JλVλ 8 (A Comprehensive Look): Part 1.2 - Insight Into Streams

by Robin Rizvi · Apr. 13, 18 · Java Zone · Tutorial

Note: This article dives into the details of Java 8 Streams and its "under the hood" implementation in Java. This is a continuation to JλVλ 8: A Comprehensive Look article, where Java 8 Streams are discussed. So its recommended to first go through the discussion on Java 8 Streams here.

You can also find Part 1.1, which discusses Lambdas, here.

Filter-Map-Reduce Through Streams

Filter-map-reduce is a pattern and facility available in most programming languages and it allows developers to **perform bulk data operations on collections of data**.

- **Filter** simply means to filter out some data that should not be processed through the pipeline.

- **Map** simply means to map the input to some other output. There is another intermediate operation, **flatMap()**, that allows you to flatten stream elements if they are some sort of collections themselves that are required for individual processing, e.g: { {1,2}, {3,4}, {5,6} } -> flatMap -> {1,2,3,4,5,6}. `FlatMap` returns a stream of elements flattened from an original stream element and replaces the original stream element with the stream of elements returned.

- **Reduce** simply means to apply reduction to the stream of values, i.e. reduce them to a single result. As a most common terminal operation in streams, **reduce(...) and collect(...)** are used.

Reduce and Collect in Streams

Both of them are terminal operation in Stream API. The primary differences between them are:

- **Collect is an aggregation operation** whereas **Reduce is a fold operation**, i.e. it folds the elements to reduce them to a single result. It applies a binary operator to each element of the stream where the first argument is the return value of the execution from the previous iteration and the second is the current stream element.

- **Collect is a mutable reduction operation**, i.e., the reduced value is a mutable container in which the results are added by updating the state of this container (reduce value) whereas, in **reduce, the result is replaced** as a consequence of the fold operation.

- **The Collect operation always returns a mutated state** of the initially supplied value for each iteration, whereas the **Reduce operation always returns a new value.**

Reduce Method:

Optional<T> reduce(BinaryOperator<T> accumulator): No identity/initial value is supplied to it, an accumulator is executed on the stream elements. **Accumulator** accepts two arguments of stream element type and returns an element (reduced value) of the stream element type. The first argument is the result of the accumulator's execution from the previous iteration and the second argument is the current stream element. The reduce operation **returns an Optional** since no identity (initial/default) value is specified.

For example:

```
System.out.println(
    Stream.of(1, 2, 3, 4, 5).reduce((a, b) -> {
      System.out.println("In accumulator: " + a + ", " + b);
      return a + b;
}).get());
```

Output:

```
In accumulator: 1, 2
In accumulator: 3, 3
In accumulator: 6, 4
In accumulator: 10, 5
15
```

T reduce(T identity, BinaryOperator<T> accumulator): An **identity**/initial value of stream element type is supplied to it and **accumulator** is applied in the same way as described above. The reduce operation **returns the reduced value**.

For example:

```
System.out.println(
    Stream.of(1, 2, 3, 4, 5).reduce(0, (a, b) -> {
      System.out.println("In accumulator: " + a + ", " + b);
      return a + b;
}));
```

Output:

```
In accumulator: 0, 1
In accumulator: 1, 2
In accumulator: 3, 3
In accumulator: 6, 4
In accumulator: 10, 5
```

15

U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner): An **identity**/initial value of any type can be supplied to it, an **accumulator** is executed on the stream elements. Accumulator accepts two arguments - one of the same type as the identity value and other of stream element type and returns an element (reduced value) of the same type as the identity value. The first argument is the result of the accumulator's execution from the previous iteration and the second argument is the current stream element. A **combiner has to be specified which is executed when a stream is executed in parallel, it combines the reduced values from parallel executions.** The reduce operation returns the reduced value of the same type as the identity value.

For example:

```
System.out.println(
    Stream.of(1, 2, 3, 4, 5).reduce(0, (a, b) -> {
      System.out.println("In accumulator: " + a + ", " + b);
      return a + b;
}, (a, b) -> {
  System.out.println("In combiner: " + a + ", " + b);
  return a + b;
}));
```

Output:

```
In accumulator: 0, 1
In accumulator: 1, 2
In accumulator: 3, 3
In accumulator: 6, 4
In accumulator: 10, 5
15
```

```
System.out.println(
    Stream.of(1, 2, 3, 4, 5).parallel().reduce(0, (a, b) -> {
      System.out.println("In accumulator: " + a + ", " + b);
      return a + b;
}, (a, b) -> {
  System.out.println("In combiner: " + a + ", " + b);
  return a + b;
}));
```

