16 - REVIEW Java Streams

Java Streams

Java Streams (`java.util.streams`) were introduced with the eighth version of Java and, with [Java Lambdas], represent functional programming aspects of Java.
The flexibility to use streams on all types of data is a by-product of [Java Generics].

A stream is a sequence of elements supporting sequential and parallel aggregate operations

Stream differ from Collection

- **no storage**: *a stream is not a data structure* that stores elements; instead it conveys elements from a *source* (which can be anything that produce a sequence of values: a data structure, an array, an I/O channel, ...) through a Pipeline of computational operations
- **functional by nature**: an operation on a stream produce a result byte does not modify the source (different from the imperative paradigm where the computation proceeds through the modification of the input and side effects)
- **laziness-seeking**: many stream operations are implemented lazily, exposing opportunities for optimization. Stream Operations are divided into *intermediate* (stream producing) and *terminal* (value- or side-effect-producing).
- **possibly unbounded**: collections have a finite size, streams not necessarily. Short circuit operations such as `limit(n)` or f `findFirst()` can allow computations on Infinite Streams: as Haskell infinite list comprehension and such
- **consumable**: the elements of a stream are only visited once during the life of a stream. Like a Java Iterators, a new stream has to be generated to revisit the same elements of the source

Types of Streams

Streams support both **reference** types and **primitive** types, unlike Java Collection.

Java Collections can contain only reference types (only pointer to objects), hence they can't contains primitive types: if we write `List<int>` the compiler translate that to `List<Integer>`, which is not as efficient.

Pipeline

A Pipeline contains:

- **source**: producing (by need, aka lazily) the elements of the stream
- **intermediate operations**: (if any) operations that take the stream and produce another stream
- **terminal operation**: producing side-effects or non-stream values. The terminal operation is unique and it closes the stream

Stream Sources

Streams can be obtained from:

- a **Collection**, by the methods `.stream()` or `.parallelStream()`
- an **array**, by `Arrays.stream(Object[])`
- **static factory methods** on the stream classes, such as
  - `Stream.of(Object[])`: obtain a stream from an array
  - `IntStream.range(int, int)`: obtain a int stream contained in an interval

- `Stream.iterate(Object, UnaryOperator)`: obtain a stream by applying more times an unary operator to an object
- **the lines of a file**, with the method `BufferedReader.lines()`
- **file paths** using the static methods in `Files`
- **generators** like `iterate` or `generate`
- ...

Intermediate Operations

Intermediate operations are performed on the stream elements. These operations **takes a stream** and **produce** another **stream** and they are **lazy**: they are not processed until the *terminal operation* is invoked.

It is possible to operate on streams elements without having to evaluate the rest of the streams: this is the lazy aspect of streams that allow the computation of [Infinite Streams]

Mind that a stream pipeline may contain some *short-circuit methods* (either intermediate or terminal) that cause the earlier intermediate methods to be processed only until the short-circuit method can be evaluated.

Several intermediate operations have arguments of *functional interfaces*, thus [Java Lambdas] can be used.

Some intermediate operations are:

- **filter:** `Stream<T> filter(Predicate<? super T> predicate)`: filter the elements of the stream
- **map:** `<R> Stream<R> map(Function<? super T, ? extends R> mapper)`
  - map function: `map f: T -> R` where `T` is defined in the class that contains the method `map`
  - `map` takes a function of type `? super T` -> `? extends R` and it apply the function to evert element of the stream. In fact `map` takes a stream of type `Stream<T>` and return a stream of type `Stream<R>`
- **distinct:** `Stream<T> distinct()`: remove the duplicate elements from the stream, it produce a duplicate-free stream (stateful operation since to remove the duplicates it has to store the already seen elements of the stream)
- **peek:** `Stream<T> peek(Consumer<? super T> action)`
  - this method perform `action` on the stream elements without modifying the original stream
  - typically used for debugging: the action could be `e -> System.out.println(e)`

Terminal Operations

A terminal operation must be the final operation on a stream, once a terminal operation is invoked the stream is consumed and no longer usable.
The typical effect is to collect values in a data structure, reduce the stream to a value, print the values or other side effects.

Some terminal operations are:

- **forEach**: `forEach(Consumer<? super T> action)`
  - this method is defined in `Stream`, it is not the `forEach` of `Collection`
  - it applies the action `accept` contained in `action` to every element of the stream
- **reduce**: analogous to the fold operations in Haskell
  - `T reduce (T identity, BinaryOperator<T> accumulator`
    - `T identity` is the base case of the fold, it is returned if the stream is empty
  - `Optional<T> reduce(BinaryOperator<T> accumulator`
    - return `Optional<T>` that covers the possibility of an empty stream
- **allMatch**: `boolean allMatch(Predicate<? super T> predicate>`
  - return true if all elements of the stream respect the predicate

  - this is a short-circuiting method
- ...

