Creating Your Own Collector

Understanding How a Collector Works

As we mentioned earlier, the `Collectors` factory class only addresses streams of objects because the `collect()` method that takes a collector object as an argument only exists in `Stream`. If you need to collect a stream of numbers, then you need to understand what are the building elements of a collector.

In a nutshell, a collector is built on four basic components. The first two are used to collect the elements of the stream. The third one is only needed for parallel streams. The fourth one is needed for certain types of collectors, which need a post-processing on the built container.

The first component is used to create the container in which the elements of the stream will be collected. This container is easy to identify. For instance, in the cases we covered in the previous part, we used the `ArrayList` class, the `HashSet` class, or the `HashMap` class. Creating such a container can be modeled with an instance of `Supplier`. This first component is called the *supplier*.

The second component models the adding of a single element from the stream to this container. This operation will be called repeatedly by the implementation of the Stream API, to add all the elements of the stream one by one to the container.

In the Collector API, this component is modeled by an instance of `BiConsumer`. This biconsumer takes two arguments.

1. The first one is the container itself, partially filled with the previous elements of the stream.
2. The second one is the element of the stream that should be added to this partially filled container.

This biconsumer is called the *accumulator* in the context of the Collector API.

These two components should be sufficient for a collector to work, but the Stream API brings a constraint that makes two more components necessary for a collector to work properly.

You may remember that the Stream API supports parallelization. This point will be covered in more detail later in this tutorial. What you need to know is that parallelization splits the elements of your stream in substreams, each one being processed by a core of your CPU. The Collector API can work in such a context: each substream will just be collected in its own instance of the container created by your collector.

Once these substreams have been processed, you have several containers each containing the elements from the sub-stream it processed. These containers are identical, because they have been created with the same *supplier*. Now, you need a way to merge them into one. To be able to do that, the Collector API needs a third

Explore our developer-friendly HTML to PDF API Printed using PDFCrowd HTML to PDF

component, a *combiner*, that will merge these containers together. A combiner is modeled by an instance of `BinaryOperator` that takes two partially filled containers and returns one.

This `BinaryOperator` is also modeled by a `BiConsumer` in the `collect()` overloads of the Stream API.

The fourth component is called the *finisher*, and will be covered later in this part.

Collecting Primitive Types in a Collection

With the first three components, you can use the `collect()` method from the specialized streams of numbers. The `IntStream.collect()` method takes three arguments:

- an instance of `Supplier`, called *supplier*;
- an instance of `ObjIntConsumer`, called *accumulator*;
- an instanceof `BiConsumer`, called *combiner*.

Let us write the code to collect an `IntStream` in a instance of `List<Integer>`.

```
Supplier<List<Integer>> supplier                  = ArrayList::new;
ObjIntConsumer<List<Integer>> accumulator         = Collection::add;
BiConsumer<List<Integer>, List<Integer>> combiner = Collection::addAll;

List<Integer> collect =
    IntStream.range(0, 10)
             .collect(supplier, accumulator, combiner );

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

Running this code produces the following result.

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

Collecting this data in a set would only require changing the implementation of `supplier` and adjusting the types accordingly.

Collecting Primitive Types in a StringBuffer

Let us examine how you can implement an equivalent of the `Collectors.joining()` to join the elements of a stream of primitive types in a single string of characters. The `String` class is immutable, so there is no way you can accumulate elements in it. Instead of using the `String` class, you can use the `StringBuffer` class, which is mutable.

Collecting elements in a `StringBuffer` follows the same pattern as the previous one.

```
Supplier<StringBuffer> supplier                = StringBuffer::new;
ObjIntConsumer<StringBuffer> accumulator       = StringBuffer::append;
```

Explore our developer-friendly HTML to PDF API Printed using PDFCrowd HTML to PDF

```
BiConsumer<StringBuffer, StringBuffer> combiner = StringBuffer::append;

StringBuffer collect =
    IntStream.range(0, 10)
             .collect(supplier, accumulator, combiner);

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

Running this code produces the following result.

```
collect = 0123456789
```

Using a Finisher to Post-Process a Collector

The code you wrote in the previous paragraph is almost doing what you need: it joins strings of characters in an instance of `StringBuffer`, from which you can create a regular `String` object by just calling its `toString()` method. But the `Collectors.joining()` collector produces a `String` directly without having you to call `toString()`. So how is it done?

The Collector API defines a fourth component precisely to handle this case, which is called a *finisher*. A finisher is an instance of `Function` that takes the container in which the elements were accumulated and transforms it to something else. In the case of the `Collectors.joining()`, this function is just the following.

```
Function<StringBuffer, String> finisher = stringBuffer -> stringBuffer.toString();
```

There are many collectors where the finisher is just the identity function. This is the case for the following collectors: `toList()`, `toSet()`, `groupingBy()`, and `toMap()`.

In all the other cases, the mutable container used internally by the collector becomes an intermediate container that will be mapped to some other object, maybe another container, before being returned to the application. This is how the Collector API handles the creation of immutable lists, sets, or maps. The finisher is used to seal the intermediate container into an immutable container before returning it to your application.

