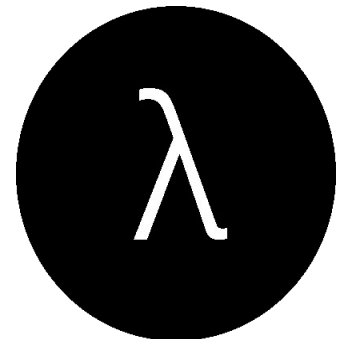


Funktionale Programmierung in Java



Lambda Calculus

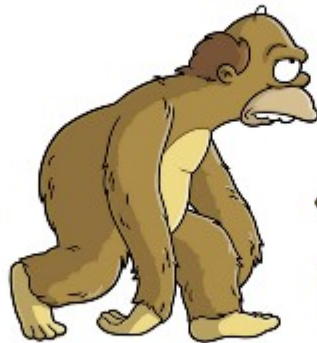




MACHINE



ASSEMBLY



PROCEDURAL



OBJECT ORIENTED



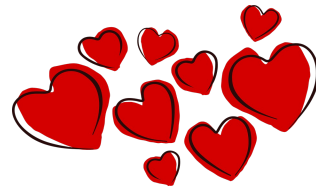
FUNCTIONAL

Übersicht

- Geschichte
- Konzepte
- Lambda-Ausdrücke
- funktionale Interfaces
- Streams + Operations
- Methodenreferenzen

Geschichte der funktionalen Programmierung

- 30s => Lambda-Kalkül
- 60s => die Programmiersprache LISP (inspiriert vom Lambda-Kalkül)
-
- Heute =>
 - Erlang
 - Haskell
 - Scala
 - ...
 - + (JS, Java,...??)



Funktionales Programmieren

- basiert auf Mathematik
- ist deklarativ (nicht imperativ)
- Innerhalb des Paradigmas ist der Ablauf eines Programms nicht definiert
- In eingeschränkter Form ist funktionale Programmierung die Programmierung ohne veränderliche Variablen, ohne Zuweisung und ohne Kontrollstrukturen.

Eigenschaften funktionaler Sprachen

- Keine Variablen
- keine Seiteneffekte (pure functions)
- keine Zuweisung
- keine imperativen Kontrollstrukturen
- Funktionen => First Class Citizen

Forget everything you know, you must. Yeesssssss.



Funktionen => First Class Citizen

- Wie alle Daten können Funktionen innerhalb von Funktionen definiert werden
- Wie alle Daten können Funktionen an Funktionen übergeben und zurückgegeben werden
- Funktionen lassen sich mit anderen Funktionen zu neuen Funktionen verknüpfen
- Insbesondere gibt es Funktionen, die andere Funktionen auf Datenstrukturen anwenden

Funktionales Programmieren

- Programmieren = **Definition von Funktionen**
- Ausführung = **Auswerten von Ausdrücken**
- Resultat **Einziges Resultat** ist der **Rückgabewert**

Resultat = Funktion($\text{Argument}_1, \dots, \text{Argument}_n$)

Lambdas

- **ist ein Ausdruck, dessen Wert eine Funktion ist**

```
btn.setOnAction(new EventHandler<ActionEvent>() {  
    @Override  
    public void handle(ActionEvent event) {  
        System.out.println("Hello World!");  
    }  
});
```

```
btn.setOnAction(  
    event -> System.out.println("Hello World!")  
);
```

Lambda Ausdrucks

- Java Lambdas **stammen von funktionalen Sprachen**
- **Anonyme Methode**
- **Prägnante Syntax, weniger Code, lesbarer**
- **ad-hoc Implementierung von Funktionalität**

lambda = ArgList "->" Body

(int x, int y) -> { return x+y; }

ArgList = Identifier

| "(" Identifier ["," Identifier]* ")"

| "(" Type Identifier ["," Type Identifier]* ")"

Body = Expression

| "{" [Statement ";"]+ "}"

Beispiele

- `(int x, int y) -> { return x+y; }`
- `// Argument type is inferred:`
`(x, y) -> { return x+y; }`
- `// No brackets needed if only one argument`
`x -> { return x+1; }`
- `// No arguments needed`
`() -> { System.out.println("I am a Runnable"); }`

Beispiele

- // Lambda using a statement block
a -> {
 if (a.balance() < limit) a.alert();
 else a.okay();
}
- // Single expression
a -> (a.balance() < limit) ? a.alert() : a.okay()
- // returns Account
(Account a) -> { return a; }
- // returns int
() -> 5;

