Difference between map() and flatMap() in Java?

And two coding interview questions based upon map and flatMap

JAVINPAUL

APR 04, 2024 • PAID

Hello guys, welcome back to our newsletter!

In the past few posts, I have shared popular Java interview questions like,, How ConcurrentHashMap work in Java? Why String is Immutable, why wait() and notify() is called from synchronized context, What is the difference between List, List<Object> , and CyclicBarrier vs CountDownLatch.

Today, we're diving into a fundamental aspect of Java programming: the difference between `map()` and `flatMap()` functions.

Whether you're a seasoned developer or just starting your journey with Java, understanding these functions can significantly enhance your coding prowess

This is also a popular Java interview question, so knowing this concept has double benefits.

We will also see coding questions which can be easily solved using functional programming like **given an array like [1, 2, 3,4, 5] return another array where each number is square of given number?**

I will show you the solution with and without map() but now let's start with what is map() and flatMap() in Java.

But before that a big news:

For #GumroadDay, My Java, Spring, and SQL Interview and certifications books are 'Pay What You Want, $1 minimum [Today Only]

Here is the link — https://javinpaul.gumroad.com/

1. Map() Function

Let's start with `map()`. In Java, `map()` is a method defined in the Stream interface, introduced in Java 8. This method transforms each element of a Stream into another object using a provided function.

The key point to remember is that `map()` preserves the structure of the Stream.

Here's a simple example:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> squaredNumbers = numbers.stream()

.map(n -> n * n)

.collect(Collectors.toList());

System.out.println(squaredNumbers); // Output: [1, 4, 9, 16, 25]
```

In this example, `map()` applies the lambda function `n -> n * n` to each element of the `numbers` list, producing a new list `squaredNumbers` containing the squared values.

2. FlatMap() Function

Now, let's explore `flatMap()`. Like `map()`, `flatMap()` also operates on a Stream. However, its purpose is slightly different. Instead of transforming each element into another object, `flatMap()` transforms each element into zero or more elements of a different type and then flattens the results into a single Stream.

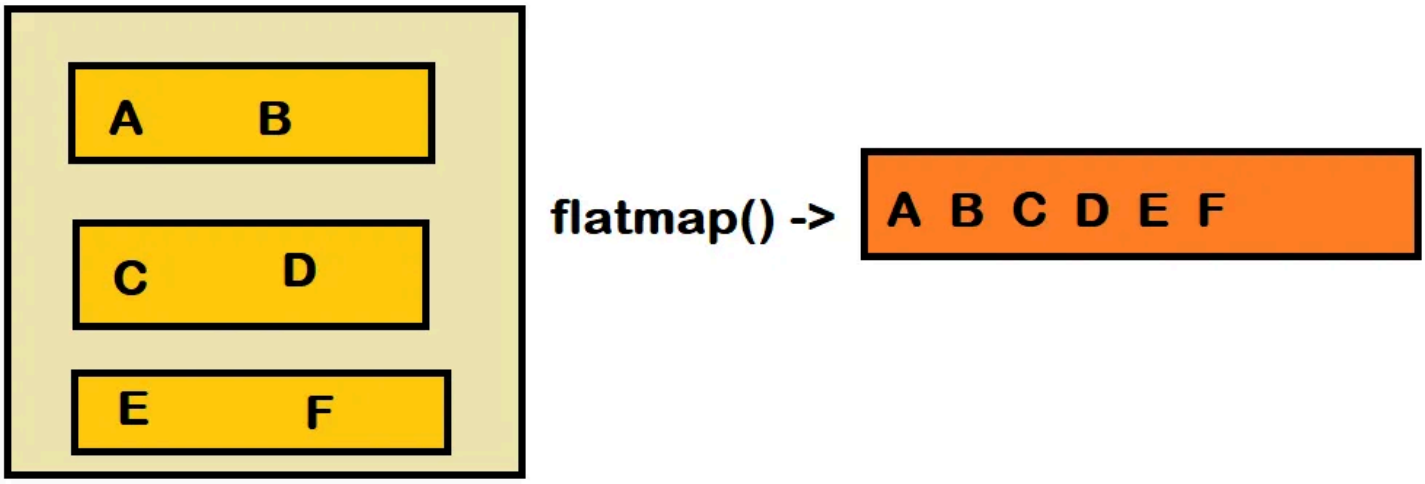
Here's an example to illustrate:

```
List<List<Integer>> nestedNumbers = Arrays.asList(

Arrays.asList(1, 2),

Arrays.asList(3, 4),

Arrays.asList(5, 6)

);

List<Integer> flatNumbers = nestedNumbers.stream()

