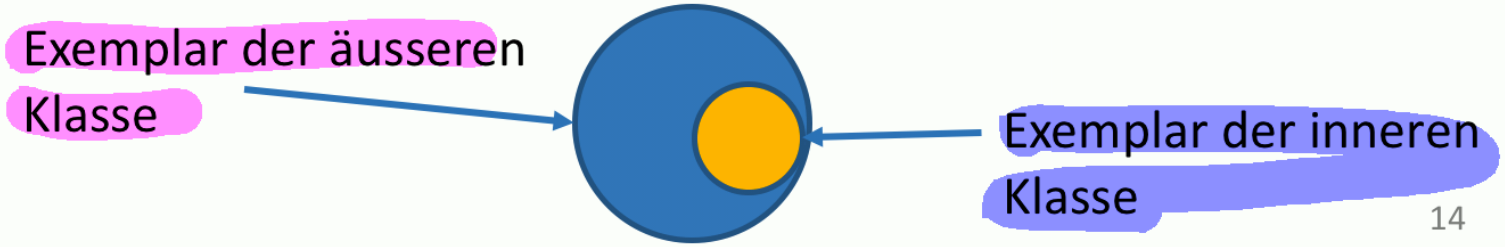
- **Warum:**
  - **Innere Klassen sind nicht sichtbar für andere Klassen (Abkapselung)**
  - **Innere Objekte können die Attribute des äusseren Objekts lesen/modifizieren**
- **Aber:**
  - **Exemplare der inneren Klasse existieren nur *innerhalb* eines Exemplars der sie umschliessenden (outer) Klasse**



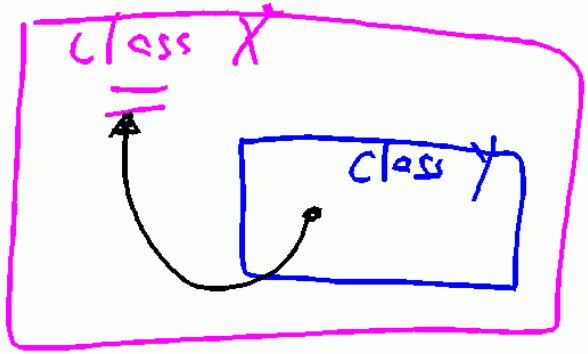
Inner classes



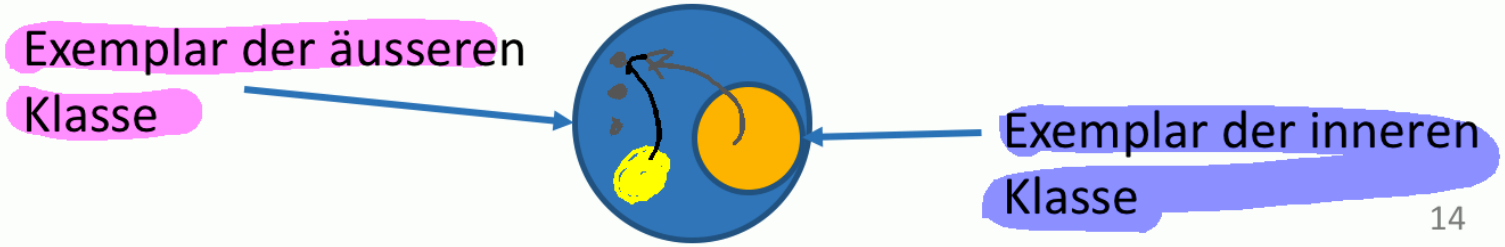
- **Warum:**
  - **Innere Klassen sind nicht sichtbar für andere Klassen (Abkapselung)**
  - **Innere Objekte können die Attribute des äusseren Objekts lesen/modifizieren**
- **Aber:**
  - **Exemplare der inneren Klasse existieren nur *innerhalb* eines Exemplars der sie umschliessenden (outer) Klasse**



Inner classes



- **Warum:**
  - **Innere Klassen sind nicht sichtbar für andere Klassen (Abkapselung)**
  - **Innere Objekte können die Attribute des äusseren Objekts lesen/modifizieren**
- **Aber:**
  - **Exemplare der inneren Klasse existieren nur *innerhalb* eines Exemplars der sie umschliessenden (outer) Klasse**



Inner Class Syntax

```
// outer (enclosing) class
public class outerName {
     ...

     // inner class
     private class innerName {
          ...
     }
}
```

Inner Class Syntax

```
// outer (enclosing) class
public class outerName {
     ...

     // inner class
     class innerName {
          ...
     }
}
```

Innere Klassen

- **Wenn `private`: Nur der Code in dieser Klasse kann die innere Klasse sehen oder Exemplare konstruieren.**
  - **Ohne private: Package**
- **Jedes innere Objekt ist mit dem äusseren Objekt, welches es konstruierte, verbunden und kann so die Attribute/Methoden des äusseren Objektes lesen/modifizieren/aufrufen.**
  - **Wenn nötig kann das äussere Objekt über die Referenzvariable OuterClassName . `this` erreicht werden**
  - **Denn `this` bezieht sich auf das innere Objekt**

LinkedIntList/ListNode Beispiel

```
class LinkedIntList {
    private ListNode front;

  class ListNode {
    int data;
    ListNode next;

    ListNode() { }
    ListNode(int v) { }
    ListNode(int v, ListNode c){}
    public String toString() {}
  } // ListNode

  public String toString() {}

  //Adds value to end of List
  void add (int value) { }
  int remove() { }
} // LinkedIntList
```

LinkedIntList/ListNode Beispiel

```
// Beispiel Inner Class

public class InnerExample {
  public static void main(String[] args) {
    new ListExample().run();
  }
}

class ListExample {
  void run() {
    LinkedIntList list =
            new LinkedIntList();
    list.add(1);
    System.out.println(list);
  }
}
```

Output: [1]

Innere und äussere Objekte

- **Exemplare der inneren Klasse existieren nur *innerhalb* eines Exemplars der sie umschliessenden (outer) Klasse**
- **Wenn es kein äusseres Objekt gibt, dann kann es auch kein inneres geben**
- **Einschränkung für `static` Methoden**

Ohne umschliessendes Exemplar

```
public class Inner2 {
  public static void main(String[] args) {
    System.out.println(new InnerClass().foo);
  }
  class InnerClass {
    int foo = 2;
  }
```

```
javac Inner2.java
Inner2.java:3: error: non-static variable this cannot be
referenced from a static context
        System.out.println(new InnerClass().foo);
                                           ^
```

Nested Classes als Namensraum

- **Wir wollen die Schachtelung (in Klasse `Outer`) nur verwenden um Namenskonflikte (für Klasse `Inner`) zu vermeiden**

- **Dann wollen wir evtl. in einem Klienten Exemplare einer solchen Klasse *Inner* konstruieren**
  - **Ohne Examplare der Klasse *Outer* zu konstruieren**

`static` nested classes

- Das Keyword `static` wird benutzt um zu zeigen dass diese innere Klasse "zur äusseren Klasse" gehört und wir *kein* Exemplar von `Outer` brauchen

```
Outer.Inner myHandle = new Outer.Inner();
```

- Fast wie eine `static` Variable – existiert in der Klasse, nicht für jedes Exemplar
- Bringt nicht mehr als Packages

`static` nested classes

- Das Keyword `static` wird benutzt um zu zeigen dass diese innere Klasse "zur äusseren Klasse" gehört und wir *kein* Exemplar von `Outer` brauchen

```
Outer.Inner myHandle = new Outer.Inner();
```

