

Michael Inden

Workshop: Best of Java 8 und 9

Speaker – Kurzlebenslauf

- 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
- Autor und Gutachter
beim dpunkt.verlag



Table of Contents – Java 8

- Part 1: Lambdas, Defaultmethoden und Methodenreferenzen
- Part 2: Bulk Data Operations on Collections
- Part 3: Streams und Filter-Map-Reduce
- Part 4: Date And Time API
- Part 5: JavaFX 8
- Part 6: Weitere Funktionen

Part 1: Lambdas

Motivation, Syntax & SAM Defaultmethoden und Methodenreferenzen

Warum Lambdas?



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

Warum Lambdas?

- Als Krücke auch in Java!
Wirklich, aber wie?
- **Schön**, bereits seit langem in Sprachen wie Groovy und Scala
- Lösungen auf sehr elegante Art und Weise formulieren
- andere Denkweise und neuer Programmierstil (funktional)
- Hilfe für Parallelverarbeitung und Ausnutzung von Multicores



Syntax von Lambdas

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

(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

Ein Lambda ist KEIN Object

- Lambdas besitzen keinen Obertyp wie in Groovy etwa den Typ **Closure**
- **Können nicht dem Typ Object zugewiesen werden**

```
Object lambda = () -> System.out.println("compile-error");
```

“The target type of this expression must be a functional interface”

- Was ist denn nun ein Functional Interface?

Beispiele für Functional Interfaces (SAM-Typen)

Viele bekannt aus Funk und Fernsehen ... äh ... dem JDK

- Runnable, Callable, Comparable, Comparator, FileFilter, FilenameFilter, ActionListener, ChangeListener usw.

Neue Annotation **@FunctionalInterface** (Angabe optional)

@FunctionalInterface

```
public interface Runnable {  
    public abstract void run();  
}
```

@FunctionalInterface

```
public interface FileFilter{  
    boolean accept(File pathname);  
}
```

Grundlagen zu Lambdas

Lambdas als Implementierung eines Functional Interface:

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

=>

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

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

Wie geht es weiter SAM?



Grundlagen zu Lambdas

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()); };
```

Grundlagen zu Lambdas

Lambdas als Rückgabewerte für SAM

```
public static Comparator<String> stringLengthCompare() {  
    return (str1, str2) -> Integer.compare(str1.length(), str2.length());  
}
```

Lambdas als Eingabe für SAM

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

```
Collections.sort(names, (str1, str2) -> Integer.compare(str1.length(), str2.length()) );
```

```
Collections.sort(names, stringLengthCompare());
```

Beispiel: Sortierung nach Länge und komma-separierte Aufbereitung

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

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

`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

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: Defaultmethoden

Was passiert bei Konflikten?

```
interface Interface1 {  
    default int sameMethod(int x) {  
        return 0;  
    }  
}
```

```
interface Interface2 {  
    default int sameMethod(int x) {  
        return 4711;  
    }  
}
```

```
class ErroneousCombination implements Interface1, Interface2  
{}
```

Kompilierfehler “Duplicate default methods named sameMethod with the parameters (int) and (int) are inherited from the types Interface2 and Interface1”

Neuheit: Defaultmethoden

Konflikte auflösen: Eigene Methodenimplementierung vorgeben:

```
public class Right implements Interface1, Interface2 {  
    public int sameMethod(int x) {  
        return 7;  
    }  
}
```

Aufruf der Funktionalität der Defaultmethoden (**Spezialsyntax**)

```
public class Right implements Interface1, Interface2  
    public int sameMethod(int x) {  
        return Interface1.super.sameMethod(x);  
    }  
}
```

Neuheit: Methodenreferenzen

Methodenreferenz verweist auf ...

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

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

«Real World» Example



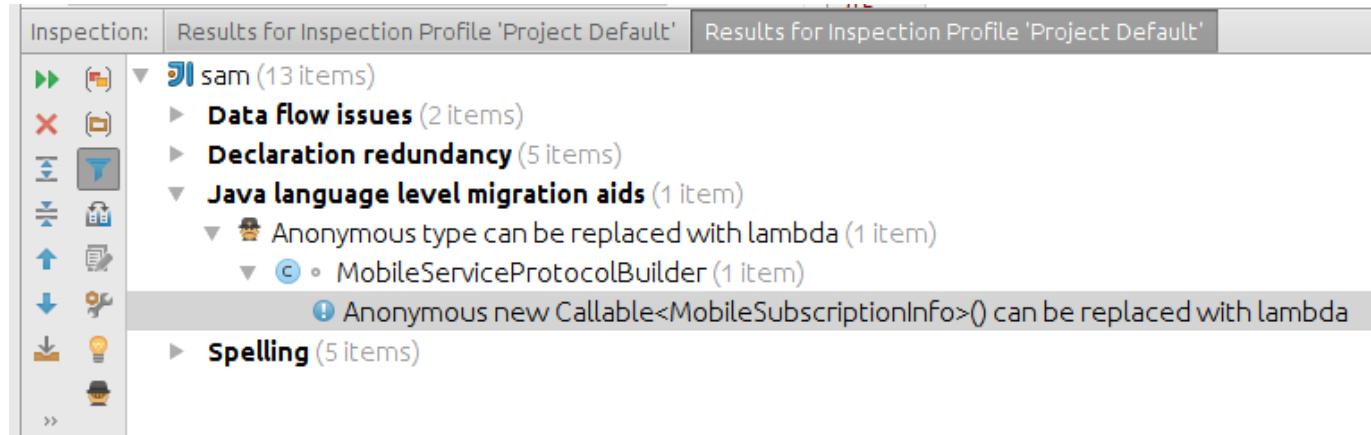
Folie 23

Example: Convert to lambda

