
JAVA 8
STREAMS
...
Daniel Kocsis
2016. December

Recap

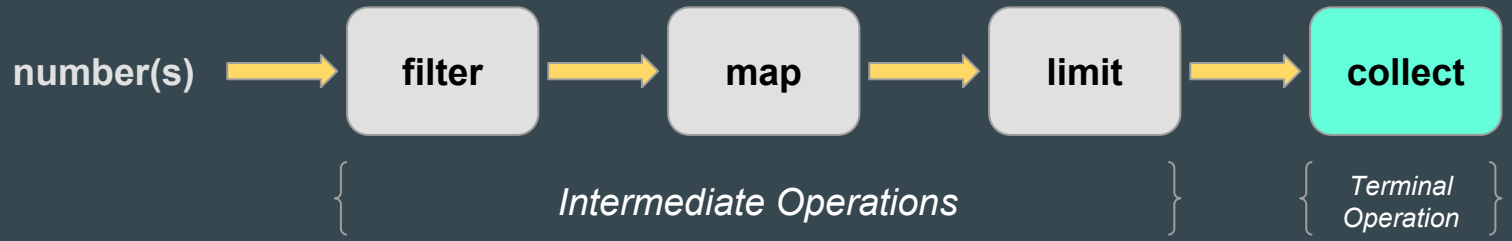
- Functional interfaces: `@FunctionalInterface` + **only 1** abstract method
- Lambda expression(**closures** over finals): `(int y) -> {return 2 * y;}`
- Optional: `Optional.of("string_literal");`
- Functional Programming (FP) Concepts
  - Closure = Function pointer + Stack frame
  - Functor

```
public <R> Functor<R, F> map(Function<T, R> function);
```

  - Monad

```
public <S> Monad<S> flatMap(Function<T, Monad<S>> function);
```

What is a Stream?


number(s)
filter
map
limit
collect
Intermediate Operations
Terminal Operation

"A sequence of elements supporting sequential and parallel aggregate operations."

- The characteristics of a Stream
  - **Sequence of elements**
  - **Source**: it takes Collections, Arrays, or I/O resources as input source
  - **Aggregate operations:** filter, map, limit, reduce, find, match...
  - **Pipelining:** some stream operations (non-**terminal** or **intermediate** ones) can be chained
  - **Automatic (internal) iterations**
  - **Sequential** vs **Parallel** streams
- All the stream related classes are in the java.util.stream package
- `Stream<T>` is the main interface, represents a stream of object references, however there are specialized versions mostly for primitives, such as `IntStream`, `LongStream`, `DoubleStream` ...

Streams vs Collections

- As an API, Streams is completely independent from Collections. While it is easy to use a collection as the source for a stream
- Streams don't have storage for values; they carry values from a source through a pipeline of computational steps.
- Functional in nature. Operations on a **stream** produce a result, but **do not modify its underlying data source**.
- Laziness-seeking. Many stream operations, such as filtering, mapping, sorting, or duplicate removal) can be implemented lazily.
- Bounds optional. There are many problems that are sensible to express as infinite streams, letting clients consume values until they are satisfied. While a Collection is constrained to be finite, a stream is not.

Remember! Streams are not collections!

```
List<String> list = new ArrayList<>();
list.add("a");
list.add("b");

Stream<String> stream = list.stream();

list.add("c");

stream.forEach(System.out::println); // a,b,c
```

Java Streams are Monads!

<R> **Stream**<R>	**flatMap**(**Function**<? super **T**,? extends **Stream**<? extends R>> mapper) Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.

<R> **Stream**<R>	**map**(**Function**<? super **T**,? extends R> mapper) Returns a stream consisting of the results of applying the given function to the elements of this stream.

```
List<String> myList = Arrays.asList("apple", "pear", "cucumber", "cherry", "tomato");


myList.stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);

// out: CHERRY, CUCUMBER
```

Creating Streams

From Arrays

1. `Stream.of(T… values);`
2. `Stream.of(T t);`
3. `Arrays.stream();` with various parameter list

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

Stream<String> stream = Stream.of(array);

Stream<String> stream = Arrays.stream(array);

Stream<String> stream = Stream.of("a", "b", "c", "d");
```

From Collections

- `Stream<E> stream()`

```
List<String> list = new ArrayList<String>();


list.add("a");
list.add("b");
list.add("c");


stream = list.stream();
```

Stream.generate()

- `static <T> Stream<T> generate(Supplier<T> s)`

```
Stream<String> stream = Stream.generate(() -> "test").limit(10);

