Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

The features of Java stream are –

* A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
* Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
* Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

What is a Stream?

A sequence of elements from a source that supports data processing operations.

Sequence of elements - Like a collection, a stream provides an interface to a sequenced set of values of a specific type.

Source - Streams refer to collections, arrays, or I/O resources.

Data processing operations - Supports common operations from functional programming languages.

e.g. filter, map, reduce, find, match, sort etc

They have nothing to do with java.io -- InputStream or outputstream

The Streams also support Pipelining and Internal Iterations. The Java 8 Streams are designed in such a way that most of its stream operations returns Streams only. This help us creating chain of various stream operations. This is called as pipelining. The pipelined operations looks similar to a sql query.(or Hibernate Query API)

Concurrency is IMPORTANT. But it comes with a learning curve.

So , Java 8 goes one more step ahead and has developed a Streams API which allows us to use multi cores easily.

Parallel processing = divide a larger task into smaller sub tasks (forking), then processing the sub tasks in parallel and then combining the results together to get the final output (joining).

Java 8 Streams API provides a similar mechanism to work with Java Collections.

The Java 8 Streams concept is based on converting Collections to a Stream (or arrays to a stream) , processing the elements in parallel and then gathering the resulting elements into a Collection.

Most stream operations accept some kind of lambda expression parameter, a functional interface specifying the exact behavior of the operation. Most of those operations must be both non-interfering and stateless. What does that mean?

A function is non-interfering when it does not modify the underlying data source of the stream, e.g.

List<String> myList =Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList.stream().filter(s -> s.startsWith("c")).map(String::toUpperCase) .sorted()

.forEach(System.out::println);

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.

API

The starting point is java.util.stream.Stream i/f

Different ways of creating streams

1. Can be created of any type of Collection (Collection, List, Set):

java.util.Collection<E> API

1.1 default Stream<E> stream()

1.2 public default Stream<E> parallelStream()

NOTE that Java 8 streams can’t be reused, will raise IllegalStateException

2. Stream of Array

How to create stream from an array?

Arrays class API

public static <T> Stream<T> stream(T[] array)

Returns a sequential Stream with the specified array as its source.

3. Can be attached to Map ,via entrySet method.

Refer to CreateStreams.java

4. To create streams out of three primitive types: int, long and double.

As Stream<T> is a generic interface , can't support primitives.

So IntStream, LongStream, DoubleStream are added.

API of java.util.stream.IntStream

4.1 static IntStream of(int... values)

Returns a sequential ordered stream whose elements are the specified values.

4.2 static IntStream range(int startInclusive,int endExclusive)

Returns a sequential ordered IntStream from startInclusive (inclusive) to endExclusive (exclusive) by an incremental step of 1.

4.3 static IntStream rangeClosed(int startInclusive,int endInclusive)

Returns a sequential ordered IntStream from startInclusive (inclusive) to endInclusive (inclusive) by an incremental step of 1.

5. To perform a sequence of operations over the elements of the data source and aggregate their results, three parts are needed – the source, intermediate operation(s) and a terminal operation.

6.java.util.stream.Stream<T> i/f API

6.1 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(stateful intermediate operation)

6.2 map

<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(intermediate stateless operation)

mapToInt

IntStream mapToInt(ToIntFunction<? super T> mapper)

Returns an IntStream consisting of the results of applying the given function to the elements of this stream.

6.3 filter

Stream<T> filter(Predicate<? super T> predicate)

Returns a stream consisting of the elements of this stream that match the given predicate.(intermediate stateless operation)

ref : StreamAPI1.java

-------------------

7. Confirm lazyness of streams.

Intermediate operations are lazy. This means that they will be invoked only if it is necessary for the terminal operation execution.

ref : LazyStreams.java

8. Reduce operation

Readymade methods of IntStream

count(), max(), min(), sum(),average()

9. Customized reduce operation

ref : ReduceStream.java

10 collect

Reduction of a stream can also be executed by another terminal operation – the collect() method.

eg : StreamCollect.java

Good examples in java.util.stream.Collectors -api docs.

-----------------------

Details ---

1. Streams are functional programming design pattern for processing sequence of elements sequentially or in parallel.(a.k.a Monad in functional programming)

2. Stream represents a sequence of elements and supports different kind of operations to perform computations upon those elements

3. Stream operations are either intermediate or terminal. Intermediate operations return a stream so we can chain multiple intermediate operations .

Terminal operations are either void or return a non-stream result.

4. They can't be reused.

5. Collections vs Streams:

Collections are in-memory data structures which hold elements within it. Each element in the collection is computed before it actually becomes a part of that collection. On the other hand Streams are fixed data structures which computes the elements on-demand basis.

The Java 8 Streams -- lazily constructed Collections, where the values are computed when user demands for it.

Actual Collections behave absolutely opposite to it and they are set of eagerly computed values (no matter if the user demands for a particular value or not).