```
private Future<MobileSubscriptionInfo> loadFutureSisData(final Msisdn msisdn) {  
    final Callable<MobileSubscriptionInfo> callableMobileSubscriptionInfo =  
        new Callable<MobileSubscriptionInfo>() {  
  
    @Override  
    public MobileSubscriptionInfo call() throws ExternalSystemException {  
        SisInfoApi sisInfoApi = DataAccessApiFactory.getInstance().getSisInfoApi();  
        return sisInfoApi.getDetailedMobileSubscriptionInfoByMSISDN(msisdn);  
    }  
};  
  
return getThreadPool().submit(callableMobileSubscriptionInfo);  
}
```

Example: Convert to lambda

ALT + ENTER //
Analyze -> Inspect Code



```
private Future<MobileSubscriptionInfo> loadFutureSisData(final Msisdn msisdn)
{
    final Callable<MobileSubscriptionInfo> callableMobileSubscriptionInfo = () ->
    {
        final SisInfoApi sisInfoApi = DataAccessApiFactory.getInstance().getSisInfoApi();
        return sisInfoApi.getDetailedMobileSubscriptionInfoByMSISDN(msisdn);
    };

    return getThreadPool().submit(callableMobileSubscriptionInfo);
}
```

Part 2: Bulk Operations on Collections

Externe und interne Iteration

Predicate<T>, UnaryOperator<T>

Externe Iteration vs interne Iteration

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);
```

Überblick Functional Interfaces

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

- **Consumer<T>** – Beschreibt eine Aktion auf einem Element vom Typ T. Dazu ist eine Methode **void accept(T)** definiert.

Neu:

- **Predicate<T>** – Definiert eine Methode **boolean test(T)**. Diese berechnet für eine Eingabe vom Typ T einen booleschen Rückgabewert
- **Function<T,R>** – Abbildungsfunktion in Form der Methode **R apply(T)**. Damit wird ein allgemeines Konzept von Transformationen beschrieben.
- **Supplier<T>** – Stellt ein Ergebnis vom Typ T bereit: Methode **T get()**

Prädikate und Bulk Operationen

- `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
```

Prädikate und Bulk Operationen

- Achtung: `asList()` -> **unmodifiableList!!**
- `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

- **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',
```

Part 3: Streams

Filter, Map, Reduce

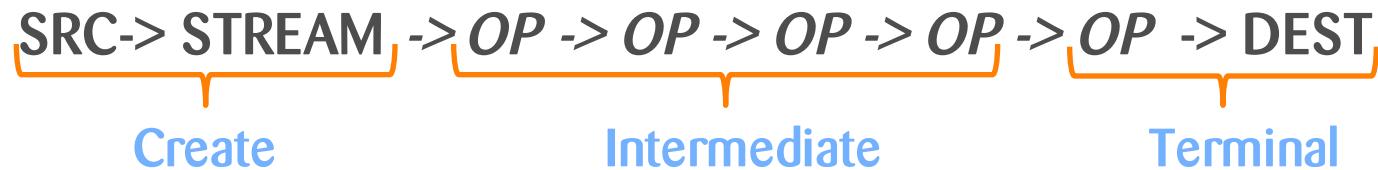
Was sind Streams?

