Stream

Stream is a new abstract layer introduced in Java 8. Using stream, you can process data in a declarative way similar to SQL statements. For example, consider the following SQL statement.

```
SELECT max(salary), employee_id, employee_name FROM Employee
```

The above SQL expression automatically returns the maximum salaried employee's details, without doing any computation on the developer's end.

Using collections framework in Java, a developer has to use loops and make repeated checks.

Another concern is efficiency; as multi-core processors are available at ease, a Java developer has to write parallel code processing that can be pretty error-prone.

To resolve such issues, Java 8 introduced the concept of stream that lets the developer to process data declaratively and leverage multicore architecture without the need to write any specific code for it.

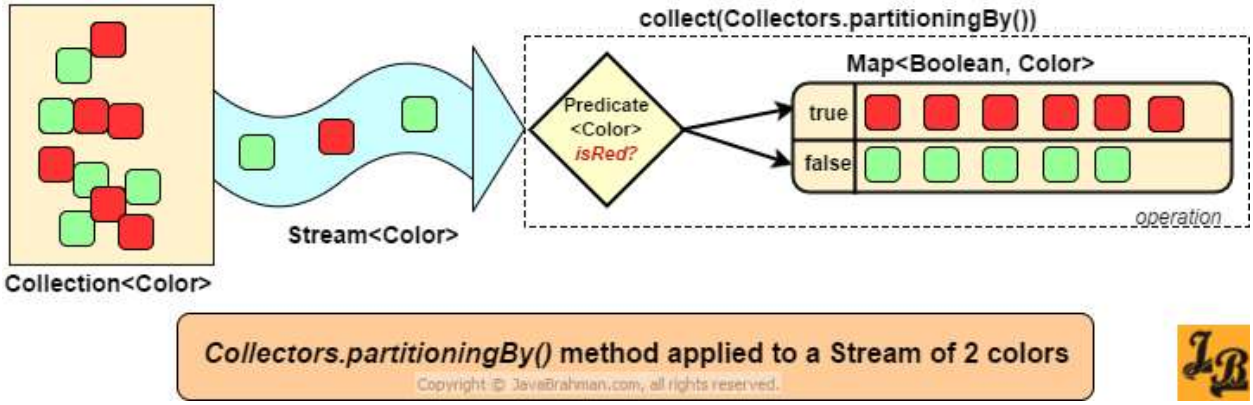
What is Stream?

Stream represents a sequence of objects from a source, which supports aggregate operations. Following are the characteristics of a Stream −

- **Sequence of elements** − A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.
- **Source** − Stream takes Collections, Arrays, or I/O resources as input source.
- **Aggregate operations** − Stream supports aggregate operations like **filter, map, limit, reduce, find, match, and so on.**
- **Pipelining** − Most of the stream operations return stream itself so that their result can be pipelined. These operations are called **intermediate operations**

and their function is to take input, process them, and return output to the target. collect() method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.

