Using a Collector as a Terminal Operation

Collecting Stream Elements with a Collector

You already used a very useful pattern to collect the elements processed by a stream in an `List`: `collect(Collectors.toList())`. This `collect()` method is a terminal method defined in the `Stream` interface that takes an object of type `Collector` as an argument. This `Collector` interface defines an API of its own, that you can use to create any kind of in-memory structure to store the data processed by a stream. Collecting can be made in any instance of `Collection` or `Map`, it can be used to create strings of characters, and you can create your own instance of the `Collector` interface to add your own structures to this list.

Most of the collectors you will be using can be created using one of the factory methods of the `Collectors` factory class. This is what you did when you wrote `Collectors.toList()`, or `Collectors.toSet()`. Some collectors created with these methods can be combined, leading to even more collectors. All these points are covered in this tutorial.

If you can't find what you need in this factory class, then you can decide to create your own collector by implementing the `Collector` interface. Implementing this interface is also covered in this tutorial.

The Collector API is handled differently in the `Stream` interface and in the specialized streams of numbers: `IntStream`, `LongStream`, and `DoubleStream`. The `Stream` interface has two overloads of the `collect()` method, whereas the streams of numbers have only one. The missing one is precisely the one that takes a collector object as an argument. So you cannot use a collector object with a specialized stream of numbers.

Collecting in a Collection

The `Collectors` factory class gives you three methods to collect the elements of your stream in an instance of the `Collection` interface.

1. `toList()` collects them in an `List` object.

2. `toSet()` collectors them in a `Set` object.
3. If you need any other `Collection` implementation, you can use `toCollection(supplier)`, where the `supplier` argument will be used to create the `Collection` object you need. If you need your data to be collected in an instance of `LinkedList`, this is the method you should be using.

Your code should not rely on the exact implementation of `List` or `Set` that is currently returned by these methods as it is not part of the specification.

You can also get immutable implementations of `List` and `Set` using the two methods `toUnmodifiableList()` and `toUnmodifiableSet()`.

The following example shows this pattern in action. First, let us collect in a plain `List` instance.

```
List<Integer> numbers =
IntStream.range(0, 10)
         .boxed()
         .collect(Collectors.toList());
System.out.println("numbers = " + numbers);
```

This code uses the `boxed()` intermediate method to create a `Stream<Integer>` from the `IntStream` created by `IntStream.range()` by boxing all the elements of that stream. Running this code prints the following.

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This second example creates a `HashSet` with only even numbers and without duplicates.

```
Set<Integer> evenNumbers =
IntStream.range(0, 10)
         .map(number -> number / 2)
         .boxed()
        .collect(Collectors.toSet());
System.out.println("evenNumbers = " + evenNumbers);
```

Running this code gives you the following result.

```
evenNumbers = [0, 1, 2, 3, 4]
```

And this last example uses a `Supplier` object to create the instance of `LinkedList` used to collect the elements of the stream.

```
LinkedList<Integer> linkedList =
IntStream.range(0, 10)
         .boxed()
         .collect(Collectors.toCollection(LinkedList::new));
System.out.println("linked listS = " + linkedList);
```

Running this code gives you the following result.

```
linked list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Counting with a Collector

The `Collectors` factory class gives you several methods to create collectors that are doing the same kind of things that a plain terminal method offers you. This is the case for the `Collectors.counting()` factory method, which does the same thing as calling `count()` on a stream.

This is worth noting, and you may be wondering why such a feature has been implemented twice with two different patterns. This question is answered in the next section about collecting in maps where you will be combining collectors to create more collectors.

For now, writing the two following lines of code lead to the same result.

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

long count = strings.stream().count();
long countWithACollector = strings.stream().collect(Collectors.counting());

System.out.println("count = " + count);
System.out.println("countWithACollector = " + countWithACollector);
```

Running this code gives you the following result.

```
count = 3
countWithACollector = 3
```

Collecting in a String of Characters

Another very useful collector provided by the `Collectors` factory class is the `joining()` collector. This collector only works on a stream of strings of characters and joins the elements of that stream together in a single string. It has several overloads.

- The first one takes a separator as an argument.
- The second one takes a separator, a prefix, and a suffix as arguments.

Let us see this collector in action.

```
String joined =
    IntStream.range(0, 10)
             .boxed()

```

```
                .map(Object::toString)
                .collect(Collectors.joining());

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

Running this code produces the following result.

