zühlke
empowering ideas

Neuerungen in Java 8

Wichtige neue Features im Überblick

Michael Inden

Speaker – Kurzlebenslauf

zühlke
empowering ideas

- Michael Inden, Jahrgang 1971
- Diplom-Informatiker, C.v.O. Uni Oldenburg
- ~9 Jahre bei Heidelberger Druckmaschinen AG in Kiel
- ~7 Jahre bei IVU Traffic Technologies AG in Aachen
- Seit 2013 bei Zühlke Engineering AG in Zürich
  (We are hiring …)
- Autor und Gutachter beim dpunkt.verlag

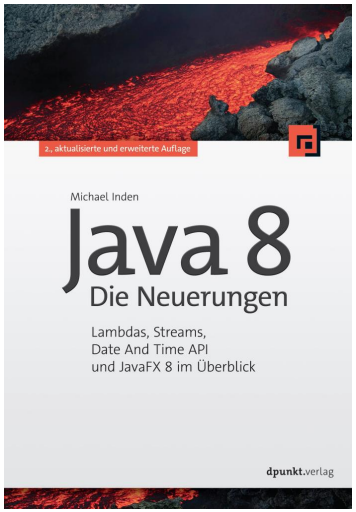


Table of Contents

zühlke
empowering ideas

zühlke
empowering ideas

Part 1: Lambdas

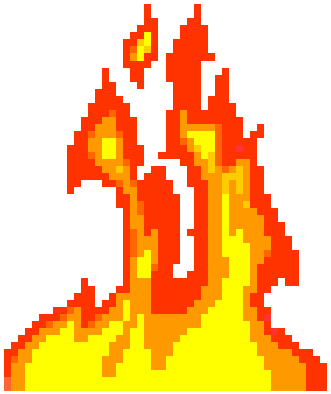
Motivation, Syntax & SAM

Default-Methoden und Methodenreferenzen

Michael Inden

Warum Lambdas?

zühlke
empowering ideas



Lambdas als ein neues und heiß ersehntes Sprachkonstrukt

- Lösungen auf sehr elegante Art und Weise formulieren
- andere Denkweise und neuer Programmierstil (funktional)
- Hilfe für Parallelverarbeitung und Ausnutzung von Multicores

Syntax von Lambdas

zühlke
empowering ideas

Lambda: eine spezielle Art von Methode bzw. ein Stück Code mit einfacher Syntax:

```
Parameter-Liste -> Ausdruck oder Anweisungen
```

(String name) --> name.length()

aber …

- ohne Namen (ad-hoc und anonym)
- ohne Angabe eines Rückgabetyps (wird vom Compiler ermittelt)
- ohne Deklaration von Exceptions (wird vom Compiler ermittelt)

Beispiele für Lambdas

zühlke
empowering ideas

```
(int x)    ->  { return x + 1; }                  // Typed Param, Statement
(int x)    ->   x + 1                             // Typed Param, Expression
(x,y)      ->  { x = x / 2; return x * y; }       // Untyped Param, Multi Statements
it         ->  it.startsWith("M")
()         ->  System.out.println("no param")     // No Param, No Return
```

Grundlagen zu Lambdas

zühlke
empowering ideas

Lambdas als Implementierung eines SAM:

```
new SAMTypeAnonymousClass()
{
  public void samMethod(METHOD-PARAMETERS)
  {
     METHOD-BODY
  }
}
```

=>

```
(METHOD-PARAMETERS) -> { METHOD-BODY }
```

Das geht, wenn Lambda die abstrakte Methode des SAM “erfüllen” kann, d.h. Parameter stimmen überein und der Rückgabetyp ist kompatibel.

Grundlagen zu Lambdas

zühlke
empowering ideas

Beispiele

```
Runnable runner = () -> { System.out.println("Hello Lambda"); };

Predicate<String> isLongWord = (final String word) -> { return word.length() > 15; };

Comparator<String> byLength = (str1, str2) -> { Integer.compare(str1.length(), str2.length()); };
```

Beispiel: Sortierung nach Länge und komma-separierte Aufbereitung

zühlke
empowering ideas

Mit JDK 7 erfolgte das in etwa so:

```
List<String> names = Arrays.asList("Andy", "Michael", "Max", "Stefan");

Collections.sort(names, new Comparator<String>() {
        @Override
       public int compare(String o1, String o2) {
          return Integer.compare(str1.length(), str2.length()));
        }
});

Iterator<String> it = names.iterator();
while (it.hasNext()) {
   System.out.print(it.next().length() + ", ");
}

// => 3, 4, 6, 7,
```

Lambdas im Einsatz: Sortierung und komma-separierte Aufbereitung

zühlke
empowering ideas

Mit JDK 8 und Lambdas schreibt man das kürzer wie folgt:

```
List<String> names = Arrays.asList("Max", "Andy", "Michael", "Stefan");

names.sort( (str1, str2) -> Integer.compare(str1.length(), str2.length));
names.forEach( it -> System.out.print(it.length() + ", ") );
// => 3, 4, 6, 7,
```

- Bei gleicher Ausgabe 12 : 3 Zeilen, Verhältnis 4:1 (alt:neu)
- Aber Moment …

Lambdas im Einsatz: Sortierung und komma-separierte Aufbereitung

zühlke
empowering ideas

`sort()` und `forEach()` … auf `List`? Wo kommen diese denn her?

