Java 8

Lambdas und Streams

Inhalte

- 1. Teil: Lambda Ausdrücke
  - Theoretische Grundlagen
  - Lambdas in Action
- 2. Teil: Streams
  - Funktionsweise
  - Sequentiell vs Parallel
  - Praktische Beispiele
- Fazit

Teil 1 - Lambdas

Funktionale Programmierung

- Programme aus Funktionen
  - Rekursion
  - Funktionen höherer Ordnung
- Einfluss von anderen JVM Sprachen …
  - Scala
  - Clojure
  - Groovy

Funktionen als Parameter?
… das gab es doch schon

```
button.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    doSomething();
  }
});
```

➔ **anonyme Klassen**

Funktionen als Prameter?
… das gabs doch schon

```
button.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    doSomething();
  }
});
```

Nachteile:

- Boilerplate Code (unübersichtlich)
- Instanziierung eines Objektes mittels Konstruktor
- Separate Bytecode Klasse (SampleApp$1)

➔ je Anwendungsfall viel Overhead

mit Java 8 Lambda …

```
button.addActionListener((e) -> {
        doSomething();
    }
);
```

mit Java 8 Lambdas …

```
button.addActionListener(e -> doSomething());
```

Ist ein Lambda-Ausdruck nur eine Kurzform einer anonymen Klasse?

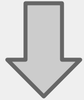
Fachlich ja, technisch nein

Es gibt unterschiedliche Strategien für Transformation in Bytecode:

- innere Klassen
- Methodhandles
- Dynanmic Proxies
- ...

Bsp.: Lambda in Methode überführen

```
list.removeIf(p -> p.age < minAge)
```



```
static boolean lambda$1(int minAge, Person p) {
    return p.age < minAge;
}
```

Strategie hängt vom Kontext ab

invokedynamic

- Bytecode Befehl ab Java 7 - Eingeführt für Skriptsprachen auf der JVM
- Ermöglicht Trennung zw. Evaluierung des Lambdas und Bytecode Repräsentation

➔ Realisierung mit wenig Overhead und hoher Flexibilität

Siehe: Brian Goetz - "Lambdas in Java: A peek under the hood"

Konsequenz: Kein Funktionstyp

- Lambdas ist immer Typ eines Interfaces
  - feste Signatur
- es gibt kein Funktionstyp mit variabler Signatur

Bedingte Alternativen:

- viele fertige Funktion-Interfaces
- Generics

➔ Kein vollständiges funktionales Sprachmittel

Wie kann man ein Lambda-Ausdruck schreiben?

Lambda-Syntax

```
(args) -> { multiline body;}
```

Eine Befehlszeile	`(args) -> befehl`
Ein Argument	`x -> befehl`
Kein Argument	`() -> befehl`
Methodenreferenz	`System.out::println`

Wo kann man Lambdas verwenden?

Überall dort,
wo eine Instanz eines funktionalen Interfaces benötigt wird

Funktionale Interfaces

- Ein beliebiges Java Interface mit **einer abstrakten** Methode
- Keine Klasse oder abstrakte Klasse
- Kann mit der Annotation “@FunctionalInterface” markiert werden
- Einige ältere Java Interfaces sind bereits funktionale Interfaces:
    - Comparable
    - Runnable
    - ActionListener

java.util.function.*

42 vordefinierte funktionale Interfaces:

- Function<T,R>   T => R
- Consumer<T>   T => void
- Predicate<T>   T => boolean
- IntBinaryOperator   int,int => int
- ...

Erweiterung der Klassenbibliothek

- Erweiterungen für die Verwendung von Lambdas:
  - `Arrays.setAll(array, IntFunction)`
- Mittels Default-Methoden wurden nützliche Methoden in ältere Schnittstellen eingeführt:
  - `Collection`
    - `removeIf(Predicate)`
  - `Iterable`
    - forEach(Consumer)
  - Map
    - forEach(BiConsumer)

Instanziierung von Objekten

```
public interface Test {
      boolean test(int a);
}

Test evenTester = n -> n % 2 == 0;

assert evenTester(2);
assert !evenTester(3);
assert evenTester instanceof Test;
```

In Action - Map

```
Map<Person, Integer> booking = ...;

booking.forEach((p, room) ->
    System.out.println(
        String.format("room %d: %s", room, p.getName())
));
```

In Action - Map

```
Map<Person, Integer> booking = ...;
Person bob = ...;

booking.computeIfPresent(bob, (p, room) -> room + 1);
```

In Action - Arrays

```
Arrays.sort(words, (s1, s2) ->
    s1.trim().compareTo( s2.trim()));
```

In Action - Arrays

```
int[] a = new int[] {1,2,3};
int[] b = new int[] {4,5,6};
int[] c = new int[3];

Arrays.parallelSetAll(c, n -> a[n] * b[n]);
```

Eigene Klassen mit Lambda-Support?

