

# Software Entwicklung 1

Annette Bieniusa

AG Softech  
FB Informatik  
TU Kaiserslautern

# Überblick für heute

## Funktionale Programmierung mit Java Lambdas

- Syntax von Lambda-Ausdrücken
- Streams
- **Beispiele**
- Behind the Scenes: Wie sind die Lambda-Ausdrücke in Java eingebettet?



TAKIPI

## Wiederholung: Funktionen höherer Ordnung im Lambda-Kalkül I

- Funktion, die eine (oder mehrere) Funktionen als Argument nimmt oder als Ergebnis liefert
- Parametrisierung über eine Berechnung möglich!
- **Beispiel:** Anwendung von Funktion auf alle Elemente einer Liste

```
map (\x -> x + 2) [1,2,3,4] = [3,4,5,6]
```

```
map (\y -> 3 * y) [1,2,3,4] = [3,6,9,12]
```

## Wiederholung: Funktionen höherer Ordnung im Lambda-Kalkül II

- **Beispiel:** Filtern von Elementen

```
filter (\x -> x < 10)      [1,32,7]   = [1,7]
filter (\x -> x % 2 == 0) [1,32,8]   = [32,8]
```

- **Beispiel:** Faltung von Listelementen, ausgehend von Initialwert, mittels einer Funktion zu einem Wert

```
foldl (\y x -> y + x) 0 [1,2,3]
= (((0 + 1) + 2) + 3)
= 6
```

# Lambda-Ausdrücke in Java

- Bis Java 8: Trennung von Daten und Programmlogik
- Mit Java: Einführung von Lambda-Ausdrücken  
( $\Rightarrow$  "Methoden ohne Namen")
- Erlauben die Parametrisierung von Berechnungen
- Sehr kompakte Syntax erlaubt elegantes Programmieren
- Viele Möglichkeiten zur Optimierung durch den Compiler und die Laufzeitumgebung (JVM)

# Syntax von Lambda-Ausdrücken I

`(x, y) -> x + y`

- Formale Parameter + Pfeil `->` + Funktionsrumpf
- Rückgabebetyp und Exceptions nicht spezifiziert, sondern vom Compiler inferiert
- Syntax sehr vielseitig

```
(int x)  ->  x + 1  //Parameter mit expliziter Typangabe
x       ->  x + 1  //Parameter ohne explizite Typangabe
()      ->  5      //leere Parameterliste
```

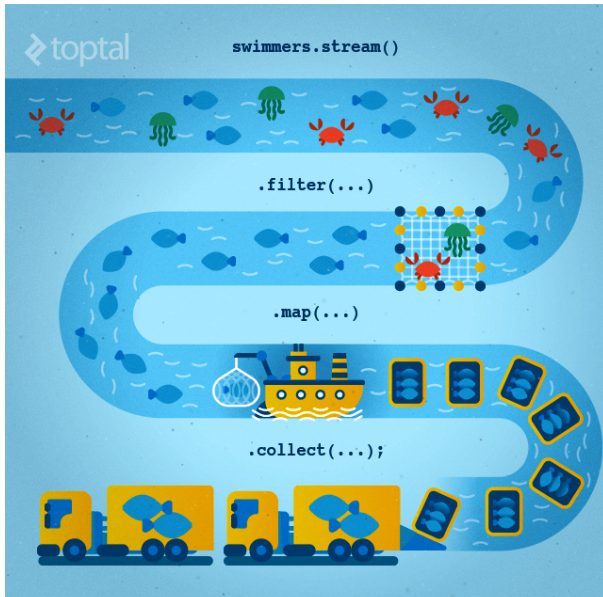
## Syntax von Lambda-Ausdrücken II

- Funktionsrumpf ist entweder ein Ausdruck oder ein Anweisungsblock mit abschließender `return`-Anweisung

```
(x, y) -> x * y
```

```
(x, y) -> { ... /* weitere Anweisungen */  
          return x * y; }
```

- Die Parameternamen in Lambda-Ausdrücken dürfen nicht bereits als Variablen in der umschließenden Methode definiert sein.





