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

```java
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.

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

Traditionelle Lösung

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

Lösung ab Java 8

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

Lösung ab Java 8 (kürzer)

```java
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:

```java
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; }

```

```
5       @Override public void accept(List<T> l) {
6           for (T x : l) action.accept(x);
7   }   }
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.

3. Java Streams

Streams - Einführung

Streams sind umgeformte Sammlungen, die durch die Umformung für funktional-orientiere Massen-Operationen geeignet sind.

Beispiel

```java
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

List<Integer> l = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
List<Integer> ll = l
    .stream() // list -> stream
    .filter(x -> x % 2 == 0) // filter list with predicate
    .map(x -> 10 * x) // map each element to a new one
    .collect(Collectors.toList()); // back to a list
ll.forEach(x -> System.out.println(x));
```

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

Streams mit primitiven Daten und Objekten

- `Stream<T>` ist der Typ der Streams mit Objekten vom Typ `T`
- Streams mit primitiven Daten:
  - `IntStream`
  - `LongStream`
  - `DoubleStream`

  Dies Streams mit primitiven Daten arbeiten in vielen Fällen effizienter jedoch sind manche Operationen nur auf `Object`-Streams erlaubt. „Primitive“ Streams können mit der Methode `boxed` in `Object`-Streams umgewandelt werden.

Beispiel

```
1 IntStream isPrim = IntStream.range(1, 10);
2 Stream<Integer> isObj = isPrim.boxed();
```

Erzeugung von Streams

Statische Methoden in `Arrays`

- Die Klasse `java.util.Arrays` hat mehrere überladene statische stream-Methoden, mit denen Arrays in Ströme umgewandelt werden können.
- Die Streams können Objekte oder primitive Daten enthalten.

Beispiel

```
// Stream of primitive data:
IntStream isP = Arrays.stream(new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 });
// Stream of objects:
Stream<Integer> isO = Arrays.stream(
    new Integer[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 }
);
```

Statische Methoden in `Stream`

- Das Interface `java.util.stream.Stream` enthält mehrere statische Methoden mit denen Streams erzeugt werden können.
- Für die Klassen der Streams mit primitiven Werten (z.B. `java.util.stream.IntStream`) gibt es äquivalente Methoden.
- Mit `of` werden die übergebenen Wert in einen Stream gepackt.
- Mit `iterate` und `generate` hat man eine einfache Möglichkeit unendliche Ströme zu erzeugen.

Beispiel

```
// Object-Stream 1, 2 ... 9, 0:
Stream<Integer> is1a = Stream.of(1,2,3,4,5,6,7,8,9,0);
```

```
// int-Stream 1, 2, ... 9, 0
IntStream is1b = IntStream.of(1,2,3,4,5,6,7,8,9,0);
```

```
// (infinite) Stream 1, 2, ...
Stream<Integer> is2 = Stream.iterate(1, ((x) -> x+1));
```

```
int[] z = new int[]{1};
Stream<Integer> is3 = Stream.generate((() -> z[0]++)); // (infinite) Stream 1, 2, ...
```

Statische range-Methoden in `IntStream` und `LongStream`

Die Interfaces `java.util.stream.IntStream` und `java.util.stream.LongStream` enthalten jeweils zwei statische `range`-Methoden mit denen Streams erzeugt werden können.

Beispiel

```
IntStream isPrimA = IntStream.range(1, 10); // 1,2, .. 9
IntStream isPrimA = IntStream.rangeClosed(1, 10); // 1,2, .. 9, 10
```

Nicht-statische Methoden der Collection-API

Das Interface `java.util.Collection` enthält die Methode stream mit der die jeweilige Kollektion in einen Stream umgewandelt werden kann.

Beispiel

```
Stream<Integer> is = Arrays.asList(1,2,3,4,5,6,7,8,9,0).stream();
```

Verwendung von Streams

Streams werden typischerweise in einer Pipeline-artigen Struktur genutzt:

1. Erzeugung
2. Folge von Verarbeitungs-/Transformationsschritten
3. Abschluss mit einer terminalen Operation

Verarbeitungsoperationen

Verarbeitungs-Operationen transformieren die Elemente eines Streams. Man unterscheidet:

- zustandslose Operationen
  Transformieren die Elemente jeweils völlig unabhängig von allen anderen.
- zustandsbehaftete Operationen
  Transformieren die Elemente abhängig von anderen.

Zustandslose Verarbeitungsoperationen

- `map(Function<? super T,? extends R> mapper)`: Transformiert jedes Element in ein anderes.
- `filter(Predicate<? super T> predicate)`: Filtert Elemente heraus.
- `flatMap(Function<? super T,? extends Stream<? extends R>> mapper)`: Transformiert jedes Element in einen Stream und fügt die Streams zusammen.

Beispiel