```
joined = 0123456789
```

You can add a separator to this string with the following code.

```
String joined =
    IntStream.range(0, 10)
                .boxed()
                .map(Object::toString)
                .collect(Collectors.joining(", "));

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

The result is the following.

```
joined = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Let us see the last overload in action, which takes a separator, a prefix, and a suffix.

```
String joined =
    IntStream.range(0, 10)
                .boxed()
                .map(Object::toString)
                .collect(Collectors.joining(", ", "{"), "}");

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

The result is the following.

```
joined = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Note that this collector handles properly the corner cases where you stream is empty or only processes a single element.

This collector is very handy when you need to produce this kind of string of characters. You may be tempted to use it even if your data is not in a collection in the first place or with only a few elements. If this is the case, maybe using the `String.join()` factory class, or a `StringJoiner` object will work all the same, without paying the overhead of creating a stream.

Partitioning Elements with a Predicate

The Collector API offers three patterns to create maps from the elements of a stream. The first one we cover creates map with boolean keys. It is created with the `partitionningBy()` factory method.

All the elements of the stream will be bound to either the `true` or the `false` boolean value. The map stores all the elements bound to each value in a list. So, if this collector is applied to a `Stream`, it will produce a map with the following type: `Map<Boolean, List<T>>`.

Deciding if a given element should be bound to `true` or `false` is made by testing this element with a predicate, which is provided as an argument to the collector.

The following example shows this collector in action.

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

Map<Boolean, List<String>> map =
    strings.stream()
           .collect(Collectors.partitioningBy(s -> s.length() > 4));

map.forEach((key, value) -> System.out.println(key + " :: " + value));
```

Running this code produces the following result.

```
false :: [one, two, four, five, six, nine, ten]
true :: [three, seven, eight, eleven, twelve]
```

This factory method has an overload, which takes a collector as a further argument. This collector is called a *downstream collector*. We will cover these downstream collectors in the next paragraph of this tutorial, when we present the `groupingBy()` collector.

Collecting in a Map with Grouping By

The second collector we present is very important because it allows you to create histograms.

Grouping the Elements of a Stream in a Map

The collector you can use to create histogram is created with the `Collectors.groupingBy()` method. This method has several overloads.

The collector creates a map. A key is computed for each element of the stream by applying an instance of `Function` to it. This function is provided as an argument of the `groupingBy()` method. It is called a *classifier* in the Collector API.

There is no restriction on this function apart from the fact that it should not return null.

Applying this function may return the same key for more than one element of your stream. The `groupingBy()` collector supports this, and gather all these elements in a list, bound to that key.

So, if you are processing a `Stream` and use a `Function<T, K>` as a classifier, the `groupingBy()` collector creates a `Map<K, List<T>>`.

Let examine the following example.

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

Map<Integer, List<String>> map =
    strings.stream()
           .collect(Collectors.groupingBy(String::length));

map.forEach((key, value) -> System.out.println(key + " :: " + value));
```

The classifier used in this example is a function that returns the length of each string from that stream. So, the map grouped the strings in lists by their length. It has the type `Map<Integer, List<String>>`.

Running this code prints the following.

```
3 :: [one, two, six, ten]
4 :: [four, five, nine]
5 :: [three, seven, eight]
6 :: [eleven, twelve]
```

Post-processing the Values Created with a Grouping By

Counting the Lists of Values

The `groupingBy()` method also accepts another argument, which is another collector. This collector is called a *downstream collector* in the Collector API, but it is just a regular collector. What makes it a *downstream collector* is the fact that it is passed as an argument to the creation of another collector.

This downstream collector is used to collect the values of the map created by the `groupingBy()` collector.

In the previous example, the `groupingBy()` collector created a map which values are lists of strings. If you give a downstream collector to the `groupingBy()` method, the API will stream these lists one by one and collect these streams with your downstream collector.

Suppose you pass the `Collectors.counting()` as a downstream collector. What will be computed is the following.