Gibt es etwa neue Methoden im Interface `List`? JA!

Sind etwa alle alten Implementierungen nun nicht mehr kompatibel?

Braucht man vollständig neue spezielle Versionen etwa von Spring, Hibernate o.ä für Java 8?

Neuheit: Defaultmethoden

zühlke
empowering ideas

NEIN! Wieso nicht? Interfaces können nun Defaultmethoden enthalten

```
public interface List<E> extends Collection<E> {
    ...
    default void sort(Comparator<? super E> c) {
            Collections.sort(this, c);
    }
}

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

Neuheit: Methodenreferenzen

zühlke
empowering ideas

Methodenreferenz verweist auf …

- Methoden: a) Instanz-Methoden System.out::println, Person::getName, …
  String::compareTo => public int compareTo(String anotherString)
  b) statische Methoden: System::currentTimeMillis

- Konstruktoren: ArrayList::new, Person[]::new

Methodenreferenz kann anstelle eines Lambda-Ausdrucks genutzt werden

```
List<String> names = Arrays.asList("Max", "Andy", "Michael", "Stefan");

names.forEach( it -> System.out.println(it) );      // Lambda
names.forEach( System.out::println );               // Methodenreferenz
```

Neuheit: Methodenreferenzen

zühlke
empowering ideas

Bessere Lesbarkeit – Lambda (teilweise) durch Methodenreferenz ersetzbar

```
List<String> names = Arrays.asList("Max", "Andy", "Michael", "Stefan");
names.sort(String::compareTo);  // Instanz-Methode aus dem JDK
```

```
// VORHER: names.sort( (str1, str2) -> Integer.compare(str1.length(), str2.length));
names.sort(LambdaReturnExample::stringLengthCompare);
```

```
// Methodenreferenz nicht nutzbar (da in Lambda weitere Funktionalität aufgerufen wird)
names.forEach( it -> System.out.print(it.length() + ", ") );
```

ABER ….

zühlke
empowering ideas

Part 2: Bulk Operations on Collections

- Externe und interne Iteration
- Predicate<T>, UnaryOperator<T>

Michael Inden

Externe Iteration vs interne Iteration

zühlke
empowering ideas

Extern mit Iterator

```
Iterator<String> it = names.iterator();

while  (it.hasNext()) {
     String value = it.next();

     System.out.println(value);

}
```

Intern mit forEach

```
names.forEach(System.out::println);
```

Prädikate und Bulk Operationen

zühlke
empowering ideas

- `Predicate<T>` -- **Bedingungen formulieren**

```
Predicate<String> isEmpty = String::isEmpty;

Predicate<String> isShortWord = word -> word.length() <= 3;
Predicate<String> notIsShortWord = isShortWord.negate();

Predicate<String> notIsEmptyAndIsShortWord =
                                isEmpty.negate().and(isShortWord);
```

- `Collection.removeIf()`

```
List<String> names = new ArrayList<>(Arrays.asList("Tim",
                                        "Tom", "Andy", "Mike"));

names.removeIf(isShortWord)
names.forEach(System.out::println);    => Andy Mike
```

UnaryOperator und Bulk Operationen

zühlke
empowering ideas

- UnaryOperator<T> -- Aktionen formulieren

```
UnaryOperator<String> nullToEmpty = str -> str == null ? "" : str;

UnaryOperator<String> trimmer = String::trim;
```

- Collection.replaceAll() -- Aktionen ausführen