There are other uses for this finisher that may improve the readability of your code. The `Collectors` factory class has a factory method, that we have not covered yet: the `collectingAndThen()` method. This method takes a collector as a first argument and a finisher as a second argument. It will just apply this function to the result computed by collecting your stream with the first collector and then map it using the function you provide.

You may remember the following example that we already examined several times in the previous sections. It is about extracting the maximum value of a histogram.

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

Map<Integer, Long> histogram =
    strings.stream()
           .collect(

```

Explore our developer-friendly HTML to PDF API Printed using PDFCrowd HTML to PDF

```
                    Collectors.groupingBy(
                            String::length,
                            Collectors.counting()));

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

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

In a first step, you built an histogram of type `Map<Inter, Long>`, and in a second step, you extracted the maximum value of this histogram, comparing the key-value pairs by value.

This second step is in fact a transformation of a map to a special key/value pair from this map. You can model it using the following function.

```
Function<Map<Integer, Long>, Map.Entry<Integer, Long>> finisher =
    map -> map.entrySet().stream()
              .max(Map.Entry.comparingByValue())
              .orElseThrow();
```

The type of this function may look complex at first. In fact, it just extracts a key-value pair from a map. So it takes an instance of `Map` of a certain type and returns a key-value pair from that map, which is an instance of `Map.Entry` with the same type.

Now that you have this function, you can integrate this maximum value extraction step within the collector itself by using `collectingAndThen()`. The pattern then becomes the following.

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

Function<Map<Integer, Long>, Map.Entry<Integer, Long>> finisher =
    map -> map.entrySet().stream()
              .max(Map.Entry.comparingByValue())
              .orElseThrow();

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

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

You may be wondering why would you need to write this code that looks quite complicated?

Now that you have a maximum value extractor modeled by a single collector, you can use it as a downstream collector for another collector. Being able to do that enables the combination of more collectors to conduct more sophisticated computations on your data.

Explore our developer-friendly HTML to PDF API Printed using PDFCrowd HTML to PDF

Combining the Results of Two Collectors with the Teeing Collector

A method was added in the `Collectors` class in Java SE 12 called `teeing()`. This method takes two downstream collectors and a merging function.

Let us go through a use case to see what you can do with collector. Imagine you have the following `Car` and `Truck` record.

```
enum Color {
    RED, BLUE, WHITE, YELLOW
}

enum Engine {
    ELECTRIC, HYBRID, GAS
}

enum Drive {
    WD2, WD4
}

interface Vehicle {}

record Car(Color color, Engine engine, Drive drive, int passengers) {}

record Truck(Engine engine, Drive drive, int weight) {}
```

A car object has several components: a color, an engine, a drive, and a certain amount of passengers it can transport. A truck has an engine, a drive, and it can transport a certain amount of freight. Both implement the same interface: `Vehicle`.

Suppose you have a collection of vehicles, and you need to find all the car with an electric engine. Depending on your application, you may end up filtering your collection of cars by using a stream. Or, if you know that the next request will be to get the car with a hybrid engine, you may prefer to prepare a map, with the engine as a key, and the list of car with type of engine as values. In both cases, the Stream API will give you the right pattern to get what you need.

Suppose that you need to add to this collection all the electric trucks. It is still possible create this union on one pass your collection of vehicles, but the predicate you will be using to filter your data is becoming more and more complex. It will probably look like the following.

```
Predicate<Vehicle> predicate =
    vehicle -> vehicle instanceof Car car && car.engine() == Engine.ELECTRIC ||
               vehicle instanceof Truck truck && truck.engine() == Engine.ELECTRIC;
```

What you really need is the following:

1. Filter the vehicles to get all the electric cars
2. Filter them to get all the electric trucks
3. Merge the two results.

Explore our developer-friendly HTML to PDF API Printed using PDFCrowd HTML to PDF

This is exactly what the teeing collector can do for you. The teeing collector is created by the `Collectors.teeing()` factory method that takes three arguments.

1. A first downstream collector, used to collect the data of your stream.
2. A second downstream collector, also used to collect your data, in an independent way.
3. A bifunction, used to merge the two containers created by the two downstream collector.

Your data is processed in one pass to guarantee the best performances.

We already covered the pattern that you can use to filter the elements of stream with a collector. The merging function is just a call to the `Collection.addAll()` method. The following is the code:

```
List<Vehicle> electricVehicles = vehicles.stream()
    .collect(
        Collectors.teeing(
            Collectors.filtering(
                vehicle -> vehicle instanceof Car car && car.engine() == Engine.ELECTRIC,
                Collectors.toList()),
            Collectors.filtering(
                vehicle -> vehicle instanceof Truck truck && truck.engine() == Engine.ELECTRIC,
                Collectors.toList()),
            (cars, trucks) -> {
                cars.addAll(trucks);
                return cars;
            }));
```

Last update: *September 14, 2021*

Using a Collector as a Terminal Operation ← **Creating Your Own Collector** → Implementing the Collector Interface

Home > Tutorials > The Stream API > Creating Your Own Collector

Explore our developer-friendly HTML to PDF API Printed using PDFCrowd **HTML to PDF**