Arten von Fehlern

Es gibt viele verschiedene Arten von Fehlern in Programmen.

- Typos (z.B. $n + 1$ statt $n - 1$)
- Fall vergessen
- Schnittstelle falsch erinnert
- Fehler im Algorithmus
- Mißverständnis
- ...

Sie sind unterschiedlich schwer zu finden und zu beseitigen.

Überblick

3. Fehler finden und vermeiden
   - Contract Programming und der Hoare-Kalkül
   - Debugging
   - Testen

Neues Kapitel

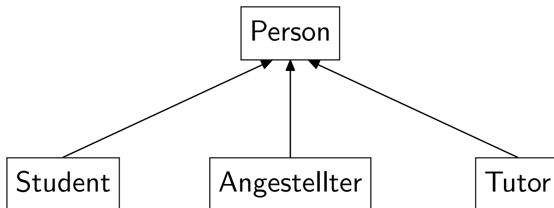
1. Einführung in die objektorientierte Programmierung
2. Rekursion
3. Fehler finden und vermeiden
4. **Objektorientiertes Design**
5. Effiziente Programme
6. Nebenläufigkeit
7. Laufzeitfehler
8. Refactoring
9. Persistenz
10. Eingebettete Systeme
11. Funktionale Programmierung
12. Logische Programmierung

Überblick

4. Objektorientiertes Design
   - Interfaces, Container, Iterator, Visitor
   - Composition, Decorator
   - Schichtenmodell und Filter-Pipelines

Mehrfachvererbung

Eine Klasse kann mehrere Unterklassen haben:

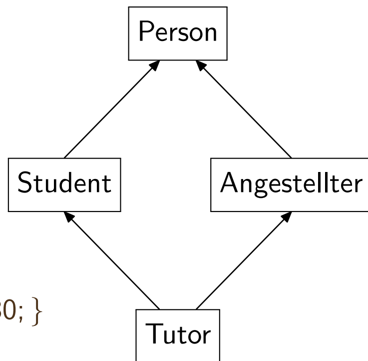


Aber: Ein *Tutor* ist ein *Student* und *Angestellter*.

Darf eine Klasse mehrere Oberklassen haben?

Das nennen wir *Mehrfachvererbung*.

```
class Student extends Person {
    ...
    double getGehalt() {return 0.0; }
    ...
}
class Angestellter extends Person {
    ...
    double getGehalt() {return 458.80; }
    ...
}
class Tutor extends Student, Angestellter {
    ...
}
...
Tutor egon = new Tutor(...);
double geld = egon.getGehalt();
```



Mehrfachvererbung

Mehrfachvererbung kann Probleme bereiten.

Java verbietet Mehrfachvererbung.

Andere Sprachen erlauben sie aber (z.B. C++).

Können wir ohne Mehrfachvererbung leben?

Ja, durch den **interface**-Mechanismus.

Abstrakte Klassen

Eine *abstrakte Klasse* kann Methoden enthalten, die nicht implementiert sind: Ihr Rumpf fehlt.

```
abstract class Tonerzeuger {
  void macheTon();
}
class Katze extends Tonerzeuger {
  void macheTon() {
    System.out.println("Miau");
  }
}
class Trompete extends Tonerzeuger {
  void macheTon() {
    System.out.println("Trara");
  }
}
```

Jeder *Tonerzeuger* hat so sicherlich *macheTon*().

Abstrakte Klassen

Abstrakte Klassen werden oft für die Spezifikation abstrakter Datentypen verwendet:

```
abstract class Stack⟨T⟩ {
  public boolean isempty();
  public T top();
  public void pop();
  public void push(T x);
}
```

Jede Unterklasse von *Stack*⟨*T*⟩ hat jetzt garantiert diese Methoden.

(Sie sollte auch die Semantik eines Stacks richtig umsetzen.)

```
class LinkedStack<T> implements Stack<T> {
  private T element = null;
  private LinkedStack<T> next = null;
  public boolean isempty() {return element == null;}
  public T top() {return element;}
  public void pop() {
    if(next.isempty()) {element = null;}
    else {
      element = next.element;
      next = next.next;
    }
  }
  public void push(T x) {
    LinkedStack<T> s = new LinkedStack<T>();
    s.element = element; s.next = next;
    next = s; element = x;
  }
}
```

LinkedStack ist eine konkrete Implementierung eines *Stack*.
(Sie ist nicht besonders schön, aber kurz...)