Example

```java
double average = pList // collection of Person
        .stream() // stream wrapper over a collection
        .filter(p -> p.getGender() == Person.Sex.MALE) //filter the stream
        .mapToInt(Person::getAge) // extracts stream of ages
        .average() // compute average (reduce operation)
        .getAsDouble() // extract result from OptionalDouble
```

where:

- `.stream()` open the stream from a collection (a list): the result of the method is a *source*
- all the operations besides `.average()` are intermediate operations that generate streams
- `.average()` is a terminal operation that extract a single value `Optioanl` and it is implemented as a fold and it use average as accumulator
- `.getAsDouble()` extract the average if it exists from the result of `.average()` which is `Optional`. If there were at least a person in `pList` then the result will be a `double`, otherwise it will be `Nothing`

Mutable Reductions

The reduce() methods discussed above is an example of immutable reduction, as it reduce the result into a single valued immutable variable.

Mutable reductions collect the desired results into a mutable container object such as a `java.util.Collection` or an array. Mutable reduction in Java stream API are implemented as collect() methods.

Suppose we want to concatenate a stream of strings, which can be done with

```java
String concatatenated = sList.stream().reduce("", String::concat);
```

Starting from the base case `""` we concatenate every string that arrives from the stream with the result of the previous concatenations.
This works but it is highly inefficient: with reduce we create a new string with every concatenation because `String` is immutable in Java: the chars of the previous concatenations have to be copied in the newly allocated string.

It is way better to "accumulate" the elements in a mutable object (a `StringBuilder` , a `Collection`, ...).
The *mutable reduction* operation is called `collect()` and it requires three functions:

- a supplier function to construct new instances of the result container
- an accumulator function to incorporate an input element into the result container
- a combining function to merge the contents of one result container into another

```java
<R> R collect(
        Supplier<R> supplier,
        BiConsumer<R, ? extends T> accumulator,
        Biconsumer<R, R> combiner
)
```

where:

- `collect` is defined in the interface `Stream<T>`
- `R` is a generic type declared in the method, `T` is from the interface

strings to a list:

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

where:

- `supplier` is a function with no pars that returns a new arraylist in every call
- `accumulator` adds an element of the stream (a string) to the newly created arraylist
- `combiner` merges the two arraylists (the new one with just one string to the old one with the old concatenated strings)

Collectors

The method `collect` can also be invoked with a `Collector` (which is an interface) argument

```java
<R, A> R collect(Collector<> super T, A, R> collector)
```

A collector encapsulates the functions used as arguments of collect (supplier, accumulator, combiner) allowing for reuse of collection strategies and composition of collect operations.
There are a lot of already written collectors in the API.

strings to a list:

```java
List<String> sList = sStream.collect(Collectors.toList());
```

Infinite Streams

Collections are stored eagerly, in Java there can't be an infinite collection. Hence a stream from a collection can be infinite.

Infinite streams can be generated with:

- **iterate**: `static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)`
  - `seed` is the starting element of the stream
  - `f` is a function $f: T \to T$;
  - the semantic is that to the first iteration we return `seed`, at the second iteration $f(seed)$, at the third iteration $f(f(seed))$, ...
- **generate**: `static <T> Stream<T> generate(Supplier<T> s)`
  - if `s` is a deterministic function than we have an infinite stream of the same element
  - if `s` is a non deterministic function (maybe `Math.random()`) than we have an infinite stream of random elements

Parallelism

Streams facilitate parallel execution: streams are pure functional and lazy evaluated, the order of evaluation do not matter, hence it can be parallelized.

```java
double average = pList
        .parallelStream() // parallel stream
        .filter(p -> p.getGender() == Person.Sex.MALE)
```

```java
            .mapToInt(Person::getAge)
            .average()
            .getAsDouble();
```

The runtime support takes care of using multithreading for parallel execution in a transparent way (which is good for simplicity, bad for debugging).
If the operations don't have side-effects the thread-safety is guaranteed even if non-thread-safe collections are used

The parallelism is integrated with everything we have seen

- concurrent mutable reduction is supported for parallel stream, there are suitable methods of `Collector`
- we can retrieve parallel stream from collections using `SplitIterator` which is an iterator on collections that support splitting data to allow parallel streams

When to use Parallel Stream

- when operations are independent and
- either or both
  - operations are computationally expensive (for very light operations the cost to allocate threads and to do context switches could outweigh the parallel speedup)
  - operations are applied to many elements of efficiently splitable data structures