# Übungsblatt 07

## 3 Aufgaben, 20 Punkte

Objektorientierte Modellierung und Programmierung (inf031) Sommersemester 2020
Carl von Ossietzky Universität Oldenburg, Fakultät II, Department für Informatik

Dr. C. Schönberg

---

**Ausgabe:** 2020-06-02 14:00 **Abgabe:** 2020-06-09 12:00

**Aufgabe 1:** *Lambda-Ausdrücke* **(1 + 2 + 1 + 2 + 1 Punkte)**

Erinnern Sie sich an Aufgabe 1 von Übungsblatt 04. Dort ging es um die Definition von Funktionsketten.

Definieren Sie zunächst eine Klasse `LambdaTest` mit einer `main`-Methode.

**a)** Definieren Sie ein *generisches funktionales Interface* `Function` mit einer Methode `calculate`. Diese Methode bildet einen Wert von einem Typ auf einen Wert von einem (möglicherweise anderen) Typ ab. Beide Typen für Definitions- und Wertebereich sollen Zahlen sein, ohne eine Einschränkung auf z.B. `Integer` oder `Double`.

**b)** Verwenden Sie **Lambda-Ausdrücke**, um in der `main`-Methode vier Instanzen des `Function`-Interfaces zu erzeugen, die `Double`-Werte auf `Double`-Werte abbilden:

i) eine Instanz `id`, welche einen Wert auf sich selbst abbildet (Identitätsfunktion)
ii) eine Instanz `inverse`, welche einen Wert auf sein Inverses abbildet (z.B. $-5.3$ auf $5.3$)
iii) eine Instanz `timesTen`, welche einen Wert mit $10$ multipliziert
iv) eine Instanz `divideByPi`, welche einen Wert durch $\pi$ teilt

**c)** Verwenden Sie außerdem einen **Lambda-Ausdruck**, um eine Instanz `round` des `Function`-Interfaces zu erzeugen, das `Double`-Werte auf `Long`-Werte abbildet.
Referenzieren Sie dazu die Methode `Math.round(v: double): long`.

**d)** Schreiben Sie eine Hilfsmethode
`Function<Double, Double> makeChain(final Function<Double, Double>[] funs)`,
die eine `Function`-Instanz zurückgibt, die eine Verkettung der als Parameter übergebenen `functions` ist.
Für `Function`-Instanzen `f`, `g` und `h` gibt `makeChain([f, g, h])` also eine `Function`-Instanz `c` zurück, so dass `c.calculate(x)== h.calculate(g.calculate(f.calculate(x)))` für einen beliebigen **`double`**-Wert `x`.
Mathematisch gesehen soll gelten: $c = h \circ g \circ f$.

**e)** Ergänzen Sie Ihre `main`-Methode um die beiden Zeilen

```
1 @SuppressWarnings("unchecked")
2 Function<Double, Double> chain = makeChain(new Function[] { inverse, id,
      timesTen, divideByPi });
```

Verwenden Sie anschließend die `Function`-Instanz `round`, um das Ergebnis von `chain` angewandt auf die Zahl $5.5$ zu runden. Geben Sie die gerundete Zahl aus.
**Hinweis:** Das korrekte Ergebnis ist $-18$.

**Aufgabe 2:** *Lambdas und Streams* **(1 + 2 + 4 Punkte)**

Erinnern Sie sich an Aufgabe 2 von Übungsblatt 04. Dort ging es um die Definition von Zahlenfolgen.

Definieren Sie zunächst eine Klasse `StreamTest` mit einer `main`-Methode.

**a)** Erzeugen Sie einen *unendlichen* `Stream<Integer> naturals`, der die natürlichen Zahlen $1, 2, 3, \ldots$ in dieser Reihenfolge enthält.

**b)** Erzeugen Sie einen *unendlichen* `Stream<Integer> integers`, der die ganzen Zahlen $0, 1, -1, 2, -2, 3, \ldots$ in dieser Reihenfolge enthält.

**c)** Definieren Sie eine Hilfsmethode `filterAndSum(stream: Stream<Integer>): Integer`, welche nacheinander

i) den übergebenen Stream auf *gerade* Zahlen beschränkt

ii) den beschränkten Stream wiederum auf die *ersten zehn* Elemente beschränkt

iii) die verbliebenen Elemente aufsummiert

iv) diese Summe zurückgibt, oder die Zahl $0$ zurückgibt falls der Stream leer ist.

Verwenden Sie dabei Lambda-Ausdrücke, und verzichten Sie soweit möglich auf Hilfsvariablen zum Zwischenspeichern von Streams.
Geben Sie die Ergebnisse für die beiden Streams `naturals` und `integers` auf der Konsole aus.