```
[one, two, six, ten]   .stream().collect(Collectors.counting()) -> 4L
[four, five, nine]     .stream().collect(Collectors.counting()) -> 3L
[three, seven, eight] .stream().collect(Collectors.counting()) -> 3L
```

```
[eleven, twelve]      .stream().collect(Collectors.counting()) -> 2L
```

This code is not Java code, so you cannot execute it. It is just there to explain how this downstream collector is used.

The map that will be created now depends on the downstream collector you provide. The keys are not modified, but the values may be. In the case of the `Collectors.counting()`, the values are transformed to `Long`. The type of the map then becomes `Map<Integer, Long>`.

The previous example becomes the following.

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

Map<Integer, Long> map =
    strings.stream()
            .collect(
                Collectors.groupingBy(
                    String::length,
                    Collectors.counting()));

map.forEach((key, value) -> System.out.println(key + " :: " + value));
```

Running this code prints the following result. It gives the number of string per length, which is the histogram of the strings by their length.

```
3 :: 4
4 :: 3
5 :: 3
6 :: 2
```

Joining the Lists of Values

You can also pass the `Collectors.joining()` collector as a downstream collector, because the values of this map are lists of strings. Remember that this collector can only be used on streams of strings of characters. This creates an instance of `Map<Integer, String>`: the values take the type created by this collector. You can change the previous example to the following.

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

Map<Integer, String> map =
        strings.stream()
                .collect(
                        Collectors.groupingBy(
                                String::length,
                                Collectors.joining(", ")));

```

```
    map.forEach((key, value) -> System.out.println(key + " :: " + value));
```

Running this code produces the following result.

```
3 :: one, two, six, ten
4 :: four, five, nine
5 :: three, seven, eight
6 :: eleven, twelve
```

Controlling the Instance of Map

The last overload of this `groupingBy()` method takes an instance of a `Supplier` as an argument to give you control on which instance of `Map` you need this collector to create.

Your code should not rely on the exact type of map that the `groupingBy()` collector is returning because it is not part of the specification.

Collecting in a Map with To Map

The Collector API offers you a second pattern to create maps: the `Collectors.toMap()` pattern. This pattern works with two functions, both applied to the elements of your stream.

1. The first one is called the *key mapper* and is used to create the key.
2. The second one is called the *value mapper* and is used to create the value.

This collector is not used in the same cases as the `Collectors.groupingBy()`. In particular, it does not handle the case where several elements of your stream generate the same key. In that case, by default, an `IllegalStateException` is raised.

This collector is very handy to create caches. Suppose you have a `User` class with a `primaryKey` property of type `Long`. You can create a cache of your `User` objects with the following code.

```
List<User> users = ...;

Map<Long, User> userCache =
    users.stream()
         .collect(Collectors.toMap(
                 User::getPrimaryKey,
                 Function.idendity()));
```

The use of the `Function.identity()` factory method just tells the collector not to transform the elements of the stream.

If you expect several elements of the stream to generate the same key, then you can pass a further argument to the `toMap()` method. This argument is of type `BinaryOperator`. It will be applied by the

implementation to the conflicting elements when they are detected. Your binary operator will then produce a result that will be put in the map in place of the previous value.

The following shows you how you can use this collector with conflicting values. Here the values are concatenated together with a separator.

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

Map<Integer, String> map =
    strings.stream()
            .collect(
                    Collectors.toMap(
                            element -> element.length(),
                            element -> element,
                            (element1, element2) -> element1 + ", " + element2));

map.forEach((key, value) -> System.out.println(key + " :: " + value));
```

In this example, the three arguments passed to the `toMap()` method are the following:

1. `element -> element.length()` is the *key mapper*.
2. `element -> element` is the *value mapper*.
3. `(element1, element2) -> element1 + ", " + element2)` is the *merge function*, called with the two elements that have generated the same key.

Running this code produces the following result.

```
3 :: one, two, six, ten
4 :: four, five, nine
5 :: three, seven, eight
6 :: eleven, twelve
```

As for the `groupingBy()` collector, you can pass a supplier as an argument to the `toMap()` method to control what instance of the `Map` interface this collector will use.

The `toMap()` collector has a twin method, `toConcurrentMap()` that will collect your data in a concurrent map. The exact type of the map is not guaranteed by the implementation.

Extracting Maxes from a Histogram

The `groupingBy()` collector is your best pattern to compute histograms on the data you need to analyze. Let us examine a complete example where you build a histogram and then try to find the maximum value in it based a certain criterion.

Extracting a Non-Ambiguous Max

The histogram you are going to analyze is the following. It looks like the one we used in a previous example.

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

Map<Integer, Long> histogram =
    strings.stream()
            .collect(
                    Collectors.groupingBy(
                            String::length,
                            Collectors.counting()));

histogram.forEach((key, value) -> System.out.println(key + " :: " + value));
```

Printing this histogram gives you the following result.

