2.– 5. September 2013
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Lambdas II

Funktionale Erweiterungen der Java-Collections

Klaus Kreft

Java 8

Bulk Data Operations for Collections

Klaus Kreft

http://www.AngelikaLanger.com/

agenda

- introduction + basics
- categorization of stream operations
- parallel streams

formal introduction

- bulk data operations JEP 107
  http://openjdk.java.net/jeps/107
  - closely related to JSR 335:
    Lambda Expressions for the Java Programming Language

- coming with Java 8 (March 2014)
  http://openjdk.java.net/projects/jdk8/milestones

bulk data operations for collections in Java 8

- extension of the JDK collections

- with …
  - functional view: sequence + operations
  - object-oriented view: collection + internal iteration
  - for-each/filter/map/reduce for Java

  - for-each
    apply a certain functionality to each element of the sequence

```
accounts.forEach(a -> a.addInterest());
```

(cont.)

- filter

  build a new sequence that is the result of a filter applied to each element in the original collection

```
accounts.filter(a -> a.balance() > 1_000_000 );
```

- map

  build a new sequence, where each element is the result of a mapping from an element of the original sequence

```
accounts.map(a -> a.balance());
```

- reduce

  produce a single result from all elements of the sequence

```
accounts.map(a -> a.balance())
        .reduce(0, (b1, b2) -> b1+b2);
```

streams

- interface `java.util.stream.Stream<T>`
  - supports `forEach`, `filter`, `map`, `reduce`, and more

- two new methods in `java.util.Collection<T>`
  - `Stream<T> stream()`, sequential functionality
  - `Stream<T> parallelStream()`, parallel functionality

```
List<Account> accountCol = … ;

Stream<Account> accounts = accountCol.stream();

Stream<Account> moreThanAMillion =
                accounts.filter(a -> a.balance() > 1_000_000);
```

more about streams and their operations

- streams do not store their elements
  - not a collection, but created from a collection, array, …
  - view/adaptor of a data source (collection, array, …)

- streams provide functional operations

  `forEach`, `filter`, `map`, `reduce`, …

  which are applied to the elements of the underlying data source

(cont.)

- actually applied functionality is two-folded
  - user-defined: functionality passed as parameter
  - framework method: stream operations

- separation between "*what* to do" & "*how* to do"
  - user => *what* functionality to apply
  - library => *how* to apply functionality
    (parallel/sequential, lazy/eager, out-of-order)

```
accounts.filter(a -> a.balance() > 1_000_000);

accounts.forEach(a -> a.addInterest());
```

parameters of stream operations …

… can be

- lambda expressions
- method references
- (inner classes)

streams operations and …

… their corresponding parameter functional interfaces
(= signature of the user-defined functionality)

- for-each

```
void forEach(Consumer<? super T> consumer);
```

```
public interface Consumer<T> {
   public void accept(T t);
}
```

```
accounts.forEach(a -> a.addInterest());
```

```
accounts.forEach(a -> { a.addInterest(); });
```

```
accounts.forEach(Account::addInterest);
```

`Stream.map()` - possible

- `balance` is of type `double`

```
public interface Stream<T> ... {
     ...
  <R> Stream<R> map(Function<? super T, ? extends R> mapper);
     ...
}
```

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

```
Stream<Double> balances = accounts.map(a -> a.balance());
```

```
Stream<Double> balances =
                accounts.map(a ->  { return a.balance(); });
```

`Stream.mapToDouble()` - preferred

- `balance` is of type `double`

```
public interface Stream<T> ... {
     ...
  DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper);
     ...
}
```

```
public interface ToDoubleFunction<T> {
   public double applyAsDouble(T t);
}
```

```
DoubleStream balances = accounts.mapToDouble(a -> a.balance());
```

```
DoubleStream balances =
             accounts.mapToDouble(a -> { return a.balance(); });
```

primitive streams

- streams for elements with primitive type:

  `IntStream, LongStream, DoubleStream`

- reason: performance

- no stream types for `char` and `float`
  - use stream type of respective ‘bigger’ primitive type
    - `IntStream` for `char`, and `DoubleStream` for `float`

how to obtain a stream ?

- `java.util.Collection<T>`
  - `Stream<T> stream()`, sequential functionality
  - `Stream<T> parallelStream()`, parallel functionality

- `java.util.Arrays`
  - `static <T> Stream<T> stream(T[] array)`
  - plus overloaded versions (primitive types, …)

- and in some more places …
  - see javadoc of package `java.util.stream`

(cont.)