```
List<String> names = new ArrayList<>(Arrays.asList("Tim", null,
                                        " Tom ", "  Andy", "Mike"));

names.replaceAll(nullToEmpty);
names.replaceAll(trimmer);
names.forEach(s -> System.out.print("'" + s + "', "));
```

=> 'Tim', '', 'Tom', 'Andy', 'Mike',

zühlke
empowering ideas

Part 3: Streams

Filter, Map, Reduce

Michael Inden

Was sind Streams?

zühlke
empowering ideas

Streams als **neue Abstraktion** für Folgen von Verarbeitungsschritten

Analogie **Collection**, aber **keine Speicherung** der Daten

Analogie **Iterator**, Traversierung, aber **weitere Möglichkeiten zur Verarbeitung**

Design der Abarbeitung als Pipeline oder Fliessband

SRC-> STREAM *-> OP ->OP -> OP -> OP -> OP* **-> DEST**

Create Intermediate Terminal

Umschaltung **sequentiell** <-> **parallel** nach jedem Schritt der Pipeline **möglich,**
aber letzter gewinnt

Streams – Create-Operations

zühlke
empowering ideas

Aus Arrays oder Collections: stream(), parallelStream()

```
String[] namesData = { "Karl", "Ralph", "Andi", "Andi«, "Mike" };
List<String> names = Arrays.asList(namesData);

Stream<String> streamFromArray = Arrays.stream(namesData);
Stream<String> streamFromList = names.parallelStream();
```

Für definierte Wertebereiche: of(), range()

```
Stream<Integer> streamFromValues = Stream.of(17, 23, 2, 6, 7, 2, 14, 7);
IntStream values = IntStream.range(0, 100);
IntStream chars = "This is a test".chars();
```

Streams – Intermediate- und Terminal-Operations

zühlke
empowering ideas

Intermediate-Operations

- beschreiben **Verarbeitung**, sind **aber LAZY** (führen nichts aus!)
- erlauben es, **Verarbeitung bzw. Ausgabe auf spezielle Elemente zu beschränken**
- geben **Streams zurück** und erlauben so **Stream-Chaining**

```
streamFromValues.sorted().distinct().skip(10).limit(25). ….
```

Terminal-Operations

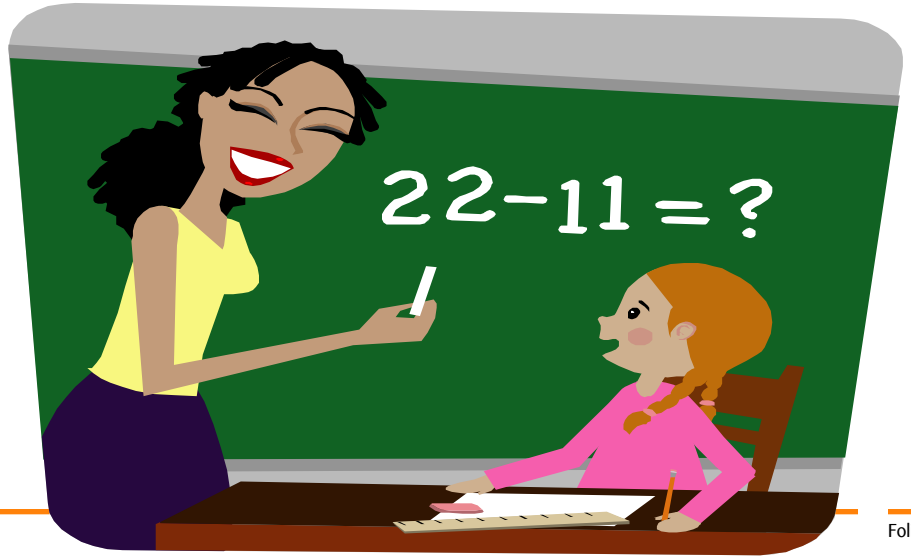
- sind **EAGER** und führen zur **Abarbeitung** der Pipeline
- **produzieren Ergebnis**: Ausgabe oder Sammlung in Collection usw.

```
streamFromArray.forEach(System.out::println);
streamFromValues.sorted().distinct().forEach(System.out::println);
```

Terminal-Operations – Collectors.joining, groupingBy, partitioningBy

zühlke
empowering ideas

```
List<String> names = Arrays.asList("Stefan", "Ralph", "Andi", "Mike",
                                   "Florian", "Michael", "Sebastian");

String joined = names.stream().sorted().collect(Collectors.joining(", "));

Object grouped = names.stream().collect(groupingBy(String::length));

Object grouped2 = names.stream().collect(groupingBy(String::length, counting()));

