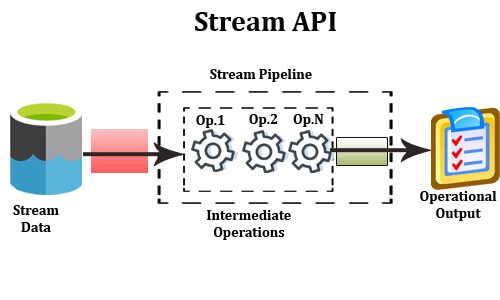
**Stream API**



The Streams API allows JavaScript to programmatically access streams of data received over the network and process them as desired by the developer.

**What is the use of stream API?**

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. A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.

The makers of Java 8 recently introduced a new feature called Stream API. The Stream API along with lambda expressions can be used to perform bulk operations on a sequence of elements, but that’s not all it can (or should) be used for. Multiple Stream operations can also be chained to perform a number of sequential operations.

In order to fully understand the concept behind Stream API, you’ll need a basic understanding of lambda expressions and functional interfaces in Java 8.

**Why use the Stream API?**

Suppose you have an integer array and you want to find all of the elements in that array that are even numbers. Before having Java 8 Streams, you’d need to write code similar to the following example:

**public** **static** **void** main(String[] args) {

List<Integer> aList = Arrays.*asList*(6,88,23,14,17,12,32,51,79,94);

List<Integer> evenNumbersList = **new** ArrayList<Integer>();

**for**(**int** num:aList){

**if**(num %2 == 0)

evenNumbersList.add(num);

}

}

Essentially, you’d need to iterate through the list and check if each number is even. If so, you’ll need to add this information to an output list.

By using Java 8 streams instead, you can re-write the above code as follows:

**public** **static** **void** main(String[] args) {

List<Integer> aList = Arrays.*asList*(6,88,23,14,17,12,32,51,79,94);

aList.stream().filter(num -> num %2 == 0);

}

Compared to the pre-Java 8 code, the code using Streams is far more concise. The stream API allows you to perform operations on collections without external iteration. In this case, we’re performing a filter operation which will filter the input collection based on the condition specified.

**How do Streams work?**

As we mentioned earlier, the Stream API in conjunction with lambda expressions can be used to perform bulk operations on Collections without the need for external iteration. The basic interface in the Stream API is called the ‘[java.util.Stream](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html)’, and there are various methods on the[Stream](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html) interface that perform various operations on the ***Stream*** instance.

In the above example, we’re invoking the[filter](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#filter-java.util.function.Predicate-) method. This accepts a[Predicate](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html) instance which is an in-built functional interface. Here, a lambda expression is used to implement the ***Predicate***interface. This lambda expression accepts an Integer value, checks if it is even via the modulus operator and returns a Boolean value accordingly. The[***filter***](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#filter-java.util.function.Predicate-) operation runs this lambda expression on every element in the input ***Stream*** and creates a new ***Stream***with the output. The result? A ***Stream*** with only even numbers.

An important point to remember is that ***Streams*** do not modify the underlying collection and the data in the underlying collection will not be affected by the Stream operations.

**Creating Streams**

There are several ways to create a Stream, including the following:

**From a Collection**

Our first example shows how a method called[stream](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html#stream--) has been added to all the Collection interfaces. When this method is invoked a ***Stream*** corresponding to the underlying Collection is returned. The following code demonstrates this:

List<Integer> aList = Arrays.asList(6,88,23,14,17,12);

Stream<Integer> stream = aList.stream();

**From an Array**

Streams can also be created using an array. There is an[Arrays.stream](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#stream-int:A-) method available which can be used as follows:

**int**[] array = **new** **int**[] {5,8,12};

IntStream arrayStream= Arrays.*stream*(array);

**From a List of elements**

Streams can also be created using a List of values using the[Stream.of](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#of-T...-) method like so:

Stream<Integer> anotherStream = Stream.*of*(5,7,14);

**Using Stream.generate method**

There is a[Stream.generate](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#generate-java.util.function.Supplier-) method available which can also be used to create a Stream as follows:

Stream<Integer> stream = Stream.generate(() -> new Random().nextInt(100));

The ***generate*** method accepts a[Supplier](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html) instance which is a functional interface. Here, a lambda expression is passed which generates a random number less than 100. This code will create a Stream of random integers less than 100.