ORACLE

JDK 8: Stream Style

Sergey Kuksenko
sergey.kuksenko@oracle.com, @kuksenk0

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Motivation

Motivation

Everything is well. Why do we need any Streams?

Motivation

Everything is well. Why do we need any Streams?

```
public void printGroups(List<People> people) {
    Set<Group> groups = new HashSet<>();
    for (Person p : people) {
        if (p.getAge() >= 65)
            groups.add(p.getGroup());
    }
    List<Group> sorted = new ArrayList<>(groups);
    Collections.sort(sorted, new Comparator<Group>() {
        public int compare(Group a, Group b) {
            return Integer.compare(a.getSize(), b.getSize())
        }
    });
    for (Group g : sorted)
        System.out.println(g.getName());
}
```

Motivation

It would be awesome to omit miles and miles of duplicated code.

```
public void printGroups(List<People> people) {
    people.stream()
     .filter(p -> p.getAge() > 65)
     .map(p -> p.getGroup())
     .distinct()
     .sorted(comparing(g -> g.getSize()))
     .map(g -> g.getName())
     .forEach(n -> System.out.println(n));
}
```

Motivation

It would be awesome to do less work, and do it later (laziness).

```
public void printGroups(List<People> people) {
    people.stream()
     .filter(p -> p.getAge() > 65)
     .map(p -> p.getGroup())
     .distinct()
     .sorted(comparing(g -> g.getSize()))
     .map(g -> g.getName())
     .forEach(n -> System.out.println(n));//⇐ACTIONS!!!
}
```

Motivation

Parallelism?

```
Collection<Item> data;
 ...

for (int i = 0; i < data.size(); i++) {
    processItem(data.get(i));
}
```

Motivation

Parallelism?

```
Collection<Item> data;
 ...

for (Item item : data) {
    processItem(item);
}
```

Motivation

Parallelism?

```
Collection<Item> data;
 ...

#pragma ~~omg~~omp parallel
for (Item item : data) {
    processItem(item);
}
```

Motivation

Parallelism?

```
Collection<Item> data;
 ...

#pragma omp parallel
for (Item item : data) {
    processItem(item);
}
```

Motivation

Parallelism?

```
Collection<Item> data;
  ...

parallel_for (Item item : data) {
    processItem(item);
}
```

Motivation

Parallelism!

```
Collection<Item> data;
 ...

data.parallelStream()
    .forEach(item -> processItem(item));
```

Design

Design

- Most of the code fits the same simple pattern:

$$source$$

Design

- Most of the code fits the same simple pattern:

$$source \rightarrow op$$

Design

- Most of the code fits the same simple pattern:

$$source \rightarrow op \rightarrow op$$

Design

- Most of the code fits the same simple pattern:

$$source \rightarrow op \rightarrow op \rightarrow op$$

Design

- Most of the code fits the same simple pattern:

$$source \rightarrow op \rightarrow op \rightarrow op \rightarrow$$

Design

- Most of the code fits the same simple pattern:

$$source \rightarrow op \rightarrow op \rightarrow op \rightarrow gangnamstyle$$

Design

- Most of the code fits the same simple pattern:

$$source \rightarrow op \rightarrow op \rightarrow op \rightarrow sink$$

Design

- Most of the code fits the same simple pattern:

$$source \rightarrow op \rightarrow op \rightarrow op \rightarrow sink$$

- «sources»: collections, iterators, channels, ...
- «operations»: filter, map, reduce, ...
- «sinks»: collections, locals, ...

Sources

- Standard classes?
  - not-yet-created classes?
  - 3^{rd} party classes?

Sources

- Standard classes?
  - not-yet-created classes?
  - 3^{rd} party classes?
- `Collection`?
  - should we put everything into collection?