Output:

```
In accumulator: 0, 3
In accumulator: 0, 5
In accumulator: 0, 4
In combiner: 4, 5
In combiner: 3, 9
In accumulator: 0, 2
In accumulator: 0, 1
In combiner: 1, 2
```

In combiner: 3, 12
15

Collect Method:

R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner): A **supplier** has to be defined which serves as the container that will be used to collect the values. An **accumulator** has to be specified which accumulates the results into the container specified in the supplier, it accepts two arguments - the first one is the supplied container and the other one is the current stream element. A **combiner** also has to be specified which is executed when a stream is executed in parallel, it combines the collected values from parallel executions. The collect operation **returns the container with all the values collected/accumulated in it**.

For example:

```
List<String> x1 = Stream.of("R", "O", "B", "I", "N")
.collect(() -> new ArrayList<>(), (a, e) -> {
  System.out.println("In accumulator: " + a + ", " + e);
  a.add(e.toLowerCase());
}, (a, e) -> {
  System.out.println("In combiner: " + a + ", " + e);
  a.addAll(e);
});
```

Output:

In accumulator: [], R
In accumulator: [r], O
In accumulator: [r, o], B
In accumulator: [r, o, b], I
In accumulator: [r, o, b, i], N

```
List<String> x2 = Stream.of("R", "O", "B", "I", "N").parallel()
.collect(() -> new ArrayList<>(), (a, e) -> {
  System.out.println("In accumulator: " + a + ", " + e);
  a.add(e.toLowerCase());
}, (a, e) -> {
  System.out.println("In combiner: " + a + ", " + e);
  a.addAll(e);
});
```

Output:

In accumulator: [], B
In accumulator: [], O
In accumulator: [], R
In accumulator: [], N
In accumulator: [], I
In combiner: [r], [o]
In combiner: [i], [n]
In combiner: [b], [i, n]

In combiner: [r, o], [b, i, n]

R collect(Collector<? super T,A,R> collector): A **collector** has to be specified which serves the same purpose as the above overload. There are different pre-built collectors already present in the JDK library which can be used as is or we can also create our own collector by implementing one or using **static factory methods (of(...)) from the Collector class** supplying it with supplier, accumulator, combiner, and stream characteristics.
The factory methods in Collector class to create collectors are:

```
Collector.of(Supplier supplier, BiConsumer accumulator, BinaryOperator combiner, Characteri
Collector.of(Supplier supplier, BiConsumer accumulator, BinaryOperator combiner, Function f
```

You can have a look at the **different collectors available** in the Oracle docs.

Streams Execution Order

Streams are evaluated lazily and the stream is not traversed until the terminal operation of the stream is executed. Also, streams are always **traversed vertically and not horizontally** (when stateless operations are used, stateful and stateless operations are discussed later in the article) i.e., each element from source is processed **through the pipeline from source to terminal level** and then the next element is taken up for processing. Let's have a look at some examples:

```
Stream.of("R", "O", "B")
.peek(System.out::print)
.forEach(System.out::print);
```

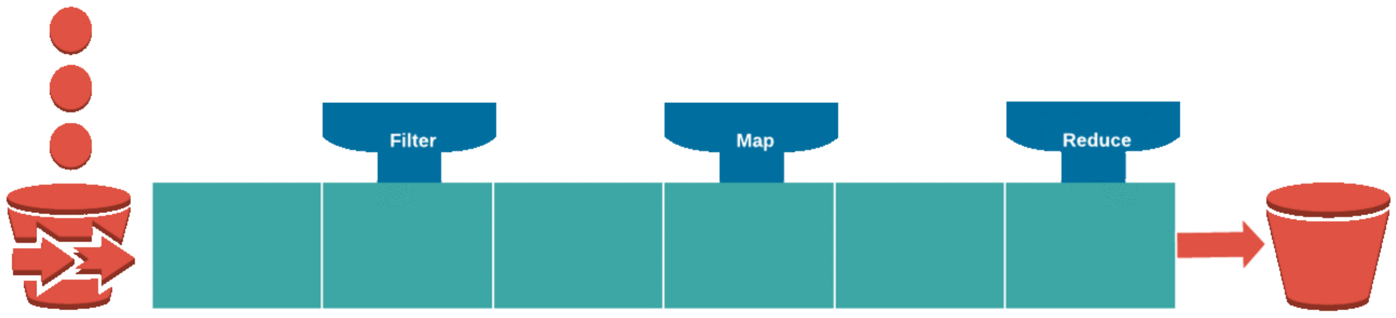
The output is:

RROOBB (because the source was traversed vertically)
And not ROBROB (if the source was traversed horizontally)

```
Stream.iterate(0, i -> i + 1) // infinite stream
.peek(System.out::println)
.findFirst();
```

In spite of being an infinite stream, the above operation completes almost immediately because of the lazy characteristic of Stream and its vertical traversal.

