Twitter Thread by Vikas Rajput

Vikas Rajput
@vikasrajputin

Java: Beginner Guide to Stream API

a thread...

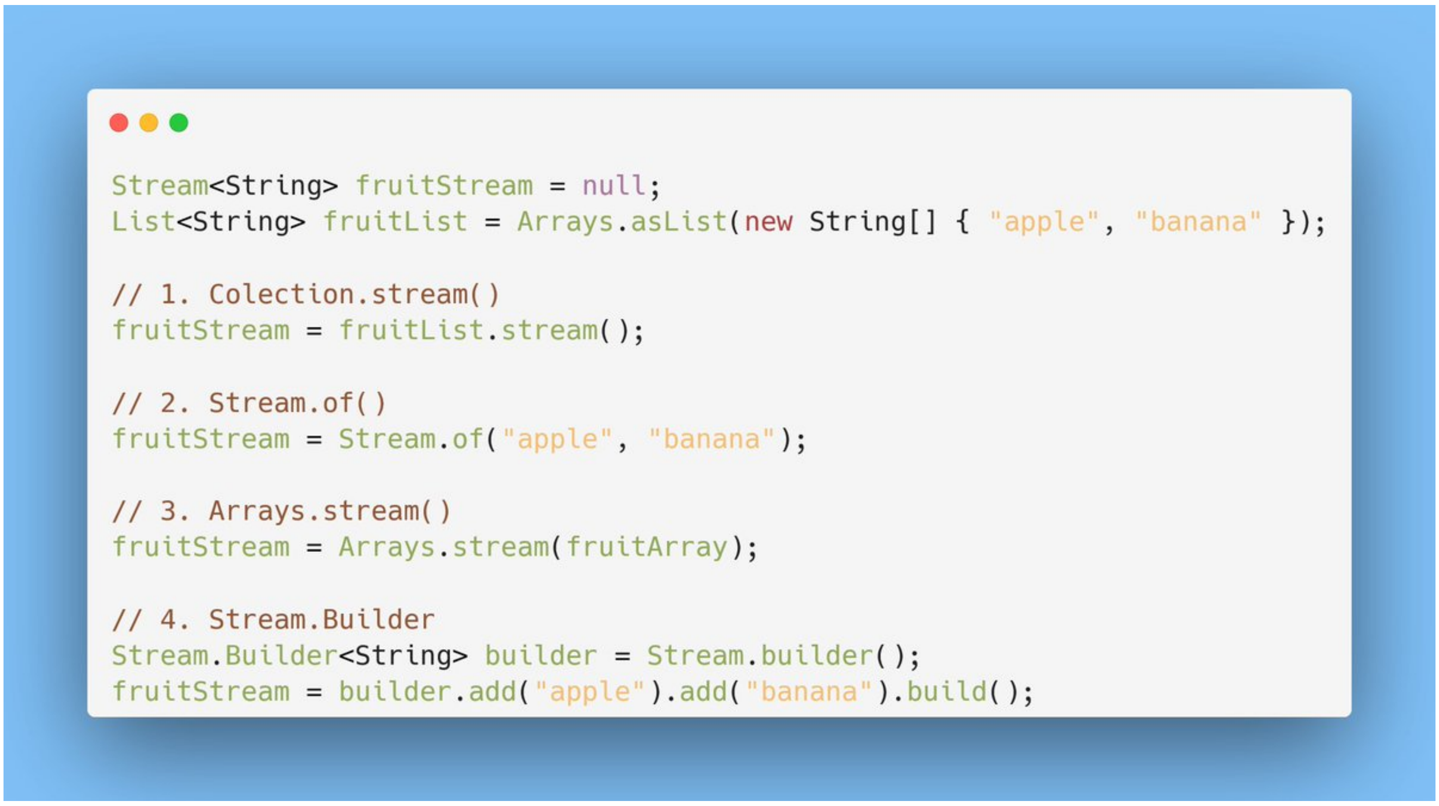
Streams are a way to perform operations on the whole collection of data.

Stream never modifies the source data, it processes it and gives us the result without modifying the original data.