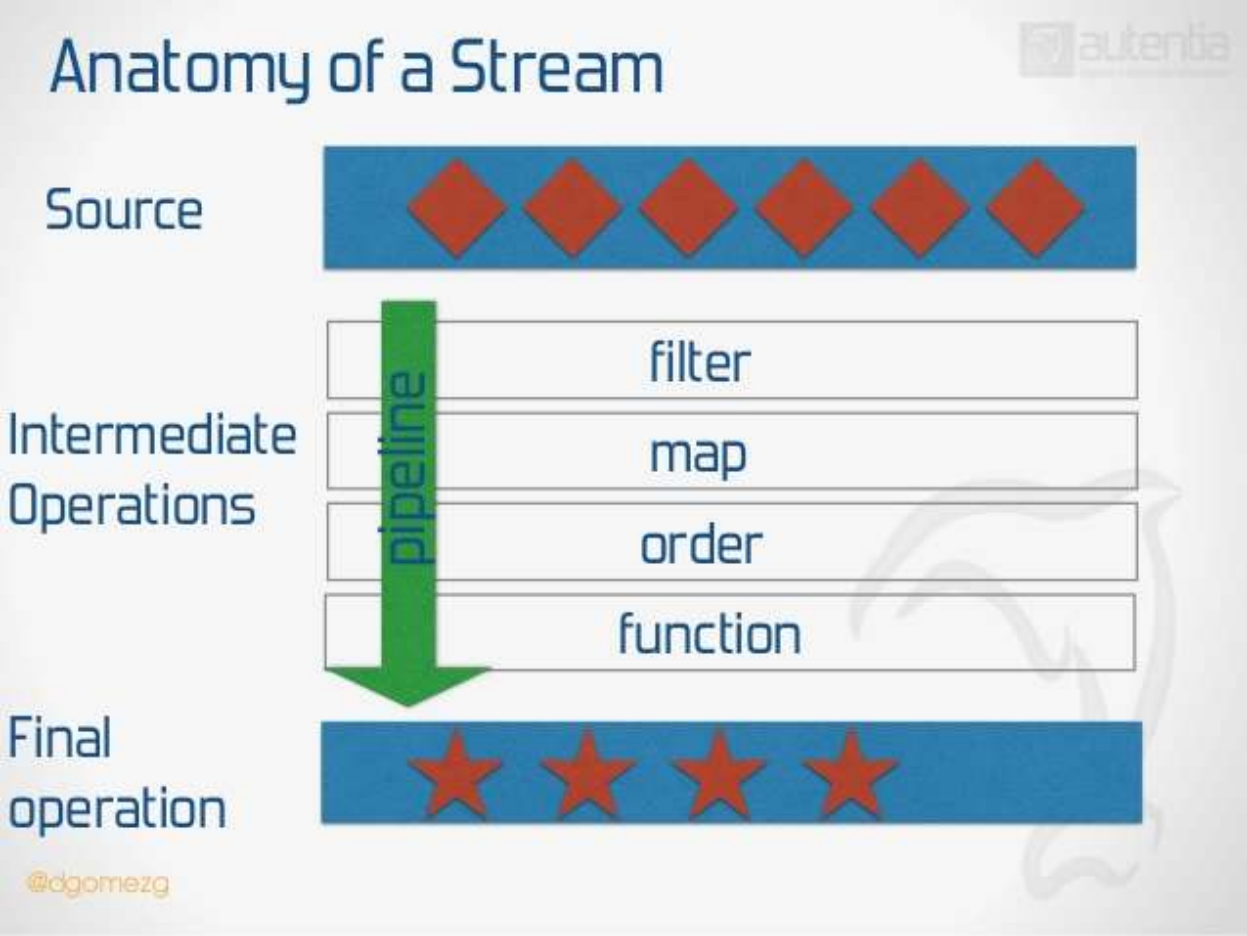
- **Automatic iterations** − Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.



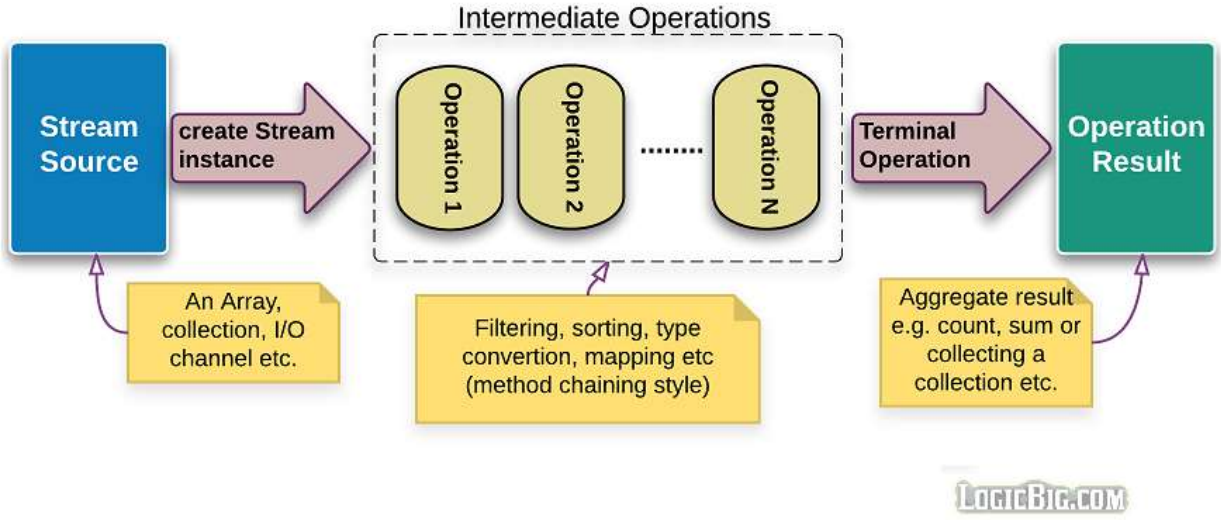
Generating Streams

With Java 8, Collection interface has two methods to generate a Stream.

- **stream()** − Returns a sequential stream considering collection as its source.
- **parallelStream()** − Returns a parallel Stream considering collection as its **source.**



Java Streams



```
List<String> strings = Arrays.asList("abc", "", "bc",
"efg", "abcd","", "jkl");

List<String> filtered = strings.stream().filter(string
-> !string.isEmpty()).collect(Collectors.toList());
```

forEach

Stream has provided a new method ‘forEach’ to iterate each element of the stream. The following code segment shows how to print 10 random numbers using forEach.

