Java 8 Stream Tutorial

July 31, 2014

This example-driven tutorial gives an in-depth overview about Java 8 streams. When I first read about the `Stream` API, I was confused about the name since it sounds similar to `InputStream` and `OutputStream` from Java I/O. But Java 8 streams are a completely different thing. Streams are Monads, thus playing a big part in bringing *functional programming* to Java:

> *In functional programming, a monad is a structure that represents computations defined as sequences of steps. A type with a monad structure defines what it means to chain operations, or nest functions of that type together.*

This guide teaches you how to work with Java 8 streams and how to use the different kind of available stream operations. You'll learn about the processing order and how the ordering of stream operations affect runtime performance. The more powerful stream operations `reduce`, `collect` and `flatMap` are covered in detail. The tutorial ends with an in-depth look at parallel streams.

If you're not yet familiar with Java 8 lambda expressions, functional interfaces and method references, you probably want to read my Java 8 Tutorial first before starting with this tutorial.

How streams work

A stream represents a sequence of elements and supports different kind of operations to perform computations upon those elements:

```
List<String> myList =
    Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList
    .stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
```

```
        .sorted()
        .forEach(System.out::println);

// C1
// C2
```

Stream operations are either intermediate or terminal. Intermediate operations return a stream so we can chain multiple intermediate operations without using semicolons. Terminal operations are either void or return a non-stream result. In the above example `filter`, `map` and `sorted` are intermediate operations whereas `forEach` is a terminal operation. For a full list of all available stream operations see the Stream Javadoc. Such a chain of stream operations as seen in the example above is also known as *operation pipeline*.

Most stream operations accept some kind of lambda expression parameter, a functional interface specifying the exact behavior of the operation. Most of those operations must be both *non-interfering* and *stateless*. What does that mean?

A function is non-interfering when it does not modify the underlying data source of the stream, e.g. in the above example no lambda expression does modify `myList` by adding or removing elements from the collection.

A function is stateless when the execution of the operation is deterministic, e.g. in the above example no lambda expression depends on any mutable variables or states from the outer scope which might change during execution.

Different kind of streams

Streams can be created from various data sources, especially collections. Lists and Sets support new methods `stream()` and `parallelStream()` to either create a sequential or a parallel stream. Parallel streams are capable of operating on multiple threads and will be covered in a later section of this tutorial. We focus on sequential streams for now:

```
Arrays.asList("a1", "a2", "a3")
    .stream()
    .findFirst()
    .ifPresent(System.out::println);  // a1
```

Calling the method `stream()` on a list of objects returns a regular object stream. But we don't have to create collections in order to work with streams as we see in the next code sample:

```
Stream.of("a1", "a2", "a3")
    .findFirst()
    .ifPresent(System.out::println);  // a1
```

Just use `Stream.of()` to create a stream from a bunch of object references.

Besides regular object streams Java 8 ships with special kinds of streams for working with the primitive data types `int`, `long` and `double`. As you might have guessed it's `IntStream`, `LongStream` and `DoubleStream`.

IntStreams can replace the regular for-loop utilizing `IntStream.range()`:

```
IntStream.range(1, 4)
    .forEach(System.out::println);

// 1
// 2
// 3
```

All those primitive streams work just like regular object streams with the following differences: Primitive streams use specialized lambda expressions, e.g. `IntFunction` instead of `Function` or `IntPredicate` instead of `Predicate`. And primitive streams support the additional terminal aggregate operations `sum()` and `average()`:

```
Arrays.stream(new int[] {1, 2, 3})
    .map(n -> 2 * n + 1)
    .average()
    .ifPresent(System.out::println);  // 5.0
```

Sometimes it's useful to transform a regular object stream to a primitive stream or vice versa. For that purpose object streams support the special mapping operations `mapToInt()`, `mapToLong()` and `mapToDouble`:

```
Stream.of("a1", "a2", "a3")
    .map(s -> s.substring(1))
    .mapToInt(Integer::parseInt)
    .max()
    .ifPresent(System.out::println);  // 3
```

Primitive streams can be transformed to object streams via `mapToObj()`:

```
IntStream.range(1, 4)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);

// a1
// a2
// a3
```

Here's a combined example: the stream of doubles is first mapped to an int stream and than mapped to an object stream of strings:

```
Stream.of(1.0, 2.0, 3.0)
    .mapToInt(Double::intValue)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);

// a1
// a2
// a3
```

Processing Order

Now that we've learned how to create and work with different kinds of streams, let's dive deeper into how stream operations are processed under the hood.

