



ISA

Institut für
SoftwareArchitektur



THM

TECHNISCHE HOCHSCHULE MITTELHESSEN



Funktionale Programmierung in Java 8

Kurzeinführung

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Was ist das

Ein *Lambda*, (auch *Closure*)

ist ein Ausdruck, dessen Wert eine Funktion ist.

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

Beispiel

Angenommen wir haben eine Klasse `Person` und eine Liste von `Person`en, die nach Alter sortiert werden soll. Dazu muss eine Vergleichsfunktion übergeben werden. In Java <8 kommt dazu nur ein Objekt in Frage. Die Vergleichsfunktion muss in ein Objekt einer passend zu definierenden Klasse gepackt werden. In Java 8 kann die Funktion „direkt“ übergeben werden.

```
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 + "];"
    }
}
```

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

```
Collections.sort(
    persons,
    new Comparator<Person>() {
        @Override
        public int compare(Person o1, Person o2) {
            return o1.getAge() - o2.getAge();
        }
    });
```

```
Collections.sort(
    persons,
    (Person o1, Person o2)
        -> { return o1.getAge() - o2.getAge(); }
);
```

oder kürzer noch:

```
Collections.sort(
    persons,
    (o1, o2) -> o1.getAge() - o2.getAge()
);
```

Lambdas sind Closures

Ein *Lambda-Ausdruck*

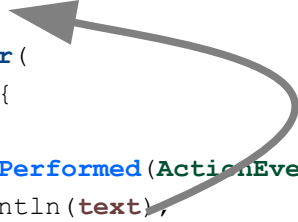
ist ein Ausdruck dessen Wert eine Funktion ist: ein funktionaler Ausdruck.

Closure

nennt man Funktionsausdrücke, deren freie Variablen an Definitionsstelle gebunden werden.


Das ist etwas völlig normales und gilt bekanntlich für innere Klassen:

```
 JButton button = new JButton("Click ME!");  
 String text = "Klick!";  
 button.addActionListener(  
     new ActionListener() {  
         @Override  
         public void actionPerformed(ActionEvent e) {  
             System.out.println(text);  
         }  
     });
```



Der Name `text` ist innerhalb des ActionListeners frei: er wird außerhalb definiert. Trotzdem wird der Text "Klick!" ausgegeben, auch wenn an der Aufrufstelle eine andere Definition für `text` gültig ist.

```
 JButton button = new JButton("Click ME!");  
 String text = "Klick!";  
 button.addActionListener(  
     (e) -> System.out.println(text);  
 );
```



Diese Bindung „in den Kontext der Definition“ gilt natürlich auch, wenn mit Lambdas gearbeitet wird

Lambdas

Typen von Lambdas

Object

Bis zu Java 7 ist **Object** der Basis aller Referenztypen: Jeder Typ ist mit **Object** kompatibel. Lambdas haben einen Typ der nicht mit **Object** kompatibel ist:

```
Object actionListener_1 = new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        System.out.println(text);  
    }  
};  
  
Object actionListener_2 = (e) -> System.out.println(text);
```

OK!

The target type of this expression must be a functional interface

Fehler!

```
Object actionListener_2 = (e) -> System.out.println(text);
```

```
Consumer<ActionEvent> actionListener_2 = (e) -> System.out.println(text);
```

OK, kann aber nicht an `addActionListener` übergeben werden.

```
ActionListener actionListener_2 = (e) -> System.out.println(text);
```

OK! kann auch an `addActionListener` übergeben werden.