Beispiele

```
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)
);
```

Beispiele

```
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(); }
);

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


Typen von Lambdas

- Functional Interface
- \approx Interface mit einer abstrakten Methode

```
Consumer<Account> myLambda =  
    (Account a) -> {if (a.balance() < limit)  
a.alert();  
};
```

Functional Interface

- Vordefinierte Functional Interfaces
- Consumer: Kein Resultat
- Function: Produziert Resultat
- Operator: Produziert Resultat vom Argument-Typ
- Supplier: Produziert Resultat ohne Argument
- Predicate: Produziert boolean-Resultat

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
            ));
    }
}
```

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 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);
    }
}
```

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));
    }
}
```

Methoden-Referenzen

- existierende Methode (einer Klasse) als LambdaAusdruck verwenden können
- Brauchen Kontext, damit korrekter Ziel-Typ abgeleitet werden kann
- Weniger Code, verständlicher

Methode als 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;
    }
    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
        ));
}
```

Erweiterungen bei Collections

forEach(Consumer<T> c)

```
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) );
    }
}
```


Erweiterungen bei Collections

removeIf(Predicate<T> p)

```
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));
        l.removeIf( x -> x % 2 != 0 );
        l.forEach( x -> System.out.println(x) );
    }
}
```

Erweiterungen bei Collections

replaceAll(UnaryOperator<T> operator)

```
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) );
    }
}
```

Scoping

- Lambdas haben Zugriff auf lokale Variablen vom umschliessenden Scope
- Lambdas führen keinen neuen Scope ein
- Variablen müssen final sein

```
int x = 5;
```

```
return y -> { x = 6; return x + y; };
```

```
// does not work!
```

Streams

- `interface java.util.stream.Stream<T>`
- A sequence of elements supporting sequential and parallel aggregate operations
- Funktionales Bearbeiten und Behandeln von Sequenzen
- Streams unterstützen filter, map, reduce, findAny, skip, peek
- Haben nichts mit `java.io.InputStream`, resp. `java.io.OutputStream` zu tun! -> als ob! ^^

Streams

- `Stream<T>` ist der Typ der Streams mit Objekten vom Typ `T`
- Streams für Elemente von diesen drei primitiven Datentypen
 - `IntStream`
 - `DoubleStream`
 - `LongStream`
- diese „primitive“ Versionen erben nicht von `Stream`
- Dies Streams mit primitiven Daten arbeiten in vielen Fällen effizienter

Beispiel

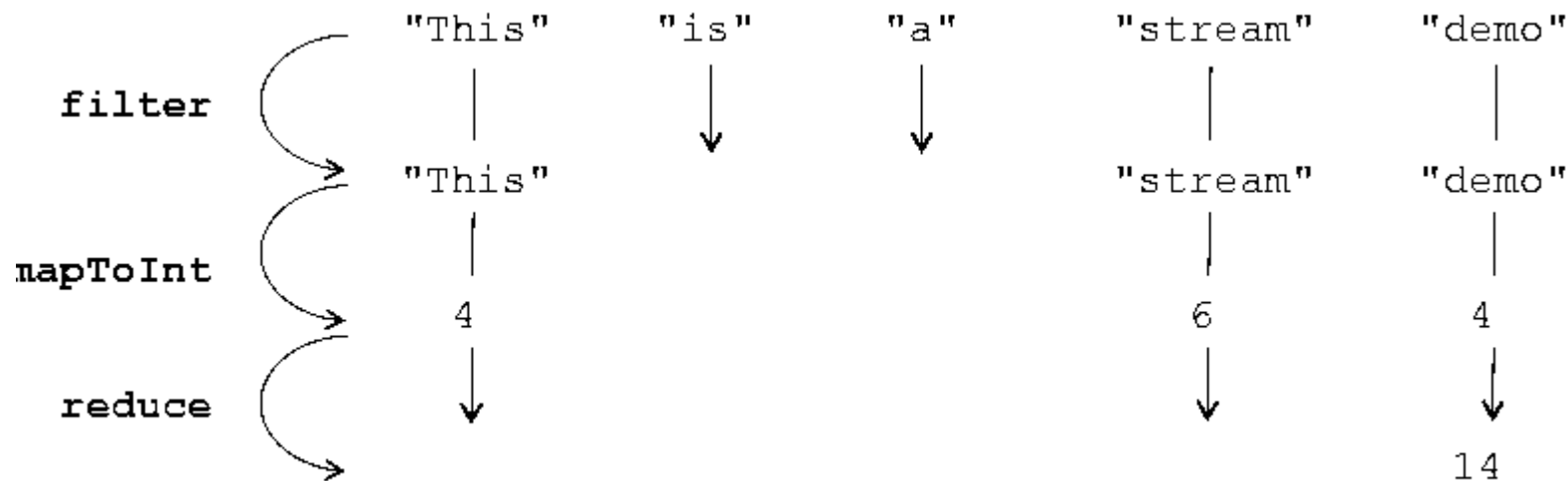
```
String[] txt = { "This", "is", "a", "stream", "demo"};
```