An important characteristic of intermediate operations is laziness. Look at this sample where a terminal operation is missing:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return true;
    });
```

When executing this code snippet, nothing is printed to the console. That is because intermediate operations will only be executed when a terminal operation is present.

Let's extend the above example by the terminal operation `forEach`:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
```

```
            return true;
        })
        .forEach(s -> System.out.println("forEach: " + s));
```

Executing this code snippet results in the desired output on the console:

```
filter:  d2
forEach: d2
filter:  a2
forEach: a2
filter:  b1
forEach: b1
filter:  b3
forEach: b3
filter:  c
forEach: c
```

The order of the result might be surprising. A naive approach would be to execute the operations horizontally one after another on all elements of the stream. But instead each element moves along the chain vertically. The first string "d2" passes `filter` then `forEach`, only then the second string "a2" is processed.

This behavior can reduce the actual number of operations performed on each element, as we see in the next example:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .anyMatch(s -> {
        System.out.println("anyMatch: " + s);
        return s.startsWith("A");
    });

// map:      d2
// anyMatch: D2
// map:      a2
// anyMatch: A2
```

The operation `anyMatch` returns `true` as soon as the predicate applies to the given input element. This is true for the second element passed "A2". Due to the vertical execution of the stream chain, `map` has

only to be executed twice in this case. So instead of mapping all elements of the stream, `map` will be called as few as possible.

Why order matters

The next example consists of two intermediate operations `map` and `filter` and the terminal operation `forEach`. Let's once again inspect how those operations are being executed:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("A");
    })
    .forEach(s -> System.out.println("forEach: " + s));

// map:     d2
// filter:  D2
// map:     a2
// filter:  A2
// forEach: A2
// map:     b1
// filter:  B1
// map:     b3
// filter:  B3
// map:     c
// filter:  C
```

As you might have guessed both `map` and `filter` are called five times for every string in the underlying collection whereas `forEach` is only called once.

We can greatly reduce the actual number of executions if we change the order of the operations, moving `filter` to the beginning of the chain:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
```

```
    })
    .forEach(s -> System.out.println("forEach: " + s));

// filter:  d2
// filter:  a2
// map:     a2
// forEach: A2
// filter:  b1
// filter:  b3
// filter:  c
```

Now, `map` is only called once so the operation pipeline performs much faster for larger numbers of input elements. Keep that in mind when composing complex method chains.

Let's extend the above example by an additional operation, `sorted`:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .sorted((s1, s2) -> {
        System.out.printf("sort: %s; %s\n", s1, s2);
        return s1.compareTo(s2);
    })
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

Sorting is a special kind of intermediate operation. It's a so called *stateful operation* since in order to sort a collection of elements you have to maintain state during ordering.

Executing this example results in the following console output:

```
sort:    a2; d2
sort:    b1; a2
sort:    b1; d2
sort:    b1; a2
sort:    b3; b1
sort:    b3; d2
sort:    c; b3
sort:    c; d2
filter:  a2
map:     a2
```

```
forEach: A2
filter:  b1
filter:  b3
filter:  c
filter:  d2
```

First, the sort operation is executed on the entire input collection. In other words `sorted` is executed horizontally. So in this case `sorted` is called eight times for multiple combinations on every element in the input collection.

Once again we can optimize the performance by reordering the chain:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .sorted((s1, s2) -> {
        System.out.printf("sort: %s; %s\n", s1, s2);
        return s1.compareTo(s2);
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println("forEach: " + s));

// filter:  d2
// filter:  a2
// filter:  b1
// filter:  b3
// filter:  c
// map:     a2
// forEach: A2
```

In this example `sorted` is never been called because `filter` reduces the input collection to just one element. So the performance is greatly increased for larger input collections.

