

Neuerungen in Java 8

Wichtige neue Features im Überblick

Michael Inden

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
- ~4 Jahre bei Zühlke Engineering AG in Zürich
- Ab Juni 2017 bei Direct Mail Company 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: Weitere Funktionen
- Part 6: JavaFX 8

Part 1: Lambdas

Motivation, Syntax & SAM

Defaultmethoden und Methodenreferenzen

Warum Lambdas?



Lambdas als ein neues und heiß ersehntes Sprachkonstrukt

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

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

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 Rückgabetyt ist kompatibel.

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> byLength() {  
    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, byLength());
```

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(o1.length(), o2.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: Statische Methoden in Interfaces

Beispiel Interface Comparator<T>

```
public static <T extends Comparable<? super T>> Comparator<T> reverseOrder()  
{  
    return Collections.reverseOrder();  
}
```

```
public static <T extends Comparable<? super T>> Comparator<T> naturalOrder()  
{  
    return (Comparator<T>) Comparators.NaturalOrderComparator.INSTANCE;  
}
```

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

```
// Lambda  
// 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

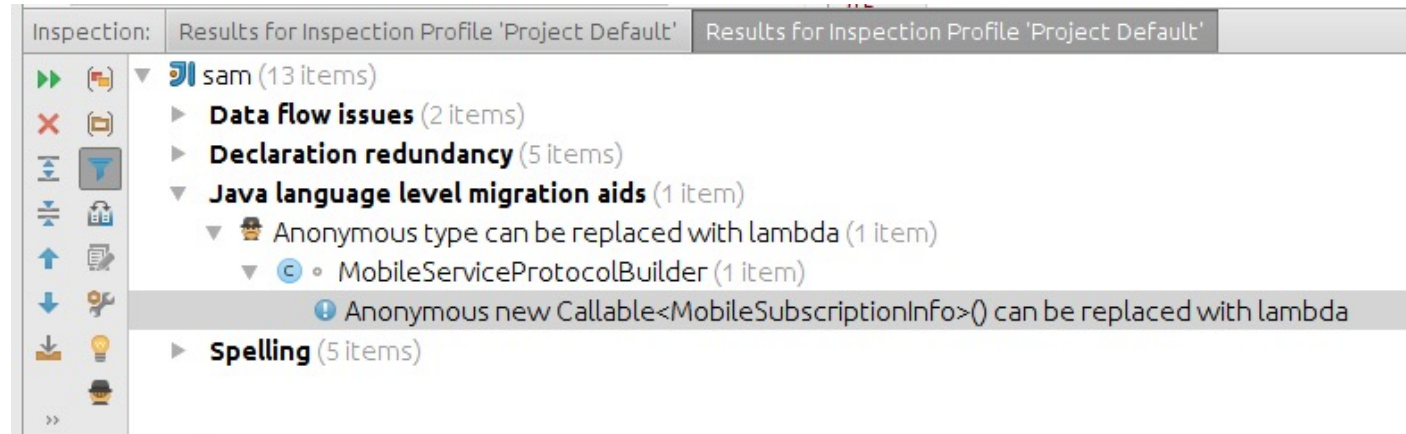


Example: Convert to lambda

```
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



```
List<Person> adults = persons.stream().           // Create  
                        filter(Person::isAdult).    // Intermediate  
                        collect(Collectors.toList()); // Terminal
```

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, 3, 11, 7, 5, 14, 9);  
IntStream values = IntStream.range(0, 100);  
IntStream chars = "This is a test".chars();
```

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

Streams – Intermediate- und Terminal-Operations

Intermediate-Operations

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

```
final Stream<Person> allAdultMikes = persons.stream().  
    filter(Person::isAdult).  
    filter(person -> person.getName().equals("Mike")).  
    filter(mike -> mike.livesIn("Zürich"));
```

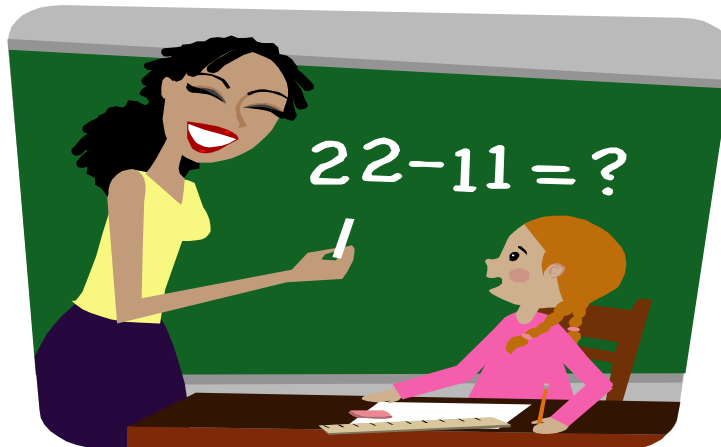
Terminal-Operations

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

```
streamFromValues.filter(n -> n < 9).sorted().forEach(System.out::println); // 3 5 7
```

