Streams (Java 8)

Basics

Monads

- Java streams are not about IO
- Java streams are monads and part of the functional programming area.
- In functional programming, 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.

What is a stream?

- A sequence of elements.
- Not reusable.
- It is not a data structure.
- Usually it is infinite.
- We can perform functional-style operations on them like map-reduce.
- Reduction operations can be performed either sequentially or in parallel.
- Lazy evaluation for performance optimization.
- Function chaining (operation pipeline).

Intro example

Java stream example

```
void helloWorld(){
 "hello".chars()
   .mapToObj( i -> (char)i )
   .map( c -> Character.toUpperCase(c) )
   .forEach( System.out::println );
}
```

Parallel streams

- The stream can be parallel.
- Other than that we can work with the same way.
- By default, the streams are sequential (not parallel).
- Operations of parallel streams work on multiple threads.

Java stream example

```
void helloWorld(){
 "hello".chars().parallel()
   .mapToObj( i -> (char)i )
   .map( c -> Character.toUpperCase(c) )
   .forEach( System.out::println );
}
```

Operations

- Two types
  - intermediary
  - terminal
- Usually accept a lambda expression parameter (a functional interface) specifying the exact behaviour of the operation.
- Usually they must be both non-interfering and stateless.

Intermediary operations

- Return a new stream
- Usually it is lazy
- Intermediate operations will only be executed when a terminal operation is present.
- E.g. filter(), map()

Terminal operations

- All elements are processed
- Return a non-stream or has some side-effect.
- E.g. forEach(), collect(), min()

Non-interfering

- It does NOT modify the underlying stream
- Interfering operations can cause problems in parallel execution.

Stateless

- The execution of the operation is deterministic.
- The result does not depend on the mutable state (e.g. outer scope instance variable).
- Stateful operations can cause problems with parallel streams.

The java.util.stream package

- It includes the Java stream API classes and interfaces.
- Most notable types:
  - Stream<T>
    - A sequence of elements supporting sequential and parallel aggregate operations.
  - Collector<T,A,R>
    - A mutable reduction operation that accumulates input elements into a mutable result container, optionally transforming the accumulated result into a final representation after all input elements have been processed.

java.util.stream.Stream<T>

- A stream of elements that supports sequential and parallel aggregate operations.
- https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

Special streams

Types:

- IntStream
- LongStream
- DoubleStream

Special operations

- max()
- average()

Stream transformation

Object stream to Int/Long/Double stream

- Stream#mapToInt
- Stream#mapToLong
- Stream#mapToDouble

Int/Long/Double stream to Object stream

- IntStream#mapToObj
- LongStream#mapToObj
- DoubleStream#mapToObj

Stream creation

From any object

- Stream#of

Stream#of

```
Stream.of( new Date(), new Date() );
```

Special stream factory methods

- Factory methods like IntStream#range, IntStream#of, IntStream#generate ...

Special stream factory methods

```
IntStream.of( 1,2,3);
IntStream.range(0, 10);
IntStream.generate( () -> new Random().nextInt() );
```

From collection

- java.util.Collection#stream

Java stream example

```
new ArrayList<String>().stream();
```

From array

- java.util.Arrays#stream

Java stream example

```
Object[] array = new Object[]{};
Arrays.stream( array );
```

Operation processing order

- Operation pipeline. E.g. stream.filter().map().filter().max()
- Intermediary operations are lazy
- Lazy operations are executed when a terminal operation is called.
- Every element move along the pipeline independently from each other. ->
  It helps to reduce the number of operations. E.g. stream.filter().forEach().
  The forEach will only be executed with elements where the filter returned with true

For demo:

Java stream example

```
Arrays.stream( new int[]{1,2,3,4,5} )
     .map( i -> {
       System.out.println( "map: " + i);
       return i;
     })
     .filter( i -> {
      boolean filtered = i%2 == 0;
      System.out.println( "filter: " + i + " - " + filtered);
      return filtered;
     })
     .forEach( i-> System.out.println( "forEach: " + i) );

/*
map: 1
filter: 1 - false
map: 2
filter: 2 - true
forEach: 2
map: 3
filter: 3 - false
map: 4
filter: 4 - true
forEach: 4
map: 5
filter: 5 - false
*/
```

Complex Operations

Collect

- Transform the elements of the stream into a different kind of result, e.g. a `List`, `Set` or `Map`.
- Collect accepts a **`Collector`** which consists of four different operations: a supplier, an accumulator, a combiner and a fi nisher.