Reusing Streams

Java 8 streams cannot be reused. As soon as you call any terminal operation the stream is closed:

```
Stream<String> stream =
    Stream.of("d2", "a2", "b1", "b3", "c")
```

```
        .filter(s -> s.startsWith("a"));

stream.anyMatch(s -> true);    // ok
stream.noneMatch(s -> true);   // exception
```

Calling `noneMatch` after `anyMatch` on the same stream results in the following exception:

```
java.lang.IllegalStateException: stream has already been operated upon or closed
    at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:229)
    at java.util.stream.ReferencePipeline.noneMatch(ReferencePipeline.java:459)
    at com.winterbe.java8.Streams5.test7(Streams5.java:38)
    at com.winterbe.java8.Streams5.main(Streams5.java:28)
```

To overcome this limitation we have to to create a new stream chain for every terminal operation we want to execute, e.g. we could create a stream supplier to construct a new stream with all intermediate operations already set up:

```
Supplier<Stream<String>> streamSupplier =
    () -> Stream.of("d2", "a2", "b1", "b3", "c")
            .filter(s -> s.startsWith("a"));

streamSupplier.get().anyMatch(s -> true);   // ok
streamSupplier.get().noneMatch(s -> true);  // ok
```

Each call to `get()` constructs a new stream on which we are save to call the desired terminal operation.

Advanced Operations

Streams support plenty of different operations. We've already learned about the most important operations like `filter` or `map`. I leave it up to you to discover all other available operations (see Stream Javadoc). Instead let's dive deeper into the more complex operations `collect`, `flatMap` and `reduce`.

Most code samples from this section use the following list of persons for demonstration purposes:

```
class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
```

```
    @Override
    public String toString() {
        return name;
    }
}

List<Person> persons =
    Arrays.asList(
        new Person("Max", 18),
        new Person("Peter", 23),
        new Person("Pamela", 23),
        new Person("David", 12));
```

Collect

Collect is an extremely useful terminal operation to transform the elements of the stream into a different kind of result, e.g. a `List`, `Set` or `Map`. Collect accepts a `Collector` which consists of four different operations: a *supplier*, an *accumulator*, a *combiner* and a *finisher*. This sounds super complicated at first, but the good part is Java 8 supports various built-in collectors via the `Collectors` class. So for the most common operations you don't have to implement a collector yourself.

Let's start with a very common usecase:

```
List<Person> filtered =
    persons
        .stream()
        .filter(p -> p.name.startsWith("P"))
        .collect(Collectors.toList());

System.out.println(filtered);    // [Peter, Pamela]
```

As you can see it's very simple to construct a list from the elements of a stream. Need a set instead of list - just use `Collectors.toSet()`.

The next example groups all persons by age:

```
Map<Integer, List<Person>> personsByAge = persons
    .stream()
    .collect(Collectors.groupingBy(p -> p.age));

personsByAge
    .forEach((age, p) -> System.out.format("age %s: %s\n", age, p));
```

```
// age 18: [Max]
// age 23: [Peter, Pamela]
// age 12: [David]
```

Collectors are extremely versatile. You can also create aggregations on the elements of the stream, e.g. determining the average age of all persons:

```
Double averageAge = persons
    .stream()
    .collect(Collectors.averagingInt(p -> p.age));

System.out.println(averageAge);     // 19.0
```

If you're interested in more comprehensive statistics, the summarizing collectors return a special built-in summary statistics object. So we can simply determine *min*, *max* and arithmetic *average* age of the persons as well as the *sum* and *count*.

```
IntSummaryStatistics ageSummary =
    persons
        .stream()
        .collect(Collectors.summarizingInt(p -> p.age));

System.out.println(ageSummary);
// IntSummaryStatistics{count=4, sum=76, min=12, average=19.000000, max=23}
```

The next example joins all persons into a single string:

```
String phrase = persons
    .stream()
    .filter(p -> p.age >= 18)
    .map(p -> p.name)
    .collect(Collectors.joining(" and ", "In Germany ", " are of legal age."));

System.out.println(phrase);
// In Germany Max and Peter and Pamela are of legal age.
```

The join collector accepts a delimiter as well as an optional prefix and suffix.

