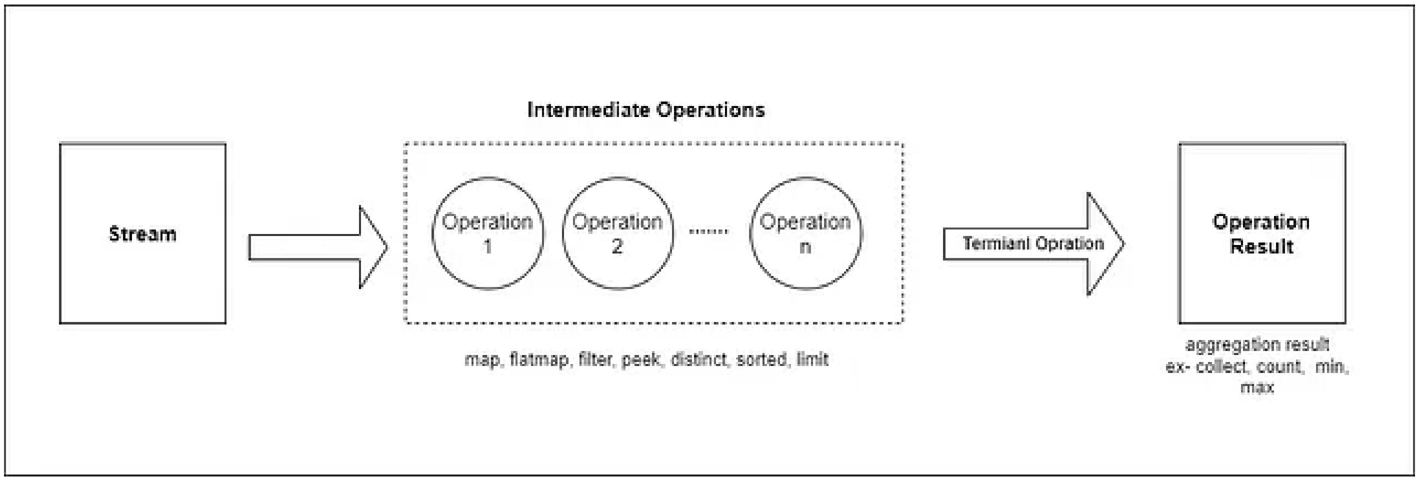
Java Stream API

Hasitha Hiran · Follow
4 min read · Feb 22

This tutorial is designed to provide you with a comprehensive overview of the various intermediate and terminal operations that can be performed using streams.

Java Streams API allows java developers to perform complex data manipulation and processing operations. Java Stream operations are mainly based on two types:



1. Intermediate operations — return a new stream that can be further processed. just transform one stream into another stream.

map, flatmap, filter, peek, distinct, sorted, limit, skip etc..

2. Terminal operations — produces end result. return a non-stream result (collection, single value).

collect, count, reduce, anyMatch, allMatch, noneMatch, findAny, findFirst, min, max, toArray etc..

Let's code.

Intermediate Operations

1. **Map —** returns a stream consisting of the results of applying the given function to the elements of this stream.

```
// returns mock List<Users> [abc@gmail,com,xyz@gmail.com,jcz@gmail.com]
List<User> users = Database.getAll();

// returns List<String> emails of users
List<String> emails = users.stream()                           // Stream<User>
                .map(user ->  user.getEmail().toUpperCase())   // Stream<String>
                .collect(Collectors.toList());

// output - [ ABC@GMAIL.COM, XYZ@GMAIL.COM, JCZ@GMAIL.COM ]
```

Here, map operation is used to create a new stream of Strings using a stream of Users.

The map() operation is used to transform the elements of a stream into another type. As an example, we can convert a stream of strings into uppercase values. This is a simple example but in real-world scenarios, we can use the map() function for more complex transformation tasks. some times we need to transform some API request data to another form using the map() operation. likewise based on the scenario, we have to use the appropriate operation.

2. Filter — filter out the results based on conditions.

```
// returns List<String> emails staring from letter "J"
List<String> emails = users.stream()                           // Stream<User>
                .map(user ->  user.getEmail().toUpperCase())   // Stream<String>
                .filter(email -> email.startsWith("J"))        // Stream<String>
                .collect(Collectors.toList());

// output - [JCZ@GMAIL.COM]
```

Above example, both map and filter operations are used. we can filter the emails that are starting from the letter 'J' from the stream of Strings.