Collector<T,A,R>

- https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html
- Type Parameters:
  - T - the type of input elements to the reduction operation
  - A - the mutable accumulation type of the reduction operation (often hidden as an implementation detail)
  - R - the result type of the reduction operation
- supplier
  - creation of a new result container (supplier())

`Supplier<A>`	`supplier()` A function that creates and returns a new mutable result container.

- accumulator
  - incorporating a new data element into a result container (accumulator())

`BiConsumer<A,T>`	`accumulator()` A function that folds a value into a mutable result container.

- combiner
  - combining two result containers into one (combiner())

`BinaryOperator<A>`	`combiner()` A function that accepts two partial results and merges them.

- finisher
  - performing an optional final transform on the container (finisher())

`Function<A,R>`	`finisher()` Perform the final transformation from the intermediate accumulation type `A` to the final result type `R`.

E.g.:

Collector to collect to a list

supplier - (Supplier<List<T>>) ArrayList::new,

accumulator - List::add,

combiner - (left, right) -> { left.addAll(right); return left; },

finisher - none

java.util.stream.Collectors

- https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html
- Contains predefined reduction operators (Collector implementations).
- E.g.:
  - group by
  - collect toList, toSet, toMap
  - count
  - averaging
  - joining
  - mapping
  - partitioning
  - reducing
  - summing

flatMap

- Map one element of the stream to 0..n elements.
- FlatMap transforms each element of the stream into a stream of other objects and flat them on a new stream.
- The contents of those streams will then be placed into the returned stream of the `flatMap` operation.
- The `flatMap()` operation has the effect of applying a one-to-many transformation to the elements of the stream, and then flattening the resulting elements into a new stream.

Stream flatMap example

```
public List<LineItem> getAllLineItems( List<Order> orders ){
  return orders
    .stream()
    .flatMap( order -> order.getLineItems().stream() )
    .collect( Collectors.toList() );
}
```

reduce

- Combines all elements of the stream into a single result
- 3 types of reduce:
  - reducing the stream to one element that is on the stream
  - reducing the stream of T objects to a new T object
  - reducing the stream of T objects to a new T2 objects

Reducing the stream to one element that is on the stream

- Selecting one element from the stream
- It uses a BinaryOperator<T> accumulator to reduce elements.
- Method signature:

`Optional<T>`	`reduce(BinaryOperator<T> accumulator)` Performs a reduction on the elements of this stream, using an associativeaccumulation function, and returns an `Optional` describing the reduced value, if any.

Reduction of a stream to one element on the stream

```
streamOfCars
    .reduce((car1, car2) -> car1.getPrice() > car2.getPrice() ? car1 : car2)
    .ifPresent(System.out::println);
```

Reducing the stream to a new element

- Reducing the stream of T objects to a new T object
- It can be used to merge objects to a new object
- Metod signature:

`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.

Merging elements on a stream with reduce()

```
Account mergeAccounts(){
  return getAccounts()
    .stream()
    .reduce( new Account(),
      (account1, account2) -> { account1.addAmount( account2.getBalance() ); return account1; });
 }
```

Reducing the stream of T objects to a new T2 objects

- We transform a stream of objects to 0..n different objects.
- Metod signature:

`<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.

Merging elements on a stream with reduce()

```
Long sumBalance(){
  return getAccounts()
    .stream()
    .reduce( 0L, // Identity. The type is different than the type of the objects on the stream
      (sum, account) ->{ return sum += account.getBalance();}, // Accumulator
      (sum1, sum2) -> { return sum1 +sum2; }// Combiner
      );
}
```

Parallel execution

- Stream#parallel() -> Returns an equivalent stream that is parallel.

Default behaviour (common fork join pool)

- Parallel streams use the ForkJoinPool#commonPool() method.
- The size of the underlying thread-pool uses up to the amount of available physical CPU cores.
- You can overwrite the this default behaviour with the
    - -Djava.util.concurrent.ForkJoinPool.common.parallelism=5 JVM parameter.
- The common pool is a single point of contention contention.

Custom fork join pool

- You can create your own ForkJoinPool with it's thread pool and use it.
- See the explanation here:
    - https://dzone.com/articles/think-twice-using-java-8
    - http://stackoverflow.com/questions/21163108/custom-thread-pool-in-java-8-parallel-stream

Parallel streams with custom ForkJoinPool

```
final ForkJoinPool forkJoinPool = new ForkJoinPool(5);
final List primeNumbers = forkJoinPool.submit(() -> candidates.parallelStream().filter(Prime::isPrime).
                                        collect(Collectors.toList())).get();
```