**Java Stream**

1. Refer <https://netjs.blogspot.com/2016/11/stream-api-in-java-8.html>
2. Refer <https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/>
3. A stream can be visualized as a pipeline. A stream pipeline consists of a **source** (which might be an array, a collection, a generator function, an I/O channel, etc), zero or more **intermediate operations** (which transform a stream into another stream, such as filter(Predicate)), and a **terminal operation**
4. List of some intermediate operations in stream

* filter()
* anyMatch()
* findFirst()
* map()
* flatMap()
* distinct()
* sorted()
* peek()
* limit()
* skip()

1. List of some terminal operations

* forEach(consumer) : applies same function to each element.
* collect() : saves the elements onto a collection
* count(), min(), max(), reduce(), summaryStatistics() : reduces the stream into a single summary element.

1. Refere this video link

<https://www.youtube.com/watch?v=t1-YZ6bF-g0>

1. What is ava.lang.IllegalStateException: stream has already been operated upon or closed ?

When we do any terminal operation on stream then compiler close the stream for further operation. That time if we again want to do some more operation it will throw this exception

Solution : Use supplier to generate a new new stream everytime by implementing get() method

1. What is the use of accumulator ?

<https://www.youtube.com/watch?v=_n7GOopfvWA&t=468s>

1. Difference between reduce() and collect method in java 8 <https://www.youtube.com/watch?v=oWlWEKNM5Aw>
2. **Why use combiner in reduce function for parallel stream ?**

One of the design principles of the Streams API is that the API shouldn't differ between sequential and parallel streams, or put another way, a particular API shouldn't prevent a stream from running correctly either sequentially or in parallel. If your lambdas have the right properties (associative, non-interfering, etc.) a stream run sequentially or in parallel should give the same results.

Let's first consider the two-arg version of reduction:

T reduce(I, (T, T) -> T)

The sequential implementation is straightforward. The identity value I is "accumulated" with the zeroth stream element to give a result. This result is accumulated with the first stream element to give another result, which in turn is accumulated with the second stream element, and so forth. After the last element is accumulated, the final result is returned.

The parallel implementation starts off by splitting the stream into segments. Each segment is processed by its own thread in the sequential fashion I described above. Now, if we have N threads, we have N intermediate results. These need to be reduced down to one result. Since each intermediate result is of type T, and we have several, we can use the same accumulator function to reduce those N intermediate results down to a single result.

Now let's consider a hypothetical two-arg reduction operation that reduces Stream<T> to U. In other languages this is called a "fold" or "fold-left" operation so that's what I'll call it here. Note this doesn't exist in Java.

U foldLeft(I, (U, T) -> U)

(Note that the identity value I is of type U.)

The sequential version of foldLeft is just like the sequential version of reduce except that the intermediate values are of type U instead of type T. But it's otherwise the same. (A hypothetical foldRight operation would be similar except that the operations would be performed right-to-left instead of left-to-right.)

Now consider the parallel version of foldLeft. Let's start off by splitting the stream into segments. We can then have each of the N threads reduce the T values in its segment into N intermediate values of type U. Now what? How do we get from N values of type U down to a single result of type U?

What's missing is another function that combines the multiple intermediate results of type U into a single result of type U. If we have a function that combines two U values into one, that's sufficient to reduce any number of values down to one -- just like the original reduction above. Thus, the reduction operation that gives a result of a different type needs two functions:

U reduce(I, (U, T) -> U, (U, U) -> U)

Or, using Java syntax:

<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)

In summary, to do parallel reduction to a different result type, we need two functions: one that accumulates T elements to intermediate U values, and a second that combines the intermediate U values into a single U result. If we aren't switching types, it turns out that the accumulator function is the same as the combiner function. That's why reduction to the same type has only the accumulator function and reduction to a different type requires separate accumulator and combiner functions.

Finally, Java doesn't provide foldLeft and foldRight operations because they imply a particular ordering of operations that is inherently sequential. This clashes with the design principle stated above of providing APIs that support sequential and parallel operation equally.