Guide to Java 8 groupingBy Collector

Last updated: January 9, 2024

Written by: Devansh Trivedi (https://www.baeldung.com/author/devanshtrivedi)

Reviewed by: Sunil Mogadati (https://www.baeldung.com/editor/sunil-author)

Java Streams (https://www.baeldung.com/category/java/java-streams)

>= Java 8 (https://www.baeldung.com/tag/jdk8-and-later)

(/)

Get started with Spring and Spring Boot, through the *Learn Spring* course:

>> CHECK OUT THE COURSE (/ls-course-start)

1. Introduction

In this tutorial, **we'll see how the *groupingBy* collector works using various examples.**

To better understand this tutorial, we'll need a basic knowledge of Java 8 features. Have a look at the intro to Java 8 Streams (/java-8-streams-introduction) and the guide to Java 8's Collectors (/java-8-collectors) for these basics.

Further reading:

Introduction to Java 8 Streams (/java-8-streams-introduction)

A quick and practical introduction to Java 8 Streams.

Read more (/java-8-streams-introduction) →

Guide to Stream.reduce() (/java-stream-reduce)

Learn the key concepts of the Stream.reduce() operation in Java and how to use it to process sequential and parallel streams.

Read more (/java-stream-reduce) →

The Difference Between map() and flatMap() (/java-difference-map-and-flatmap)

Learn about the differences between map() and flatMap() by analyzing some examples of Streams and Optionals.

Read more (/java-difference-map-and-flatmap) →

2. *groupingBy* Collectors

The Java 8 *Stream* API lets us process collections of data in a declarative way.

The static factory methods *Collectors.groupingBy()* and *Collectors.groupingByConcurrent()* provide us with functionality similar to the '*GROUP BY*' clause in the SQL language. **We use them for grouping objects by some property and storing results in a *Map* instance.**

The overloaded methods of *groupingBy* are:

- First, with a classification function as the method parameter:

```
static <T,K> Collector<T,?,Map<K,List<T>>>
  groupingBy(Function<? super T,? extends K> classifier)
```

- Second, with a classification function and a second collector as method parameters:

```
static <T,K,A,D> Collector<T,?,Map<K,D>>
  groupingBy(Function<? super T,? extends K> classifier,
    Collector<? super T,A,D> downstream)
```

- Finally, with a classification function, a supplier method (that provides the *Map* implementation which contains the end result), and a second collector as method parameters:

```
static <T,K,D,A,M extends Map<K,D>> Collector<T,?,M>
  groupingBy(Function<? super T,? extends K> classifier,
    Supplier<M> mapFactory, Collector<? super T,A,D> downstream)
```

2.1. Example Code Setup

To demonstrate the usage of *groupingBy()*, let's define a *BlogPost* class (we will use a stream of *BlogPost* objects):

```
class BlogPost {
    String title;
    String author;
    BlogPostType type;
    int likes;
}
```

Next, the *BlogPostType*:

```
enum BlogPostType {
    NEWS,
    REVIEW,
    GUIDE
}
```

Then the *List* of *BlogPost* objects:

```
List<BlogPost> posts = Arrays.asList( ... );
```

Let's also define a *Tuple* class to group posts by the combination of their *type* and *author* attributes:

```
class Tuple {
    BlogPostType type;
    String author;
}
```

2.2. Simple Grouping by a Single Column

Let's start with the simplest *groupingBy* method, which only takes a classification function as its parameter. A classification function is applied to each element of the stream.

We use the value returned by the function as a key to the map that we get from the *groupingBy* collector.

To group the blog posts in the blog post list by their *type*:

```
Map<BlogPostType, List<BlogPost>> postsPerType = posts.stream()
  .collect(groupingBy(BlogPost::getType));
```

2.3. *groupingBy* with a Complex *Map* Key Type

The classification function is not limited to returning only a scalar or String value. The key of the resulting map could be any object as long as we make sure that we implement the necessary *equals* and *hashcode* methods.