Object partition = names.stream().filter(str -> str.contains("i")).
                                  collect(partitioningBy(str -> str.length() > 4));
```



Terminal-Operations – Collectors.joining, groupingBy, partitioningBy

zühlke
empowering ideas

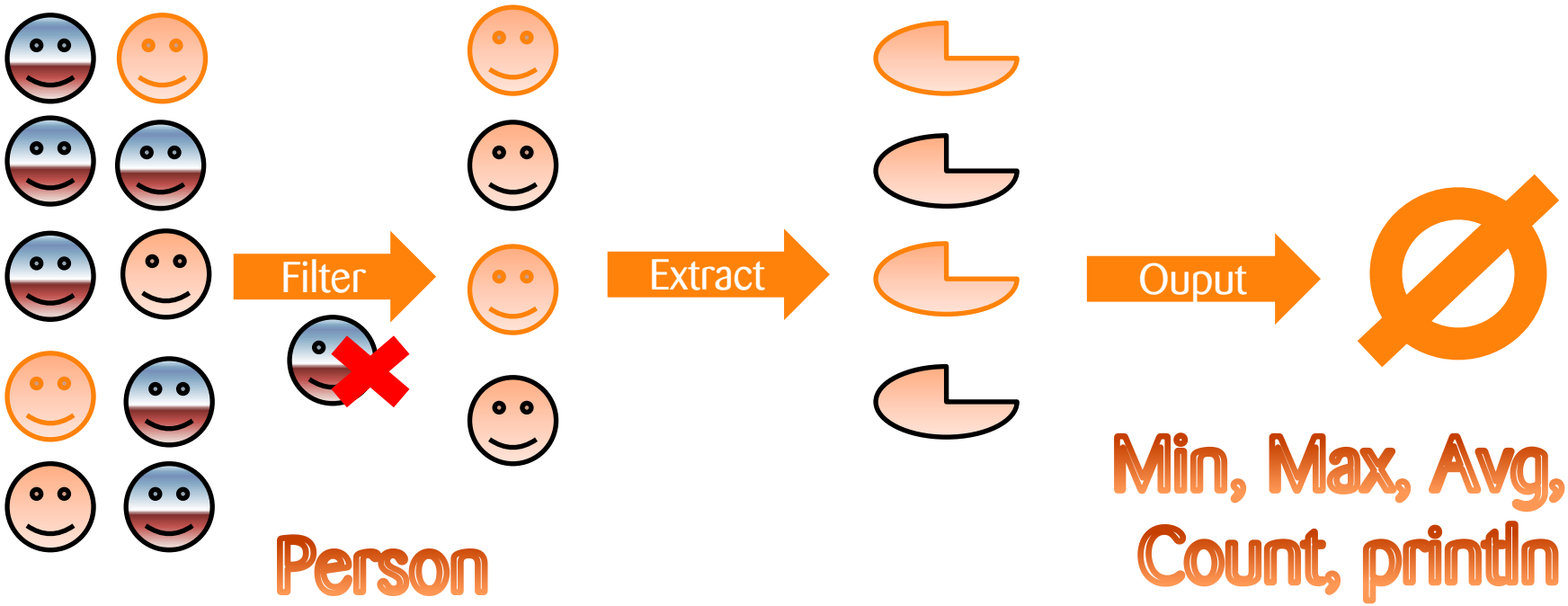
```
List<String> names = Arrays.asList("Stefan", "Ralph", "Andi", "Mike",
                                   "Florian", "Michael", "Sebastian");

String joined = names.stream().sorted().collect(Collectors.joining(", "));
Object grouped = names.stream().collect(groupingBy(String::length));
Object grouped2 = names.stream().collect(groupingBy(String::length, counting()));
Object partition = names.stream().filter(str -> str.contains("i")).
                                   collect(partitioningBy(str -> str.length() > 4));
```

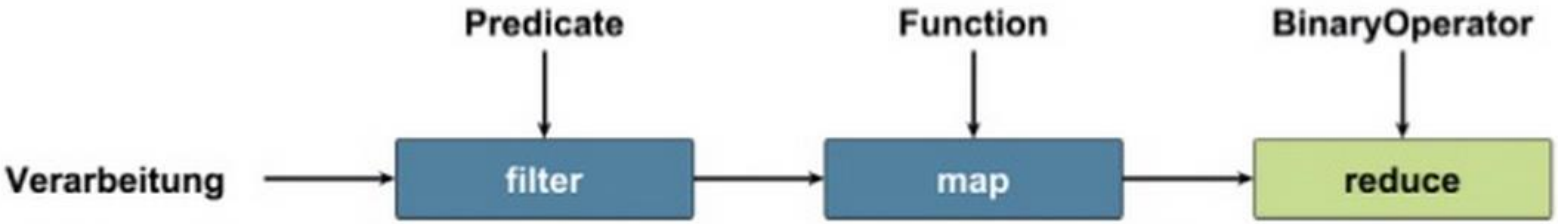
---

joined:	Andi, Florian, Michael, Mike, Ralph, Sebastian, Stefan
grouped:	{4=[Andi, Mike], 5=[Ralph], 6=[Stefan], 7=[Florian, Michael], 9=[Sebastian]}
grouped2:	{4=2, 5=1, 6=1, 7=2, 9=1}
partition:	{false=[Andi, Mike], true=[Florian, Michael, Sebastian]}

Filtere eine Liste und extrahiere Daten

zühlke
empowering ideas





Aufgabenstellung: Filtere eine Liste und extrahiere Daten

zühlke
empowering ideas

Gegeben sei folgende `List<Person>`:

