Java “Pipelines”

The use of Java 8 streams and “pipelines” allows us, in a certain sense, to simply declare *what* we want to happen and not have to worry about *how* to make it happen. This oversimplifies things, of course, but streams allow us to manipulate data in a “declarative” way that permits us to chain together or “compose” a sequence of operations to express a complicated data-processing “pipeline”. All this can help to produce more readable code and with less effort than was possible before. In summary, Java 8 streams

- Are declarative, which allow more concise and readable code
- Are composable, which allows for more flexible code
- Are “parallelizable” (with no extra effort by the programmer), which can provide improved performance

These are the components of a Java “pipeline”:

- A data “source” provides a “stream” of data values “one at a time”. [It may be useful to think of a Java “container” as as “collection of values in space” and a “stream” as a “collection of values in time”.]
- An “intermediate operation” is one that performs some action on some or all of the values in a stream and returns another stream of (the modified) values. A pipeline may have zero or more intermeiate operations. If there are several intermediate operations, these are the ones “chained together” or “composed” to achieve the overall effect. Any intermediate operation can be “stateful” (needs to keep track of previous results to do its work, such as distinct()) or “stateless” (does not need to do this, such as filter()).
- A “terminal operation” produces a “final result” of some kind that involves the values of a stream and terminates the pipeline.

A “pipeline” may thus be conceptualized to look like this:

source ➔ intermediate operation ➔ intermediate operation ➔ … ➔ terminal operation

Some sources	Some intermediate operations	Some terminal operations

-From a container with stream()
-From a “primitive specialized stream”
-From values with Stream.of()
-From an array
-From a file
-From a function with
--iterate()
--generate()
	filter()
	sorted()
	distinct()
	limit()
	skip()
	map(), flatmap()
	mapToInt(), mapToLong(), mapToDouble()
		forEach()
		count()
		collect()
		reduce()
		findFirst()
		findAny()
		anyMatch()
		noneMatch()
		allMatch()
		max()
		min()
		toArray()

Often what you want to happen at the “end of a pipeline” may be characterized as one of the following:

- Reducing or summarizing stream elements to a single value
- Grouping elements in some way
- Partitioning elements in some way

Java has a `java.util.stream.Collectors` class containing a lot of `static` methods that implement the *`java.util.stream.Collector`* interface in ways that perform different variations on the above tasks. Here are some of them:

- toList()
- toSet()
- toCollection()
- counting()
- summingInt(), summingLong() and summingDouble()
- averagingInt(), averagingLon()g and averagingDouble()
- summarizingInt(), summarizingLong() and summarizingDouble()
- joining()
- maxBy()
- minBy()
- reducing() (can be used as a generalization of other collectors)
- collectingAndThen()
- groupingBy()
- partitioningBy()