

# Java - Funktionale Programmierung

---

**Dozent:** Prof. Dr. Michael Eichberg  
**Kontakt:** michael.eichberg@dhbw.de, Raum 149B  
**Version:** 1.0

Teile der Folien basieren auf: Th. Letschert - Funktionale Programmierung in Java.

---

**Folien:** <https://delors.github.io/prog-adv-java-funktionale-programmierung/folien.de.rst.html>  
<https://delors.github.io/prog-adv-java-funktionale-programmierung/folien.de.rst.html.pdf>  
**Fehler melden:** <https://github.com/Delors/delors.github.io/issues>

# 1. Einführung in die Funktionale Programmierung

# Grundlagen funktionaler Programmierung

- Programmierparadigma, bei dem Funktionen im Mittelpunkt stehen
- Vermeidet veränderliche Zustände (🚫 *Mutable State*)
- Fördert deklarativen Code statt imperativem Code

## Frage

Wie unterscheidet sich dieses Paradigma von der objektorientierten Programmierung?

## Antwort

- Methoden ohne Seiteneffekte
- Daten sind standardmäßig unveränderlich
- Fokus auf Funktionsanwendungen und -komposition

## Wichtige Konzepte

- Funktionen Höherer Ordnung
- Lambda-Ausdrücke
- Funktionskomposition
- Currying und Partielle Anwendung

## 2. Funktionale Programmierung in Java

---

# Lambdas

## Lambda (auch Closure):

Ein Ausdruck, dessen Wert eine Funktion ist.

Solche Ausdrücke sind sehr nützlich, mussten in Java bisher aber mit anonymen inneren Klassen emuliert werden.

### Ein einfache Personenklasse

```
1 class Person {
2     private String name;
3     private int age;
4
5     public Person(String name, int age) {
6         this.name = name;
7         this.age = age;
8     }
9     public String getName() { return name; }
10    public int getAge() { return age; }
11    public String toString() { return "Person[" + name + ", " + age + "]; }
12 }
```

### Sortieren von Personen nach Alter

Angenommen wir haben eine Klasse Person und eine Liste von Personen, die nach Alter sortiert werden soll. Dazu muss eine Vergleichsfunktion übergeben werden. In Java <8 kommt dazu nur ein Objekt in Frage.

```
1 List<Person> persons = Arrays.asList(
2     new Person("Hugo", 55),
3     new Person("Amalie", 15),
4     new Person("Anelise", 32) );
```

### Traditionelle Lösung

```
1 Collections.sort(persons, new Comparator<Person>() {
2     public int compare(Person p1, Person p2) {
3         return p1.getAge() - p2.getAge();
4     }
5 });
```

### Lösung ab Java 8

```
1 Collections.sort(
2     persons,
3     (p1, p2) -> { return p1.getAge() - p2.getAge(); }
4 );
```

### Lösung ab Java 8 (kürzer)

```
1 Collections.sort(persons, (p1, p2) -> p1.getAge() - p2.getAge());
```

### Achtung!

Bis Java 7 ist `java.lang.Object` der Basistyp aller Referenztypen. Der Typ eines Lambdas ist jedoch der Typ eines funktionalen Interfaces, das nur eine Methode hat und dieser Typ muss explizit angegeben werden.

## Instanzen von inneren Klassen können immer Object zugewiesen werden:

```
1 Object actionListener = new ActionListener() {
2     @Override
3     public void actionPerformed(ActionEvent e) {
4         System.out.println(text);
5     }
6 };
```

## Illegale Zuweisung:

```
1 Object actionListener = (e) → System.out.println(text);
2
3 // Error: The target type of this expression must be a functional interface
```

## Zuweisung an ein funktionales Interface:

```
1 ActionListener actionListener = (e) → System.out.println(text);
```

## Funktionale Interfaces

### Functional Interface / SAM-Interface (Single Abstract Method Interface):

Ein Functional Interface ist ein Interface das genau eine Methode enthält (die natürlich abstrakt ist) optional kann die Annotation `@FunctionalInterface` hinzugefügt werden.

#### Beispiel

```
1 @FunctionalInterface
2 interface MyActionListener extends java.awt.event.ActionListener {
3     /*final static*/ int MAGIC_NUMBER = 42;
4 }
5
6 MyActionListener actionListener = (e) → System.out.println(text + MyActionListener.MAGIC_NUMBER);
```

## Vordefinierte Funktionsinterfaces

`java.util.function` enthält viele vordefinierte Funktionsinterfaces, die in der funktionalen Programmierung häufig verwendet werden.

Beispiele sind:

- `Function<T,R>`: Eine Funktion, die ein Argument vom Typ `T` entgegennimmt und ein Ergebnis vom Typ `R` zurückgibt.
- `Predicate<T>`: Eine Funktion, die ein Argument vom Typ `T` entgegennimmt und ein Ergebnis vom Typ `boolean` zurückgibt.
- `Consumer<T>`: Eine Funktion, die ein Argument vom Typ `T` entgegennimmt und kein Ergebnis zurückgibt.
- `Supplier<T>`: Eine Funktion, die kein Argument entgegennimmt und ein Ergebnis vom Typ `T` zurückgibt.

