Adding a Terminal Operation on a Stream

Avoiding the Use of the Reduce Method

A stream does not process any data if it does not end with a terminal operation. We already covered the terminal operation `reduce()`, and you saw several terminal operations in other examples. Let us now present the other terminal operations you can use on a stream.

Using the `reduce()` method is not the easiest way to reduce a stream. You need to make sure that the binary operator you provide is associative, then you need to know if it has an identity element. You need to check many points to make sure your code is correct and produces the results you expect. If you can avoid using the `reduce()` method, then you definitely should, because it's very easy to make mistakes with it.

Fortunately, the Stream API offers you many other ways to reduce streams: the `sum()`, `min()`, and `max()` that we covered when we presented the specialized streams of numbers are convenient methods that you can use instead of the equivalent `reduce()` calls. We are going to cover more methods in this part, which you should know, to avoid using the `reduce()` method. In fact, you should use this `reduce()` method as a last resort, only if you have no other solution.

Counting the Elements Processed by a Stream

The `count()` method is present in all the stream interfaces: both in specialized streams and streams of objects. It just returns the number of elements processed by that stream, in a `long`. This number can be huge, in fact greater than `Integer.MAX_VALUE`, because it is a `long`. So a stream can count more object than you can put in an `ArrayList` for instance.

You may be wondering why you would need such a great number. In fact, you can create streams with many sources, including sources that can produce huge amounts of elements, greater than `Integer.MAX_VALUE`. Even if it is not the case, it is easy to create an intermediate operation that will multiply the number of elements your stream processes. The `flatMap()` method, which we covered earlier in this tutorial can do that. There are many ways where you may end up with more elements to process than `Integer.MAX_VALUE`. This is the reason why the Stream API supports it.

Here is an example of the `count()` method in action.

```
Collection<String> strings =
        List.of("one", "two", "three", "four", "five", "six", "seven", "eight", "nine", "ten");

long count =
        strings.stream()
                .filter(s -> s.length() == 3)
                .count();
System.out.println("count = " + count);
```

Running this code produces the following result.

```
count = 4
```

Consuming Each Element One by One

The `forEach()` method of the Stream API allows you to pass each element of your stream to an instance of the `Consumer` interface. This method is very handy for printing the elements processed by a stream. This is what the following code does.

```
Stream<String> strings = Stream.of("one", "two", "three", "four");
strings.filter(s -> s.length() == 3)
       .map(String::toUpperCase)
       .forEach(System.out::println);
```

Running this code prints out the following.

```
ONE
TWO
```

This method is so simple that you may be tempted to use it in wrong use cases.

Remember that the lambda expressions you write should avoid mutating their outside scope. Sometimes, mutating outside the state is called *conducting side-effects*. The case of the consumer is special because a consumer that does not have any side effect will not do much for you. In fact, calling `System.out.println()` creates a side effect on the console of your application.

Let us consider the following example.

```
Stream<String> strings = Stream.of("one", "two", "three", "four");
List<String> result = new ArrayList<>();

strings.filter(s -> s.length() == 3)
       .map(String::toUpperCase)
       .forEach(result::add);

System.out.println("result = " + result);
```

Running the previous code prints out the following.

```
result = [ONE, TWO]
```

So you may be tempted to use this code because it's simple, and it "just works". Well, there are several wrong things that this code is doing. Let us go through them.

Calling `result::add` adds all the elements processed by that stream to the outside `result` list by mutating that list from within the stream. This consumer is creating a side effect to a variable outside the scope of the stream itself.

Accessing such a variable makes your lambda expression a *capturing lambda expression*. It is perfectly legal to create such lambda expressions; you just need to be aware that there is an important performance hit in doing so. If performance is an important matter in your application, then you should avoid writing capturing lambdas.

Moreover, this way of writing things prevents you from making this stream parallel. Moreover, this way of consuming elements is problematic if you try to make this stream parallel. If you do, then you will have several threads accessing your result list concurrently. This list is an instance of `ArrayList`, not a class tailored to handle concurrent access.

You have two patterns to store the elements of a stream in a list. The following example demonstrates the first pattern, which uses a collection object. The second pattern, which uses Collector objects, is covered later.

```
Stream<String> strings = Stream.of("one", "two", "three", "four");

List<String> result =
    strings.filter(s -> s.length() == 3)
           .map(String::toUpperCase)
           .collect(Collectors.toList());
```

This collector creates an instance of `ArrayList` and adds the elements processed by your stream in it. So this pattern is not creating any side effect so there is no performance hit.

