Till now, we have discussed most of the operations that are supported in a **HashMap**. We have also looked at some of the operations that were added in Java 8, but we have not covered all of them. In this lesson, we will look at a few more operations that were added with Java 8.

The `compute()` method

The `compute(Key, BiFunction)` method allows us to update a value in **HashMap**. This method tries to compute a mapping for the specified key and its current mapped value (or null if no current mapping is found). This method is used to atomically update a value for a given key in **HashMap**.

1. If the remapping function passed in compute returns `null`, the mapping is removed from the Map (or remains absent if initially absent).
2. If the remapping function throws an exception, the exception is rethrown, and the current mapping is left unchanged.

The syntax of this method is:

```
compute(K key,
          BiFunction<? super K, ? super V, ? extends V> remappingFunction)
```

Let's say we have a **HashMap** in which the key is a String, and the value is an Integer. We need to increment the value for a given key by one, and if the key is not present, we need to insert the key with the default value of 10. We can create a lambda expression and pass it to the `compute()` method.

```
import java.util.HashMap;
import java.util.Map;

public class HashMapDemo {

    public static void main(String args[]) {

        Map<String, Integer> map = new HashMap<>();
        map.put("India", 5);
        map.put("USA", 3);
        map.put("China", 5);
        map.put("Russia", 6);

        // This will increment the value of India by 1 as it is present in the Map
        map.compute("India", (k, v) -> v == null ? 10 : v + 1);

        // This will insert Vietnam in the Map with default value of 10.
        map.compute("Vietnam", (k, v) -> v == null ? 10 : v + 1);

        System.out.println(map);

    }
}
```

Run Save Reset

The `computeIfAbsent()` method

The `computeIfAbsent(Key, Function)` method of the **HashMap** class is used to compute the value for a given key using the given mapping function and enter that computed value in **HashMap**; otherwise, it's `null`. Please note that the `computeIfAbsent()` will work only if the key is not present or if the key is present with a `null` value.

The syntax of this method is:

```
public V
       computeIfAbsent(K key,
            Function<? super K, ? extends V> remappingFunction)
```

Let's say we have a **HashMap** in which the key is a String and the value is the length of the String. We can use the `computeIfAbsent()` method to insert new pairs in the Map. We will pass a lambda function that returns the length of the key.

```
import java.util.HashMap;
import java.util.Map;

public class HashMapDemo {

    public static void main(String args[]) {

        Map<String, Integer> map = new HashMap<>();
        map.put("India", 5);
        map.put("USA", 3);
        map.put("China", 5);
        map.put("Russia", 6);

        map.computeIfAbsent("Vietnam", k -> k.length());

        System.out.println(map);

    }
}
```

Run Save Reset

The `computeIfPresent()` method

The `computeIfPresent(Key, BiFunction)` method of the **HashMap** class allows you to compute the value of mapping for a specified key if the key is already associated with a value or is mapped to null.

1. If the mapping function of this method returns `null`, the mapping is removed.
2. If the remapping function throws an exception, the exception is rethrown, and the mapping is left unchanged.

The syntax of this method is:

```
public Object computeIfPresent(Object key,
                BiFunction remappingFunction)
```

Let's say we have a **HashMap** in which the key is a String and the value is some Integer. Then we can use `computeIfPresent()` method to update the value in the Map. We will pass a lambda function that will calculate a value if the key is already present in the Map.

```
import java.util.HashMap;
import java.util.Map;

public class HashMapDemo {

    public static void main(String args[]) {

        Map<String, Integer> map = new HashMap<>();
        map.put("India", 5);
        map.put("USA", 3);
        map.put("China", 5);
        map.put("Russia", 6);

        // This will increment the value of India by 1 as it is present in the Map
        map.computeIfPresent("India", (k, v) -> v == null ? 10 : v + 1);

        // This will not insert Vietnam in the Map.
        map.computeIfPresent("Vietnam", (k, v) -> v == null ? 10 : v + 1);

        System.out.println(map);

    }
}
```

Run Save Reset

The `merge()` method

The merge function can be used to merge two Maps. This method takes three arguments:

1. `key` - The key that needs to be merged.
2. `value` - The value that needs to be inserted in case the key is not present.
3. `remappingFunction` - This is a BiFunction that is used to update the value if the key is present.

We will begin to understand the working of this method using an example. Let's say we have two Maps in which the key is the name of a person and the value is the amount of money that person has borrowed from us. It is possible that a person is present in both the Maps. So, we need to merge these Maps to find the total amount that each person has borrowed from us.

The syntax of this method is:

```
merge(K key, V value,
            BiFunction remappingFunction)
```

```
import java.util.HashMap;
import java.util.Map;

public class HashMapDemo {

    public static void main(String args[]) {

        Map<String, Integer> map1 = new HashMap<>();
        map1.put("Jay", 5000);
        map1.put("Rahul", 3000);
        map1.put("Nidhi", 4500);
        map1.put("Amol", 60000);

        Map<String, Integer> map2 = new HashMap<>();
        map2.put("Jay", 7000);
        map2.put("Rahul", 4500);
        map2.put("Nidhi", 1200);
        map2.put("Sauray", 25000);

        map1.forEach((key,value) -> map2.merge(key, value, (v1, v2) -> v1 + v2));

        System.out.println(map2);

    }
}
```

Run Save Reset