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));
```



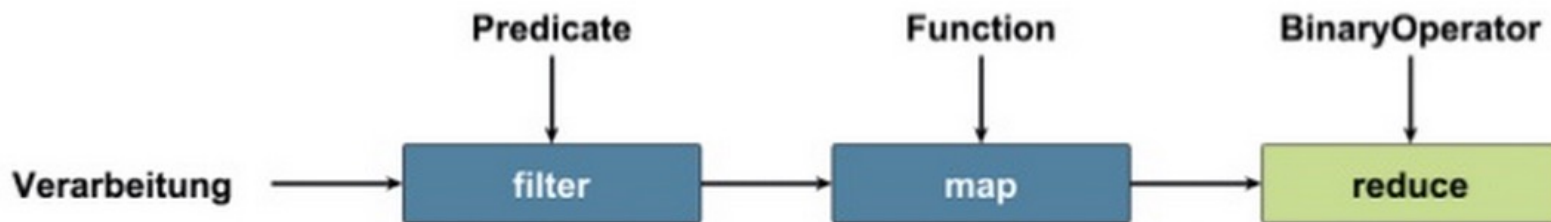
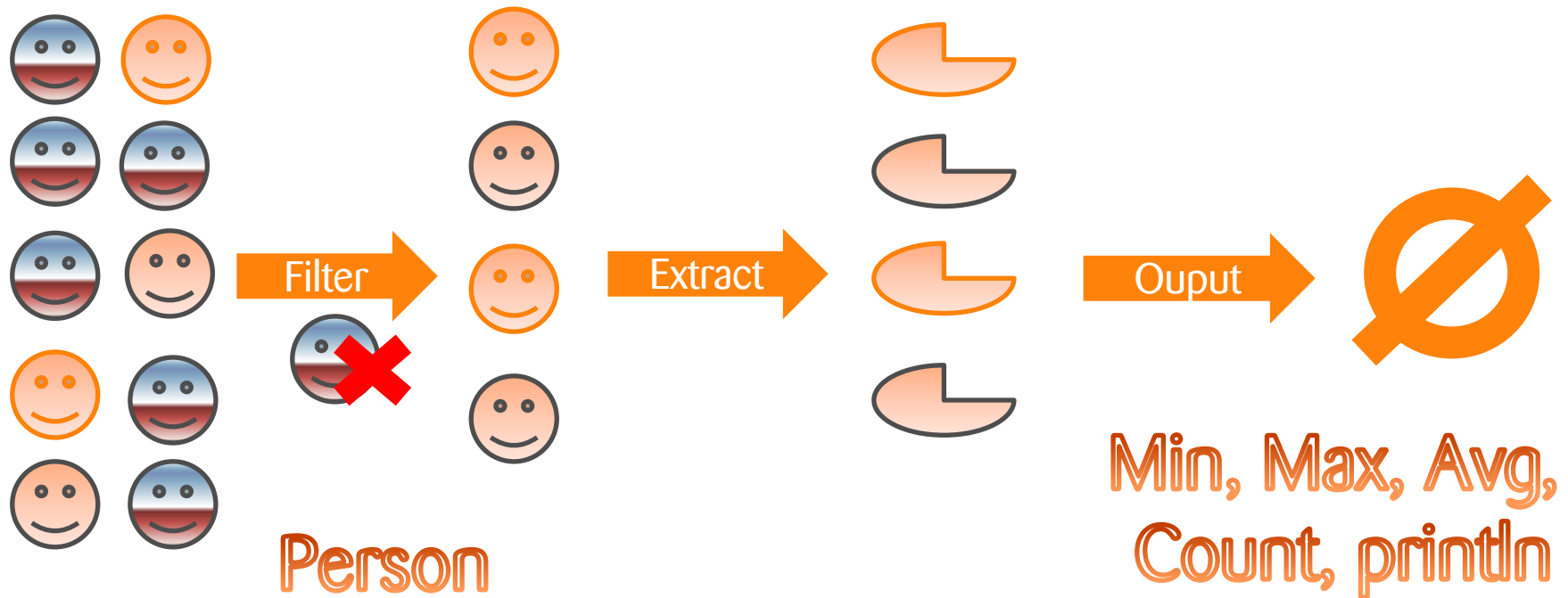
Terminal-Operations – Collectors.joining, groupingBy, partitioningBy

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

=>

(a, 1) (b, 2) (c, 3)

-4711

7777777

«Real World» Example



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

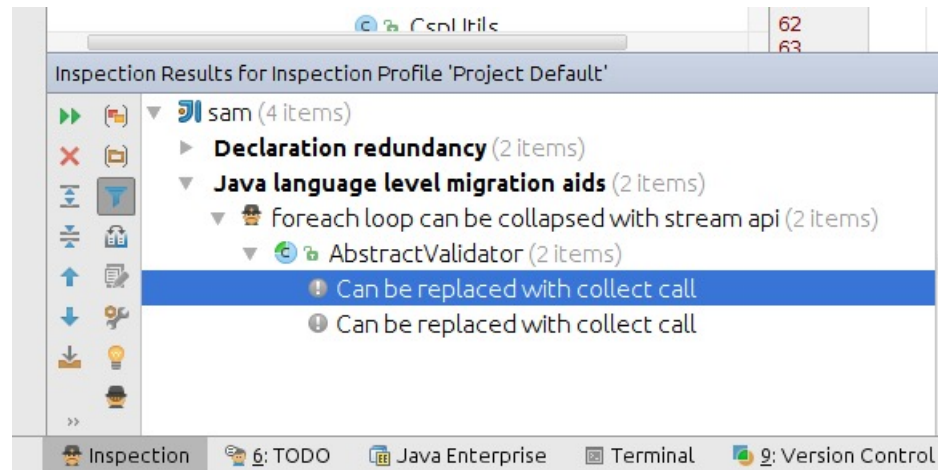
```
@Override
public List<SimpleValidationMessage> getErrorListForId(final int id)
{
    final List<SimpleValidationMessage> list = new ArrayList<>();

