

Theory: Collectors

🕒 19 minutes 0 / 5 problems solved

Skip this topic

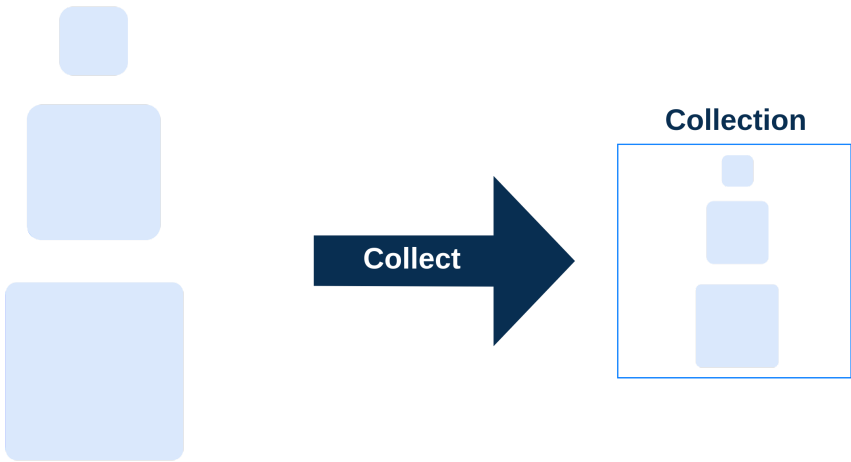
Start practicing

543 users solved this topic. Latest completion was about 4 hours ago.

So far we know how to produce a single value from a stream of elements by using `reduce` operation. However, collecting stream elements into a collection such as a `List` or a `Set` is a much more popular scenario than reducing them to a single value. For that, Java Stream API provides a terminal operation called `collect`. In a combination with a utility `stream.Collectors` class that contains a lot of useful reduction operations, `collect` allows us to easily produce collections from streams as well as single values like the `reduce` operations do.

§1. Producing collections

The `collect` is a terminal reduction operation that can accept an object of the `Collector` type. But instead of focusing on the `Collector`, let's consider the `Collectors` class more closely. It is important that the `Collectors` class contains static methods that return the `Collector` and implement functionality for accumulating stream elements into a collection, summarizing them, repacking to a single string, etc.



Note that the `collect` is a terminal operation, which means that it begins all evaluations with the stream and produces a final result.

To be more concrete, let's consider an example where the `Account` class is given:

```
1 public class Account {
2     private long balance;
3     private String number;
4
5     // getters and setters
6 }
```

We want to produce a list of accounts from the stream of accounts `Stream<Account> accountStream`. To do so, we can accumulate stream elements to the list using `Collectors.toList` method:

```
1 List<Account> accounts = accountStream.collect(Collectors.toList());
```

As you can see, the `Collectors.toList` method did all the work for us. Similarly to producing a `List` from a stream, we can produce a `Set`. Again, we can delegate that responsibility to the `Collectors` class and use `Collectors.toSet` method:

```
1 Set<Account> accounts = accountStream.collect(Collectors.toSet());
```

If you need more control over producing collections and want to accumulate stream elements to the particular collection that is not a `List` or a `Set`, than `Collectors.toCollection` method may come in handy:

Current topic:

Collectors ...

Topic depends on:

✗ Reduction methods ...

Topic is required for:

Grouping collectors ...

Table of contents:

- 1 Collectors
- §1. Producing collections
- §2. Producing values
- §3. Conclusion
- Feedback & Comments

```
1 |  
LinkedList<Account> accounts = accountStream.collect(Collectors.toCollection(LinkedList::new));
```

The `Collections.toCollection` method accepts a function that generates a new collection of a specified type. In the example above, we've accumulated a stream of account numbers to the `LinkedList<Account>` by providing a reference to its default constructor.

§2. Producing values

Similarly to the `reduce` operation, `collect` is able to accumulate stream elements into a single value. Here you can see some `Collectors` methods that produce a single value:

- `summingInt`, `summingLong`, `summingDouble`;
- `averagingInt`, `averagingLong`, `averagingDouble`;
- `maxBy`, `minBy`;
- `counting`.

The names of the methods are quite self-explanatory regarding their purpose. We'll employ one in the example below.

Note that you can make your code shorter and more clear by using static import of necessary collectors such as `import static java.util.stream.Collectors.averagingLong`;

Now let's summarize balances on the accounts. We can use `summingLong` method for that:

```
1 | long summary = accounts.stream()  
2 |     .collect(summingLong(Account::getBalance));
```

Also, we can calculate the mean value:

```
1 | double average = accounts.stream()  
2 |     .collect(averagingLong(Account::getBalance));
```

Note that all averaging collectors (`averagingLong`, `averagingInt`, `averagingDouble`) return a `double` value.

If you need to perform more specific calculations, you can use `Collectors.reducing` method. Similarly to the `reduce` operation, `Collectors.reducing` method implementations can accept an accumulator function or the identity value together with an accumulator. However, there is one additional implementation that accepts identity, *mapper*, and an accumulation function.

It is notable that the mapper is a mapping function that is applied to stream elements, while the reducing accumulator function reduces the mapped values of a stream.

Let's consider an example:

```
1 | String meganumber = accountStream.collect(Collectors.reducing("",  
2 |     account -> account.getNumber(),  
3 |     (numbers, number) -> numbers.concat(number)  
4 | ));
```

The code above maps each account to its number and concatenates all account numbers into one single number using a reducing collector.

§3. Conclusion

The `collect` is a terminal operation that allows us to accumulate stream elements to a collection or a single value. The `collect` method accepts the object of the `Collector` type. Instead of implementing the `Collector`, we can

use the `Collectors` class that contains useful methods that return a `Collector` with already implemented logic. By using `Collectors` we can accumulate stream elements into a `List` or a `Set` by using `toList` and `toSet` methods respectively. If we need to produce some other collection we can use the method called `toCollection`. Besides producing collections, the `collect` operation can be used for calculating such values as the average, summarized, maximum, minimum, etc.

 Report a typo

67 users liked this theory. 1 didn't like it. What about you?



Start practicing

[Comments \(0\)](#)

[Hints \(0\)](#)

[Useful links \(0\)](#)

[Show discussion](#)