In order to transform the stream elements into a map, we have to specify how both the keys and the values should be mapped. Keep in mind that the mapped keys must be unique, otherwise an

`IllegalStateException` is thrown. You can optionally pass a merge function as an additional parameter to bypass the exception:

```
Map<Integer, String> map = persons
    .stream()
    .collect(Collectors.toMap(
        p -> p.age,
        p -> p.name,
        (name1, name2) -> name1 + ";" + name2));

System.out.println(map);
// {18=Max, 23=Peter;Pamela, 12=David}
```

Now that we know some of the most powerful built-in collectors, let's try to build our own special collector. We want to transform all persons of the stream into a single string consisting of all names in upper letters separated by the `|` pipe character. In order to achieve this we create a new collector via `Collector.of()`. We have to pass the four ingredients of a collector: a *supplier*, an *accumulator*, a *combiner* and a *finisher*.

```
Collector<Person, StringJoiner, String> personNameCollector =
    Collector.of(
        () -> new StringJoiner(" | "),          // supplier
        (j, p) -> j.add(p.name.toUpperCase()),  // accumulator
        (j1, j2) -> j1.merge(j2),               // combiner
        StringJoiner::toString);                // finisher

String names = persons
    .stream()
    .collect(personNameCollector);

System.out.println(names);  // MAX | PETER | PAMELA | DAVID
```

Since strings in Java are immutable, we need a helper class like `StringJoiner` to let the collector construct our string. The supplier initially constructs such a StringJoiner with the appropriate delimiter. The accumulator is used to add each persons upper-cased name to the StringJoiner. The combiner knows how to merge two StringJoiners into one. In the last step the finisher constructs the desired String from the StringJoiner.

FlatMap

We've already learned how to transform the objects of a stream into another type of objects by utilizing the `map` operation. Map is kinda limited because every object can only be mapped to exactly one other object. But what if we want to transform one object into multiple others or none at all? This is where `flatMap` comes to the rescue.

FlatMap transforms each element of the stream into a stream of other objects. So each object will be transformed into zero, one or multiple other objects backed by streams. The contents of those streams will then be placed into the returned stream of the `flatMap` operation.

Before we see `flatMap` in action we need an appropriate type hierarchy:

```
class Foo {
    String name;
    List<Bar> bars = new ArrayList<>();

    Foo(String name) {
        this.name = name;
    }
}

class Bar {
    String name;

    Bar(String name) {
        this.name = name;
    }
}
```

Next, we utilize our knowledge about streams to instantiate a couple of objects:

```
List<Foo> foos = new ArrayList<>();

// create foos
IntStream
    .range(1, 4)
    .forEach(i -> foos.add(new Foo("Foo" + i)));

// create bars
foos.forEach(f ->
    IntStream
        .range(1, 4)
        .forEach(i -> f.bars.add(new Bar("Bar" + i + " <- " + f.name))));
```

Now we have a list of three foos each consisting of three bars.

FlatMap accepts a function which has to return a stream of objects. So in order to resolve the bar objects of each foo, we just pass the appropriate function:

```
foos.stream()
    .flatMap(f -> f.bars.stream())
    .forEach(b -> System.out.println(b.name));

// Bar1 <- Foo1
// Bar2 <- Foo1
// Bar3 <- Foo1
// Bar1 <- Foo2
// Bar2 <- Foo2
// Bar3 <- Foo2
// Bar1 <- Foo3
// Bar2 <- Foo3
// Bar3 <- Foo3
```

As you can see, we've successfully transformed the stream of three foo objects into a stream of nine bar objects.

Finally, the above code example can be simplified into a single pipeline of stream operations:

```
IntStream.range(1, 4)
    .mapToObj(i -> new Foo("Foo" + i))
    .peek(f -> IntStream.range(1, 4)
        .mapToObj(i -> new Bar("Bar" + i + " <- " f.name))
        .forEach(f.bars::add))
    .flatMap(f -> f.bars.stream())
    .forEach(b -> System.out.println(b.name));
```

FlatMap is also available for the `Optional` class introduced in Java 8. Optionals `flatMap` operation returns an optional object of another type. So it can be utilized to prevent nasty `null` checks.