    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");
```

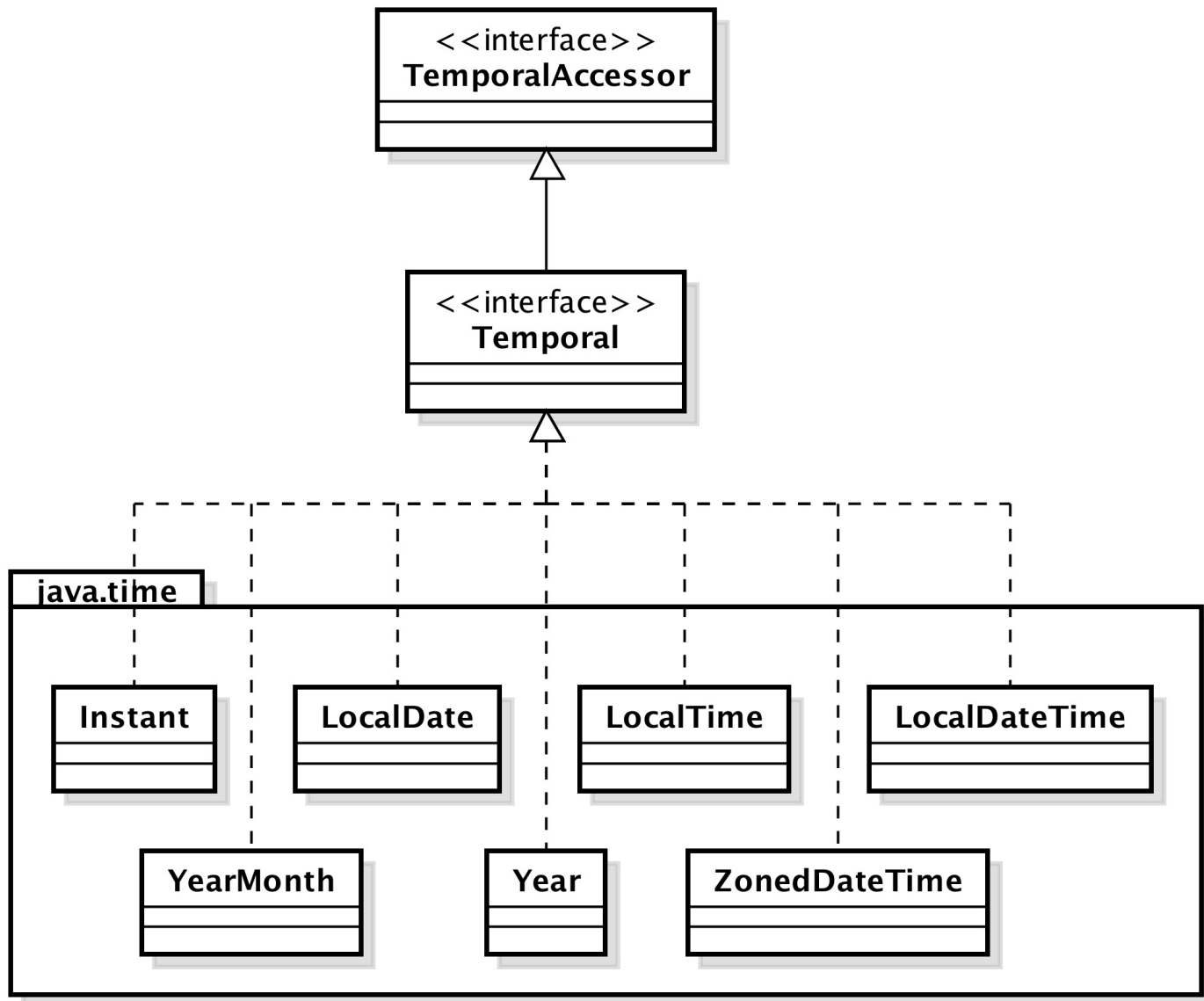
```
LocalTime at_12_11_10 = LocalTime.of(12, 11, 10);
```

// **Kombination aus Datum und Zeit**

```
LocalDateTime nowWithTime = LocalDateTime.now();
```

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

JSR-310: Date And Time API Overview



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 Parsing

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

```
System.out.println("original date: " + date);
```

```
final DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy MM dd");
```

```
final String text = date.format(formatter);
```

```
System.out.println("as text: " + text);
```

```
final LocalDate parsedDate = LocalDate.parse(text, formatter);
```

```
System.out.println("parsed date: " + parsedDate);
```

```
original date: 2017-06-04  
as text: 2017 06 04  
parsed date: 2017-06-04
```

JSR-310: Vordefinierte Formatierungen

```
final LocalDate date = LocalDate.now();  
System.out.println("original date: " + date);
```

```
final DateTimeFormatter formatter1 = DateTimeFormatter.BASIC_ISO_DATE;  
final DateTimeFormatter formatter2 = DateTimeFormatter.ISO_DATE;  
final DateTimeFormatter formatter3 =  
    DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM);
```

```
System.out.println("BASIC_ISO_DATE: " + date.format(formatter1));  
System.out.println("ISO_DATE: " + date.format(formatter2));  
System.out.println("ofLocalizedDate: " + date.format(formatter3));
```

