Contents

Streams

Streams are one of the two most notable features in `JAVA 8`, the second being the lambda function. The lambda functional interface however enables the existence of the Streams API. Streams in java are types of objects which are not meant to hold any data, they also operate on data structures such as Lists, Maps etc. However they do not modify that structure in any way, in case the stream has to produce a result which represents an aggregate or otherwise a filtered version of the original data structure, the stream would produce a completely new instance of the structure such that it contains the relevant elements but will never modify the source structure.

`Streams provide a wrapper API around common data structures, to allow certain operations such as filtering, searching, ordering, aggregation, combining, grouping and many more data transformation utilities over the standard java collection utilities`

BaseStream

```java
interface BaseStream<T, S extends BaseStream<T, S>> extends AutoCloseable
   // the signature of the BaseStream API
```

The `BaseStream` is the super interface for all other stream interfaces, including the Stream interface itself, it contains the most core methods that each stream must support, one of which is the `close` and `iterator` methods, which vare most probably the ones that are considered the most important.

Method	Description
iterator()	obtain an iterator to the stream, using the iterator of the underlying structure
spliterator()	obtain an split-iterator to the stream, using the split-iterator of the underlying structure
sequential()	return a sequential stream representation of the source stream, if the stream is already sequential reutrns the same instance
parallel()	return a parallel stream representation of the source stream, if the stream is already parallel reutrns the same instance
unordered()	return an unordered stream representation of the source stream, if the stream is already unordered reutrns the same instance
close()	closes the stream, meaning that no termination operations can be called on the stream any more

Stream

```java
interface Stream<T> extends BaseStream<T, Stream<T>> // the signature of
   the Stream API
```

The Stream interface extends the BaseStream, and is meant to provide the most commonly used aggregation and transformation operations that can be performed on any data structure

Method Signature	Description
`filter(Predicate<? super T> predicate)`	Returns a stream consisting of elements that match the given predicate.
`map(Function<? super T, ? extends R> mapper)`	Transforms each element of the stream using the provided mapping function and returns a new stream of the mapped elements.
`mapToInt(ToIntFunction<? super T> mapper)`	Transforms each element into an `int` and returns an `IntStream`.
`mapToLong(ToLongFunction<? super T> mapper)`	Transforms each element into a `long` and returns a `LongStream`.
`mapToDouble(ToDoubleFunction<? super T> mapper)`	Transforms each element into a `double` and returns a `DoubleStream`.
`flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)`	Flattens a stream of streams into a single stream by replacing each element with the contents of a mapped stream.
`flatMapToInt(Function<? super T, ? extends IntStream> mapper)`	Flattens a stream of streams of integers into an `IntStream`.
`flatMapToLong(Function<? super T, ? extends LongStream> mapper)`	Flattens a stream of streams of longs into a `LongStream`.
`flatMapToDouble(Function<? super T, ? extends DoubleStream> mapper)`	Flattens a stream of streams of doubles into a `DoubleStream`.
`mapMulti(BiConsumer<? super T, ? super Consumer<R>> mapper)`	Applies a function to each element, producing zero or more results per input element and returns a new stream with these results.
`mapMultiToInt(BiConsumer<? super T, ? super IntConsumer> mapper)`	A multi-mapping function that produces an `IntStream`.
`mapMultiToLong(BiConsumer<? super T, ? super LongConsumer> mapper)`	A multi-mapping function that produces a `LongStream`.
`mapMultiToDouble(BiConsumer<? super T, ? super DoubleConsumer> mapper)`	A multi-mapping function that produces a `DoubleStream`.
`distinct()`	Returns a stream with unique elements (no duplicates).
`sorted()`	Returns a stream where the elements are sorted in natural order.
`sorted(Comparator<? super T> comparator)`	Returns a stream where the elements are sorted based on the provided comparator.
`peek(Consumer<? super T> action)`	Returns a stream where an action is performed on each element as they are consumed, mainly for debugging purposes.
`limit(long maxSize)`	Limits the stream to the specified number of elements.
`skip(long n)`	Skips the first `n` elements in the stream and returns the remaining elements.
`takeWhile(Predicate<? super T> predicate)`	Returns a stream consisting of the longest prefix of elements that match the predicate.
`dropWhile(Predicate<? super T> predicate)`	Drops the longest prefix of elements that match the predicate and returns the remaining elements.
`forEach(Consumer<? super T> action)`	Performs the given action for each element of the stream.
`forEachOrdered(Consumer<? super T> action)`	Performs the given action for each element in encounter order, preserving the order of operations.
`toArray()`	Returns an array containing all elements of the stream.

