HowToDoInJava

Java Streams API

Last Updated: April 18, 2022 By: Lokesh Gupta Java Streams Java 8, Java Stream Basics, Lambda Expression

A *Stream in Java* can be defined as a **sequence of elements from a source**. The source of elements here refers to a Collection or Array that provides data to the Stream.

- Java streams are designed in such a way that most of the stream operations (called **intermediate operations**) return a Stream. This helps to create a chain of stream operations. This is called **stream pipe-lining**.
- Java streams also support the **aggregate or terminal operations** on the elements. The aggregate operations are operations that allow us to express common manipulations on stream elements quickly and clearly, for example, finding the max or min element, finding the first element matching giving criteria, and so on.
- Not that a ***stream maintains the same ordering*** **of the elements** ***as the ordering in the stream source***.

Table Of Contents

1. What is a Stream? Stream vs Collection?

All of us have watched online videos on Youtube. When we start watching a video, a small portion of the video file is first loaded into our computer and starts playing. we don't need to download the complete video before we start watching it. This is called video streaming.

At a very high level, we can think of the small portions of the video file as a stream and the whole video as a Collection.

At the granular level, the difference between a Collection and a Stream is to do with when the things are computed. A **Collection is an in-memory data structure, which holds all the values** that the data structure currently has.

Every element in the Collection has to be computed before it can be added to the Collection. While a **Stream is conceptually a pipeline, in which elements are computed on demand**.

This concept gives rise to significant programming benefits. The idea is that a user will extract only the values they require from a Stream, and these elements are produced, invisibly to the user, as and when required. This is a form of a producer-consumer relationship.

In Java, `java.util.Stream` interface represents a stream on which one or more operations can be performed. **Stream operations are either intermediate or terminal**.

The **terminal operations** return a result of a certain type and **intermediate operations** return the stream itself so we can chain multiple methods in a row to perform the operation in multiple steps.

Streams are created on a source, e.g. a `java.util.Collection` like `List` or `Set`. The `Map` is not supported directly, we can create stream of map keys, values or entries.

Stream operations can either be executed sequentially or parallel. when performed parallelly, it is called a *parallel stream*.

Based on the above points, **a stream is** :

- Not a data structure
- Designed for lambdas
- Do not support indexed access
- Can easily be aggregated as arrays or lists
- Lazy access supported
- Parallelizable

2. Creating Streams

The given below ways are the most popular different ways to build streams from collections.

2.1. Stream.of()

In the given example, we are creating a stream of a fixed number of integers.

```
Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9);
stream.forEach(p -> System.out.println(p));
```

2.2. Stream.of(array)

In the given example, we are creating a stream from the array. The elements in the stream are taken from the array.

```
Stream<Integer> stream = Stream.of( new Integer[]{1,2,3,4,5,6,7,8,9} );
stream.forEach(p -> System.out.println(p));
```

2.3. List.stream()

In the given example, we are creating a stream from the List. The elements in the stream are taken from the List.

```
List<Integer> list = new ArrayList<Integer>();

for(int i = 1; i< 10; i++){
      list.add(i);
}

Stream<Integer> stream = list.stream();
stream.forEach(p -> System.out.println(p));
```

2.4. Stream.generate() or Stream.iterate()

In the given example, we are creating a stream from generated elements. This will produce a stream of 20 random numbers. We have restricted the elements count using `limit()` function.

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

randomNumbers.limit(20).forEach(System.out::println);
```

2.5. Stream of String chars or tokens

In the given example, first, we are creating a stream from the characters of a given string. In the second part, we are creating the stream of tokens received from splitting from a string.

```
IntStream stream = "12345_abcdefg".chars();
stream.forEach(p -> System.out.println(p));
```

```
//OR

Stream<String> stream = Stream.of("A$B$C".split("\\$"));
stream.forEach(p -> System.out.println(p));
```

There are some more ways also such as using **Stream.Buider** or using intermediate operations. We will learn about them in separate posts from time to time.

3. Stream Collectors

After performing the intermediate operations on elements in the stream, we can collect the processed elements again into a Collection using the stream Collector methods.

3.1. Collect Stream elements to a List

In the given example, first, we are creating a stream on integers 1 to 10. Then we are processing the stream elements to find all even numbers.

At last, we are collecting all even numbers into a `List`.