```
original date:      2017-06-04  
BASIC_ISO_DATE:    20170604  
ISO_DATE:          2017-06-04  
ofLocalizedDate:   04.06.2017
```

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));
```

```
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(" / ZoneId " + zdt.getZone());
System.out.println(" / ZoneOffset " + zdt.getOffset());
```

```
formattedDate Datum: 14.07.2016 / Uhrzeit: 05:25
```

```
2007-12-03T10:15:30+01:00[Europe/Paris] as LocalDateTime 2007-12-03T10:15:30
/ ZoneId Europe/Paris / ZoneOffset +01:00
```

Part 5: 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);
```

Comparator<T> Pitfalls – reverse order

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());
```

Comparator – Umgang mit null-Werten

Mitunter muss/möchte man auch null-Werte geeignet einsortieren:

* **nullsFirst()** / **nullsLast()**

```
public static void main(final String[] args)
{
    final List<String> names = Arrays.asList("A", null, "B", "C", null, "D");

    // Null-sichere Komparatoren zur Dekoration bestehender Komparatoren
    final Comparator<String> naturalOrder = Comparator.naturalOrder();
    final Comparator<String> nullsFirst = Comparator.nullsFirst(naturalOrder);
    final Comparator<String> nullsLast = Comparator.nullsLast(naturalOrder);

    names.sort(nullsFirst);
    System.out.println("nullsFirst: " + names);
    names.sort(nullsLast);
    System.out.println("nullsLast: " + names);
}
```

```
nullsFirst: [null, null, A, B, C, D]
nullsLast: [A, B, C, D, null, null]
```


Optional<T>

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

```
public Customer findCustomerByNameOldStyle(final String name)
{
    for (final Customer customer : customers)
    {
        if (customer.getName().equals(name))
        {
            return customer;
        }
    }

    return null;
}
```

Diese Realisierung ist leicht verständlich. Problematisch wird das Ganze, wenn kein passender Datensatz gefunden wurde. Dies wird vielfach wie auch hier über den Wert `null` ausgedrückt.

Optional<T>

Durch Einsatz der Klasse Optional<T> wird das API sofort verständlich und verdeutlicht, dass mitunter kein Treffer geliefert werden kann.

```
public Optional<Customer> findCustomerByNameNewStyle(final String name)
{
    for (final Customer customer : customers)
    {
        if (customer.getName().equals(name))
        {
            return Optional.of(customer);
        }
    }

    return Optional.empty();
}
```

Optional<T>

Zudem ist man als Aufrufer durch das Typsystem dazu gezwungen, erst durch verschiedene Aktionen sicherzustellen, dass ein Wert vorhanden ist.

```
final Optional<Customer> optCustomer = findCustomerByName(name);  
  
if (optCustomer.isPresent())  
{  
    final Customer customer = optCustomer.get();  
  
    doSomethingWithCustomer(customer);  
}  
else  
{  
    handleMissingCustomer(name);  
}
```

Optional<T> -- Behandlung von Alternativen

```
public static void main(final String[] args)
{
    // Minimum für leeren Eingabe-Stream ermitteln
    final OptionalInt min = IntStream.empty().min();

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

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

    // Berechne Ersatzwert, wenn nicht vorhanden
    final IntSupplier randomGenerator = () -> (int)(100 * Math.random());
    System.out.println(min.orElseGet(randomGenerator));

    // Löse eine Exception aus, wenn nicht vorhanden
    min.orElseThrow(() -> new NoSuchElementException("there is no minimum"));
}
```

Optional<T> -- Behandlung von Alternativen

Aus der Praxis kennt man Aufruf wie diesen:

```
final String version = computer.getGraphicscard().getFirmware().getVersion();
```

Abhilfe 1: Horde von null-Prüfungen

```
String version = "UNKNOWN";
if (computer != null)
{
    final Graphicscard graphicscard = computer.getGraphicscard();
    if (graphicscard != null)
    {
        final Firmware firmware = graphicscard.getFirmware();
        if (firmware != null)
        {
            version = firmware.getVersion();
        }
    }
}
```

Optional<T> – Behandlung von Alternativen

```
final String version = computer.getGraphicscard().getFirmware().getVersion();
```

Abhilfe 2: **Alle Methoden umschreiben, sodass sie Optional<T> liefern**

```
final String version = computer.flatMap(Computer::getGraphicscard) .  
                                flatMap(Graphicscard::getFirmware) .  
                                map(Firmware::getVersion).orElse("UNKNOWN");
```

⇒ Lösung ist nicht besonders verständlich

⇒ aufwendig, da ggf. an vielen Stellen und dann auch noch an allen nutzenden Stellen angepasst werden muss

⇒ Mitunter hat man den Code nicht im Zugriff

Abhilfe 3: **Hilfsmethode safeResolve() ⇒ Übung**

Neuerung CompletableFuture<T>

- Erweiterung des `Future<T>`-Interface
- Asynchrone Verarbeitungen beschreiben
- Basisschritte
 - **`supplyAsync()`** => Berechnung definieren
 - **`thenApply()`** => Ergebnis der Berechnung verarbeiten
 - **`thenAccept()`** => Ergebnis verarbeiten, aber ohne Rückgabe
 - **`thenCompose()`** => Verarbeitungsschritte sequenziell verbinden
 - **`thenCombine()`** => Verarbeitungsschritte zusammenführen