Name da Klasse

- Fast wie eine `static` Variable – existiert in der Klasse, nicht für jedes Exemplar
- Bringt nicht mehr als Packages

Mit `static`, ohne umschliessendes Exemplar

```
public class Inner2 {
  public static void main(String[] args) {
    System.out.println(new InnerClass().foo);
  }
  static class InnerClass {
    int foo = 2;
  }
}
```

- **Übung: Ändern Sie die `LinkedIntList` so dass eine innere Klasse `ListNode` verwendet wird.**

Zusammenfassung

- **Wichtig ist dass Ihre Programme nicht auf beliebige Attribute zugreifen**
  - **z.B. soll `LinkedIntList` der einzige Klient sein**
- **`private` schützt vor unerwünschten Zugriffen**
- **Wir arbeiten fürs erste mit der default Package**
  - **Default Sichtbarkeit (innerhalb der Package)**

252-0027

Einführung in die Programmierung

7.0 Vererbung

Thomas R. Gross

Department Informatik
ETH Zürich

Übersicht

7.1 Einleitung

7.2 Neue Klassen aus existierenden Klassen

7.3 Vererbung und Konstruktoren

7.4 Selektiv Verhalten (von Objekten) festlegen

7.5 Klasse Object

7.6 Polymorphismus

Software Entwicklung

- **Programmieren ist anspruchsvoll**
  - **Es ist leicht Fehler zu machen**
  - **`LinkedIntList` Aufgaben**

- **Wir würden gerne Software wiederverwenden**
  - **Auf früheren Lösungen aufbauen**
  - **Von `LinkedIntList` zu `DoubleLinkedIntList`**
  - **Von `LinkedIntList` zu `LinkedBooleanList`, `LinkedPersonList`, ….**

Software Entwicklung

- Programmieren ist anspruchsvoll
  - Es ist leicht Fehler zu machen
  - `LinkedIntList` Aufgaben

- Wir würden gerne Software wiederverwenden
  - Auf früheren Lösungen aufbauen
  - Von `LinkedIntList` zu `DoubleLinkedIntList` ← 1.
  - Von `LinkedIntList` zu `LinkedBooleanList`, `LinkedPersonList`, ….

> später

Software Entwicklung

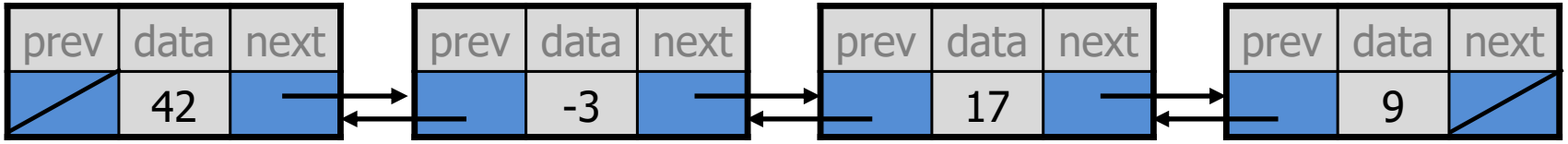
- **Programmieren ist anspruchsvoll**
  - **Es ist leicht Fehler zu machen**
  - **`LinkedIntList` Aufgaben**

- **Wir würden gerne Software wiederverwenden**
  - **Auf früheren Lösungen aufbauen**
  - **Von `LinkedIntList` zu `DoubleLinkedIntList`**
  - **Von** `LinkedIntList` **zu** `LinkedBooleanList`, `LinkedPersonList`, **….**

Knoten einer doppelt verknüpften Liste

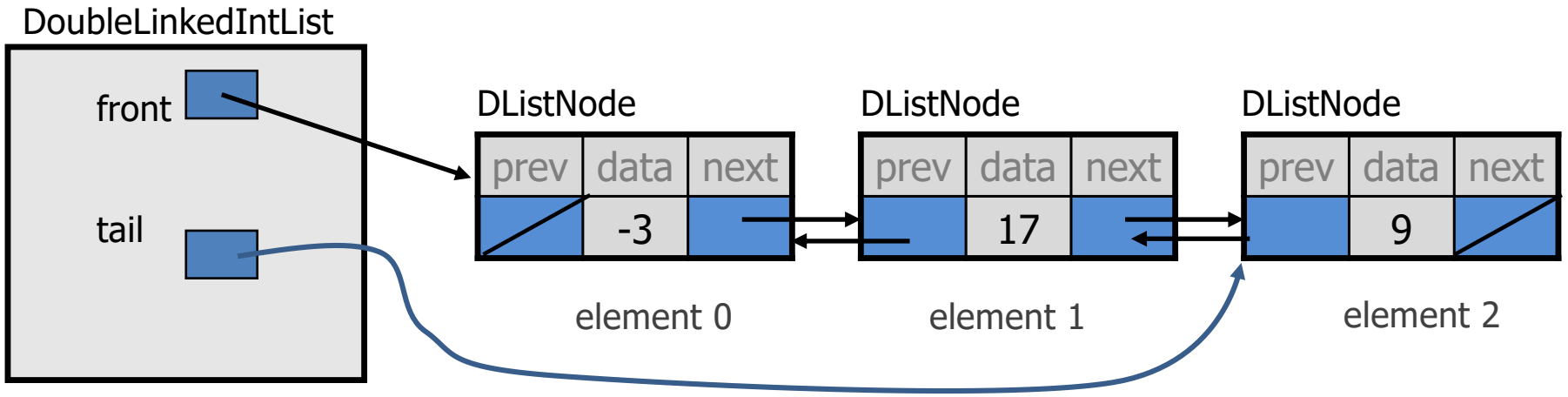
```
public class DListNode {
    int data;
    DListNode next;
    DListNode prev;
}
```

- **Jeder Knoten der Liste speichert:**
  - **Den Wert einer ganzen (`int`) Zahl**
  - **Einen Verweis auf einen Vorgänger Listenknoten**
  - **Einen Verweis auf einen Nachfolger Listenknoten**