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



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

Streams – Create-Operations

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

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

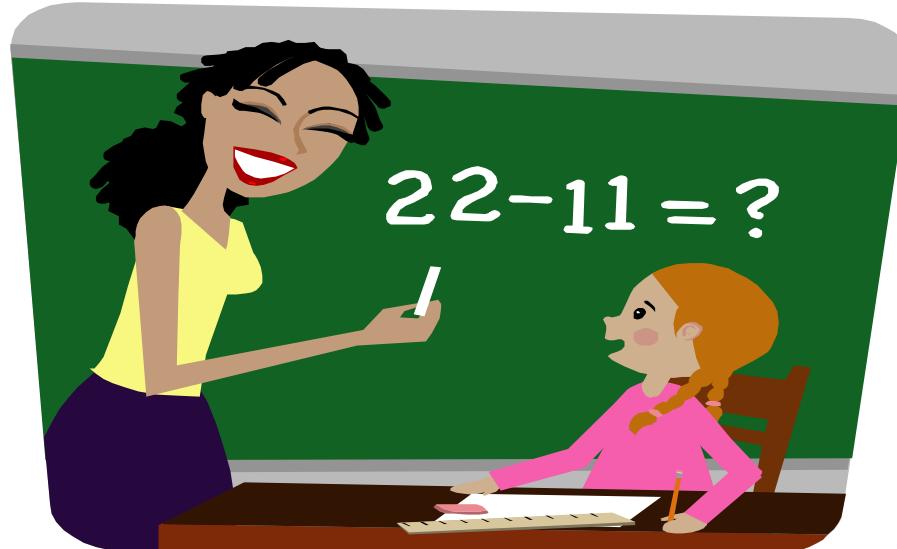
```
List<String> names = Arrays.asList("Stefan", "Ralph", "Andi", "Mike",  
        "Florian", "Michael", "Sebastian");
```

```
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

```
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));
```

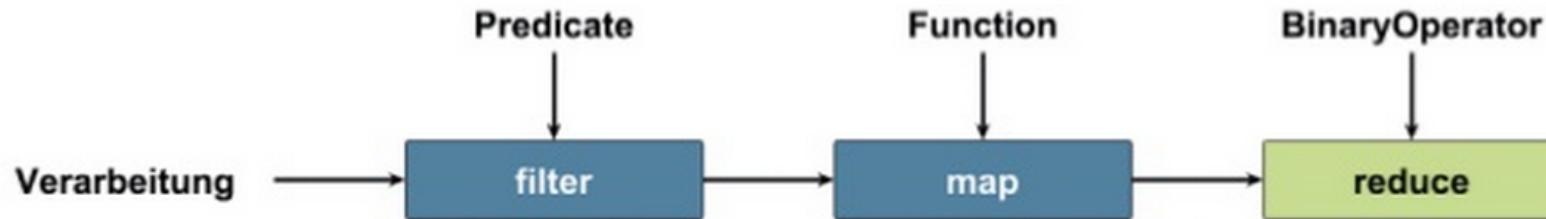
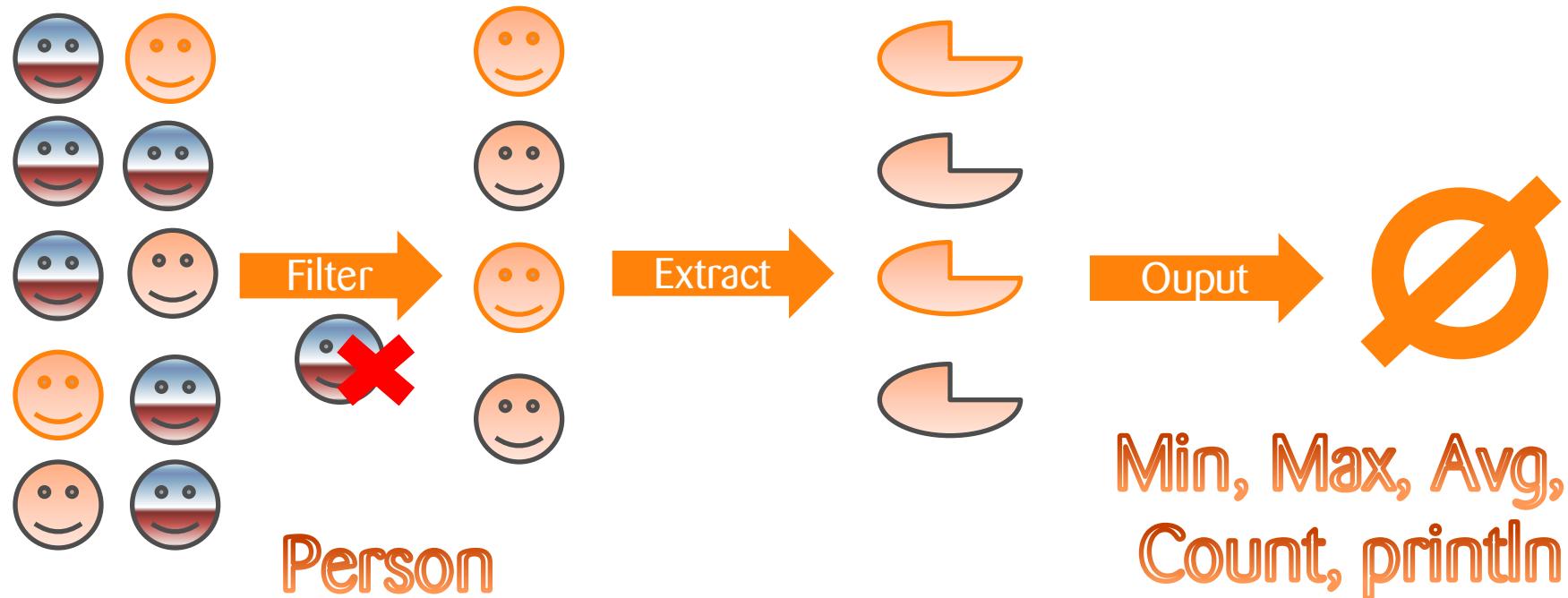
joined: Andi, Florian, Michael, Mike, Ralph, Sebastian, Stefan

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



Aufgabenstellung: Filtere eine Liste und extrahiere Daten

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

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

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

Wie findet ihr den Code? Was könnte problematisch sein?



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

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

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