CompletableFuture<T> – asynchrone Aktionen

```
CompletableFuture<String> completableFuture1 =  
    CompletableFuture.supplyAsync(() -> "Hello from " +  
                                     Thread.currentThread());  
CompletableFuture<String> completableFuture2 =  
    completableFuture1.thenApply(s -> s + " World " +  
                                     Thread.currentThread());  
CompletableFuture<Void> future =  
    completableFuture1a.thenAccept(System.out::println);
```

```
Hello from Thread[ForkJoinPool.commonPool-worker-9,5,main] World Thread[main,5,main]
```

```
CompletableFuture.supplyAsync(this::calculateContent)  
    .thenApply(this::createNotificationMsg)  
    .thenAccept(this::notifyObservers);
```


CompletableFuture<T> – Aktionen verbinden

```
CompletableFuture<String> completableFuture1 =  
    CompletableFuture.supplyAsync(() -> "Hello from " +  
        Thread.currentThread());  
  
CompletableFuture<String> completableFuture2 =  
    completableFuture1.supplyAsync(() -> " World " +  
        Thread.currentThread());  
  
CompletableFuture<String> combined =  
    completableFuture1.thenCombine(completableFuture2,  
        (s1, s2) -> s1 + s2);  
  
System.out.println(combined.get());
```

```
Hello from Thread[ForkJoinPool.commonPool-worker-9,5,main]  
WorldThread[ForkJoinPool.commonPool-worker-2,5,main]
```

CompletableFuture – Exception Handling

```
CompletableFuture<String> future =  
    CompletableFuture.supplyAsync(() -> "Start");  
future = future.thenApply(str -> {  
    System.out.println("Stage 1: " + str);  
    return "A";  
});  
future = future.thenApply(str -> {  
    System.out.println("Stage 2: " + str);  
    if (true) throw new RuntimeException();  
    return "B";  
});  
future = future.thenApply(str -> {  
    System.out.println("Stage 3: " + str);  
    return "C";  
});  
future.exceptionally(e -> {  
    System.out.println("Exceptionally");  
    return null;  
});
```

Stage 1: Start
Stage 2: A
Exceptionally

Neuerungen in java.util.Arrays

Parallel Operations on Arrays

- `parallelSort()` – sort arrays in parallel using multiple threads
- `parallelSetAll()` – calculate a function for every element of an array
- `parallelPrefix()` – combine the elements of an array in parallel

```
final int[] numbers = { 1, 19, 2, 8, 17, 3, 5, 6, 4, 20 };  
Arrays.parallelSort(numbers); // [1, 2, 3, 4, 5, 6, 8, 17, 19, 20]  
System.out.println(Arrays.toString(numbers));  
Arrays.parallelSetAll(numbers, idx -> idx * 100 );  
System.out.println(Arrays.toString(numbers)); // [0, 100, ..., 900]
```

Neuerungen in java.util.Arrays

parallelPrefix() – combine the elements of an array in parallel

```
final int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
Arrays.parallelPrefix(array, (x, y) -> x * y);  
System.out.println(Arrays.toString(array));  
// [1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

```
final int[] array2 = { 1, 2, 3, 4, 5, 60, 70, 80, 90, 100 };  
  
int startIndex = 5;  
int endIndex = 10;  
  
Arrays.parallelPrefix(array2, startIndex, endIndex, ( x, y ) -> x + y );  
System.out.println(Arrays.toString(array2)); // [1, 2, 3, 4, 5, 60, 130, 210, 300, 400]
```

Neuerungen in java.nio.file.Files

Enhancements NIO

- `lines(Path)` – contents of a file as `Stream<String>`
- `readAllLines(Path)` – reads a file and provides the lines as `List<String>`
- `list(Path)` – directory contents as `Stream<Path>`.
- `ZipFile`

