

# Programmieren III (Java)

## 3. Praktikum: Lambda-Ausdrücke

Wintersemester 2023

Christopher Auer



### Lernziele

- ▶ Lambda-Ausdrücke erstellen und verwenden
- ▶ Funktionen höherer Ordnung
- ▶ Funktionale Interfaces
- ▶ Funktionale Interfaces mit `default`-Methoden erweitern

### Hinweise

- ▶ Die Praktika sind eine *ausgezeichnete Prüfungsvorbereitung*; aber nur, wenn Sie sie *eigenständig* bearbeiten oder es zumindest *versuchen*. Nachvollziehen der Lösungsvorschläge *reicht nicht* aus.
- ▶ Bearbeiten Sie die Aufgaben *vor* dem Praktikumstermin.
- ▶ Im Praktikum können Sie *Ihre Lösung* zeigen und *Fragen* stellen.
- ▶ Je nach *zeitlichem Verlauf*, wird während des Praktikumstermins der *Lösungsvorschlag besprochen*.
- ▶ Der *Lösungsvorschlag* wird online gestellt, nachdem *alle Gruppen* das Praktikum durchlaufen haben.

## Aufgabe 1: Koans

Ein Koan ist eine „kurze Anekdote oder Sentenz, die eine beispielhafte Handlung oder Aussage eines Zen-Meisters, ganz selten auch eines Zen-Schülers, darstellt“. In unserem Falle handelt es sich bei Koans um meist *kurze Programmieraufgaben* zu bestimmten Themen. Koans sind besonders beim Erlernen *funktionaler Programmiersprachen* wie Clojure oder Scala beliebt. Für diese Aufgaben gehen Sie jeweils wie folgt vor:

- ▶ Erstellen Sie eine *Klasse Koans* in der Sie die unten beschriebenen *Methoden* implementieren.
- ▶ *Alle Methoden* in Koans sind **public static**.
- ▶ Erstellen Sie eine *JUnit-Test-Klasse KoansTest* in der Sie die Methoden aus Koans *testen*.
- ▶ Verwenden Sie die *vordefinierten funktionalen Interfaces* aus `java.util.function`.
- ▶ Verwenden Sie *keine Streams*!
- ▶ Verwenden Sie *Lambda-Ausdrücke* für Referenzen auf *funktionale Interfaces*!
- ▶ Ersetzen Sie **?** im Folgenden durch einen geeigneten *Typen*.

### Koans `mapArray` ★★

Implementieren Sie eine *Methode*

```
void mapArray(int[] array, ? f)
```

mit:

- ▶ *f* ist eine *Funktion* `int → int`
- ▶ `mapArray` *wendet f* auf *jeden Eintrag* in *array* an und *schreibt* das Ergebnis an *die gleiche Stelle*.
- ▶ *Beispiel* wenn *f* *eins hinzuzählt*

```
[1,2,3,4] -> [2,3,4,5]
```

- ▶ *Testen* Sie `mapArray` *jeweils* mit einer Funktion, die:
  - ▶ *eins hinzuzählt*
  - ▶ die Zahl *quadriert*
- ▶ *Hinweis: Verwenden* Sie `Assert.assertEquals` um die *Gleichheit* zwei Arrays zu testen!

### Koan `fillArray` ★★

Implementieren Sie eine *Methode*

```
double[] fillArray(int length, ? s)
```

mit:

- ▶ *s* *generiert* *double*-Werte (`Supplier`)
- ▶ `fillArray` *erstellt einen double-Array* der Länge *length* und *belegt den Array* durch die von *s* generierten Werte
- ▶ *Beispiel*: Das *Ergebnis* wenn *f* *immer 2.0 liefert* und *length=5* ist der Array

```
[2.0,2.0,2.0,2.0,2.0]
```

- ▶ *Testen* Sie `fillArray` *jeweils* mit einer Funktion, die
  - ▶ *immer Math.PI liefert*

- ▶ eine *Zufallszahl liefert* (mit der Klasse `Random` erzeugt)
- ▶ *Hinweis* für den Test: `Random` liefert für den *gleichen „seed“* die *gleiche Folge von Zufalls-werte* (dieser Teil der Aufgabe ist ★★)

**Koan** *iterateFunction* ★★

Implementieren Sie eine *Methode*

```
int[] iterateFunction(int length, int first, ? f)
```

