Vorlesung 26

17.09.2024

Objektorientierte Programmierung entwickelte sich aufgrund der “Softwarekriese” 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 überschrieben

```
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: Se 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
- Attribut 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):
    //vorraussetzungen von Attributen prüfen
    require(power > 0, "Power muss größer 0 sein")
    //primärer Konstruktor, legt gleich ein Attribut 'power' a
    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
```