Streams

Sequences of data, they are processed through a *stream pipeline*.
It has 3 types of operations:

1. Source or stream generation → the stream is created
2. Intermediate operations → the stream is modified in some way, and a new stream is returned
3. Terminal operation → the result of all processing is obtained (and no stream is returned)

There is only **one source operation**, as **many intermediate operations as needed (or none)** and only **one terminal operation**
Streams **can not be reused**: when a terminal operation is called, the stream is not usable anymore (exception).
Streams use **lazy evaluation**: the source and intermediate operations do not run until the terminal operation starts, and only as long as the terminal operation needs them to be run.
Streams may be ordered or unordered (depending on the source/generation operation used, or on the intermediate operations).
Streams obtanined from collections or arrays **do not modify the source**, but the source may be modified unitl the terminal operation is called.

Source/generation operations

- **Stream <T> s = Stream.empty();** → creates an empty stream
- " = **Stream.of(T args...);** → creates a stream with all the varargs
- " = **collection.stream();** → creates a stream from a collection's elements
- " = **Arrays.stream(T [] array);** → creates a stream from an array's elements
- " = **Stream.generate(Supplier<T> s);** → creates an infinite stream using the supplier given, which is run any times as needed
- " = **Stream.iterate(T t1, UnaryOperator<T> uo):** → creates an inifinite stream starting with t1, and getting the next T using the UnaryOperator on the previous T

Intermediate operations

- **filter(Predicate <T> p)** → returns a new Stream with only the elements of the Stream that return true for the given predicate
- **distinct()** → returns a new Stream where duplicates from the Stream are removed (using equals!)
- **limit(int maxSize)** → returns a new Stream which size is reduced to maxSize if greater (or infinite) by discarding the latter elements. If the size is not greater, the Stream is .
- **skip(int skipped)** → returns a new Stream which is missing the first *skipped* elements
- **sorted()/sorted(Comparator<? super T> c)** → returns a new Stream with its elements sorted. If no comparator is provided, and the elements are not Comparable, ClassCastException.
- **peek(Consumer <? super T> c)** → returns a new Stream that has the same elements, but applies the consumer to each of the elements. **The stream is not changed, the elements may be changed, depending on what does the consumer.**
- **map(Function <? super T, ? extends R> f)** → returns a new Stream of another type (R), by converting each T of the Stream to an R using the given function.
- **flatMap(Function <? super T, ? extends Stream<? extends R>> f)** → returns a new Stream of another type R which combines all the streams generated from each T using the function provided

Intermediate operations may be stateful(result depends on other processed stream elements) or stateless (result is independent of previuosly processed elements)

→ stateful operations may need to process all stream elements to produce a result:
distinct(), sorted()...

Terminal operations
A terminal operation is needed to run any intermediates.
Some operations are **short-circuit**: at a given point they prevent from processing more elements → they may be used on infinite streams without hanging
Reductions are a special type of terminal operation where all of the contents of the stream are combined into a single primitive or Object

- **long count()** → returns the number of elements in the stream. For infinite streams, it hangs. It is a **reduction**
- **Optional<T> min(Comparator <? super T> c)** → Returns the minimum element in the stream using the given comparator. For infinite streams, it hangs. It is a **reduction**
- **Optional<T> max(Comparator <? super T> c)** → Returns the maximum element in the stream using the given comparator. For infinite streams, it hangs. It is a **reduction**
- **Optional<T> findAny()** → returns an element of the stream. Is a **short-circuit** operation.
- **Optional<T> findFirst()** → returns the first element of the stream. Is a **short-circuit** operation.
- **boolean anyMatch(Predicate <? super T> p)** → returns true if there is at least one element that matches the predicate on the Stream. Is **short-circuit only if there is such element**.
- **boolean allMatch(Predicate <? super T> p)** → returns true if all elements match the predicate on the Stream. Is **short-circuit only if there is one or more elements not matching**.
- **boolean noneMatch(Predicate <? super T> p)** → returns true if there is no element that matches the predicate on the Stream. Is **short-circuit only if there is one or more element matching**.
- **void forEach(Consumer <? super T> c)** → runs the given consumer for each element in the stream.
- **T reduce (T identity, BinaryOperator<T> accumulator)** → It is a **reduction** operation, which combines all elements in the Stream into a single T, starting with the identity given and by calling the accumulator with the previous accumulated value, and the current element.
- **Optional<T> reduce (BinaryOperator<T> accumluator)** → It is a **reduction** operation, combining all the elements using the given accumulator, and returning an optional (empty if no elements)
- **<U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)** → Combines all T in the stream into some U's, and then combines all those U's into a single one. It is a **reduction**, that makes sense mostly for parallel processing.
- **Collect** →It is a **reduction** operation (a mutable reduction), may get complex. **In next sessions**