Let's see this vertical traversal of streams works, through a visualization:



Keeping the vertical traversal aspect of streams in view can help to **optimize the performance of stream execution by ordering the operations sensibly** in a stream pipeline. Let's see through an example how the number of operations that are executed can decrease/increase based on the ordering of operations in the pipeline:

```
Stream.of("R", "O", "B")
.map(e -> {
  System.out.println("");
  System.out.print("In map: " + e +" -> ");
  return e.toLowerCase();
})
.filter(e-> {
  System.out.print("In filter: " + e + " -> ");
  return e.equalsIgnoreCase("R");
 })
.forEach(e -> {
  System.out.print("In forEach: " + e);
});
```

Output:

```
In map: R -> In filter: r -> In forEach: r
In map: O -> In filter: o ->
In map: B -> In filter: b ->
```

In this example, **7 operations** in total in the pipeline are executed

```
Stream.of("R", "O", "B")
.filter(e-> {
  System.out.println("");
  System.out.print("In filter: " + e + " -> ");
  return e.equalsIgnoreCase("R");
 })
.map(e -> {
  System.out.print("In map: " + e +" -> ");
  return e.toLowerCase();
})
.forEach(e -> {
  System.out.print("In forEach: " + e);
});
```

Output:

```
In filter: R -> In map: R -> In forEach: r
In filter: O ->
In filter: B ->
```

This is the same example as above with the same output, the **difference being that the order of operations is changed**, i.e. filter and map are interchanged. Here, **only 5 operations** are executed in total.

Since each element goes through the pipeline of operations vertically, we should **apply the operations that restrict or decrease the contents that should be processed, to the beginning of the pipeline** so that we are not unnecessarily executing operations on elements that are not to be included in the result.

Stateful and Stateless Stream Operations and Functions

A stateful function/operation is one whose result depends on any state that might change during the execution of the stream. Such a function in terms of functional programming would be seen as an **un-pure function since it is depending or mutating some state**.

Java Streams API has **some intermediate operations that are stateful**, they maintain some state internally to achieve their purpose. Those intermediate operations are:

- distinct()
- sorted()
- limit()
- skip()

These operations are stateful in the sense that they keep the state from previously processed elements while processing current elements. **Since they just use this state internally, they are deterministic and safe in nature** and don't have adverse side effects.

When **Java Stream's stateful intermediate operations are used, they process the stream horizontally** instead of vertically, as an example, think about the `sorted()` intermediate operation - if it would process stream elements vertically it would not able to sort the elements, so it needs to depend on the state of already processed elements to be able to achieve sorting functionality.This is the case with other stateful intermediate operations as well. Let's see an example that demonstrates this horizontal execution order when stateful operations are used:

```
Stream.of("R", "O", "B")
.sorted((a,b)->{
  System.out.println("sorted: " + b + a);
  return a.compareTo(b);
})
.forEach(e -> System.out.println("forEach: " + e));
```

Output:

sorted: RO
sorted: OB
forEach: B
forEach: O
forEach: R

Since `sorted` is a stateful operation, here the stream elements are processed horizontally instead of vertically. Let's compare it with a stateless operation and see how the stream elements are processed vertically instead:

```
Stream.of("R", "O", "B")
```

```
.map(a->{
  System.out.println("map: " + a);
  return a;
})
.forEach(e -> System.out.println("forEach: " + e));
```

Output:

```
map: R
forEach: R
map: O
forEach: O
map: B
forEach: B
```

Here, instead of sorted (stateful operation), the `map` (stateless) operation is used, and you can see from the output that stream elements are processed vertically in this case.

Stateful Functions

You need to be very careful to avoid passing any stateful function (lambda expression/ method reference) as a parameter to any stream operations. **A stateful function (un-pure function) is one that depends on any state or mutates state**, on the other hand, a stateless function does not depend on or mutate any state and is a pure function. If a stateful function is used in a steam pipeline, **it may lead to non-deterministic results** (different results for a specific input), especially **evident in case of parallel pipelines**. Also, if an external state is accessed/mutated we need to take care of synchronization (thread-safety) in such cases. Using a **stateful function dissolves all our parallelism advantage** that is inherent with streams. Let's see how this affects the code through an example:

```
for (int i = 0; i < 5; i++) {
  Set<String> stringSet = new HashSet<>();
  System.out.println(Stream.of("R", "B", "I", "N", "O", "B", "I", "N", "I", "N")
    .parallel()
    .map(
        // stateful function
        e -> {
          if (stringSet.add(e))
            return e.toLowerCase();
          else
            return "";
        })
    .collect(Collectors.joining()));
}
```