Method Signature	Description
`toArray(IntFunction<A[]> generator)`	Returns an array containing the elements of the stream, with the provided array generator function.
`reduce(T identity, BinaryOperator<T> accumulator)`	Performs a reduction on the elements using an identity value and an accumulator function.
`reduce(BinaryOperator<T> accumulator)`	Performs a reduction on the elements using an accumulator function, without an identity.
`reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)`	Performs a reduction with an identity, an accumulator function, and a combiner for parallel streams.
`collect(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R, R> combiner)`	Performs a mutable reduction (e.g., collects elements into a collection).
`collect(Collector<? super T, A, R> collector)`	Performs a reduction using a `Collector`, which handles accumulation and final value production.
`toList()`	Collects the elements into a `List`.
`min(Comparator<? super T> comparator)`	Finds the minimum element of the stream using the given comparator.
`max(Comparator<? super T> comparator)`	Finds the maximum element of the stream using the given comparator.
`count()`	Returns the count of elements in the stream.
`anyMatch(Predicate<? super T> predicate)`	Returns `true` if any elements match the provided predicate.
`allMatch(Predicate<? super T> predicate)`	Returns `true` if all elements match the provided predicate.
`noneMatch(Predicate<? super T> predicate)`	Returns `true` if no elements match the provided predicate.
`findFirst()`	Returns the first element of the stream, if present.
`findAny()`	Returns any element of the stream, useful in parallel streams.
`builder()`	Returns a `Stream.Builder` to incrementally build a stream.
`empty()`	Returns an empty stream.
`of(T value)`	Returns a stream containing a single element.
`ofNullable(T value)`	Returns a stream containing the provided element if non-null, otherwise returns an empty stream.
`of(T... values)`	Returns a stream of the provided elements.
`iterate(T seed, UnaryOperator<T> f)`	Creates an infinite stream where each next element is generated by applying the function `f` to the previous one, starting with the seed.
`iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> f)`	Generates a stream where each element is produced by applying the function `f`, while the `hasNext` predicate returns `true`.
`generate(Supplier<? extends T> supplier)`	Creates an infinite stream where each element is generated by the provided supplier.
`concat(Stream<? extends T> a, Stream<? extends T> b)`	Concatenates two streams into a single stream.

Now each of these operations are classified in two major groups, either an operation is intermediate or terminal. Intermediate operations are such that they do not close the stream, they do not produce output, and more intermediate or terminal operations can follow them. Terminal operations usually close the stream, no more terminal or intermediate operations can be used after them and the stream is translated into a closed state, the stream is consumed, all intermediate operations up to this point attached or related to the stream instance are executed, a result is produced.

`Intermediate operations always produce another stream instance, usually they return the same/this instance of the stream and not a copy of the stream, while terminal operations produce a result of some sort, or in the case of forEach have no return type. Intermediate operations are not executed immediately after they are attached/intermediate function is called on a stream object, they are executed when a terminal operation on the stream is called.`

Another key point to note about the intermediate operations is that they are not `stateful` in the sense that they operate on elements of the stream or more precisely the underlying data structure independently of each other. However some like `sorted` do actually preserve some sort of state or relation between stream elements, because they have in order for the elements to be sorted. This plays a crucial role in talking about parallel or sequential or unordered stream types