3. Flatmap — Flattening (flat)+ mapping (map)

Flat map operation is a combination of map and flattens operations. Check my previous blog to learn more about flatmap vs map. click here.

```
String[][] letters = new String[][]{{"a", "b"}, {"c", "d"}, {"e", "f"}};

  List<String> collect = Stream.of(array)     // Stream<String[]>
          .flatMap(Stream::of)                // Stream<String>
          .filter(x -> !"c".equals(x))        // filter !c
          .collect(Collectors.toList());      // returns List<String>

 // output - [a,b,d,e,f]
```

4. Peek

Returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream. This method exists mainly to support debugging, where you want to see the elements as they flow past a certain point in a pipeline. **This operation does nothing if we do not specify a terminal operation.**

```
List<String> names = Stream.of("bob", "peter", "mickey", "james")
                .filter(e -> e.length() > 3)
                .peek(e -> System.out.println("Filtered value: " + e))
                .map(String::toUpperCase)
                .peek(e -> System.out.println("Mapped value: " + e))
                .collect(Collectors.toList());

//output

Filtered value: peter
Mapped value: PETER
Filtered value: mickey
Mapped value: MICKEY
Filtered value: james
Mapped value: JAMES
[PETER, MICKEY, JAMES]
```

5. Distinct — remove the duplicate items

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 1, 2, 3, 4);

// Displaying the distinct elements in the list
list.stream().distinct().forEach(System.out::println);

//output
[1,2,3,4]
```

Terminal Operations

1. Collect — returns the result of the reduction

Above most of the examples, we have used the collect() operation. This is used to perform a mutable reduction operation on the elements of this stream using a Collector.

2. AllMatch — returns `true` whether all elements of this stream match the provided condition.

```
List<Integer> numbers = Arrays
                .asList(1,2,3,4,5,6,7,8,9,10);

// check all number are divisible by 2
        boolean isDivibleBy2 = numbers
                .stream()
                .allMatch(num -> num % 2 == 0);

//output

false
```

3. AnyMatch — returns `true` if any of the elements in a stream matches the given predicate.

```
List<Integer> numbers = Arrays
                .asList(1,2,3,4,5,6,7,8,9,10);

// check any number is divisible by 2
        boolean isDiviBy2 = numbers
                .stream()
                .anyMatch(num -> num % 2 == 0);

//output

true
```

4. FindFirst — returns the first element in a Stream.

The return type of this operation is Optional. To learn more about Optional class click here.

```
Stream.of(86, 70, 35).findFirst()
                .ifPresent(s -> System.out.println(s));

// output
86
```

5. FindAny — returns any element from a Stream. In a non-parallel operation, **more likely will return the first element in the Stream, but there is no guarantee for this.** The return type of this operation is Optional.