Sources

- Standard classes?
  - not-yet-created classes?
  - 3^{rd} party classes?
- `Collection`?
  - should we put everything into collection?
- `Iterable`?
  - "Iterator Hell" (inherently sequential)
  - interface pollution

Sources

- Standard classes?
  - not-yet-created classes?
  - 3^{rd} party classes?
- `Collection`?
  - should we put everything into collection?
- `Iterable`?
  - "Iterator Hell" (inherently sequential)
  - interface pollution
- `Stream`!
  - new (just invented) class with required semantic
  - inject the only `stream()` method into existing classes

Stream

Stream

- "A multiplicity of values"
- May be unordered
- Not a collection (no storage)
- Operations are deferred as long as possible
- May be infinite
- Source is unmodifiable
- Can be used only once
- Sequential or parallel
- Primitive specializations: `IntStream`, `LongStream`, `DoubleStream`

Stream pipeline

a source: Source → Stream

intermediate operations: Stream → Stream

a terminal operation: Stream → PROFIT!

```
public void printGroups(List<People> people) {
  people.stream()
    .filter(p -> p.getAge() > 65)
    .map(p -> p.getGroup())
    .distinct()
    .sorted(comparing(g -> g.getSize()))
    .map(g -> g.getName())
    .forEach(n -> System.out.println(n));
}
```

Stream pipeline

a source: Source → Stream

intermediate operations: Stream → Stream

a terminal operation: Stream → PROFIT!

```
public void printGroups(List<People> people) {
  people.stream()
    .filter(p -> p.getAge() > 65)
    .map(p -> p.getGroup())
    .distinct()
    .sorted(comparing(g -> g.getSize()))
    .map(g -> g.getName())
    .forEach(n -> System.out.println(n));
}
```

Stream pipeline

a source: Source → Stream

intermediate operations: Stream → Stream

a terminal operation: Stream → PROFIT!

```
public void printGroups(List<People> people) {
  people.stream()
    .filter(p -> p.getAge() > 65)
    .map(p -> p.getGroup())
    .distinct()
    .sorted(comparing(g -> g.getSize()))
    .map(g -> g.getName())
    .forEach(n -> System.out.println(n));
}
```

Stream pipeline

a source: Source → Stream

intermediate operations: Stream → Stream

a terminal operation: Stream → PROFIT!

```
public void printGroups(List<People> people) {
  Stream<People> s1=people.stream();
  Stream<People> s2=s1.filter(p -> p.getAge() > 65);
  Stream<Group>  s3=s2.map(p -> p.getGroup());
  Stream<Group>  s4=s3.distinct();
  Stream<Group>  s5=s4.sorted(comparing(g->g.getSize()));
  Stream<String> s6=s5.map(g -> g.getName());
  s6.forEach(n -> System.out.println(n));
}
```

Stream Sources

Stream Sources: collections

```
ArrayList<T> list;
Stream<T> s = list.stream();      // sized, ordered
```

Stream Sources: collections

```
ArrayList<T> list;
Stream<T> s = list.stream();      // sized, ordered

HashSet<T> set;
Stream<T> s = set.stream();       // sized, distinct
```

Stream Sources: collections

```
ArrayList<T> list;
Stream<T> s = list.stream();      // sized, ordered

HashSet<T> set;
Stream<T> s = set.stream();       // sized, distinct

TreeSet<T> set;
Stream<T> s = set.stream();       // sized, distinct,
                                  // ordered, sorted
```

Stream Sources: factories, builders

```
T[] arr;
Stream<T> s = Arrays.stream(arr);
```

Stream Sources: factories, builders

```
T[] arr;
Stream<T> s = Arrays.stream(arr);
```

```
Stream<T> s = Stream.of(v0, v1, v2);
```

Stream Sources: factories, builders

```
T[] arr;
Stream<T> s = Arrays.stream(arr);
```

```
Stream<T> s = Stream.of(v0, v1, v2);
```

```
Stream<T> s = Stream.builder().add(v0).add(v1).build();
```

