Finding the Characteristics of a Stream

Characteristics of a Stream

The Stream API relies on a special object, an instance of the `Spliterator` interface. The name of this interface comes from the fact that the role of a spliterator in the Stream API looks like the role of an iterator has in the Collection API. Moreover, because the Stream API supports parallel processing, a spliterator object also controls how a stream splits its elements among the different CPUs that handle parallelization. The name is the contraction of *split* and *iterator*.

Covering this spliterator object in details is beyond the scope of this tutorial. What you need to know is that this spliterator object carries the *characteristics* of a stream. These characteristics are not something you will often use, but knowing what they are will help you to write better and more efficient pipelines in certain cases.

The characteristics of a stream are the following.

Characteristic	Comment
ORDERED	The order in which the elements of the stream are processed matters.
DISTINCT	There are no doubles in the elements processed by that stream.
NONNULL	There are no null elements in that stream.
SORTED	The elements of that stream are sorted.
SIZED	The number of elements this stream processes is known.
SUBSIZED	Splitting this stream produces two *SIZED* streams.

There are two characteristics, *IMMUTABLE* and *CONCURRENT*, which are not covered in this tutorial.

Every stream has all these characteristics set or unset when it is created.

Remember that a stream can be created in two ways.

1. You can create a stream from a source of data, and we covered several different patterns.
2. Every time you call an intermediate operation on an existing stream, you create a new stream.

The characteristics of a given stream depend on the source it has been created on, or the characteristics of the stream with which it was created, and the operation that created it. If your stream is created with a source, then its characteristics depend on that source, and if you created it with another stream, then they will depend on this other stream and the type of operation you are using.

Let us present each characteristic in more details.

Ordered Streams

ORDERED streams are created with ordered sources of data. The fist example that may come to mind is any instance of the `List` interface. There are others: `Files.lines(path)` and `Pattern.splitAsStream(string)` also produce *ORDERED* streams.

Keeping track of the order of the elements of a stream may lead to overheads for parallel streams. If you do not need this characteristic, then you can delete it by calling the `unordered()` intermediate method on an existing stream. This will return a new stream without this characteristic. Why would you want to do that? Keeping a stream *ORDERED* may be costly in some cases, for instance when you are using parallel streams.

Sorted Streams

A *SORTED* stream is a stream that has been sorted. This stream can be created from a sorted source, such as a `TreeSet` instance, or by a call to the `sorted()` method. Knowing that a stream has already been sorted may be used by the stream implementation to avoid sorting again an already sorted stream. This optimization may not be used all the time because a *SORTED* stream may be sorted again with a different comparator than the one used the first time.

There are some intermediate operations that clear the *SORTED* characteristic. In the following code, you can see that both `strings` and `filteredStream` are *SORTED* streams, whereas `lengths` is not.

```
Collection<String> stringCollection = List.of("one", "two", "two", "three", "four", "five");

Stream<String> strings = stringCollection.stream().sorted();
Stream<String> filteredStrings = strings.filtered(s -> s.length() < 5);
Stream<Integer> lengths = filteredStrings.map(String::length);
```

Mapping or flatmapping a *SORTED* stream removes this characteristic from the resulting stream.

Distinct Streams

A *DISTINCT* stream is a stream with no duplicates among the elements it is processing. Such a characteristic is acquired when building a stream from a `HashSet` for instance, or from a call to the

`distinct()` intermediate method call.

The *DISTINCT* characteristic is kept when filtering a stream but is lost when mapping or flatmapping a stream.

Let us examine the following example.

```
Collection<String> stringCollection = List.of("one", "two", "two", "three", "four", "five");

Stream<String> strings = stringCollection.stream().distinct();
Stream<String> filteredStrings = strings.filtered(s -> s.length() < 5);
Stream<Integer> lengths = filteredStrings.map(String::length);
```

- `stringCollection.stream()` is not *DISTINCT* as it is build from an instance of `List`.
- `strings` is *DISTINCT* as this stream is created by a call to the `distinct()` intermediate method.
- `filteredStrings` is still *DISTINCT*: removing elements from a stream cannot create duplicates.
- `length` has been mapped, so the *DISTINCT* characteristic is lost.

Non-Null Streams

A *NONNULL* stream is a stream that does not contain null values. There are structures from the Collection Framework that do not accept null values, including `ArrayDeque` and the concurrent structures like `ArrayBlockingQueue`, `ConcurrentSkipListSet`, and the concurrent set returned by a call to `ConcurrentHashMap.newKeySet()`. Streams created with `Files.lines(path)` and `Pattern.splitAsStream(line)` are also *NONNULL* streams.

As for the previous characteristics, some intermediate operations can produce a stream with different characteristics.

- Filtering or sorting a *NONNULL* stream returns a *NONNULL* stream.
- Calling `distinct()` on a *NONNULL* stream also returns a *NONNULL* stream.
- Mapping or flatmapping a *NONNULL* stream returns a stream without this characteristic.

Sized and Subsized Streams

Sized Streams

This last characteristic is very important when you want to use parallel streams. Parallel streams are covered in more detail later in this tutorial.

A *SIZED* stream is a stream that knows how many elements it will process. A stream created from any instance of `Collection` is such a stream because the `Collection` interface has a `size()` method, so getting

this number is easy.

On the other hand, there are cases where you know that your stream will process a finite number of elements, but you cannot know this number unless you process the stream itself.

This is the case for streams created with the `Files.lines(path)` pattern. You can get the size of the text file in bytes, but this information does not tell you how many lines this text file has. You need to analyze the file to get this information.

This is also the case for the `Pattern.splitAsStream(line)` pattern. Knowing the number of characters there are in the string you are analyzing does not give any hint about how many elements this pattern will produce.

Subsized Streams

The *SUBSIZED* characteristic has to do with the way a stream is split when computed as a parallel stream. In a nutshell, the parallelization mechanism splits a stream in two parts and distribute the computation among the different available cores on which the CPU is executing. This splitting is implemented by the instance of the `Spliterator` the stream uses. This implementation depends on the source of data you are using.

Suppose that you need to open a stream on an `ArrayList`. All the data of this list is held in the internal array of your `ArrayList` instance. Maybe you remember that the internal array on an `ArrayList` object is a compact array because when you remove an element from this array, all the following elements are moved one cell to the left so that no hole is left.

This makes the splitting an `ArrayList` straightforward. To split an instance of `ArrayList`, you can just split this internal array in two parts, with the same amount of elements in both parts. This makes a stream created on an instance of `ArrayList` *SUBSIZED*: you can tell in advance how many elements will be held in each part after splitting.

Suppose now that you need to open a stream on an instance of `HashSet`. A `HashSet` stores its elements in an array, but this array is used differently than the one used by `ArrayList`. In fact, more than one element can be stored in a given cell of this array. There is no problem in splitting this array, but you cannot tell in advance how many elements will be held in each part without counting them. Even if you split this array by the middle, you can never be sure that you will have the same number of elements in both halves. This is the reason why a stream created on an instance of `HashSet` is *SIZED* but not *SUBSIZED*.

Transforming a stream may change the *SIZED* and *SUBSIZED* characteristics of the returned stream.

- Mapping and sorting a stream preserves the *SIZED* and *SUBSIZED* characteristics.
- Flatmapping, filtering, and calling `distinct()` erases these characteristics.

It is always better to have *SIZED* and *SUBSIZED* stream for parallel computations.

Last update: *September 14, 2021*