Functional Interface

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 ActionListener {
    int magicNumber = 42;
}

public static void main(String[] args) {
    JButton button1 = new JButton("Click me!");
    JButton button2 = new JButton("Click me too!");

    String text = "Klick!";

    MyActionListener actionListener_1 = new MyActionListener() {
        @Override public void actionPerformed(ActionEvent e) { System.out.println(text + magicNumber); }
    };

    MyActionListener actionListener_2 = (e) -> System.out.println(text + MyActionListener.magicNumber);

    button1.addActionListener(actionListener_1);
    button2.addActionListener(actionListener_2);

    JOptionPane.showMessageDialog(
        null,
        new Object[]{button1,button2},
        "Button Frame",
        JOptionPane.INFORMATION_MESSAGE,
        null);
}
```

Functional Interface

Vordefinierte Functional Interfaces

Im *Package* `java.util.function` sind etliche funktionale Interfaces vordefiniert. Beispielsweise

- **`Consumer<T>`** Konsument von Werten vom Typ T
Das Interface von funktionalen Objekten, die mit der void Methode `accept` einen Wert vom Typ T annehmen.
- **`Predicate<T>`** Prädikat das Werte vom Typ T testet
Das Interface von funktionalen Objekten, die mit der booleschen Methode `test` einen Wert vom Typ T annehmen und testen.
- **`Function<T,R>`** Funktion vom Typ $T \Rightarrow R$
Das Interface von funktionalen Objekten, die mit einer Methode `apply` Wert vom Typ T annehmen und Werte vom Typ R liefern.

Functional Interface

Vordefinierte Functional Interfaces, Beispiel Predicate

```
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;
import java.util.function.Predicate;

public class Lambda_Ex {

    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;
    }

    public static void main(String[] args) {
        List<Integer> l = Arrays.asList(1,2,3,4,5,6,7,8,9);
        System.out.println(
            filterList(
                l,
                (x) -> x%2 == 0
            ));
    }
}
```



[2, 4, 6, 8]

Functional Interface

Vordefinierte Functional Interfaces, Beispiel **Consumer**

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;


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);
        }
    }
}

public class Lambda_Ex {

    public static void main(String[] args) {
        List<Integer> l = Arrays.asList(1,2,3,4,5,6,7,8,9);
        WorkerOnList<Integer> worker = new WorkerOnList<>( (i) -> System.out.println(i*10) );
        worker.accept(l);
    }
}
```



10
20
30
40
50
60
70
80
90

Functional Interface

Vordefinierte Functional Interfaces, Beispiel `Function`

```
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;
import java.util.function.Function;

class ListTransformer<T,R> implements Function<List<T>, List<R>> {
    private Function<T, R> fun;

    public ListTransformer(Function<T, R> fun) {
        this.fun = fun;
    }

    @Override
    public List<R> apply(List<T> l) {
        List<R> res = new LinkedList<>();
        for (T x: l) {
            res.add(fun.apply(x));
        }
        return res;
    }
}

public class Lambda_Ex {

    public static void main(String[] args) {
        List<Integer> l = Arrays.asList(1,2,3,4,5,6,7,8,9);
        ListTransformer<Integer, Integer> worker = new ListTransformer<>( (i) -> i*10 );
        System.out.println(worker.apply(l));
    }
}
```

 [10, 20, 30, 40, 50, 60, 70, 80, 90]

Referenzen auf Methoden

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

```
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;
import java.util.function.Predicate;

public class Lambda_Ex {

    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 even(int x) {
        return x % 2 == 0;
    }

    public static void main(String[] args) {
        List<Integer> l = Arrays.asList(1,2,3,4,5,6,7,8,9);
        System.out.println(
            filterList(
                l,
                Lambda_Ex::even
            ));
    }
}
```

Übergabe einer statischen Methode als Prädikat

Referenzen auf Methoden

Als Implementierung eines funktionalen Interfaces (als „Lambda“) können auch Methoden verwendet werden. – Das gilt auch für nicht-statische Methoden.

```
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;
import java.util.function.Predicate;

class Tester {
    private int magicNumber;

    public Tester(int magicNumber) { this.magicNumber = magicNumber; }

    boolean isMagic(int x) { return x == magicNumber; }
}

public class Lambda_Ex_7 {

    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;
    }

    public static void main(String[] args) {
        List<Integer> l = Arrays.asList(1,2,3,4,5,6,7,8,9);
        System.out.println(
            filterList(
                l,
                new Tester(5)::isMagic
            ));
    }
}
```