```
3 :: 4
4 :: 3
5 :: 3
6 :: 2
```

Extracting the maximum value from this histogram should give you the result: `3 :: 4`. The Stream API has all the tools you need to extract a maximum value. Unfortunately, there is no `stream()` method on the `Map` interface. So to create a stream on a map, you first need to get one of the collections you can get from a map.

1. The set of the entries with the `entrySet()` method.
2. The set of the keys with the `keySet()` method.
3. Or the collection of the values with the `values()` method.

Here you need both the key and the maximum value, so the right choice is to stream the set returned by `entrySet()`.

The code you need is the following.

```
Map.Entry<Integer, Long> maxValue =
    histogram.entrySet().stream()
             .max(Map.Entry.comparingByValue())
             .orElseThrow();

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

You can notice that this code uses the `max()` method from the `Stream` interface, which takes a comparator as an argument. It turns out that the `Map.Entry` interface has several factory methods to

create such a comparator. The one we use in this example creates a comparator that can compare `Map.Entry` instances, using the value of these key-value pairs to compare them. This comparison can work only if the value implements the `Comparable` interface.

This pattern of code is very generic and can be used on any map as long as it has comparable values. We can make it less generic and more readable, thanks to the introduction of records in Java SE 16.

Let us create a record to model the key-value pairs of this map. Creating a record is a one-liner. Because local records are allowed in the language, you can copy these lines within any method.

```
record NumberOfLength(int length, long number) {

    static NumberOfLength fromEntry(Map.Entry<Integer, Long> entry) {
        return new NumberOfLength(entry.getKey(), entry.getValue());
    }

    static Comparator<NumberOfLength> comparingByLength() {
        return Comparator.comparing(NumberOfLength::length);
    }
}
```

With this record, the previous pattern becomes the following.

```
NumberOfLength maxNumberOfLength =
    histogram.entrySet().stream()
             .map(NumberOfLength::fromEntry)
             .max(NumberOfLength.comparingByLength())
             .orElseThrow();

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

Running this example prints out the following.

```
maxNumberOfLength = NumberOfLength[length=3, number=4]
```

You can see that this record looks like the `Map.Entry` interface. It has a factory method for the mapping of a key-value pair and a factory method to create the comparator you need. The analysis of your histogram becomes much more readable and easy to understand.

Extracting an Ambiguous Maximum Value

The previous example was a nice example, because there was only on maximum value in your list. Unfortunately, real life cases are often not that nice, and you may have several key-value pairs that match the maximum value.

Let us remove one element from the collection of the previous example.

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

```
            "ten", "eleven", "twelve");

Map<Integer, Long> histogram =
    strings.stream()
            .collect(
                    Collectors.groupingBy(
                            String::length,
                            Collectors.counting()));

histogram.forEach((key, value) -> System.out.println(key + " :: " + value));
```

Printing this histogram gives you the following result.

```
3 :: 3
4 :: 3
5 :: 3
6 :: 2
```

Now we have three key-value pairs for the maximum value. If you use the previous pattern of code to extract it, one of these three will be selected and returned, hiding the two others.

A solution to tackle this issue would be to create another map, where the keys are the number of strings with a given length, and the value the lengths that match this number. In other words: you need to invert this map. This is a good use case for the `groupingBy()` collector. This example will be covered later in this part, as we need one more element to write this code.

Using Intermediate Collectors

The collectors that we covered so far are counting, joining, and collecting to a list or a map. They are all modeling terminal operations. The Collector API offers other collectors that are conducting intermediate operations: mapping, filtering and flatmapping. You may be wondering what could be the sense of having a terminal method `collect()` that would model an intermediate operation. In fact, these special collectors cannot be created alone. The factory methods that you can use to create them all need a downstream collector as a second argument.

So, the overall collector you can create with these methods is a combination of an intermediate operation and a terminal operation.

Mapping with a Collector

The first intermediate operation we can examine is the mapping operation. A mapping collector is created with the `Collectors.mapping()` factory method. It takes a regular mapping function as a first argument and a mandatory downstream collector as a second argument.

In the following example, we are combining a mapping with the collection of the mapped elements in a list.

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

List<String> result =
    strings.stream()
        .collect(
            Collectors.mapping(String::toUpperCase, Collectors.toList()));

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

The `Collectors.mappping()` factory method creates a regular collector. You can pass this collector as a downstream collector to any method that accepts one, including, for instance, `groupingBy()` or `toMap()`. You may remember from the section "Extracting an Ambiguous Maximum Value" that we left an open question about inverting a map. Let us use this mapping collector to solve this problem.

In this example, you created a histogram. You now need to invert this histogram with a `groupingBy()` to find all the maximum values.

The following code creates such a map.