```
List<Integer> list = new ArrayList<Integer>();

for(int i = 1; i< 10; i++){
      list.add(i);
}

Stream<Integer> stream = list.stream();
List<Integer> evenNumbersList = stream.filter(i -> i%2 == 0)
                                      .collect(Collectors.toList());
System.out.print(evenNumbersList);
```

3.2. Collect Stream elements to an Array

The given example is similar to the first example shown above. The only difference is that we are collecting the even numbers in an Array.

```
List<Integer> list = new ArrayList<Integer>();

for(int i = 1; i< 10; i++){
      list.add(i);
}

Stream<Integer> stream = list.stream();
Integer[] evenNumbersArr = stream.filter(i -> i%2 == 0).toArray(Integer[]::new);
System.out.print(evenNumbersArr);
```

There are plenty of other ways also to collect stream into a `Set`, `Map` or into multiple ways. Just go through `Collectors` class and try to keep them in mind.

4. Stream Operations

Stream abstraction has a long list of useful functions. Let us look at a few of them.

Before moving ahead, let us build a `List` of strings beforehand. We will build our examples on this list so that it is easy to relate and understand.

```
List<String> memberNames = new ArrayList<>();
memberNames.add("Amitabh");
memberNames.add("Shekhar");
memberNames.add("Aman");
memberNames.add("Rahul");
memberNames.add("Shahrukh");
memberNames.add("Salman");
memberNames.add("Yana");
memberNames.add("Lokesh");
```

These core methods have been divided into 2 parts given below:

4.1. Intermediate Operations

Intermediate operations return the stream itself so you can chain multiple methods calls in a row. Let's learn important ones.

4.1.1. Stream.filter()

The `filter()` method accepts a Predicate to filter all elements of the stream. This operation is intermediate which enables us to call another stream operation (e.g. forEach()) on the result.

```
memberNames.stream().filter((s) -> s.startsWith("A"))
                    .forEach(System.out::println);
```

Program Output:

```
Amitabh
Aman
```

4.1.2. Stream.map()

The `map()` intermediate operation converts each element in the stream into another object via the given function.

The following example converts each string into an UPPERCASE string. But we can use `map()` to transform an object into another type as well.

```
memberNames.stream().filter((s) -> s.startsWith("A"))
                  .map(String::toUpperCase)
                  .forEach(System.out::println);
```

Program Output:

```
AMITABH
AMAN
```

4.1.2. Stream.sorted()

The `sorted()` method is an intermediate operation that returns a sorted view of the stream. The elements in the stream are sorted in natural order unless we pass a custom Comparator.

```
memberNames.stream().sorted()
                   .map(String::toUpperCase)
                   .forEach(System.out::println);
```

Program Output:

```
AMAN
AMITABH
LOKESH
RAHUL
SALMAN
SHAHRUKH
SHEKHAR
YANA
```

Please note that the `sorted()` method only creates a sorted view of the stream without manipulating the ordering of the source Collection. In this example, the ordering of string in the `memberNames` is untouched.

4.2. Terminal operations

Terminal operations return a result of a certain type after processing all the stream elements.

Once the terminal operation is invoked on a Stream, the iteration of the Stream and any of the chained streams will get started. Once the iteration is done, the result of the terminal operation is returned.

4.2.1. Stream.forEach()

The `forEach()` method helps in iterating over all elements of a stream and perform some operation on each of them. The operation to be performed is passed as the lambda expression.

```
memberNames.forEach(System.out::println);
```

4.2.2. Stream.collect()

The `collect()` method is used to receive elements from a steam and store them in a collection.

```
List<String> memNamesInUppercase = memberNames.stream().sorted()
                            .map(String::toUpperCase)
                            .collect(Collectors.toList());

System.out.print(memNamesInUppercase);
```

Program Output:

```
[AMAN, AMITABH, LOKESH, RAHUL, SALMAN, SHAHRUKH, SHEKHAR, YANA]
```

4.2.3. Stream.match()

Various matching operations can be used to check whether a given predicate matches the stream elements. All of these matching operations are terminal and return a `boolean` result.