- collections allow to obtain a parallel stream directly
  - in all other cases use stream’s method: `parallel()`

```
Account[] accArray = …;
Arrays.stream(accArray).parallel().forEach(Account::addInterest);
```

agenda

- introduction
- categorization of stream operations
  - intermediate / terminal
  - (intermediate) stateless / stateful
  - short-circuiting
- parallel streams

intermediate / terminal

- already seen that there are …

- stream operations that produce a stream again:
  `filter()`, `map()`, …
  - intermediate (lazy)

- stream operations that do something else:
  `forEach()`, `reduce()`, …
  - terminal (eager)

```
double sum = accountCol.stream()
                       .mapToDouble(a -> a.balance())
                       .reduce(0, (b1, b2) -> b1+b2);
```

example

```
String[] txt = { "State", "of", "the", "Lambda",
                                  "Libraries", "Edition"};

IntStream is = Arrays.stream(txt).filter(s -> s.length() > 3)
                                   .mapToInt(s -> s.length());

int sum = is.reduce(0, (l1, l2) -> l1 + l2);
```

- `filter()` and `mapToInt()` return streams
  - intermediate
- `reduce()` returns `int`
  - terminal
- intermediate stream not evaluated

  until a terminal operation gets invoked

(intermediate) stream

```
IntStream is = Arrays.stream(txt).filter(s -> s.length() > 3)
                                 .mapToInt(s -> s.length());
```

- view/adaptor of a data source (collection, …)

+

- intermediate stream operations
  - their order
  - their parameters
  - …

reason: performance

- code optimization
- no buffering of intermediate stream results
- easier to handle parallel streams

terminal operations == consuming operations

- terminal operations are also consuming operations

```
IntStream is = Arrays.stream(txt).filter(s -> s.length() > 3)
                                 .mapToInt(s -> s.length());

is.forEach(l -> System.out.print(l +", "));
System.out.println();

int sum = is.reduce(0, (l1, l2) -> l1 + l2);
```

```
5, 6, 9, 7,
Exception Exception in thread "main" java.lang.IllegalStateException:
                                       stream has already been operated upon or closed
          at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:226)
          at java.util.stream.IntPipeline.reduce(IntPipeline.java:466)
          at Main.test(Main.java:224)
          at Main.main(Main.java:249)
```

- why? – less confusing (side effects ?!?)

advice: fluent programming

- best avoid reference variables to stream objects
- instead:
  - construct the stream
  - apply a sequence of intermediate stream operations
  - terminate with an terminal stream operation
  - one statement
  - *fluent programming*
    - build the next operation on top of the result of the previous one

```
int sum = Arrays.stream(txt).filter(s -> s.length() > 3)
                            .mapToInt(s -> s.length())
                            .reduce(0, (l1, l2) -> l1 + l2);
```

agenda

- introduction
- categorization of stream operations
  - intermediate / terminal
  - (intermediate) stateless / stateful
  - short-circuiting
- stream operations' javadoc
- advanced stream operations

stateless intermediate operations

- `filter()`, `map()`, …
- operation needs only the stream element … … to decide what to do
- e.g. `filter()`
  - predicate applied to the element evaluates to
    - `true` – element goes to the next stage
    - `false` – element gets dropped
- easy to handle

stateful intermediate operations

- `Stream<T> limit(long maxSize)`
  `Stream<T> substream(long start)`
  `Stream<T> substream(long start, long end)`
  `Stream<T> distinct()`
  `Stream<T> sorted(Comparator<? super T> c)`
  `Stream<T> sorted()`
- operation needs additional state (previous elements)
  + stream element … to decide what to do
- e.g. `distinct()`
  – element goes to the next stage, if it hasn’t already appeared before (according to `equals()`)
- not so easy to handle, especially for parallel streams

(cont.)

- `sorted()`, only operation that
  - uses buffering extensively
  - uses only one thread for the operation (with parallel stream)
- for sequential streams
  - all other operations similar to stateless operations
  - but with additional state
- for parallel streams
  - `distinct()`
    - buffering after the operation, details later
  - `limit()` / `substream()`
    - (try to) adjusts the splitterator
    - but buffering in certain situations possible

agenda

- introduction
- categorization of stream operations
  - intermediate / terminal
  - (intermediate) stateless / stateful
  - short-circuiting
- stream operations' javadoc
- advanced stream operations

short-circuiting operations ...

