Grouping Elements With Java Streams

One of the most interesting features introduced with Java Streams is the ability of grouping collections easily by using the [groupingBy](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html" \l "groupingBy-java.util.function.Function-" \t "_blank) collector.  
Although in some cases it could be pretty straightforward, there are different combinations of ***groupingBy*** and some of them are not so obvious to understand for many developers, so that’s why I’ve decided that it’d be beneficial to take a deep look at them together.

First of all, let’s start by looking at what the Java API offers to us in [Collectors](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html) class.

**Introduction**

Java offers a combination of collectors to group elements of a Java Stream, let’s list them all before we continue:

* [groupingBy(classifier)](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html#groupingBy-java.util.function.Function-)  
  Groups elements using a function to assign a key to each element; this key will specify what group each element belongs to.
* [groupingBy(classifier, downstream)](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html#groupingBy-java.util.function.Function-java.util.stream.Collector-)  
  Same as previous method but we can transform the elements assigned to each key in many ways.
* [groupingBy(classifier, mapFactory, downstream)](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html#groupingBy-java.util.function.Function-java.util.function.Supplier-java.util.stream.Collector-)  
  Same as previous method but also we can provide how to initialise the Map we use to store our data.
* [groupingByConcurrent(classifier)](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html#groupingByConcurrent-java.util.function.Function-)
* [groupingByConcurrent(classifier, downstream)](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html#groupingByConcurrent-java.util.function.Function-java.util.stream.Collector-)
* [groupingByConcurrent(classifier, mapFactory, downstream)](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html#groupingByConcurrent-java.util.function.Function-java.util.function.Supplier-java.util.stream.Collector-)

As you can see, there are three different types of collectors and on top of those we have a “concurrent” version for each of them.

Nothing better than an example to explain each case clearly, let’s start!

**Group elements in a Map by key**

The first method what it does is to take each element in our collection and assign it to a key using a ***classifier***; our collector will place each of these elements in a Java [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html), associating a [List](https://docs.oracle.com/javase/8/docs/api/java/util/List.html) to each existing key in the [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html).

I know it might sound confusing, so let’s see it with an example. We are going to group our list of employees by sex. Let’s see how would it look like:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | final Collection<Employee> employees = List.of( | |  | new Employee("Karen Smith", 51200.0, 29, Employee.Sex.FEMALE), | |  | new Employee("John Smith", 24000.0, 32, Employee.Sex.MALE), | |  | new Employee("Anthony Jackson", 44000.0, 33, Employee.Sex.MALE), | |  | new Employee("Alyson Palmer", 34320.0, 36, Employee.Sex.FEMALE), | |  | new Employee("Jessica Sanders", 64320.0, 34, Employee.Sex.FEMALE) | |  | ); | |  |  | |  | final Map<Employee.Sex, List<Employee>> employeesBySex = employees.stream() | |  | .collect(groupingBy(Employee::getSex)); | |  |  | |  | System.out.println(employeesBySex); | |

This is the simplest of the cases, we just specify a classifier, which is a Function that accepts each element in the collection and returns a key; this key will be the value being used to assign each element to a given group.

Please also notice that the key that we use to classify each element doesn’t need to be a Java simple type, it can be any object that we might need. We could even create our own object to **classify the elements based on a combination of fields.**

In our example there are only two groups, because the “sex” field has only two values: **MALE** and **FEMALE**. So as a result what we get back is a **Map<Sex, List<Employee>>**.  
That’s the default structure that we’ll get when we use [groupingBy](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html" \l "groupingBy-java.util.function.Function-" \t "_blank) collector in Java (**Map<Key, List<Element>>**).

Let’s see now what happens if we need to modify this default structure.

**Grouping by key specifying a different collector**

Let’s say that instead of storing our results in a List, as our groupingBy collector does by default, we want to store our elements in a Set; for instance, we might want to avoid duplicates.

In that case a different method is provided which accepts a “downstream” collector that we can use to transform our resulting collection for each group as we see fit. In our example this will be really easy:

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | final Map<Employee.Sex, Set<Employee>> employeesBySex = employees.stream() | |  | .collect(groupingBy(Employee::getSex, toSet())); | |

As you can see, the collections are now stored in a [Set](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html) instead. What if we want to collect this data differently? We might want to use any kind of aggregation operation; for instance, we might need to calculate the average age for each group. We can use averagingInt collector in Collectors class to achieve that.

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | final Map<Employee.Sex, Double> averageAgeBySex = employees.stream() | |  | .collect(groupingBy(Employee::getSex, averagingInt(Employee::getAge))); | |

Please notice that in this case we don’t assign a collection to each key in the map, just a double as a result of calculating the average age for each group.

What else does Java allow us to do? We could also **group by a secondary field!** Imagine that we want to group by sex first and then by age, as easy as this:

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | final Map<Employee.Sex, Map<Integer, List<Employee>>> groupBySexAndAge = employees.stream() | |  | .collect(groupingBy(Employee::getSex, groupingBy(Employee::getAge))); | |

There are multiple combinations we can use with these collectors, the API that Java provides to us to treat collections easily is very powerful.

