Java 8
Lambdas und Streams

Dr. Jan Schäfer

Big Techday 7

23. Mai 2014, München

TNG TECHNOLOGY CONSULTING

Warum sollte mich Java 8 interessieren?

- Java ist eine der meistverwendeten Programmiersprachen, insbesondere bei Enterprise-Software
- Java 8 ist das größte Java-Release seit mindestens 10 Jahren (seit Java 5)

Neue Features in Java 8

Spracherweiterungen

- Lambda-Literale
- Functional Interfaces
- Methodenreferenzen
- Defaultmethoden in Interfaces (Virtual Extension Methods)
- Statische Methoden in Interfaces
- Verbesserte Typinferenz
- Annotationserweiterungen

Neue Features in Java 8 (Forts.)

Bibliothekserweiterungen

- Erweiterungen für Lambdas
- Streams
- Neue Zeit- und Datums-API
- Neue Concurrency-Klassen
- ...

Sonstiges

- Neue Javascript Engine (Nashorn)
- ...

Focus des Talks

Lambdas und Streams

Lambdas - Warum?

- Funktionale, zustandslose Programmierung
- Asynchrone, event-basierte Programmierung
- Parallele Programmierung
- Higher-Order Programmierung

λ-Literale

```
new Function<String,String>() {
    public String apply(String s) {
        return "Hello" + s;
    }
}
```

```
(String s) -> { return "Hello" + s; }
```

```
(String s) -> "Hello" + s
```

```
(s) -> "Hello" + s
```

```
s -> "Hello" + s
```

Target Typing

```
x -> x + x // Typ??
```

- Der Typ eines Lambda-Ausdrucks wird anhand des Ziel-Typen ermittelt

```
IntUnaryOperator fun = x -> x + x;
```

```
interface IntUnaryOperator {
   int apply(int i); // (int) => int
}
```

```
Object o = x -> x + x;
```

ERROR: Target type of this expression must be a functional interface.

Functional Interfaces

- Interfaces mit genau einer abstrakten Methode

```
@FunctionalInterface
public interface MyFunction {
    String fooBar(String s);
}
```

```
MyFunction fun = x -> x + x; // OK
```

Functional Interfaces

Existierende Functional Interfaces in Java < 8

```
java.lang.Runnable { void run(); }

java.util.Comparator<T> { int compare(T o1, T o2);}

java.io.FileFilter { boolean accept(File file); }

java.awt.event.ActionListener {
    void actionPerformed(ActionEvent e);
}

...
```

Functional Interfaces

java.util.function

Interface	Typ
`Function<S,T>`	`(S) => T`
`UnaryOperator<S>`	`(S) => S`
`Consumer<S>`	`(S) => void`
`Predicate<S>`	`(S) => boolean`
`Supplier<T>`	`() => T`
`BiFunction<R,S,T>`	`(R,S) => T`
`BinaryOperator<S>`	`(S,S) => S`

Functional Interfaces (Forts.)

Interface	Typ
`IntFunction<S>`	`(int) => S`
`ToIntFunction<S>`	`(S) => int`
`ToIntBiFunction<S,T>`	`(S,T) => int`
`IntUnaryOperator`	`(int) => int`
`IntConsumer`	`(int) => void`
`IntPredicate`	`(int) => boolean`
`IntSupplier`	`() => int`
`IntToLongFunction`	`(int) => long`
`IntToDoubleFunction`	`(int) => double`

Long..., Double...

Variable Capturing

```
int foo = 42;
return x -> x + foo;
```

- Variablen aus dem Kontext können benutzt werden wenn sie "Effectively Final" sind

Kein eigener Scope

```
int x = 42;
return x -> x;
```

ERROR: Lambda expression's parameter x cannot redeclare another local variable defined in an enclosing scope.

This und Super

```
class Foo {
  int x;
  IntSupplier supplyX() {
     return () -> this.x;
  }
}
```

this und super referenzieren den äußeren Kontext

Anwendungsbeispiele (1/3)

Event-Basierte Programmierung

Java < 8

```
btn.addActionListener( new ActionListener() {
  public void actionPerformed(ActionEvent event) {
    label.setText("Hallo Java 7");
  }
}
```

Java 8

```
btn.addActionListener(
   (event) -> label.setText("Hallo Java 8") );
```

Anwendungsbeispiele (2/3)

Thread-Erzeugung

Java < 8

```
new Thread(new Runnable() {
    public void run() {
        System.out.println("Thread started");
    }
}).start();
```

Java 8

```
new Thread(() ->
    System.out.println("Thread started")).start();
```

Anwendungsbeispiele (3/3)

Resource Handling (Java < 8)

```
Output someMethod1(Input input) {
  try (Connection conn = ds.getConnection()) {
    return doSomething1(input, conn);
  } catch (RuntimeException e) {
    log(e);
    throw e;
  }
}

Output someMethod2(Input input) {
  try (Connection conn = ds.getConnection()) {
    return doSomething2(input, conn);
  } catch (RuntimeException e) {
    log(e);
    throw e;
  }
}
```

