MATHEMA

Lambdas, Collections und Streams

Michael Wiedeking

Java Forum Stuttgart

16. Juli 2014

Agenda

- Lambdas – in aller Kürze
- Streams – ein kurze Einführung
- Parallele Streams
- Spliterator (wenn die Zeit reicht)

Lambdas –
in aller Kürze

```
interface Function<T, R> {
    public R apply(T t);
}

public void printPerson(
    Person person,
    Function<Person, String> fun
) {
    print(fun.apply(person));
}
```

```
class MyFunction implements Function<Person, String> {

    private final String prefix;

    public MyFunction(String prefix) {
        this.prefix = prefix;
    }

    public String apply(Person p) {
        return prefix + p.getName();
    }
}

printPerson(person, new MyFunction("Name: "));
```

```
final String prefix = “Name: ”;

printPerson(person, new Function<Person, String> {
        public String apply(Person p) {
                return prefix + p.getName();
        }
});
```

```
final String prefix = “Name: ”;

printPerson(person, (Person p) -> {
        return prefix + p.getName();
});
```

```
final String prefix = “Name: ”;

printPerson(person, p -> prefix + p.getName());
```

```
class MyFunction implements Function<Person, String> {
    private final String prefix;
    public MyFunction(String prefix) {
        this.prefix = prefix;
    }
    public String apply(Person p) {
        return prefix + p.getName();
    }
}

mf = new MyFunction("Name: ");

printPerson(person, mf::apply);

printPerson(person, Object::toString);
```

Streams –
eine kurze Einführung

- Eine „einmalige“ Sicht auf eine Collection
  - forEach
  - map
  - filter
  - reduce

```
Collection collection = …
collection.stream()
        .map(…)
        .filter(…)
        .reduce(…)
        .forEach(…)
```

- void forEach(Consumer<? **super** T> *consumer*)

```
persons.stream()
        .forEach(person -> println(person.getName()))

strings.stream()        // Hat keinen Effekt!
        .forEach(s -> s.toLowerCase())

sizes.stream()
        .forEach(System.out::println)
```

- void forEachOrdered(Consumer<? **super** T> *consumer*)

- boolean allMatch(Predicate<? **super** T> *predicate*)
- boolean anyMatch(Predicate<? **super** T> *predicate*)
- boolean noneMatch(Predicate<? **super** T> *predicate*)

```
<R> Stream<R> map(
        Function<? super T,? extends R> mapper
)
```

```
persons.stream()
        .map(person -> person.getName())

strings.stream()
        .map(s -> s.length())

sizes.stream()
        .map(i -> new String[i])
```

- DoubleStream mapToDouble(
  ToDoubleFunction<? **super** T> *mapper*
)
- IntStream mapToInt(
  ToIntFunction<? **super** T> *mapper*
)
- LongStream mapToLong(
  ToLongFunction<? **super** T> *mapper*
)

```
<R> Stream<R> flatMap(
        Function<? super T,? extends Stream<? extends R>> m
)
```

```
persons.stream()
        .flatMap(person -> person.getTimeReports().stream())
```

- DoubleStream flatMapToDouble(
  Function<? **super** T,? **extends** DoubleStream> *mapper*
)
- IntStream flatMapToInt(
  Function<? **super** T,? **extends** IntStream> *mapper*
)
- LongStream flatMapToLong(
  Function<? **super** T,? **extends** LongStream> *mapper*
)

- Stream<T> filter(Predicate<? **super** T> *predicate*)

```
persons.stream()
       .filter(person -> person.getSalary() >= threshold)

strings.stream()
       .filter(s -> s.endsWith(“.java”))

sizes.stream()
       .filter(i -> (i % 2) == 0)
```

- Optional<T> reduce(BinaryOperator<T> *accumulator*)

```
strings.stream()
       .reduce(String::append)

sizes.stream()
       .reduce((x, y) -> x + y)
```

- Optional<T> reduce(BinaryOperator<T> *accumulator*)

```
// Funktioniert leider nicht!
persons.stream()
    .reduce((p1, p2) -> p1.getSalary() + p2.getSalary())
```

- Optional<T> reduce(BinaryOperator<T> *accumulator*)

```
persons.stream()
       .map(person -> person.getSalary())
      .reduce((salary1, salary2) -> salary1 + salary2)
```

```
<U> U reduce(
        U identity,
        BiFunction<U,? super T,U> accumulator,
        BinaryOperator<U> combiner
)


persons.stream()
        .reduce(
                0,
                (sum, person) -> sum + person.getSalary(),
                (s, t) -> s + t
        )
```

- Optional<T> reduce(BinaryOperator<T> *accumulator*)
- T reduce(T *identity*, BinaryOperator<T> *accumulator*)

- long count()
- Optional<T> max(Comparator<? **super** T> *comparator*)
- Optional<T> min(Comparator<? **super** T> *comparator*)