Übergabe einer Objekt-Methode als Prädikat

Erweiterungen bei Collections

API-Erweiterungen bei Collections

Die Collection-API wurde erweitert, um so den Ausdrucksmöglichkeiten von Lambdas gerecht zu werden.


forEach(Consumer<T> c) :

Mit der Methode forEach kann eine Kollektion „im funktionalen Stil“ durchlaufen werden. Der übergebene Consumer wird dabei auf jedes Element der Liste angewendet.

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

public class Lambda_Ex {

    public static void main(String[] args) {
        List<Integer> l = Arrays.asList(1,2,3,4,5,6,7,8,9);
        l.forEach( x -> System.out.println(x) );
    }
}
```



1
2
3
4
4
6
7
8
9

Erweiterungen bei Collections

API-Erweiterungen bei Collections

removeIf(Predicate<T> p) :

Mit der Methode `removeIf` können in einer Kollektion alle Element entfernt werden, die nicht zu einem Prädikat passen.

```
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

public class Lambda_Ex {

    public static void main(String[] args) {
        List<Integer> l = new LinkedList<>(Arrays.asList(1,2,3,4,5,6,7,8,9));
        l.removeIf( x -> x % 2 != 0 );
        l.forEach( x -> System.out.println(x) );
    }
}
```



2
4
6
8

Erweiterungen bei Collections

API-Erweiterungen bei Listen


replaceAll(UnaryOperator<T> operator) :

Listen haben mit `replaceAll` eine weitere Methode zur „Massenverarbeitung“. Mit ihr können alle Listenelemente auf einfache Art ersetzt / umgewandelt werden.

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

public class Lambda_Ex {

    public static void main(String[] args) {
        List<Integer> l = Arrays.asList(1,2,3,4,5,6,7,8,9);
        l.replaceAll( x -> 2 * x );
        l.forEach( x -> System.out.println(x) );
    }
}
```



2
4
6
8
10
12
14
16
18

Streams

Streams

Streams sind

- umgeformte Kollektionen, die durch die Umformung
- für funktional-orientierte Massen-Operationen geeignet sind

Beispiel


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

public class Stream_Ex_1 {

    public static void main(String[] args) {
        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) );
    }
}
```



20
40
60
80

Streams

Stream 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
Manche Operationen sind nur auf Objekt-Streams erlaubt

„Primitive“ Stream können mit der Methode `boxed` in Objekt-Streams umgewandelt werden.

Beispiel

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

Streams

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

```
import java.util.Arrays;
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class Stream_Ex_2 {

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

Streams

Erzeugung von Streams

statische Methoden in Stream

- Das Interface `java.util.stream.Stream` enthält mehrere statische Methoden (*ja in Java 8 gibt es statische Methoden in Interfaces*) 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

```
Stream<Integer> is1a = Stream.of(1,2,3,4,5,6,7,8,9,0); // Object-Stream 1, 2, ... 9, 0
IntStream      is1b = IntStream.of(1,2,3,4,5,6,7,8,9,0); // int-Stream 1, 2, ... 9, 0
Stream<Integer> is2 = Stream.iterate(1, ((x) -> x+1)); // (infinite) Stream 1, 2, ...

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

Streams

Erzeugung von Streams

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
```