```
final ZipFile zipFile = new ZipFile(file);  
final Stream<? extends ZipEntry> streamOfZipEntries = zipFile.stream();
```

«Real World» Example



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:

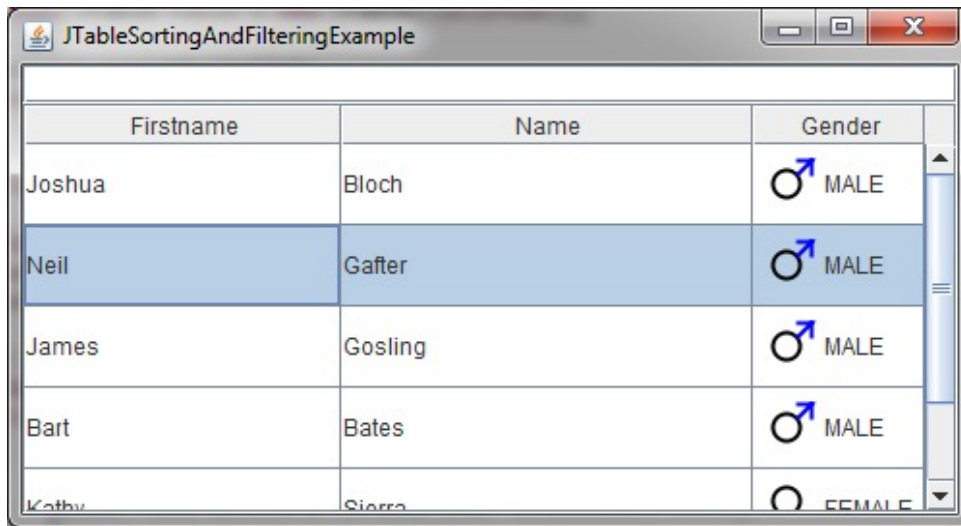
```
Comparator<ShopDbShopDto> byName = comparing(ShopDbShopDto::getName);  
shopListDto.sort(byName);
```


Part 6: JavaFX 8

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

JavaFX 8 – Old Swing vs. New JavaFX

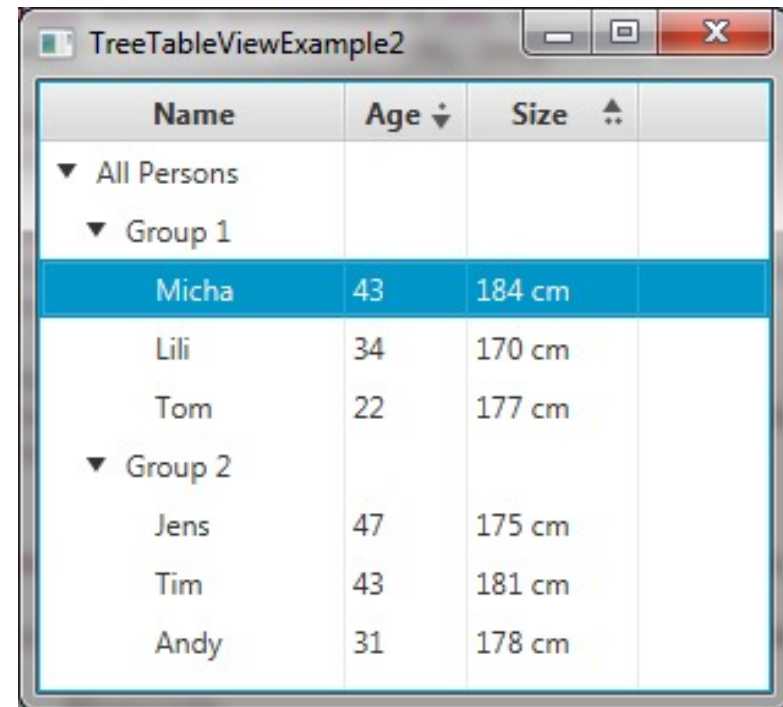
Swing



A screenshot of a Swing window titled "JTableSortingAndFilteringExample". It displays a table with three columns: "Firstname", "Name", and "Gender". The table contains five rows of data. The second row, with "Neil" as the first name and "Gaftar" as the name, is selected and highlighted in blue. The gender column uses male symbols (♂) for males and a female symbol (♀) for females.

Firstname	Name	Gender
Joshua	Bloch	♂ MALE
Neil	Gaftar	♂ MALE