Anwendungsbeispiele (3/3)

Resource Handling (Java 8)

```
<T> T withConn( Function<Connection, T> fun ) {
  try (Connection conn = ds.getConnection()) {
    return fun.apply(conn);
  } catch (RuntimeException e) {
    log(e);
    throw e;
  }
}

Output someMethod1(Input input) {
  return withConn( conn -> doSomething1(input, conn) );
}

Output someMethod2(Input input) {
  return withConn( conn -> doSomething2(input, conn) );
}
```

Methodenreferenzen

Statische Methoden

```
IntUnaryOperator abs = Math::abs;
int absInt = abs.applyAsInt(-5);
```

Konstruktoren

```
Supplier<List<String>> newList = ArrayList::new;
List<String> list = newList.get();
```

Methodenreferenzen (Forts.)

Instanzmethoden mit Objekt

```
String string = "foo";
Supplier<String> fun = string::toUpperCase;
String upperCase = fun.get();
```

Instanzmethoden mit Typ

```
Function<String,String> fun = String::toUpperCase;
String upperCase = fun.apply("foo");
```

Externe vs. Interne Iteration

Externe Iteration (Wie + Was)

```
for (String s : list) {
   System.out.println(s);
}
```

Interne Iteration (Was)

```
list.forEach( s -> {
   System.out.println(s);
});
```

Iterable.forEach

Default-Implementierung

```
public interface Iterable<T> {
  ...
  default void forEach(Consumer<? super T> action) {
    for (T t : this) {
      action.accept(t);
    }
  }
}
```

Streams

Interne Iteration auf Steroiden

- Schnittstelle zum Verarbeiten von gleichförmigen Datenströmen
- Kein eigener Datenspeicher (benötigt unterliegende Datenquelle)
- Funktional (unterliegende Daten werden nicht verändert)
- Können unendlich sein
- Werden genau einmal konsumiert (danach nicht mehr verwendbar)
- Lazy (Auswertung erst dann wenn wirklich angefordert)
- Parallelisierbar

Beispiel (Java 7)

Gib eine sortierte Liste aller Dateien im Root-Verzeichnis aus, die mit "s" anfangen.

Java 7

```
List<String> strings = new ArrayList<>();
for (File file : new File("/").listFiles()) {
    String s = file.toString();
    if (s.startsWith("/s")) {
       strings.add(s);
    }
}
Collections.sort(strings);
for (String s : strings) {
    System.out.println(s);
}
```

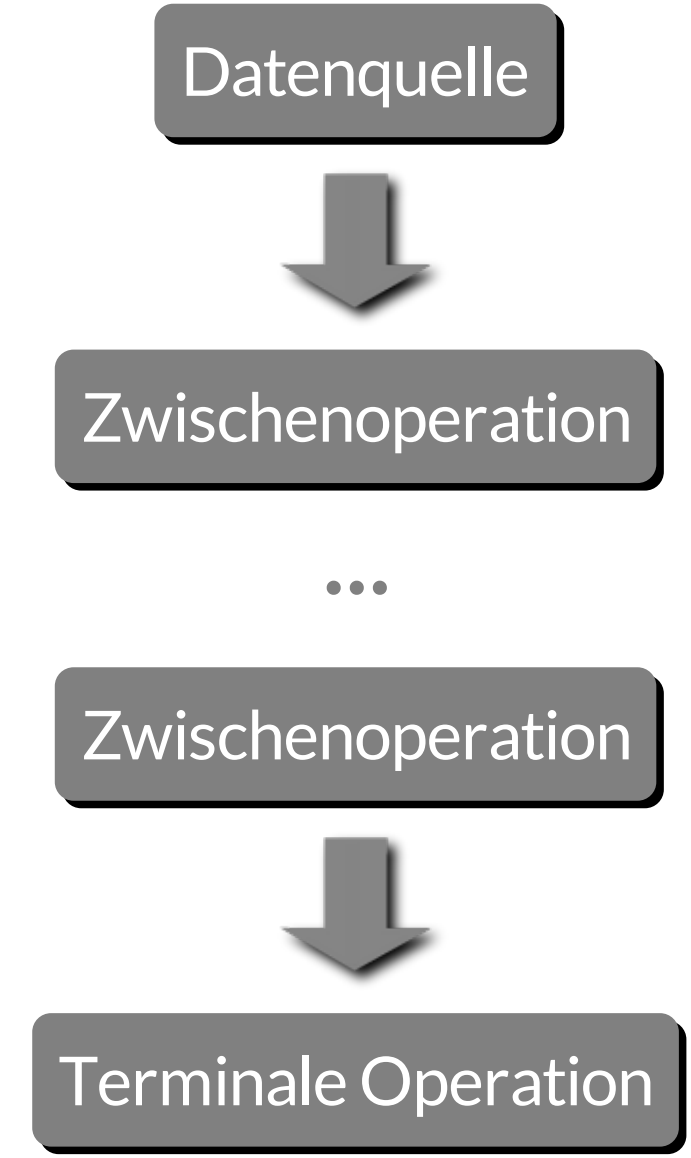
Beispiel (Java 8)

Gib eine sortierte Liste aller Dateien im Root-Verzeichnis aus, die mit "s" anfangen.