```
List<Person> persons = Arrays.asList(
        new Person("Stefan", LocalDate.of(1971, Month.MAY, 20)),
        new Person("Micha", LocalDate.of(1971, Month.FEBRUARY, 7)),
        new Person("Andi Bubolz", LocalDate.of(1968, Month.JULY, 17)),
        new Person("Andi Steffen", LocalDate.of(1970, Month.JULY, 17)),
        new Person("Merten", LocalDate.of(1975, Month.JUNE, 14)));
```

Aufgabe:

1. Filtere auf alle im Juli Geborenen
2. Extrahiere ein Attribut, z.B. den Namen
3. Bereite eine kommaseparierte Liste auf

Herkömmlicher Ansatz: Alles einzeln ausprogrammieren

zühlke
empowering ideas

1. Filtere auf alle im Juli Geborenen

```
List<Person> bornInJuly = new ArrayList<>();
for (Person person : persons) {
    if (person.birthday.getMonth() == Month.JULY) {
        bornInJuly.add(person);
    }
}
```

2. Extrahiere ein Attribut, z. B. den Namen

```
List<String> names = new ArrayList<>();
for (Person person : bornInJuly) {
    names.add(person.name);
}
```

Herkömmlicher Ansatz: Alles einzeln ausprogrammieren

zühlke
empowering ideas

3. Bereite eine kommaseparierte Liste auf

```
String result = "";
Iterator<String> it = names.iterator();
while (it.hasNext())
{
   result += it.next();
   if (it.hasNext()) {
        result += ", ";
   }
}
```

=> Andi Bubolz, Andi Steffen

Herkömmlicher Ansatz

zühlke
empowering ideas

- Wie findet ihr den Code? Was könnte problematisch sein?

JDK 8-Lösung: Filter-Map-Reduce und Lambdas einsetzen

zühlke
empowering ideas

1. **Filter**: Filtere auf alle im Juli Geborenen

```
String result = persons.stream().filter(person -> person.birthday.getMonth() == Month.JULY).
```

2. **Map**: Extrahiere ein Attribut, z.B. den Namen

```
map(person -> person.name).
```

3. **Reduce**: Bereite eine kommaseparierte Liste auf

```
reduce("", (str1, str2) -> { if (str1.isEmpty()) {
                                 return str2;
                             } else {
                                 return str1 + ", " + str2;
                             }} );
```

JDK 8-Lösung: Filter-Map-Reduce und Lambdas einsetzen

zühlke
empowering ideas

Lesbarkeit durch eigene Klasse verbessern

```
String result = persons.stream().filter(person -> person.birthday.getMonth() == Month.JULY).
                                 map(person -> person.name).
                                 reduce("", stringCombiner);
```

```
BinaryOperator<String> stringCombiner = (str1, str2) -> { if (str1.isEmpty()) {
                                                              return str2;
                                                          } else {
                                                              return str1 + ", " + str2;
                                                          }};
```

Alternative: Ersetze reduce() durch collect() und nutze Collectors

```
Strinbg result = persons.stream(). filter(person -> person.birthday.getMonth() == Month.JULY).
                                   map(person -> person.name).
                                   collect(Collectors.joining(", "));
```

Streams & Separation Of Concerns

rot = I/O, grün = Ergebnislist,
gelb = Auswahl, blau = Zähllogik

zühlke
empowering ideas

Aufgabe: Ermittle alle Zeilen aus einer Log-Dateien die den Text «Error» enthalten, beschränke die Treffermenge auf die ersten 10 Vorkommen

```
final List<String> errorLines = new ArrayList<>();
try (final BufferedReader reader = new BufferedReader(new FileReader(inputFile)))
{
        String currentLine = reader.readLine();
        while (errorLines.size() < maxCount && currentLine != null)
        {
                if (currentLine.contains("ERROR"))
                {
                        errorLines.add(currentLine);
                }
                currentLine = reader.readLine();
        }
}
return errorLines;
```

- Nutzt externe Iteration
- Vielmehr Code als eigentlich zu erwarten, viel Glue Code
- Zugrundeliegender Algorithmus / Aufgabe kaum ersichtlich

Separation Of Concerns

rot = I/O, grün = Ergebnislist,
gelb = Auswahl, blau = Zähllogik

zühlke
empowering ideas

JDK 8-Realisierung deutlich einfacher:

```
final List<String> errorLines = Files.lines(inputFile.toPath())
                                .filter(line -> line.contains("ERROR"))
                                .limit(maxCount)
                                .collect(Collectors.toList());

return errorLines;
```

- Nutzt interne Iteration
- Nahezu kein Glue Code, sondern nur relevanter Code
- Zugrundeliegender Algorithmus / Aufgabe klar ersichtlich und gut lesbar

Streams and Maps …

zühlke
empowering ideas

- Maps arbeiten nicht mit Streams ;-(
- Aber … Das Interface `Map` wurde um eine Vielzahl an Methoden erweitert, die das Leben erleichtern:
  - `forEach()`
  - `putIfAbsent()`
  - `computeIfPresent()`
  - `getOrDefault()`

Map-Neuerungen im Überblick

zühlke
empowering ideas

```
final Map<String, Integer> map = new TreeMap<>();
map.put("c", 3);
map.put("b", 2);
map.put("a", 1);

final StringBuilder result = new StringBuilder();
map.forEach((key,value) -> result.append("(" + key + ", " + value + ") "));
System.out.println(result);

System.out.println(map.getOrDefault("XXX", -4711));
map.putIfAbsent("XXX", 7654321);
map.computeIfPresent("XXX", (key,value) -> value + 123456);
System.out.println(map.getOrDefault("XXX", -4711));
```

=>

```
(a, 1) (b, 2) (c, 3)
-4711
7777777
```

zühlke
empowering ideas

Part 4: Date and Time API

Warum noch ein weiteres Datums-API?

JSR-310 – Date and Time API im Einsatz

Michael Inden

Warum noch ein weiteres Datums-API?

zühlke
empowering ideas

- **Verarbeitung von Datumswerten und Zeit scheint einfach, ist es aber nicht**

- **Tatsächlich ist es sogar ziemlich kompliziert**
  - Einfluss von Zeitzonen
  - Einfluss von Schaltjahren
  - Einfluss von Sommer- und Winterzeit
  - Usw.

- **Beispiel "Gehe einen Monat in die Vergangenheit / Zukunft"**
  - Was ist ein Monat und wie wird dieser dargestellt?
  - Monat anpassen
  - Schaltjahr berücksichtigen
  - Ggf. Jahr anpassen
  - Ggf. Uhrzeit anpassen
  - usw.

Warum noch ein weiteres Datums-API?

zühlke
empowering ideas

Wurf 1: `java.util.Date` (JDK1.0)

- nur **minimale Abstraktion** eines `long` zum Offset 1.1.1970 00:00:00 Uhr
- Verschiedene Offsets (1900 / 1970, 0- und 1-basiert usw.)
- Verarbeitung von Datum und Zeit ist damit mühseelig und fehleranfällig


```
// Mein Geburtstag: 7.2.1971
final int year = 1971;
final int month = 2;
final int day = 7;
final Date myBirthday = new Date(year, month, day);
System.out.println(myBirthday);
```


Warum noch ein weiteres Datums-API?

zühlke
empowering ideas

=> Tue Mar 07 00:00:00 CET 3871

Korrektur: new Date(year - 1900, month - 1, day)

Warum noch ein weiteres Datums-API?

zühlke
empowering ideas

Wurf 2: `java.util.Calendar` (JDK1.1)

- ist **besser gelungen** und bietet eine bessere Abstraktion (Konstanten für Monate, Addition von Zeitwerten usw.)
- Verarbeitung wird deutlich leichter, vor allem Berechungen
- **ABER: Es ist immer noch Einiges ziemlich kompliziert, etwa wenn man nur mit Zeitangaben oder Datumswerten rechnen möchte**

Alternative: Joda-Time

zühlke
empowering ideas

- Probleme auch bei SUN / Oracle im Bewusstsein, aber es passierte nichts
- Abhilfe für JDK 7 versprochen, aber erst für JDK 8 adressiert
- **Zwischenzeitlich: Joda-Time**

JSR-310: Date And Time API

zühlke
empowering ideas

Wurf 3: JSR 310 – Neuer (dritter) Wurf eines Datums-APIs im JDK

- Viel ist besser gelungen als die Vorgänger
- basiert auf der erfolgreichen JodaTime-Bibliothek (von S. Colebourne)

Designziele:

- **Klarheit und Verständlichkeit,** “Works-as-expected”
- **Fluent Interface,** sprechende Methodennamen, Method-Chaining
- **Immutable,** somit automatisch Thread-Safe

ABER: **kommt viel zu spät, da Probleme seit Jahren (Jahrzehnten) existieren**

JSR-310: Berechnungen und mehr

zühlke
empowering ideas

```
final LocalDate now = LocalDate.now();

System.out.println("Today: " + now);