… (might) stop the processing
before the last element is reached

- intermediate

```
Stream<T> limit(long maxSize)
Stream<T> substream(long start, long end)
```

- terminal

```
boolean anyMatch(Predicate<? super T> predicate)
boolean allMatch(Predicate<? super T> predicate)
boolean noneMatch(Predicate<? super T> predicate)
Optional<T> findFirst()
Optional<T> findAny()
```

agenda

- introduction
- categorization of stream operations
- parallel streams

example

- what it does:
  - `String` array of stock symbols
  - get stock information from Yahoo finance, and convert it into a `StockData` object (`symbol`, `price`, `change`)
  - find the one with the maximum increase, and
  - use it

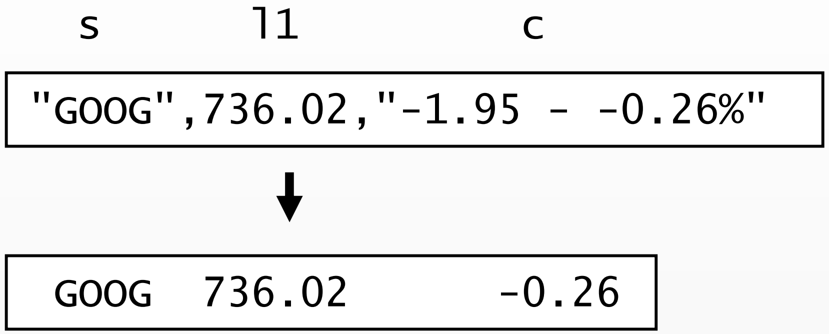
```
String[] stockSymbols = {"GOOG", "AAPL", "MSFT", "YHOO"};

Arrays.stream(stockSymbols)
   .map(s -> createStockData(getStockInfo(s)))
   .filter(s -> s != null)
   .reduce((s1,s2) -> s1.getChange() > s2.getChange() ? s1 : s2)
   .ifPresent(s -> System.out.println(s));
```

getStockInfo()

```
private static String getStockInfo(String s) {
  try {
    URL yahoofinance =
       new URL("http://finance.yahoo.com/d/quotes.csv?s="+ s +"&f=sl1c");

    URLConnection yc = yahoofinance.openConnection();

    try (BufferedReader in =
         new BufferedReader(new InputStreamReader(yc.getInputStream()))) {

      String result = in.readLine();
      return result;
    }
  } catch (Exception e) { return null; }
}
```

```
  s      l1        c

"GOOG",736.02,"-1.95 - -0.26%"
```

createStockData()



```
s       l1           c
"GOOG",736.02,"-1.95 - -0.26%"

GOOG  736.02       -0.26
```

```
private static StockData createStockData(String data) {
  try {
    System.out.println(Thread.currentThread().getName() + ": “ + data);

    String[] result = data.split(",", 0);

    return new StockData(
           result[0].substring(1,result[0].length()-1),
           result[1],
           Float.parseFloat(result[2]
                 .substring(result[2].length()-7, result[2].length()-2)));
  } catch (Exception e) { return null; }
}
```

without parallel()

```
String[] stockSymbols={"GOOG", "AAPL", "MSFT", "YHOO"};

Arrays.stream(stockSymbols)
   .map(s -> createStockData(getStockInfo(s)))
   .filter(s -> s != null)
   .reduce((s1,s2) -> s1.getChange() > s2.getChange() ? s1 : s2)
   .ifPresent(s -> System.out.println(s));
```

```
main: "GOOG",736.02,"-1.95 - -0.26%"
main: "AAPL",522.06,"-4.94 - -0.94%"
main: "MSFT",26.76,"+0.02 - +0.07%"
main: "YHOO",19.35,"-0.51 - -2.57%"
MSFT - 26.76 - 0.07
```

with parallel()

```
String[] stockSymbols={"GOOG", "AAPL", "MSFT", "YHOO"};

Arrays.stream(stockSymbols)
   .parallel()
   .map(s -> createStockData(getStockInfo(s)))
   .filter(s -> s != null)
   .reduce((s1,s2) -> s1.getChange() > s2.getChange() ? s1 : s2)
   .ifPresent(s -> System.out.println(s));
```

```
main: "MSFT",26.76,"+0.02 - +0.07%"
ForkJoinPool.commonPool-worker-1: "AAPL",522.06,"-4.94 - -0.94%"
main: "YHOO",19.35,"-0.51 - -2.57%"
ForkJoinPool.commonPool-worker-1: "GOOG",736.02,"-1.95 - -0.26%"
MSFT - 26.76 - 0.07
```

advantages

- stock info retrieved in parallel
  - multiple threads to handle parallel HTTP requests
  - eliminate the accumulated latency of the sequential solution

- minimal implementation effort
  - compared to user-implemented solution:
    - HTTP request = task implemented as `Callable`
    - submitted to a `ThreadPoolExecutor`
    - retrieve stock info strings via `Future`’s `get()` method

- ‘maximum change’ is computed in parallel
  - utilizes all (virtual) CPU cores of the platform
  - relevant when using a large number of stock symbols

(cont.)