Output:

```
rbioiin
robin
robin
rnobi
```

rnobi

As you can see, this stream pipeline is **executed 5 times and the results are not consistent** and are non-deterministic due to the usage of a stateful function in the `map` operation. If we replace this stateful function with a stateless one we can see that we will not suffer from this problem:

```
for (int i = 0; i < 5; i++) {
  Set<String> stringSet = new HashSet<>();
  System.out.println(Stream.of("R", "B", "I", "N", "O", "B", "I", "N", "I", "N")
    .parallel()
    .map(
        // stateless function
        e -> {
          return e.toLowerCase();
        })
    .collect(Collectors.joining()));
}
```

Output:

rbinobinin
rbinobinin
rbinobinin
rbinobinin
rbinobinin

As you can see, since a stateless function is used in a `map` operation, the output of the stream execution is consistent.

Streams Under the Hood:

Let's have a peek under the hood to understand how Streams API in Java is implemented and how it executes. This will help us understand how the features like lazy evaluation, vertical and horizontal execution, etc., are achieved. **Having some insights into the implementation helps to easily identify bugs and bottlenecks, and helps to optimize performance and construct efficient pipelines** with Streams API.

Streams API in Java **follows a fluent API design** to construct a data processing pipeline. The data source constructors/generators and intermediate operations in the Streams API return a stream type themselves which allows linking more operations to the pipeline in a fluent manner.

As we have already discussed, Streams pipeline is composed of 3 levels - a single data source, zero or more intermediate operations, and a single terminal operation. Let's start understanding the source step in the context of how streams are evaluated at runtime. The **source is expressed through a Spliterator** during stream evaluation. Spliterator allows you to iterate and access the elements of the data source. From a usage point of view, spliterator serves the same purpose that an Iterator does, it's just that they differ in their implementation. **Spliterator can be loosely seen as a better version of the Iterator**, because if we compare how elements are accessed in an Iterator vs a Spliterator, we can see that to access an element in an Iterator we have to perform the two-step process of `hasNext()` and `next()`, whereas Spliterator allows you to access the elements in a more direct way. **Primary methods in a Spliterator are:**

- **boolean tryAdvance(Consumer<? super T> action):** If any element is remaining, it executes the

- **boolean tryAdvance(Consumer<? super T> action):** If any element is remaining, it executes the specified action on it and returns true, otherwise, it returns false

- **void forEachRemaining(Consumer<? super T> action):** Executes the specified action on all remaining elements sequentially.

- **Spliterator<T> trySplit():** Splits the elements and represents the split elements in a new Spliterator and returns that Spliterator. The elements represented by the returned Spliterator are then removed and not represented by the original Spliterator. This is used for partitioning the elements for parallel stream execution.

- **int characteristics():** Returns a bitmap that represents the characteristics (ORDERED, DISTINCT, SORTED, SIZED, etc.) of the spliterator and it's elements.

An important part to understand in the Streams API is the **usage of characteristic bitmap flags** that are **used during stream evaluation to optimize** the execution of the stream. Characteristics bitmap flags **represent the characteristics of the elements of the stream at any step** of the pipeline. For each step (source creation, intermediate operations, terminal operation) in the pipeline, the characteristics from the previous step are combined to form suitable characteristics for the current step.

Let's follow an example to understand this better:

```
list.stream().sorted().filter(e -> true).map(e -> e).forEach(System.out::println)
```

At the source step, Spliterator is representing the data source that defines the characteristics of the elements (there is a default `spliterator()` implementation in the Collection interface but it is overridden by sub-classes that return a more specialized and efficient Spliterator). If we have a **look at the java.util.ArrayList** class, we can see that it contains a static inner class **ArrayListSpliterator<E> implements Spliterator<E>**, which overrides the `characteristics()` method as below:

```
public int characteristics() {
  return Spliterator.ORDERED | Spliterator.SIZED | Spliterator.SUBSIZED;
}
```

It indicates the characteristics of this collection:

- Spliterator.ORDERED - the elements of the array list are ordered.

- Spliterator.SIZED - the array list has a definite size.

- Spliterator.SUBSIZED - all the Spliterators returned after `trySplit` representing a section of this array list will also be sized.