```
boolean matchedResult = memberNames.stream()
        .anyMatch((s) -> s.startsWith("A"));

System.out.println(matchedResult);     //true

matchedResult = memberNames.stream()
        .allMatch((s) -> s.startsWith("A"));

System.out.println(matchedResult);     //false

matchedResult = memberNames.stream()
        .noneMatch((s) -> s.startsWith("A"));

System.out.println(matchedResult);     //false
```

4.2.4. Stream.count()

The `count()` is a terminal operation returning the number of elements in the stream as a `long` value.

```
long totalMatched = memberNames.stream()
    .filter((s) -> s.startsWith("A"))
    .count();
```

```
System.out.println(totalMatched);     //2
```

4.2.5. Stream.reduce()

The `reduce()` method performs a reduction on the elements of the stream with the given function. The result is an `Optional` holding the reduced value.

In the given example, we are reducing all the strings by concatenating them using a separator **#**.

```
Optional<String> reduced = memberNames.stream()
        .reduce((s1,s2) -> s1 + "#" + s2);

reduced.ifPresent(System.out::println);
```

Program Output:

```
Amitabh#Shekhar#Aman#Rahul#Shahrukh#Salman#Yana#Lokesh
```

5. Short-circuit Operations

Though stream operations are performed on all elements inside a collection satisfying a Predicate, it is often desired to break the operation whenever a matching element is encountered during iteration.

In external iteration, we will do with the if-else block. In the internal iterations such as in streams, there are certain methods we can use for this purpose.

5.1. Stream.anyMatch()

The `anyMatch()` will return `true` once a condition passed as predicate satisfies. Once a matching value is found, no more elements will be processed in the stream.

In the given example, as soon as a String is found starting with the letter `'A'`, the stream will end and the result will be returned.

```
boolean matched = memberNames.stream()
        .anyMatch((s) -> s.startsWith("A"));

System.out.println(matched);    //true
```

5.2. Stream.findFirst()

The `findFirst()` method will return the first element from the stream and then it will not process any more elements.

```
String firstMatchedName = memberNames.stream()
            .filter((s) -> s.startsWith("L"))
            .findFirst()
                        .get();

System.out.println(firstMatchedName);     //Lokesh
```

6. Parallel Streams

With the Fork/Join framework added in Java SE 7, we have efficient machinery for implementing parallel operations in our applications.

But implementing a fork/join framework is itself a complex task, and if not done right; it is a source of complex multi-threading bugs having the potential to crash the application. With the introduction of internal iterations, we got the possibility of operations to be done in parallel more efficiently.

To enable parallelism, all we have to do is to create a parallel stream, instead of a sequential stream. And to surprise, this is really very easy.

In any of the above-listed stream examples, anytime we want to do a particular job using multiple threads in parallel cores, all we have to call **parallelStream()** method instead of `stream()` method.

```
List<Integer> list = new ArrayList<Integer>();
for(int i = 1; i< 10; i++){
 list.add(i);
}

//Here creating a parallel stream
Stream<Integer> stream = list.parallelStream();