Parallelism and concurrency are handled by the Collector API itself, so you can safely make this stream parallel.

This pattern code is as simple and readable as the previous one. It does not have any of the drawbacks of creating side effects within a consumer object. This is definetely the pattern you should be using in your code.

Starting with Java SE 16, you have a second, even simpler, pattern.

```
Stream<String> strings = Stream.of("one", "two", "three", "four");

List<String> result =
    strings.filter(s -> s.length() == 3)
           .map(String::toUpperCase)
           .toList();
```

This pattern produces a special instance of `List` that is unmodifiable. If what you need is a modifiable list, you should stick to the first collector pattern. It also may perform better than collecting your stream in an instance of `ArrayList`. This point is covered in the next paragraph.

Collecting Stream Elements in a Collection, or an Array

The Stream API offers you several ways of collecting all the elements processed by a stream into a collection. You had a first glimpse at two of those patterns in the previous section. Let us see the others.

There are several questions you need to ask yourself before choosing which pattern you need.

- Do you need to build an immutable list?
- Are you comfortable with an instance of `ArrayList`? Or would you prefer an instance of `LinkedList`?
- Do you have a precise idea of how many elements your stream is going to process?
- Do you need to collect your element in a precise, maybe third party or homemade implementation of `List`?

The Stream API can handle all these situations.

Collecting in a Plain ArrayList

You already used this pattern in a previous example. It is the simplest you can use and returns the elements in an instance of `ArrayList`.

Here is an example of such a pattern in action.

```
Stream<String> strings = Stream.of("one", "two", "three", "four");

List<String> result =
    strings.filter(s -> s.length() == 3)
            .map(String::toUpperCase)
            .collect(Collectors.toList());
```

This pattern creates a simple instance of `ArrayList` and accumulates the elements of your stream in it. If there are too many elements for the internal array of the `ArrayList` to store them, then the current array will be copied into a larger one and will be handled by the garbage collector.

If you want to avoid that, and you know the amount of elements your stream will produce, then you can use the `Collectors.toCollection()` collector, which takes a supplier as an argument to create the collection in which you will be collecting the processed elements. The following code uses this pattern to create an instance of `ArrayList` with an initial capacity of 10,000.

```
Stream<String> strings = ...;

List<String> result =
    strings.filter(s -> s.length() == 3)
            .map(String::toUpperCase)
            .collect(Collectors.toCollection(() -> new ArrayList<>(10_000)));
```

Collecting in an Immutable List

There are cases where you need to accumulate your elements in an immutable list. This may sound paradoxical because collecting consists in adding elements to a container that has to be mutable. Indeed, this is how the Collector API works as you will see in more details later in this tutorial. At the end of this accumulating

operation, the Collector API can proceed with a last, optional, operation, which, in this case, consists in sealing the list before returning it.

To do that, you just need to use the following pattern.

```
Stream<String> strings = ...;

List<String> result =
    strings.filter(s -> s.length() == 3)
           .map(String::toUpperCase)
           .collect(Collectors.toUnmodifiableList()));
```

In this example, `result` is an immutable list.

Starting with Java SE 16, there is a better way to collect your data in an immutable list, which can be more efficient on some cases. The pattern is the following.

```
Stream<String> strings = ...;

List<String> result =
    strings.filter(s -> s.length() == 3)
           .map(String::toUpperCase)
           .toList();
```

How can it be more efficient? The first pattern, built on the use of a collector, begins by collecting your elements in a plain `ArrayList` and then seals it to make it immutable when the processing is done. What your code sees is just the immutable list built from this `ArrayList`.

As you know, an instance of `ArrayList` is built on an internal array that has a fixed size. This array can become full. In that case, the `ArrayList` implementation detects it and copies it into a larger array. This mechanism is transparent for the client, but it comes with an overhead: copying this array takes some time.

There are cases where the Stream API can keep track of how many elements are to be processed before all the stream is consumed. In that case, creating an internal array of the right size is more efficient because it avoids the overhead of copying small arrays into larger ones.

This optimization has been implemented in the `Stream.toList()` method, which has been added to Java SE 16. If what you need is an immutable list, then you should be using this pattern.