Method Name	Type
filter	Intermediate
map	Intermediate
mapToInt	Intermediate
mapToLong	Intermediate
mapToDouble	Intermediate
flatMap	Intermediate
flatMapToInt	Intermediate
flatMapToLong	Intermediate
flatMapToDouble	Intermediate
mapMulti	Intermediate
mapMultiToInt	Intermediate
mapMultiToLong	Intermediate
mapMultiToDouble	Intermediate
distinct	Intermediate
sorted	Intermediate
sorted(Comparator)	Intermediate
peek	Intermediate
limit	Intermediate
skip	Intermediate
takeWhile	Intermediate
dropWhile	Intermediate
forEach	Terminal
forEachOrdered	Terminal
toArray	Terminal
toArray(IntFunction)	Terminal
reduce	Terminal
reduce(BinaryOperator)	Terminal
collect	Terminal
toList	Terminal
min	Terminal
max	Terminal
count	Terminal
anyMatch	Terminal

Method Name	Type
allMatch	Terminal
noneMatch	Terminal
findFirst	Terminal
findAny	Terminal
iterate	Intermediate
generate	Intermediate
concat	Intermediate
accept (Builder)	Intermediate
add (Builder)	Intermediate
build (Builder)	Terminal

Obtaining

To obtain a stream the most common use case from an instanc of a data structure implementing the `Collection` interface is to call the `stream()` or the `parallel()` methods, keep in mind that if a parallel stream can not be obtained a regular sequential stream will be returned the method will not throw exception

`A stream can be converted into a sequential or parallel at any time based on the use case at hand, each call to sequential or parallel (from BaseStream)creates a new instance of the stream using the source data of the original, it is either of parallel or sequential type`

Mapping

One of the most common operations is to transform a stream of one type of elements into another type. This is called a mapping and usually it accepts a single input argument of the original type and single output argument of the resulting mapped type. That then is applied to all elements

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> result = numbers.parallelStream()
        .map(n -> n * n)        // convert to squares
        .toList();              // collect result
```

Filtering

Another very common operation is to apply a filter on the list of elements, the function reference itself, receives the element from the stream, and returns a `boolean` result, `true` implies the element should be retained in the stream, while `false`, implies that the element can be filtered out / discarded. while

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> result = numbers.parallelStream()
        .filter(n -> n % 2 == 0) // only even numbers
        .toList();              // collect result
```

Collecting

Another common operation is to collect, or convert a stream into a collection, of some sort, that could be a List, Map, Set or any other type of `collection`, even custom user collections, there are no real restrictions on what / how the collection is done, as long as the collection provides implementation against the `Collector` interface. There exists a companion utility class called `Collectors`, which contains common methods which are used for collecting streams into lists, sets, maps etc, with exposed methods such as `toList, toSet, toMap`

ToList One of the most frequently used methods on the Collectors API, it has two major methods which are used to create a list from the unerlying stream.

- `toList` - which basically uses an `ArrayList` as the underlying implementation, and adds all elements from the stream to the list, remember that the elements themselves are added by reference, the only thing that is getting `discarded` is actually the original data structure, which the stream wrapped around in the first place during its creation
- `toUnmodifiableList` - that is the same implementation as above, still usihng an `ArrayList` to collect the elements, however, at the end the elements are moved from the mutable `ArrayList` into a unmutable list, using the `List.of` API which collects the elements of the source array into a unmodifiable list - using the internal jdk implemented type `ImmutableCollections`

ToSet This one works similarly to the `toList` interface from the collectors API, it does again provide two methods, one to produce a mutable `set` and another one to produce an immutable one. Works by internally storing into a `HashSet`, and the immutable version is using `Set.of`

ToMap The toMap method has a lot of overloads, providing different means of collecting elements into a map, mostly the overloaded methods deal with `key` handling, and `collisions`.