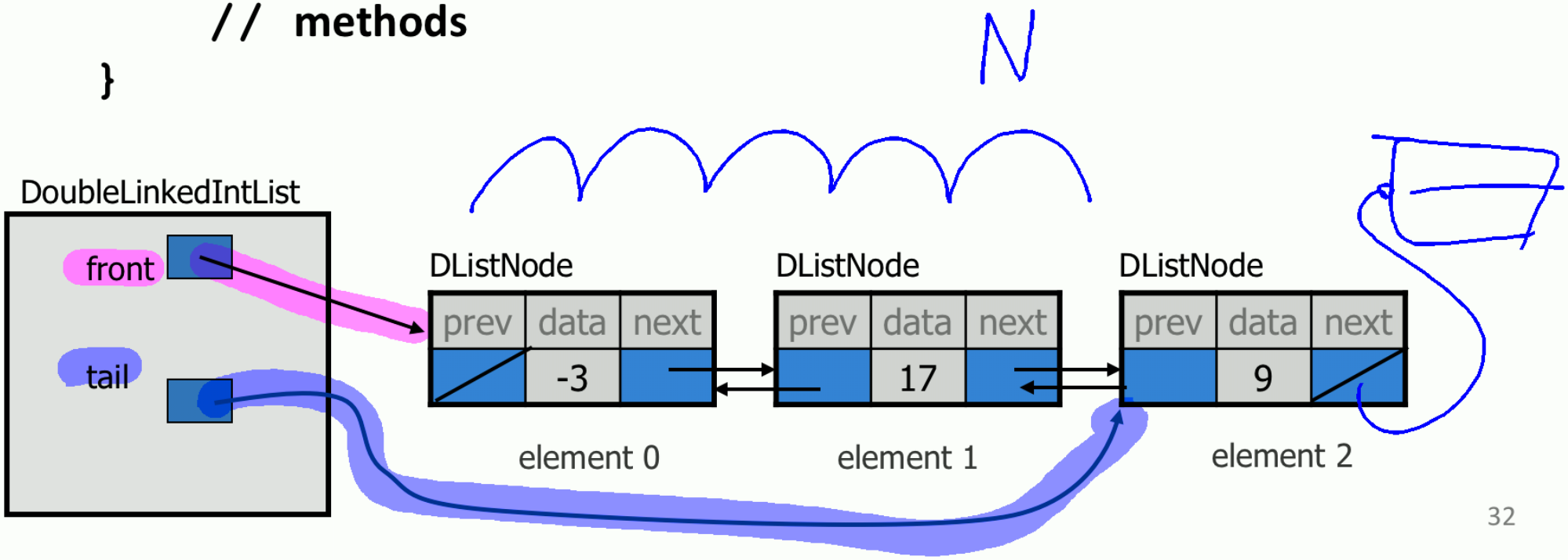
Doppelt verknüpfte Liste

```
public class DoubleLinkedIntList {
    private DListNode front;
    private DListNode tail;

    // methods
}
```



Doppelt verknüpfte Liste

```
public class DoubleLinkedIntList {
    private DListNode front;
    private DListNode tail;

    // methods
}
```



Software Entwicklung

- **Auf früheren Lösungen aufbauen**
  - **Von** `LinkedIntList` **zu** `DoubleLinkedIntList`

- **Welches Verhalten sollte DoubleLinkedIntList haben?**

- **Welches Verhalten sollte LinkedDoubleList haben?**

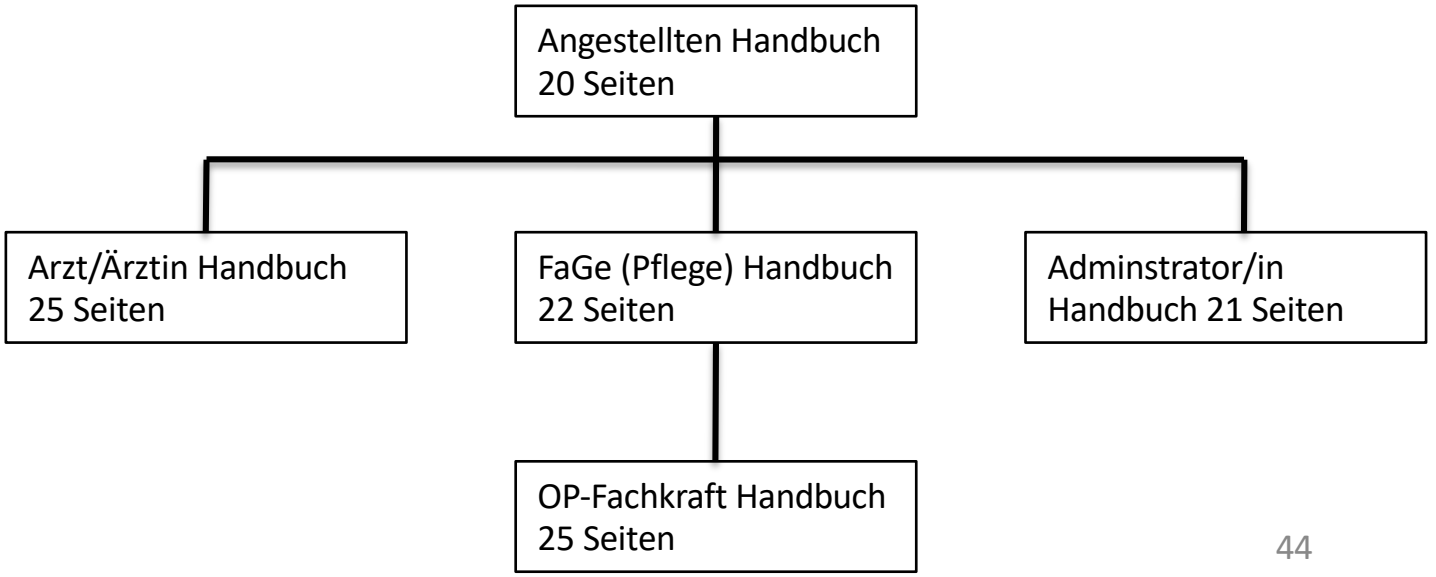
```
public class DoubleLinkedIntList {
    public double get(int index){ }
    public void set(int index, int value){ }
    public boolean isEmpty(){ }
    public void addFirst(int value){ }
    public void addLast(int value){ }
    public int removeFirst(){ }
    public int removeLast(){ }
    public void clear(){ }
    public int[] toArray(){ }
    public String toString(){ }
}
```

7.2 Neue Klasse aus existierenden Klassen bilden

Spital Beispiel

- **Gemeinsame Regeln für alle Angestellten: Arbeitszeit, Urlaub, Sozialleistungen, Fortbildung, Pflichtenheft, …**
  - **Alle Angestellten besuchen eine gemeinsame Orientierungveranstal-tung um die für alle gültigen Regeln des Krankenhauses zu erhalten**
    - **Verhalten im Notfall, professionelles Verhalten, Lohnfortzahlung im Krankheitsfall, …**
  - **Alle Angestellten erhalten das 20-seitige Handbuch, das die Arbeitsverhältnisse im Spital regelt**
- **Aber …**

- **Aber jede Abteilung hat noch ihre eigenen Regeln**
  - **Angestellte erhalten ein weiteres Handbuch (mit 1-5 Seiten) für die Regeln, die für ihre Abteilung gelten**
  - **Das Zusatzhandbuch enthält weitere Regeln und ändert oder erweitert Regeln aus dem (allgemeinen) Spitalregelwerk.**

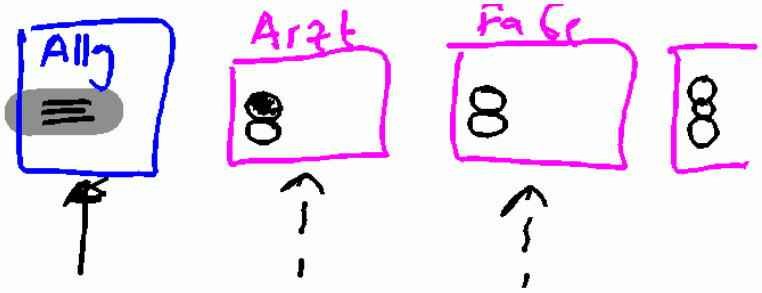


Organisation der Regeln

- **Warum wollen wir nicht je ein Handbuch für Ärzte/innen (25 Seiten), eines für FaGe (22 Seiten), eines für Administratoren (21 Seiten) und eines für OP-Fachkräfte (25 Seiten) haben?**

- **Vorteile wenn wir die Handbücher aufteilen :**
  - **Anpassungen: Nur eine Stelle muss geändert werden, wenn sich die allgemeinen Regeln ändern.**
  - **Lokalitätsprinzip: Schneller Überblick über die Regeln die nur für Ärzte/innen gelten.**

Organisation der Regeln



- Warum wollen wir nicht je ein Handbuch für Ärzte/innen (25 Seiten), eines für FaGe (22 Seiten), eines für Administratoren (21 Seiten) und eines für OP-Fachkräfte (25 Seiten) haben?

