java.util.stream

From Functional programing a **monad** is a structure that represents computations defined as sequences of steps. A type with a monad structure defines what it means to chain operations, or nest functions of that type together. Stream represent monad in java, It is all about processing order

Classes to support functional-style operations on streams of elements, such as map-reduce transformations on collections.

Streams differ from collections

- No storage. A stream is not a data structure that stores elements; instead, it conveys elements from a source such as a data structure, an array, a generator function, or an I/O channel, through a pipeline of computational operations.
- An operation on a stream produces a result, but does not modify its source
- operations: intermediate (laizy) and terminal

Stream can be obtained from

- Collection, Arrays
- static factory methods Stream.of(), InitStream.range(), Stream.iterate
- lines from file BufferdReader.lines()
- files fron Files class
- random numbers Random.ints()
- StreamSupport for low.level creating stream (use Spliiterator)

Stream pipline have sorice of the stream zero or more intermediate operations and a terminal operation

Intermediate operations return a new stream. They are always *lazy*; executing an intermediate operation such as filter() does not actually perform any filtering, but instead creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate. Traversal of the pipeline source does not begin until the terminal operation of the pipeline is executed.

Intermediate operations are further divided into ***stateless*** and ***stateful*** operations. **Stateless** operations, such as filter and map, retain no state from previously seen element when processing a new elemen. **Stateful** operations, such as distinct and sorted, may incorporate state from previously seen elements when processing new elements

Terminal operations, such as Stream.forEach or IntStream.sum, may traverse the stream to produce a result or a side-effect. After the terminal operation is performed, the stream pipeline is considered consumed, and can no longer be used

Parallelism

Except for operations identified as explicitly nondeterministic, such as findAny(), whether a stream executes sequentially or in parallel should not change the result of the computation.

To preserve correct behavior, lambda expresions must be *non-interfering*, and in most cases must be *stateless.* A stateful lambda is one whose result depends on any state which might change during the execution of the stream pipeline. An example of a stateful lambda is the parameter to map() in:

```
Set<Integer> seen = Collections.synchronizedSet(new HashSet<>());
stream.parallel().map(e -> { if (seen.add(e)) return 0; else return e; })
```

Ensure that the data source is *not modified at all* during the execution of the stream pipeline. Example (neve do this)

```
List<String> l = new ArrayList(Arrays.asList("one", "two"));

Stream<String> sl = l.stream();
```

```
l.add("three");
String s = sl.collect(joining(" "));
```

Side-effects in behavioral parameters to stream operations are, in general, discouraged. Example:

```
ArrayList<String> results = new ArrayList<>();
stream.filter(s -> pattern.matcher(s).matches())
      .forEach(s -> results.add(s));  // Unnecessary use of side-effects!
```

This code unnecessarily uses side-effects. If executed in parallel, the non-thread-safety of ArrayList would cause incorrect results

For sequential streams, the presence or absence of an encounter order does not affect performance, only determinism. For parallel streams, relaxing the ordering constraint can sometimes enable more efficient execution

Operation or function must be associative: (a op b) op c == a op (b op c)

So we can evaluate (a op b) in parallel with (c op d), and then invoke op on the results.
Examples of associative operations include numeric addition, min, and max, and string concatenation.

Reduction operations

A *reduction* operation (also called a *fold*) takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation, such as finding the sum or maximum of a set of numbers, or accumulating elements into a list. You can use reduce() and colect()

a reduce operation on elements of type <T> yielding a result of type <U> requires three parameters:

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

Example:

```
OptionalInt heaviest = widgets.parallelStream()
                       .mapToInt(Widget::getWeight)
                       .max();
```

the ***identity*** element is both an initial seed value for the reduction and a default result if there are no input elements.
The ***accumulator*** function takes a partial result and the next element, and produces a new partial result.
The ***combiner*** function combines two partial results to produce a new partial result

A *mutable reduction operation* accumulates input elements into a mutable result container, such as a Collection or StringBuilder, as it processes the elements in the stream It is **collect**(), (collects together the desired results into a result container such as a Collection)

```
<R> R collect(Supplier<R> supplier,
         BiConsumer<R, ? super T> accumulator,
         BiConsumer<R, R> combiner);
```

A supplier function to construct new instances of the result container,

Example:

```
ArrayList<String> strings = stream.collect(() -> new ArrayList<>(),
                                (c, e) -> c.add(e.toString()),
                                (c1, c2) -> c1.addAll(c2));
```

The class **Collectors** contains a number of predefined factories for collectors (Collectors.summingInt,

Collectors.groupingBy)

Note in stream you pass lambda ex, in oder to work for paralele stream lambda must not be

A function is non-interfering when it does not modify the underlying data source of the stream, e.g. in the above example no lambda expression does modify myList by adding or removing elements from the collection.