To **group using two fields as keys, we can use the *Pair* class provided in the *javafx.util* or *org.apache.commons.lang3.tuple* packages**.

For example, to group the blog posts in the list, by the type and author combined in an Apache Commons *Pair* instance:

```
Map<Pair<BlogPostType, String>, List<BlogPost>>
postsPerTypeAndAuthor = posts.stream()
  .collect(groupingBy(post -> new ImmutablePair<>(post.getType(),
post.getAuthor())));
```

Similarly, we can use the Tuple class defined before, this class can be easily generalized to include more fields as needed. The previous example using a Tuple instance will be:

```
Map<Tuple, List<BlogPost>> postsPerTypeAndAuthor = posts.stream()
  .collect(groupingBy(post -> new Tuple(post.getType(),
post.getAuthor())));
```

Java 16 has introduced the concept of a *record* (/java-record-keyword) as a new form of generating immutable Java classes.

The *record* feature provides us with a simpler, clearer, and safer way to do *groupingBy* than the Tuple. For example, we have defined a *record* instance in the *BlogPost*:

```
public class BlogPost {                    (/)
    private String title;
    private String author;
    private BlogPostType type;
    private int likes;
    record AuthPostTypesLikes(String author, BlogPostType type, int
likes) {};

    // constructor, getters/setters
}
```

Now it's very simple to group the *BlotPost* in the list by the type, author, and likes using the *record* instance:

```
Map<BlogPost.AuthPostTypesLikes, List<BlogPost>>
postsPerTypeAndAuthor = posts.stream()
  .collect(groupingBy(post -> new
BlogPost.AuthPostTypesLikes(post.getAuthor(), post.getType(),
post.getLikes())));
```

2.4. Modifying the Returned *Map* Value Type

The second overload of *groupingBy* takes an additional second collector (downstream collector) that is applied to the results of the first collector.

When we specify a classification function, but not a downstream collector, the *toList()* collector is used behind the scenes.

Let's use the *toSet()* collector as the downstream collector and get a *Set* of blog posts (instead of a *List*):

```
Map<BlogPostType, Set<BlogPost>> postsPerType = posts.stream()
  .collect(groupingBy(BlogPost::getType, toSet()));
```

2.5. Grouping by Multiple Fields

A different application of the downstream collector is to do a secondary *groupingBy* to the results of the first group by.

To group the *List* of *BlogPosts* first by *author* and then by *type*:

```
Map<String, Map<BlogPostType, List>> map = posts.stream()
  .collect(groupingBy(BlogPost::getAuthor,
groupingBy(BlogPost::getType)));
```

2.6. Getting the Average from Grouped Results

By using the downstream collector, we can apply aggregation functions in the results of the classification function.

For instance, to find the average number of *likes* for each blog post *type*:

```
Map<BlogPostType, Double> averageLikesPerType = posts.stream()
  .collect(groupingBy(BlogPost::getType,
averagingInt(BlogPost::getLikes)));
```

2.7. Getting the Sum from Grouped Results

To calculate the total sum of *likes* for each *type*:

```
Map<BlogPostType, Integer> likesPerType = posts.stream()
  .collect(groupingBy(BlogPost::getType,
summingInt(BlogPost::getLikes)));
```

2.8. Getting the Maximum or Minimum from Grouped Results

Another aggregation that we can perform is to get the blog post with the maximum number of likes:

```
Map<BlogPostType, Optional<BlogPost>> maxLikesPerPostType =
posts.stream()
  .collect(groupingBy(BlogPost::getType,
  maxBy(comparingInt(BlogPost::getLikes))));
```

Similarly, we can apply the *minBy* downstream collector to obtain the blog post with the minimum number of *likes*.

Note that the *maxBy* and *minBy* collectors take into account the possibility that the collection to which they are applied could be empty. This is why the value type in the map is *Optional<BlogPost>*.