- Vorteile wenn wir die Handbücher aufteilen :
  - Anpassungen: Nur eine Stelle muss geändert werden, wenn sich die allgemeinen Regeln ändern.
  - Lokalitätsprinzip: Schneller Überblick über die Regeln die nur für Ärzte/innen gelten.

Was wir aus dem Beispiel mitnehmen

- **Allgemeine Regeln sind sinnvoll (das 20-seitige allgemeine Handbuch).**
- **Einzelne Gruppen brauchen evtl. Zusatzregeln die nur für diese Abteilung gelten**
- **Die Gruppenregeln haben Vorrang vor den allgemeinen Regeln**
- **Die Gruppenregeln können evtl. allgemeine Regeln ausser Kraft setzen.**

Regeln für Angestellte

- **Hier ist ein Satz von Regeln der für Angestellte des Krankenhauses gilt:**
  - **Die Arbeitszeit ist 42 Stunden pro Woche.**
  - **Angestellte erhalten einen Basislohn von 80’000 sFr/Jahr, bis auf OP-Fachkräfte die 10’000 sFr/Jahr extra erhalten, und Administratoren/-innen, die 5’000 sFr/Jahr extra erhalten.**
  - **Angestellte haben 4 Wochen Urlaub pro Jahr, bis auf Ärzte/innen, die eine Woche extra (also insgesamt 5 Wochen) bekommen**
  - **Um einen Urlaub zu beantragen sollten Angestellte ein grünes Formular verwenden, bis auf Ärzte/innen, die ein gelbes Formular brauchen.**

Regeln für Angestellte

- Hier ist ein Satz von Regeln der für Angestellte des Krankenhauses gilt:
  - Die Arbeitszeit ist 42 Stunden pro Woche.
  - Angestellte erhalten einen Basislohn von 80’000 sFr/Jahr, bis auf OP-Fachkräfte die 10’000 sFr/Jahr extra erhalten, und Administratoren/-innen, die 5’000 sFr/Jahr extra erhalten.
  - Angestellte haben 4 Wochen Urlaub pro Jahr, bis auf Ärzte/innen, die eine Woche extra (also insgesamt 5 Wochen) bekommen
  - Um einen Urlaub zu beantragen sollten Angestellte ein grünes Formular verwenden, bis auf Ärzte/innen, die ein gelbes Formular brauchen.

Aufgaben und Fähigkeiten der Angestellten

- **Jede Art von Angestellten spielt im Spitalbetrieb eine bestimmte Rolle**
  - **Ärzte/innen behandeln Patienten (untersuchen und stellen eine Diagnose).**
  - **Administratoren verarbeiten Rechnungen.**
  - **FaGe pflegen Patienten auf einer Station (müssen also einer Station zugeteilt sein, für einen bestimmten Zeitraum)**
  - **OP-Fachkräfte managen den OP-Saal.**
- **Wir wollen jetzt ein System erstellen, das die Spitaldirektion bei der Einsatzplanung, Rechnungsstellung etc unterstützt**
  - **Für jede/r Angestellte/n gibt es ein Exemplar einer Klasse mit der wir planen/verwalten können**

Eine Klasse für Angestellte ("`Angestellte`")

```
// A class to represent employees in general (20-page manual).
public class Angestellte {
    public int getHours() {
        return 42;           // works 42 hours / week
    }

    public double getSalary() {
        return 80000.0;      // sFr 80,000.00 / year
    }

    public int getVacationDays() {
        return 20;           // 4 weeks paid vacation, not including weekends
    }

    public String getVacationForm() {
        return "green";      // use the green form
    }
}
```

- **Erstellen Sie eine Klasse für FaGe**
  - **Alle gemeinsamen Regeln gelten ohne Einschränkung**
  - **FaGe zeichnen sich dadurch aus, dass sie (für einen Tag) auf einer Station arbeiten**

- **Erstellen Sie eine Klasse für FaGe**
  - **Alle gemeinsamen Regeln gelten ohne Einschränkung**
  - **FaGe zeichnen sich dadurch aus, dass sie (für einen Tag) auf einer Station arbeiten**

- **Spezielles Verhalten von FaGe**
  - **workAtStation(int station) // arbeitet auf Station**

Redundante `FaGe` Klasse

```
// A redundant class to represent FaGe (nurses), 22 page manual
public class FaGe {
    public int getHours() {
        return 42;            // works 42 hours / week
    }

    public double getSalary() {
        return 80000.0;       // sFr 80,000.00 / year
    }

    public int getVacationDays() {
        return 20;            // 4 weeks paid vacation, not including weekends
    }

    public String getVacationForm() {
        return ”green";      // use the green form
    }

    public void workAtStation(int station) {
        System.out.println(”Working at station: " + station);
    }
}
```

Redundante FaGe Klasse

```
// A redundant class to represent FaGe (nurses), 22 page manual
public class FaGe {
    public int getHours() {
        return 42;              // works 42 hours / week
    }

    public double getSalary() {
        return 80000.0;         // sFr 80,000.00 / year
    }

    public int getVacationDays() {
        return 20;              // 4 weeks paid vacation, not including weekends
    }

    public String getVacationForm() {
        return "green";         // use the green form
    }

    public void workAtStation(int station) {
        System.out.println("Working at station: " + station);
    }
}
```

Viel (redundanter) Code

- `workAtStation` ist das einzige Verhalten das `FaGe` von anderen Angestellten unterscheidet
- Die Struktur des Programms sollte die Verhältnisse in der realen Welt widerspiegeln

- Was wir wirklich gerne sagen wollen ist
  - Die Klasse für `FaGe` ist wie die Klasse für alle `Angestellten` und enthält noch eine weitere Methode (da sie auf einer Station arbeiten)