```
String 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 = Ergebnisliste,
gelb = Auswahl, blau = Zähllogik

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 = Ergebnisliste,
gelb = Auswahl, blau = Zähllogik

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 ...

- **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

```
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
```

«Real World» Example



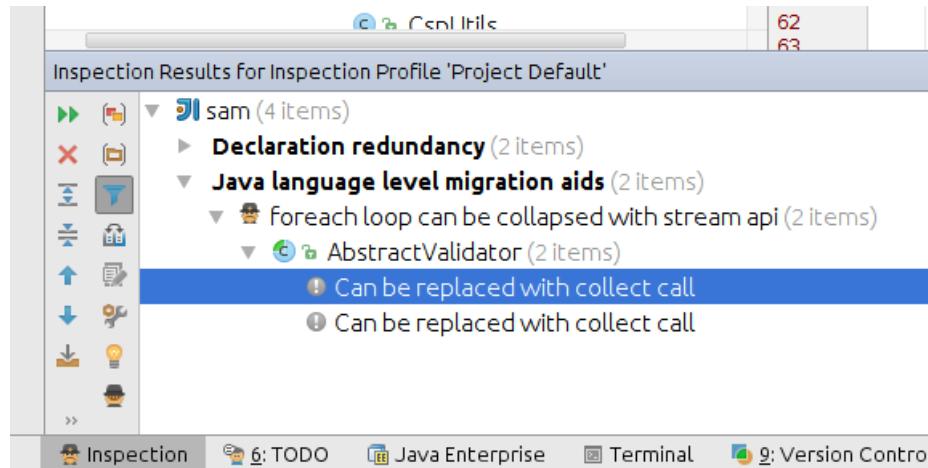
Folie 49

Example: External to Internal Iteration using Stream and Filter / Map / Reduce

```
@Override
public List<SimpleValidationMessage> getErrorListForId(final int id)
{
    final List<SimpleValidationMessage> list = new ArrayList<>();
    for (final SimpleValidationMessage simpleValidationMessage : errorMessages)
    {
        if (simpleValidationMessage.getId() == id)
        {
            list.add(simpleValidationMessage);
        }
    }
    return list;
}
```

Example: Umwandlung mit Automatik aus IntelliJ

Analyze > Inspect Code



```
@Override
public List<SimpleValidationMessage> getErrorListForId(final int id) {

    final List<SimpleValidationMessage> list = errorMessages.stream().
        filter(simpleValidationMessage -> simpleValidationMessage.getId() == id).
        collect(Collectors.toList());

    return list;
}
```

Hands on: Finish per Hand

```
@Override  
public List<SimpleValidationMessage> getErrorListForId(final int id) {  
  
    final Predicate<SimpleValidationMessage> withSameId = msg -> msg.getId() == id;  
  
    final List<SimpleValidationMessage> list = errorMessages.stream()  
        .filter(withSameId)  
        .collect(toList());  
  
    return list;  
}
```



Part 4: Date And Time API

Warum noch ein weiteres Datums-API?

JSR-310 – Date And Time API im Einsatz

Warum noch ein weiteres Datums-API?

- 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?

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 mühselig 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?



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

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

Warum noch ein weiteres Datums-API?