## Beispiele: Listen von Integern bearbeiten I

```
List<Integer> list = ....
```

```
// Sammelt alle Elemente der Liste um 1 inkrementiert  
// in einer neuen Liste  
// Variante mit Lambdas
```

```
List<Integer> incremented2 = list.stream()  
    .map(x -> x + 1)  
    .collect(Collectors.toList());
```

```
// Variante mit Iterator
```

```
List<Integer> incremented = new ArrayList<Integer>();  
for(int n : list) {  
    incremented.add(n+1);  
}
```

## Beispiele: Listen von Integern bearbeiten II

```
// Skaliert die Element in der Liste um den Faktor s
// und addiert dazu jeweils t;
// sammelt das Ergebnis in einer neuen Liste
final int s = 2;
final int t = 1;
List<Integer> scaled = list.stream()
    .map(x -> s * x + t)
    .collect(Collectors.toList());

// Filtert alle geraden Elemente aus der Liste
// und sammelt sie in einer neuen Liste
List<Integer> even = list.stream()
    .filter(x -> x % 2 == 0)
    .collect(Collectors.toList());
```

## Beispiele: Listen von Integern bearbeiten III

```
// Summiert die Elemente in der Liste
int summe_reduce = 0;
for(int n : list) {
    summe_reduce += n;
}
```

```
// Variante mit Lambdas
int summe_reduce2 = list.stream()
    .reduce(0, (a, b) -> a + b);
```

# Streams

# Streams

- Ein **Datenstrom** (engl. *Stream*) ist eine (potentiell unendliche) Folge von Daten gleicher Art.
- Wird von einer (oder mehreren) Quellen versorgt
- Auch verwendet bei Input-/Outputstreams bzw. Reader/Writer
- Hier: Deklarative Datenverarbeitung

## Beispiel: Streams

```
public static void listStream(List<String> list) {  
    Stream<String> s1 = list.stream();  
    Stream<Integer> s2 = s1.map(x -> Integer.valueOf(x));  
    Stream<Integer> s3 = s2.filter(x -> x > 0);  
    long i = s3.count();  
    System.out.println("i = " + i);  
}
```

- **Intermediären Operationen** nehmen Elemente aus einem Strom und erzeugen einen neuen Strom
  - Können hintereinander geschaltet werden
  - **Beispiele:** `map`, `filter`
- **Terminale Operationen** aggregieren ein Ergebnis
  - Konsumieren einen Strom
  - **Beispiel:** `count` (Reduktion auf einen Wert), Umwandlung in Collections, ...

# Erzeugen von Streams I

## ■ Quellen:

- Collections: `stream()` erzeugt einen (sequentiellen) Stream

```
List<String> list = ....;  
Stream<String> liststream = list.stream();  
  
Map<Integer, String> map = ...;  
Stream<Map.Entry<Integer, String>> entrystream =  
    map.entrySet().stream();  
Stream<Integer> keystream = map.keySet().stream();
```

## ■ Arrays

```
String[] ar = ....;  
Stream<String> arrstream = Arrays.stream(ar);
```

## Erzeugen von Streams II

### ■ Generatorfunktionen

```
Stream<Double> random =  
    Stream.generate(() -> Math.random());  
Stream<Double> seq    =  
    Stream.iterate(1, x -> x + 1);
```

### ■ I/O-Kanäle

```
try (Stream<String> filestream =  
    Files.lines(Paths.get("data.txt"))) {  
    .... // Verwendung von filestream  
}
```



## Operationen auf Streams: map

- Wendet eine Funktion auf alle Elemente im Stream an und liefert den Ergebnis-Stream

```
// wandelt alle Strings in Kleinbuchstaben um:  
Stream<String> words = ...  
words.map(s -> s.toLowerCase())
```

```
// Quadriert alle Zahlen im Stream:  
Stream<Integer> numbers = ...  
numbers.map(x -> x*x)
```

## Operationen auf Streams: map

- Filtert alle Elemente aus einem Stream heraus, die eine gegebene Bedingung erfüllen, und liefert einen Stream mit diesen Elementen

```
// Liefert alle positiven Zahlen aus dem Stream:  
numbers.filter(x -> x > 0)
```

```
// liefert die Strings, die das Wort "toll" enthalten  
words.filter(s -> s.contains("toll"))
```

## Operationen auf Streams: reduce

- Wendet *zweistellige Funktion*  $\oplus$  auf die Elemente des Streams an und reduziert diese so zu einem einzelnen Wert

$$x_1 \oplus x_2 \oplus x_3 \oplus x_4 \dots$$

- Auch bekannt im Lambda-Kalkül als **fold**
- Reihenfolge der Auswertung nicht definiert  
⇒ Funktion sollte assoziativ sein

$$(((x_1 \oplus x_2) \oplus x_3) \oplus x_4) = ((x_1 \oplus x_2) \oplus (x_3 \oplus x_4))$$