Stream.generate(Math::random).limit(2).forEach(System.out::println)
```

Stream.iterate()

- `static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)`

```
Stream<BigInteger> bigIntStream =
    Stream.iterate(
              BigInteger.ZERO,
              n -> n.add(BigInteger.ONE))
          .limit(10);
```

From popular API's

- It depends on the API's, but for example:

```
String sentence = "It is good to learn new things:)";
Stream<String> wordStream =
    Pattern.compile("\\W").splitAsStream(sentence);
```

Infinite Streams

- Without limiting the `Stream.iterate` and `Stream.generate` methods we generate infinite streams
- **Infinite Stream -> Infinite runtime** (without any additional logic)
- To avoid infinite stream creation use the `limit` intermediate operation
- Examples of infinite streams:

```
Stream<Integer> evenNumbers = Stream.iterate(0, n -> n + 2);
```

AND

```
Stream<Integer> randomNumbers = Stream.generate(Math::random);
```

Stream operations

Intermediate versus Terminal Operations

- Evaluation
  - Intermediate operations are not evaluated until we chain it with a Terminal Operation of Stream. Terminal Operations can be independently evaluated.
- Output
  - The output of Intermediate Operations is another Stream. The output of Terminal Operations is not a Stream.
- Laziness
  - Intermediate Operations are evaluated in lazy manner. Terminal Operations are evaluated in eager manner.
- Chaining
  - We can chain multiple Intermediate Operations in a Stream. Terminal Operations cannot be chained multiple times.
- Multiplicity
  - There can be multiple Intermediate operations in a Stream operation. There can be only one Terminal operation in Stream processing statement.

Intermediate Stream Operations

filter(Predicate<? super T> predicate)

- This operation will return a new stream that contains elements that match its predicate.

```
long elementsLessThanThree =
    Stream.of(1, 2, 3, 4)
            .filter(p -> p.intValue() < 3)
            .count();
```

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

- Functor operation
- This operation will transform the elements elements in a stream using the provided mapper function.

```
List<String> strings =
      Stream.of("one", null, "three")
            .map(s ->
                  {
                        if (s == null) {
                              return "[unknown]";
                        }
                        else {
                              return s;
                        }
                  })
            .collect(Collectors.toList());
```

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

- Monadic operation!
- This operations will transform each element into zero or more elements by a way of another stream.

```
File file = new File(sourceFileURI);


long uniqueWords = java.nio.file.Files
          .lines(Paths.get(file.toURI()), Charset.defaultCharset())
          .flatMap(line -> Arrays.stream(line.split(" .")))
          .distinct()
          .count();
```

peek(Consumer<? super T> action)

- This is very useful when you need to debug your code
- It allows you to peek into the stream before an action is encountered

```
List<String> strings =
     Stream.of("apple", "cherry", "pineapple")
          .peek(s -> System.out.println(s)) // "apple", "cherry", "pineapple"
          .filter(s -> s.length() > 6)
          .peek(s -> System.out.println(s)) // "pineapple"
          .map(s -> s.toUpperCase())
          .peek(s -> System.out.println(s)) // "PINEAPPLE"
          .collect(Collectors.toList());
```

distinct()

- This operation will find unique elements in a stream according to their `equals()` method.

```
List<Integer> distinctIntegers =
     IntStream.of(5, 6, 6, 6, 3, 2, 2)
          .distinct()
          .boxed()
          .collect(Collectors.toList());
```

sorted()

- This method will return a stream sorted according to natural order

```
List<Integer> sortedNumbers =
     Stream.of(5, 3, 1, 3, 6)
          .sorted()
          .collect(Collectors.toList());
```

limit(long maxSize)

- Using limit is a useful technique to limit the number or truncate elements to be processed in the stream.

```
List<String> vals =
    Stream.of("limit", "by", "two")
        .limit(2)
        .collect(Collectors.toList()); // "limit", "by"
```

Terminal Stream Operations

forEach(Consumer<? super T> action)

- This method will perform an action for each element in the stream.
- It is a simplified inline way to write a for loop.

```
Stream.of("Hello", "World").forEach(p -> System.out.println(p));
```

toArray()

- This method will returns an array containing the elements of the stream

```
Object[] objects = Stream.of(“a”, “b”).toArray();
```

reduce

- **reduce**`(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)`
  - Performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.
  - When you run the stream in parallel, the task is spanned into multiple threads. Then the `combiner` is used to merge their results. For non-parallel streams `combiner` will be ignored.
- **reduce**`(T identity, BinaryOperator<T> accumulator)`
  - Performs a reduction on the elements of this stream, using the provided identity, accumulation and combining functions.