```
Arrays.stream(txt)
```

```
.filter(s -> s.length() > 3)
```

```
.mapToInt(s -> s.length())
```

```
.reduce(0, (l1, l2) -> l1 + l2);
```



Beispiel

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

public class Ex {
    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 )
            .map( (x) -> 10*x )
            .collect( Collectors.toList() ); // back to a list

        ll.forEach( x -> System.out.println(x) );
    }
}
```

Erzeugung von Streams

- **statische Methoden in Arrays**

```
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
```


Erzeugung von Streams

- **statische Methoden in `java.util.stream.Stream`**
- **mit `iterate` und `generate` hat man eine einfache Möglichkeit unendliche Ströme zu erzeugen**

```
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, ...
```

Erzeugung von Streams

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

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

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

Erzeugung von Streams

- nicht-statische Methoden der Kollektionen
- Das Interface `Collection` enthält die Methode `stream` mit der die jeweilige Kollektion in einen Stream umgewandelt werden kann.

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

Pipeline-Operationen

- Streams werden typischerweise in einer Pipeline-artigen Struktur genutzt
- Verarbeitungs-Operationen transformieren die Elemente eines Streams
- Erzeugung
- Folge von Verarbeitungs-/Transformationsschritten
- Abschluss mit einer terminalen Operation

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.

Verarbeitungsoperationen

- `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“

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

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

Beispiel

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

public class Ex {
    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);
    }
}
```

Beispiel

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

public class Ex{
    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);
    }
}
```


Verarbeitungsoperationen

- Zustandsbehaftete Verarbeitungsoperationen
- `distinct()`
- `sorted()`

Sortiert die Elemente eines Stream nach ihrer natürlichen Ordnung.

- `sorted(Comparator<? super T> comparator)`
- `limit(long maxSize)`
- `skip(long n)`

Beispiel

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

public class Ex {
    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);
    }
}
```

Streams

- Es gibt Stream-Operationen, welche
- wieder einen Stream produzieren: `filter()`, `map()`
 - intermediate
- etwas anderes tun: `forEach()`, `reduce()`, `collect()`
 - terminal
- intermediate-Streams werden nicht direkt ausgewertet

Terminale Operation

- ohne Ergebnis
- `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 );
```

Terminale Operation

- mit Array-Ergebnis
- toArray
- **Die Methode toArray erzeugt einen Array aus den Elementen des Streams**

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

Terminale Operation

- mit Kollektions-Ergebnis
- collect
- erzeugt eine Kollektion aus den Elementen des Streams
- Für die Erzeugung einer Kollektion verwendet man typischerweise einen vordefinierten Collector **aus** `java.util.stream.Collectors`.

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

Terminale Operation

- mit Kollektions-Ergebnis
- collect
- erzeugt eine Kollektion aus den Elementen des Streams
- In Collectors findet sich auch Kollektoren mit denen Maps erzeugt werden können

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

Terminale Operation

- Gruppieren und partitionieren

```
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 Ex {
    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 ) );
        Map<Integer, List<Integer>> groupedMod3 =
            Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 0)
                .collect( groupingBy( (x) -> x%3 ) );
        Map<Integer, List<String>> groupedByLength =
            Stream.of("one", "two", "three", "four",
                    "five", "six", "seven", "eight", "nine")
                .collect( groupingBy( (s) -> s.length() ) );
        Map<Integer, Long> countGroupsByLength =
            Stream.of("one", "two", "three", "four",
                    "five", "six", "seven", "eight", "nine")
                .collect( groupingBy( String::length, counting() ) );
    }
}
```


Terminale Operation

- Summe, Minimum, Maximum, Durchschnitt

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

public class Ex {
    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);
    }
}
```

Terminale Operation

```
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
```

```
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
```

Terminale Operation

- reduzierende Operationen
- Produziert einzelnes Resultat aus allen Elementen

```
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 );
```

Beispiel

```
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));
        }
    }
}
```