James	Gosling	♂ MALE
Bart	Bates	♂ MALE
Kathy	Sierra	♀ FEMALE

JavaFX

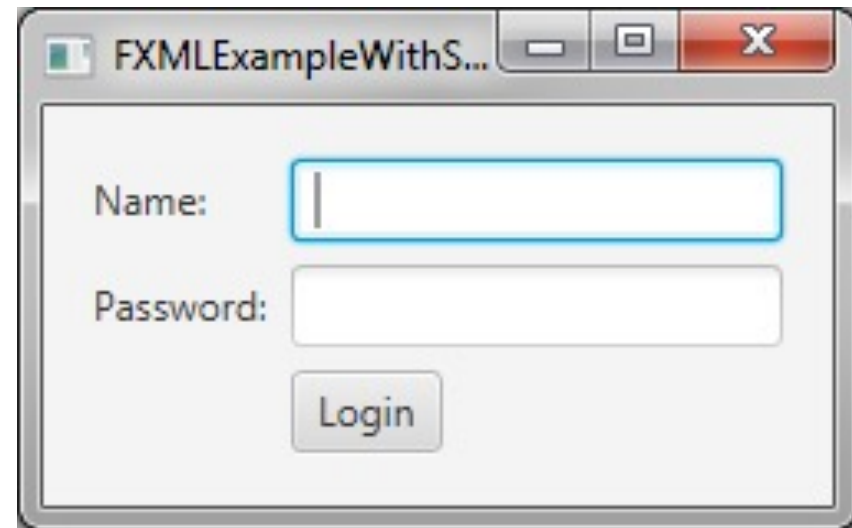
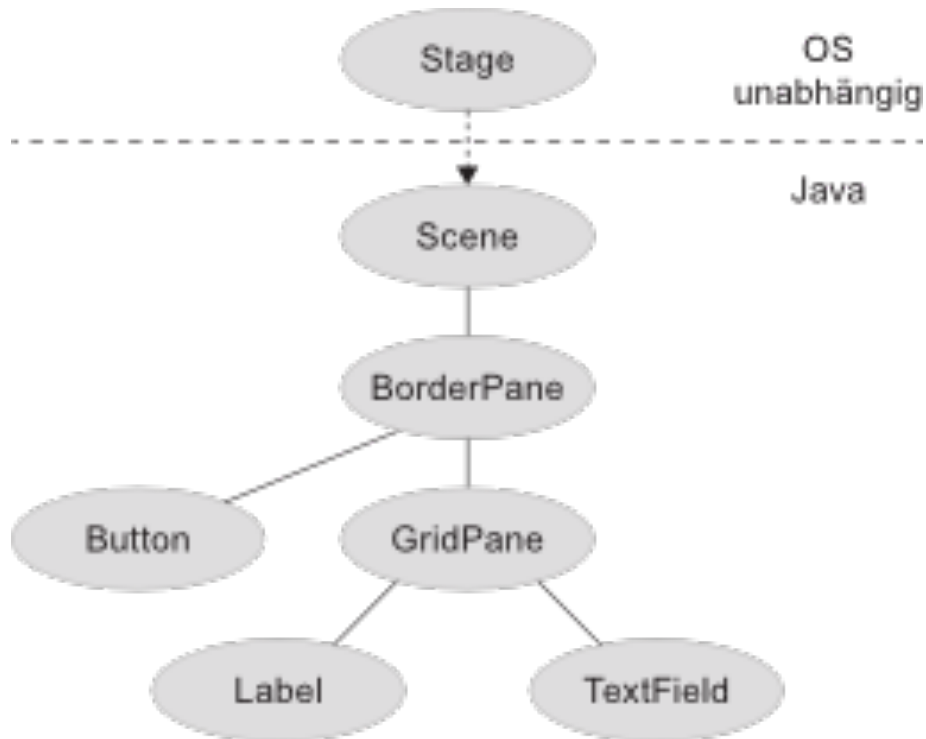


A screenshot of a JavaFX window titled "TreeTableViewExample2". It displays a tree table view with three columns: "Name", "Age", and "Size". The table is organized into a hierarchy. The root node is "All Persons", which has a child node "Group 1". "Group 1" contains three rows: "Micha" (Age 43, Size 184 cm), "Lili" (Age 34, Size 170 cm), and "Tom" (Age 22, Size 177 cm). "Group 1" also has a child node "Group 2", which contains three rows: "Jens" (Age 47, Size 175 cm), "Tim" (Age 43, Size 181 cm), and "Andy" (Age 31, Size 178 cm). The "Micha" row is highlighted in blue.

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

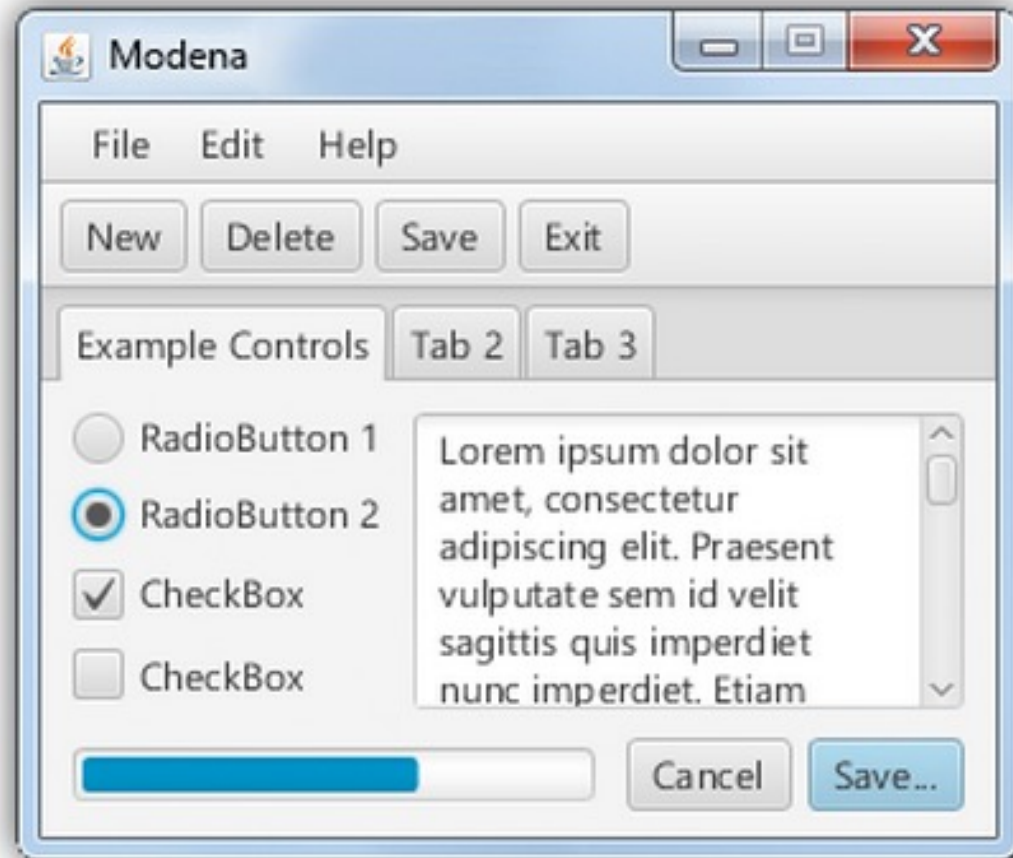


JavaFX 8 – Rich Text und Look And Feel

Rich Text Support



New Look And Feel



JavaFX 8 – Neue Controls