```
int sum = IntStream.of(1, 2, 3, 4)
                   .reduce(0, (a, b) -> a + b); // 10
```

```
List<String> results = Arrays.asList("Apple", "Bear", "Anaconda", "Cherry");

Long countOfAWords =
    results.stream()
           .reduce(
               0L, //identity
               (a, b) -> b.charAt(0) == 'A' ? a + 1 : a,  //accumulator
               (a, b) -> Long.sum(a, b));  //combiner
```

collect

- Unlike the reduce method, which always creates a new value when it processes an element, the collect method modifies, or mutates, an existing value.
- Collect is a mutable reduction
- Has 2 different versions
  - `collect(Collector<? super T,A,R> collector)`
  - ```
    collect(
        Supplier<R> supplier, BiConsumer<R,? super T> accumulator,
        BiConsumer<R,R> combiner)
    ```
- The type `Collector` encapsulates the functions used as arguments in the collect operation that requires three arguments
- There are built in Collector functions in java.util.Collectors, such as toList, toMap...
    ```

Be aware!

```
Arrays.asList("alpha", "bravo", "charlie")

   .stream()

   .map(e -> e.toUpperCase())

   .collect(Collectors.toList()) // this is a terminal operation

   .forEach(System.out::println) // this is starting a new iteration
```

collect - continued

- `supplier`: The supplier is a factory function; it constructs new instances. For the collect operation, it creates instances of the result container.
- `accumulator`: The accumulator function incorporates a stream element into a result container.
- `combiner`: The combiner function takes two result containers and merges their contents.
- Note the following:
  - The `supplier` is a lambda expression (or a method reference) as opposed to a value like the identity element in the reduce operation.
  - The `accumulator` and `combiner` functions do not return a value.
  - You can use the collect operations with parallel streams

min, max, count

- Min, max return the min, max value of a stream based on a given `Comparator<? super T>` comparator
- Count will find the number of elements in the stream.

anyMatch, allMatch, noneMatch

- `anyMatch` will find out whether at least one of the elements in the stream matches a given predicate.
- `allMatch` will check every element in the stream and find out if it matches the predicate.
- Just the opposite of `anymatch`, `noneMatch` will find if no elements in the stream match the specified predicate.

```
List<String> words = Lists.newArrayList(
     "apple", "beer", "cinnamon", "dolphin");

boolean anyStartsWithB = words.stream().anyMatch(
     p -> p.toLowerCase().startsWith("b")); // true

boolean allStartsWithB = words.stream().allMatch(
     p -> p.toLowerCase().startsWith("b")); // false

boolean noneStartsWithB = words.stream().noneMatch(
     p -> p.toLowerCase().startsWith("b")); // false
```

findFirst, findAny

- `findFirst` will find the first element in the stream which is like the same behavior as getting the first element in a list.
- Similar to finding any element in array, `findAny` will find any element in a given stream
- When using parallel streams `findAny, anyMatch` are much better options since no ordering is involved, so no synchronization is necessary

```
List<String> words = Lists.newArrayList(
     "apple", "beer", "cinnamon", "dolphin");

Optional<String> val = words.findFirst(); // “apple”
Optional<String> val = words.findAny(); // “apple”
```

Performance

How does it perform compared to old iterations?

- There are many articles in the internet, but you should **measure** your solution if performance is a requirement!
- Using Streams and declarative style code is comfortable for the coder, but it does not bring an out-of-the-box performance improvement!
- In very simple cases and a naive programming approach some bloggers measure 3-5 times performance overhead when using streams
- See these articles for more details:
  - https://jaxenter.com/java-performance-tutorial-how-fast-are-the-java-8-streams-118830.html
  - http://blog.codefx.org/java/stream-performance/
  - http://blog.takipi.com/benchmark-how-java-8-lambdas-and-streams-can-make-your-code-5-times-slower/

Primitive Streams

- `IntStream`, `LongStream`, `DoubleStream` has its own implementation
- `IntStream` != `Stream<Integer>` and there is no such thing ~~`Stream<int>`~~
- [Auto]boxing is something that degrades the Stream performance, so **when you work with primitives use these specialized Streams**!
- `OptionalInt`, `OptionalLong` and `OptionalDouble` are also exist
- `OptionalInt` != `Optional<Integer>` and there is no such thing ~~`Optional<int>`~~
- These specialized classes have specialized methods based on their type
  - `IntStream.average()`
  - `IntStream.summaryStatistics()` - contains informations about the numerical Stream like min, max, avg, sum, count

Parallel Streams

Strea  allel()

How does it work?