- *Variante 1*: Faltet zweistellige Funktion über die Elemente im Stream
  - Bei leerem Stream: leeres **Optional**
  - Sonst: **Optional** mit Wert

```
// Aufsummieren der Zahlen im Stream:
Optional<Integer> sum = numbers.reduce((x,y) -> x+y)
System.out.println(sum.orElse(0));
```

## Operationen auf Streams: reduce

- Variante 2: Faltungsfunktion mit zusätzlichem Startwert
  - Liefert immer ein direktes Ergebnis, kein `Optional`
  - Bei leerem Stream: Startwert
  - Startwert sollte Identität in Bezug auf Faltungsfunktion sein

```
// Aufsummieren der Zahlen im Stream:  
numbers.reduce(0, (x,y) -> x+y);  
// Aufmultiplizieren der Zahlen im Stream:  
numbers.reduce(1, (x,y) -> x*y);
```

## Quizz

Gegeben eine Liste von Integern `list` mit Einträgen 3, 17, 9, 1, 0 -3.  
Was liefern die folgende Ausdrücke?

```
list.stream()  
  .filter(x -> x < 10 && x > 0)  
  .reduce((x,y) -> {if (x > y)  
    return x  
    else return y;})
```

```
list.stream()  
  .map(x -> 2 * x)  
  .filter(x -> x % 2 == 1)  
  .reduce((x,y) -> x + y)
```

## Quizz: Auflösung

Gegeben eine Liste von Integern `list` mit Einträgen 3, 17, 9, 1, 0 -3.  
Was liefern die folgende Ausdrücke?

```
list.stream()  
  .filter(x -> x < 10 && x > 0) // [3,9,1]  
  .reduce((x,y) -> {if (x > y)  
    return x  
    else return y;}) // Optional<Integer> mit 9
```

```
list.stream()  
  .map(x -> 2 * x) // [6, 34, 18, 2, 0, -6]  
  .filter(x -> x % 2 == 1) // []  
  .reduce((x,y) -> x + y) // Optional<Integer>.empty()
```

## Operationen auf Streams: collect

- Baut das Ergebnis in einem (veränderbaren) Objekt auf

```
// Stream von Strings in einer ArrayList speichern:  
words.collect(  
    () -> new ArrayList<String>(),           // initial  
    (list, s) -> list.add(s),               // akkumuliert  
    (list1, list2) -> list1.addAll(list2) // kombiniert  
)
```

- Häufig in `Collector` zusammengefasst

```
// Stream von Strings in einer Liste speichern:  
List<String> l = words1.collect(Collectors.toList());  
// Stream von Strings in einer Menge speichern:  
Set<String> s = words2.collect(Collectors.toSet());  
// Elemente durch Komma getrennt als String:  
String str = words3.collect(Collectors.joining(", "));
```

## Operationen auf Streams: forEach

- Terminale Operation ohne Ergebnis
- Ruft eine Funktion ohne Ergebnis für jedes Element im Stream auf
- Achtung: Reihenfolge nicht spezifiziert!
- Alternative: `forEachOrdered()`

```
// Alle Strings im Stream ausgeben:  
words.forEach(w -> {  
    System.out.println(w);  
});
```



## Vergleichen und Sortieren mit Lambdas

- Wiederholung: Interface `Comparator<T>` mit Methode `int compare(T x, T y)`
- Verkürzte Schreibweise mit Lambdas

```
Comparator<String> nachLaenge = (x, y) -> {  
    if (x.length() > y.length()) {  
        return 1;  
    } else if (x.length() < y.length()) {  
        return -1;  
    } else {  
        return x.compareTo(y);  
    }  
};
```

## Sortieren von Listen mit Comparators

- `sort` ist statische Methode der Klasse `java.util.Collections`
- Sortiert eine Liste mit Hilfe eines `Comparator` aufsteigend

```
// Liste nach Laenge der Strings sortieren:  
List<String> list = Arrays.asList("aaaa", "b", "ccc",  
                                "dd", "e");  
  
Collections.sort(list, nachLaenge);  
  
System.out.println(list);  
// Ausgabe: [b, dd, ccc, aaaa]
```

