**Stream API in Java 8**

The Stream API is a new addition to Java 8 that allows for functional-style processing of collections. Streams provide a simple and concise way to process collections of data by allowing operations such as filtering, mapping, and reducing to be performed in a fluent and readable manner. In this blog, we’ll explore the basics of the Stream API and some of the most common operations that can be performed on streams.

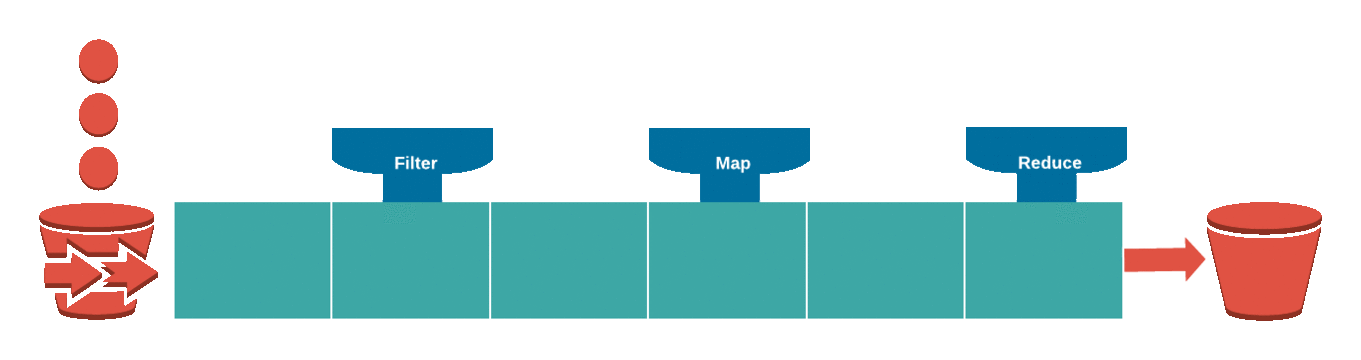


Image Ref link : <https://dz2cdn1.dzone.com/storage/temp/8567030-streams.gif>

**What is a Stream?**

A Stream is a sequence of elements that can be processed in parallel or sequentially. It is not a data structure, but rather an abstraction of a collection of elements that can be processed one at a time. Streams are created from collections or arrays using the stream() or parallelStream() method, respectively.

Here’s an example of creating a Stream from a list of integers:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);  
  
Stream<Integer> numberStream = numbers.stream();

**Basic Stream Operations**

Streams support a variety of operations that can be performed on their elements. These operations are divided into two types: intermediate operations and terminal operations.

Intermediate operations are operations that return a new Stream, allowing for chaining of operations. Some of the most common intermediate operations include filter(), map(), and sorted().

The filter() operation is used to select elements from the Stream that meet a specified condition. Here's an example of using filter() to select even numbers from a Stream of integers:

Stream<Integer> evenNumbers = numbers.stream()  
 .filter(n -> n % 2 == 0);

The map() operation is used to transform elements of the Stream. Here's an example of using map() to convert a Stream of integers to a Stream of strings:

Stream<String> numberStrings = numbers.stream()  
 .map(n -> "Number " + n);

The sorted() operation is used to sort elements of the Stream. Here's an example of using sorted() to sort a Stream of strings in ascending order:

Stream<String> sortedStrings = numberStrings.sorted();

Terminal operations are operations that produce a result or a side-effect. Some of the most common terminal operations include forEach(), count(), and reduce().

The forEach() operation is used to perform an action on each element of the Stream. Here's an example of using forEach() to print each element of a Stream of strings:

sortedStrings.forEach(System.out::println);

The count() operation is used to count the number of elements in the Stream. Here's an example of using count() to count the number of elements in a Stream of integers:

long count = numbers.stream().count();

The reduce() operation is used to combine the elements of the Stream into a single result. Here's an example of using reduce() to calculate the sum of a Stream of integers:

int sum = numbers.stream().reduce(0, (a, b) -> a + b);

In this example, we use reduce() with an initial value of 0 and a lambda expression that adds each element of the Stream to the accumulator a.

**Conclusion**

The Stream API is a powerful addition to Java 8 that provides a simple and concise way to process collections of data. By understanding the basics of the Stream API and the most common operations that can be performed on streams, developers can write more efficient and readable code for a variety of tasks in Java.