Handling Map Values with Lambda Expressions

Consuming the Content of a Map

The `Map` interface has a `forEach()` method that works in the same way as the `forEach()` method on the `Iterable` interface. The difference is that this `forEach()` method takes a `BiConsumer` as an argument instead of a simple `Consumer`.

Let us create a simple map and print out its content.

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "one");
map.put(2, "two");
map.put(3, "three");

map.forEach((key, value) -> System.out.println(key + " :: " + value));
```

This code produces the following result:

```
1 :: one
2 :: two
3 :: three
```

Replacing Values

The `Map` interface gives you three methods to replace a value bound to a key with another value.

The first one is `replace(key, value)`, which replaces the existing value with the new one, blindly. This is the equivalent of a put-if-present operation. This method returns the value that was removed from your map.

If you need finer control, then you can use an overload of this method, which takes the existing value as an argument: `replace(key, existingValue, newValue)`. In this case, the existing value is

replaced only if it matches the new value. This method returns `true` if the replacement occurred.

The `Map` interface has also a method to replace all the values of your map using a `BiFunction`. This `BiFunction` is a remapping function, which takes the key and the value as arguments, and returns a new value, which will replace the existing value. A call to this method iterates internally on all the key/value pairs of your map.

The following example shows how you can use this `replaceAll()` method:

```
Map<Integer, String> map = new HashMap<>();

map.put(1, "one");
map.put(2, "two");
map.put(3, "three");

map.replaceAll((key, value) -> value.toUpperCase());
map.forEach((key, value) -> System.out.println(key + " :: " + value));
```

Running this code produces the following result:

```
1 :: ONE
2 :: TWO
3 :: THREE
```

Computing Values

The `Map` interface gives you a third pattern to add key-value pairs to a map or modify a map's existing values in the form of three methods: `compute()`, `computeIfPresent()`, and `computeIfAbsent()`.

These three methods take the following arguments:

- the key on which the computation is made
- the value bound to that key, in the case of `compute()` and `computeIfPresent()`
- a `BiFunction` that acts as a remapping function, or a mapping function in the case of `computeIfAbsent()`.

In the case of `compute()`, the remapping bi-function is called with two arguments. The first one is the key, and the second one is the existing value if there is one, or `null` if there is none. Your remapping bifunction can be called with a null value.

For `computeIfPresent()`, the remapping function is called if there is a value bound to that key and if it is not null. If the key is bound to a null value, then the remapping function is not called. Your remapping function cannot be called with a null value.

For `computeIfAbsent()`, beacause there is no value bound to that key, the remapping function is in fact a simple `Function` that takes the key as an argument. This function is called if the key is not present in the map or if it is bound to a null value.

In all cases, if your bifunction (or function) returns a null value, then the key is removed from the map: no mapping is created for that key. No key/value pair with a null value can be put in the map using one of these three methods.

In all cases, the value returned is the new value bound to that key in the map or null if the remapping function returned null. It is worth pointing out that this semantic is different from the `put()` methods. The `put()` methods return the previous value, whereas the `compute()` methods return the new value.

A very interesting use case for the `computeIfAbsent()` method is the creation of maps with lists as values. Suppose you have the following list of strings: `[one two three four five six seven]`. You need to create a map, where the keys are the length of the words of that list, and the values are the lists of these words. What you need to create is the following map:

```
3 :: [one, two, six]
4 :: [four, five]
5 :: [three, seven]
```

Without the `compute()` methods, you would probably write this:

```
List<String> strings = List.of("one", "two", "three", "four", "five", "six", "seven");
Map<Integer, List<String>> map = new HashMap<>();
for (String word: strings) {
    int length = word.length();
    if (!map.containsKey(length)) {
        map.put(length, new ArrayList<>());
    }
    map.get(length).add(word);
}

map.forEach((key, value) -> System.out.println(key + " :: " + value));
```

Running this code produces the expected result:

```
3 :: [one, two, six]
4 :: [four, five]
5 :: [three, seven]
```

By the way, you could use a `putIfAbsent()` to simplify this for loop:

```
for (String word: strings) {
    int length = word.length();
    map.putIfAbsent(length, new ArrayList<>());
    map.get(length).add(word);
}
```

But using `computeIfAbsent()` can make this code even better:

```
for (String word: strings) {
    int length = word.length();
    map.computeIfAbsent(length, key -> new ArrayList<>())
       .add(word);
}
```

How does this code work?

- If the key is not in the map, then the mapping function is called, which creates an empty list. This list is returned by the `computeIfAbsent()` method. This is the empty list in which the code adds `word`.
- If the key is in the map, the mapping function is not called, and the current value bound to that key is returned. This is the partially filled list in which you need to add `word`.

This code is much more efficient than the `putIfAbsent()` one, mostly because in that case, the empty list is created only if needed. The `putIfAbsent()` call requires an existing empty list, which is used only if the key is not in the map. In cases where the object you add to the map has to be created on demand, then using `computeIfAbsent()` should be preferred over `putIfAbsent()`.

Merging Values

The `computeIfAbsent()` pattern works well if your map has values that are aggregations of other values. But there is a restriction on the structure that is supporting this aggregation: it has to be mutable. This is the case for `ArrayList`, and this is what the code you wrote does: it adds your values to an `ArrayList`.

Instead of creating lists of words, suppose you need to create a concatenation of words. The `String` class is seen here as an aggregation of other strings, but it is not a mutable container: you cannot use the `computeIfAbsent()` pattern to do that.

This is where the `merge()` pattern comes to the rescue. The `merge()` method takes three arguments:

- a key
- a value, that you need to bind to that key
- a remapping `BiFunction`.

If the key is not in the map or bound to a null value, then the value is bound to that key. The remapping function is not called in this case.

On the contrary, if the key is already bound to a non-null value, then the remapping function is called with the existing value, and the new value passed as an argument. If this remapping function returns

null, then the key is removed from the map. The value it produces is bound to that key otherwise.

You can see this `merge()` pattern in action on the following example:

```
List<String> strings = List.of("one", "two", "three", "four", "five", "six", "seven");
Map<Integer, String> map = new HashMap<>();
for (String word: strings) {
    int length = word.length();
    map.merge(length, word,
              (existingValue, newWord) -> existingValue + ", " + newWord);
}

map.forEach((key, value) -> System.out.println(key + " :: " + value));
```

In this case, if the `length` key is not in the map, then the `merge()` call just adds it and binds it to `word`. On the other hand, if the `length` key is already in the map, then the bifunction is called with the existing value and `word`. The result of the bifunction then replaces the current value.

Running this code produces the following result:

```
3 :: one, two, six
4 :: four, five
5 :: three, seven
```

In both patterns, `computeIfAbsent()` and `merge()`, you may be wondering why the lambda created takes an argument that is always available in the context of this lambda, and that could be captured from that context. The answer is: you should favor non-capturing lambdas over capturing ones, for performance reasons.

Last update: *September 14, 2021*

Managing the Content of a Map ← **Handling Map Values with Lambda Expressions** → Keeping Keys Sorted with SortedMap and NavigableMap

Home > Tutorials > The Collections Framework > Handling Map Values with Lambda Expressions