The most basic usage of `toMap`, simply takes in the value from the source to be mapped and it is passed to the two mapper functions, one is supposed to return the key of the map, the other is supposed to return the value for that key,

- in first the example below, the name is mapped through the identity function, meaning the key is simply the entry from the list (the length of the name), the value for the mapping is the length of the entry from the list (the length of the name).
- the second example below, the mapping function remains the same, however a merging function is added, which tells the underlying `Collectors` how to reconcile values that map to the same key
- the third example is the same as the first one, and the second one, with the difference being that the third argument for the map, is actually the supplier, or map constructor, this allows clients to pass in custom map implementations instead

`Note that by default if there are duplicate keys the basic version of toMap which is not passed in a merger function will use an internal one, the internal one will throw exception if duplicate key is inserted into the map`

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David",
   "Jon");

Map<String, Integer> nameLengthMap =
   names.stream().collect(Collectors.toMap(
     name -> name, // Key the name itself
     String::length // Value the length of the naem
));

Map<String, Long> nameCountMap = names.stream().collect(Collectors.toMap(
    name -> name,             // Key: the name itself
    name -> 1L,               // Value: initial count of 1
    Long::sum                 // Merging function to sum counts for
       duplicates
));
```

```java
Map<String, Long> nameCountMap = names.stream().collect(Collectors.toMap(
    name -> name,            // Key: the name itself
    name -> 1L,              // Value: initial count of 1
    Long::sum                // Merging function to sum counts for
       duplicates
    MyHashMap::new           // Provide a custom map implementation
));
```

The toMap functions have a version called toConcurrentMap which is meant to optimize performance when collecting elements, by leveraging parallel streams (see below)however the interface for them is the same as for the basic toMap methods

Grouping

A special case for mapping function which produces a map where the default is that a merging function is specially designed to collect all values with the same keys into a bucket. Usually an array or list. The idea is that a list of entries or values can be grouped by some common properties, in this case when they end up mapping to the same key.

In the example below, the names are grouped by the length of the name itself, the most simple classifier function used for this grouping. In the second and third example a grouping factory methods are provided, one is for the actual top level map result which represents the grouping, the second is for the value type, which is where the elements are accumulated into.

Similarly to the toMap methods, the grouping methods also support a concurrent version to optimize the grouping by using parallel streaming

```java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David",
   "Eve");
Map<Integer, List<String>> groupedByLength =
   names.stream().collect(Collectors.groupingBy(
    String::length // Key for the grouping is the length of the name
));

Map<Integer, List<String>> groupedByLength =
   names.stream().collect(Collectors.groupingBy(
    String::length, // Key for the grouping is the length of the name
    MyHashMap::new, // Map factory can be provided as second argument
    MyArrayList::new, // The factory for the grouped map values
));
```

Reduction

One of the key features of streams is reduction operations, such operations are terminal operations that return a result that is based on the elements in the stream, and are not of the same type as the initial data structure wrapped in the stream.

```java
ArrayList<Integer> myList = new ArrayList<>();
myList.add(7);
myList.add(18);
myList.add(10);
myList.add(24);
```

```
myList.add(17);
myList.add(5);
Stream<Integer> myStream = myList.stream(); // return a stream
   representation of the list, simply wraps the collection into a stream
Optional<Integer> minVal = myStream.min(Integer::compare); // reducing the
   result, into a single min value from the stream of integers
// the call to min on myStream will terminate the stream, meaning that no
   more terminal or intermediate operations can be called on it

