(http://baeldung.com)

Guide to Java 8’s Collectors

Last modified: August 31, 2017

by Grzegorz Piwowarek (http://www.baeldung.com/author/grzegorz-author/)

Java (http://www.baeldung.com/category/java/) +

1. Overview

In this tutorial, we will be going through Java 8’s Collectors, which are used at the final step of processing a *Stream*.

If you want to read more about *Stream* API itself, check this article (/java-8-streams).

2. The *Stream.collect()* Method

Stream.collect() is one of the Java 8’s *Stream API*'s terminal methods. It allows to perform mutable fold operations (repackaging elements to some data structures and applying some additional logic, concatenating them, etc.) on data elements held in a *Stream* instance.

The strategy for this operation is provided via *Collector* interface implementation.

3. *Collectors*

All predefined implementations can be found in the *Collectors* class. It’s a common practice to use a following static import with them to leverage increased readability:

```
import static java.util.stream.Collectors.*;
```

or just single import collectors of your choice:

```
import static java.util.stream.Collectors.toList;
import static java.util.stream.Collectors.toMap;
import static java.util.stream.Collectors.toSet;
```

In the following examples we will be reusing the following list:

```
List<String> givenList = Arrays.asList("a", "bb", "ccc", "dd");
```

3.1. *Collectors.toList()*

ToList collector can be used for collecting all *Stream* elements into a *List* instance. The important thing to remember is the fact that we can't assume any particular *List* implementation with this method. If you want to have more control over this, use *toCollection* instead.

Let's create a *Stream* instance representing a sequence of elements and collect them into a *List* instance:

```
List<String> result = givenList.stream()
  .collect(toList());
```

3.2. *Collectors.toSet()*

ToSet collector can be used for collecting all *Stream* elements into a *Set* instance. The important thing to remember is the fact that we can't assume any particular *Set* implementation with this method. If you want to have more control over this, use *toCollection* instead.

Let's create a Stream instance representing a sequence of elements and collect them into a *Set* instance:

```
Set<String> result = givenList.stream()
  .collect(toSet());
```

3.3. *Collectors.toCollection()*

As you probably already noticed, when using *toSet and toList* collectors, you can't make any assumptions of their implementations. If you want to use a custom implementation, you will need to use the *toCollection* collector with a provided collection of your choice.

Let's create a *Stream* instance representing a sequence of elements and collect them into a *LinkedList* instance:

```
List<String> result = givenList.stream()
  .collect(toCollection(LinkedList::new))
```

Notice that this will not work with any immutable collections. In such case, you would need to either write a custom *Collector* implementation or use *collectingAndThen*.

3.4. *Collectors.toMap()*

ToMap collector can be used to collect *Stream* elements into a *Map* instance. To do this, you need to provide two functions:

- keyMapper
- valueMapper

keyMapper will be used for extracting a *Map* key from a *Stream* element, and *valueMapper* will be used for extracting a value associated with a given key.

Let's collect those elements into a Map that stores strings as keys and their lengths as values:

```
Map<String, Integer> result = givenList.stream()
  .collect(toMap(Function.identity(), String::length))
```

Function.identity() is just a shortcut for defining function that accepts and return the same value;

Sometimes you might encounter a situation where you might end up with a key collision. In such case, you should use *toMap* with another signature.

```
Map<String, Integer> result = givenList.stream()
  .collect(toMap(Function.identity(), String::length, (i1, i2) -> i1));
```

The third argument here is a *BinaryOperator*, where you can specify how you want collisions to be handled. In this case, we will just pick any of these two colliding values because we know that same strings will always have same lengths too.

3.5. *Collectors.collectingAndThen()*

CollectingAndThen is a special collector that allows performing another action on a result straight after collecting ends.

Let's collect *Stream* elements to a *List* instance and then convert the result into an *ImmutableList* instance:

```
List<String> result = givenList.stream()
  .collect(collectingAndThen(toList(), ImmutableList::copyOf))
```

3.6. *Collectors.joining()*

Joining collector can be used for joining *Stream<String>* elements.