Let's break down the example above to see characteristics at each step:

- **list.stream()** - Spliterator.ORDERED | Spliterator.SIZED | Spliterator.SUBSIZED

- **list.stream().sorted()** - Spliterator.ORDERED | Spliterator.SIZED | Spliterator.SUBSIZED |

Spliterator.SORTED

- **list.stream().sorted().filter(e -> true)** - Spliterator.ORDERED | Spliterator.SORTED

- **list.stream().sorted().filter(e -> true).map(e -> e)** - Spliterator.ORDERED
  After the source step, we have applied a sorted operation, which will preverse all three characteristics and also add a SORTED flag to the list, then we applied the filter operation/step, which will preserve the ORDERED flag but will clear the SIZED and SUBSIZED flags. And then a map operation is applied which will preesrve the ordered flag but clear the SORTED flag.

During the execution of the stream, the information gathered from the characteristics at each step **could be utilized to optimize** the stream execution, e.g., **some operations in the stream pipeline could even be skipped.**

For example:

```
HashSet<String> stringSet = new HashSet<>();
stringSet.stream().distinct().forEach(System.out::println);
```

Since `HashSet` 's Spliterator already defines the DISTINCT flag as it's characteristic, the `distinct()` operation would be skipped during stream evaluation.

Now, let's continue to understanding the streams execution process. For easy comprehension, you can view a stream setup in your code **as a linked list** (with a **HEAD node** representing the data source, linked to other **intermediate nodes** which represent the operations through which each element of the data source is passed, and then at the end of this linked list is the **terminal node** which actually computes the final result for you). Let's see how streams execute at runtime through an example:

```
ArrayList<String> al = new ArrayList<>();
al.add("R");
al.add("O");
al.add("B");

al.stream()
  .filter(e -> e.equalsIgnoreCase("O"))
  .sorted()
  .map(e -> "In lowercase: " + e.toLowerCase())
  .forEach(System.out::println);
```

Output: In lowercase: o

1. When the **`stream()` method** is executed on the `ArrayList` instance, it calls the **`Collection.stream()`** default method and returns an object of Stream type

```
default Stream<E> stream() {
  return StreamSupport.stream(spliterator(), false);
}
```

The `spliterator()` method is overridden in the `ArrayList` class, which returns an instance of **ArrayListSpliterator**<E> and implements the Spliterator<E> class.

The `StreamSupport.stream(...)` method returns an instance of the `ReferencePipeline.Head` class which is an implementation of Stream type. This represents the head node of the stream's pipeline. This **head node stores the information about the source, spliterator,** which is provided to it.

```
public static <T> Stream<T> stream(Spliterator<T> spliterator, boolean parallel) {
  Objects.requireNonNull(spliterator);
  return new ReferencePipeline.Head<>(spliterator, StreamOpFlag.fromCharacteristics(spliter
}
```

2. Next, the **`filter()` method** is executed on the head node returned in Step#1. This calls the `ReferencePipeline.filter(...)` method which returns an instance of the `ReferencePipe.StatelessOp` class (in fact, an instance of an anonymous class which extends `ReferencePipe.StatelessOp`) which is an implementation of the Stream type. This represents an **intermediate node which is a stateless** stream operation in the stream's pipeline. This intermediate node sets the characteristics at this step based on the stream operation, sets the source node (which is the head node from Step#1) which is used to get the spliterator from it in the terminal operation (to iterate and pass the elements through the stream) and also **establishes the link between the previous node and itself** (previousStage.nextStage = this; this.previousStage = previousStage;)

```
public final Stream<P_OUT> filter(Predicate<? super P_OUT> predicate) {
    Objects.requireNonNull(predicate);
    return new StatelessOp<P_OUT, P_OUT>(this, StreamShape.REFERENCE, StreamOpFlag.NOT_SIZE
        @Override
        Sink<P_OUT> opWrapSink(int flags, Sink<P_OUT> sink) {
            return new Sink.ChainedReference<P_OUT, P_OUT>(sink) {
                @Override
                public void begin(long size) {
                    downstream.begin(-1);
                }
                @Override
                public void accept(P_OUT u) {
                    if (predicate.test(u))
                        downstream.accept(u);
                }
            };
        }
    };
}
```

3. Next, the **`sorted()` method** is executed on the intermediate node returned in Step#2. This calls the `ReferencePipeline.sorted(...)` method which returns an instance of the `ReferencePipe.StatefulOp` class which is an implementation of the Stream type. This represents an **intermediate node which is a stateful** stream operation in the stream's pipeline. This intermediate node sets the characteristics at this step based on the stream operation, sets the source node (which is the head node from Step#1) which is used to get the spliterator from it in the terminal operation (to iterate and pass the elements through the stream) and also **establishes the link** between the previous node and itself (previousStage.nextStage = this; this.previousStage = previousStage;)

```
 @Override
public final Stream<P_OUT> sorted() {
  return SortedOps.makeRef(this);
}
```

