Managing the Content of a Map

Adding a Key Value Pair to a Map

You can simply add a key/value pair in a map with `put(key, value)`. If the key is not already present in the map, then the key/value pair is simply added to the map. If it is, then the existing value is replaced with the new one.

In both cases, the `put()` method returns the existing value currently bound to the key. This means that if this a new key, a call to `put()` will return null.

Java SE 8 introduces the `putIfAbsent()` method. This method can also add a key/value pair to the map, only if the key is not already present *and* not associated to a null value. This may seem a bit confusing at first, but `putIfAbsent()` will replace a null value with the new value provided.

This method is very handy if you need to get rid of faulty null values in your map. For instance the following code will fail with a `NullPointerException` because you cannot auto-unbox a null `Integer` to an `int` value.

```
1  Map<String, Integer> map = new HashMap<>();
2
3  map.put("one", 1);
4  map.put("two", null);
5  map.put("three", 3);
6
```

```
map.put("four", null);
map.put("five", 5);

for (int value : map.values()) {
    System.out.println("value = " + value);
}
```

If you take a close look at this code, you will see that `map.values()` is a `Collection<Integer>`. So iterating on this collection produces instances of `Integer`. Because you declared `value` as an `int`, the compiler will auto-unbox this `Integer` to an `int` value. This mechanism fails with a `NullPointerException` if the instance of `Integer` is null.

You may fix this map with the following code, which replaces the faulty null values with a default value, `-1`, that will not generate any `NullPointerException` anymore.

```
for (String key : map.keySet()) {
    map.putIfAbsent(key, -1);
}
```

Running the previous code will print the following. As you can see this map does not contain any null values anymore:

```
value = -1
value = 1
value = -1
value = 3
value = 5
```

Getting a Value From a Key

You can get a value bound to a given key simply by calling the `get(key)` method.

Java SE 8 introduced the `getOrDefault()` method that takes a key and a default value which is returned if the key is not in the map.

Let us see this method in action in an example:

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

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

List<String> values = new ArrayList<>();
for (int key = 0; key < 5; key++) {
    values.add(map.getOrDefault(key,"UNDEFINED"));
}

System.out.println("values = " + values);
```

Or, if you are familiar with streams (which are covered later in this tutorial):

```
List<String> values =
    IntStream.range(0, 5)
        .mapToObj(key -> map.getOrDefault(key, "UNDEFINED"))
        .collect(Collectors.toList());

System.out.println("values = " + values);
```

Both codes print out the same result:

```
values = [UNDEFINED, one, two, three, UNDEFINED]
```

Removing a Key from a Map

Removing a key/value pair is made by calling the `remove(key)` method. This method returns the value that was bound to that key, so it may return `null`.

It may be risky to blindly remove a key/value pair from a map if you do not know the value that is bound to that key. Thus, Java SE 8 added an overload that takes a value as a second argument. This time, the key/value pair is removed only if it fully matches the key/value pair in the map.

This `remove(key, value)` method returns a boolean value, `true` if the key/value pair was removed from the map.

Checking for the Presence of a Key or a Value

You have two methods to check for the presence of a given key or a given value: `containsKey(key)` and `containsValue(value)`. Both methods return `true` if the map contains the given key or value.

Checking for the Content of a Map

The `Map` interface also brings methods that look like the ones you have in the `Collection` interface. These methods are self-explanatory: `isEmpty()` returns `true` for empty maps, `size()` returns the number of key/value pairs, and `clear()` removes all the content of the map.

There is also a method to add the content of a given map to the current map: `putAll(otherMap)`. If some keys are present in both maps, then the values of `otherMap` will erase those of this map.

Getting a View on the Keys, the Values or the Entries of a Map

You can also get different sets defined on a map.

- `keySet()`: returns an instance of `Set`, containing the keys defined in the map
- `entrySet()`: returns an instance of `Set<Map.Entry>`, containing the key/value pairs contained in the map
- `values()`: returns an instance of `Collection`, containing the values present in the map.

The following examples show these three methods in action:

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

map.put(1, "one");
map.put(2, "two");
map.put(3, "three");
map.put(4, "four");
map.put(5, "five");
map.put(6, "six");

Set<Integer> keys = map.keySet();
System.out.println("keys = " + keys);

Collection<String> values = map.values();
System.out.println("values = " + values);

Set<Map.Entry<Integer, String>> entries = map.entrySet();
System.out.println("entries = " + entries);
```

Running this code produces the following result:

```
keys = [1, 2, 3, 4, 5]
values = [one, two, three, four, five]
entries = [1=one, 2=two, 3=three, 4=four, 5=five]
```

These sets are *views* backed by the current map. Any change made to the map is reflected in those views.

Removing a Key From the Set of Keys

Modifying one of these sets will also be reflected in the map: for instance, removing a key from the set returned by a call to `keySet()` removes the corresponding key/value pair from the map.

For instance, you can run this code on the previous map:

```
keys.remove(3);
entries.forEach(System.out::println);
```

It will produce the following resut:

```
1=one
2=two
4=four
5=five
6=six
```

Removing a Value From the Collection of Values

Removing a value is not as simple because a value can be found more than once in a map. In that case, removing a value from the collection of values just removes the first matching key/value pair.

You can see that on the following example.

```
Map<Integer, String> map =
    Map.ofEntries(
        Map.entry(1, "one"),
        Map.entry(2, "two"),
        Map.entry(3, "three"),
        Map.entry(4, "three")
    );
System.out.println("map before = " + map);
map = new HashMap<>(map);
map.values().remove("three");

```

```
    System.out.println("map after = " + map);
```

Running this code will produce the following result.

```
map before = {1=one, 2=two, 3=three, 4=three}
map after  = {1=one, 2=two, 4=three}
```

As you can see, only the first key/value pair has been removed in this example. You need to be careful in this case because if the implementation you chose is a `HashMap`, you cannot tell in advance what key/value pair will be found.

You do not have access to all the operations on these sets though. For instance, you cannot add an element to the set of keys, or to the collection of values. If you try to do that, you will get an `UnsupportedOperationException`.

If what you need is to iterate over the key/value pairs of a map, then your best choice is to iterate directly on the set of key/value pairs. It is much more efficient to do that, rather than iterating on the set of keys, and getting the corresponding value. The best pattern you can use is the following:

```
for (Map.Entry<Integer, String> entry : map.entrySet()) {
    System.out.println("entry = " + entry);
}
```

Last update: *September 14, 2021*

Using Maps to Store Key Value Pairs ← **Managing the Content of a Map** → Handling Map Values with Lambda Expressions

Home > Tutorials > The Collections Framework > Managing the Content of a Map