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::println);
```

map

The ‘map’ method is used to map each element to its corresponding result. The following code segment prints unique squares of numbers using map.

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3,
5);

//get list of unique squares
List<Integer> squaresList = numbers.stream().map( i ->
i*i).distinct().collect(Collectors.toList());
```

filter

The ‘filter’ method is used to eliminate elements based on a criteria. The following code segment prints a count of empty strings using filter.

```
List<String>strings = Arrays.asList("abc", "", "bc",
"efg", "abcd","", "jkl");

//get count of empty string
int count = strings.stream().filter(string ->
string.isEmpty()).count();
```

limit

The ‘limit’ method is used to reduce the size of the stream. The following code segment shows how to print 10 random numbers using limit.

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::println);
```

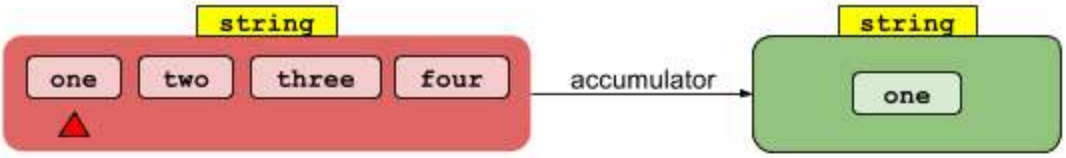
sorted

The ‘sorted’ method is used to sort the stream. The following code segment shows how to print 10 random numbers in a sorted order.

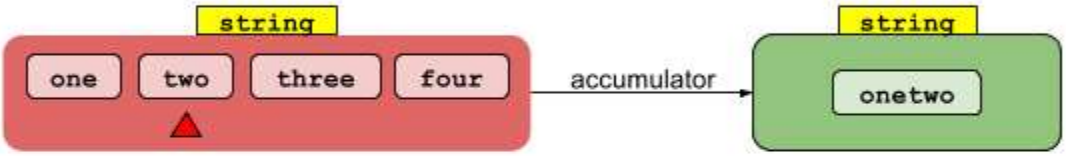
```
Random random = new Random();
random.ints().limit(10).sorted().forEach(System.out::println);
```

Parallel Processing

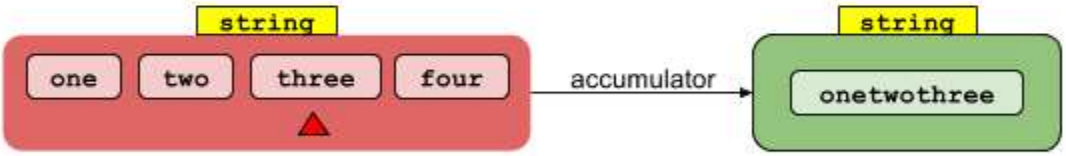
1. **Current element `one` is accumulated: string `one` is produced**



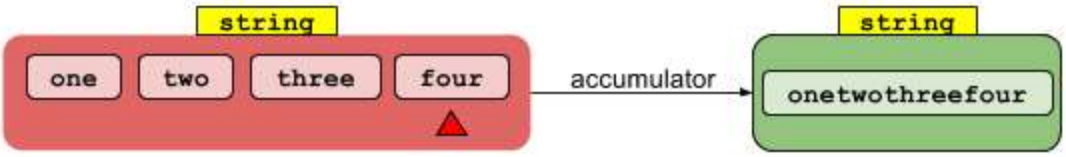
2. **Current element `two` is accumulated: string `onetwo` is produced**



3. **Current element `three` is accumulated: string `onetwothree` is produced**

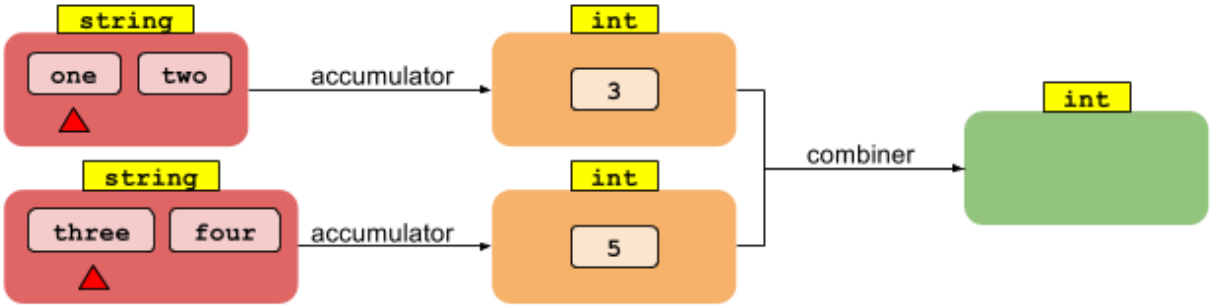


4. **Current element `four` is accumulated: string `onetwothreefour` is produced**

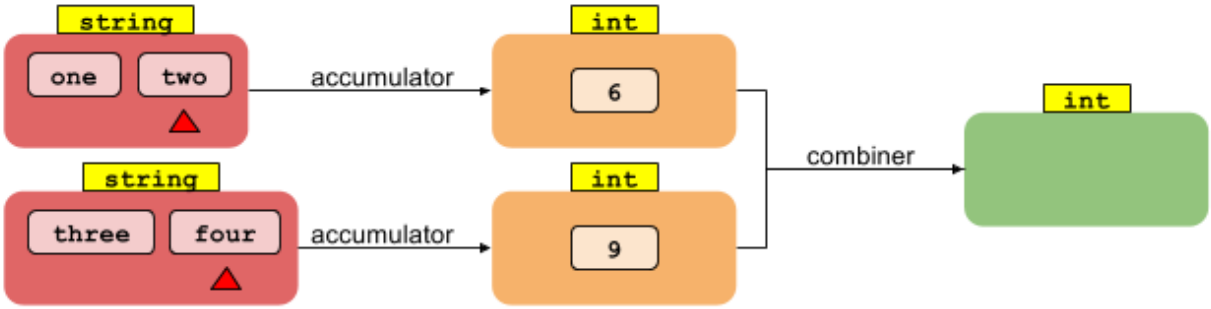


Parallel

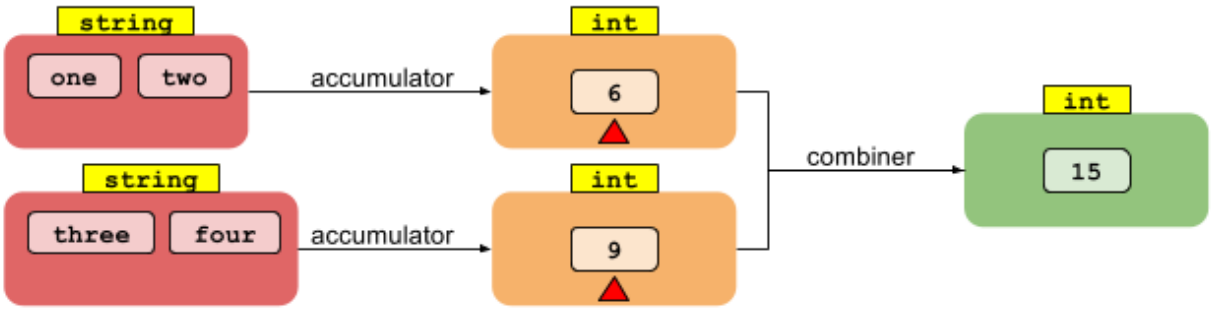