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 Berechnungen
- **ABER:** Es ist immer noch Einiges ziemlich kompliziert, etwa wenn man nur mit Zeitangaben oder Datumswerten rechnen möchte

Alternative: Joda-Time

- 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

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: Intuitive Datumswerte

Klarheit und Verständlichkeit, analog zu Denkweise von Menschen

// Varianten von LocalDate: Datum ohne Uhrzeit und Zeitzone

```
LocalDate today = LocalDate.now();
LocalDate jan23 = LocalDate.parse("2014-01-23");
LocalDate feb7 = LocalDate.of(2014, 2, 7);
LocalDate mar24 = LocalDate.of(2014, Month.MARCH, 24);
```

// Zeitangabe ohne Datum

```
LocalTime now = LocalTime.now();
LocalTime at_15_30 = LocalTime.parse("15:30");
LocalTime at_12_11_10 = LocalTime.of(12, 11, 10);
```

// Kombination aus Datum und Zeit

```
LocalDateTime nowWithTime = LocalDateTime.now();
LocalDateTime feb8_at_10_11 = LocalDateTime.parse("2014-02-08T10:11:12");
```

JSR-310: Berechnungen und mehr

```
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

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: Zeitspannen – Period & Duration

- **Period** – Datumsbasierter Zeitabschnitt: Monate, Wochen, Tage, ...
- **Duration** – Zeitbasierte Bereiche: Stunden, Minuten, Sekunden, ...

```
final LocalDateTime christmasEve = LocalDateTime.of(2016, 12, 24, 17, 30, 00);
final LocalDateTime silvester = LocalDateTime.of(2016, 12, 31, 23, 59, 59);
```

```
final Period week = Period.between(christmasEve.toLocalDate(),
                                   silvester.toLocalDate());
System.out.println("a week: " + week); // a week: P7D
System.out.println("period: " + Period.of(1, 2, 7)); // period: P1Y2M7D
```

```
final Duration sevenDays = Duration.ofDays(7);
System.out.println("sevenDays: " + sevenDays); // sevenDays: PT168H
```

```
final Duration duration = Duration.between(christmasEve, silvester);
System.out.println("duration: " + duration); // duration: PT174H29M59S
```

JSR-310: TemporalAdjusters & Lesbarkeit

// 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 midOfAugust = LocalDate.of(2015, AUGUST, 14);
```

// MONDAY 2015-08-31

```
LocalDate lastOfAugust = midOfAugust.with(TemporalAdjusters.lastDayOfMonth());
```

// WEDNESDAY 2015-08-05

```
LocalDate firstWednesday = lastOfAugust.with(firstInMonth(DayOfWeek.WEDNESDAY));
```

// SUNDAY 2015-08-30

```
LocalDate lastSunday = lastOfAugust.with(lastInMonth(SUNDAY));
```

JSR-310: Formatierung und Zeitzonen

```
final LocalDateTime ldt = LocalDateTime.of(2016, 7, 14, 5, 25, 45);
final String pattern = "Datum:' dd.MM.yyyy ' / Uhrzeit:' HH:mm";
final DateTimeFormatter formatter = DateTimeFormatter.ofPattern(pattern);
System.out.println("formattedDate " + formatter.format(ldt));
// Datum: 14.07.2016 / Uhrzeit: 05:25

final String zonedDateTime = "2007-12-03T10:15:30+01:00[Europe/Paris]";
final ZonedDateTime zdt = ZonedDateTime.parse(zonedDateTime);
System.out.print(zdt + " as LocalDateTime " + zdt.toLocalDateTime());
System.out.println(" / Zoneld " + zdt.getZone());
System.out.println(" / ZoneOffset " + zdt.getOffset());
// 2007-12-03T10:15:30+01:00[Europe/Paris]as LocalDateTime 2007-12-
03T10:15:30 / Zoneld Europe/Paris / ZoneOffset +01:00
```

