Creating Your Own Collector ← Implementing the Collector Interface → Using Optionals

Implementing the Collector Interface

Why Would You Implement the Collector Interface?

There are three ways to create your own collector.

We examined a first one in this tutorial. It consists of combining existing collectors with the different mechanisms offered by the `Collectors` factory class, namely passing a collector as a downstream collector to another collector or using a finisher with the `collectingAndThen()` collector.

You can also call the `collect()` method that takes the three elements upon which a collector is built. These methods are available on both the streams of primitive types and the stream of objects. They take the three arguments we presented in the previous sections.

1. The *supplier* used to create the mutable container in which the elements of the stream are accumulated.
2. The *accumulator*, modeled by a biconsumer.
3. The *combiner*, also modeled by a bi-consumer, used to combine two partially filled containers, used in the case of parallel streams.

The third way consists in implementing the `Collector` interface yourself, and pass your implementation to the `collect()` method we already covered. Implementing your own collector gives you maximum flexibility, but it is also more technical.

Understanding the Parameter Types of Collector

Let us examine the parameters of this interface.

```
interface Collector<T, A, R> {

    // content of the interface
}
```

Let us first examine the following types: `T` and `R`.

The first type is `T`, and it corresponds to the type of the elements of the stream this collector is processing.

The last type is `R`, and it is the type produced by this collector.

For the `toList()` collector, called on an instance of `Stream`, the type `R` would be `List<T>`. It would be `Set<T>` for the `toSet()` collector.

The `groupingBy()` method takes a function to compute the keys of the returned map. If you are collecting a `Stream` with this collector, then you need to pass a function that can map instances of `T`. It can map them to any instance of type `K` to create the keys of the map. So the type of the resulting map will be `Map<K, List<T>>`. So the type `R` is just this one: `Map<K, List<T>>`.

The type `A` is more complex to handle. You may have tried to use your IDE to store one of the collectors you created in the previous examples. If you did that, you probably realized that your IDE did not give an explicit value for this type. This is the case for the following examples.

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

Collector<String, ?, List<String>> listCollector = Collectors.toList();
List<String> list = strings.stream().collect(listCollector);

Collector<String, ?, Set<String>> setCollector = Collectors.toSet();
Set<String> set = strings.stream().collect(setCollector);

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

For all these collectors, the second parameter type is just `?`.

If you need to implement the `Collector` interface, then you will have to give an explicit value to `A`. The type `A` is the type of the intermediate mutable container used by this collector. For the `toList()` collector it would be `ArrayList`, and for the `toSet()` collector it would be `HashSet`. It turns out that this `A` type is hidden by the returned type declared by the `toList()` factory method, which is the reason why you cannot replace the `?` type with `ArrayList<T>` in the previous example.

Even if the internal mutable container is directly returned by the implementation, it may happen that the types `A` and `R` are different. For instance, in the case of the `toList()` collector, you could implement the `Collector<T, A, R>` interface by fixing `ArrayList<T>` for `A` and `List<T>` for `R`.

Understanding the Characteristics of a Collector

A collector defines internal characteristics that are used by the stream implementation to optimize the use of this collector.

There are three characteristics.

1. The `IDENTITY_FINISH` characteristic indicates that the finisher of this collector is the identity function. The implementation will not call the finisher for a collector with this characteristic.
2. The `UNORDERED` characteristic indicates that this collector does not preserve the order in which it processes the elements of the stream. This is the case for the `toSet()` collector, which has this characteristic. The `toList()` collector, on the other hand, does not have it.
3. The `CONCURRENT` characteristic indicates that the container in which the accumulator stores the processed elements supports concurrent access. This point is important for parallel streams.

These characteristics are defined in the `Collector.Characteristics` enumeration and are returned in a set by the `characteristics()` method defined on the `Collector` interface.

Implementing the toList() and toSet() Collector

With these elements, you can now recreate an implementation of a collector that works like the `toList()` collector.

```
class ToList<T> implements Collector<T, List<T>, List<T>> {


    public Supplier<List<T>> supplier() {
        return ArrayList::new;
    }

    public BiConsumer<List<T>, T> accumulator() {
        return Collection::add;
    }

    public BinaryOperator<List<T>> combiner() {
        return (list1, list2) -> {list1.addAll(list2); return list1; };
    }

    public Function<List<T>, List<T>> finisher() {
        return Function.identity();
    }

    public Set<Characteristics> characteristics() {
        return Set.of(Characteristics.IDENTITY_FINISH);
    }
}
```

You can use this collector using the following pattern.

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

List<String> result = strings.stream().collect(new ToList<>());
System.out.println("result = " + result);
```

This code prints out the following result.

```
result = [one, two, three, four, five]
```

Implementing a collector that works like the `toSet()` collector would need only two modifications.

- The `supplier()` method would return `HashSet::new`.
- The `characteristics()` method would add `Characteristics.UNORDERED` to the returned set.

Implementing the joining() Collector

Recreating the implementation of this collector is interesting because it only operates on strings, and its finisher is not the identity function.

This collector accumulates the strings it processes in an instance of `StringBuffer` and then calls the `toString()` method on this accumulator to produce the final result.

The set of characteristics is empty for this collector. It does preserve the order in which the elements are processed (so no *UNORDERED* characteristics), its finisher is not the identity function, and it cannot be used concurrently.

Let us see how this collector could be implemented.

```
class Joining implements Collector<String, StringBuffer, String> {

    public Supplier<StringBuffer> supplier() {
        return StringBuffer::new;
    }

    public BiConsumer<StringBuffer, String> accumulator() {
        return StringBuffer::append;
    }

    public BinaryOperator<StringBuffer> combiner() {
        return StringBuffer::append;
    }

    public Function<StringBuffer, String> finisher() {
        return Object::toString;
    }

    public Set<Characteristics> characteristics() {
        return Set.of();
    }
}
```

You can see how this collector can be used in the following example.

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

String result = strings.stream().collect(new Joining());
System.out.println("result = " + result);
```

Running this code produces the following result.

```
result = onetwothreefourfive
```

Supporting a delimiter, a prefix, and a suffix would use a `StringJoiner` instead of a `StringBuilder`, which already supports these elements.

Last update: *September 14, 2021*

Creating Your Own Collector ← **Implementing the Collector Interface** → Using Optionals

Home > Tutorials > The Stream API > Implementing the Collector Interface