```java
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

List<Integer> is = IntStream.range(1, 10)
    .filter(i -> i % 2 == 0)
    .peek(i -> System.out.print(i+ " "))
    .map(i -> 10 * i)
    .boxed()
    .collect(Collectors.toList());
System.out.println(is);
```

Ausgabe: `1 3 5 7 9 [10, 30, 50, 70, 90]`

Beispiel

```java
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import java.util.stream.Stream;

static Stream<Integer> range(int from, int to) {
    return IntStream.range(from, to).boxed();
}

List<Integer> is = Stream.of(0, 1, 2)
    .flatMap(i -> range(10 * i, 10 * i + 10))
    .collect(Collectors.toList());
```

Ausgabe: `is ⟹ [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]`

Zustandsbehaftete Verarbeitungsoperationen

- `distinct()`: Entfernt Duplikate.
- `sorted()`: Sortiert die Elemente.

- `sorted(Comparator<? super T> comparator)`: Sortiert die Elemente mit einem gegebenen Comparator.
- `limit(long maxSize)`: Begrenzt die Anzahl der Elemente.
- `skip(long n)`: Überspringt die ersten n Elemente.

Beispiel

```
    import java.util.List;
    import java.util.stream.Collectors;
    import java.util.stream.Stream;

    List<Integer> lst = Stream.of(9, 0, 3, 1, 7, 3, 4, 7, 2, 8, 5, 0, 6, 2)
        .distinct()
        .sorted((i, j) -> i - j)
        .skip(1)
        .limit(3)
        .collect(Collectors.toList());
```

Ausgabe: `is ⟹ [1, 2, 3]`

Verarbeitungsoperationen

Eine terminale Operation hat im Gegensatz zu den Verarbeitungsoperationen keinen Stream als Ergebnis (`void`).

Terminale Operationen ohne Ergebnis

- `forEach(Consumer<? super T> action)`
  Wendet die übergebene Aktion auf alle Elemente des Streams an.

Terminale Operationen mit Ergebnis

- Operationen mit Array-Ergebnis: `Stream` ⇒ `Array`
  Operationen die den Stream in ein äquivalentes Array umwandeln.
- Operationen mit Kollektions-Ergebnis: `Stream` ⇒ `Kollektion`
  Operationen die den Stream in eine äquivalente Kollektion umwandeln.
- Operationen mit Einzel-Ergebnis: Aggregierende Operationen
  Operationen die den Stream zu einem einzigen Wert verarbeiten.

Beispiel

`forEach`

```
Stream.of(9, 0, 3, 1, 7, 3, 4, 7, 2, 8, 5, 0, 6, 2)
    .distinct()
    .sorted( (i,j) -> i-j )
    .limit(3)
    .forEach( System.out::println );
```

Beispiel

`toArray`

```
int[] a = IntStream.range(1, 3).toArray();

Object[] a = Stream.of("1", "2", "3").map( Integer::parseInt )
        .toArray();

Integer[] a = (Integer[]) Stream.of(1, 2, 3)
        .toArray();

String[] a = Stream.of(1, 2, 3).map( (i) -> i.toString() )
        .toArray( String[]::new ); // using generator
```

Terminale Operationen mit Kollektions-Ergebnis

- Die Methode `collect` erzeugt eine Kollektion aus den Elementen des Streams.
- `IntStream` und andere Streams mit primitiven Daten haben keine entsprechende Operation.
- Das Argument von `collect` ist ein *java.util.stream.Collector*. Die Erzeugung einer Kollektion ist damit Sonderfall einer aggregierenden Operation.
- Für die Erzeugung einer Kollektion verwendet man typischerweise einen vordefinierten `Collector` aus der Klasse *java.util.stream.Collectors*.
- Einfache Kollektionserzeuger in Collectors sind:

```
toList()
toSet()
toCollection(Supplier<C> collectionFactory)
```

Beispiel

`collect`

```
List<Integer> l1 = Stream.of(1, 2, 3).collect( Collectors.toList() );

List<Integer> l2 = IntStream.range(1, 4).boxed()
        .collect( Collectors.toList() );

Set<String> s1 = (Set<String>) Stream.of("1", "2", "3")
        .collect( Collectors.toSet());

Set<String> s2 = (Set<String>) Stream.of("1", "2", "3")
        .collect( Collectors.toCollection( HashSet::new) );
```

```
// Generating a map from a stream of strings

Map<String, Integer> m = Stream.of("1", "2", "3")
        .collect(
            Collectors.toMap(
                (s) -> s,
                Integer::parseInt
            )
        );
```

In `Collectors` finden sich **Kollektoren mit denen Maps erzeugt werden können**, die eine Gruppierung bzw. eine Partitionierung der Stream-Elemente darstellen:

- `static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(Function<? super T,? extends K> classifier)`

  Gruppiert die Elemente entsprechend einer Klassifizierungsfunktion.
- `static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(Predicate<? super T> predicate)`

  Partitioniert die Elemente entsprechend einem Prädikat.

Beispiel

`collect(groupingBy)`

Hilfreiche Methoden