Let's see few popular ways to create stream:

1. Collection. stream()
2. Stream.of(T... values)
3. Arrays. stream()
4. Stream.builder()

Eg:■



```java
Stream<String> fruitStream = null;
List<String> fruitList = Arrays.asList(new String[] { "apple", "banana" });

// 1. Colection.stream()
fruitStream = fruitList.stream();

// 2. Stream.of()
fruitStream = Stream.of("apple", "banana");

// 3. Arrays.stream()
fruitStream = Arrays.stream(fruitArray);

// 4. Stream.Builder
Stream.Builder<String> builder = Stream.builder();
fruitStream = builder.add("apple").add("banana").build();
```

Consider stream as a pipeline, where we perform mainly two different operations on the source data:

1. Intermediate Operations(IO)
2. Terminal Operations(TO)

Here's what the stream pipeline looks like:

Source -> Intermediate Operations -> Terminal Operations

Intermediate Operations(IO):

It returns another stream object, after this, we can either call another IO or TO.

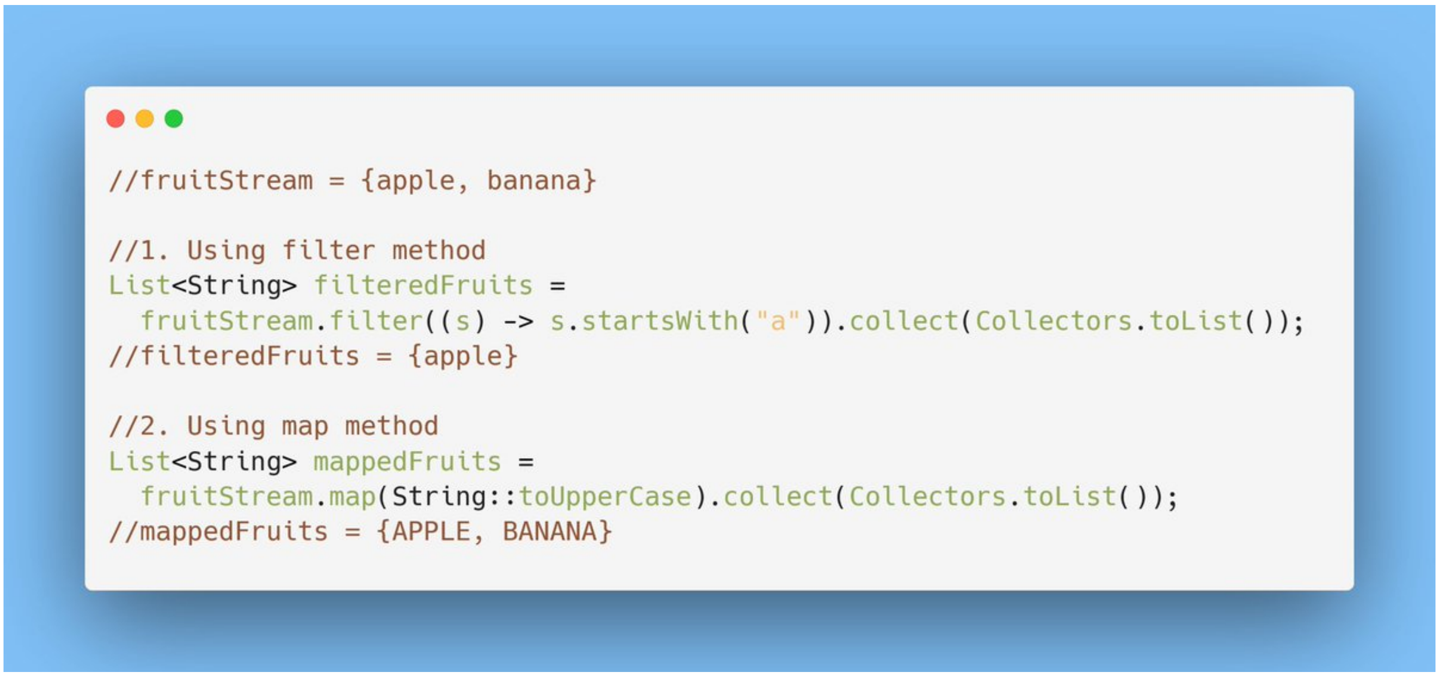
Few methods in Stream API for Intermediate operations:
1. filter()
2. map()
3. sorted()

and many more...