Let’s see one more example; what if we want to find the youngest employee in each age group? Very easy as well, as simple as this:

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | final Map<Employee.Sex, Optional<Employee>> youngestEmployeeBySex = employees.stream() | |  | .collect(groupingBy(Employee::getSex, minBy(comparing(Employee::getAge)))); | |

We’re using [minBy](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html" \l "minBy-java.util.Comparator-" \t "_blank) collector, which accepts a [Comparator](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html); Java provides a very easy way to provide comparators now by using [Comparator.comparing(keyExtractor)](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html" \l "comparing-java.util.function.Function-" \t "_blank). KeyExtractor is a [Function](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html), so passing in a method reference is enough in this case.

That’s brilliant, right? Manipulating data using Java Streams using a functional style has made our lives so much easier as developers!

We could even filter some employees based on a condition after they’re grouped; for example, let’s get all the employees over thirty grouped by sex:

|  |
| --- |
| final Map<Employee.Sex, List<Employee>> employeesOverThirtyBySex = employees.stream() .collect(groupingBy(Employee::getSex, filtering(employee -> employee.getAge() > 30, toList()))); |

Very easy by using [filtering](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Collectors.html#filtering(java.util.function.Predicate,java.util.stream.Collector)) collector in [Collectors](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html) class; **filtering** was introduced in JDK 9, so make sure you use a version older than JDK 8 to be able to use it!

We could keep showing examples forever, as there are multiple combinations, but I’d suggest that you use autocomplete in [Collectors](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html) class to show all the available collectors every time you need to do something related to grouping collections.

Let’s take a look now at the third groupingBy option!

**Grouping elements specifying what Map to use**

The third option allows us to initialise the Map used to store our groups, most of the times we won’t need it because in most of the cases using a Java standard Map is perfectly valid.

Let’s see an example of how can we use it; in our example, although it will look as a quite silly and useless example, we are going to initialise the Map with an existing Map with a couple of elements:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | final Collection<Employee> employees = List.of( | |  | new Employee("Karen Smith", 51200.0, 29, Employee.Sex.FEMALE), | |  | new Employee("John Smith", 24000.0, 32, Employee.Sex.MALE), | |  | new Employee("Anthony Jackson", 44000.0, 33, Employee.Sex.MALE), | |  | new Employee("Alyson Palmer", 34320.0, 36, Employee.Sex.FEMALE), | |  | new Employee("Jessica Sanders", 64320.0, 34, Employee.Sex.FEMALE) | |  | ); | |  |  | |  | final Map<Employee.Sex, List<Employee>> existingMap = new HashMap<>() { | |  | { | |  | put(Employee.Sex.MALE, new ArrayList<>() {{add(new Employee("Peter Parker", 28100.0, 33, Employee.Sex.MALE));}}); | |  | put(Employee.Sex.FEMALE, new ArrayList<>() {{add(new Employee("Sarah Williams", 18100.0, 19, Employee.Sex.FEMALE));}}); | |  | } | |  | }; | |  |  | |  | final Map<Employee.Sex, List<Employee>> employeesOverThirtyBySex = employees.stream() | |  | .collect(groupingBy(Employee::getSex, () -> existingMap, filtering(employee -> employee.getAge() > 30, toList()))); | |  |  | |  | System.out.println(employeesOverThirtyBySex); | |

As you can see, we tell our collector to start with an existing map, which is provided by a supplier we specify in the “***mapFactory***” argument of our ***groupingBy*** method.

What’s interesting about our example is that after running the code you’ll see that our final Map contains employees below thirty! Why is that? That’s because our initial elements provided in the initial map don’t get processed by the **filtering** collector. This is something important to consider if you ever need to use ***mapFactory*** argument.

**Grouping elements concurrently**

After having had a look at the three main methods provided by groupingBy collectors, we know that there’s a concurrent version for each of them.

To execute these methods we’ll need a parallel stream and it will return a [ConcurrentHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html" \t "_blank). This method has the flags CONCURRENT and UNORDERED set, what means that multiple threads will accumulate elements into the same accumulator instead of combining multiple accumulators; in this case our single accumulator is a [ConcurrentHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html" \t "_blank).

So that’s all about ***groupingBy***! I hope you’ve enjoyed it!

If you still struggle with understanding Java Streams and functional programming in general, I’d recommend that you read “***Functional Programming in Java: Harnessing the Power Of Java 8 Lambda Expressions***”; you can buy it on Amazon in the following [link](https://geni.us/Bdoyif8).

**Conclusion**

Before JDK 8 was released, grouping elements was never even close to being as easy as it is nowadays; if you consider the amount of code and the lack of readability that we would achieve by writing any of the examples we’ve seen in this articles using plain Java 7 code, the difference will be substantial.

Also is much more enjoyable as a developer to write our solutions using a functional approach, just expressing the steps needed and not how to do it. This has been a huge step towards good readability and simplicity for our Java code.

I think we can all agree that now grouping elements in Java is very straightforward in most of the cases, except for some special complex cases that we could find, in general it’s very simple.