- *"central element of this feature"* (from the JEP 107 description)
  - easy implementation of parallel solutions that use all CPU cores

- most obvious:

```
.parallel().forEach(...)
```

  - is the parallel for-loop in Java

but …

… not really a ‘good’ example

- instructive
  - and okay if run as the only parallel operation
- problem:
  - synchronous I/O ‘freezes’ the pool threads
  - they have to wait, until synch. I/O is through
  - other parallel operations have to wait, too
  - parallel operations are meant to be CPU bound !!!

another example

- to show implementation details

```
final int SIZE = 64;
int[] ints = new int[SIZE];
ThreadLocalRandom rand = ThreadLocalRandom.current();
for (int i=0; i<SIZE; i++) ints[i] = rand.nextInt();

Arrays.stream(ints).parallel()
      .reduce((i,j) -> Math.max(i,j))
      .ifPresent(System.out.println(m -> “max is: ” + m));
```

- find (in parallel) the largest element in an `int` array
  - could be implemented shorter: `max()`
    - but this illustrates better: have to do a comparison with each element

- `parallelStream()`’s functionality is based on the fork-join framework

fork join tasks

- original task is divided into two sub-tasks by splitting the stream source into two parts
  - original task's result are based on sub-tasks' results
  - sub-tasks are divided again … fork phase
- at a certain depth partitioning stops
  - tasks at this level (leaf tasks) are executed
  - execution phase
- completed sub-task results are 'combined' to super-task results
  - join phase

parallel streams + intermediate operations

- what if the stream contains
  upstream intermediate operations

```
        ....parallel().filter(...)
                      .mapToInt(...)
                      .reduce((i,j) -> Math.max(i,j));
```

when/where are these applied to the stream ?

parallel streams + state

- examples:
  - stateful intermediate operations
  - collect stream elements into a non-thread-safe collection
  - …

- concurrency and state == not easy
  - important aspect: performance

example

- concatenate in parallel
  the string representation of numbers 0 … 31

```
StringBuffer sb = new StringBuffer();
IntStream.range(0,32).parallel()
         .mapToObj(Integer::toString)
         .forEach(sb::add);
```

  - locking is good, prevents concurrent access ☺
  - locking is not good, ‘freezes’ pool threads ☹
  - order ? ☹

- approach: *reduce instead of accumulate*
  - `collect()` with `StringBuilder`, ie. no lock
  - `Collector` returned from `Collectors.joining()`

(cont.)

```
System.out.println(
   IntStream.range(0,32)
            .parallel()
            .mapToObj(Integer::toString)
            .collect(Collectors.joining());
```

```
012345678910111213141516171819202122232425262728293031
```

collect() **and** Collector **(for** joining()**)**

```
<R,A> R collect(Collector<? super T, A, R> collector);
```

```
Supplier<A> supplier();
                    // StringBuilder::new

BiConsumer<A,T> accumulator();
                    // (a, t) -> { a.append(t); }

BinaryOperator<A> combiner();
                    // (a1, a2) -> { a1.append(a2); return a1; }

Function<A,R> finisher()
                   // StringBuilder::toString

Set<Characteristics> characteristics();
                   // not: CONCURRENT, UNORDERED, IDENTITY_FINISH
```

more …

… but not here and now

- order
- complex stream operations:
  - `collect()` with `groupingBy()`
  - …
- a new type: `java.util.Optional<T>`
- …

wrap up: streams

- new abstraction: `java.util.stream.Stream`
  - provides the bulk operations
  - specializations for primitive types: `int`, `long`, `double`

- categories of operations:
  - produce a stream `->` intermediate / lazy
  - produce something else `->` terminal / eager + consuming

- while the code reads: each operation for all elements

  streams are process in the order …
  … all operations for each element

wrap up: parallel streams

- allow to utilize all (virtual) CPU cores of the platform

- based on fork-join framework
  - splits underlying stream source into chunks (= tasks)
  - applies stream operations sequentially at each chunk/task
  - runs chunks/tasks in parallel with multiple CPU-cores

- easy to use with stateless stream operations
  + stateless lambdas
  - performance improvement over sequential operations

- otherwise … more complex approach
  - performance ?

authors

Angelika Langer

Klaus Kreft

http://www.AngelikaLanger.com

Bulk Operations

Q & A