Stream map vs flatMap

Stream API

In Stream API there is both map and flatMap that's available for use. They are both used for transforming a stream into another, but does it a little bit differently.

Map

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

Map has the above Java method header. It takes in a function in which it takes in the stream's type and then map it to another type. Basically, transforming each of the elements within the stream into a different element depending on the function that's provided.

Then it will return a new Stream which has the mapped element's type and each of the mapped element's value.

For example:

```
Stream.of(1, 2, 3).map(num -> num * 2).toList();
```

This will output a list of value `[2, 4, 6]`. Simple, each of the number is transformed by doubling it. A one to one mapping.

FlatMap

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)
```

FlatMap on the other hand as it's name suggests, does two operations, it first it does the mapping, in this case, the one element can be mapped to multiple elements (Another Stream if you will), then each of the Streams are flatten into a single stream.

Let's understand what a flattening operation does

Flatten Example

Let's say we have the following array:

```
{{1,2,3,4},{5,6,7},{8,9}}
```

This is a 2D array, where each of the element in the 2D array is a 1D array. If we want to flatten it it means to converted the nested array into one dimension less, unwrapping the inner nested array if you will.

```
{1,2,3,4,5,6,7,8,9}
```

This is done typically to make the array easier to work with for analytical purposes.

Back to flatMap

Now after understanding flatten, we can go back to understand how flatMap works.

It is the same idea, you have a function that takes in the input, but the function will map it to another Stream, meaning, one input can become 0 or more output, **whereas compared to map it is one to one**. If you have [1, 2, 3] the function will:

- Take the 1 and might produce Stream(1, 2)
- Take 2 and produce Stream() - Empty Stream
- Take 3 and produce Stream(3, 4)

Then at the end flatMap will take all three of the Stream. Combine it and then flatten it, so the process goes

- Stream(Stream(1, 2), Stream(), Stream(3, 4))
- Stream(1, 2, 3, 4)

That's it!

Another way to think about it

You can think about the mapping as list outputs and the flattening as unwrapping the nested array.

If you have say:

```
const words = ["hello", "world"];
const characters = words.flatMap(word => word.split(""));
```

- "hello" → split("") → `["h", "e", "l", "l", "o"]`.
- "world" → split("") → `["w", "o", "r", "l", "d"]`
- flatMap flattens `[["h", "e", "l", "l", "o"], ["w", "o", "r", "l", "d"]]` into a single array

3D example

A more complex example

```
const array = [1, 2, 3];
const result = array.flatMap(x => [[x], [x * 2]]);
console.log(result); // [[1], [2], [2], [4], [3], [6]]
```

In this case we are taking an array of numbers, taking each number and mapping it into a 2D list.

- 1 → `[[1], [2]]`
- 2 → `[[2], [4]]`
- 3 → `[[3], [6]]`
- flatMap flattens one level: `[[[1], [2]], [[2], [4]], [[3], [6]]]` becomes `[[1], [2], [2], [4], [3], [6]]`.

flatMap implementation

```
function customFlatMap(array, callback) {
  // Map each element to an array using the callback
  const mapped = array.map(callback);
  // Flatten the resulting arrays by one level
  return mapped.reduce((flat, current) => flat.concat(current), []);
}
```

An example flatMap implementation can be done above. The callback is used to do the function, it maps the element to an array.

Then `reduce` is used to flatten the array into a list that is one nested level less.

```
const words = ["hello", "world"];
const characters = customFlatMap(words, word => word.split(""));
console.log(characters); // ["h", "e", "l", "l", "o", "w", "o", "r", "l", "d"]
```

- Callback: word => `word.split("")` returns `["h", "e", "l", "l", "o"]` for "hello", `["w", "o", "r", "l", "d"]` for "world"
- Map: Produces `[["h", "e", "l", "l", "o"], ["w", "o", "r", "l", "d"]]`

- Reducing: Merges into `["h", "e", "l", "l", "o", "w", "o", "r", "l", "d"]` - You are just basically taking each of the array and concatenating the element into an aggregating list (which starts off as empty)

Revision #1
Created 2025-05-06 03:10:23 UTC by Tamarine
Updated 2025-05-06 04:02:36 UTC by Tamarine