```
import static java.util.stream.Collectors.groupingBy;
import static java.util.stream.Collectors.partitioningBy;
import static java.util.stream.Collectors.counting;
```

```
Map<Integer, List<Integer>> groupedByMod3 = Stream.of(1, 2, 3, 4, 5, 6 ,7 ,8, 9)
        .collect( groupingBy( (x) -> x%3 ) );
```

Ausgabe: `groupedByMod3 = {0=[3, 6, 9], 1=[1, 4, 7], 2=[2, 5,8]}`

```
Map<Integer, List<String>> groupedByLength = Stream.of(
            "one", "two", "three", "four", "five", "six", "seven", "eight")
        .collect( groupingBy( (s) -> s.length() ) );
```

Ausgabe: `groupedByLength ⟹ {3=[one, two, six], 4=[four, five], 5=[three, seven, eight]}`

Das Interface `Stream` bzw. die Interfaces für Ströme primitiver Daten (`IntStream`, etc.) bieten einige **einfache aggregierende Funktionen für Standardoperationen** auf allen Elementen des Stroms.

Beispiel

```
long count = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9).count();

long sum = IntStream.of(1, 2, 3, 4, 5, 6, 7, 8, 9).sum();

OptionalDouble av = IntStream.of(1, 2, 3, 4, 5, 6, 7, 8, 9).average();
```

Das Interface Stream bieten einige einfache **aggregierende Funktionen für den Test aller Elemente des Stroms** mit einem übergebenen Prädikat.

Beispiel

```
boolean anyEven = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9)
        .anyMatch( (x) -> x%2 == 0 );

boolean allEven = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9)
        .allMatch( (x) -> x%2 == 0 );

boolean noneEven = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9)
        .noneMatch( (x) -> x%2 == 0 );
```

Das Interface `Stream` bietet die Funktionen `findFirst` und `findAny` für die „Suche“ nach dem ersten bzw. irgendeinem Element in einem Stream.

Achtung!

Diese Methoden haben kein Prädikat als Parameter. Es empfiehlt sich darum den `Stream` vorher mit dem entsprechenden Prädikat zu filtern.

Beispiel

```
Optional<Integer> firstEven = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9)
        .filter( (x) -> x%2 == 0 )
        .findFirst();
```

Ausgabe: `firstEven ⟹ Optional[2]`

Das Interface Stream bietet die Funktion

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

mit der eine Funktion auf jedes Element und das bisherige Zwischenergebnis angewendet werden kann.

Falls der erste Wert nicht der Startwert sein soll, verwendet man:

```
Optional<T> reduce(T identity, BinaryOperator<T> accumulator)
```

Beispiel

`reduce`

```
Optional<Integer> sumOfAll = Stream.of(1, 2, 3, 4, 5).reduce( (a, x) -> a+x );
```

Ausgabe: `sumOfAll ⟹ Optional[15]`

```java
Optional<Integer> subOfAll = Stream.of(1, 2, 3, 4, 5).reduce( (a, x) -> a-x );
```

Ausgabe: `subOfAll ⟹ Optional[-13]`

```java
int sumOfAllPlus100 = Stream.of(1, 2, 3, 4, 5)
        .reduce(100, (a, x) -> a+x );
```

Ausgabe: `sumOfAllPlus100 ⟹ 115`

Es gibt einen Kollektor mit dem String-Elemente zu einem String konkateniert werden können:

```java
static Collector<CharSequence,?,String> joining(CharSequence delimiter)
```

Beispiel

reduce

```java
String concat = Stream.of("one", "two", "three")
        .collect( joining("+") );
```

Ausgabe: `concat = one+two+three`

Java `Optionals`

Instanzen der Klasse `java.util.Optional<T>` (bzw. `java.util.OptionalInt` etc.) **repräsentieren Werte die vorhanden sind oder auch nicht**.

Insbesondere `java.util.Optional<T>` kann/sollte anstelle von `null` verwendet werden, in Fällen in denen unter bestimmten Umständen kein sinnvoller Wert angegeben werden kann.

Bemerkung

Es gibt moderne Programmiersprachen, die auf `null` komplett verzichten und stattdessen immer auf `Optionals` oder ähnliche Konstrukte setzen.

Beispiel

```
static Optional<Integer> min(int[] a ) {
    if(a == null || a.length == 0)
        return Optional.empty();

    int min = a[0];
    for(int x: a) { if (x < min) { min = x; } }
    return Optional.of(min);
}
```