We can join them together by doing:

```
String result = givenList.stream()
  .collect(joining());
```

which will result in:

```
"abbcccdd"
```

You can also specify custom separators, prefixes, postfixes:

```
String result = givenList.stream()
  .collect(joining(" "));
```

which will result in:

```
"a bb ccc dd"
```

or you can write:

```
String result = givenList.stream()
  .collect(joining(" ", "PRE-", "-POST"));
```

which will result in:

```
"PRE-a bb ccc dd-POST"
```

3.7. *Collectors.counting()*

Counting is a simple collector that allows simply counting of all *Stream* elements.

Now we can write:

```
Long result = givenList.stream()
  .collect(counting());
```

3.8. *Collectors.summarizingDouble/Long/Int()*

SummarizingDouble/Long/Int is a collector that returns a special class containing statistical information about numerical data in a *Stream* of extracted elements.

We can obtain information about string lengths by doing:

```
DoubleSummaryStatistics result = givenList.stream()
  .collect(summarizingDouble(String::length));
```

In this case, the following will be true:

```
assertThat(result.getAverage()).isEqualTo(2);
assertThat(result.getCount()).isEqualTo(4);
assertThat(result.getMax()).isEqualTo(3);
assertThat(result.getMin()).isEqualTo(1);
assertThat(result.getSum()).isEqualTo(8);
```

3.9. *Collectors.averagingDouble/Long/Int()*

AveragingDouble/Long/Int is a collector that simply returns an average of extracted elements.

We can get average string length by doing:

```
Double result = givenList.stream()
  .collect(averagingDouble(String::length));
```

3.10. *Collectors.summingDouble/Long/Int()*

SummingDouble/Long/Int is a collector that simply returns a sum of extracted elements.

We can get a sum of all string lengths by doing:

```
Double result = givenList.stream()
  .collect(summingDouble(String::length));
```

3.11. *Collectors.maxBy()/minBy()*

MaxBy/*MinBy* collectors return the biggest/the smallest element of a *Stream* according to a provided *Comparator* instance.

We can pick the biggest element by doing:

```
Optional<String> result = givenList.stream()
  .collect(maxBy(Comparator.naturalOrder()));
```

Notice that returned value is wrapped in an *Optional* instance. This forces users to rethink the empty collection corner case.

3.12. *Collectors.groupingBy()*

GroupingBy collector is used for grouping objects by some property and storing results in a *Map* instance.

We can group them by string length and store grouping results in *Set* instances:

```
Map<Integer, Set<String>> result = givenList.stream()
  .collect(groupingBy(String::length, toSet()));
```

This will result in following being true:

```
assertThat(result)
  .containsEntry(1, newHashSet("a"))
  .containsEntry(2, newHashSet("bb", "dd"))
  .containsEntry(3, newHashSet("ccc"));
```

Notice that the second argument of the *groupingBy* method is a *Collector* and you are free to use any *Collector* of your choice.

3.13. *Collectors.partitioningBy()*

PartitioningBy is a specialized case of *groupingBy* that accepts a *Predicate* instance and collects *Stream* elements into a *Map* instance that stores *Boolean* values as keys and collections as values. Under the "true" key, you can find a collection of elements matching the given *Predicate*, and under the "false" key, you can find a collection of elements not matching the given *Predicate*.

You can write:

```
Map<Boolean, List<String>> result = givenList.stream()
  .collect(partitioningBy(s -> s.length() > 2))
```

Which results in a Map containing:

```
{false=["a", "bb", "dd"], true=["ccc"]}
```

4. Custom Collectors

If you want to write your Collector implementation, you need to implement Collector interface and specify its three generic parameters:

```
public interface Collector<T, A, R> {...}
```

1. **T** – the type of objects that will be available for collection,
2. **A** – the type of a mutable accumulator object,
3. **R** – the type of a final result.