System.out.println("DayOfWeek: " + now.getDayOfWeek());
System.out.println("DayOfMonth: " + now.getDayOfMonth());
System.out.println("DayOfYear: " + now.getDayOfYear());

System.out.println("Month: " + now.getMonth());
System.out.println("LengthOfMonth: " + now.lengthOfMonth());
System.out.println("Days in Month: " + now.getMonth().length(now.isLeapYear()));

System.out.println("LengthOfYear: " + now.lengthOfYear());
```

```
Today: 2014-12-01

DayOfWeek: MONDAY
DayOfMonth: 1
DayOfYear: 335

Month: DECEMBER
LengthOfMonth: 31
Days in Month: 31

LengthOfYear: 365
```

JSR-310: Berechnungen und mehr

zühlke
empowering ideas

Fluent API

```
LocalDate jan15 = LocalDate.parse("2015-01-15");

LocalDate myStartAtSwisscom = jan15.plusDays(5) ;
myStartAtSwisscom = myStartAtSwisscom .minusYears(1);
System.out.println(myStartAtSwisscom);                    // 2014-01-20

LocalDate jan15_2015 = LocalDate.of(2015, Month.JANUARY, 15);
System.out.println(jan15_2015.getDayOfWeek());            // THURSDAY

LocalDate feb7_2015 = jan15_2015.withMonth(2).withDayOfMonth(7);
System.out.println(feb7_2015.getDayOfYear());             // 38
```

```
LocalDate feb7_2015 = jan15_2015.withMonth(Month.FEBRUARY).withDayOfMonth(7);
```

JSR-310: TemporalAdjusters & Lesbarkeit

zühlke
empowering ideas

// STATISCHE IMPORTS vs. QUALIFIZIERTE REFERENZIERUNG

```
import static java.time.Month.AUGUST;
import static java.time.DayOfWeek.SUNDAY;
import static java.time.temporal.TemporalAdjusters.firstInMonth;
import static java.time.temporal.TemporalAdjusters.lastInMonth;
```

```
// FRIDAY 2015-08-14
LocalDate midOfAgust = LocalDate.of(2015, AUGUST, 14);

// MONDAY 2015-08-31
LocalDate lastOfAgust = midOfAgust.with(TemporalAdjusters.lastDayOfMonth());

// WEDNESDAY 2015-08-05
LocalDate firstWednesday = lastOfAgust.with(firstInMonth(DayOfWeek.WEDNESDAY));

// SUNDAY 2015-08-30
LocalDate lastSunday = lastOfAgust.with(lastInMonth(SUNDAY));
```

zühlke
empowering ideas

Part 5: Weitere Funktionen

- Comparator<T>

Michael Inden

Comparator<T>

zühlke
empowering ideas

- comparing() – Definiert einen Komparator basierend auf der Extraktion zweier Werte, die sich mit Comparable<T> vergleichen lassen.
- thenComparing(), thenComparingInt()/-Long() und -Double() – Hintereinanderschaltung von Komparatoren

```
Comparator<Person> byName = Comparator.comparing(Person::getName);

Comparator<Person> byAge = Comparator.comparing(Person::getAge);

// Kombination von Komparatoren

Comparator<Person> byNameAndFirstname = byName.
                                        thenComparing(byFirstname);

Comparator<Person> byNameAndAge = byName.thenComparing(byAge);
```


Comparators Pitfalls – reverse order

zühlke
empowering ideas

What about reverse sorting of the ages?

```
Comparator<Person> byCityAndAge = comparing(Person::getCity).
                                  thenComparingInt(Person::getAge).
                                  reversed();
```

Uups, we reversed the hole sorting … but how to reverse just ages?

```
Comparator<Person> byCity= comparing(Person::getCity);
Comparator<Person> byAge = comparing(Person::getAge);