1. **Current elements `one` and `three` are accumulated: integers `3` and `5` are produced**



2. **Current elements `two` and `four` are accumulated: integers `6` and `9` are produced**



3. **Current elements `6` and `9` are combined: integer `15` is produced**



parallelStream is the alternative of stream for parallel processing. Take a look at the following code segment that prints a count of empty strings using parallelStream.

```
List<String> strings = Arrays.asList("abc", "", "bc",
"efg", "abcd","", "jkl");

//get count of empty string
long count = strings.parallelStream().filter(string ->
string.isEmpty()).count();
```

It is very easy to switch between sequential and parallel streams.

Collectors

Collectors are used to combine the result of processing on the elements of a stream. Collectors can be used to return a list or a string.

```
List<String>strings = Arrays.asList("abc", "", "bc",
"efg", "abcd","", "jkl");

List<String> filtered = strings.stream().filter(string
-> !string.isEmpty()).collect(Collectors.toList());

System.out.println("Filtered List: " + filtered);

String mergedString = strings.stream().filter(string ->
!string.isEmpty()).collect(Collectors.joining(", "));

System.out.println("Merged String: " + mergedString);
```

Statistics

With Java 8, statistics collectors are introduced to calculate all statistics when stream processing is being done.

```
List numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

IntSummaryStatistics stats = numbers.stream().mapToInt((x) ->
x).summaryStatistics();

System.out.println("Highest number in List : " +
stats.getMax());
```

```
System.out.println("Lowest number in List : " +
stats.getMin());
System.out.println("Sum of all numbers : " +
stats.getSum());
System.out.println("Average of all numbers : " +
stats.getAverage());
```