A function is stateless when the execution of the operation is deterministic, e.g. in the above example no lambda expression depends on any mutable variables or states from the outer scope which might change during execution.

From what we discussed so far, *Stream* is a stream of object references. However, there are also the *IntStream*, *LongStream*, and *DoubleStream* – which are primitive specializations for *int*, *long* and *double* respectively.

Some Streams classes

AbstractPipeline

*** Abstract base class for "pipeline" classes, which are the core**
* implementations of the Stream interface and its primitive specializations.
- Manages construction and evaluation of stream pipelines.

interface BaseStream<T, S extends BaseStream<T, S>>
extends AutoCloseable

*** Base interface for streams, which are sequences of elements supporting**
- sequential and parallel aggregate operations.

interface TerminalOp<E_IN, R> {
*** An operation in a stream pipeline that takes a stream as input and produces**
- a result or side-effect.

StreamSupport
*** Low-level utility methods for creating and manipulating streams.**
*

interface Spliterator<T>
*** An object for traversing and partitioning elements of a source. The source**
* of elements covered by a Spliterator could be, for example, an array, a
- {@link Collection}, an IO channel, or a generator function

Create you own stream from custom source

So there are two things you need to keep in mind while creating your own streams from custom sources

1. What is your data source and what is the implementation of your Spliterator to access the data from that source?

2. How data processing is going to happen (Using ReferencedPipeline)?

Example: Stream<String> myStream = StreamSupport.stream(mySpliterator, false);

ReferencedPiplines. This is a class having a lot of complicated implementations like map, filter, flatmap reduce etc. So you are only responsible for implementing your own custom spliterator

form more info see: https://basicsstrong.com/creating-your-own-streams-using-custom-spliterator-and-how-streams-works-internally-in-java/

Libary: https://streamplify.beryx.org/releases/latest/
- provide useful Java 8 streams and to assist you in building new streams
- combinatorics streams: permutations, combinations, Cartesian products, powers sets, derangements, partial permutations.
- classes that help you implement your own efficient parallel streams.

Operations On Streams

boolean	**allMatch**(Predicate<? super T> predicate) Returns whether all elements of this stream match the provided predicate.
boolean	**anyMatch**(Predicate<? super T> predicate) Returns whether any elements of this stream match the provided predicate.
<R,A> R	**collect**(Collector<? super T,A,R> collector) Performs a mutable reduction operation on the elements of this stream using a Collector.

<R> R	**collect**(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner) Performs a mutable reduction operation on the elements of this stream.
long	**count**() Returns the count of elements in this stream.
Stream<T>	**distinct**() Returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream.
Stream<T>	**filter**(Predicate<? super T> predicate) Returns a stream consisting of the elements of this stream that match the given predicate.
Optional<T>	**findAny**() Returns an Optional describing some element of the stream, or an empty Optional if the stream is empty.
Optional<T>	**findFirst**() Returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty.
<R> Stream<R>	**flatMap**(Function<? super T,? extends Stream<? extends R>> mapper) Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
DoubleStream	**flatMapToDouble**(Function<? super T,? extends DoubleStream> mapper) Returns an DoubleStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
IntStream	**flatMapToInt**(Function<? super T,? extends IntStream> mapper) Returns an IntStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
LongStream	**flatMapToLong**(Function<? super T,? extends LongStream> mapper) Returns an LongStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
void	**forEach**(Consumer<? super T> action) Performs an action for each element of this stream.
void	**forEachOrdered**(Consumer<? super T> action) Performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order.
Stream<T>	**limit**(long maxSize) Returns a stream consisting of the elements of this stream, truncated to be no longer than maxSize in length.

<R> Stream<R>	**map**(Function<? super T,? extends R> mapper) Returns a stream consisting of the results of applying the given function to the elements of this stream.
DoubleStream	**mapToDouble**(ToDoubleFunction<? super T> mapper) Returns a DoubleStream consisting of the results of applying the given function to the elements of this stream.
IntStream	**mapToInt**(ToIntFunction<? super T> mapper) Returns an IntStream consisting of the results of applying the given function to the elements of this stream.
LongStream	**mapToLong**(ToLongFunction<? super T> mapper) Returns a LongStream consisting of the results of applying the given function to the elements of this stream.
Optional<T>	**max**(Comparator<? super T> comparator) Returns the maximum element of this stream according to the provided Comparator.
Optional<T>	**min**(Comparator<? super T> comparator) Returns the minimum element of this stream according to the provided Comparator.
boolean	**noneMatch**(Predicate<? super T> predicate) Returns whether no elements of this stream match the provided predicate.
Stream<T>	**peek**(Consumer<? super T> action) Returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.
Optional<T>	**reduce**(BinaryOperator<T> accumulator) Performs a reduction on the elements of this stream, using an associative accumulation function, and returns an Optional describing the reduced value, if any.
T	**reduce**(T identity, BinaryOperator<T> accumulator) Performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.
<U> U	**reduce**(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner) Performs a reduction on the elements of this stream, using the provided identity, accumulation and combining functions.
Stream<T>	**skip**(long n) Returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream.
Stream<T>	**sorted**() Returns a stream consisting of the elements of this stream, sorted according to natural order.
Stream<T>	**sorted**(Comparator<? super T> comparator) Returns a stream consisting of the elements of this stream, sorted according to the