Streams

Erzeugung von Streams

nicht-statische Methoden der Kollektionen

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();
```

Streams – Pipeline-Operationen

Verarbeitungsoperationen

Streams werden typischerweise in einer Pipeline-artigen Struktur genutzt:

- *Erzeugung*
- *Folge von Verarbeitungs-/Transformationsschritten*
- *Abschluss mit einer terminalen Operation*

Wir betrachten nur einige wichtige Verarbeitungs-Operationen

Verarbeitungsoperationen

Verarbeitungs-Operationen transformieren die Elemente eines Streams

Man unterscheidet:

- **Zustanslose Operationen**
Transformieren die Elemente jeweils völlig unabhängig von allen anderen
- **Zustandsbehaftete Operationen**
Transformieren die Elemente abhängig an anderen

Verarbeitungsoperationen

Zustandslos Verarbeitungsoperationen

- `filter(Predicate<T> pred)`

Entfernt alle Elemente für die das übergebene Prädikat *false* liefert / Belässt alle Elemente im Stream für die das übergebene Prädikat *true* liefert

- `map(Function<? super T,? extends R> mapper)`

Wendet auf jedes Element die übergebene Funktion an.

- `flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)`

Wendet auf jedes Element die übergebene – Stream-erzeugende – Funktion an und „klopft die entstehenden Stream flach“, d.h. verknüpft sie zu einem einzigen Stream

- `peek(Consumer<? super T> action)`

Wendet die übergebene ergebnislose Funktion auf alle Elemente an, ohne dabei den Stream selbst zu verändern (Debug-Hilfe)

Streams – Pipeline-Operationen

Zustandslose Verarbeitungsoperationen

Beispiel – 1

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


public class Stream_Ex_3 {

    public static void main(String[] args) {

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

        System.out.println(is);

    }
}
```



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

Zustandslose Verarbeitungsoperationen

Beispiel – 2

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

public class Stream_Ex_4 {

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

    public static void main(String[] args) {

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

        System.out.println(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]

Streams – Pipeline-Operationen

Verarbeitungsoperationen

Zustandsbehaftete Verarbeitungsoperationen

- `distinct()`
Entfernt alle Duplikate aus einem Stream
- `sorted()`
Sortiert die Elemente eines Stream nach ihrer natürlichen Ordnung.
- `sorted(Comparator<? super T> comparator)`
Sortiert die Elemente eines Streams mit dem übergeben Comperator
- `limit(long maxSize)`
Beschränkt die Zahl der Elemente eines Stroms auf die übergebene Maximalzahl
- `skip(long n)`
Entfernt die ersten n Elemente eines Stroms

Streams – Pipeline-Operationen

Zustandsbehaftete Verarbeitungsoperationen


Beispiel

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

public class Stream_Ex_5 {

    public static void main(String[] args) {
        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() );

        System.out.println(lst);
    }
}
```

 [1, 2, 3]

Verarbeitungs-Operation

Übersicht:

- filter
- map, mapToXY
- flatMap, flatMapToXY
- distinct
- sorted
- peek
- limit
- skip

Streams – Pipeline-Operationen

Terminale Operationen

Streams werden typischerweise in einer Pipeline-artigen Struktur genutzt:

- *Erzeugung*
- *Folge von Verarbeitungs-/Transformationsschritten*
- *Abschluss mit einer terminalen Operation*

Die terminale Operation hat im Gegensatz zu den Verarbeitungsoperationen keinen Stream als Ergebnis.

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

Streams – Pipeline-Operationen

Terminale Operation ohne Ergebnis

forEach

Die Methode `forEach` schließt einen Stream mit einer Aktion ab, die auf jedes Element angewendet wird.

Beispiel

```
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 );
```

Streams – Pipeline-Operationen

Terminale Operation mit Array-Ergebnis

toArray

- **Object[] toArray()**

Die Methode `toArray` erzeugt einen Array aus den Elementen des Streams.

- **IntStream** und andere Streams mit primitiven Daten werden in Arrays mit primitiven Daten umgewandelt.
- Streams die Objekte enthalten werden in Object-Arrays umgewandelt.
- **A[] toArray(IntFunction<A[]> generator)**

Mit einem Parameter kann die Operation angegeben werden, mit der das Arrays erzeugt wird.

Beispiele:

```
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 );
```

Terminale Operation mit Kollektions-Ergebnis

collect

- 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)`

Beispiele:

```
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 ) );
```

Streams – Pipeline-Operationen

Terminale Operation mit Kollektions-Ergebnis

Map mit collect erzeugen

- In **Collectors** findet sich auch Kollektoren mit denen Maps erzeugt werden können.

Die einfachste ist:

```
toMap( Function<? super T,? extends K> keyMapper,  
       Function<? super T,? extends U> valueMapper)
```

Beispiel:

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

Terminale Operation mit Kollektions-Ergebnis

Gruppieren und partitionieren mit collect

In **Collectors** findet 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)`
gruppirt die Elemente entsprechend einer Klassifizierungsfunktion
- `static <T> Collector<T,?,Map<Boolean,List<T>>>`
`partitioningBy(`
 `Predicate<? super T> predicate)`
partitioniert die Elemente entsprechend einem Prädikat

Streams – Pipeline-Operationen

Terminale Operation mit Kollektions-Ergebnis

Gruppieren und partitionieren mit collect Beispiele:

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

public class Stream_Ex_10 {

    public static void main(String[] args) {

        Map<Boolean, List<Integer>> oddAndEven = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 0)
            .collect( partitioningBy( (x) -> x%2 == 0 ) );

        {false=[1, 3, 5, 7, 9], true=[2, 4, 6, 8, 0]}

        Map<Integer, List<Integer>> groupedMod3 = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 0)
            .collect( groupingBy( (x) -> x%3 ) );

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

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

        {3=[one, two, six], 4=[four, five, nine], 5=[three, seven, eight]}

        Map<Integer, Long> countGroupsByLength = Stream.of("one", "two", "three", "four",
            "five", "six", "seven", "eight", "nine")
            .collect( groupingBy( String::length, counting() ) );

        {3=3, 4=3, 5=3}
    }
}
```

Streams – Pipeline-Operationen

Aggregierende terminale Operation

Summe, Minimum, Maximum, Durchschnitt

Das Interface **Stream** bzw. die Interfaces für Ströme primitiver Daten **xyStream** bieten einige einfache aggregierende Funktionen für Standardoperationen auf allen Elementen des Stroms.

Beispiel:

```
import java.util.OptionalDouble;
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class Stream_Ex_7 {

    public static void main(String[] args) {
        long count = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9)
            .count();

        System.out.println("count = " + count);

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

        System.out.println("sum = " + sum);

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

        System.out.println("average = " + av);
    }
}
```

Streams – Pipeline-Operationen

Aggregierende terminale Operation

testende Operationen: und/oder–Aggregationen

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 );
```

Streams – Pipeline-Operationen

Aggregierende terminale Operation

suchende Operationen

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

Interessanterweise haben diese Methoden 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();

System.out.println(
    firstEven.isPresent() ?
    firstEven.get() :
    "no even number found" );
```

Streams – Pipeline-Operationen

Aggregierende terminale Operation

prüfende („*matchende*“) Operationen

Das Interface **Stream** bietet die Funktionen **allMatch**, **anyMatch** und **noneMatch** für die Prüfung der Elemente eines Streams mit einem Prädikat.

Beispiel:

```
boolean allEven = Stream.of(2, 4, 6)
    .allMatch( (x) -> x%2 == 0 );

System.out.println(allEven); // => true

boolean anyEven = Stream.of(1, 2, 3, 4)
    .anyMatch( (x) -> x%2 == 0 );

System.out.println(anyEven); // => true

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

System.out.println(noneEven); // => false
```

Streams – Pipeline-Operationen

Aggregierende terminale Operation

reduzierende Operationen

- 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:

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

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

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

System.out.println(sumOfAll); // (((((1+2)+3)+4)+5)
System.out.println(subOfAll); // (((((1-2)-3)-4)-5)
System.out.println(sumOfAllPlus100); // (((((100+1)-2)-3)-4)-5)
```