4. Next, the **`map()` method** is executed on the intermediate node returned in Step#3. This calls the `ReferencePipeline.map(...)` method which returns a Stream object, sets characteristics, sets the head node, and establishes links. It is built **similarly to the filter method in Step#2**.

```
public final <R> Stream<R> map(Function<? super P_OUT, ? extends R> mapper) {
    Objects.requireNonNull(mapper);
    return new StatelessOp<P_OUT, R>(this, StreamShape.REFERENCE,
                                 StreamOpFlag.NOT_SORTED | StreamOpFlag.NOT_DISTINCT) {
        @Override
        Sink<P_OUT> opWrapSink(int flags, Sink<R> sink) {
            return new Sink.ChainedReference<P_OUT, R>(sink) {
                @Override
                public void accept(P_OUT u) {
                    downstream.accept(mapper.apply(u));
                }
            };
        }
    };
}
```

5. Next, the **`forEach()` method** is executed on the intermediate node returned in Step#4. This calls the `ReferencePipeline.forEach(...)` method which **does not return a Stream object** because forEach is a terminal operation. Most of the methods representing terminal operations, in turn, **create an object of TerminalOp type and execute the evaluate(TerminalOp<E_OUT, R> terminalOp) method**.

```
public void forEach(Consumer<? super P_OUT> action) {
  evaluate(ForEachOps.makeRef(action, false));
}
```

The TerminalOp type exposes two main methods - **`evaluateParallel` and `evaluateSequential` -** and ,in our case, the evaluate method, in turn, calls the `evaluateSequential` method on the created TerminalOp Object.

```
final <R> R evaluate(TerminalOp<E_OUT, R> terminalOp) {
    assert getOutputShape() == terminalOp.inputShape();
    if (linkedOrConsumed)
        throw new IllegalStateException(MSG_STREAM_LINKED);
    linkedOrConsumed = true;

    return isParallel()
           ? terminalOp.evaluateParallel(this, sourceSpliterator(terminalOp.getOpFlags()))
           : terminalOp.evaluateSequential(this, sourceSpliterator(terminalOp.getOpFlags())
}
```

The **`evaluateSequential` method is responsible for creating a wrapped sink and passing stream elements through it**. In the current context, you can think of sink as a kitchen sink where each of the stream elements is dropped one by one. The sink here is, in fact, a **Sink type which extends the Consumer type,** i.e., it accepts an element of the stream and performs an operation on it. A **wrapped sink is one which wraps another sink** into it. So what really happens is that each node in the stream creates a sink which wraps a sink which is next to it in the stream pipeline, so this **creates a top-level sink for the first intermediate node**

in the pipeline which wraps the next sink and this process goes on until it reaches the terminal node. The following method creates a wrapped sink. This method is passed with the terminal node which is of type Sink, which is then wrapped into the previous intermediate node's sink.

```
final <P_IN> Sink<P_IN> wrapSink(Sink<E_OUT> sink) {
    Objects.requireNonNull(sink);

    for ( @SuppressWarnings("rawtypes") AbstractPipeline p=AbstractPipeline.this; p.depth >
        sink = p.opWrapSink(p.previousStage.combinedFlags, sink);
    }
    return (Sink<P_IN>) sink;
}
```

Each of the **intermediate nodes of Steam types have this `opWrapSink` method to allow wrapping sinks**. I will pull out the example of `map()`, but filter is also similar to this, and you can have a look at the code for `filter()` above:

```
public final <R> Stream<R> map(Function<? super P_OUT, ? extends R> mapper) {
    Objects.requireNonNull(mapper);
    return new StatelessOp<P_OUT, R>(this, StreamShape.REFERENCE,
                                 StreamOpFlag.NOT_SORTED | StreamOpFlag.NOT_DISTINCT) {
        @Override
        Sink<P_OUT> opWrapSink(int flags, Sink<R> sink) {
            return new Sink.ChainedReference<P_OUT, R>(sink) {
                @Override
                public void accept(P_OUT u) {
                    downstream.accept(mapper.apply(u));
                }
            };
        }
    };
}
```

Have a look at the `opWrapSink` method above. It creates a new Sink which wraps the sink passed in arguments. Also, have a look at the **accept method, this is where the actual operation logic is written** which gets executed for an element of the stream. In the accept method, once the operation's own logic has been executed (mapper.apply(u)), **it calls the accept method of the sink it has wrapped (downstream.accept(...))** to execute the next operation in the pipeline.

