Stream Collectors

The data in a stream may be collected by using the method *collect()*. It has 2 signatures:

<R> R collect(Supplier <R> s, BiConsumer<R, ? super T> accumulator, BiConsumer<R, R> combiner)

Accumulates the data from a stream of T's in an R object
The supplier creates the R object(s) needed.
The first BiConsumer collects a T to an R
The second BiConsumer combines two R's (it is used in case the processing is split).

```
Stream<String> stream = Stream.of("w", "o", "l", "f");
StringBuilder word = stream.collect(StringBuilder::new,
        StringBuilder::append, StringBuilder::append);

Stream<String> stream = Stream.of("w", "o", "l", "f");
TreeSet<String> set = stream.collect(TreeSet::new, TreeSet::add,
        TreeSet::addAll);
System.out.println(set);                // [f, l, o, w]
```

Java provides the **Collector** interface, as a general way of collecting results (it encapsulates the generation of the Supplier and BiConsumers). It also provides the class **Collectors** with methods to create and combine Collectors for often needed strategies. When using a Collector, the signature is:

<R,A> R collect(Collector <? super T, A, R> c)

R is the final result type
A is the mutable accumulator
T is the type of the stream

Basic Collectors:

- **Collectors.joining(String s) /.joining()/.joining(String delim, String prefix, String suffix)**→ returns a Collector that merges all elements into a String, using the given String as delimiter between each (the stream has to be of CharSequence type). If used with 3 arguments, the prefix and suffix are added to the result string. It uses the toString() method of the Stream elements!
- **Collectors.counting()** → a Collector that returns a Long with the number of elements in the stream.
- **Collectors.toList()** → a Collector that adds all the elements to a List (arbitrary implementation)
- **Collectors.toSet()** → returns a Collector that adds all the elements to a Set (arbitrary implementation → don't rely on order)
- **Collectors.toCollection(Supplier<? extends Collection<T>> s)** → returns a Collector that adds all elements to a collection of the type generated by the Supplier.
- **Collectors.maxBy(Comparator<? super T> c) / minBy(Comparator<T> c)** → returns a Collector that returns the T that is the maximum/minimum of the T's in the stream.

Statistics Collectors

- **Collectors.averagingDouble(ToDoubleFunction<T> tdf) /**

.averagingInt(ToIntFunction<T> tif) / .averagingLong(ToLongFunction<T> tlf) → returns a collector that converts all T's to a double/int/long and calculates the average (when applied to a Stream returns **always a Double**)

- **Collectors.summingDouble(ToDoubleFunction<T> tdf) / .summingInt(ToIntFunction<T> tif) / .summingLong(ToLongFunction<T> tlf)** → returns a collector that converts all T's to a double/int/long and calculates the sum (when applied to a Stream returns a Double/Integer/Long)
- **Collectors.summarizingDouble(ToDoubleFunction<T> tdf) / .summarizingInt(ToIntFunction<T> tif) / .summarizingLong(ToLongFunction<T> tlf)** → returns a collector that converts all T's to a double/int/long and returns an object of type
    - DoubleSummaryStatistics
    - IntSummaryStatistics
    - LongSummaryStatistics

    These objects gather the numeric statistics for all the elements in the stream:
    - **.getMax()/.getMin()**
    - **.getCount()** →returns always a long
    - **.getSum()** → returns a long (int, long) or a double (double)
    - **.getAverage()** → returns a double

Collecting to Maps

- **Collectors.toMap(Function<? super T, ? extends K> keyf, Function<? super T, ? extends V> valf)**
    - returns a Map<K,V> with the data in the Stream. The first function generates the key for each element, and the second generates the value
    - if two elements give the same key → **exception**
    - The Map implementation is arbitrary (usually a HashMap)

- **Collectors.toMap(Function<? super T, ? extends K> keyf, Function<? super T, ? extends V> valf, BinaryOperator<T> merger)**
    - returns the same of the previous toMap, but when two same keys are generated, the values are merged used the BinaryOperator
    - The Map implementation is arbitrary (usually a HashMap)

- **Collectors.toMap(Function<? super T, ? extends K> keyf, Function<? super T, ? extends V> valf, BinaryOperator<T> merger, Supplier<M extends Map<K,V> s)**
    - returns the same of the previous toMap, but the map implementation is the one of the Supplier

Grouping
The collector generates a Map which holds, as Keys the result of applying a function, and as values, a reduction of the values that give that result. → if no entries give a given result, the key is not in the map!

- **Collectors.groupingBy(Function<? super T, ? extends K> classifier)**
    - returns a **Map<K,List<T>>** with all the T's in the stream classified by what the function returns
- **Collectors.groupingBy(Function<? super T, ? extends K> classifier, Collector<? super T, A, D> groupCollector)**
    - returns a Map<K,D> with the data all the T's in the stream classified by what the function returns, and each of the groups is reduced using the given collector.
- **Collectors. groupingBy (Function<? super T, ? extends K> classifier, Supplier<M extends Map> s, Collector<? super T, A, D> groupCollector)**

    - the same as the previous, but the map implementation is the one given by the supplier

Partitioning

Is a special case of grouping, where only two groups are created, one for true, and one for false. Unlinke in grouping, there are always the two keys as a result of partitioning

- **Collectors.partitioningBy(Predicate <? super T> p)** → returns a Map<Boolean, List<T>>, with elements divided by the result of the predicate
- **Collectors.partitioningBy(Predicate <? super T> p, Collector<? super T, A, D> groupCollector)** → returns a Map<Boolean, D>, with elements divided by the result of the predicate, and each group reduced with the given collector.

Mapping

The mapping operation adapts a collector that accepts elements of type U to a collector that accepts elements of type T, by giving a conversion function:

- **Collectors.mapping(Function<? super T, ? extends U> mapper, Collector<? super U, A, R> collector)**