**Hinweis 1:** Die korrekte Ausgabe ist $110$ bzw. $10$.
**Hinweis 2:** Sie können `Stream.iterate(seed: T, f: UnaryOperator<T>): Stream<T>` zur Erzeugung der Streams nutzen.
**Hinweis 3:** Sie können `Optional<T>.orElse(other: T): T` zum Zurückgeben der Summe nutzen. Die Methode gibt den Wert des Optionals zurück, oder `other` falls der Optional nur **`null`** enthält. Die Methode `Optional<T>.get(): T` wirft eine Exception, wenn der Optional nur **`null`** enthält.

**Aufgabe 3:** *I/O* **(1 + 1 + 1 + 1 + 1 + 1 Punkte)**

Gegeben sei folgende Java-Klasse `Person`, die eine Person mit Vor- und Nachnamen abbildet. Sie speichert außerdem ein Attribut `sortname`, das nicht direkt von außen gesetzt wird, sondern das sich aus Nach- und Vornamen zusammensetzt (Methode `updateSortname()`).

```
1  class Person {
2    private String firstname;
3    private String lastname;
4    private String sortname;
5    public Person() { }
6    public Person(String firstname, String lastname) {
7      this.firstname = firstname;
8      this.lastname = lastname;
9      updateSortname();
10   }
11   public String getFirstname() {
12     return firstname;
13   }
14   public void setFirstname(String firstname) {
15     this.firstname = firstname;
16     updateSortname();
17   }
18   public String getLastname() {
19     return lastname;
20   }
21   public void setLastname(String lastname) {
22     this.lastname = lastname;
23     updateSortname();
24   }
25   public String getSortname() {
26     return sortname;
27   }
28   public void updateSortname() {
29     sortname = lastname + firstname;
30   }
31   @Override
32   public String toString() {
33     return firstname + " " + lastname + " (" + sortname + ")";
34   }
35 }
```

Gegeben sei außerdem eine Test-Klasse `PersonTest`, welche eine Liste von Personen anlegt, in einer Datei speichert und wieder ausliest.

```
1  public class PersonTest {
2
3    public static void main(String[] args)
4          throws IOException, ClassNotFoundException {
5      List<Person> persons = new ArrayList<>();
6      persons.add(new Person("Willy", "Wonka"));
7      persons.add(new Person("Charlie", "Bucket"));
8      persons.add(new Person("Grandpa", "Joe"));
9      System.out.println(persons);
10
11     Person.save("persons.sav", persons);
12     persons = Person.load("persons.sav");
13     System.out.println(persons);
14     Person.serialize("persons.ser", persons);
15     persons = Person.unserialize("persons.ser");
```

```
16      System.out.println(persons);
17    }
18
19  }
```

Verändern Sie die Klasse `Person` so, dass sich das Testprogramm erfolgreich ausführen lässt. Implementieren Sie dazu folgende statische Methoden in der Klasse `Person`:

**a)** `load(filename: String): List<Person> throws IOException`: Verwendet die Methode `load(...): Person`, um eine Liste von Personen aus einer Datei zu lesen.

**b)** `load(in: DataInputStream): Person throws IOException`: Verwendet einen `DataInputStream`, um eine Person zu laden.

**c)** `save(filename: String, list: List<Person>)throws IOException`: Verwendet die Methode `save(..., Person)`, um eine Liste von Personen in eine Datei zu schreiben.

**d)** `save(out: DataOutputStream, person: Person)throws IOException`: Verwendet einen `DataOutputStream`, um eine Person zu speichern.

**e)** `unserialize(filename: String): List<Person> throws IOException, ClassNotFoundException`:
Verwendet die Java-Serialisierung, um eine Liste von Personen aus einer Datei zu lesen.

**f)** `serialize(filename: String, persons: List<Person>)throws IOException`: Verwendet die Java-Serialisierung, um eine Liste von Personen in eine Datei zu schreiben.

**Hinweis 1:** Beachten Sie dabei, dass der `sortname` *nicht* gespeichert werden soll.
**Hinweis 2:** Denken Sie daran, dass Sie zur Verwendung des Java-Serialisierungs-Mechanismus die Klasse `Person` um eine Schnittstellen-Implementierung erweitern müssen.

Das Programm soll folgende Ausgabe erzeugen:

```
[Willy Wonka (WonkaWilly), Charlie Bucket (BucketCharlie), Grandpa Joe (JoeGrandpa)]
[Willy Wonka (WonkaWilly), Charlie Bucket (BucketCharlie), Grandpa Joe (JoeGrandpa)]
[Willy Wonka (WonkaWilly), Charlie Bucket (BucketCharlie), Grandpa Joe (JoeGrandpa)]
```