2.9. Getting a Summary for an Attribute of Grouped Results

The *Collectors* API offers a summarizing collector that we can use in cases when we need to calculate the count, sum, minimum, maximum and average of a numerical attribute at the same time.

Let's calculate a summary for the likes attribute of the blog posts for each different type:

```
Map<BlogPostType, IntSummaryStatistics> likeStatisticsPerType =
posts.stream()
  .collect(groupingBy(BlogPost::getType,
  summarizingInt(BlogPost::getLikes)));
```

The *IntSummaryStatistics* object for each type contains the count, sum, average, min and max values for the *likes* attribute. Additional summary objects exist for double and long values.

2.10. Aggregating Multiple Attributes of a Grouped Result

In the previous sections we've seen how to aggregate one field at a time. **There are some techniques that we can follow to do aggregations over multiple fields**.

The first approach is to use *Collectors::collectingAndThen* for the downstream collector of *groupingBy*. For the first parameter of *collectingAndThen* we collect the stream into a list, using *Collectors::toList*. The second parameter applies the finishing transformation, we can use it with any of the *Collectors'* class methods that support aggregations to get our desired results.

For example, let's group by *author* and for each one we count the number of *titles*, list the *titles*, and provide a summary statistics of the *likes*. To accomplish this, we start by adding a new record to the *BlogPost*:

```
public class BlogPost {
    // ...
    record PostCountTitlesLikesStats(long postCount, String titles,
IntSummaryStatistics likesStats){};
     // ...
}
```

The implementation of *groupingBy* and *collectingAndThen* will be:

```
Map<String, BlogPost.PostCountTitlesLikesStats> postsPerAuthor =
posts.stream()
  .collect(groupingBy(BlogPost::getAuthor,
collectingAndThen(toList(), list -> {
    long count = list.stream()
      .map(BlogPost::getTitle)
      .collect(counting());
    String titles = list.stream()
      .map(BlogPost::getTitle)
      .collect(joining(" : "));
    IntSummaryStatistics summary = list.stream()
      .collect(summarizingInt(BlogPost::getLikes));
    return new BlogPost.PostCountTitlesLikesStats(count, titles,
summary);
  })));
```

In the first parameter of collectingAndThen we get a list of *BlogPost*. We use it in the finishing transformation as an input to the lambda function to calculate the values to generate *PostCountTitlesLikesStats*.

To get the information for a given *author* is as simple as:

```
BlogPost.TitlesBoundedSumOfLikesStats result =
postsPerAuthor.get("Author 1");
assertThat(result.postCount()).isEqualTo(3L);
assertThat(result.titles()).isEqualTo("News item 1 : Programming
guide : Tech review 2");
assertThat(result.likesStats().getMax()).isEqualTo(20);
assertThat(result.likesStats().getMin()).isEqualTo(15);
assertThat(result.likesStats().getAverage()).isEqualTo(16.666d,
offset(0.001d));
```

We can also do more sophisticated aggregations if we use *Collectors::toMap* to collect and aggregate the elements of the stream.

Let's consider a simple example where we want to group the *BlogPost* elements by *author* and concatenate the *titles* with an upper bounded sum of *like* scores.

First, we create the record that is going to encapsulate our aggregated result:

```
public class BlogPost {
    // ...
    record TitlesBoundedSumOfLikes(String titles, int
boundedSumOfLikes) {};
    // ...
}
```

Then we group and accumulate the stream in the following manner:

```
int maxValLikes = 17;
Map<String, BlogPost.TitlesBoundedSumOfLikes> postsPerAuthor =
posts.stream()
  .collect(toMap(BlogPost::getAuthor, post -> {
    int likes = (post.getLikes() > maxValLikes) ? maxValLikes :
post.getLikes();
    return new BlogPost.TitlesBoundedSumOfLikes(post.getTitle(),
likes);
  }, (u1, u2) -> {
    int likes = (u2.boundedSumOfLikes() > maxValLikes) ?
maxValLikes : u2.boundedSumOfLikes();
    return new
BlogPost.TitlesBoundedSumOfLikes(u1.titles().toUpperCase() + " : "
+ u2.titles().toUpperCase(), u1.boundedSumOfLikes() + likes);
  }));
```

The first parameter of *toMap* groups the keys applying *BlogPost::getAuthor*.