```
  Stream.of(86, 70, 35).findAny()
                .ifPresent(s -> System.out.println(s));
```

Thanks for reading. **follow me** for more updates.

Java Java Stream Api

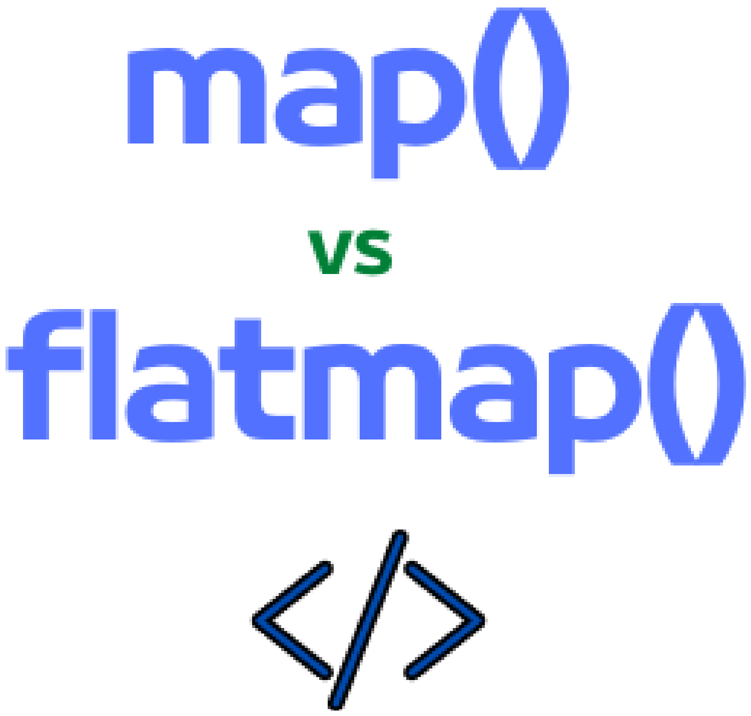
Follow

Written by Hasitha Hiran

251 Followers

Software Engineering Graduate at Sri Lanka Institute of Information Technology(SLIIT) || Associate Software Engineer at WILEY

More from Hasitha Hiran



Hasitha Hiran

Map vs flatmap — Java Streams API

Both map and flatmap methods are intermediate operations of Java 8 Stream API. Map operation produces one output value for each input...

2 min read · Feb 5

60

Hasitha Hiran

Containerized a simple java spring boot application

This blog provides a comprehensive guide to containerizing a simple Spring Boot application using Docker. The goal of this tutorial is to...

3 min read · Feb 12

79

Hasitha Hiran

Java 8—Optional Class

Java 8 introduced the Optional class as a way to handle null values more elegantly. This new addition to the Java Standard Library has...

3 min read · Feb 5

17

See all from Hasitha Hiran

Recommended from Medium

Sharad Pawar in Towards Dev

Arrays.asList() vs List.of() in java

What is the difference between Arrays.asList() and List.of() in Java?

3 min read · Aug 26

81

Ionut Anghel

REST Endpoint Best Practices Every Developer Should Know

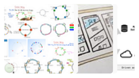
5 min read · Mar 18

470 4

Lists

It's never too late or early to start something

15 stories · 104 saves



General Coding Knowledge

20 stories · 297 saves

New_Reading_List

174 stories · 94 saves

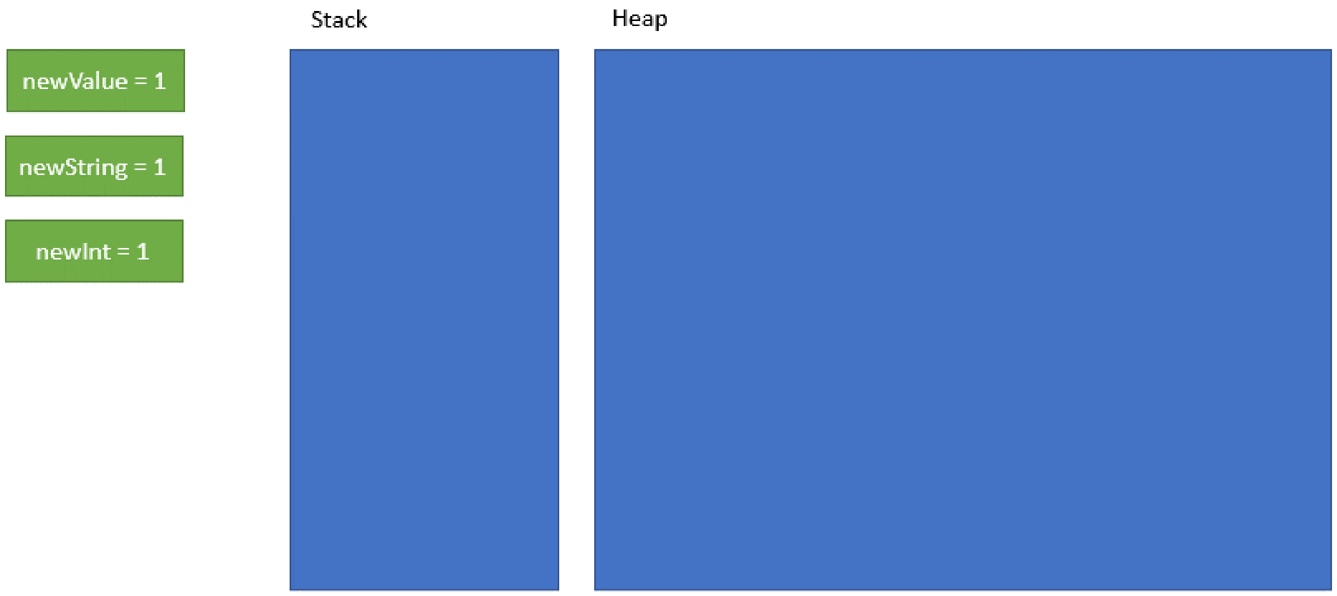
Ramit Raj

Check nulls in Java Stream

In the previous blog we got to know about how to avoid NullPointer in Java Objects in this post we learn about how we can avoid the...

1 min read · Aug 21

16 3



Berkay Haberal in Stackademic

How Java Memory Works?

Before we move on to the performence things, we need to learn that what is really going on in the background of JVM (Java Virtual Machine)...

4 min read · Jul 30

398 3

Caffeinated Developer

Mastering Java Development: Best Practices for Efficient Coding

Java continues to be a popular and powerful programming language, relied upon by developers for a wide range of applications. To excel in...

10 min read · Jul 4

167

Phạm Tuân

Top 10 Interview Questions for Java Developers

Preparing for a Java interview requires a solid understanding of fundamental concepts, algorithms, and data structures. In this blog post...

10 min read · Jul 1

65

See more recommendations