



Vorlesung 26

17.09.2024

Objektorientierte Programmierung entwickelte sich aufgrund der "Softwarekrise" 1965.

'Guter Code' macht sich aus durch folgende Eigenschaften:

- Beherrschbarkeit
- *Flexibilität*
- Wartbarkeit
- Übersichtlichkeit
- Erweiterbarkeit
- Arbeitsteilung problemlos möglich
- Fehlervermeidung
- Wiederverwendbarkeit

→ Funktionale Programmierung

→ *Prozedurale Programmierung*

→ Objektorientierte Programmierung behauptet alle der obigen Eigenschaften zu erfüllen.

Objektorientierte Programmierung

- Fasst Daten und Funktionen zu einer Einheit (Objekt) zusammen
- Objekt besteht aus:
 - Attributen (Daten)
 - Methoden (Funktionen, die auf den Daten arbeiten)
- Jedes Objekt hat einen Zustand und eine Identität
- Verschiedene Objekte haben verschiedene Identitäten, auch wenn sie den gleichen Zustand haben
- Im folgenden Code haben zwar beide Arrays a1 und a2 den gleichen Zustand (0,1,2), aber nicht die selbe Identität (unterschiedliche Namen/Speicherorte). Ein Aufruf von `a1(2)=10` verändert nur a1

- a3 hingegen referenziert a2 und hat daher die selbe Identität wie a2, da sie trotz unterschiedlicher Namen den selben Speicherort referenzieren.

```
val a1: Array[Int] = Array[Int](0,1,2)
val a2: Array[Int] = Array[Int](0,1,2)
val a3: Array[Int] = a2
```

Objekte

- In Scala ist alles ein Objekt und es gibt alleinstehende: *singleton Objekte*

```
object Katharina: //Beschreibe ein neues Objekt
  //Attribute
  //private sorgt dafür, dass die Attribute nur innerhalb d
  //-> arbeitsteilung, fehlervermeidung
  private var age: Int = 20
  private var job: String = "Dozentin"

  //Methoden
  def getAge: Int = age
  def getJob: String = job
  def celebrateBirthday(): Unit =
    age = age+1
  def changeJob(newJob: String): Unit =
    job = newJob
```

- In den folgenden scala3 aufrufen haben k und l haben wieder die selbe Identität, zeigen also auf das gleiche Objekt.
- Scala Konvention: Getter Funktionen, die nur einen Wert zurück liefern, werden ohne Klammern geschrieben, Im Gegensatz dazu werden Funktionen, die den Zustand des Objekts verändern MIT Klammern geschrieben, auch wenn sie keinen Input bekommen. So weiß der Nutzer immer, dass durch den Aufruf der Zustand geändert wird

```
val k = Katharina
k.getAge
k.celebrateBirthday()
val l = Katharina
```

```
l.getAge
//type wird durch folgenden aufruf returned, ist aber wenig h.
l
```

- Das override keyword wird verwendet, wenn wir bewusst bereits bestehende Funktionen überschreiben

```
object Katharina: //Beschreibe ein neues Objekt
  //Attribute
  //private sorgt dafür, dass die Attribute nur innerhalb d
  //-> arbeitsteilung, fehlervermeidung
  private var _age: Int = 20
  private var _job: String = "Dozentin"

  //Methoden
  def getAge: Int = _age
  def getJob: String = _job
  def celebrateBirthday(): Unit =
    _age = _age+1
  def changeJob(newjob: String): Unit =
    _job = newjob

  //eigene REPL toString Methode
  override def toString(): String =
    "Katharina (" + _age.toString + ", " + _job + ")"
```

Klassen

- Wenn wir mehrere gleichartige Objekte benötigen, sind singleton Objekte unpraktisch
- Stattdessen wollen wir einen "Bauplan", also eine Klasse aus dem Objekte gebaut werden können
- Ein konkretes Objekt ist dann eine Instanz einer Klasse

Konstruktor und Destruktor

- Objekte werden durch einen Konstruktor erzeugt und können durch einen Destruktor zerstört werden, um wieder Speicherplatz freizugeben
- Es gibt unterschiedliche Konstruktoren: Sie können sich in Anzahl und Typen der Parameter unterscheiden
- Scala: Es gibt einen primären und mehrere sekundäre Konstruktoren

```
class Lampe(private var power: Int):
  //primärer Konstruktor, legt gleich ein Attribut 'power'
  println("Lampe erstellt: Primärer Konstruktor")

  //Sekundärer Konstruktor
  def this() = this(200)
  println("Lampe erstellt: Sekundärer Konstruktor")
```

