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

```
class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() { return age; }
    public String toString() { return "Person[" + name + ", " + age + "]"; }
}
```

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.

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

Traditionelle Lösung

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

Lösung ab Java 8

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

Lösung ab Java 8 (kürzer)

```
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:

```
Object actionListener = new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println(text);
    }
};
```

Illegale Zuweisung:

```
Object actionListener = (e) -> System.out.println(text);

// Error: The target type of this expression must be a functional interface
```

Zuweisung an ein funktionales Interface:

```
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

```
@FunctionalInterface
interface MyActionListener extends java.awt.event.ActionListener {
    /*final static*/ int MAGIC_NUMBER = 42;
}

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>`

```
static <T> List<T> filterList(List<T> l, Predicate<T> pred) {
    List<T> res = new LinkedList<>();
    for (T x : l) {
        if (pred.test(x)) { res.add(x); }
    }
    return res;
}

List<Integer> l = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
System.out.println(filterList(l, x -> x % 2 == 0));
```

Ausgabe: `[2, 4, 6, 8]`

Beispiel

`Consumer<T>`

```
class WorkerOnList<T> implements Consumer<List<T>> {
    private Consumer<T> action;
    public WorkerOnList(Consumer<T> action) { this.action = action; }

    @Override public void accept(List<T> l) {
        for (T x : l) action.accept(x);
}   }
```

```
8
9  WorkerOnList<Integer> worker =
10         new WorkerOnList<>( (i) -> System.out.println(i*10) );
11 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

```
class ListMethods {
    static <T> List<T> filterList(List<T> l, Predicate<T> pred) {
        List<T> res = new LinkedList<>();
        for (T x : l) if (pred.test(x)) { res.add(x); }
        return res;
    }
    static boolean isEven(int x) { return x % 2 == 0; }
}

List<Integer> l = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
System.out.println(filterList(l, ListMethods::isEven));
```

Ausgabe: `[2, 4, 6, 8]`

Beispiel

Referenz auf Instanzmethode

```
class Tester {
    private int magicNumber;
    public Tester(int magicNumber) { this.magicNumber = magicNumber; }
    boolean isMagic(int x) { return x == magicNumber; }
}
class ListMethods {
    static <T> List<T> filterList(List<T> l, Predicate<T> pred) {
        List<T> res = new LinkedList<>();
        for (T x : l) if (pred.test(x)) res.add(x);
        return res;
}   }

List<Integer> l = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
System.out.println(filterList(l, new Tester(5)::isMagic));
```

Beispiel

Referenz auf Constructor

```
class Tester {
    private int magicNumber;
    public Tester(int magicNumber) { this.magicNumber = magicNumber; }
    boolean isMagic(int x) { return x == magicNumber; }
}

Function<Integer,Tester> create = Tester::new;
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

2.1. Erste Implementierung von Funktionen höherer Ordnung

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

2.2. Implementierung einer Warteschlange mittels verketteter Liste

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      <X> Queue<X> map(Function<T, X> mapper);
10     static <T> Queue<T> empty() { TODO }
11 }
```

Erklärungen

`map`: Erzeugt eine neue `Queue<X>` bei der die Elemente der neuen Queue das Ergebnis der Anwendung der Funktion mapper sind.

`empty`: Erzeugt eine leere Queue.

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