So, once this wrapping is done and we get the top level sink which represents the first operation in the pipeline, then **each element is passed through this top-level sink using the source `spliterator`**.

```
final <P_IN> void copyInto(Sink<P_IN> wrappedSink, Spliterator<P_IN> spliterator) {
    Objects.requireNonNull(wrappedSink);

    if (!StreamOpFlag.SHORT_CIRCUIT.isKnown(getStreamAndOpFlags())) {
        wrappedSink.begin(spliterator.getExactSizeIfKnown());
        spliterator.forEachRemaining(wrappedSink);
        wrappedSink.end();
    }
    else {
        copyIntoWithCancel(wrappedSink, spliterator);
    }
}
```

Here, the **spliterator.forEachRemaining(Consumer<? super T> action)** method, calls the `accept` method on the top level sink for each element sequentially and sink(s) in turn may call the next sink in the pipeline.

Note: **`tryAdvance`** in a loop is called in case of **short-circuiting terminal operations** (the operations that can break out in between without processing all remaining elements, like `findFirst` , `anyMatch` , etc.).

Let's get back to our original example and see how the pipeline is executed. The top-level sink would be the one that is returned when the `opWrapSink` method was executed on the Stream object returned for the filter operation. So when `spliterator.forEachRemaining` is executed, it will call the accept method of this sink for each element:

```
public void accept(P_OUT u) {
  if (predicate.test(u))
    downstream.accept(u);
}
```

After testing the element through the predicate, if it satisfies the predicate then the next sink's `accept` method is executed. The next downstream sink in our case is the one that is returned when the `opWrapSink` method was executed on the Stream object returned for the sorted operation. Since this **operation is stateful, it does not directly process and pass the element on to the next sink, but, instead, it collects all the elements** into its buffer (see the `accept` method below) and then does its process and then passes on the elements sequentially to downstream sinks.

```
@Override
public void begin(long size) {
    if (size >= Nodes.MAX_ARRAY_SIZE)
        throw new IllegalArgumentException(Nodes.BAD_SIZE);
    array = (T[]) new Object[(int) size];
}

@Override
public void end() {
    Arrays.sort(array, 0, offset, comparator);
    downstream.begin(offset);
    if (!cancellationWasRequested) {
        for (int i = 0; i < offset; i++)
            downstream.accept(array[i]);
    }
    else {
        for (int i = 0; i < offset && !downstream.cancellationRequested(); i++)
            downstream.accept(array[i]);
    }
    downstream.end();
```

```
        array = null;
    }

    @Override
    public void accept(T t) {
        array[offset++] = t;
    }
```

After the **sorting process is complete, the next downstream sink's `accept` method** is executed. The next downstream sink, in our case, is the one that is returned when the `opWrapSink` method was executed on the Stream object returned for the map operation.

```
public void accept(P_OUT u) {
    downstream.accept(mapper.apply(u));
}
```

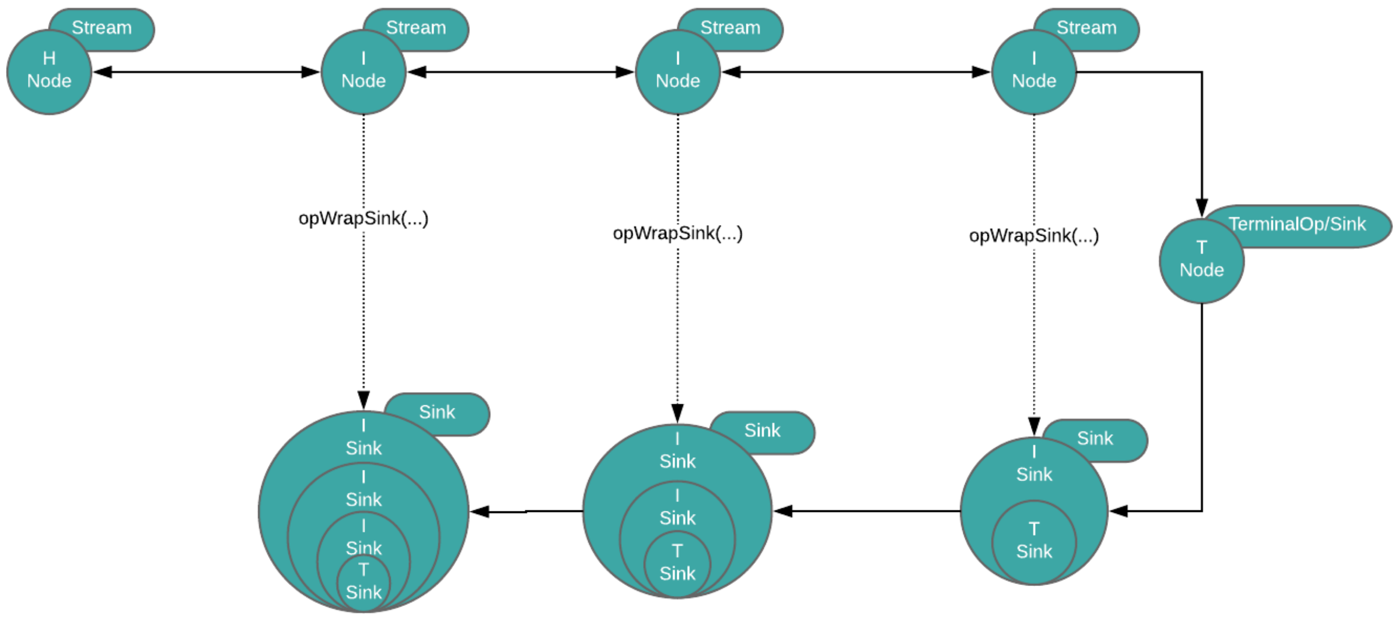
`mapper.apply(u)` is executed, which performs the map operation and then the next downstream sink's `accept` method is executed. The **next downstream sink, in our case, is the TerminalOp forEach**.

```
public void accept(T t) {
    consumer.accept(t);
}
```