.flatMap(Collection::stream)
```

```
.collect(Collectors.toList());

System.out.println(flatNumbers); // Output: [1, 2, 3, 4, 5, 6]
```

In this example, `flatMap()` takes a list of lists (`nestedNumbers`) and flattens it into a single list (`flatNumbers`).

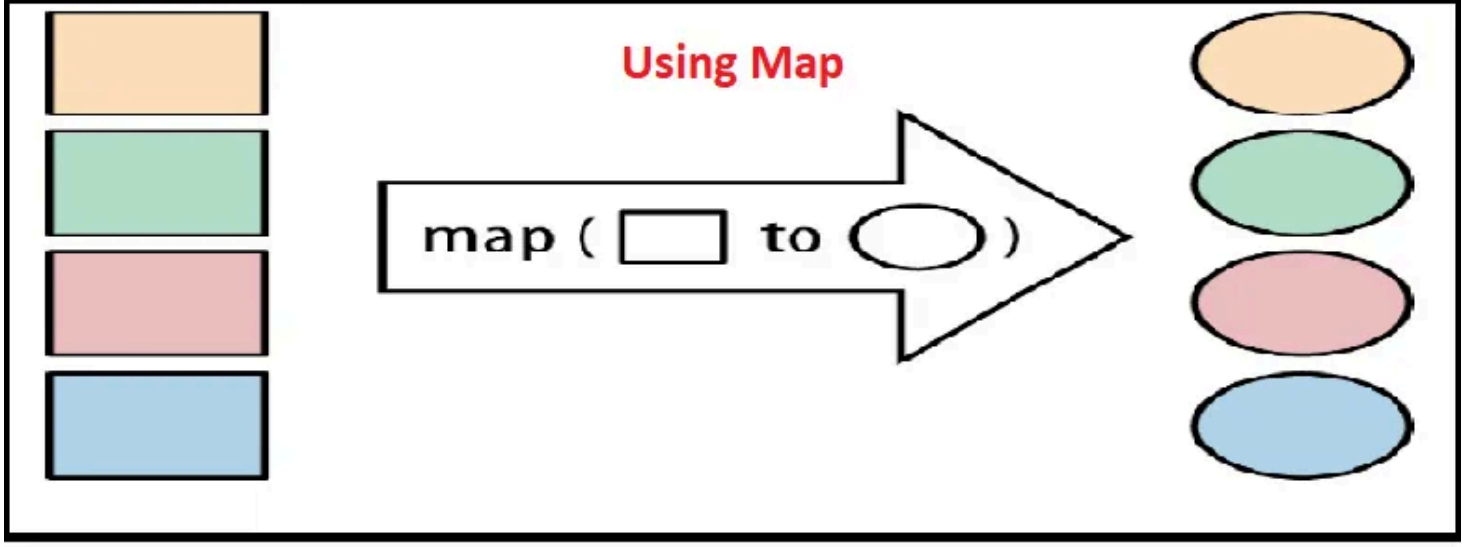
Here is a nice diagram which shows you flatMap() can be used to flatten list.

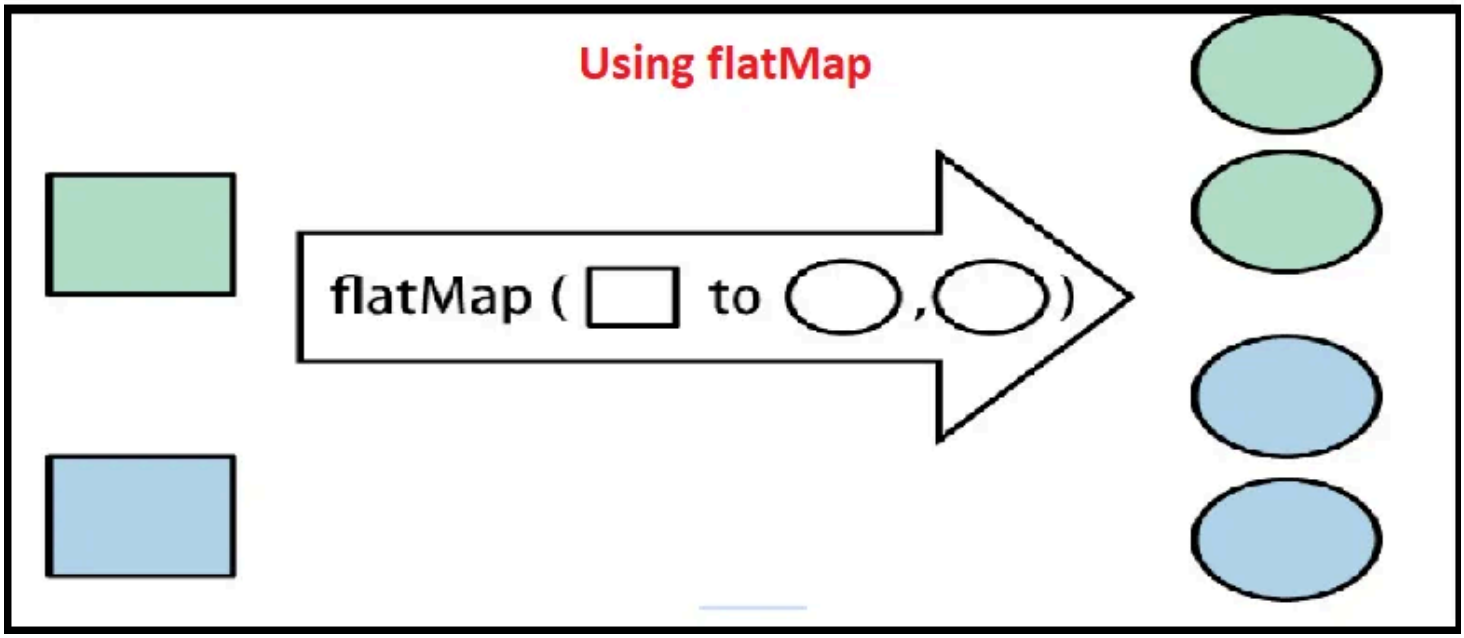


Difference between map() and flatMap() in Java?

To summarize, here are the key differences between `map()` and `flatMap()`:

- `map()` transforms each element of a Stream into another object while preserving the structure of the Stream.
- `flatMap()` transforms each element into zero or more elements of a different type and then flattens the results into a single Stream.





Coding Interview Question based upon map() and flatMap()

Now, let's see few coding questions which can be easily solved using map and flatMAp but otherwise it would be difficult. Interviewer often use this to check whether you can use the map and flatMap or not

Given an array like [1, 2, 3,4, 5] return another array where each number is square of given number?

let's see how you would solve without using functional programming:

```
public class SquareArrayWithoutMap {

public static void main(String[] args) {

int[] inputArray = {1, 2, 3, 4, 5};

int[] squaredArray = squareArray(inputArray);
```

```
for (int num : squaredArray) {

System.out.print(num + " ");

}

}

public static int[] squareArray(int[] array) {

int[] result = new int[array.length];

for (int i = 0; i < array.length; i++) {

result[i] = array[i] * array[i];

}

return result;

}

}
```

With map, the solution become just one liner like below:

```
Arrays.stream(array) .map(num -> num * num) .toArray();
```

Both approaches will give you the same output:

```
1 4 9 16 25
```

The first approach iterates through the input array and squares each element manually, whereas the second approach uses `map()` to transform each element of the input array into its square.

Now, let's change the question a bit

Given a list [1, 2, 3,4, 5] return another list containing negative of those number along with original number, I mean [1, -1, 2, -2, 3, -3, 4, -4, 5, -5]

Now the question became more interesting because the length of the list you return is different.

let's see how we can solve this using flatMap() function in Java:

```
originalList.stream() .flatMap(num -> Arrays.asList(num, -
num).stream()) .collect(Collectors.toList());
```

You can see that we have used flatMap() to flatten the list return for each number.

Here is the complete program

```
import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class NumberAndNegativeList {

public static void main(String[] args) {

List<Integer> originalList = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> resultList = createNegativeList(originalList);

System.out.println(resultList);

}

public static List<Integer> createNegativeList(List<Integer>
originalList) {

return originalList.stream()

.flatMap(num -> Arrays.asList(num, -num).stream())

.collect(Collectors.toList());

}

}
```

When you run this code, it will produce the desired output:

[1, -1, 2, -2, 3, -3, 4, -4, 5, -5]

This output contains each number from the original list along with its negative counterpart in the same order.

Conclusion

That's all about **difference between map() and flatMap() in Java**. Understanding the nuances between `map()` and `flatMap()` is crucial for writing clean, efficient code in Java. By leveraging

these functions effectively, you can streamline your code and make it more expressive.

We hope this newsletter post has clarified the distinction between `map()` and `flatMap()` in Java. Stay tuned for more insights and tips in future editions!

Happy coding!

*And, if need help with your interview preparation, you can also check my books, Grokking the Java Interview and Grokking the Spring boot Interview for better preparation, use code **friends20** to get a 20% discount also*

3 Likes · 1 Restack

← Previous

Next →

Comments

Write a comment...