## Sortieren von Listen mit Comparators

- `sorted` ist Sortiermethode für Streams
- Sortiert die Elemente aus einem Stream mit Hilfe eines `Comparator` aufsteigend
- Liefert sortierten Stream

```
List<String> list = Arrays.asList("aaa", "b", "ccc",  
    "dd", "e");
```

```
List<String> sorted = list.stream()  
    .sorted(nachLaenge)  
    .collect(Collectors.toList());
```

```
System.out.println(list);  
// Ausgabe: [aaa, b, ccc, dd, e]  
// list ist nicht veraendert!
```

```
System.out.println(sorted);  
// Ausgabe: [b, dd, ccc, aaa]
```

# Weiteres Vergleichsmöglichkeiten I

- `Comparator.naturalOrder()`: natürliche Ordnung für Subtypen von `Comparable`
- `Comparator.reverseOrder()`: invertierte Ordnung
- `Comparator.comparing`: Ordnung auf  $f(x)$  und  $f(y)$  für Funktion  $f$

```
// Strings nach ihrer Laenge vergleichen:  
Comparator<String> nachLaenge =  
    Comparator.comparing(s -> s.length());
```

## Weiteres Vergleichsmöglichkeiten II

- `Comparator.thenComparing()` erweitert eine Ordnung
- Bei gleichen Elementen wird eine weitere Ordnung verwendet

```
// Strings erst nach Laenge, dann nach natuerlicher  
Ordnung vergleichen:
```

```
Comparator<String> nachLaenge = Comparator  
    .comparing((String s) -> s.length())  
    .thenComparing(Comparator.naturalOrder());
```

## Weiteres Vergleichsmöglichkeiten III

```
// Orte erst nach ihrer Postleitzahl und dann nach ihrem
    Namen sortieren:
Comparator<Ort> ortVergleicher =
    Comparator
        .comparing((Ort o) -> o.getPostleitzahl())
        .thenComparing(o -> o.getName());

// Adressen bei gleichem Ort nach Strasse, dann nach
    Hausnummer ordnen:
Comparator<Adresse> addressVergleicher =
    Comparator
        .comparing((Adresse a) -> a.getOrt(), ortC)
        .thenComparing(a -> a.getStrasse())
        .thenComparing(a -> a.getHausnummer());
```

## Weiteres Vergleichsmöglichkeiten IV

```
Comparator<Ort> ortVergleicher =  
    Comparator  
        .comparing((Ort o) -> o.getPostleitzahl())  
        .thenComparing(o -> o.getName());
```

```
// Variante mit Methodenreferenzen:  
Comparator<Ort> ortVergleicher =  
    Comparator  
        .comparing(Ort::getPostleitzahl)  
        .thenComparing(Ort::getName);
```

## Auswertung von Streamoperationen und Seiteneffekte

- Kombination von Streams und Lambdas ermöglicht deklaratives Programmieren in Java
- Wie die Verarbeitung passiert, ist offengelassen

```
9 public static void listIteration(Collection<Integer> coll)
10 {
11     // Variante 1: Iteration mit Iterator
12     Iterator<Integer> it = coll.iterator();
13     while (it.hasNext()) {
14         int i = it.next();
15         System.out.print(i + " ");
16     }
17     // Variante 2: Explizites Iterieren
18     for (int i : coll) {
19         System.out.print(i + " ");
20     }
21     // Variante 3: Implizites Iterieren
22     coll.stream().forEach(i -> System.out.print(i + " "));
23 }
```



## Problem: Seiteneffekte

- Funktionen in der rein-funktionalen Programmierung frei von Seiteneffekten
- Kein Verändern von Attributen, globalen Variablen, Konsole, ...
- Ergebnis (d.h. der Effekt) der Funktionsauswertung hängt allein von den Parametern ab
- Gleiche Parameterwerten  $\Rightarrow$  gleiches Ergebnis
- Konzepte “Methode” und “Funktion” werden in Java nicht sauber getrennt
- **Ziel:** Seiteneffekte beschränken auf geordnete / sortierte Streams

```
Arrays.asList("Amsel", "Drossel", "Affe", "Kamel").stream()  
    .filter(s -> s.startsWith("A"))  
    .map(s-> s.toUpperCase())  
    .forEachOrdered(s -> System.out.println(s));
```