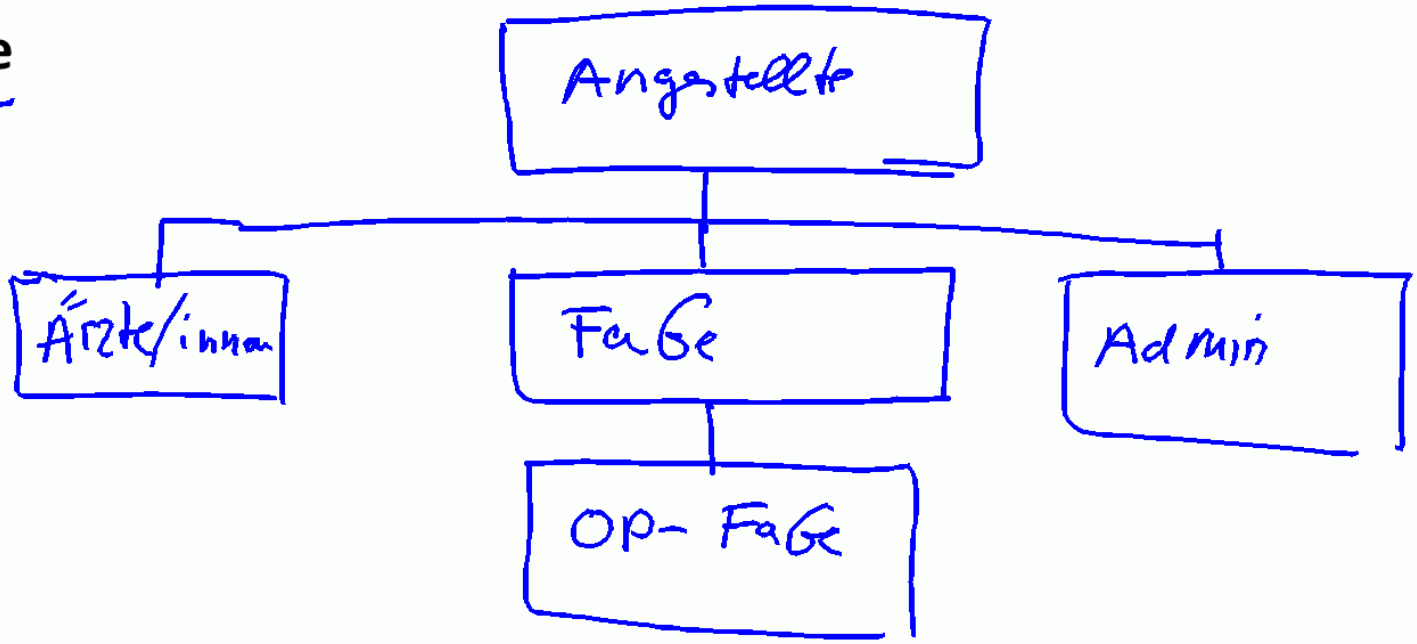
Beziehungen und Vererbung

- *Vererbung* (“*inheritance*”) erlaubt uns, eine Klasse als Erweiterung einer anderen Klasse auszudrücken

- Grundlage der Vererbung sind die Verhältnisse der verschiedenen Arten von Angestellten zueinander
  - Ein/e Arzt/Ärztin ist ein/e Angestellte/r
  - Jede OP-Fachkraft ist ein/e FaGe
- Die *ist-ein* (“*is a*”) Beziehung hält fest: diese Gruppe von Angestellten *ist ein* Spezialfall einer anderen Gruppe.

Beziehungen und Vererbung

- **Die *ist-ein* (“*is a*”) Beziehung hält fest, wann eine Gruppe von Angestellten als ein Spezialfall einer anderen Gruppe gilt**
  - **Hierarchie**

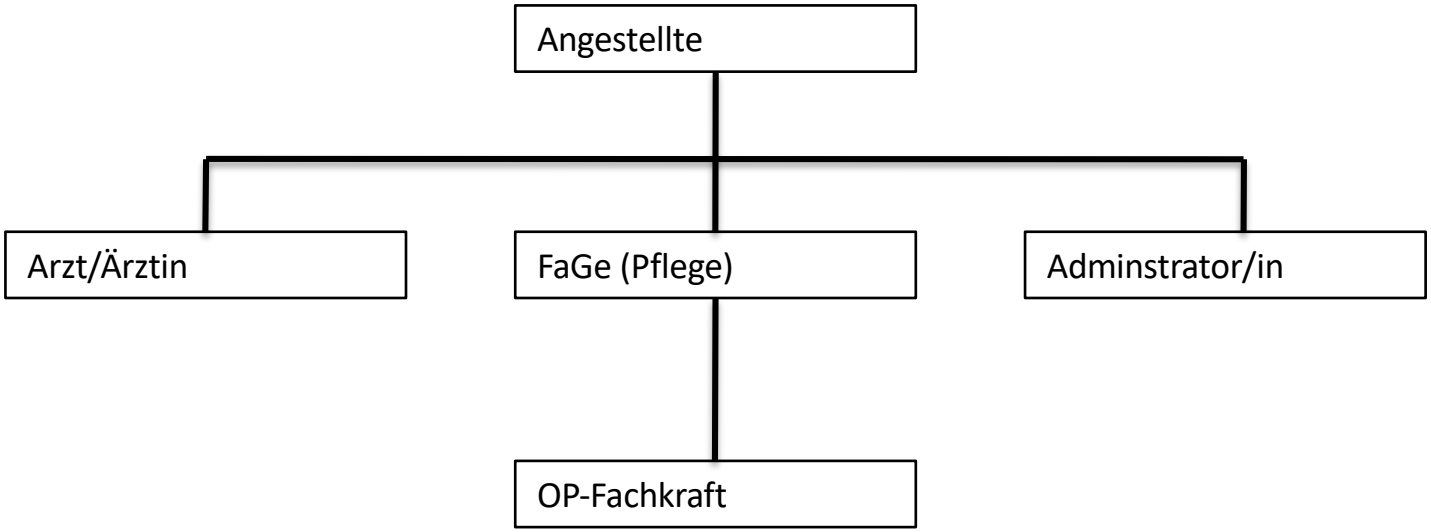


Beziehungen und Vererbung

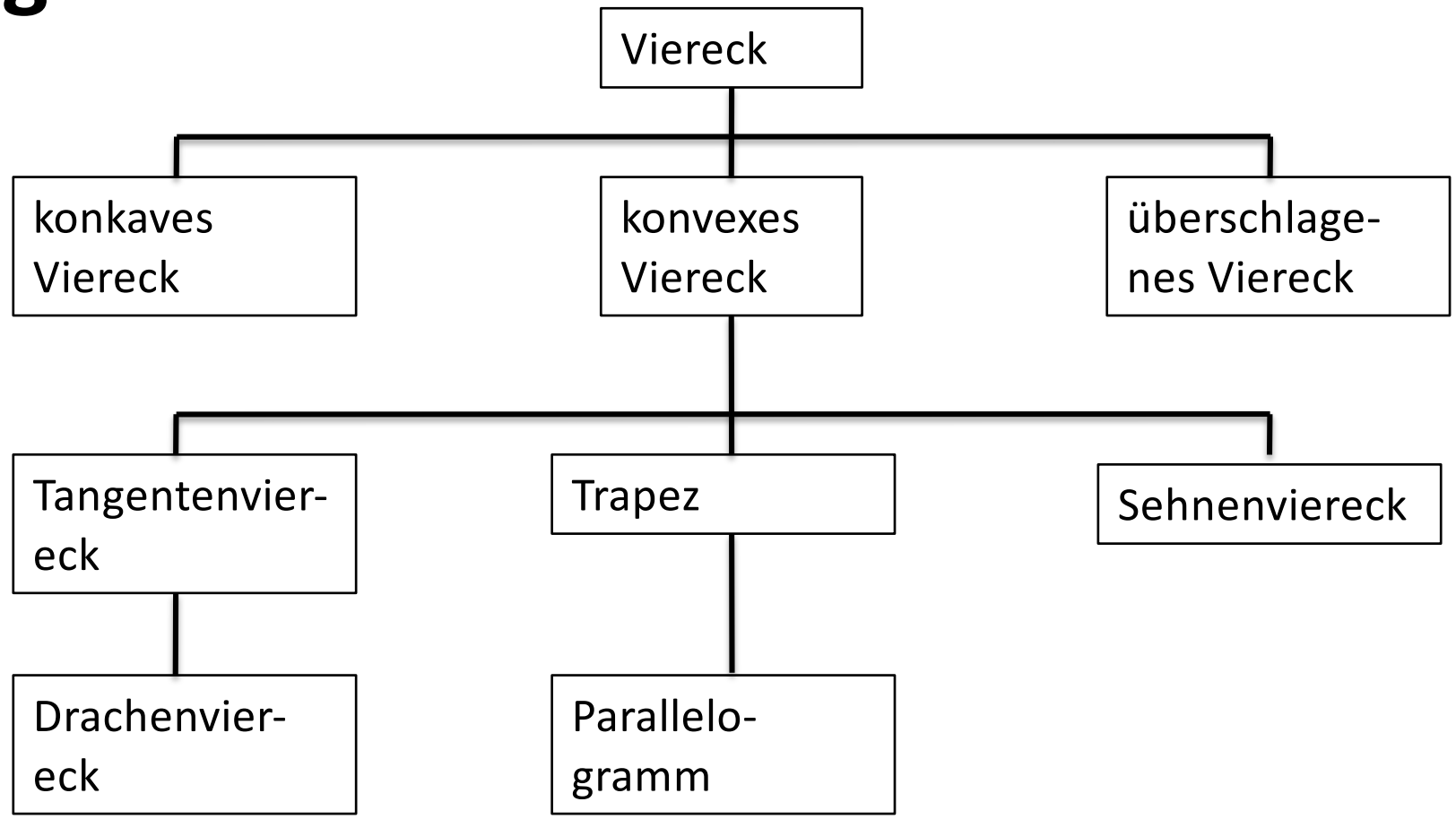
- **Die *ist-ein* (“*is a*”) Beziehung hält fest, wann eine Gruppe von Angestellten als ein Spezialfall einer anderen Gruppe gilt**
  - **Hierarchie**