TreeTableView

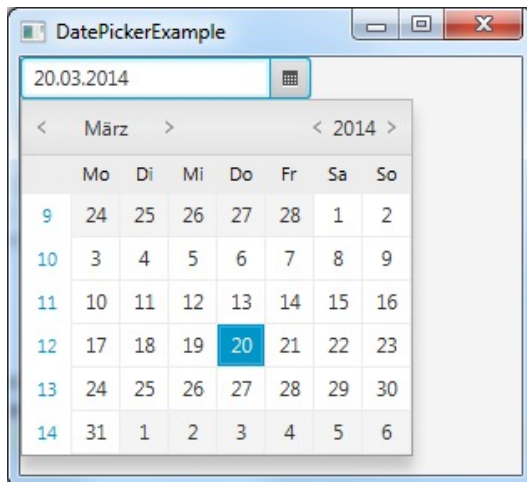
Year ▼	Balance		Comments	+
	Income	Spending		
2001	\$12,000	\$12,000	Years 2001 and 2002 w significant improvemen	▲
2002	\$14,000	\$14,000		
2003	\$6,000	\$8,000	Drop followed by a size	
2004	\$18,000	\$18,000		
2005 ▼	\$19,000	\$19,000	<html>2005 and 2006 b success. <a href="http:/	▼
2006	12000	\$12,000		
2007	\$20,000	\$20,000		

Editor
Viewer

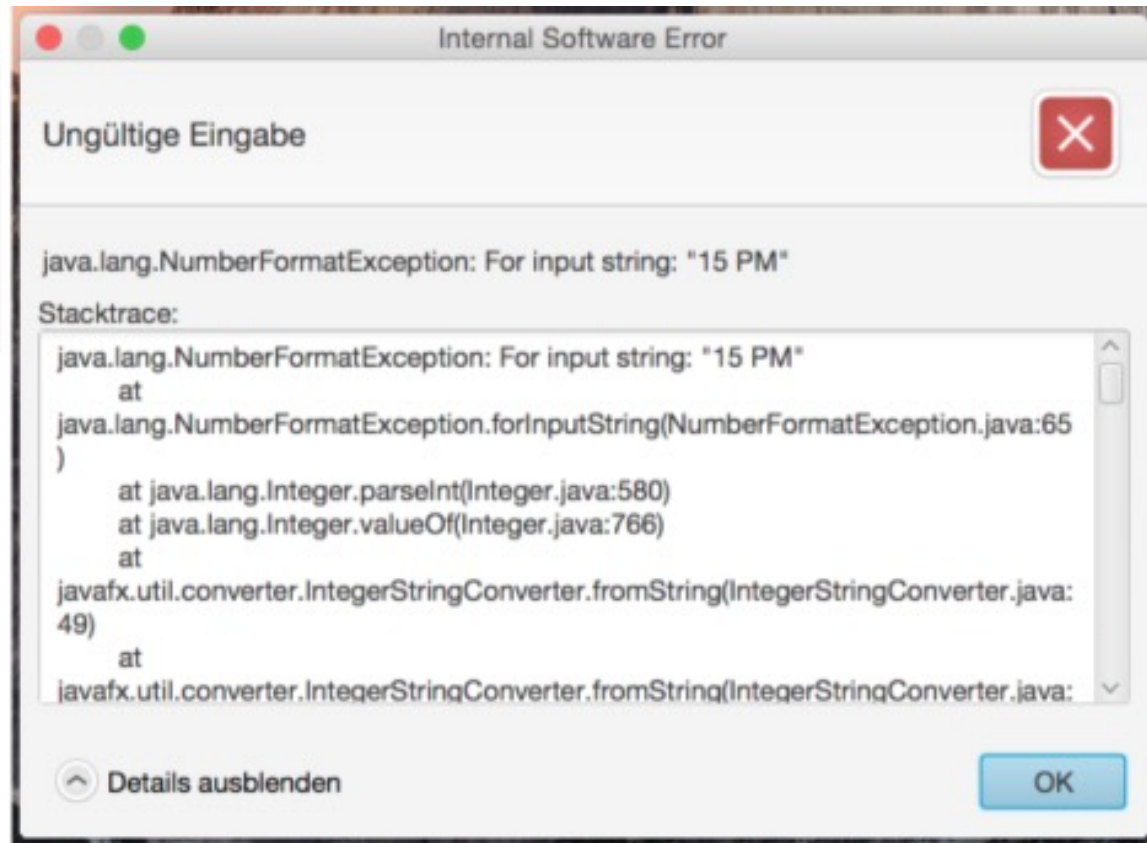
Editor - Viewer

Editor - Viewer

DatePicker

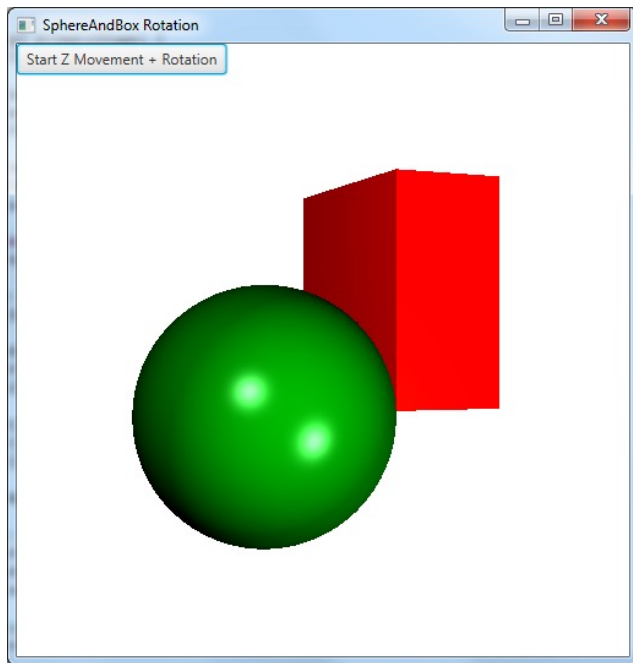


JavaFX 8 – Neue Controls: Standarddialoge



JavaFX 8 – 3D Support

Beispiel



VIDEO:

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





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

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-Terminal

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

The End

Vielen Dank für die Aufmerksamkeit!

Viel Spaß bei der eigenen Entdeckungsreise zu Java 8!