#### Beispiel

`Predicate<T>`

```
1 static <T> List<T> filterList(List<T> l, Predicate<T> pred) {
2     List<T> res = new LinkedList<>();
3     for (T x : l) {
4         if (pred.test(x)) { res.add(x); }
5     }
6     return res;
7 }
8
9 List<Integer> l = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
10 System.out.println(filterList(l, x → x % 2 == 0));
```

Ausgabe: [2, 4, 6, 8]

#### Beispiel

`Consumer<T>`

```
1 class WorkerOnList<T> implements Consumer<List<T>> {
2     private Consumer<T> action;
3     public WorkerOnList(Consumer<T> action) { this.action = action; }
4
5     @Override public void accept(List<T> l) {
6         for (T x : l) action.accept(x);
7     }
8 }
```

```
7 }  
8  
9 WorkerOnList<Integer> worker = new WorkerOnList<>( (i) → System.out.println(i*10) );  
10 worker.accept(Arrays.asList(1,2,3,4));
```

**Ausgabe:** 10 20 30 40

# Lambdas - Method References

Als Implementierung eines funktionalen Interfaces (als „Lambda“) können auch Methoden verwendet werden.

## Beispiel

### Referenz auf statische Methode

```
1 class ListMethods {
2     static <T> List<T> filterList(List<T> l, Predicate<T> pred) {
3         List<T> res = new LinkedList<>();
4         for (T x : l) if (pred.test(x)) { res.add(x); }
5         return res;
6     }
7     static boolean isEven(int x) { return x % 2 == 0; }
8 }
9
10 List<Integer> l = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
11 System.out.println(filterList(l, ListMethods::isEven));
```

Ausgabe: [2, 4, 6, 8]

## Beispiel

### Referenz auf Instanzmethode

```
1 class Tester {
2     private int magicNumber;
3     public Tester(int magicNumber) { this.magicNumber = magicNumber; }
4     boolean isMagic(int x) { return x == magicNumber; }
5 }
6 class ListMethods {
7     static <T> List<T> filterList(List<T> l, Predicate<T> pred) {
8         List<T> res = new LinkedList<>();
9         for (T x : l) if (pred.test(x)) res.add(x);
10        return res;
11    }
12 }
13 List<Integer> l = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
14 System.out.println(filterList(l, new Tester(5)::isMagic));
```

## Beispiel

### Referenz auf Constructor

```
1 class Tester {
2     private int magicNumber;
3     public Tester(int magicNumber) { this.magicNumber = magicNumber; }
4     boolean isMagic(int x) { return x == magicNumber; }
5 }
6
7 Function<Integer, Tester> create = Tester::new;
8 create.apply(5).isMagic(5);
```

Ausgabe: true



# Erweiterungen der Collection API

## Neue Methoden in der Collection API

- `forEach(Consumer<? super T> action)`
- `removeIf(Predicate<? super T> filter)`
- `replaceAll(UnaryOperator<T> operator)`
- `sort(Comparator<? super T> c)`

### Beispiel

```
1 List<Integer> l = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);  
2 l.replaceAll(x -> x * 10);  
3 l.forEach(System.out::println);
```

*Ausgabe:* 10 20 30 40 50 60 70 80 90

# Übung

Schreiben Sie eine Klasse `Tuple2<T>`; d. h. eine Variante von `Pair` bei der beide Werte vom gleichen Typ `T` sein müssen. Die Klasse soll Methoden haben, um die beiden Werte zu setzen und zu lesen und weiterhin um folgende Methoden ergänzt werden:

- `void forEach(Consumer<...> action)`: Führt die Aktion für jedes Element in der `Queue` aus.
- `void replaceAll(UnaryOperator<...> operator)`: Ersetzt alle Elemente in der `Queue` durch das Ergebnis der Anwendung des Operators auf das Element.

Schreiben Sie Tests für die neuen Methoden. Stellen Sie 100% *Statementcoverage* sicher.

## Hinweis

Sorgen Sie ggf. vorher dafür, dass Sie eine angemessene Projektstruktur haben.

Passen Sie ggf. die `pom.xml` von ihren anderen Projekten an.

# Übung

Implementieren Sie eine Warteschlange (`Queue<T>`) basierend auf einer verketteten Liste. Die Klasse `Queue<T>` soll folgendes Interface implementieren.

```
1 public interface Queue<T> {
2     void enqueue(T item);
3     T dequeue();
4     boolean isEmpty();
5     int size();
6
7     void replaceAll(UnaryOperator<T> operator);
8     void forEach(Consumer<T> operator);
9     /** Erzeugt eine neue Queue<X> bei der die Elemente der neuen Queue das Ergebnis der Anwendung der Fun
10    <X> Queue<X> map(Function<T, X> mapper);
11
12    /** Erzeugt eine leere Queue. */
13    static <T> Queue<T> empty() { TODO }
14 }
```

Schreiben Sie Testfälle, um die Implementierung zu überprüfen. Zielen Sie auf mind. 100% *Statementcoverage* ab.