Es kann auch andere Implementierungen geben.

```
Stack<String> s = new LinkedStack<String>();
s.push("A");
s.push("B");
System.out.println(s.top());
```

```
LinkedStack<String> s = new LinkedStack<String>();
s.push("A");
s.push("B");
System.out.println(s.top());
```

Die erste Möglichkeit ist besser!

Warum???

Abstrakte Klassen und Interfaces

Eine abstrakte Klassen kann auch *einige* Methoden vollständig implementieren.

Wenn dies *nicht* so ist, dann gibt die abstrakte Klasse nur ein *Versprechen* ab, daß es bestimmte Methoden gibt.

So etwas nennt man auch ein *Interface*.

Ein Interface bestimmt, daß bestimmte Methoden existieren müssen.

```
interface Tonerzeuger {
  void macheTon();
}
```

Java stellt explizit **interface**s zur Verfügung.

In Sprachen mit Mehrfachvererbung werden sie durch abstrakte Klassen umgesetzt.

```
interface Tonerzeuger {
  void macheTon();
}
interface Stack<T> {
  public boolean isempty();
  public T top();
  public void pop();
  public void push(T x);
}
class LinkedStack<T> implements Stack<T> {
  ...
}
class LinkedStackMitTon<T>
      extends LinkedStack<T> implements Tonerzeuger {
  void macheTon() {
    System.out.println("Klack");
  }
}
```

Iterator

Iteratoren sind ein Standardverfahren, um auf Daten zuzugreifen.

```
public interface Iterator<E> {
  boolean hasNext(); // gibt es ein weiteres Element?
  E next(); // gib das nächste Element zurück
}
public interface Iterable<E> {
  Iterator<E> iterator(); // gibt einen Iterator über alle Elemente zurück
}
```

Ist *coll* ein *Iterable⟨String⟩*, dann sind solche Schleifen möglich:

```
Iterator<String> it = coll.iterator();
while(it.hasNext()) {
  String s = it.next();
  System.out.println(s);
}
```

Iterator

In Java gibt es eine hübsche Abkürzung.

Es sei wieder *coll* ein *Iterable⟨String⟩*. Anstatt

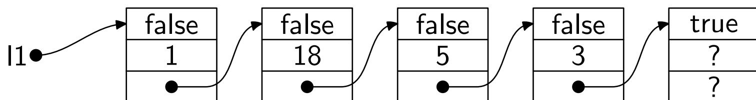
```
Iterator<String> it = coll.iterator();
while(it.hasNext()) {
   String s = it.next();
   System.out.println(s);
}
```

können wir auch dies schreiben:

```
for(String s : coll) {
   System.out.println(s);
}
```

Beispiel: Unsere Liste

Erinnern wir uns an unsere Listenimplementierung $List\langle T\rangle$.



Die öffentlichen Methoden sind

- **void** *isempty*(),
- T *head*(),
- $List\langle T\rangle$ *tail*() und
- $List\langle T\rangle$ *add*(T x).

Wir implementieren einen Iterator.

```
class ListIterator<T> implements Iterator<T> {
  private List<T> currentNode;
  public ListIterator(List<T> list) {
    currentNode = list;
  }
  public T next() {
    T result = currentNode.head();
    currentNode = currentNode.tail();
    return result;
  }
  public boolean hasNext() {
    return!currentNode.isempty();
  }
  public void remove() {}
}
```

Die Klasse *List*⟨*T*⟩ muß nun noch das Interface *Iterable*⟨*T*⟩ implementieren.

```
class List⟨T⟩ implements Iterable⟨T⟩ {
  private final boolean empty; // ist diese Liste leer?
  private final T value; // das erste Element dieser Liste
  private final List⟨T⟩ rest; // restliche Liste ohne das erste Element
  public Iterator⟨T⟩ iterator() {
    return new ListIterator⟨T⟩(this);
  }
```

Nun können wir Iteratoren für die Klasse $List\langle T\rangle$ sofort verwenden.

```
public static void main(String args[]) {
  List<String> list = new List<String>();
  list = list.add("Informatik");
  list = list.add("Physik");
  list = list.add("Mathematik");
  list = list.add("Chemie");
  list = list.add("Biologie");
  for(String s : list) {
    System.out.println(s);
  }
}
```