Think of a highly hierarchical structure like this:

```
class Outer {
    Nested nested;
}

class Nested {
    Inner inner;
}

class Inner {
```

```
    String foo;
}
```

In order to resolve the inner string `foo` of an outer instance you have to add multiple null checks to prevent possible NullPointerExceptions:

```
Outer outer = new Outer();
if (outer != null && outer.nested != null && outer.nested.inner != null) {
    System.out.println(outer.nested.inner.foo);
}
```

The same behavior can be obtained by utilizing optionals `flatMap` operation:

```
Optional.of(new Outer())
    .flatMap(o -> Optional.ofNullable(o.nested))
    .flatMap(n -> Optional.ofNullable(n.inner))
    .flatMap(i -> Optional.ofNullable(i.foo))
    .ifPresent(System.out::println);
```

Each call to `flatMap` returns an `Optional` wrapping the desired object if present or `null` if absent.

Reduce

The reduction operation combines all elements of the stream into a single result. Java 8 supports three different kind of `reduce` methods. The first one reduces a stream of elements to exactly one element of the stream. Let's see how we can use this method to determine the oldest person:

```
persons
    .stream()
    .reduce((p1, p2) -> p1.age > p2.age ? p1 : p2)
    .ifPresent(System.out::println);    // Pamela
```

The `reduce` method accepts a `BinaryOperator` accumulator function. That's actually a `BiFunction` where both operands share the same type, in that case `Person`. BiFunctions are like `Function` but accept two arguments. The example function compares both persons ages in order to return the person with the maximum age.

The second `reduce` method accepts both an identity value and a `BinaryOperator` accumulator. This method can be utilized to construct a new Person with the aggregated names and ages from all other

persons in the stream:

```
Person result =
    persons
        .stream()
        .reduce(new Person("", 0), (p1, p2) -> {
            p1.age += p2.age;
            p1.name += p2.name;
            return p1;
        });

System.out.format("name=%s; age=%s", result.name, result.age);
// name=MaxPeterPamelaDavid; age=76
```

The third `reduce` method accepts three parameters: an identity value, a `BiFunction` accumulator and a combiner function of type `BinaryOperator`. Since the identity values type is not restricted to the `Person` type, we can utilize this reduction to determine the sum of ages from all persons:

```
Integer ageSum = persons
    .stream()
    .reduce(0, (sum, p) -> sum += p.age, (sum1, sum2) -> sum1 + sum2);

System.out.println(ageSum);  // 76
```

As you can see the result is *76*, but what's happening exactly under the hood? Let's extend the above code by some debug output:

```
Integer ageSum = persons
    .stream()
    .reduce(0,
        (sum, p) -> {
            System.out.format("accumulator: sum=%s; person=%s\n", sum, p);
            return sum += p.age;
        },
        (sum1, sum2) -> {
            System.out.format("combiner: sum1=%s; sum2=%s\n", sum1, sum2);
            return sum1 + sum2;
        });

// accumulator: sum=0; person=Max
// accumulator: sum=18; person=Peter
// accumulator: sum=41; person=Pamela
// accumulator: sum=64; person=David
```

As you can see the accumulator function does all the work. It first get called with the initial identity value *0* and the first person *Max*. In the next three steps `sum` continually increases by the age of the last steps person up to a total age of *76*.

Wait wat? The combiner never gets called? Executing the same stream in parallel will lift the secret:

```
Integer ageSum = persons
    .parallelStream()
    .reduce(0,
        (sum, p) -> {
            System.out.format("accumulator: sum=%s; person=%s\n", sum, p);
            return sum += p.age;
        },
        (sum1, sum2) -> {
            System.out.format("combiner: sum1=%s; sum2=%s\n", sum1, sum2);
            return sum1 + sum2;
        });

// accumulator: sum=0; person=Pamela
// accumulator: sum=0; person=David
// accumulator: sum=0; person=Max
// accumulator: sum=0; person=Peter
// combiner: sum1=18; sum2=23
// combiner: sum1=23; sum2=12
// combiner: sum1=41; sum2=35
```

Executing this stream in parallel results in an entirely different execution behavior. Now the combiner is actually called. Since the accumulator is called in parallel, the combiner is needed to sum up the separate accumulated values.