- ***Vererbungshierarchie* (“*Inheritance Hierachy*”): Eine Menge von Klassen -- verbunden durch eine *ist-ein* Beziehung -- die gemeinsamen Code verwenden können.**

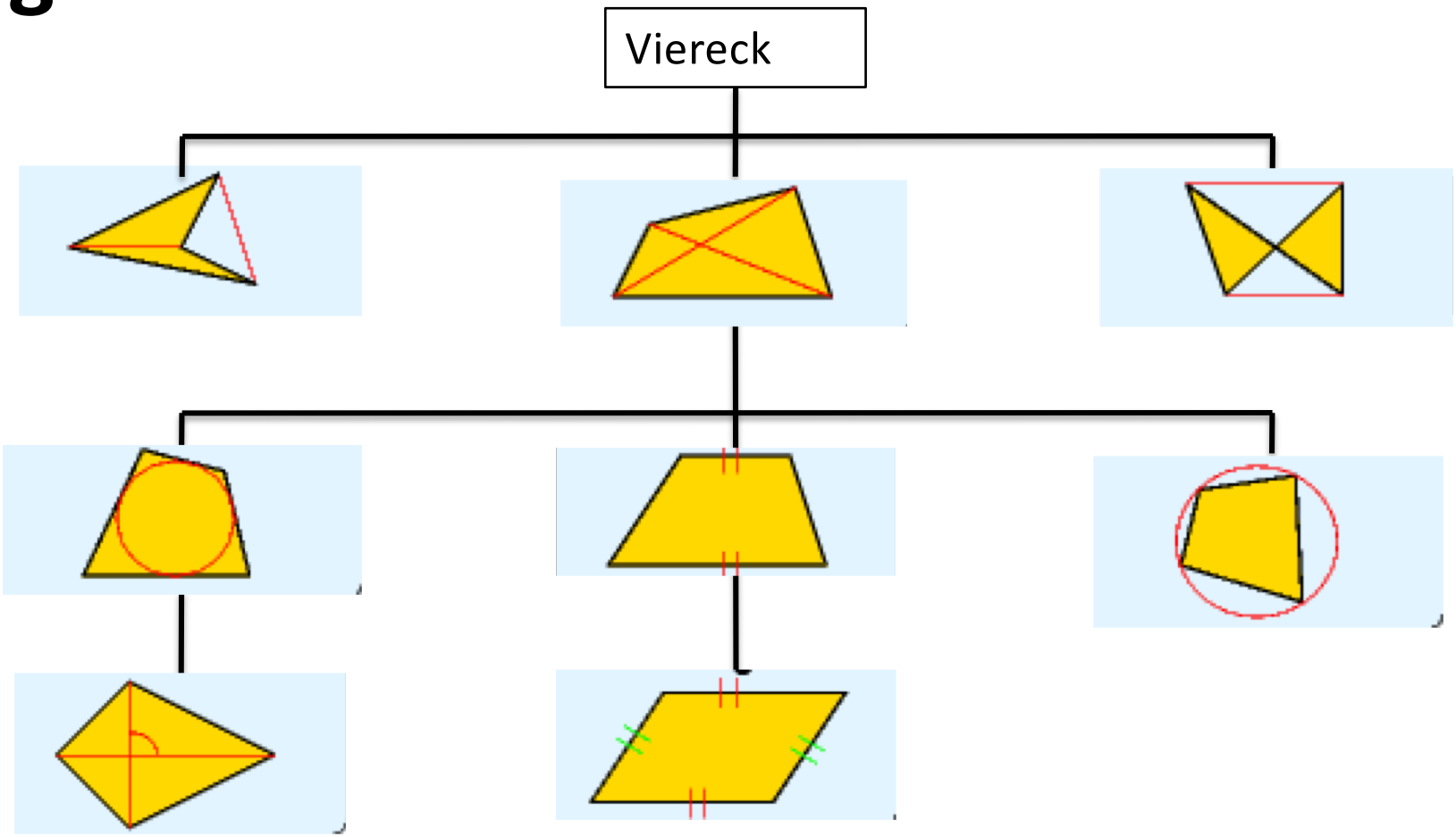
Vererbungshierarchie



Vererbungshierarchie



Vererbungshierarchie



Figuren aus Wikipedia

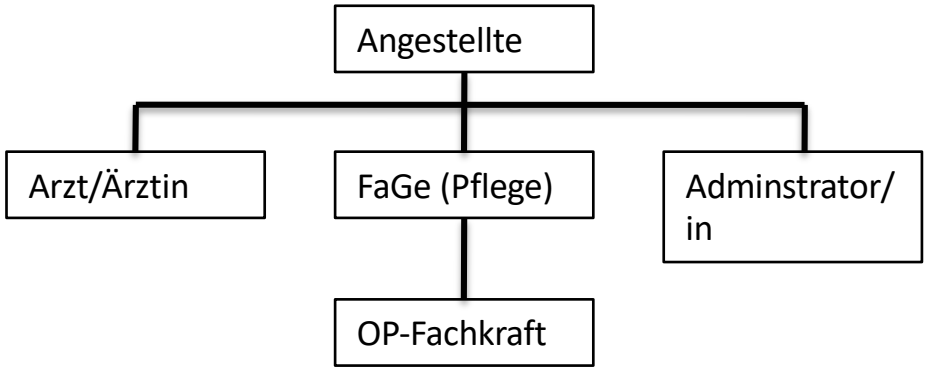
Vererbung

- ***Vererbung*** **erlaubt es neue Klassen aus existierenden Klassen zu bilden so dass die neue Klasse die Attribute bzw das Verhalten der alten Klasse übernimmt.**
  - **Erlaubt es verwandte Klassen in Gruppen anzuordnen**
  - **Erlaubt dass zwei oder mehr Klassen Code teilen**

- **Eine Klasse kann eine andere *erweitern (“extend")* und Daten und Zustand sowie Verhalten absorbieren**

