STREAM API In JAVA

Creating streams

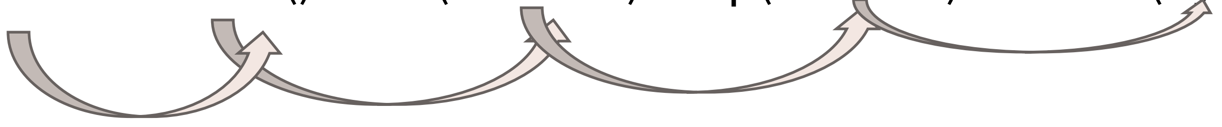
Pipelining tasks

Types of streams and tasks

Introduction to stream

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

```
numlist.stream().filter(e->e>20).map(e->e*20).forEach(System.out::println);
```


List converted to stream
Intermediatory operation on stream
Terminal operation on stream

In addition to Stream, which is a stream of object references, there are primitive specializations for IntStream, LongStream, and DoubleStream, all of which are referred to as "streams"

- To perform a computation, stream operations are composed into a stream 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 (which produces a result or side-effect, such as count() or forEach(Consumer)).

- Streams are lazy; computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed.

Stream Interface and it’s functions

allMatch(Predicate)

```
System.out.println(numlist.stream().allMatch((e)->e>20));
```

anyMatch(Predicate)

```
System.out.println(numlist.stream().anyMatch((e)->e>20));
```

builder

```
Stream.Builder<String> builder=Stream.builder();
builder.add("Switch");
builder.add("Router");
builder.add("Firewall");
builder.add("BTS");
builder.add("BSC");
builder.add("RNC");

Stream stream=builder.build();
stream.forEach(System.out::println);
```

Understanding Collector

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. Reduction operations can be performed either sequentially or in parallel.

```
List<Integer> list=Arrays.asList(12,23,11,19,55,34,29,32);
List<Integer> squareNumber=list.stream().map((e)->e*e).collect(Collectors.toList());
```

Creating a Collector using Collector.of function

```
Supplier<Integer> supplier=()->10;
BiConsumer<Integer,Integer> consumer=(a,b)->System.out.println(a+b);
BinaryOperator<Integer> binary=BinaryOperator.minBy((a,b)->{

 if(a==b)
 {
   return 0;
 }
else if(a>b)
 {
    return -1;
 }
  else
 {
 return 1;
 }
});
Collector<Integer,Integer,Integer>
collector=Collector.of(suppler,consumer,binary,Collector.Characteristics.CONCURRENT);

collector.accumulator().accept(10,20);
BinaryOperator<Integer> bio=collector.combiner();
System.out.println(bio.apply(10,20));
```

For examples visit
https://github.com/devopsasp/sdlcandotherdocs/tree/streamdemo