Stream Sources: factories, builders

```
T[] arr;
Stream<T> s = Arrays.stream(arr);
```

```
Stream<T> s = Stream.of(v0, v1, v2);
```

```
Stream<T> s = Stream.builder().add(v0).add(v1).build();
```

```
IntStream s = IntStream.range(0, 100);
```

Stream Sources: generators

```
AtomicInteger count = new AtomicInteger(0);
Stream<Integer> s =
    Stream.generate(count::incrementAndGet);
```

Stream Sources: generators

```
AtomicInteger count = new AtomicInteger(0);
Stream<Integer> s =
    Stream.generate(count::incrementAndGet);
```

```
IntStream s = Stream.iterate(0, i -> i+1);
```

Stream Sources: others

```
Stream<String> s = bufferedReader.lines();
```

Stream Sources: others

```
Stream<String> s = bufferedReader.lines();
```

```
Stream<String> s = Pattern.compile(myRegEx)
                          .splitAsStream(myStr);
```

Stream Sources: others

```
Stream<String> s = bufferedReader.lines();
```

```
Stream<String> s = Pattern.compile(myRegEx)
                          .splitAsStream(myStr);
```

```
DoubleStream s = new SplittableRandom().doubles();
```

Intermediate Operations

Intermediate Operations

```
Stream<S> s;
Stream<S> s.filter(Predicate<S>);
Stream<T> s.map(Function<S, T>);
Stream<T> s.flatMap(Function<S, Stream<T>>);
Stream<S> s.peek(Consumer<S>);
Stream<S> s.sorted();
Stream<S> s.distinct();
Stream<S> s.unordered();
Stream<S> s.limit(long);
Stream<S> s.substream(long);
Stream<S> s.substream(long, long);
```

Intermediate Operations

```
Stream<S> s;
Stream<S> s.filter(Predicate<S>);
Stream<T> s.map(Function<S, T>);
Stream<T> s.flatMap(Function<S, Stream<T>>);
Stream<S> s.peek(Consumer<S>);
Stream<S> s.sorted();
Stream<S> s.distinct();
Stream<S> s.unordered();
Stream<S> s.limit(long);
Stream<S> s.substream(long);
Stream<S> s.substream(long, long);

Stream<S> s.parallel();
Stream<S> s.sequential();
```

Terminal Operations a.k.a. PROFIT

Terminal Operations

- Terminal operations yield final result
- Parallel or sequential execution

- Terminal operations 'flavors':
  - iteration: `forEach`, `iterator`
  - searching: `findFirst`, `findAny`
  - matching: `allMatch`, `anyMatch`, `noneMatch`
  - aggregation:
    - *reducers*
    - *collectors*

Short-circuting

- Do not consume the entire stream, drop it on the floor as necessary
- May operate infinite streams

- e.g.: `find*`, `*Match`

```
int v = Stream.iterate(1, i -> i+1)
              .filter( i % 2 == 0)
              .findFirst().get();
```

Iteration

- Process each stream element:

```
IntStream.range(0, 100)
         .forEach(System.out::println);
```

- Convert to old style iterator[1]:

```
Iterator<Integer> =
      IntStream.range(0, 100).iterator();
```

[1] for compatibility

Example

How to get sum of `Stream<Integer>`?

Example

How to get sum of Stream<Integer>?

```
public int getSum(Stream<Integer> s){
    int sum;
    s.forEach( i -> sum += i );
    return sum;
}
```

Example

How to get sum of Stream<Integer>?

```
public int getSum(Stream<Integer> s){
    int sum;
    s.forEach( i -> sum += i ); // Compile error
    return sum;
}
```

Example

How to get sum of Stream<Integer>?

```
public int getSum(Stream<Integer> s){
    int[] sum = new int[1];
    s.forEach( i -> sum[0] += i );
    return sum[0];
}
```

Example

Result?

```
getSum(IntStream.range(0, 100).map(i -> 1))
```

Example

Result?

```
getSum(IntStream.range(0, 100).map(i -> 1))
```

100

Example

Result?

```
getSum(IntStream.range(0, 100).map(i -> 1))
```

100

```
getSum(IntStream.range(0, 100).map(i -> 1).parallel())
```

Example

Result?

```
getSum(IntStream.range(0, 100).map(i -> 1))
```

100

```
getSum(IntStream.range(0, 100).map(i -> 1).parallel())
```

79, 63, 100, ...