Comparator<Person> byCityAndJustAgeReversed =
                          byCity.thenComparing(byAge.reversed());
```

zühlke
empowering ideas

Zusammenfassung und Links

Michael Inden

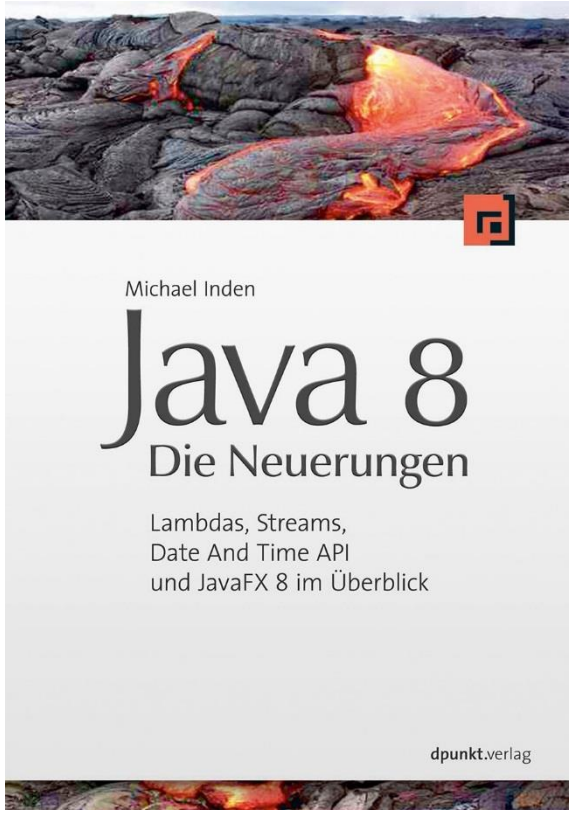
Zusammenfassung – Java 8 bietet mit…

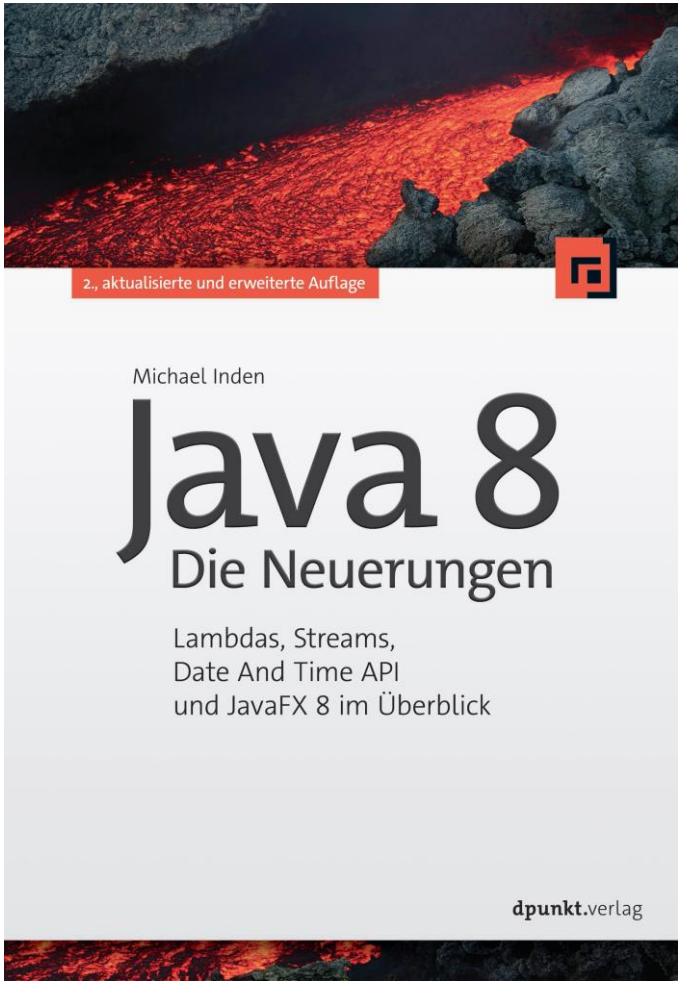
zühlke
empowering ideas

- **Lambdas** ein **neues Programmiermodell (funktional)**
- **Streams** mit **filter/map/reduce** eine **umfangreiche Erweiterung im Collections-Framework**
- dem **neuen Date & Time API** eine **deutliche Vereinfachung**
- **JavaFX 8** verschiedene **neue Controls** und Unterstützung für **3D**
- diversen **API-Erweiterungen** eine **Erleichterung** beim täglichen Entwickeln
- "**Nashorn**" eine neue **performantere JavaScript-Engine**

Weiterführende Infos und Links

zühlke
empowering ideas





Weiterführende Infos und Links

zühlke
empowering ideas

JDK 8 Project
https://jdk8.java.net/

Trying Out Lambda Expressions in the Eclipse IDE
http://www.oracle.com/technetwork/articles/java/lambda-1984522.html

Lambda Expressions and Streams in Java - Tutorial & Reference
http://www.angelikalanger.com/Lambdas/Lambdas.html

JavaFX
http://docs.oracle.com/javafx/

Getting Started with JavaFX 3D Graphics
http://docs.oracle.com/javafx/8/3d_graphics/jfxpub-3d_graphics.htm

JavaFX 8 Container-Terminal
http://www.youtube.com/embed/AS26gZrYNy8?rel=0

The End

zühlke
empowering ideas

Vielen Dank für die Aufmerksamkeit!

Viel Spaß bei der eigenen Entdeckungsreise zu Java 8!