Java 8

Johannes Hellrich, Erik Fäßler

Oberseminar 29.6.2017

Quellen

- https://github.com/winterbe/java8-tutorial
- http://www.lambdafaq.org
- https://github.com/jOOQ

Größte Änderung: Funktionale Programmierung

- Lambdas
- Neue Interfaces
- Streams

Lambdas

- Alter Ansatz: Anonyme Innere Klassen

```
Collections.sort(names,
        new Comparator<String>() {
            @Override
            public int compare(String a, String b) {
                return b.compareTo(a);
            }
        }
);
```

Lambdas

- Neu: Lambdas
- Besser als in Python (Typsicher!)

```
Collections.sort(names,
        (String a, String b) -> {
                return b.compareTo(a);
        }
);
```

Lambdas

- Bei Einzeilern sind Klammern und return optional
- Achtung: “;” entfällt!

```
Collections.sort(names,
        (String a, String b) ->
                b.compareTo(a)
);
```

Lambdas

- Der Typ der Argumente ist für den Compiler zumeist aus dem Kontext klar und optional

```
Collections.sort(names,
        (a, b) -> b.compareTo(a)
);
```

Lambdas: Scope

- Innerhalb des Lambdas sind lokale Variablen der Umgebung sichtbar, die sich wie eine final Variable verhalten
- Analog zu Anonymen Inneren Klassen, bei denen das final aber explizit stehen musste
- Voller Zugriff auf statische und Instanzfelder

Lambdas Beispiel: JeSemE

- API des Spark Webservers für HTTP Requests

```
get("/api/similarity",
        (request, response) ->
                getAssociation(request, db,
                DatabaseService.SIMILARITY_TABLE,
                false),
        new Gson()::toJson);
```

Referenzen auf Methoden/Konstruktoren

- "::" ermöglicht Referenzen auf Methoden und Konstruktoren
- Erfordert passendes functional Interface

```
ResponseTransformer rt = new Gson()::toJson;
PersonFactory<Person> pf = Person::new;
```

Functional Interfaces

```
package spark;

@FunctionalInterface
public interface ResponseTransformer {

    /**
     * Method called for rendering the output.
     *
     * @param model object used to render output.
     * @return message that it is sent to client.
     * @throws java.lang.Exception when render fails
     */
    String render(Object model) throws Exception;
}
```

Functional Interfaces

- Praktisch um Factories zu implementieren

```
interface PersonFactory<P extends Person> {
        P create(String firstName, String lastName);
}
```

- Haben genau eine (abstrakte) Methode
- Optionale Annotation @FunctionalInterface
- Grundalge für das Funktionieren von Lambdas!

Außerdem bei Interfaces: Default Methods

```
interface Formula {
        double calculate(int a);

        default double sqrt(int a) {
                return Math.sqrt(a);
        }
}
```

Achtung: Nicht in Lambdas des entsprechenden Interfaces sichtbar ☹

Mehrfachvererbung bei Default Methods

- Mehrfachvererbung := Klasse hat 2 Interfaces mit Defaultmethode gleichen Namens
- Ein Interface ist Unterklasse des anderen
  -> Implementierung der Unterklasse wird genommen
- Interfaces unabhängig
  -> Klasse muss Methode überschreiben
  (Kann INTERFACE.super.NAME für Zugriff nutzen)

Wie nutze ich das?

Funktionales Programmieren
mit Java 8

Eingebaute Funktionale Interfaces

- Predicate<X> X -> boolean
- Function<X,Y> X -> Y
- Supplier<X> () -> X
- Consumer<X> X -> ()
- Comparator<X> x1, x2 -> -1, 0, 1

Streams

- Datenströme im Zentrum
- Komponenten modifizieren diese
- Bekannte Idee aus der UNIX Kommandozeile oder NLP Pipelines

```
echo "Viel zu heiß" | sed "s/.*h//" | tr "ß" "s"
```

Streams

- Wie bekomme ich einen Stream?

  Collection.stream()

  oder

  Stream.of(...)

- Streams sind typsicher, unterstützen Objekte über Generics und auch Primitive über extra Klassen

Filter

- filter(Predicate)

```
Stream.of(2,1,3).filter(x -> x > 1)
```

- Lässt nur Daten durch, die zu einem true führen

Sorted

- sorted()

```
Stream.of(2,1,3).sorted()
```

- Bringt Daten in natürliche Ordnung
- Optionales Argument: Comparator

Map

- Map(Function)

```
Stream.of(2,1,3).map(x -> x.toString())
```

- Wendet Funktion auf jedes Element des Datenstroms an
- Kann den Datentyp ändern

flatMap

- flatMap(Function<X, Stream>)
- Entpackt komplexe Objekte in Streams, die dann kombiniert werden

```
Stream.of(Arrays.asList(1,2), Arrays.asList(3,4))
        .flatMap(x ->x.stream())
```