## Lazy-Auswertung von Operationen

- Auswertung erfolgt nur dann, wenn das Ergebnis zwingend erforderlich
- Erlaubt unendliche Datenströme!
- Braucht terminale Operation, um die Verarbeitung eines Streams zu erzwingen

```
List<Integer> seq = Stream
    .iterate(1, x -> x+1)
    .limit(5)    // Nur die ersten 5 Elemente aus Stream
    .collect(Collectors.toList());
```

## Frage: Was ist die Ausgabe?

```
List<String> list = Arrays.asList("Apfel", "Birne", "Kiwi");
String res = list.stream()
    .map(o -> {
        System.out.println("map " + o);
        return o.toLowerCase();
    })
    .filter(o -> {
        System.out.println("filter " + o);
        return o.startsWith("b");
    })
    .findFirst()
    .get();

System.out.println("res = " + res);
```

## Frage: Was ist die Ausgabe?

```
List<String> list = Arrays.asList("Apfel", "Birne", "Kiwi");
String res = list.stream()
    .map(o -> {
        System.out.println("map " + o);
        return o.toLowerCase();
    })
    .filter(o -> {
        System.out.println("filter " + o);
        return o.startsWith("b");
    })
    .findFirst()
    .get();

System.out.println("res = " + res);
```

```
map Apfel
filter apfel
map Birne
filter birne
res = birne
```

## Wiederverwendung von Streams

Streams können nicht “wiederverwendet” werden.

```
List<String> l = Arrays.asList("a", "b", "c");  
  
Stream<String> s1 = l.stream();  
Stream<String> s2 = s1.map(s -> "!");  
Stream<String> s3 = s1.map(s -> "?");  
// IllegalStateException: stream has already been operated  
// upon or closed
```

Nach der terminalen Operation ist der Stream auch nicht mehr verfügbar.

# Behind the Scenes: Wie funktionieren Lambda-Ausdrücke in Java? I

- Wie können Lambda-Ausdrücke in Java-Typsystem integriert werden?  
⇒ *Funktionale Interfaces*: Interface-Typen mit nur einer einzigen abstrakten Methode
- Passender Interface-Typ wird nicht vom Programmierer spezifiziert, sondern vom Compiler aus dem Kontext abgeleitet

Interface	Lambda-Ausdruck	Auswertungsmethode
<code>Function&lt;T,R&gt;</code>	<code>x -&gt; x+1;</code>	<code>apply(T x)</code>
<code>BiFunction&lt;T,U,R&gt;</code>	<code>(x,y) -&gt; x*y;</code>	<code>apply(T x, U y)</code>
<code>BinaryOperator&lt;T&gt;</code>	<code>(x,y) -&gt; x-y;</code>	<code>apply(T x, T y)</code>
<code>Predicate&lt;T&gt;</code>	<code>x -&gt; x&lt;0;</code>	<code>test(T x)</code>

# Behind the Scenes: Wie funktionieren Lambda-Ausdrücke in Java? II

```
1 import java.util.function.*;
2
3 public class Funktionen {
4
5     public static void functionReferences() {
6         int a = 3;
7         int b = 4;
8
9         Function<Integer, Integer> f = (x) -> x * x - 1;
10        BiFunction<Integer, Integer, Integer> g =
11            (x, y) -> x * x + 2 * x - 1;
12
13        System.out.println("f(" + a + ") = " + f.apply(a));
14        System.out.println("g(" + a + ", " + b + ") = " + g.
15        apply(a, b));
16    }
17 }
```

## Operationen auf Streams (Auszug)

- `Stream<T> filter(Predicate<? super T> p)`  
Beispiel: `stream.filter(x -> x>5)`
- `Stream<R> map(Function<? super T, ? super R> f)`  
Beispiel: `stream.map(x -> x * x)`
- `T reduce(T start, BinaryOperator<T> f)`  
Beispiel `stream.reduce(0, (x,y) -> x * y)`



# Zusammenfassung

- Lambda-Ausdrücke erlauben es knapp und prägnant in Java zu programmieren
- Programme verlieren ihren prozeduralen Charakter
- Ausblick: Parallelisierung von Berechnungen (Vertiefung in SE3)
  - Verteilung von Berechnungen in Multi-core Prozessoren
  - Berechnungen, die unabhängig voneinander sind, können auf verschiedenen Prozessoren **gleichzeitig** durchgeführt werden  
⇒ Reduziert gesamte Berechnungszeit
  - Idee: Parallele Streams mit automatischer Aufgabenverteilung
  - Problem: Ressourcenkonflikte wegen Nichtdeterminismus