Let's dive deeper into parallel streams in the next chapter.

Parallel Streams

Streams can be executed in parallel to increase runtime performance on large amount of input elements. Parallel streams use a common `ForkJoinPool` available via the static `ForkJoinPool.commonPool()` method. The size of the underlying thread-pool uses up to five threads - depending on the amount of available physical CPU cores:

```
ForkJoinPool commonPool = ForkJoinPool.commonPool();
System.out.println(commonPool.getParallelism());    // 3
```

On my machine the common pool is initialized with a parallelism of 3 per default. This value can be decreased or increased by setting the following JVM parameter:

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=5
```

Collections support the method `parallelStream()` to create a parallel stream of elements. Alternatively you can call the intermediate method `parallel()` on a given stream to convert a sequential stream to a parallel counterpart.

In order to understate the parallel execution behavior of a parallel stream the next example prints information about the current thread to `sout`:

```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
    .parallelStream()
    .filter(s -> {
        System.out.format("filter: %s [%s]\n",
            s, Thread.currentThread().getName());
        return true;
    })
    .map(s -> {
        System.out.format("map: %s [%s]\n",
            s, Thread.currentThread().getName());
        return s.toUpperCase();
    })
    .forEach(s -> System.out.format("forEach: %s [%s]\n",
        s, Thread.currentThread().getName()));
```

By investigating the debug output we should get a better understanding which threads are actually used to execute the stream operations:

```
filter:  b1 [main]
filter:  a2 [ForkJoinPool.commonPool-worker-1]
map:     a2 [ForkJoinPool.commonPool-worker-1]
filter:  c2 [ForkJoinPool.commonPool-worker-3]
map:     c2 [ForkJoinPool.commonPool-worker-3]
filter:  c1 [ForkJoinPool.commonPool-worker-2]
map:     c1 [ForkJoinPool.commonPool-worker-2]
forEach: C2 [ForkJoinPool.commonPool-worker-3]
forEach: A2 [ForkJoinPool.commonPool-worker-1]
map:     b1 [main]
forEach: B1 [main]
filter:  a1 [ForkJoinPool.commonPool-worker-3]
map:     a1 [ForkJoinPool.commonPool-worker-3]
```

```
forEach: A1 [ForkJoinPool.commonPool-worker-3]
forEach: C1 [ForkJoinPool.commonPool-worker-2]
```

As you can see the parallel stream utilizes all available threads from the common `ForkJoinPool` for executing the stream operations. The output may differ in consecutive runs because the behavior which particular thread is actually used is non-deterministic.

Let's extend the example by an additional stream operation, `sort`:

```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
    .parallelStream()
    .filter(s -> {
        System.out.format("filter: %s [%s]\n",
            s, Thread.currentThread().getName());
        return true;
    })
    .map(s -> {
        System.out.format("map: %s [%s]\n",
            s, Thread.currentThread().getName());
        return s.toUpperCase();
    })
    .sorted((s1, s2) -> {
        System.out.format("sort: %s <> %s [%s]\n",
            s1, s2, Thread.currentThread().getName());
        return s1.compareTo(s2);
    })
    .forEach(s -> System.out.format("forEach: %s [%s]\n",
        s, Thread.currentThread().getName()));
```

The result may look strange at first:

```
filter:  c2 [ForkJoinPool.commonPool-worker-3]
filter:  c1 [ForkJoinPool.commonPool-worker-2]
map:     c1 [ForkJoinPool.commonPool-worker-2]
filter:  a2 [ForkJoinPool.commonPool-worker-1]
map:     a2 [ForkJoinPool.commonPool-worker-1]
filter:  b1 [main]
map:     b1 [main]
filter:  a1 [ForkJoinPool.commonPool-worker-2]
map:     a1 [ForkJoinPool.commonPool-worker-2]
map:     c2 [ForkJoinPool.commonPool-worker-3]
sort:    A2 <> A1 [main]
sort:    B1 <> A2 [main]
sort:    C2 <> B1 [main]
sort:    C1 <> C2 [main]
sort:    C1 <> B1 [main]
```

