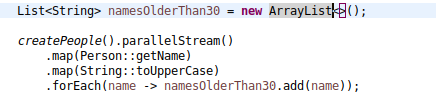
# Collectors

## Overview

In the approach of functional programming, mutability needs to be avoided to extreme. So the functions should be **pure functions.**

“Mutability is ok, shared mutability is purely evil”

Below code modifies a shared variable in parallel operation:



“Collector” in java is a reduce operation. Using which we can avoid shared mutability.

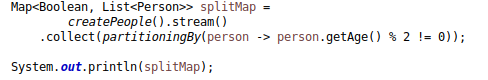
Some common operations are collecting the objects to List, Set, Map.

## Partitioning the data

Partitioning the data means splitting the data to two different group (to a map), based on a filter condition (Predicate)

{false=[Jill---28, Hari---44, Jit---52, Kalyan---20],

true=[Jack---25, Amy---35, Mary---39, Nancy---75]}

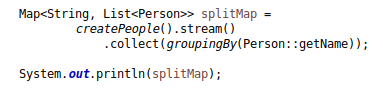


## Grouping to multiple different groups

Grouping List of people based on their name, if they have same name, they will be to a single bucket.

Map<String, List<Person>>

Use a “groupingBy” from Collectors. “groupingBy” is an overloaded function in Collectors.

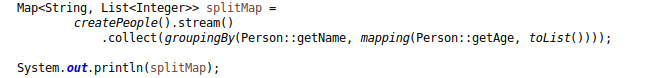


Grouping by person name and list of age:

Use overloaded method “groupingBy” which accepts a Function and a Collector

groupingBy(Function, Collector)

Map<String, List<Integer>>

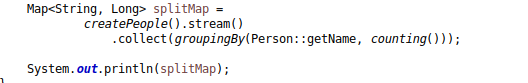


Here “groupingBy” structure is like below

groupingBy(Function, Collector(Function, Collector))

### Counting

“counting” returns a Long. It’s helpful when we want count in a group.

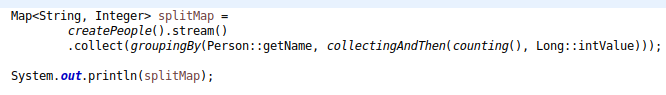


Counting returns Long to convert to a Integer

In “grouping” and “mapping” we have (Function, Collector)

Here what we need is (Collector, Function) – 1st Collector to run then a Function to count

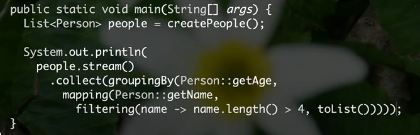
So we have a different method that is “collectingAndThen”



## Filtering

Like in “map” where we are performing transformation on Stream and in “mapping” transformation performed in middle of reduce.

Similarly in “filtering” transformation is performed in middle of reduce.



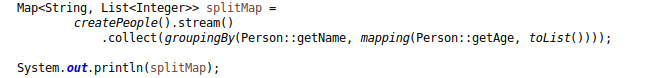
Introduced in JAVA11

# Some useful collectors methods

## Mapping

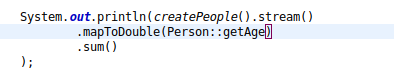
Mapping(Function, Collector) return a Collector which applies mapping function to input element and provides the mapped results to downstream collector.

Highly used in middle of reduce.



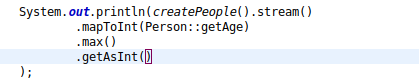
## Sum

“sum” is also a reduce operation which operates on primitive stream like “IntStream”, “DoubleStream” etc..



## Max

“max” is a reduce operation which gives Optional, because collection can be empty



## maxBy

If we are interested not in getting the max value, but the object itself ex. Person object. We need to use maxBy and pass a Comparator to it.

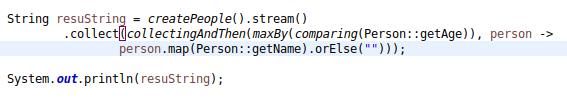


Here collect(maxBy(… returns an optional. To operate on optional we need to use **“map”. Note this map is on optional not on stream.**

If we don’t want the entire person, but just a property ex. Name of the person

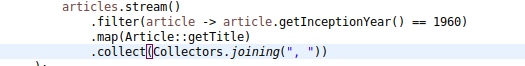
Again here we have chance to use “collectingAndThen(Collector, Function)” where it first takes the collector and then a function for transformation.

As “maxBy” returns an Optional<Person> to get the value we need to call “map” on optional. “orElse” will be executed when collection is empty and no result came out.



## joining

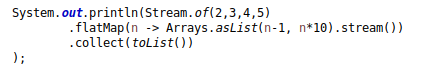
Join a stream with provided delimiter.

