Choosing Immutable Types for Your Key

Avoiding the Use of Mutable Keys

Using mutable key is an antipattern, and you should definitely avoid doing that. The side effects you may get if you do are terrible: you may end up making the content of your map unreachable.

It is quite easy to set up an example to show that. Here is a `Key` class, which is just a mutable wrapper on an `String`. Note that the `equals()` and `hashCode()` methods have been overridden by a code that your IDE could generate.

```
//
// !!!!! This an example of an antipattern !!!!!!
// !!! do not do this in your production code !!!
//
class Key {
    private String key;

    public Key(String key) {
        this.key = key;
    }

    public String getKey() {
        return key;
    }

    public void setKey(String key) {
```

```
        this.key = key;
    }

    @Override
    public String toString() {
        return key;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Key key = (Key) o;
        return Objects.equals(this.key, key.key);
    }

    @Override
    public int hashCode() {
        return key.hashCode();
    }
}
```

You can use this wrapper to create a map in which to put key-value pairs in.

```
Key one = new Key("1");
Key two = new Key("2");

Map<Key, String> map = new HashMap<>();
map.put(one, "one");
map.put(two, "two");

System.out.println("map.get(one) = " + map.get(one));
System.out.println("map.get(two) = " + map.get(two));
```

So far this code is OK and prints out the following:

```
map.get(one) = one
map.get(two) = two
```

What will happen if someone mutates your key? Well, it really depends on the mutation. You can try the ones in the following example, and see what is happening when you try to get your values back.

In the following case, you are mutating one of the existing key with a new value that does not correspond to an already existing key.

```
one.setKey("5");

System.out.println("map.get(one) = " + map.get(one));
System.out.println("map.get(two) = " + map.get(two));
System.out.println("map.get(new Key(1)) = " + map.get(new Key("1")));
System.out.println("map.get(new Key(2)) = " + map.get(new Key("2")));
System.out.println("map.get(new Key(5)) = " + map.get(new Key("5")));
```

The result is the following. You cannot get the value from the key anymore, even if you use the same object. And getting the value from a key that is holding the original value also fails. This key-value pair is lost.

```
map.get(one) = null
map.get(two) = two
map.get(new Key(1)) = null
map.get(new Key(2)) = two
map.get(new Key(5)) = null
```

If you mutate your key with a value that is used for another, existing key, the result is different.

```
one.setKey("2");

System.out.println("map.get(one) = " + map.get(one));
System.out.println("map.get(two) = " + map.get(two));
System.out.println("map.get(new Key(1)) = " + map.get(new Key("1")));
System.out.println("map.get(new Key(2)) = " + map.get(new Key("2")));
```

The result is now the following. Getting the value bound to the mutated key returns the value bound to the other key. And, as in the previous example, you cannot get the value bound to the mutated key anymore.

```
map.get(one) = two
map.get(two) = two
map.get(new Key(1)) = null
map.get(new Key(2)) = two
```

As you can see, even on a very simple example, things can go terribly wrong: the first key cannot be used to access the right value anymore, and you can lose values in the process.

In a nutshell: if you really cannot avoid using mutable keys, do not mutate them. But your best choice is to use unmodifiable keys.

Diving in the Structure of HashSet

You may be wondering why would it be interesting to talk about the `HashSet` class in this section? Well, it turns out that the `HashSet` class is in fact built on an internal `HashMap`. So the two classes share some common features.

Here is the code of the `add(element)` of the `HashSet` class:

```
private transient HashMap<E,Object> map;
private static final Object PRESENT = new Object();

public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}
```

What you can see is that in fact, a hashset stores your object in a hashmap (the `transient` keyword is not relevant). Your objects are the keys of this hashmap, and the value is just a placeholder, an object with no significance.

The important point to remember here is that if you mutate your objects after you have added them to a set, you may come across weird bugs in your application, that will be hard to fix.

Let us take the previous example again, with the mutable `Key` class. This time, you are going to add instances of this class to a set.

```
Key one = new Key("1");
Key two = new Key("2");

```

```
Set<Key> set = new HashSet<>();
set.add(one);
set.add(two);

System.out.println("set = " + set);

// You should never mutate an object once it has been added to a Set!
one.setKey("3");
System.out.println("set.contains(one) = " + set.contains(one));
boolean addedOne = set.add(one);
System.out.println("addedOne = " + addedOne);
System.out.println("set = " + set);
```

Running this code produces the following result:

```
set = [1, 2]
set.contains(one) = false
addedOne = true
set = [3, 2, 3]
```

You can see that in fact the first element and the last element of the set are the same:

```
List<Key> list = new ArrayList<>(set);
Key key0 = list.get(0);
Key key2 = list.get(2);

System.out.println("key0 = " + key0);
System.out.println("key2 = " + key2);
System.out.println("key0 == key2 ? " + (key0 == key2));
```

If you run this last piece of code, you will get the following result:

```
key0 = 3
key2 = 3
key0 == key2 ? true
```

In this example, you saw that mutating an object once it has been added to a set can lead to having the same object more than once in this set. Simply said, do not do that!

Last update: *September 14, 2021*

[Keeping Keys Sorted with SortedMap and NavigableMap](#) ← **Choosing Immutable Types for Your Key** → That's the end of the series!