Let's write an example Collector for collecting elements into an *ImmutableSet* instance. We start by specifying the right types:

```
private class ImmutableSetCollector<T>
  implements Collector<T, ImmutableSet.Builder<T>, ImmutableSet<T>> {...}
```

Since we need a mutable collection for internal collection operation handling, we can't use *ImmutableSet* for this; we need to use some other mutable collection or any other class that could temporarily accumulate objects for us.
In this case, we will go on with an *ImmutableSet.Builder* and now we need to implement 5 methods:

- *Supplier<ImmutableSet.Builder<T>>* ***supplier****()*
- *BiConsumer<ImmutableSet.Builder<T>, T>* ***accumulator****()*
- *BinaryOperator<ImmutableSet.Builder<T>>* ***combiner****()*
- *Function<ImmutableSet.Builder<T>, ImmutableSet<T>>* ***finisher****()*
- *Set<Characteristics>* ***characteristics****()*

The supplier() method returns a *Supplier* instance that generates an empty accumulator instance, so, in this case, we can simply write:

```
@Override
public Supplier<ImmutableSet.Builder<T>> supplier() {
    return ImmutableSet::builder;
}
```

The accumulator() method returns a function that is used for adding a new element to an existing *accumulator* object, so let's just use the *Builder*'s *add* method.

```
@Override
public BiConsumer<ImmutableSet.Builder<T>, T> accumulator() {
    return ImmutableSet.Builder::add;
}
```

The combiner() method returns a function that is used for merging two accumulators together:

```
@Override
public BinaryOperator<ImmutableSet.Builder<T>> combiner() {
    return (left, right) -> left.addAll(right.build());
}
```

The finisher() method returns a function that is used for converting an accumulator to final result type, so in this case, we will just use *Builder*'s *build* method:

```
@Override
public Function<ImmutableSet.Builder<T>, ImmutableSet<T>> finisher() {
    return ImmutableSet.Builder::build;
}
```

The characteristics() method is used to provide Stream with some additional information that will be used for internal optimizations. In this case, we do not pay attention to the elements order in a *Set* so that we will use *Characteristics.UNORDERED*. To obtain more information regarding this subject, check *Characteristics*' JavaDoc.

```
@Override public Set<Characteristics> characteristics() {
    return Sets.immutableEnumSet(Characteristics.UNORDERED);
}
```

Here is the complete implementation along with the usage:

```
public class ImmutableSetCollector<T>
  implements Collector<T, ImmutableSet.Builder<T>, ImmutableSet<T>> {

@Override
public Supplier<ImmutableSet.Builder<T>> supplier() {
    return ImmutableSet::builder;
}

@Override
public BiConsumer<ImmutableSet.Builder<T>, T> accumulator() {
    return ImmutableSet.Builder::add;
}

@Override
public BinaryOperator<ImmutableSet.Builder<T>> combiner() {
    return (left, right) -> left.addAll(right.build());
}

@Override
public Function<ImmutableSet.Builder<T>, ImmutableSet<T>> finisher() {
    return ImmutableSet.Builder::build;
}

@Override
public Set<Characteristics> characteristics() {
    return Sets.immutableEnumSet(Characteristics.UNORDERED);
}

public static <T> ImmutableSetCollector<T> toImmutableSet() {
    return new ImmutableSetCollector<>();
}
```

and here in action:

```
List<String> givenList = Arrays.asList("a", "bb", "ccc", "dddd");

ImmutableSet<String> result = givenList.stream()
  .collect(toImmutableSet());
```

5. Conclusion

In this article, we explored in depth Java 8's *Collectors* and showed how to implement one.

All code examples are available on the GitHub (https://github.com/eugenp/tutorials/tree/master/core-java-8). You can read more interesting articles on my site (http://4comprehension.com).

Sort by: newest|oldest|most voted

Binh Thanh Nguyen

Guest

Thanks, nice tips

👍 0 👎

1 year 1 month ago

Shivang

Guest

Very helpful article; thanks! 🙂

Minor issue: Given that the example list is defined as:

List givenList = Arrays.asList("a", "bb", "ccc", "dd");

It appears the result in the example in section 3.12, "Collectors.groupingBy()" is wrong. The result map should only contain entries for 1, 2, and 3.