Part 5: JavaFX 8

Rich Text Support, Look And Feel, Neue Controls, 3D-Support

JavaFX 8 – Old Swing vs. New JavaFX

Swing

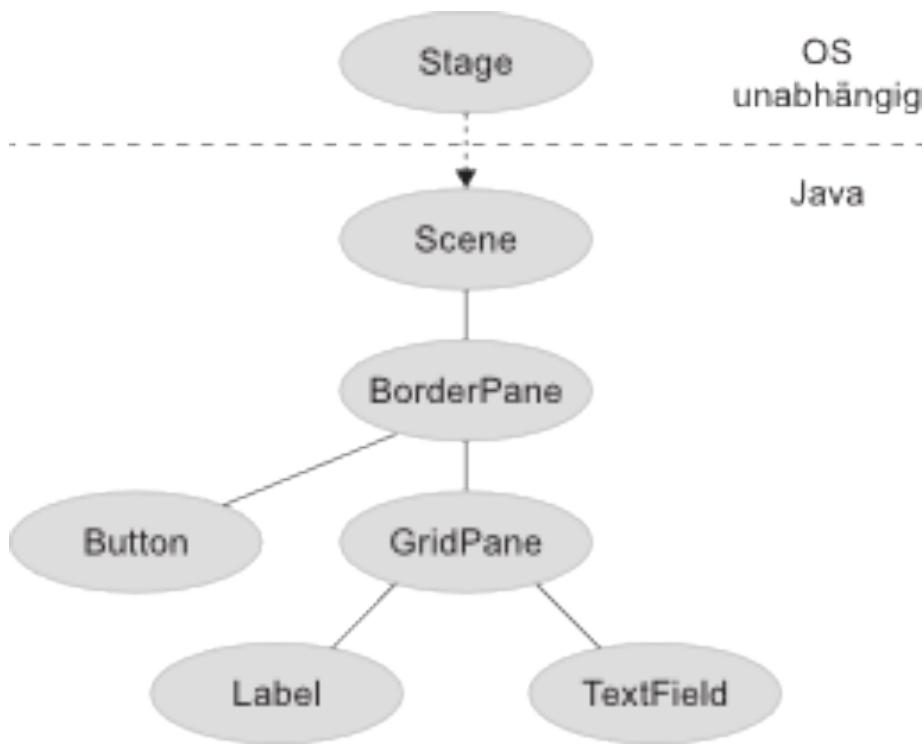
Firstname	Name	Gender
Joshua	Bloch	<input checked="" type="radio"/> MALE
Neil	Gafter	<input checked="" type="radio"/> MALE
James	Gosling	<input checked="" type="radio"/> MALE
Bart	Bates	<input checked="" type="radio"/> MALE
Kathi	Sierra	<input type="radio"/> FEMALE

JavaFX

Name	Age	Size	
All Persons			
Group 1			
Micha	43	184 cm	
Lili	34	170 cm	
Tom	22	177 cm	
Group 2			
Jens	47	175 cm	
Tim	43	181 cm	
Andy	31	178 cm	