- Alle Schnittstellen möglichst als FunctionalInterfaces konzipieren

Beschränkungen und Grenzen

- ConcurrentModification

```
List<Integer> list = ...;
list.forEach(n -> list.add(n)); //crash at runtime
```

- Local variables must be final

```
int var = 12;
list.forEach(n -> var += n); //crash at compile time
```

Beschränkungen und Grenzen

- Ausbruch aus Lambda?

```
List<Integer> list = ...;
list.forEach(n -> {
    if (n < 100) {
        result.add(n);
    } else {
        return;
    }});
```

Collections und Streams

Lambda-Ausdrücke

Methodenreferenzen

Funktionale Interfaces

java.util.function

Verbesserte Typinferenz

Default-Methoden

Teil 2 - Streams

Streams aus theoretischer Sicht

- endliche oder potentiell unendliche Folge von Daten
- Gierige Evaluation
- Verzögerte Evaluation
    - Evaluation der Daten wird solange aufgeschoben, bis diese wirklich benötigt werden
- in der Praxis wird bei einem unendlicher Stream eine oberes Limit verwendet

Streams in Java 8

- Fluent API für Operations Pipeline
- Lazy Evaluation
- Keine Datenstruktur
- non-interfering (Stream darf nicht manipuliert werden)

Streams in Java 8

- Erleichtert Parallelisierung auf hohem Abstraktionsniveau
- Zugriff via `stream()` bzw. `parallelStream()` an Collections, Datenströmen

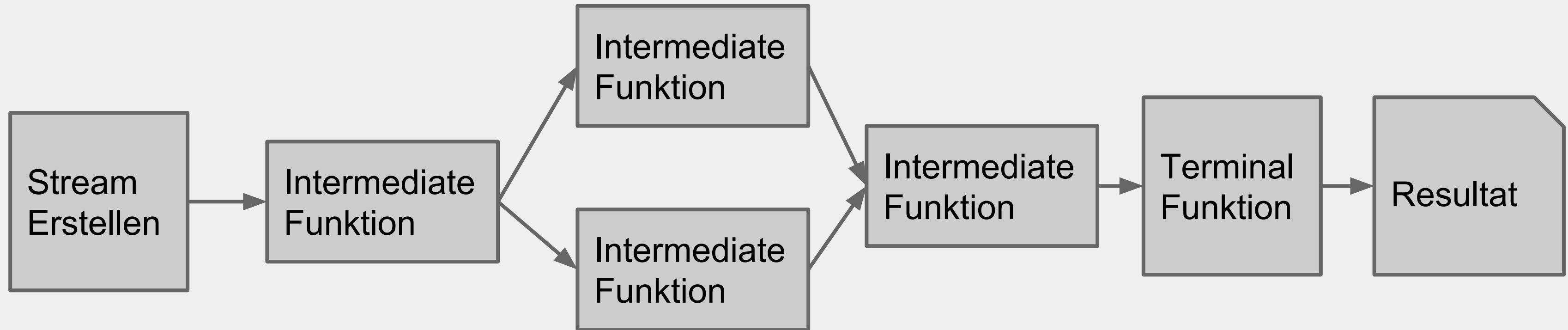
Stream Operationen

- Intermediate Operations:
  - `filter(Predicate)`
  - `map(Function)`
  - `...`
- Stateful Intermediate Operations:
  - `distinct()`
  - `sorted()`
  - `skip(long)`
  - `...`

Stream Operationen

- Terminal Operations:
  - `count()`
  - `collect(Collector)`
  - `forEach(Consumer)`
  - `findAny()`
  - `...`
- Verarbeitung wechseln:
  - `sequential()`
  - `parallel()`

Funktionsweise



Lazy Evaluation

```
List<Integer> source = new ArrayList<>(
      Arrays.asList(new Integer[]{1, 2, 3, 4}));

Stream<Integer> stream = source.stream();
stream = stream.filter(n -> n != 1);
source.add(1);
stream.forEach(n -> System.out.println(n));
```

Lazy Evaluation

```
List<Integer> source = new ArrayList<>(
        Arrays.asList(new Integer[]{1, 2, 3, 4}));

Stream<Integer> stream = source.stream();
stream = stream.filter(n -> n != 1);
source.add(1);
stream.forEach(n -> System.out.println(n));
```

> 2 3 4

Lazy Evaluation

```
List<Integer> source = new ArrayList<>(
      Arrays.asList(new Integer[]{1, 2, 3, 4}));

Stream<Integer> stream = source.stream();
stream = stream.filter(n -> n != 1);
source.add(5);
stream.forEach(n -> System.out.println(n));
```

Lazy Evaluation

```
List<Integer> source = new ArrayList<>(
        Arrays.asList(new Integer[]{1, 2, 3, 4}));

Stream<Integer> stream = source.stream();
stream = stream.filter(n -> n != 1);
source.add(5);
stream.forEach(n -> System.out.println(n));
```

> 2 3 4 5

Parallel vs Sequentiell Prozesskette

- Parallel mittels Fork/Join-Framework java.util.Spliterator
- nur bei großen Datenmengen effizient (> 100k)
- kein Magic-Schalter zw. parallel und seq.