Either that, or I'm missing something… 🙂

👍 0 👎

1 year 22 days ago

Grzegorz Piwowarek

Guest

You are right. Actually, code on GH is fine. We will update the article. Thanks!
https://github.com/eugenp/tutorials/blob/master/core-java-8/src/test/java/com/baeldung/collectors/Java8CollectorsTest.java#L216
(https://github.com/eugenp/tutorials/blob/master/core-java-8/src/test/java/com/baeldung/collectors/Java8CollectorsTest.java#L216)

👍 0 👎

1 year 22 days ago

Svitlana Belykova

Guest

Thanks for this article!

👍 0 👎

1 year 1 day ago

Eugen Paraschiv
(http://www.baeldung.com/)

Guest

Sure thing, happy to help.

Cheers,
Eugen.

👍 0 👎

11 months 28 days ago

Ashwani kumar

Guest

Awesome! very helpful 🙂
Collectors's concepts collected here like terminal method collect of stream.
List concepts = findOverviewCollectorsConcepts();
Set cncpts = concepts.stream().collect(baeldungCollectorsArticle,Collectors.toSet());
Any guess why baeldungCollectorsArticle can be treated as java.util.stream.Collector ?

👍 0 👎

7 months 17 days ago

Grzegorz Piwowarek

Guest

Ashwani, but what "baeldungCollectorsArticle" is? You did not define it in your snippet and it's not present in the article itself

👍 0 👎

7 months 17 days ago

Ashwani kumar

Guest

Grzegorz, baeldungCollectorsArticle is an example of Metaphor Figure of speech. 'baeldungCollectorsArticle' is nothing but this blog post.
And by mean of provided snippet I want to say:-

"All the Supplied overview Java8 collectors concepts in this blog post(which I'm saying 'baeldungCollectorsArticle') are well Accumulated for overview and Combined in a single post with conclusion as Finisher and having clear & to the point Characteristics."
In short this article according to me have Supplier, Accumulator, Combiner, Finisher and Characteristics.

👍 0 👎 7 months 17 days ago

Guest

Grzegorz Piwowarek

Ok 😃 sorry, when I saw code I automatically assumed that some mistake was in the article.

👍 0 👎 7 months 17 days ago

Guest

Mehraj Malik

That's very useful.
Great efforts 🙂
However, It took me some time to find that you done ** import static **.
Btw, stream().collect(Collectors.asList()); is more readable than static import.

👍 0 👎 6 months 22 days ago

Guest

Grzegorz Piwowarek

Thanks 🙂
Actually, collectors were designed to be used with static imports. This allows you to get something very natural sounding like "collect to list". I mentioned this in the article at the beginning of the section 3

👍 0 👎 6 months 22 days ago

CATEGORIES

SPRING (HTTP://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTP://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTP://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTP://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTP://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTP://WWW.BAELDUNG.COM/CATEGORY/JACKSON/)
HTTPCLIENT (HTTP://WWW.BAELDUNG.COM/CATEGORY/HTTP/)

ABOUT

ABOUT BAELDUNG (HTTP://WWW.BAELDUNG.COM/ABOUT/)
THE COURSES (HTTP://COURSES.BAELDUNG.COM)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
THE FULL ARCHIVE (HTTP://WWW.BAELDUNG.COM/FULL_ARCHIVE)
WRITE FOR BAELDUNG (HTTP://WWW.BAELDUNG.COM/CONTRIBUTION-GUIDELINES)
PRIVACY POLICY (HTTP://WWW.BAELDUNG.COM/PRIVACY-POLICY)
TERMS OF SERVICE (HTTP://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)
CONTACT (HTTP://WWW.BAELDUNG.COM/CONTACT)
COMPANY INFO (HTTP://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO)
ADVERTISE ON THE JAVA WEEKLY (HTTP://WWW.BAELDUNG.COM/JAVA-WEEKLY-SPONSORSHIP)