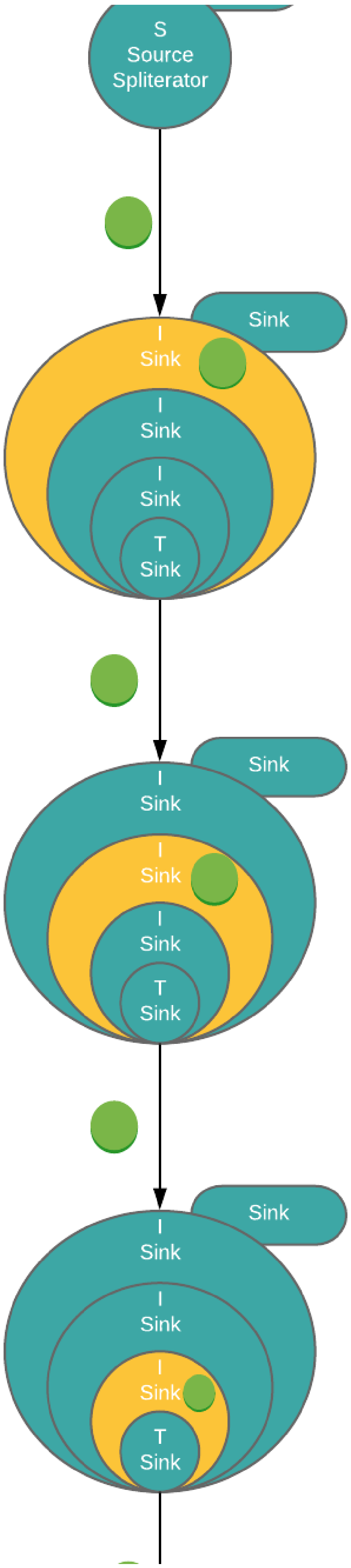
Which, finally, in our case, just prints the elements.

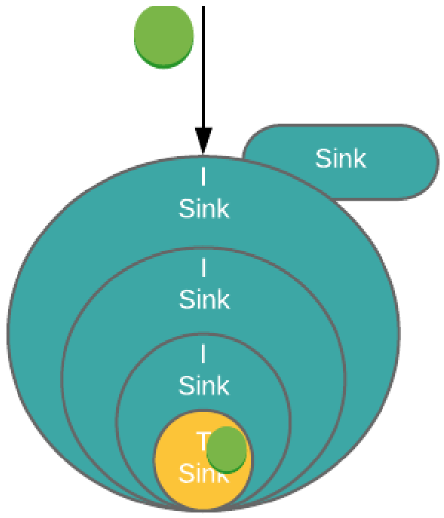
Let's see through a diagram the stream creation and evaluation/execution process:

Stream Creation:



Stream Evaluation:


S
Source
Spliterator
Sink
I
Sink
I
Sink
I
Sink
T
Sink
Sink
I
Sink
I
Sink
I
Sink
T
Sink
Sink
I
Sink
I
Sink
I
Sink
T
Sink



Like This Article? Read More From DZone

Topics: STREAMS , STREAM API , JAVA 8 , FUNCTIONAL PROGRAMMING , JAVA