- In Scala, Python und Java ist der Destruktor implizit, in anderen Sprachen wie C++ muss dieser explizit geschrieben werden, da man bei der Programmierung direkt auf den Speicher zugreifen kann

Methoden und Attribute

- Jede Instanz eines Objekts hat die gleichen Attribute und Methoden, aber mit unterschiedlichen Werten, also einen eigenen Zustand
- Attribute können als Parameter des primären Konstruktors oder in dessen Code deklariert werden
- Attribute und Methoden haben Zugriffsmodifizierer: `private`, `protected` und `public` die festlegen wie und wo auf sie zugegriffen werden kann
 - `private`: nur innerhalb der Klasse
 - `protected`: innerhalb der Klasse und in Unterklassen
 - `public` oder keine Angabe: Zugriff in Klasse & Umgebung
- Die Parameter von Methoden sind immer `val`'s, können also nicht geändert werden
- `this` ist der Name der sekundären Konstruktoren: `def this(...)`
- `this` Aufruf eines anderen Konstruktors
- `this` Zugriff auf Attribute oder Methoden der Klasse

```
//Klasse
class Lampe(private var power: Int):
    //Voraussetzungen von Attributen prüfen
    require(power > 0, "Power muss größer 0 sein")
    //primärer Konstruktor, legt gleich ein Attribut 'power'
    println("Lampe erstellt: Primärer Konstruktor")
    //Methoden
    //im Default ist die Lampe aus
    private var on: Boolean = false

    //Sekundärer Konstruktor
    def this() = this(200)
    println("Lampe erstellt: Sekundärer Konstruktor")

    //2. sekundärer Konstruktor
    def this(power: Int, b: Boolean) =
        this(power)
        this.on = b

    //Methoden
    def isOn: Boolean = this.on
    def getPower: Int = this.power
    def toggle(): Unit =
        this.on = !this.on
    def change(power: Int): Unit =
        if power >= 0 && !this.on then
            this.power = power
    //change ohne this
    def change2(pow: Int): Unit =
        if pow >= 0 && !on then
            power = pow
```

Companion Objekt

```
class Studi(private val name: String, private val matr: Int):
    private var subject: String = "Bio*Informatik"
```

```

private var ects: Int = 0

def study(topic: String): Unit =
    println(this.name + " reads a book about " + topic)
    this.ects = this.ects + 1

def finished: Boolean = this.ects >= 180
def getMatr: Int = this.matr
def getName: String = this.name

//val s1: Studi = new Studi("Katharina", 12345)
//val s2: Studi = new Studi("Kristin", 12345)

```

- Matrikelnummer wird auf bei Immatrikulation festgelegt, aber wie kann sichergestellt werden, dass keine Matrikelnummer doppelt vergeben wird ?
→ Companion Objekt
- In Scala gibt es zu jeder Klasse ein Companion Objekt mit gleichen Namen, welches automatisch erzeugt wird, aber auch manuell
- Im Companion Objekt und Klasse können gegenseitig private Variablen einsehen. So kann im Companion Objekt ein Zustand über alle Instanzen gespeichert werden
- Damit das funktioniert muss der Code für die Klasse und das singleton Objekt in der gleichen Datei sein.
- Anpassung mit singleton Objekt:

```

class Studi(private val name:String):
    private val subject: String = "Bio*Informatik"
    private var ects: Int = 0
    private val matr: Int = Studi.next_matr()

    def study(topic: String): Unit =
        println(this.name + " reads a book about " + topic)
        this.ects = this.ects + 1

    def finished: Boolean = this.ects >= 180
    def getMatr: Int = this.matr
    def getName: String = this.name

```

```
//val s1: Studi = new Studi("Katharina", 12345)
//val s2: Studi = new Studi("Kristin", 12345)

object Studi:
  private var matr_counter: Int = 100

  private def next_matr(): Int =
    matr_counter = matr_counter+1
    matr_counter-1 //geschamckssache

  def resetMatrCounter(new_counter: Int): Unit =
    matr_counter = new_counter
```