```
sort:    C1 <> C2 [main]
forEach: A1 [ForkJoinPool.commonPool-worker-1]
forEach: C2 [ForkJoinPool.commonPool-worker-3]
forEach: B1 [main]
forEach: A2 [ForkJoinPool.commonPool-worker-2]
forEach: C1 [ForkJoinPool.commonPool-worker-1]
```

It seems that `sort` is executed sequentially on the main thread only. Actually, `sort` on a parallel stream uses the new Java 8 method `Arrays.parallelSort()` under the hood. As stated in Javadoc this method decides on the length of the array if sorting will be performed sequentially or in parallel:

> *If the length of the specified array is less than the minimum granularity, then it is sorted using the appropriate Arrays.sort method.*

Coming back to the `reduce` example from the last section. We already found out that the combiner function is only called in parallel but not in sequential streams. Let's see which threads are actually involved:

```
List<Person> persons = Arrays.asList(
    new Person("Max", 18),
    new Person("Peter", 23),
    new Person("Pamela", 23),
    new Person("David", 12));

persons
    .parallelStream()
    .reduce(0,
        (sum, p) -> {
            System.out.format("accumulator: sum=%s; person=%s [%s]\n",
                sum, p, Thread.currentThread().getName());
            return sum += p.age;
        },
        (sum1, sum2) -> {
            System.out.format("combiner: sum1=%s; sum2=%s [%s]\n",
                sum1, sum2, Thread.currentThread().getName());
            return sum1 + sum2;
        });
```

The console output reveals that both the *accumulator* and the *combiner* functions are executed in parallel on all available threads:

```
accumulator: sum=0; person=Pamela; [main]
accumulator: sum=0; person=Max;    [ForkJoinPool.commonPool-worker-3]
accumulator: sum=0; person=David;  [ForkJoinPool.commonPool-worker-2]
```

```
accumulator: sum=0; person=Peter;  [ForkJoinPool.commonPool-worker-1]
combiner:    sum1=18; sum2=23;     [ForkJoinPool.commonPool-worker-1]
combiner:    sum1=23; sum2=12;     [ForkJoinPool.commonPool-worker-2]
combiner:    sum1=41; sum2=35;     [ForkJoinPool.commonPool-worker-2]
```

In summary, it can be stated that parallel streams can bring be a nice performance boost to streams with a large amount of input elements. But keep in mind that some parallel stream operations like `reduce` and `collect` need additional computations (combine operations) which isn't needed when executed sequentially.

Furthermore we've learned that all parallel stream operations share the same JVM-wide common `ForkJoinPool`. So you probably want to avoid implementing slow blocking stream operations since that could potentially slow down other parts of your application which rely heavily on parallel streams.

That's it

My programming guide to Java 8 streams ends here. If you're interested in learning more about Java 8 streams, I recommend to you the Stream Javadoc package documentation. If you want to learn more about the underlying mechanisms, you probably want to read Martin Fowlers article about Collection Pipelines.

If you're interested in JavaScript as well, you may want to have a look at Stream.js - a JavaScript implementation of the Java 8 Streams API. You may also wanna read my Java 8 Tutorial and my Java 8 Nashorn Tutorial.

Hopefully this tutorial was helpful to you and you've enjoyed reading it. The full source code of the tutorial samples is hosted on GitHub. Feel free to fork the repository or send me your feedback via Twitter.

Happy coding!

Benjamin is Software Engineer, Full Stack Developer at Pondus, an excited runner and table foosball player. Get in touch on Twitter and GitHub.

Read More

Recent | All Posts | Java | Tutorials

Integrating React.js into Existing jQuery Web Applications

Java 8 Concurrency Tutorial: Atomic Variables and ConcurrentMap

Java 8 Concurrency Tutorial: Synchronization and Locks

Java 8 Concurrency Tutorial: Threads and Executors