- <R> R collect(Collector<? **super** T,R> *collector*)

```
interface Collector<T, R> {
    BiFunction<R,T,R> accumulator()
    default Set<Collector.Characteristics> characteristics()
    BinaryOperator<R> combiner()
    Supplier<R> resultSupplier()
}
```

- **static** <T, K, D>
  Collector<T, Map<K, D>> groupingBy(
  Function<? **super** T, ? **extends** K> *classifier*,
  Collector<T, D> *downstream*
)
- **static** <T, C **extends** Collection<T>>
  Collector<T, C> toCollection(
  Supplier<C> *collectionFactory*
)
- **static** <T>
  Collector<T, DoubleSummaryStatistics> toDoubleSummaryStatistics(
  ToDoubleFunction<? **super** T> *mapper*
)

```
maxSalary = persons.stream()
        .map(person -> person.getSalary())
        .filter(salary -> salary > 1000)
        .max();

maxSalary = Streams.intRange(0, 100000)
        .filter(i -> i % 3 == 0)
        .map(i -> Integer.valueOf(i))
        .map(i -> Numbers.toHexNumber(i, 4)
        .filter(s -> !s.contains("123"))
        .filter(s -> !containsTwins(s))
        .map(s -> s.reverse())
        .map(s -> Long.valueOf(s))
        .mapToLong(l -> l.longValue())
```

Parallele Streams

```
collection.stream().parallel()
        .alles()
        .andere()
        .wie()
        .bisher()
```

Spliterator

- boolean hasCharacteristics(int characteristics)
  - CONCURRENT
  - DISTINCT
  - IMMUTABLE
  - NONNULL
  - ORDERED
  - SIZED
  - SUBSIZED
  - SORTED
- int characteristics()

- long estimateSize()
- boolean tryAdvance(Consumer<? **super** T> c)
- Spliterator<T> trySplit()

- **default** void forEachRemaining(Consumer<? **super** T> c)
- **default** Comparator<? **super** T> getComparator()

- **default** boolean hasCharacteristics(int characteristics)

```
class TaggedArray<T> {
    private final Object[] data

    public TaggedArray(T[] values, Object[] tags) {
        int n = values.length;
        Object[] data = new Object[2 * n];
        for (int i = 0, j = 0; i < n; i++) {
            data[j++] = values[i];
            data[j++] = tags[i];
        }
        this.data = data;
    }

    public Spliterator<T> spliterator() {
        return new TaggedSpliterator<>(data, 0, data.length);
    }
}
```

```
class TaggedSpliterator<T> implements Spliterator<T> {

    private final Object[] array;

    private int offset;

    private final int limit;

    TaggedSpliterator(Object[] array, int offset, int limit) {
        this.array = array;
        this.offset = offset;
        this.limit = limit;
    }

    …
}
```

```
public void forEachRemaining(Consumer<? super T> c) {
    for (; offset < limit; offset += 2) {
        c.accept((T) array[offset]);
    }
}

public boolean tryAdvance(Consumer<? super T> c) {
    if (offset < limit) {
        c.accept((T) array[offset]);
        offset += 2;
        return true;
    } else {
        return false;
    }
}
```

```
public Spliterator<T> trySplit() {
    int low = offset;
    int mid = evenMedian(low, limit);
    if (low < mid) {
        offset = mid;
        return new TaggedSpliterator<>(array, low, mid);
    } else {
        return null;
    }
```

```
public long estimateSize() {
        return ((limit - offset) / 2);
}
public int characteristics() {
        return ORDERED | SIZED | IMMUTABLE | SUBSIZED;
}
```

```
class ParEach<T> extends CountedCompleter<Void> {
    final Spliterator<T> spliterator;
    final Consumer<T> action;
    final long targetBatchSize;
    ParEach(
        ParEach<T> parent,
        Spliterator<T> spliterator,
        Consumer<T> action,
        long targetBatchSize
    ) {
        super(parent);
        this.spliterator = spliterator;
        this.action = action;
        this.targetBatchSize = targetBatchSize;
    }
```

```
public void compute() {
    Spliterator<T> sub;

    while (
        spliterator.estimateSize() > targetBatchSize
        &&
        (sub = spliterator.trySplit()) != null
    ) {
        addToPendingCount(1);
        new ParEach<>(this, sub, action, targetBatchSize)
            .fork();
    }
    spliterator.forEachRemaining(action);
    propagateCompletion();
}
```

```
static <T> void parEach(
        TaggedArray<T> a, Consumer<T> action
) {
        Spliterator<T> s = a.spliterator();
        int p = ForkJoinPool.getCommonPoolParallelism();
        long targetBatchSize = s.estimateSize() / (p * 8);
        new ParEach(null, s, action, targetBatchSize)
                .invoke();
}
```

Demo

Fragen?

Vielen Dank!

MATHEMA Software GmbH
Henkestraße 91
91052 Erlangen
www.mathema.de
info@mathema.de