```
Map<Integer, Long> histogram = ...;

var map =
    histogram.entrySet().stream()
        .map(NumberOfLength::fromEntry)
        .collect(
            Collectors.groupingBy(NumberOfLength::number));
```

Let us examine this code and determine the exact type of the map that is built.

The keys of this map are the number of times each length is present in the original stream. It is the `number` component of the `NumberOfLength` record, that is, a `Long`.

The values are the elements of this stream, collected into lists. So, the values are lists of `NumberOfLength` objects. The exact type of this map is `Map<Long, NumberOfLength>`.

It turns out that this is not exactly what you need. What you need is only the length of the strings, not the two components of the record. Extracting a component from a record is just a mapping. What you need is mapping these instances of `NumberOfLength` to their `length` component. Now that we covered the mapping collector, solving this point becomes possible. All you need to do is add the right downstream collector to the `groupingBy()` call.

The code becomes the following.

```
Map<Integer, Long> histogram = ...;

var map =
    histogram.entrySet().stream()

```

```
        .map(NumberOfLength::fromEntry)
        .collect(
            Collectors.groupingBy(
                NumberOfLength::number,
                Collectors.mapping(NumberOfLength::length, Collectors.toList())));
```

The values of the map built are now lists of mapped `NumberOfLength` objects, using the `NumberOfLength::length` mapper. This map is of type `Map<Long, List<Integer>>`, which is exactly what you need.

To get all the maximum values, you can apply the same pattern as the one we used previously, using the key to get the maximum value instead of the value.

The complete code from the histogram, including the maximum value extraction follows.

```
Map<Long, List<Integer>> map =
    histogram.entrySet().stream()
             .map(NumberOfLength::fromEntry)
             .collect(
                Collectors.groupingBy(
                    NumberOfLength::number,
                    Collectors.mapping(NumberOfLength::length, Collectors.toList())));

Map.Entry<Long, List<Integer>> result =
    map.entrySet().stream()
       .max(Map.Entry.comparingByKey())
       .orElseThrow();

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

Running this code produces the following.

```
result = 3=[3, 4, 5]
```

It means that there are three lengths of strings that are represented three times in this stream: 3, 4, and 5.

This example shows a collector nested in two more collectors, something that happens quite frequently when you are working with this API. It may look intimidating at first, but it is just combining collectors using this downstream collector mechanism.

You can see why it is interesting to have these intermediate collectors. Being able to model intermediate operations with a collector gives you the possibility to create a downstream collector for almost any kind of processing, which you can use to post process the values of maps.

Filtering and Flatmapping with a Collector

The filtering collector follows the same pattern as the mapping collector. It is created with the `Collectors.filtering()` factory method that takes a regular predicate to filter the data and a mandatory downstream collector.

The same goes for the flatmapping collector, created by the `Collectors.flatMapping()` factory method, that takes a flatmapping function (a function that returns a stream), and a mandatory downstream collector.

Using Terminal Collectors

There Collector API also offers several terminal operations that correspond to terminal operations available on the Stream API.

- `maxBy()`, and `minBy()`. These two methods both take a comparator as an argument and return an optional object that is empty if the stream processed is itself empty.
- `summingInt()`, `summingLong()`, and `summingDouble()`. These three methods take a mapping function as an argument to map the element of your stream to `int`, `long` and `double` respectively, before summing them.
- `averagingInt()`, `averagingLong()`, and `averagingDouble()`. These three methods also takes a mapping function as an argument, to map the element of your stream to `int`, `long` and `double`, respectively, before computing the average. These collectors do not work the same as the corresponding `average()` methods defined in `IntStream`, `LongStream`, and `DoubleStream`. They all return a `Double` instance and return 0 for empty streams. The `average()` methods return an optional object that is empty for empty streams.

Last update: *September 14, 2021*

Finding the Characteristics of a Stream ← **Using a Collector as a Terminal Operation** → Creating Your Own Collector

Home > Tutorials > The Stream API > Using a Collector as a Terminal Operation