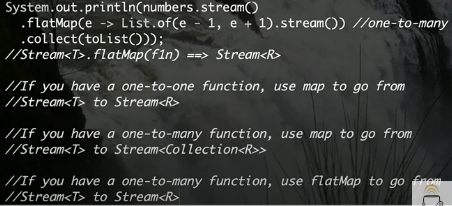


# Important Functions

## flatMap

Use flat map when we have one-to-many function and want to go from Stream<T> to Stream<R>





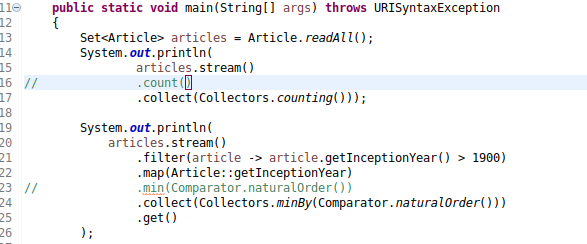
# Devoxx 2017 Paul

## Some equivalent collectors methods

Whatever we can do using stream API equivalently we can do the same using .collect passing Collectors methods.

Like for .count() equivalently we can use Collectors.counting()

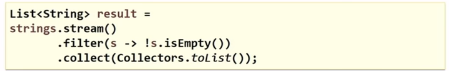
For .min(Comparator) we can use Collectors.minBy(Comparator)



## How a Collector is working

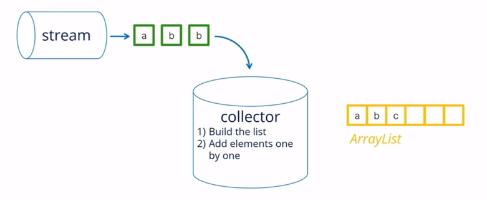
Collector is working on 3 basic operations. Supply, Accumulate and Combine.

When creating a List as below

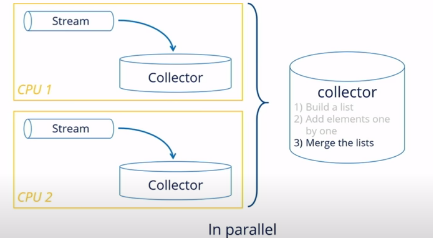


Below operations are happening:

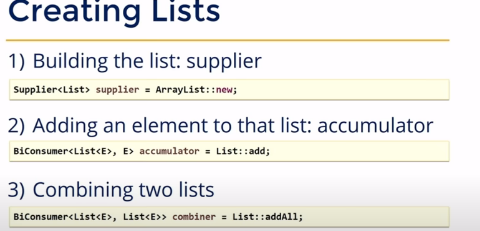
Creating a List and adding elements into List



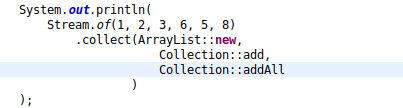
To support parallelization, in below 2 partially filled list will created on two cores in my cpu and will be merged using same object Collector.



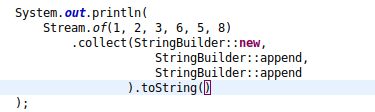
Below code is happening when creating a List



So instead of Collectors.toList(). Below code can be written to create a List

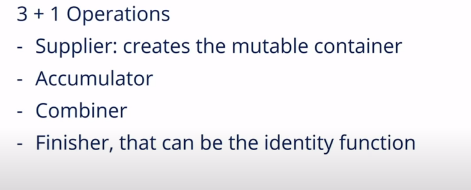


To join elements in a stream, same pattern can be followed as:



Here in the above case .toString() is the **Finisher.**

So Collector is indeed 3 + 1 operation:



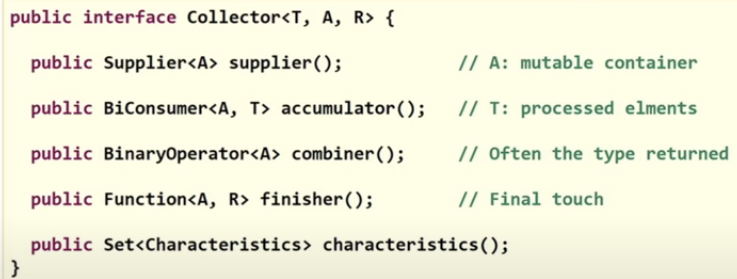
In many case finisher is an Identity function, but not in all cases like above.

### CollectingAndThen

We can pass a custom **Finisher** and that can done using “collectingAndThen(Collector, Function)

Here the Function is a custom finisher.

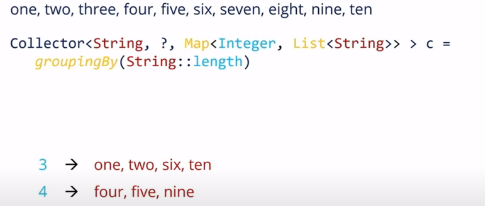
## Collector Interface



Type of a Collector:

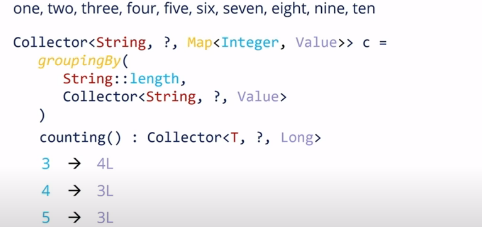
* **T :** Type of the elements of the stream
* **A:** Type of Mutable container (For the intermediate operation)
* **R:** Type of final container

When Finisher is the identity function we will have A = R



Here Collector<**String** – Input type, **?** – Don’t know the container for intermediate mutable operation, **Map** – Is the type of Return container)

One more example, where we are passing a collector “counting()” inside “groupingBy()”



## Intermediate operations