



Java8: 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 (People 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 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 → *op* → *op*

Design

- Most of the code fits the same simple pattern

source → *op* → *op* → *op*

Design

- Most of the code fits the same simple pattern

source → *op* → *op* → *op* →

Design

- Most of the code fits the same simple pattern

source → *op* → *op* → *op* → *gangnamstyle*

Design

- Most of the code fits the same simple pattern

source → *op* → *op* → *op* → *sink*

Design

- Most of the code fits the same simple pattern

source → *op* → *op* → *op* → *sink*

- “sources”: collections, iterators, channels, ...
- “operations”: filter, map, reduce, ...
- “sinks”: collections, locals, ...

Sources

- Standard classes?
 - not-yet-created classes?
 - 3rd party classes?

Sources

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

Sources

- Standard classes?
 - not-yet-created classes?
 - 3rd party classes?
- Collection?
 - should we put everything into collection?
- Iterable?
 - “Iterator Hell” (inherently sequential)
 - interface pollution

Sources

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

Stream

Stream

- “A multiplicity of values”
- Not a collection (no storage)
- Operations are deferred as long as possible
- May be infinite
- Source is unmodifiable
- Can be used only once
- Ordered/Unordered
- Parallel/Sequential
- Primitive specializations:
 IntStream, LongStream, DoubleStream

Stream pipeline

a source: Source \rightarrow Stream

intermediate operations: Stream \rightarrow Stream

a terminal operation: Stream \rightarrow 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 \rightarrow Stream

intermediate operations: Stream \rightarrow Stream

a terminal operation: Stream \rightarrow 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 \rightarrow Stream

intermediate operations: Stream \rightarrow Stream

a terminal operation: Stream \rightarrow 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 \rightarrow Stream

intermediate operations: Stream \rightarrow Stream

a terminal operation: Stream \rightarrow 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  
// sorted, ordered
```

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).add(v2)  
                    .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).add(v2)  
                    .build();
```

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


Stream Sources: generators

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

Stream Sources: generators

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

```
Stream<Integer> 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.limit(long);  
Stream<S> s.skip(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.limit(long);  
Stream<S> s.skip(long);  
  
Stream<S> s.unordered();  
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*:
 - *reduction*
 - *collectors*

Short-circuiting

- Do not consume the entire stream, drop it on the floor as necessary
- May operate infinite streams
- `find*`, `*Match`, `limit`

e.g.:

```
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 iterator¹:

```
Iterator<Integer> =
    Stream.iterate(0, i -> i + 1)
           .limit(100)
           .iterator();
```

¹for compatibility

Example

How to compute a sum over `Stream<Integer>`?

Example

How to compute a sum over `Stream<Integer>`?

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

Example

How to compute a sum over Stream<Integer>?

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

Example

How to compute a sum over Stream<Integer>?

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


Example

Result?

```
Stream<Integer> s = IntStream.range(0, 100)
                             .mapToObj(i -> 1);
System.out.println(getSum(s));
```

Example

Result?

```
Stream<Integer> s = IntStream.range(0, 100)
                             .mapToObj(i -> 1);
System.out.println(getSum(s));
```

100

Example

Result?

```
Stream<Integer> s = IntStream.range(0, 100)
                             .mapToObj(i -> 1);
System.out.println(getSum(s));
```

100

```
Stream<Integer> s = IntStream.range(0, 100)
                             .mapToObj(i -> 1)
                             .parallel();
System.out.println(getSum(s));
```

Example

Result?

```
Stream<Integer> s = IntStream.range(0, 100)
                             .mapToObj(i -> 1);
System.out.println(getSum(s));
```

100

```
Stream<Integer> s = IntStream.range(0, 100)
                             .mapToObj(i -> 1)
                             .parallel();
System.out.println(getSum(s));
```

79, 63, 100, ...

Reduction

- Take a stream and make a scalar value:

```
Stream<Integer> s;  
Integer sum = s.reduce(0, (x, y) -> x + y);
```

Reduction

- Take a stream and make a scalar value:

```
Stream<Integer> s;  
Integer sum = s.reduce(0, (x, y) -> x + y);
```

- Some operations return Optional<T>:

```
Stream<Integer> s;  
Optional<Integer> sum = s.reduce((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, 100)
    .boxed().collect(
        Collectors.toConcurrentMap(
            k -> k % 42, v -> v, (a, b) -> b
        )
    );
```

java.util.stream.Collectors

More than 30 predefined collectors, e.g.:

collector => *result of Stream<T>.collect(collector)*

`toList()` => `List`

`toSet()` => `Set`

`toCollection(Supplier<Collection<T>>)` => `Collection<T>`

`partitioningBy(Predicate<T>)` => `Map<Boolean, List<T>>`

`groupingBy(Function<T, K>)` => `Map<K, List<T>>>`

`toMap(Function<T, K>,
 Function<T, U>)` => `Map<K, U>`

Collectors

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

Hot 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"};
```

Hot 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));
```

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));
```



```
l = collect( ArrayList::new, List::add, List::addAll);
```


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.



Thank you!



Q & A ?

Appendix

Spliterator

Splitterator

```
interface Splitterator<T> {  
    ...  
  
    long estimateSize(); // Long.MAX_VALUE if unknown  
  
    boolean tryAdvance(Consumer<T> action);  
  
    Splitterator<T> trySplit();  
  
    int characteristics();  
  
    ...  
}
```


Splititerator's characteristic

ORDERED

DISTINCT

SORTED

SIZED

SUBSIZED

NONNULL

IMMUTABLE

CONCURRENT

Stream design

I like to look at this as having chosen a design center that recognizes that sequential is a degenerate case of parallel, rather than treating parallel as the “weird bonus mode”. I realize that this choice was controversial and definitely caused some compromises, but eventually people will have to start to unlearn their sequential biases, and there's no time like the present.

(c) Brian Goetz

<http://mail.openjdk.java.net/pipermail/lambda-dev/2014-February/011870.html>