Reduction

- Take a stream and make a scalar value:

```
int s = stream.reduce(0, (x, y) -> x + y);
```

Reduction

- Take a stream and make a scalar value:

```
int s = stream.reduce(0, (x, y) -> x + y);
```

- Some operations return `Optional<T>`:

```
Optional<Integer> o =
            stream.reduce((x, y) -> x + y);
Integer i = stream.reduce(0, (x, y) -> x + y);
```

Reduction

```
Stream<T> {

...
<U> U reduce(U identity,
             BiFunction<U,T,U> accumulator,
             BinaryOperator<U> combiner)
...

}
```

Collectors

- A.k.a. mutable reduction operations
- Accumulate elements into a mutable result container:

```
List<Integer> list = IntStream.range(0, 100)
                         .boxed()
                         .collect(Collectors.toList());
int[] ints = IntStream.range(0, 100).toArray();
```

Collectors

- A.k.a. mutable reduction operations
- Accumulate elements into a mutable result container:

```
List<Integer> list = IntStream.range(0, 100)
                     .boxed()
                     .collect(Collectors.toList());
int[] ints = IntStream.range(0, 100).toArray();
```

- Complex collections:

```
Map<Integer,Integer> map = IntStream.range(0, 1000)
    .boxed().collect(
          Collectors.toConcurrentMap(
            k -> k % 42,  v -> v, (a, b) -> b
          )
    );
```

Collectors

```
String[] a = new String[]{"a", "b", "c"};
```

How to get "a, b, c" ?

Collectors

```
String[] a = new String[]{"a", "b", "c"};
```

How to get "a, b, c" ?

```
Arrays.stream(a).collect(Collectors.joining(",␣"));
```

Collectors

```
String[] a = new String[]{"a", "b", "c"};
```

How to get "a, b, c" ?

```
Arrays.stream(a).collect(Collectors.joining(",␣"));
```

FYI: java.util.StringJoiner

Collectors

```
Stream<T> {

...
<R> R collect(Supplier<R> supplier,
              BiConsumer<R,T> accumulator,
              BiConsumer<R,R> combiner)
...

}
```

Collectors

```
Stream<T> s;
List<T> l = s.collect(Collectors.toList());
```

```
l = collect( () -> new ArrayList<>(),
             (list, t) -> list.add(t),
             (l1, l2) -> l1.addAll(l2));
```

Parallelism

Parallelism

- Lots of sources are naturally splittable
- Lots of operations are well parallelizable
- Streams will do it for us
- «`ForkJoinPool` inside»
- Have to ask for the parallelism explicitly

```
int v = list.parallelStream()
             .reduce(Math::max)
             .get();
```

Explicit parallelism

Q: Why not implicit?

A: Final speedup depends on:

- N – number of source elements
- Q – cost of operation
- P – available HW parallelism
- C – number of concurrent clients

We know N.
We can estimate P.
We can somehow cope with C.
Q is almost not predictable.

Splitterator

Splitterator

```
interface Splitterator<T> {
    ...

    long estimateSize();


    boolean tryAdvance(Consumer<T> action);


    Spliterator<T> trySplit();

    ...
}
```

Useful links

- JDK8: http://openjdk.java.net/projects/jdk8/
- Binary builds: https://jdk8.java.net/download.html
- Mailing list: lambda-dev@openjdk.java.net

Thanks!

Q & A ?