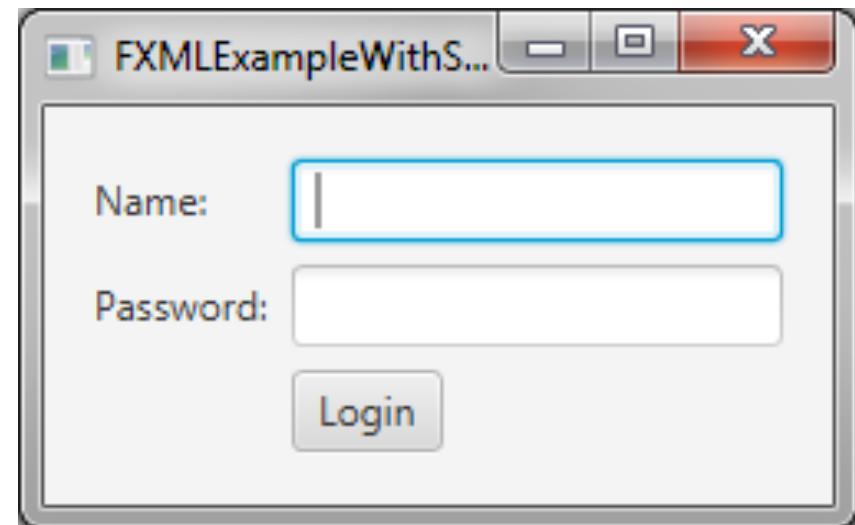
JavaFX 8 – Stage, Scene & Nodes

Scenegraph and more



OS
unabhängig

Java

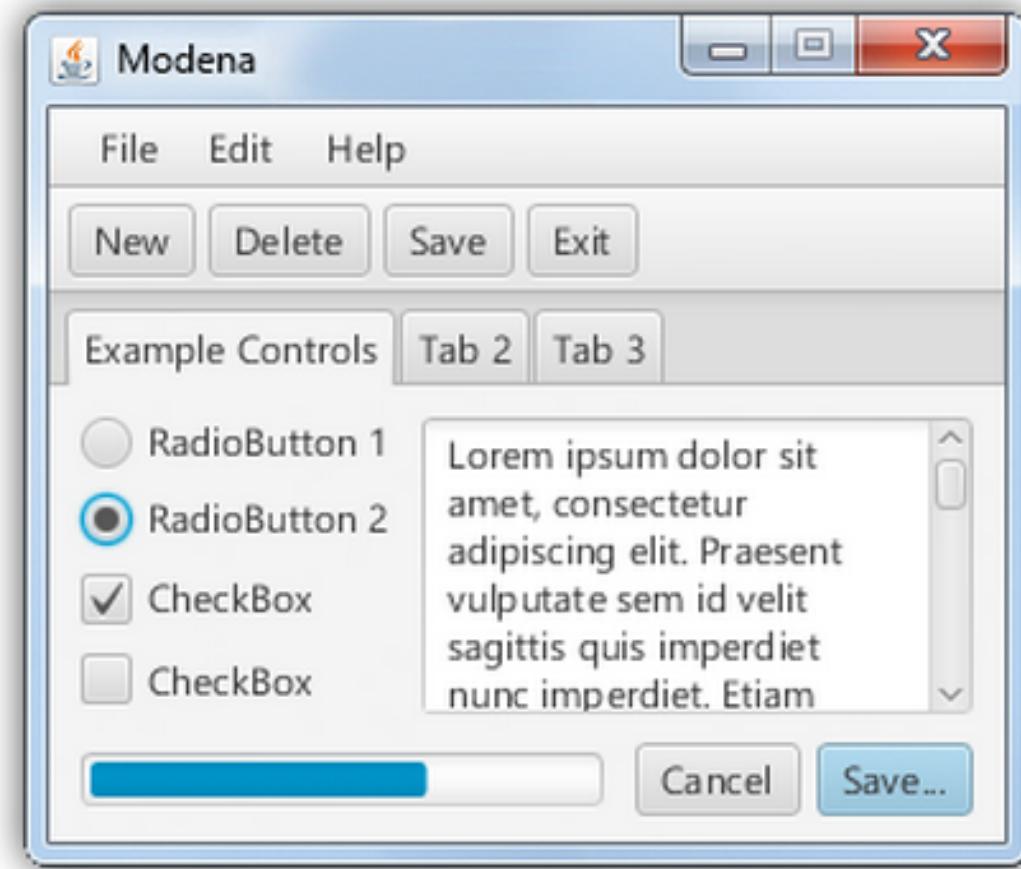


JavaFX 8 – Rich Text und Look And Feel

Rich Text Support



New Look And Feel

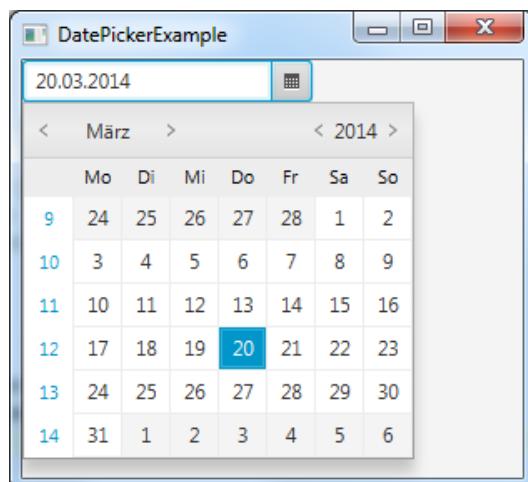


JavaFX 8 – Neue Controls

TreeTableView

Year	Balance		Comments
	Income	Spending	
2001	\$12,000	\$12,000	Years 2001 and 2002 w/ significant improvement
2002	\$14,000	\$14,000	
2003	\$6,000	\$8,000	Drop followed by a sizeable
2004	\$18,000	\$18,000	rebound
2005	\$19,000	\$19,000	<html>2005 and 2006 b/ success.
2006	12000	\$12,000	
2007	\$20,000	\$20,000	

DatePicker



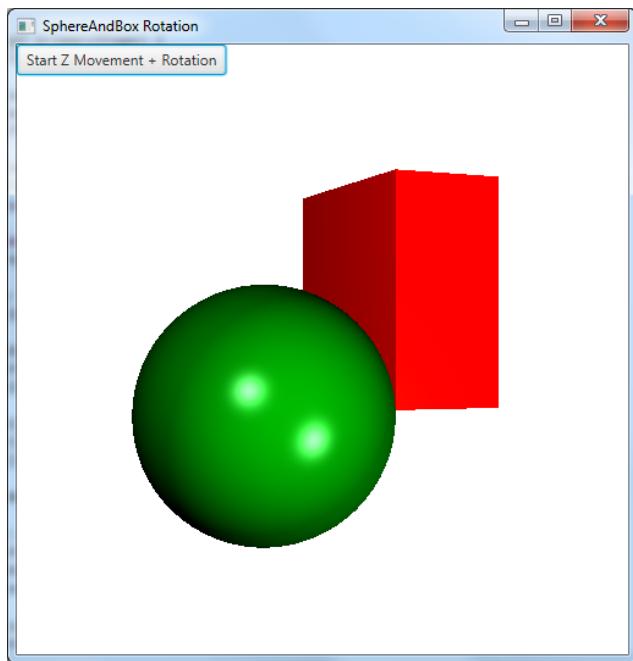
Editor
Viewer

Editor - Viewer

Editor - Viewer

JavaFX 8 – 3D Support

Beispiel



VIDEO:

<http://www.youtube.com/embed/AS26gZrYNy8?rel=0>



Part 6: Weitere Funktionen

Comparator<T> und Optional<T>

Comparator<T>

- `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

What about reverse sorting of the ages?