Collecting in a Homemade List

If you need to collect your data in your own list or third party list outside the JDK, then you can use the `Collectors.toCollection()` pattern. The supplier you used to tune the initial size of you instance of `ArrayList` can also be used to build any implementation of `Collection`, including implementations that are not part of the JDK. All you need to give is a supplier. In the following example, we provide a supplier to create an instance of `LinkedList`.

```
Stream<String> strings = ...;

List<String> result =
    strings.filter(s -> s.length() == 3)
           .map(String::toUpperCase)

```

```
            .collect(Collectors.toCollection(LinkedList::new));
```

Collecting in a Set

Because the `Set` interface is an extension of the `Collection` interface, you could use the pattern `Collectors.toCollection(HashSet::new)` to collect your data in an instance of `Set`. This is fine, but the Collector API still gives you a cleaner pattern to do that: the `Collectors.toSet()`.

```
Stream<String> strings = ...;

Set<String> result =
    strings.filter(s -> s.length() == 3)
            .map(String::toUpperCase)
            .collect(Collectors.toSet());
```

You may be wondering if there is any difference between these two patterns. The answer is yes, there is a subtle difference, which you will see later in this tutorial.

If what you need is an immutable set, the Collector API has another pattern for you: `Collectors.toUnmodifiableSet()`.

```
Stream<String> strings = ...;

Set<String> result =
    strings.filter(s -> s.length() == 3)
            .map(String::toUpperCase)
            .collect(Collectors.toUnmodifiableSet());
```

Collecting in a Array

The Stream API also has its own set of `toArray()` method overloads. There are two of them.

The first one is a plain `toArray()` method, that returns an instance of `Object[]`. If the exact type of your stream is known, then this type is lost if you use this pattern.

The second one takes an argument of type `IntFunction<A[]>`. This type may look scary at first, but writing an implementation of this function is in fact very easy. If you need to build an array of strings of characters, then the implementation of this function is `String[]::new`.

```
Stream<String> strings = ...;

String[] result =
    strings.filter(s -> s.length() == 3)
            .map(String::toUpperCase)
            .toArray(String[]::new);

System.out.println("result = " + Arrays.toString(result));
```

Running this code produces the following result.

```
result = [ONE, TWO]
```

Extracting the Maximum and the Minimum of a Stream

The Stream API gives you several methods for that, depending on what stream you are currently working with.

We already covered the `max()` and `min()` methods from the specialized streams of numbers: `IntStream`, `LongStream` and `DoubleStream`. You know that these operations do not have an identity element, so you should not be surprised to discover that there are all returning optional objects.

By the way, the `average()` method from the same streams of number also returns an optional object, since the average operation does not have an identity element neither.

The `Stream` interface also has the two methods `max()` and `min()`, that also return an optional object. The difference with the stream of objects is that the elements of a `Stream` can really be of any kind. To be able to compute a maximum or a minimum, the implementation needs to compare these objects. This is the reason why you need to provide a comparator for these methods.

Here is the `max()` method in action.

```
Stream<String> strings = Stream.of("one", "two", "three", "four");
String longest =
     strings.max(Comparator.comparing(String::length))
            .orElseThrow();
System.out.println("longest = " + longest);
```

It will print the following on your console.

```
longest = three
```

Remember that trying to open an optional object that is empty throws a `NoSuchElementException`, which is something you do not want to see in your application. It happens only if your stream does not have any data to process. In this simple example, you have a stream that processes several strings of character with no filter operation. This stream cannot be empty, so you can safely open this optional object.

Finding an Element in a Stream

The Stream API gives you two terminal operations to find an element: `findFirst()` and `findAny()`. These two methods do not take any argument and return a single element of your stream. To properly handle the case of empty streams, this element is wrapped in an optional object. If your stream is empty, then this optional is also empty.

Understanding which element is returned requires you to understand that streams may be *ordered*. An ordered stream is simply a stream in which the order of the elements matters and is kept by the Stream API. By default, a stream created on any ordered source (for instance an implementation of the `List` interface) is itself ordered.

On such a stream, it makes sense to have a first, second, or third element. Finding the *first* element of such a stream then makes perfect sense too.

If your stream is not ordered, or if the order has been lost in your stream processing, then finding the *first* element is undefined, and calling `findFirst()` returns in fact any element of the stream. You will see more details on ordered streams later in this tutorial.

Note that calling `findFirst()` triggers some checking in the stream implementation to make sure that you get the *first* element of that stream if that stream is ordered. This can be costly if your stream is a parallel stream. There are many cases in which getting the *first* found element is not relevant, including cases where your stream only processes a single element. In all these cases, you should be using `findAny()` instead of `findFirst()`.

Let us see `findFirst()` in action.

```
Collection<String> strings =
        List.of("one", "two", "three", "four", "five", "six", "seven", "eight", "nine", "ten");

String first =
    strings.stream()
            // .unordered()
            // .parallel()
            .filter(s -> s.length() == 3)
            .findFirst()
            .orElseThrow();

System.out.println("first = " + first);
```

This stream is created on an instance of `List`, which makes it an *ordered* stream. Note that the two lines `unordered()` and `parallel()` are commented in this first version.