Integer[] evenNumbersArr = stream.filter(i -> i%2 == 0).toArray(Integer[]::new);
System.out.print(evenNumbersArr);
```

A key driver for Stream APIs is making parallelism more accessible to developers. While the Java platform provides strong support for concurrency and parallelism already, developers face unnecessary impediments in migrating their code from sequential to parallel as needed.

Therefore, it is important to encourage idioms that are both sequential- and parallel-friendly. This is facilitated by shifting the focus towards describing what computation should be performed, rather than how it should be performed.

It is also important to strike the balance between making parallelism easier but not going so far as to make it invisible. Making parallelism transparent would introduce non-determinism and the possibility of data races where

users might not expect it.

7. Working with Streams

7.1 Creating Streams

- concat()
- empty()
- generate()
- iterate()
- of()

7.2 Intermediate Operations

- filter()
- map()
- flatMap()
- distinct()
- sorted()
- peek()
- limit()
- skip()

7.3. Terminal Operations

- forEach()
- forEachOrdered()
- toArray()
- reduce()
- collect()
- min()
- max()
- count()
- anyMatch()
- allMatch()
- noneMatch()
- findFirst()

- findAny()

Happy Learning !!

Sourcecode on Github

Recommended Reading:

1. Creating Streams in Java
2. Primitive Type Streams in Java
3. Boxed Streams in Java
4. Using 'if-else' Conditions with Java Streams
5. Creating Infinite Streams in Java
6. Sorting Streams in Java
7. Applying Multiple Conditions on Java Streams
8. Finding Max and Min from List using Streams
9. Java WatchService API Tutorial
0. Java Stream findFirst() vs findAny() API With Example

18 thoughts on “Java Streams API”

Shatakshi

April 10, 2020 at 3:42 pm

Its an informative post.
But I have one query here, in terms of performance streams are better or for-loops. As much I have read online, answer is for loops then why should we prefer using streams?

Reply

Lokesh Gupta

April 10, 2020 at 10:41 pm

The gains are so much for most of the real-world examples. In most applications, we will not iterate over collections having 1 million or more records. If that is ever needed, it is done on the database side.

The lambda expressions are amazing in writing complex logic in a single line.

Reply

Rajanikant

November 11, 2019 at 11:39 pm

Hi,
How to breakout from loop from 2.4
I have thought to apply break statement but in consumer we can't apply as its not a loop statement.
or is it stream should have only fixed size.?

Reply

deeks

October 1, 2019 at 3:57 pm

```
Stream s=List.StreamOf(new int[]{1,2,3});
```

not working ?what is the error

Reply

alfred

October 1, 2019 at 4:00 pm

the syntax is wrong
it is

```
Stream s=Stream,of(int[]{1,2,3,4});
```

try this

Reply

SALIL KUMAR

June 23, 2019 at 9:03 pm

Parallelism in Java Steam

```
public class StreamBuilders {
public static void main(String[] args){
List list = new ArrayList();
for(int i = 1; i< 10; i++){
list.add(i);
}
//Here creating a parallel stream
Stream stream = list.parallelStream();
Integer[] evenNumbersArr = stream.filter(i -> i%2 == 0).toArray(Integer[]::new);
System.out.print(evenNumbersArr);
}
}
```

OUTPUT
[Ljava.lang.Integer;@7cca494b

Below is the correct implementaion
————————————

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;

public class StreamBuildersParallelStream {

public static void main(String[] args) {
// TODO Auto-generated method stub
List list = new ArrayList();
for(int i = 1; i< 10; i++){
list.add(i);
}
//Here creating a parallel stream
Stream stream = list.parallelStream();
//Integer[] evenNumbersArr = stream.filter(i -> i%2 == 0).toArray(Integer[]::new);

Stream highNums = stream.filter(i -> i%2 == 0);
//System.out.print(highNums);
highNums.forEach(p -> System.out.println("Even Nums parallel="+p));
}
```

```
}
```

OUTPUT:

———

Even Nums parallel=6
Even Nums parallel=4
Even Nums parallel=8
Even Nums parallel=2

Reply

tushar singhal

April 12, 2019 at 12:40 pm

Hi ,

Thanks for a wonderful post. I saw a different post on streams and according to them sequential streams are slower than traditional loops. What are you inputs on this.

Secondly, in another post i saw that we should not use parallel streams specially in big applications beacause it uses fork()/join framework . Then why should streams be used at all. Just for the future to make something parallel down the line?

Reply

Lokesh Gupta

April 12, 2019 at 1:54 pm

If we consider performance and easiness in debugging, loops can not be beaten. Loops are much light weight and definitely perform better than streams.

But the performance is not so much that we do not consider the streams at all. Streams are more of a coding style where we can express our logic/intent in more declarative way. Stream operations are more cleaner and redable.

So it's more of coding/syntax style why we choose streams. Otherwise everything can be achieved via loops/if-else statements as well, but they may be hard to understand and may look ugly in comparison.

