

Java 9 Stream API Improvements

📅 Last Updated: August 30, 2020 👤 By: Lokesh Gupta 📁 Java 9 💡 Java Stream Basics

Learn new java 9 improvements in [Stream](#) API i.e. `takeWhile` / `dropWhile` methods, `ofNullable` and `iterate` methods with examples.

Table of Contents

[Limiting Stream with takeWhile\(\) and dropWhile\(\) methods](#)

[Overloaded Stream iterate method](#)

[New Stream ofNullable\(\) method](#)

Limiting Stream with takeWhile() and dropWhile() methods

The new methods `takeWhile` and `dropWhile` allow you to get portions of a stream based on a predicate. Here a stream can be either ordered or unordered, so :

1. On an ordered stream, `takeWhile` returns the "longest prefix" of elements taken from the stream that match the given predicate, starting at the beginning of the stream.
2. On an un-ordered stream, `takeWhile` returns a subset of the stream's elements that match the given predicate (but not all), starting at the beginning of the stream.

The `dropWhile` method does the opposite of `takeWhile` method.

1. On an ordered stream, `dropWhile` returns remaining items after the "longest prefix" that match the given predicate.
2. On an un-ordered stream, `dropWhile` returns remaining stream elements after dropping subset of elements that match the given predicate.

takeWhile and dropWhile Example

In this example, we have list of chars from 'a' to 'i'. I want all chars which may appear before char 'd' in iteration.

```
List<String> alphabets = List.of("a", "b", "c", "d", "e", "f", "g", "h", "i");

List<String> subset1 = alphabets
    .stream()
    .takeWhile(s -> !s.equals("d"))
    .collect(Collectors.toList());
```

```
System.out.println(subset1);
```

Output:

```
[a, b, c]
```

As stated before, **dropWhile** acts opposite to **takeWhile** method so in above example, if used, it will return all characters which were left by **takeWhile** predicate.

```
List<String> alphabets = List.of("a", "b", "c", "d", "e", "f", "g", "h", "i");
```

```
List<String> subset2 = alphabets  
    .stream()  
    .dropWhile(s -> !s.equals("d"))  
    .collect(Collectors.toList());
```

```
System.out.println(subset2);
```

Output:

```
[d, e, f, g, h, i]
```

Overloaded Stream iterate method

iterate() methods used for creating a stream which starts with a single element (the seed), and subsequent elements are produced by successively applying the unary operator. The result is an infinite stream. To terminate the stream, a limit or some other short-circuiting function, like **findFirst** or **findAny** is used.

The **iterate** method in Java 8 has the signature:

```
static Stream iterate(final T seed, final UnaryOperator f)
```

In Java 9, new overloaded version of **iterate** takes a Predicate as the second argument:

```
static Stream iterate(T seed, Predicate hasNext, UnaryOperator next)
```

Let's see the difference in use of **iterate** method from java 8 to java 9.

iterate method in Java 8

```
List<Integer> numbers = Stream.iterate(1, i -> i+1)  
    .limit(10)  
    .collect(Collectors.toList());
```

```
System.out.println(numbers);
```

Output:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

iterate method in Java 9

```
List<Integer> numbers = Stream.iterate(1, i -> i <= 10 ,i -> i+1)
                              .collect(Collectors.toList());
```

```
System.out.println(numbers);
```

Output:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In above examples, the first stream is the Java 8 way of using iterate with a limit. The second one uses a Predicate as the second argument.

New Stream ofNullable() method

Until Java 8, you cannot have `null` value in a stream. It would have caused `NullPointerException`.

In Java 9, the `ofNullable` method lets you create a **single-element stream** which wraps a value if not null, or is an empty stream otherwise.

```
Stream<String> stream = Stream.ofNullable("123");
System.out.println(stream.count());
```

```
stream = Stream.ofNullable(null);
System.out.println(stream.count());
```

Output:

```
1
0
```

Here, the `count` method returns the number of non-empty elements in a stream.

Technically, `Stream.ofNullable()` is very similar to null condition check, when talking in context of stream API.

Drop me your questions in comments section.

Happy Learning !!

Was this post helpful?

Let us know if you liked the post. That's the only way we can improve.

Yes

No

Recommended Reading:

1. [Java 14 – New Features and Improvements](#)
2. [Java Stream findFirst\(\) vs findAny\(\) API With Example](#)
3. [Java Stream API Tutorials](#)
4. [Java Stream reuse – traverse stream multiple times?](#)
5. [Get all Dates between Two Dates as Stream](#)
6. [Java Streams API](#)
7. [JavaMail API – Java program to send email – Gmail SMTP server example](#)
8. [Java 11 – Files writeString\(\) API](#)
9. [Java REST API Tutorials](#)
0. [Introduction to the Java Date/Time API](#)



Join 7000+ Awesome Developers

Get the latest updates from industry, awesome resources, blog updates and much more.

Email Address

Subscribe

** We do not spam !!*



Working to make
Scotland a great place
to grow older

Find out more

agescotland.org.uk

1 thought on “Java 9 Stream API Improvements”

subham

January 10, 2020 at 11:20 am

How to write this Stream.ofNullable in java 8.

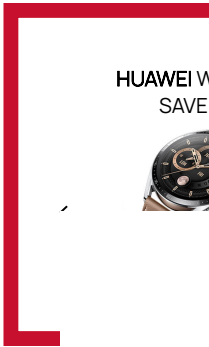
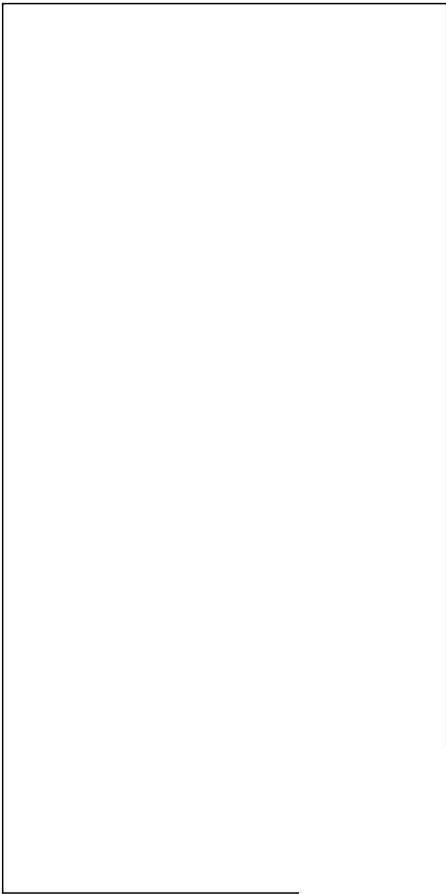
[Reply](#)

Leave a Comment

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

Post Comment





HUAWEI M
SAVE



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](#)



Copyright © 2022 · Hosted on [Cloudways](#) · [Sitemap](#)