→ Testen und je Anwendungsfall entscheiden

Tipps für Laufzeitoptimierung

- Reihenfolge beachten:

```
Arrays.stream(new int[10]).
      map(n -> {
         Thread.sleep(1000);
         return n;
      }).filter(n -> false).
      count();
```

Tipps für Laufzeitoptimierung

- Reihenfolge beachten:

```
Arrays.stream(new int[10]).
      map(n -> {
         Thread.sleep(1000);
         return n;
      }).filter(n -> false).
      count();
```

```
> 10sek
```

Tipps für Laufzeitoptimierung

- Reihenfolge beachten:

```
Arrays.stream(new int[10]).
        filter(n -> false).
        map(n -> {
           Thread.sleep(1000);
           return n;
        }).count();
```

Tipps für Laufzeitoptimierung

- Reihenfolge beachten:

```
Arrays.stream(new int[10]).
        filter(n -> false).
        map(n -> {
            Thread.sleep(1000);
            return n;
        }).count();
```

> 0sek

Tipps für Laufzeitoptimierung

- Bei parallelen Streams:
  - Beim Collector-Schritt spezielle Methoden verwenden

    z. B. `Collectors.groupingByConcurrent`
  - Synchronisation eigener Datenquellen kann zu Flaschenhals werden

Tipps für Laufzeitoptimierung

- Verwendung effizienterer Datenstrukturen
  - Problemreduktion auf primitive Datentypen
  - Guava, Trove
- Vermeidung von Masseninstanziierung von Objekten
- Bei großen Datenmengen kann angepasster eigener Fork/Join-Algorithmus effizienter sein.

Nützliches: Optional

- Null-Object-Wrapper
- Vermeidung von Null-Checks
- wird bei vielen Terminal-Funktionen verwendet
- Praktische Handhabung:
  - `orElse(other)`
  - `map(Function)`

Nützliches: summaryStatistics

- Terminal-Funktion
- Nur bei numerischen Streams
- Sammelt statistische Daten:
  - count
  - average
  - min
  - max
  - sum

Anwendungsbeispiele

- Erstellen von Streams:

```
Streams.of(“A”, “B”, ...);

IntStream.of(1,3,...);

Arrays.stream(array);
```

Anwendungsbeispiele

- Unendlicher Stream:

```
IntStream.iterate(0, i -> i + 2).count();

//Begrenzen:
IntStream.iterate(0, i -> i + 2).
          limit(100).
          count();
```

Anwendungsbeispiele

- Filter-Mapping-Auswertung:

```
List<Person> list = ...;
double averageAge = list.stream().
        filter(p -> p.getName().startsWith("A")).
        filter(p -> p.getAge() >= 18).
        mapToInt(p -> p.getAge()).
        average().
        getAsDouble();
```

Fazit

Fazit

- gelungene Spracherweiterung
- mächtiges Werkzeug
- Deskriptives Vorgehen - Hohe Abstraktion
- kleinere Probleme bei primitiven Typen
    - Stream<Integer> != IntStream
    - Value-Objects in Java 9 schaffen da vielleicht Abhilfe

Fazit

- Vorteile und Nachteile für die Softwareentwicklung muss die weitere Erfahrung damit zeigen
- noch wenig Best Practice

Auch Lambdas und Streams können Code Smells werden:

```java
return container.getGEOKnoten().stream().map(knoten -> {

    ENUMGEOArt art = findGEOKanteBy(container, knoten).
            map(kante -> kante.getGEOKanteAllg().getGEOArt().getWert()).
            orElse(ENUMGEOArt.GLEIS);

    GEONode.GEONodeBuilder builder = GEONode.builder(ENUMGEOArt.GLEIS.equals(art) ? GEONodeType.BHPG : GEONodeType.BHPS);

    builder.id(UUID.fromString(knoten.getIdentitaet().getWert()));
    builder.source(EntitySource.GND);

    if (knoten.getGEOPAD() != null) {
        builder.pad(knoten.getGEOPAD().getWert());
    }

    container.getGEOPunkt().stream().
            filter(punkt -> punkt.getIDGEOKnoten().getWert().equals(knoten.getIdentitaet().getWert())).
            forEach(punkt -> {
                TCGEOKoordinatenSystemLSys lsys = punkt.getGEOPunktAllg().getGEOKoordinatenSystemLSys();
                CoordinateSystem coordSys = lsys == null ? CoordinateSystem.UNKNOWN : CoordinateSystem.create(lsys.getWert());

                builder.point(
                        Coordinate.builder(coordSys).
                        id(UUID.fromString(punkt.getIdentitaet().getWert())).
                        coordinate(punkt.getGEOPunktAllg().getGKX().getWert().doubleValue(), punkt.getGEOPunktAllg().getGKY().getWert().doubleValue()).
                        build()
                );
            });

    return builder.build();

}).filter(Objects::nonNull).collect(Collectors.toList());
```

Danke

Quellen

- Brian Goetz - "Lambdas in Java: A peek under the hood"
- http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html
- http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
- http://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html
- http://www.tutego.de/blog/javainsel/2012/12/einfhrung-in-java-8-lambda-ausdrcke-code-sind-daten/
- http://funktionale-programmierung.de/2013/09/19/java8.html
- Bildnachweis: http://blog.stackhunter.com/wp-content/uploads/2014/05/get-started-with-lambda-expressions-in-java-8.jpg