Java 8

```
Files.list(Paths.get("/"))
    .map(Path::toString)
    .filter(s -> s.startsWith("/s"))
    .sorted()
    .forEach(System.out::println);
```

Stream Pipeline



Beispiel

```
// Datenquelle
Files.list(Paths.get("/"))

    // Zwischenoperation
    .map(Path::toString)

    // Zwischenoperation
    .filter(s -> s.startsWith("/s"))

    // Zwischenoperation
    .sorted()

    // Terminale Operation
    .forEach(System.out::println);
```

Datenquellen

- Alle Collections durch die neue stream() Methode
- Stream.iterate(), Stream.generate(), IntStream.range(0,10)
- new Random().longs()
- ...

Zwischenoperationen

- Geben wieder einen Stream zurück
- Sind Lazy (Auswertung erst wenn angefordert)
- Zustandslos: z.B. map, filter
- Zustandsbehaftet: z.B. sorted, distinct

Terminale Operationen

- Triggern die Auswertung des Streams
- Konsumieren den Stream
- Zustandslos: findFirst, reduce, forEach
- Zustandsbehaftet: collect

Ohne terminale Operation **KEINE** Auswertung!

Beispiel: map ohne terminale Operation

```
Stream.of(1, 2, 3, 4)
    .map((x) -> {
       System.out.println(x);
       return x;
    });
```

Ausgabe:

```

```

Beispiel: map und forEach

```
Stream.of(1, 2, 3, 4)
    .map((x) -> {
       System.out.println("map: " + x);
       return x;
    })
    .forEach((x) -> { // terminale Operation
       System.out.println("forEach: " + x);
    });
```

Ausgabe:

```
map: 1
forEach: 1
map: 2
forEach: 2
map: 3
forEach: 3
map: 4
forEach: 4
```

forEach

```
Stream.of(1, 2, 3, 4)
   .forEach(System.out::println); // terminal
```

Ausgabe:

```
1
2
3
4
```

reduce

Zustandslose Reduktion

```
int sum = Stream.of(1, 2, 3, 4)
    .reduce((x,y) -> x+y)
    .get();
System.out.println(sum);
```

Ausgabe:

```
10
```

collect

Reduktion mit veränderlichem Zustand

```
List<Integer> result = Stream.of(1, 2, 3, 4)
   .collect(Collectors.toList());
```

```
Map<Boolean, List<Integer>> oddsEven =
   Stream.of(1, 2, 3, 4)
       .collect(Collectors.groupingBy(
           i -> i % 2 == 0));
```

Ergebnis:

```
{false=[1, 3], true=[2, 4]}
```

Unendliche Streams

```
new Random().ints()
  .forEach(System.out::println);
```

```
Stream.iterate(0, i -> i + 2)
  .forEach(System.out::println);
```

limit(n)

```
Stream.iterate(0, i -> i + 2)
  .limit(10) // Achtung bei Parallelverarbeitung!
  .forEach(System.out::println);
```

Parallele Verarbeitung

```
Arrays.asList(1,2,3,4,5)
   .parallelStream()
   .forEach(System.out::println);
```

Ausgabe:

```
3
1
4
5
2
```

Parallele Verarbeitung

Einschränkungen

- Ein gemeinsamer Thread-Pool für **ALLE** Streams!
- Anzahl der Threads nicht konfigurierbar (-> Anzahl Prozessoren)
- Operationen sollten zustandslos sein und keine Locks halten

Fazit

- Das Warten hat sich gelohnt!
- Lambdas sind gut gelungen
- Sehr viele Anwendungsmöglichkeiten
- Streams können sehr viele for und while-Schleifen ersetzen
- Extrem einfache (eingeschränkte) parallele Verarbeitung

Aber

- Das Mindset des Programmierers muss sich ändern!

Dank An

Java 8 Techday Series

- Eric Weikl, Peter Gafert (Überblick)
- Eckhard Maass, Manfred Hanke (Lambdas)
- Achim Herwig, René Scheibe (Streams)
- Michael Echerer und Andreas Schmid (Date + Time API)

Fragen?

jan.schaefer@tngtech.com

TNG TECHNOLOGY CONSULTING

Vertiefende Lesetipps

- Oracle Java 8 Tutorial: **http://docs.oracle.com/javase/tutorial/**
- Tutorial von Angelika Langer: **http://www.angelikalanger.com/Lambdas/Lambdas.html**
- Buch: Functional Programming in Java. Venkat Subramaniam. The Pragmatic Programmers, 2014
- Buch: Java 8 Lambdas. Richard Warburton. O'Reilly Media, 2014