	provided Comparator.
Object[]	**toArray**() Returns an array containing the elements of this stream.
<A> A[]	**toArray**(IntFunction<A[]> generator) Returns an array containing the elements of this stream, using the provided generator function to allocate the returned array, as well as any additional arrays that might be required for a partitioned execution or for resizing.

Converting a stream to the *Collection* (*Collection*, *List* or *Set*):

```
List<String> collectorCollection =
        productList.stream().map(Product::getName).collect(Collectors.toList());
```

Grouping of stream's elements according to the specified function:

```
Map<Integer, List<Product>> collectorMapOfLists = productList.stream()
        .collect(Collectors.groupingBy(Product::getPrice));
```

Dividing stream's elements into groups according to some predicate:

```
Map<Boolean, List<Product>> mapPartioned = productList.stream()
        .collect(Collectors.partitioningBy(element -> element.getPrice() > 15));
```

Stream to LinkedList

```
List<String> tokenlist = tokenStream.collect(Collectors.toCollection(LinkedList::new));
```

Merge two streams

```
Stream<Integer> resultingStream = Stream.concat(firstStream, secondStream);
```

Java stream sort on multiple fields – example

Create custom comparator

Create a Map from stream

toMap takes two arguments: the first is the function to get keys (we pass identity there -- it is a function that returns exactly its argument, like `x -> x`, and the second one to get values.

```
Map<String, Integer> map = words.stream().collect(Collectors.toMap(Function.identity(),
String::length));
```

Flat map

Flat map is similar to map: it iterates over every item in the stream and calls the function which you pass as the argument. But your function should return not a single value but a stream. And flat map then concatenates all the streams returned.

We can use **map**() operation when we have a stream of objects, and we need to get some unique value for each element in the stream. There is **one-to-one** mapping between input and output element. For example, we can write a program to find the **date of birth of all employees** in a stream of employees.

In case of **flatMap**(), a **one-to-many** mapping is created where for each input element/stream, we first get a multiple values and then we flatten the values from all such input streams into a single output stream. For example, we may write program to find **all district words from all lines in a text file**.

How are stream implemented, Collection Pipeline pattern: https://martinfowler.com/articles/collection-pipeline/

TODO custom SPLITERATOR FOR HTML, XML, CSV FILES like libary https://streamplify.beryx.org/releases/latest/

From:

https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html
https://www.geeksforgeeks.org/stream-in-java/

https://stackify.com/streams-guide-java-8/

https://www.baeldung.com/java-8-streams

https://howtodoinjava.com/java8/java-streams-by-examples/

http://tutorials.jenkov.com/java-functional-programming/streams.html Stao

https://beginnersbook.com/2017/10/java-8-stream-tutorial/

https://mydeveloperplanet.com/2020/09/23/java-streams-by-example/

https://www.codingame.com/playgrounds/31592/java-8-stream-tutorial

https://medium.com/@sergeykuptsov/how-it-works-in-java-streams-735ad4393c41

http://zetcode.com/lang/java/streams/

https://pdf.co/blog/java-streams

https://reflectoring.io/processing-files-using-java-8-streams/

https://jaxenter.com/java-performance-tutorial-how-fast-are-the-java-8-streams-118830.html

https://github.com/streamsupport/streamsupport

https://opensource.com/article/20/5/functional-java

https://java2blog.com/java-8-stream/

https://www.jrebel.com/blog/java-streams-vjug-venkat-subramaniam

https://guava.dev/releases/23.0/api/docs/com/google/common/collect/Streams.html

https://stackoverflow.com/questions/32166193/understanding-java-8-streams-at-the-bytecode-level

https://www.studytonight.com/java-8/java-8-stream-api

https://www.educative.io/blog/master-stream-api-and-beyond

https://dzone.com/articles/your-guide-to-java-streams-tutorials-and-articles

https://www.javainuse.com/java/java8_streams

https://theboreddev.com/understanding-java-streams/

https://www.java67.com/2014/04/java-8-stream-examples-and-tutorial.html