Examples:

filter() - filters out the data based on any boolean condition - below we're filtering elements starting with the letter a.

map() - performs some given operation on the whole data set - below we're converting every element to upper case.



```java
//fruitStream = {apple, banana}

//1. Using filter method
List<String> filteredFruits =
  fruitStream.filter((s) -> s.startsWith("a")).collect(Collectors.toList());
//filteredFruits = {apple}

//2. Using map method
List<String> mappedFruits =
  fruitStream.map(String::toUpperCase).collect(Collectors.toList());
//mappedFruits = {APPLE, BANANA}
```

Terminal Operations(TO):

It's the last operation done on stream that's why it's called terminal.

This operation returns the final result.

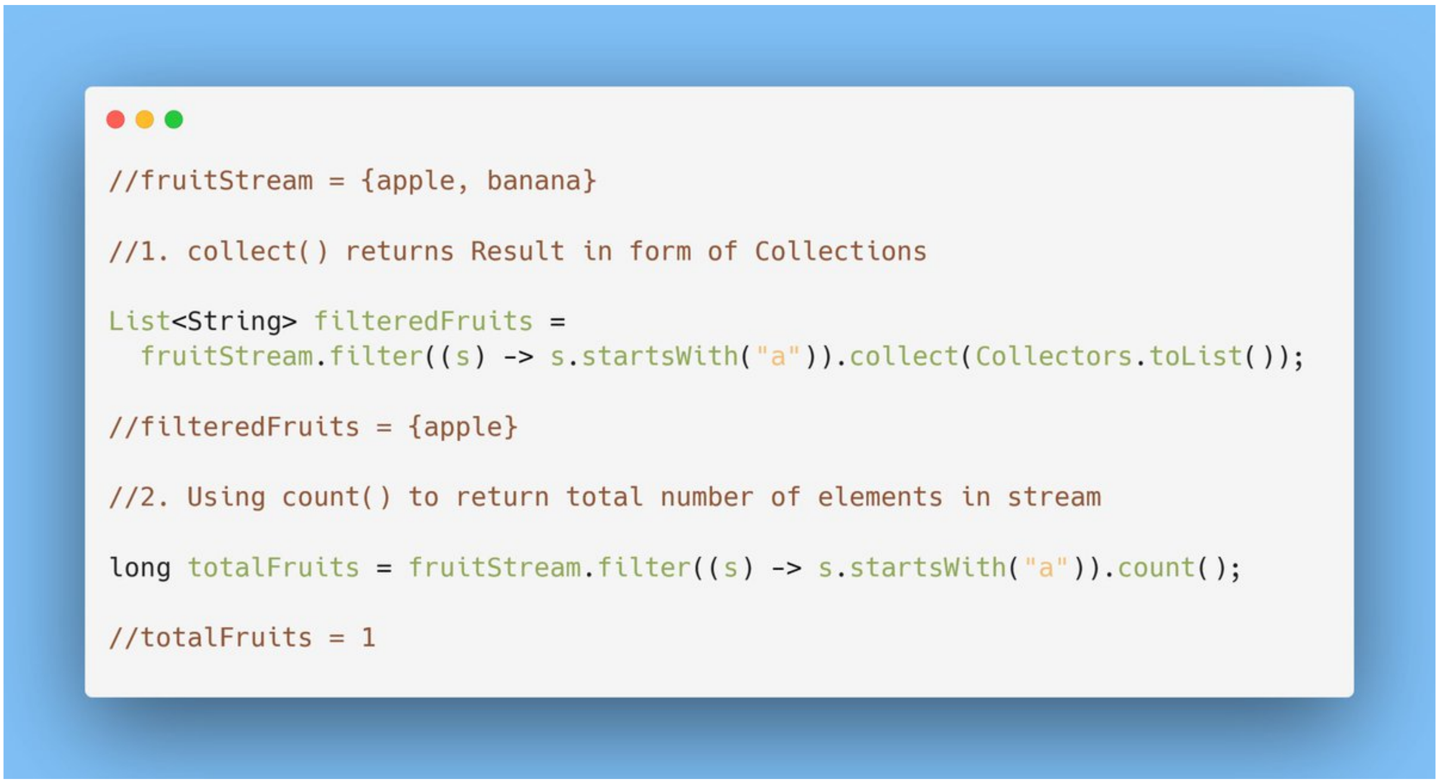
Few methods in Stream API for Terminal operations:
1. collect()
2. count()
3. forEach()