- ▶ *f* ist eine *Funktion* `int → int`
- ▶ *iterateFunction* *erstellt einen int-Array* der Länge *length* und belegt den Array durch *wiederholte Anwendung* von *f*. Dabei gilt für den resultierenden Array (*Pseudocode*):

```
array[0] == first;  
array[1] == f(first);  
array[2] == f(f(first));  
array[3] == f(f(f(first)));  
...
```

- ▶ Beispiel: *Ergebnis* wenn *f eins hinzuzählt*, `first==0` und `length==5`

```
[0,1,2,3,4]
```

- ▶ *Testen* Sie *iterateFunction* *jeweils* mit *f* für
  - ▶ *zählt 1 hinzu*
  - ▶ *verdoppelt die Zahl*Die *Startwerte* und *Länge* dürfen Sie *frei wählen*.

**Koan** *min* ★★

Implementieren Sie eine *generische Methode*

```
<T> T min(T[] elements, Comparator<T> c){
```

- ▶ *min* liefert das *kleinste Element* bezüglich des `Comparator`
- ▶ Beispiel: *Ergebnis* wenn *f Integer wie gewohnt vergleicht*

```
[4,1,-1,2,0] -> -1
```
- ▶ Testen Sie *min* mit einem `Comparator`, der...
  - ▶ *zwei Integer wie gewohnt vergleicht*
  - ▶ *zwei Strings nach der Länge vergleicht*
- ▶ Implementieren Sie den `Comparator` in Ihren Tests als *Lambda-Ausdrücke*.

**Koan** *createMultiplier* ★★

Implementieren Sie eine *Methode*



```
? createMultiplier(double d)
```

- ▶ Gibt eine *Funktion* `double → double` zurück
- ▶ Die *resultierende Funktion* multipliziert das *Argument* mit *d*
- ▶ *Beispiel*: `createMultiplier(5)` *erzeugt* eine Funktion, die einen `double`-Wert mit 5 *multipliziert*
- ▶ Testen Sie *createMultiplier* mit *verschiedenen Werten*

**Koan** forEachArray ☆☆ bis ☆☆

Implementieren Sie eine *Methode*

```
void forEachArray(String[] strings, ? c){
```

- ▶ c ist ein *String-Konsument*
- ▶ forEach *wendet* c auf jeden String in strings an
- ▶ Testen Sie forEachArray mit einem *Konsumenten*, der die Strings an einen  StringBuilder *anhängt*
  - ▶ Der Konsument ist eine *Closure*
  - ▶ Die Referenz auf den  StringBuilder ist „*gecaptured*“ (*gefangen*)

**Koan** duplicateChecker ☆☆

Implementieren Sie eine *generische Methode*

```
<T> Predicate<T> duplicateChecker(){
```

- ▶ duplicateChecker erstellt ein *Prädikat*  $T \rightarrow \text{boolean}$ .
- ▶ Das Prädikat *merkt sich* mit welchem Argument es bereits aufgerufen wurde: Bei *neuen* Argumenten gibt es *false* zurück, bei *bereits „gesehenen“* *true*.
- ▶ *Beispiel*

```
p.test("Foo") -> false
p.test("Bar") -> false
p.test("Foo") -> true
p.test("Baz") -> false
```

- ▶ Testen Sie duplicateChecker
- ▶ *Hinweise*:
  - ▶ Das Prädikat ist eine *Closure* mit einer Referenz auf eine *geeigneten Datenstruktur*.
  - ▶ Die Lösung dieser Aufgaben werden wir im nächsten Praktikum brauchen.

**Aufgabe 2: Reelle Funktionen**

Wir *bohren* das *funktionale Interface* RealFunction aus der *Vorlesung* auf

```
<<interface>>
RealFunction
+ apply(x : double): double
```

Erstellen Sie im folgenden *immer JUnit-Tests* für Ihre *Implementierungen*

**RealFunction deklarieren**

- ▶ *Deklarieren* Sie zunächst das *funktionale Interface* RealFunction wie oben gezeigt und versehen Sie es mit der *Annotation* @FunctionalInterface.
- ▶ Was unterscheidet *funktionale Interfaces* von *nicht-funktionalen Interfaces*?

- Was bewirkt die *Annotation* `@FunctionalInterface` und warum sollte man sie zu einem funktionalen Interface hinzufügen?