Terminale Operationen

- any/all/noneMatch(Predicate): Testet Elemente des Datenstroms, ergibt Boolean
- count(): Anzahl der Elemente im Strom
- forEach(Consumer): Wendet Consumer auf jedes Element an

```
Stream.of(1,2,3).forEach(x -> println(x))
```

Terminale Operationen

- collect(Collector) Wandelt Stream in Collection um, breite Auswahl in Collectors

```
Stream.of(1,2,3).collect(Collectors.toList())
```

- reduce(Function): Benutzt Function um Elemente des Streams zu kombinieren, ergibt ein Optional

```
Optional o = Stream.of(1,2,3).reduce((x,y) -> x+y)
```

Optional

```
Optional<String> optional = Optional.of("bam");
optional.isPresent(); // true
optional.get(); // "bam"
optional.orElse("fallback"); // "bam"
optional.ifPresent((s) ->
        System.out.println(s.charAt(0))); // "b"
```

Parallelität

- Collection.parallelStream() führt zu Streams die automatisch parallelisiert werden
- Kein Problem für Operationen die Elemente transformieren
- Terminale Operationen haben Varianten die sich um Ordnung/Kombination kümmern

Tricks zu Streams

- Endlich Zahlen Ranges in Java:
  IntStream.range(0, 20)
- Stream zu Iterator: Stream.iterator()
- Stream zu Iterable:

```
for (String s : (Iterable<String>)stream::iterator)
```

NIO

Files.lines(Paths.get(fileName)).map(…)

- Paths.get(„teil“, „teil2“,...) moderne Art auf Dateien zuzugreifen
- Files.lines(aPath) zeilenweiser Stream
- Files.list(aPathToDir) Dateien in Ordner als Stream
- Files.write(...) Schreibt Iterable in Datei

Weitere Neuerungen

- Neue Date API
- Annotationen
- Änderungen am Multithreading

Java 8

Johannes Hellrich, Erik Fäßler

Oberseminar 29.6.2017

Ein großes Beispiel

```
String[] mostSimilar =
        executor.invokeAll(Arrays.asList(
                getMostSimilarAsync(true, db, corpus, word),
                getMostSimilarAsync(false, db, corpus, word))
        ).stream().
        map(
                future -> {
                    try { return future.get(); }
                    catch (final Exception e) {throw new IllegalStateException(e); }
                }).
        flatMap(List::stream).
        distinct().
        toArray(String[]::new);
```

Noch ein großes Beispiel

```
Function<Integer, String> fName =
        num  -> fields.get(num).get(JulieXMLConstants.NAME);
row.put(fName.apply(0), results.docId);
row.put(fName.apply(1), results.data);
log.trace("{}={}", fName.apply(0), results.docId);
log.trace("{}={}", fName.apply(1), results.data);
if (results.getClass().equals(DocumentXmiData.class)) {
        DocumentXmiData docResults = (DocumentXmiData) results;
        row.put(fName.apply(2), docResults.newXmiId);
        row.put(fName.apply(3), docResults.serializedSofaXmiIdMap);
        log.trace("{}={}", fName.apply(2), docResults.newXmiId);
        log.trace("{}={}", fName.apply(3), docResults.serializedSofaXmiIdMap);
} else {
        // for subsets we want to store if there is data at all
        row.put(fName.apply(2), results.data != null);
        log.trace("{}={}", fName.apply(2), results.data != null);
}
```

Scala

- JVM Sprache
- Entwickelt an der EPFL von Martin Odersky
- Funktional & Objektorientiert
- Vorlage für viele Featues in Java 8 (Streams, Interfaces mit Methoden)
- Sehr kluger aber langsamer Kompiler

Was Java noch aus Scala übernehmen könnte

- Blöcke/Verzweigungen die automatisch zurückgeben

```
Integer i = { if(true)
                    19
              else
                    23
              }
```

Was Java noch aus Scala übernehmen könnte

- Pattern matching

```
x = expr match {
        case ("porridge", _) ⇒ "eating”
        case ("chair", "Mama") ⇒ "sitting"
        case ("bed", "Baby") ⇒ "sleeping”
        case _ ⇒ "what?”
}
```

Was Java noch aus Scala übernehmen könnte

- Implizite Parameter

```
class KoanIntWrapper(val original: Int) {
        def isOdd = original % 2 != 0
}

implicit def foo (value: Int) =
        new KoanIntWrapper(value)

19.isOdd == true
```

Was Java noch aus Scala übernehmen könnte

- Generics mit besserer Berücksichtigung von Über-/Untertypen
- Echte Monaden statt Streams