```
Optional[15]
Optional[-13]
115
```

Streams – Pipeline-Operationen

Aggregierende terminale Operation

Strings zu einem String reduzierende Operation

In **Collectors** findet sich ein Kollektor mit dem String-Elemente zu einem String konkateniert werden können:

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

Beispiel

```
String concat = Stream.of("one", "two", "three", "four", "five", "six", "seven", "eight", "nine")  
    .collect( joining("+") );  
  
System.out.println(concat); // => one+two+three+four+five+six+seven+eight+nine
```

Streams – Pipeline-Operationen

Iterator

Streams unterstützen die `iterator`-Methode. Sie liefert einen iterator über die Elemente des Streams.

Beispiel

```
Iterator<Integer> iter = Stream.of(1, 2, 3, 4, 5).iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

Terminale Operation

Übersicht:

- `toArray`
- `forEach`, `forEachOrdered`
- `collect`
- `reduce`
- `min`, `max`, `count`
- `anyMatch`, `allMatch`, `noneMatch`
- `findFirst`, `findAny`, `findNone`
- `iterator`

Beispiel: Fibonacci-Zahlen

```
import java.util.Iterator;
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class Fibonacci {


    static IntStream fibs() {
        int[] start = {1, 1};
        Stream<int[]> pairStream = Stream.iterate(start, (int[] p) -> new int[]{p[1], p[0]+p[1]});
        return pairStream.mapToInt((p) -> p[1]);
    }

    static int getNthFib(int n) {
        return fibs().skip(n-1).findFirst().getAsInt();
    }

    public static void main(String[] args) {
        Iterator<Integer> iter = fibs().iterator();

        int i = 1;
        while (iter.hasNext()) {
            int f = iter.next();
            System.out.println(i++ + " : " + f);
            if (i >= 10) break;
        }

        for (int n=1; n<10; n++) {
            System.out.println(n + " : " + getNthFib(n));
        }
    }
}
```



```
1 : 1
2 : 2
3 : 3
4 : 5
5 : 8
6 : 13
7 : 21
8 : 34
9 : 55
1 : 1
2 : 2
3 : 3
4 : 5
5 : 8
6 : 13
7 : 21
8 : 34
9 : 55
```

Klasse Optional<T>

optionaler Wert: Wert oder kein Wert

Instanzen der Klasse `java.util.Optional<T>` repräsentieren Werte die vorhanden sind oder auch nicht.

Sie kann anstelle von `null` verwendet werden, in Fällen in denen unter bestimmten Umständen kein sinnvoller Wert angegeben werden kann.

Beispiel:

```
import java.util.Optional;

public class Option_Ex {

    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);
    }

    public static void main(String[] args) {
        int[][] aa = {{9,1,2,8,7,4}, {}, null};
        for (int[] a: aa) {
            Optional<Integer> minO = min(a);
            System.out.println(
                minO.isPresent() ?
                    minO.get()
                    : "can not compute min value"
            );
        }
    }
}
```

Klasse OptionalXY

optionaler primitiver Wert: primitiver Wert oder kein Wert

Instanzen der Klasse `java.util.OptionalXY` für primitive Typen XY repräsentieren primitive Werte die vorhanden sind oder auch nicht.

Beispiel:

```
import java.util.Arrays;
import java.util.OptionalInt;

public class Option_Ex {

    static OptionalInt minP(int[] a ) {
        if(a == null) return OptionalInt.empty();
        return Arrays.stream(a).min();
    }

    public static void main(String[] args) {
        int[][] aa = {{9,1,2,8,7,4}, {}, null};
        for (int[] a: aa) {
            OptionalInt minO = minP(a);
            System.out.println(
                minO.isPresent() ?
                    minO.getAsInt()
                    : "can not compute min value"
            );
        }
    }
}
```

Quellen

Diese Folien beruhen im Wesentlichen auf

- Java-8 API-Dokumentation
- M. Inden: *Java 8 Die Neuerungen*, dpunkt Verlag, 2014