Vererbung

- Klassen die aufeinander aufbauen können mit Vererbung gebildet werden
- In OOP gibt es Ober und Unterklassen. Die Unterklasse erbt Methoden und Attribute der Oberklasse. Somit bilden diese Klassen eine Hierarchie
- In Scala erben alle Klassen von der Klasse `Any` → ist also automatisch Oberklasse von allen anderen Klassen. Es wird geerbt: `==, !=, toString`

```
//Klassenhierarchie
class Person(private val name: String, private val age: Int):
  def isGrownUp: Boolean = age >= 18
  def introduction: String = "Hello, I am" + name + "!"
  def work(): Unit = println("Work, work")

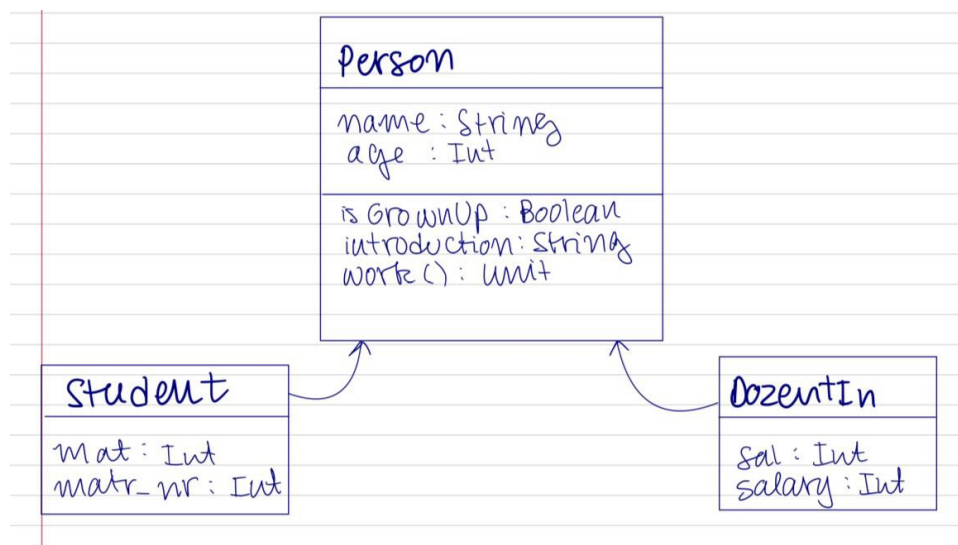
class Student(private val name: String, private var age: Int,
  def matr_nr: Int = mat
  override def work(): Unit = println("Study, study")
  override def introduction: String = "Hello, I am " + name

class DozentIn(private val name: String, private var age: Int
  def salary: Int = sal
```

```
override def work(): Unit = println("Teach, teach")  
override def introduction: String = "Hello, I am " + name
```

Vererbungsdiagramme

- Name → Attribute → Methoden
- In jeder Klasse werden nur neue Attribute und Methoden angegeben



Inklusionspolymorphie

- Jede Variable hat einen statischen und einen dynamischen Typ
- Der statische Typ wird bei der Deklaration angegeben und der Dynamische Typ, derjenige Typ des Objekts ist, welches der Variable zugewiesen ist.
- Der Dynamische Typ muss eine Unterklasse des statischen Typs sein, dies nennt man dann Inklusionspolymorphie
- Bsp: Wenn eine Instanz von 'Studi' erstellt wird, ist der statische Typ Person und der dynamische Typ Studi

```
def test():Unit =  
    var p:Person = new Student("Kim", 20, 12345) //static typ  
    p.work() //study, study
```



```
p.matr_nr //Feher, da p als Person deklariert wurde  
p = new DozentIn("Katharina", 30, 1000) //statischer Typ:  
p.work() //Teach, teach
```

Abstrakte Klassen

- Allgemeine Klassen, die generell sind, von denen man aber keine Objekte instanziiieren kann, da es sonst Methoden gäbe die nicht implementiert wären.
- Nicht alle Methoden müssen implementiert sein
- Von einer abstrakten Klasse kann kein Objekt Instanziiert werden, sie dienen als Oberklasse
- Eine Unterklasse einer abstrakten Klasse muss die fehlenden Methoden implementieren oder ist selbst abstrakt.
- Man hat i.d.R. eine Mischung aus implementierten und nicht implementierten Methoden

```
abstract class Person(private val name: String, private val age: Int) {  
    def isGrownUp: Boolean = age >= 18  
    def introduction: String //keine Implementierung: abstrakt  
    def work(): Unit = //keine Implementierung: abstrakte Methode  
}
```