Vererbungshierarchie

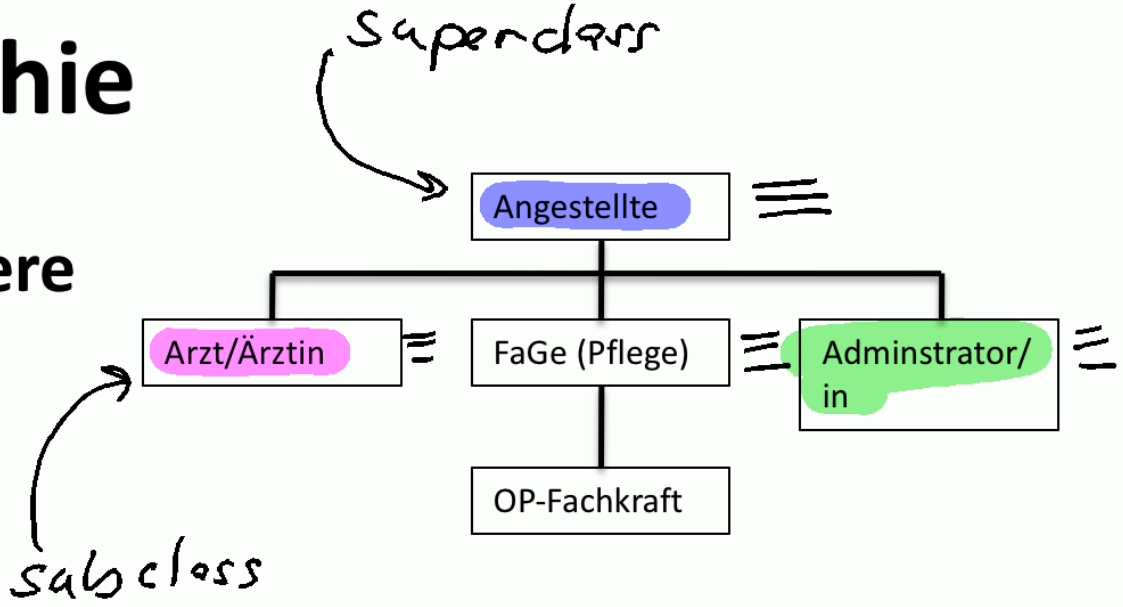
- **Klasse *erweitert* eine andere Klasse**
  - ***“superclass”* (*Oberklasse*): Klasse die erweitert wird.**



  - **“*subclass*” (*Unterklasse*): Klasse die die Oberklasse erweitert und ihre Eigenschaften/ihr Verhalten erbt.**
    - **Subclass erhält Kopie jedes Attributes/jeder Methode der Superclass**

Vererbungshierarchie

- **Klasse *erweitert* eine andere Klasse**
  - ***"superclass"* (*Oberklasse*): Klasse die erweitert wird.**



  - **"*subclass*" (*Unterklasse*): Klasse die die Oberklasse erweitert und ihre Eigenschaften/ihr Verhalten erbt.**
    - **Subclass erhält Kopie jedes Attributes/jeder Methode der Superclass**

Inheritance Syntax

```
public class name extends superclass {
```

- Beispiel:

```
public class FaGe extends Angestellte {
    ...
}
```

Die Erweiterung von `Angestellte` bewirkt für `FaGe` Objekte:

- Methoden `getHours`, `getSalary`, `getVacationDays`, und `getVacationForm` existieren automatisch
- können als `Angestellte` Objekt(exemplar) von Klienten behandelt werden (Details später)

Verbesserte `FaGe` Klasse

```
// A class to represent FaGe.
public class FaGe extends Angestellte {
    public void workAtStation (int station) {
        System.out.println("Working at station: " + station);
    }
}
```

- **Wir brauchen nur die Teile die spezifisch für eine Gruppe sind zu schreiben.**
  - **`FaGe` erbt Methoden `getHours`, `getSalary`, `getVacationDays`, und `getVacationForm` von `Angestellte`.**
  - **`FaGe` fügt die `workAtStation` Methode hinzu.**

Eine Klasse für Arzt/Ärztin

- **Sammeln wir die Regeln die für Ärzte und Ärztinnen gelten:**
  - **Ärzte/innen erhalten eine Woche mehr Urlaub (insgesamt 5).**
  - **Ärzte/innen verwenden ein gelbes Formular wenn sie Urlaub beantragen.**
  - **Ärzte/innen haben besondere Fähigkeiten: sie können Patienten behandeln.**

- **Problem: Wir wollen dass Ärzte/innen von Angestellte das *meiste* Verhalten erben aber wir wollen Teile durch neues Verhalten ersetzen (und neues Verhalten hinzufügen)**

Klasse `Arzt`

- Diese Klasse soll auf Angestellte aufbauen
- Erweiterung alleine ist aber nicht genug.

```
public class Arzt extends Angestellte {


}
```

Überschreiben von Methoden

- **Überschreiben ("override"): Definieren einer neuen Version einer Methode in einer Subclass die die Version der Superclass ersetzt.**
  - **Keine besondere Syntax erforderlich. Einfach eine neue Version der Methode schreiben.**

```
public class Arzt extends Angestellte {
  // overrides getVacationForm method in Angestellte class
   public String getVacationForm() {
          return "yellow";
      }
      ...
}
```

Aufgabe: Vervollständigen Sie Klasse `Arzt`

- **(Besondere) Eigenschaften des Verhaltens für `Arzt`.**
  - **5 Wochen Urlaub**
  - **Gelbes Formblatt für Urlaubsantrag**
  - **Behandelt Patienten (Untersuchung, Diagnose, Therapie)**
    - `treatPatient()`
- **Sonst *ist* ein Arzt *ein* `Angestellter`**

Arzt Klasse

```
// A class to represent medical personel, 25 page manual
public class Arzt extends Angestellte {
    // overrides getVacationForm from Angestellte class
    public String getVacationForm() {
        return "yellow";
    }

    // overrides getVacationDays from Angestellte class
    public int getVacationDays() {
        return 25;            // 5 weeks vacation
    }

    public void treatPatient() {
        System.out.println("I'll take care of you!");
    }
}
```

Arzt Klasse

```
// A class to represent medical personel, 25 page manual
public class Arzt extends Angestellte {
    // overrides getVacationForm from Angestellte class
    public String getVacationForm() {
        return "yellow";
    }

    // overrides getVacationDays from Angestellte class
    public int getVacationDays() {
        return 25;              // 5 weeks vacation
    }

    public void treatPatient() {
        System.out.println("I'll take care of you!");
    }
}
```

- **Erstellen Sie eine Klasse für Administratoren**
  - **Alle gemeinsamen Regeln gelten ohne Einschränkung bis auf**
    - **Administratoren verdienen sFr 5'000 extra/Jahr**
  - **Administratoren zeichnen sich dadurch aus, dass sie Rechnungen verarbeiten (Rg. schicken, Zahlungseingang kontrollieren, …)**
  - **Wir brauchen also eine Methode für dieses Verhalten**
    - **Nennen wir sie `processBill()`**

Eine Klasse für Administratoren/-innen

```
// A class to represent administrators (21-page manual).
public class Administrator extends Angestellte {

   // overrides getSalary from Angestellte
   public double getSalary() {
       return 85000.0;        // sFr 85,000.00 / year
   }

    public void processBill() {
        System.out.println("Pay now! ");
    }

}
```

Ebenen in der Inheritance Hierarchie

- Die Inheritance Hierarchie erlaubt verschiedene Ebenen.
  - Beispiel: Eine OP-Fachkraft ist ein/e FaGe aber verdient mehr (90'000 sFr/Jahr) – und kann auch einen OP Saal managen.