Reply

Swathi

February 14, 2019 at 5:24 am

Question regarding Stream, in the following code once the findFirst() method is reached, why are other elements being evaluated? for example I though “116, 345” these will not get printed because after the filter is evaluated and found the first the stream will be closed.

```
List list1 = Stream.of("11", "12", "11113", "116", "345").collect(Collectors.toList());
List list2 = Stream.of("23333", "33", "433", "5").collect(Collectors.toList());

Optional str = Stream.of(list1 , list2 )
        .peek(System.out::println)
        .flatMap(x -> x.stream())
        .peek(System.out::println)
        .filter(x -> x.length() > 3)
        .peek(System.out::println)
        .findFirst();

System.out.println("--str--" + str);
```

OutPut:

[11, 12, 11113, 116, 345]
11
12
11113
11113
116
345
–str–Optional[11113]

Reply

Lokesh Gupta

February 14, 2019 at 12:31 pm

Excellent question. I will do the research on my side and let you know. Please share if you get anything.

Reply

Laxminarsaiah Ragi

March 27, 2019 at 5:24 pm

List list1 = Arrays.asList(“11”, “12”, “11113”, “116”, “345”);

Optional str = list1.stream()
.peek(System.out::println)
.filter(x -> x.length() > 3)
.peek(System.out::println)
.findFirst();

System.out.println(“–str– ” + str);

I guess flat map is the culprit here, the simple map represents one-one check whereas flat map is one-many check when you use a flat map I am thinking it will create a new stream with same values, that why it is printing.

Sorry, I don’t know much about java8, the above code will not evaluate “116” and “345”

Reply

Laxminarsaiah Ragi

March 27, 2019 at 5:28 pm

```
List list1 = Arrays.asList("11", "12", "11113", "116", "345");

Optional str = list1.stream()
            .peek(System.out::println)
            .filter(x -> x.length() > 3)
            .peek(System.out::println)
            .findFirst();

System.out.println("--str--   " + str);
```

Output:
11
12
11113
11113
–str– Optional[11113]

Reply

Santhosh Kaitheri

June 15, 2019 at 2:42 pm

Laxminarsaiah Ragi :

The flatMap is also not the culprit. In fact the code is not evaluating further once it filters x->x.length()>3. Since you are printing the list before that, it gets displayed. If you execute below code, you will see it prints only 1113 .

```
List list1 = Stream.of("11", "12", "11113", "116", "345").collect(Collectors.toList());
        List list2 = Stream.of("23333", "33", "433", "5").collect(Collectors.toList());

        Optional str = Stream.of(list1 , list2 )
          // .peek(System.out::println)
           .flatMap(x -> x.stream())
           //.peek(System.out::println)
           .filter(x -> x.length() > 3)
           .peek(System.out::println)
           .findFirst();

        System.out.println("--str--" + str);
```

Output :

```
11113
–str–Optional[11113]
```

Reply

Pankaj

February 9, 2018 at 8:45 am

```
//I am looking for loop using streams as follows:

int inputArray[] = {4, 6, 5, -10, 8, 1, 2};

for (int i : inputArray)
{
```

```
for (int j = i+1; j {
IntStream.range(i+1,… ).forEach(j -> {

});
});
```

Reply

Jamuna Dey

November 21, 2018 at 5:57 pm

```
Stream intData = Stream.of(new Integer[] {4, 6, 5, -10, 8, 1, 2});
intData.forEach(n->System.out.println(n));
```

Reply

Prabhat

February 13, 2015 at 7:15 am

Example four –

```
public class StreamBuilders {
    public static void main(String[] args){
        Stream stream = Stream.generate(() -> { return new Date();});
        stream.forEach(p -> System.out.println(p));
    }
}
```

This is going in infinite loop…. why ??

Reply

Sorena

January 18, 2016 at 5:06 pm

Because:
generate(Supplier s) Returns an infinite sequential unordered stream where each element is generated by the provided Supplier. This is suitable for generating constant streams, streams of random elements, etc.

Reply

Jamuna Dey

November 21, 2018 at 6:06 pm

You can use limit to handle it –

```
Stream stream = Stream.generate(() -> { return new Date();}).limit(5);
stream.forEach(p -> System.out.println(p));
```

Reply

Leave a Comment

Name *

Email *

Website

Add me to your newsletter and keep me updated whenever you publish new blog posts

Post Comment

Search …

HowToDoInJava

A blog about Java and related technologies, the best practices, algorithms, and interview questions.

Meta Links

- About Me
- Contact Us
- Privacy policy
- Advertise
- Guest Posts

Blogs

REST API Tutorial