- The parallel stream uses the Fork/Join Framework for processing (from Java SE 7)
- The stream-source is getting forked (splitted) and hands over to the fork/join-pool workers for execution, then it is getting merged together
- The number of threads scales up to the number of CPU cores available in the system by default
- The order of processing is not determined and can vary run-by-run!
- The parallel stream can work efficiently only if the operations are independents and stateless

Generic Rules #1

- “N*Q – factor”: number of elements * cost per element should be large.
  - Rule of thumb – Need NQ > 10,000 to have a chance for parallel speedup
  - Source collection must be efficiently splittable (Arrays are easily splittable, meanwhile LinkedLists, Files are not)
  - Iterative generators behave like linked lists, stateless generators behave like arrays
- Locality
  - Stream.of(int[]) vs Stream.of(Integer[])
- The per-element function has to be independent
- Merging
  - For some operations (sum, max) the merge operation is really cheap
  - For others (groupingBy to a HashMap) it is insanely expensive!
-
- Source is a must see presentation by Brian Goetz (Oracle)

Generic Rules #2

- There are special `forEachOrdered()` terminal op. but of course it strongly degrades the performance. You can use this when order matters
- It is **strongly** recommended to avoid stateful operations and side effects when using parallel streams otherwise the performance and the result can be really bad!
- When using 3 param versions of `collect()` and `reduce()` use concurrent collections to store the results otherwise some mysterious error could happen
- `isPrallel()` helps to test whether a stream is parallel or not, for example flatMap() produces a new stream and independently of the source the result won't be parallel by default!
- `Stream.unordered()` can greatly improve performance when parallel streams, it tells the JVM to please optimize if possible, but doesn't reorder the collection by default!

```
Array.asList(1,2,3).stream(). unordered().parallel()....
```

Conclusion for parallel streams

Streams are cool and parallel processing is also cool!

But...

> Stream.parallel() is not a magical box, it won't give give 100% performance gain immediately

So...

- Always think first!
- then think again
- then **measure!**
- then use parallel Streams if it still looks a good idea!

Java 8 Streams Cheat Sheet

For more awesome cheat sheets visit rebellabs.org!

REBELLABS by ZEROTURNAROUND

Definitions

- A stream **is** a pipeline of functions that can be evaluated.
- Streams **can** transform data.
- A stream **is not** a data structure.
- Streams **cannot** mutate data.

Intermediate operations

- Always return streams.
- Lazily executed.

Common examples include:

Function	Preserves count	Preserves type	Preserves order
map	✓	✗	✓
filter	✗	✓	✓
distinct	✗	✓	✓
sorted	✓	✓	✗
peek	✓	✓	✓

Stream examples

Get the unique surnames in uppercase of the first 15 book authors that are 50 years old or over.

```
library.stream()
     .map(book -> book.getAuthor())
     .filter(author -> author.getAge() >= 50)
     .distinct()
     .limit(15)
     .map(Author::getSurname)
     .map(String::toUpperCase)
     .collect(toList());
```

Compute the sum of ages of all female authors younger than 25.

```
library.stream()
     .map(Book::getAuthor)
     .filter(a -> a.getGender() == Gender.FEMALE)
     .map(Author::getAge)
     .filter(age -> age < 25)
     .reduce(0, Integer::sum);
```

Terminal operations

- Return concrete types or produce a side effect.
- Eagerly executed.

Common examples include:

Function	Output	When to use
reduce	concrete type	to cumulate elements
collect	list, map or set	to group elements
forEach	side effect	to perform a side effect on elements

Parallel streams

Parallel streams use the common ForkJoinPool for threading.

```
library.parallelStream()...
```

or intermediate operation:

```
IntStream.range(1, 10).parallel()...
```

Useful operations

Grouping:

```
library.stream().collect(
     groupingBy(Book::getGenre));
```

Stream ranges:

```
IntStream.range(0, 20)...
```

Infinite streams:

```
IntStream.iterate(0, e -> e + 1)...
```

Max/Min:

```
IntStream.range(1, 10).max();
```

FlatMap:

```
twitterList.stream()
     .map(member -> member.getFollowers())
     .flatMap(followers -> followers.stream())
     .collect(toList());
```

Pitfalls

- Don't update shared mutable variables i.e.

```
List<Book> myList =
new ArrayList<>();
library.stream().forEach
(e -> myList.add(e));
```

- Avoid blocking operations when using parallel streams.

BROUGHT TO YOU BY JRebel

http://files.zeroturnaround.com/pdf/zt_java8_streams_cheat_sheet.pdf

Q&A