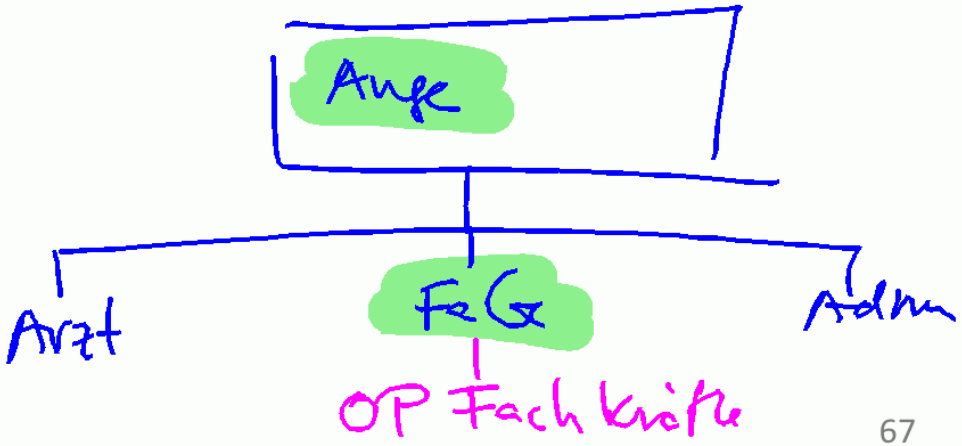
```
public class OPFachkraft extends FaGe {
    ...
}
```

  - Übung: Vervollständigen Sie die `OPFachkraft` Klasse
    - `manageOP()` besonderes Verhalten

Ebenen in der Inheritance Hierarchie

- Die Inheritance Hierarchie erlaubt verschiedene Ebenen.
  - Beispiel: Eine OP-Fachkraft ist ein/e FaGe aber verdient mehr (90'000 sFr/Jahr) – und kann auch einen OP Saal managen.

```
public class OPFachkraft extends FaGe {
    ...
}
```



OPFachkraft Klasse

```
// A class to represent OPFachkraft, 25 page manual
public class OPFachkraft extends FaGe {

    // overrides getSalary from Angestellte
    public double getSalary() {
        return 90000.0;       // sFr 90,000.00 / year
    }

    public void manageOP() {
        System.out.println("I control the tools!");
    }
}
```

Motivation

- **Wir wollen weiter am Spitalmanagement System arbeiten.**
- **Stellen wir uns vor dass es Änderungen gibt, die *alle* Spitalangestellten betreffen.**
  - **Beispiel: Jeder erhält ein um sFr 10’000 höheres Gehalt um die Teuerung auszugleichen. Das Basisgehalt ist nun sFr 90’000/Jahr.**
  - **OP-Fachkräfte verdienen nun sFr 100’000/Jahr.**
  - **Administratoren verdienen nun sFr 95’000/Jahr.**
- **Diese Änderungen erzwingen dass wir den Code überarbeiten**

Veränderungen der Superclass

```
// A class to represent employees in general (20-page manual).
public class Angestellte {
    public int getHours() {
        return 42;            // works 42 hours / week
    }

    public double getSalary() {
        return 90000.0;       // sFr 90,000.00 / year
    }

    ...
}
```

- **Ist das genug (an Änderungen)?**

- **Die Subclasses von `Angestellte` sind noch nicht richtig.**
  - **Diese Klassen hatten `getSalary()` überschrieben um andere Werte zurückzugeben**

(K)eine Lösung

```
public class OPFachkraft extends FaGe {
  public double getSalary() {
     return 100000.0;       // sFr 100,000.00 / year
  }
  ...
}
public class Administrator extends Angestellte {
   public double getSalary() {
     return 95000.0;        // sFr 95,000.00 / year
   }
    ...
}
```

- **Problem: die Löhne der Subclasses basieren auf dem Lohn für `Angestellte` aber der Code in `getSalary` macht das nicht klar.**

Aufruf von überschriebenen Methoden

- Subclasses können überschriebene Methoden mittels der Referenzvariablen `super` aufrufen.

  `super.method(parameters)`

  - Beispiel:

```
public class Administrator extends Angestellte {
   public double getSalary() {
     return super.getSalary() + 5000.0;
   }
    ...
}
```

Übung

- Verändern Sie die Klassen `Arzt` und `OPFachkraft` so dass sie `super` gebrauchen.

Verbesserte `Arzt` Subclass

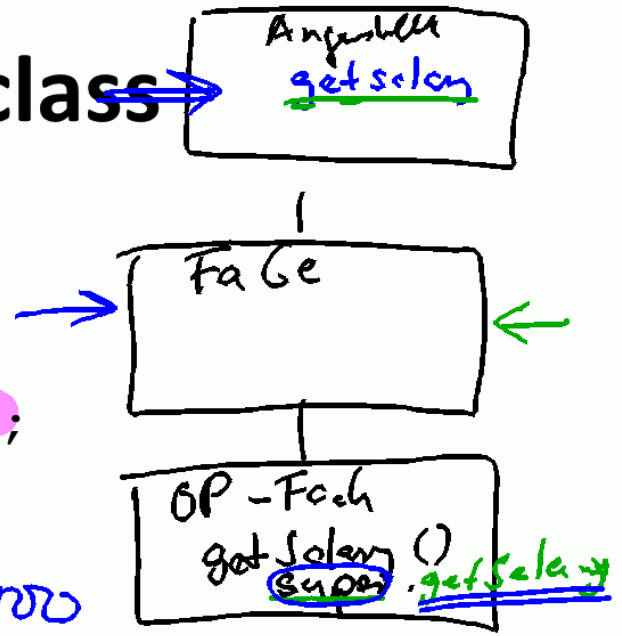
```
public class Arzt extends Angestellte {
    public String getVacationForm() {
        return "yellow";
    }

    public int getVacationDays() {
        return super.getVacationDays() + 5;
    }

    public void treatPatient() {
        System.out.println("I'll take care of you!");
 }
}
```

Verbesserte `OPFachkraft` Subclass

```
public class OPFachkraft extends FaGe {

   public double getSalary() {
      return super.getSalary() + 10000.0;
   }

  public void manageOP() {
      System.out.println("I control the tools!");
   }
}
```

- **`super` bezieht sich hier auf `FaGe` – da diese Klasse aber `getSalary` nicht überschreibt wird die Methode aus der Klasse `Angestellte` ausgeführt.**

Verbesserte OPFachkraft Subclass




```
public class OPFachkraft extends FaGe {

   public double getSalary() {
      return super.getSalary() + 10000.0;
   }

   public void manageOP() {
       System.out.println("I control the tools!");
    }
}
```

- `super` bezieht sich hier auf `FaGe` – da diese Klasse aber `getSalary` nicht überschreibt wird die Methode aus der Klasse `Angestellte` ausgeführt.

Verbesserte `Administrator` Subclass

```
public class Administrator extends Angestellte {

   public double getSalary() {
       return super.getSalary() + 5000.0;
   }

    public void processBill() {
        System.out.println("Pay now! ");
    }

}
```