Running this code several times will always give you the same result.

```
first = one
```

The `unordered()` intermediate method call makes your *ordered* stream an *unordered* stream. In this case it does not make any difference because your stream is processed sequentially. Your data is pulled from a list that always traverses its elements in the same order. Replacing the `findFirst()` method call with a `findAny()` method call does not make any difference either for the same reason.

The first modification that you can make on this code is to uncomment the `parallel()` method call. Now you have an *ordered* stream, processed in parallel. Running this code several times will always give you the same result: `one`. This is because your stream is *ordered*, so the first element is defined, no matter how your stream has been processed.

To make this stream *unordered*, you can either uncomment the `unordered()` method call or replace the `List.of()` with a `Set.of()`. In both cases, terminating your stream with `findFirst()` will return a random element from that parallel stream. The way parallel streams are processed makes it so.

The second modification that you can make in this code, is to replace `List.of()` by `Set.of()`. Now this source is not *ordered* anymore. Moreover, the implementation returned by `Set.of()` is such that the traversing of the elements of the set happens in a randomized order. Running this code several times shows you that both

`findFirst()` and `findAny()` return a random string of characters, even if `unordered()` and `parallel()` are both commented out. Finding the *first* element of *nonordered* source is not defined, and the result is random.

From these examples, you can deduce that there are some precautions taken in the implementation of the parallel stream to track which element is the first. This constitutes an overhead, so in this case, you should only call `findFirst()` if you really need it.

Checking if the Elements of a Stream Match a Predicate

There are cases where finding an element in a stream or failing to find any element in a stream may be what you really need to do. The element you find is not relevant for your application; what it is important is that this element exists.

The following code would work to check for the existence of a given element.

```
boolean exists =
    strings.stream()
           .filter(s -> s.length() == 3)
           .findFirst()
           .isPresent();
```

In fact, this code checks if the returned optional is empty or not.

The previous pattern works fine, but the Stream API gives you a more efficient way to do it. In fact, building this optional object is an overhead, which you do not pay if you use one of the three following methods. These three methods take a predicate as an argument.

- `anyMatch(predicate)`: returns `true` if one element of the stream is found, that matches the given predicate.
- `allMatch(predicate)`: returns `true` if all the elements of the stream match the predicate.
- `noneMatch(predicate)`: returns `true` if none of the elements match the predicate.

Let us see these methods in action.

```
Collection<String> strings =
    List.of("one", "two", "three", "four", "five", "six", "seven", "eight", "nine", "ten");

boolean noBlank  =
        strings.stream()
               .allMatch(Predicate.not(String::isBlank));
boolean oneGT3   =
        strings.stream()
               .anyMatch(s -> s.length() == 3);
boolean allLT10  =
        strings.stream()
               .noneMatch(s -> s.length() > 10);

System.out.println("noBlank = " + noBlank);
System.out.println("oneGT3  = " + oneGT3);
System.out.println("allLT10 = " + allLT10);
```

Running this code produces the following result.

```
noBlank = true
oneGT3  = true
allLT10 = true
```

Short-Circuiting the Processing of a Stream

You may have noticed an important difference between the different terminal operation that we have covered here.

Some of them require the processing of all the data consumed by your stream. This is the case of the *COUNT*, *MAX*, *MIN*, *AVERAGE* operations, as well as the `forEach()`, `toList()`, or `toArray()` method calls.

It is not the case for the last terminal operations we covered. The `findFirst()` or `findAny()` methods will stop processing your data as soon as an element is found, no matter how many elements are left to be processed. The same goes for `anyMatch()`, `allMatch()`, and `noneMatch()`: they may interrupt the processing of the stream with a result without having to consume all the elements your source can produce.

There are still cases where these last methods need to process all the elements:

- Returning an empty optional for `findFirst()` and `findAny()` is only possible when all the elements have been processed.
- Returning `false` for `anyMatch()` also needs to process all the elements of the stream.
- Returning `true` for `allMatch()` and `noneMatch()` also needs to process all the elements of the stream.

These methods are called *short-circuiting* methods in the Stream API because they can produce a result without having to process all the elements of your stream.

Last update: *September 14, 2021*

Reducing a Stream ← **Adding a Terminal Operation on a Stream** → Finding the Characteristics of a Stream

Home > Tutorials > The Stream API > Adding a Terminal Operation on a Stream