and many more...

Examples:

collect() - collects the elements in the given collection as shown below in the first case it is returning data in the form of a list.

count() - counts the no of the element in the stream and returns in the form of long.



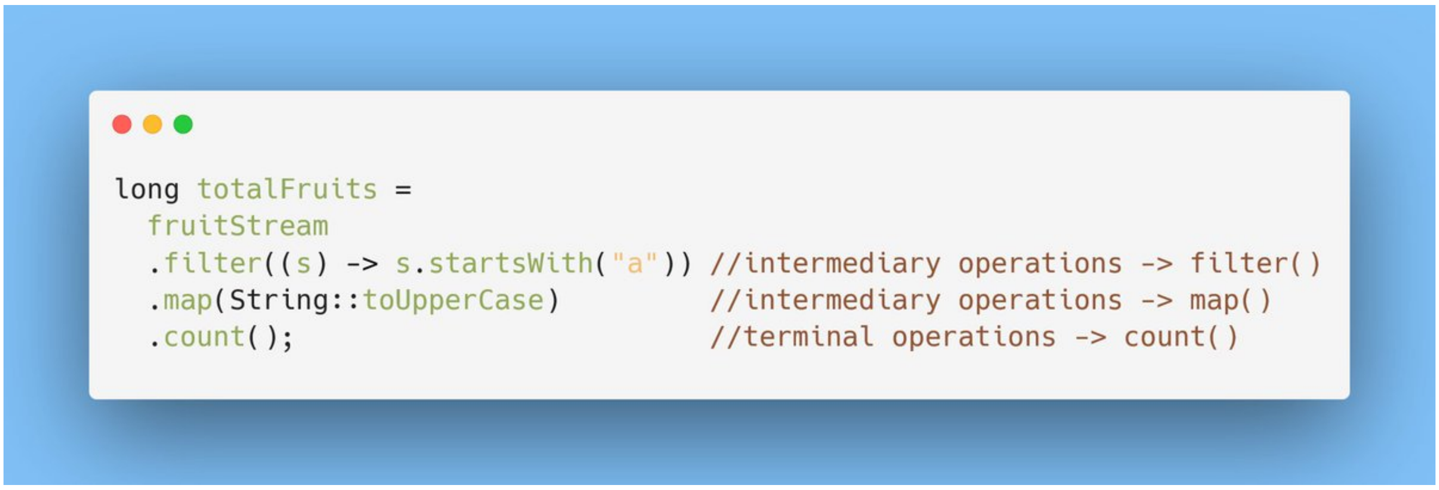
```java
//fruitStream = {apple, banana}

//1. collect() returns Result in form of Collections

List<String> filteredFruits =
  fruitStream.filter((s) -> s.startsWith("a")).collect(Collectors.toList());

//filteredFruits = {apple}

//2. Using count() to return total number of elements in stream

long totalFruits = fruitStream.filter((s) -> s.startsWith("a")).count();

//totalFruits = 1
```

Remember:

A Single pipeline can have multiple Intermediate Operations.

And only a single Terminal Operations.

Eg:
Below, we've used two Intermediate Operations i.e. filter() and map() but only used single Terminal Operation i.e. count().



```java
long totalFruits =
  fruitStream
  .filter((s) -> s.startsWith("a")) //intermediary operations -> filter()
  .map(String::toUpperCase)         //intermediary operations -> map()
  .count();                         //terminal operations -> count()
```