myStream = myList.stream(); // obtain a new reference of a new stream
   object, since it is of the same type re-use the old stream variable
Optional<Integer> maxVal = myStream.max(Integer::compare); // reducing the
   result, into a single max value from the stream of integers
// the call to min on myStream will terminate the stream, meaning that no
   more terminal or intermediate operations can be called on it
```

The list of reduction operations that stream API supports are listed below, all of them are a special case of the `reduce` operation, but exist for convenience since are very often used, operations such as min, max, sum or count for example.

- `reduce` - general reduce operation accepting an accumulator lambda
- `count` - return the count of elements in the stream
- `min` - compute the `min` element in the stream
- `max` - compute the `max` element in the stream
- `sum` - compute the `sum` of all elements in the stream
- `average` - compute the `avg` between all the elements of the stream
- `anyMatch` - check if any element matches a predicate
- `allMatch` - check if all elements match a predicate
- `noneMatch` - check if none of the elements match a predicate
- `findFirst` - find the first element that matches a predicate
- `findAny` - find any element that matches a predicate (relevant for parallel streams)

The reduce operations have some restrictions, there are certain rules that the reduction must follow in order for the result to be predictable and correct.

- `stateless` - the reduce lambda or operation itself must not store any state about the iteration process or elements being visited

- `non-interfering` - the reduce operation must never interfere or mutate the source structure, while the reduction is being executed.

- `associative` - no matter how the elements are traversed the reduce must always produce the correct result without having to store any state about the elements being traversed, given the following expression - `10 _ (2 _ 7)` - is associative, it does not matter in what order the elements are multiplied, however this - 10 * (2 + 7), is not associative, and thus one can not rely on the reduce operation to be correct

`Associativity is of particular importance to the use of reduction operations on parallel streams, discussed in the next section.`

Parallel

The parallel streams are of big importance, especially when huge amounts of data need to be processed, they can dramatically speed up execution of certain operations, a parallel stream can be obtained eitehr at the time of obtaining the stream using the `parallelStream` method, instead of `stream` from the `Collections` API, or to convert an existing stream to a parallel one using the `API` provided by the `BaseStream` interface

Parallel streams have one very specific caveat when dealing when reducing operations, since a parallel stream reduce operations can be split in such a way that multiple reduce operations are run on separate chunks of the elements in the stream, one has to provide additional `combiner` function lambda reference, which is used to tell the underlying implementation how to `combine` results coming from different parallel executions, for example let's say we would like to multiply the square roots of all elements in a stream

```
ArrayList<Integer> myList = new ArrayList<>();
myList.add(7);
myList.add(18);
myList.add(10);
myList.add(24);
myList.add(17);
myList.add(5);

Stream<Integer> myStreamPar = myList.parallelStream(); // return a
   parallel stream representation of the list, simply wraps the collection
   into a stream
Integer accPar = myStreamPar.reduce(1, (acc, elem) -> acc * (elem * elem),
   (left, right) -> left * right); // reduce the list into result (a*a) *
   (b*b)

Stream<Integer> myStreamSeq = myList.stream(); // return a sequential
   stream representation of the list, simply wraps the collection into a
   stream
Integer accSeq = myStreamSeq.reduce(1, (acc, elem) -> acc * (elem *
   elem)); // reduce the list into result (a*a) * (b*b)
```

The `first` call to `reduce` above, uses the accumulator and combiner lambda functions, to correctly compute the product of the squares of each element in the list. Why does it work, the `accumulator`, tells the stream how to accumulate the current result `acc` with an element from the list, while the `combiner` tells the stream how to combine two parallel computations, or in other words two separate accumulation results into one. The combiner function is used to tell the stream how to combine two accumulators, while the accumulator is used to tell the stream how to accumulate elements into one. The `second` call to `reduce` on the sequential stream does not need any combiner, since there is no parallel execution, there are no parallel accumulations being done, therefore no need to provide a combiner.

In general the parallel stream implementation in java uses the fork/join along with split iterators, the basic idea is as follows:

- Splitting the Data: The `Spliterator` is used to divide the source data into smaller segments. Each segment can be processed independently by different threads.
- Fork/Join Framework: Java's Fork/Join framework manages a pool of worker threads to execute tasks concurrently. Each worker thread can take one of the segments provided by the `Spliterator` and process it.
- Combining Results: After processing, the results from the segments are combined using a combiner function. This step is crucial, especially in operations like reduce, where partial results need to be aggregated into a final result.

Unordered

Another property of streams which affects the way data is being processed in certain situations, generally speaking the ordered/unordered nature of a stream depends on the underlying data structure. If the stream is wrapped around a `ArrayList, LinkedList, TreeSet`, generally these are considered `ordered` structures, while if the stream is wrapped around a `HashMap` or `HashSet` are considered `unordered`. The `unordered` part plays a role when the stream is of type `parallel` stream. How does it work ? Well if the stream is parallel but ordered, the processing is done using a fork / join, each batch of elements is processed in parallel, then when time comes to combine the results, the results are combined in order, if the stream is unordered the combination is done in whatever order, possibly as soon as there are at least two results to combine completed, instead with the ordered type, it might be waiting for all results from the parallel processing to complete, then sort them in order and only then run the combine function / lambda. Parallel streams will eagerly combine as soon as there are results/chunks finished and ready to be combined.