The second parameter transforms the values of the map using the lambda function to convert each *BlogPost* into a *TitlesBoundedSumOfLikes* record.

The third parameter of *toMap* deals with duplicate elements for a given key and here we use another lambda function to concatenate the *titles* and sum the *likes* with a max allowed value specified in *maxValLikes*.

2.11. Mapping Grouped Results to a Different Type

We can achieve more complex aggregations by applying a *mapping* downstream collector to the results of the classification function.

Let's get a concatenation of the *title*s of the posts for each blog post *type*:

```
Map<BlogPostType, String> postsPerType = posts.stream()
  .collect(groupingBy(BlogPost::getType,
  mapping(BlogPost::getTitle, joining(", ", "Post titles: [",
"]"))));
```

What we have done here is to map each *BlogPost* instance to its *title* and then reduce the stream of post titles to a concatenated *String*. In this example, the type of the *Map* value is also different from the default *List* type.

2.12. Modifying the Return *Map* Type

When using the *groupingBy* collector, we cannot make assumptions about the type of the returned *Map*. If we want to be specific about which type of *Map* we want to get from the group by, then we can use the third variation of the *groupingBy* method that allows us to change the type of the *Map* by passing a *Map* supplier function.

Let's retrieve an *EnumMap* by passing an *EnumMap* supplier function to the *groupingBy* method:

```
EnumMap<BlogPostType, List<BlogPost>> postsPerType = posts.stream()
  .collect(groupingBy(BlogPost::getType,
  () -> new EnumMap<>(BlogPostType.class), toList()));
```

3. Concurrent *groupingBy* Collector

Similar to *groupingBy* is the *groupingByConcurrent* collector, which leverages multi-core architectures. This collector has three overloaded methods that take exactly the same arguments as the respective overloaded methods of the *groupingBy* collector. The return type of the *groupingByConcurrent* collector, however, must be an instance of the *ConcurrentHashMap* class or a subclass of it.

To do a grouping operation concurrently, the stream needs to be parallel:

```
ConcurrentMap<BlogPostType, List<BlogPost>> postsPerType =
posts.parallelStream()
  .collect(groupingByConcurrent(BlogPost::getType));
```

If we choose to pass a *Map* supplier function to the *groupingByConcurrent* collector, then we need to make sure that the function returns either a *ConcurrentHashMap* or a subclass of it.

4. Java 9 Additions

Java 9 introduced two new collectors that work well with *groupingBy*, more information about them can be found here (/java9-stream-collectors).

5. Conclusion

In this article, we explored the usage of the *groupingBy* collector offered by the Java 8 *Collectors* API.

We learned how *groupingBy* can be used to classify a stream of elements based on one of their attributes, and how the results of this classification can be further collected, mutated, and reduced to final containers.

The complete implementation of the examples in this article can be found in the GitHub project (https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-streams-simple).

Comments are closed on this article!

ALL BULK COURSES (/ALL-BULK-COURSES)
ALL BULK TEAM COURSES (/ALL-BULK-TEAM-COURSES)
THE COURSES PLATFORM (HTTPS://COURSES.BAELDUNG.COM)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
APACHE HTTPCLIENT TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)
SPRING REACTIVE TUTORIALS (/SPRING-REACTIVE-GUIDE)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE FULL ARCHIVE (/FULL_ARCHIVE)
EDITORS (/EDITORS)
JOBS (/TAG/ACTIVE-JOB/)
OUR PARTNERS (/PARTNERS)
PARTNER WITH BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)