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

Ups, 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());
```

Optional<T>

Modellierung von optionalen Werten als Alternative zu null/Null-Objekt

```
final Integer[] values = {1, 3, 5, 7}; // Was passiert bei {} ?  
final Optional<Integer> min = Arrays.stream(values).  
                                         min(Comparator.naturalOrder());
```

```
// Prüfe, ob es einen Wert gibt  
System.out.println("isPresent?: " + min.isPresent());
```

```
// Zugriff auf den Wert  
System.out.println("minValue: " + min.get());
```

```
// Führe Aktion aus, wenn vorhanden  
min.ifPresent(System.out::println);
```

```
// Alternativen Wert liefern, wenn nicht vorhanden  
System.out.println(min.orElse(-1));
```

«Real World» Example



Folie 76

Example: Comparator vereinfachen

```
// sort by name
Collections.sort(shopListDto, new Comparator<ShopDbShopDto>() {
    @Override
    public int compare(ShopDbShopDto s1, ShopDbShopDto s2) {
        return s1.getName().compareTo(s2.getName());
    }
});
```

Example: Umwandlung mit Automatik aus IntelliJ und händisches Tuning

Automatik:

```
Collections.sort(shopListDto, (s1, s2) -> s1.getName().compareTo(s2.getName()));
```

Tuning:

```
Comparator<ShopDbShopDto> byName = comparing(ShopDbShopDto::getName);  
shopListDto.sort(byName);
```

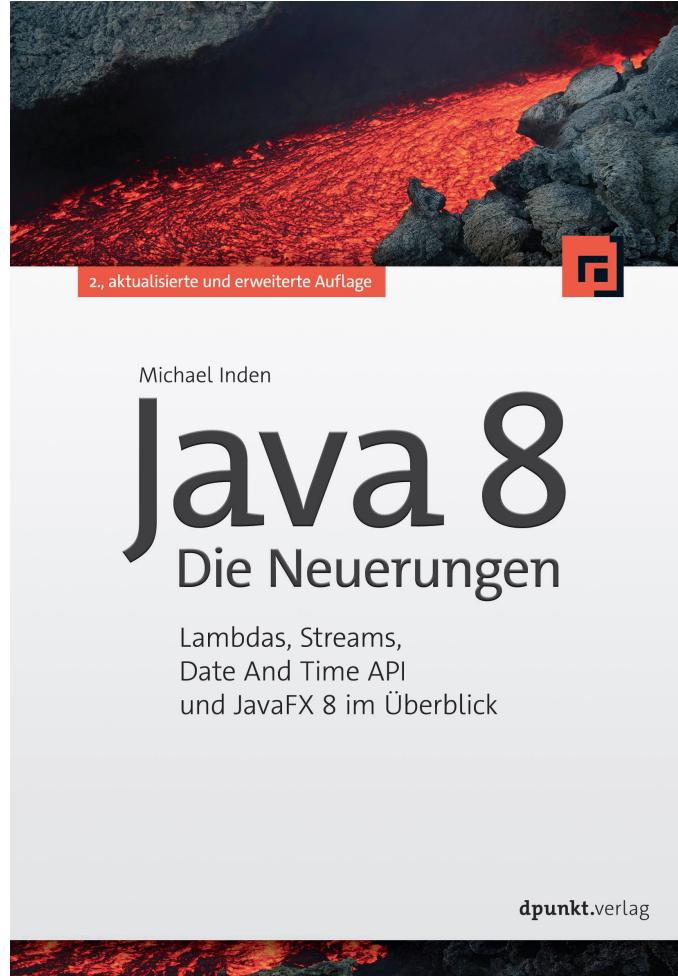


Zusammenfassung und Links

Zusammenfassung – Java 8 bietet mit...

- 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



Weiterführende Infos und Links

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-Tutorial

<http://www.youtube.com/embed/AS26gZrYNy8?rel=0>

The End

Vielen Dank für die Aufmerksamkeit!

Viel Spaß bei der eigenen Entdeckungsreise zu Java 8!