### Erweitern von `RealFunction` mit statischen Methoden ★★

*Zur Erinnerung: Funktionale Interfaces* dürfen nur *eine Instanzmethoden* beinhalten. Erlaubt sind aber beliebig viele *statische Methoden* und *default-Methoden*. *Statische Methoden* eignen sich bspw. für *Factories*, d.h. Methoden, über die man bequem Instanzen des funktionalen Interfaces erstellen kann. *Implementieren Sie* folgende *statischen Methoden* zur *Erstellung* von `RealFunctions`

- `RealFunction constant(double c)` erstellt die konstante Funktion  $f(x) = c$
- `RealFunction linear(double a, double b)` erstellt die lineare Funktion  $f(x) = ax + b$
- `RealFunction sine(double a, double f)` erstellt die Sinus-Funktion  $f(x) = a \sin(f \cdot x)$
- `RealFunction exp()` erstellt die Exponentialfunktion  $f(x) = \exp(x)$

*Verwenden* jeweils *Lambda-Ausdrücke* für die Rückgabewerte!

### default-Methode `addTo` ★★

*Funktionale Interfaces* erlauben auch *Default-Methoden*, mit denen sich funktionale Interfaces mit *hilfreichen Methoden* ausstatten lassen. *Implementieren Sie* folgende *Default-Methode* in `RealFunction`:

```
RealFunction addTo(RealFunction g)
```

`f.addTo(g)` erstellt eine *neue RealFunction-Instanz*, dessen Funktionswert sich aus der Addition von *f* und *g* ergibt. *Beispiel*:

```
RealFunction  
.linear(1,0)  
.addTo(RealFunction.sine(1,1));
```

ergibt die Funktion  $x + \sin(x)$ .

### default-Methode `compose` — Komposition ★★

*Implementieren Sie* die *Komposition*  $(f \circ g)(x) = f(g(x))$  als *Default-Methode* in `RealFunction`:

```
RealFunction composeWith(RealFunction f)
```

Das heißt, *zuerst wird* *f* auf *x* angewandt und das *Ergebnis* dann auf die Funktion angewandt, auf der `composeWith` aufgerufen wurde. *Beispiel*

```
RealFunction  
.exp()  
.composeWith(RealFunction.linear(2,1));
```

ergibt  $\exp(2 \cdot x + 1)$

**default-Methode multiplyWith — Multiplikation ★★**

*Implementieren* Sie die *Multiplikation*  $f_1(x) \cdot f_2(x) \cdot f_3(x) \dots$  als *Default-Methode* in RealFunction

```
RealFunction multiplyWith(RealFunction... funs){
```

*Beispiel*

```
RealFunction
    .linear(1,0)
    .multiplyWith(
        RealFunction.exp(),
        RealFunction.sine(1,1));
```

gibt  $f(x) = x \cdot \exp(x) \cdot \sin(x)$  zurück. *Beachten Sie:* Es handelt sich um einen *varargs-Parameter*!

**default-Methode approxDiff — angenäherte Ableitung ★★**

*Implementieren* Sie *angenäherte Ableitung der Funktion* als *Default-Methode* in RealFunction. Sie können die Ableitung über den *Differenzenquotienten* annähern:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

für ein „kleines  $h$ “. Implementieren Sie dazu eine **default-Methode**:

```
RealFunction approxDiff(double h)
```

Beispiel:

```
RealFunction.sin(1,0).approxDiff(1e-5)
```

liefert *ungefähr*  $f(x) = \cos(x)$ . Hinweise:

- ▶ *Theoretisch gilt:* je *kleiner*  $h$ , desto *genauer die Approximation*
- ▶ *Praktisch gilt* das aus *numerischen Gründen* nicht für ein beliebig kleines  $h$

**Optional: default-Methode max — Maximum ★★ bis ★★**

Folgende Methode gibt eine Funktion zurück, die immer den *maximalen Funktionswert* aller Funktionen zurückliefert:

```
RealFunction max(RealFunction... funs){
```

*Beispiel*

```
RealFunction
    .linear(1,0)
    .max(
        RealFunction.exp(),
        RealFunction.sine(1,1));
```

gibt  $f(x) = \max\{x, \exp(x), \sin(x)\}$  zurück. Beachten Sie, dass das Maximum auch auf die Funktion angewandt wird, auf der `max` aufgerufen wird.