Top 9 questions about Java Maps

In general, **Map** is a data structure consisting of a set of *key-value* pairs, and each key can only appears once in the map. This post summarizes Top 9 FAQ of how to use Java **Map** and its implemented classes. For sake of simplicity, I will use generics in examples. Therefore, I will just write `Map` instead of specific `Map`. But you can always assume that both the **K** and **V** are comparable, which means `K extends Comparable` and `V extends Comparable`.

0. Convert a Map to List

In Java, **Map** interface provides three collection views: key set, value set, and key-value set. All of them can be converted to **List** by using a constructor or *addAll()* method. The following snippet of code shows how to construct an **ArrayList** from a map.

```
// key list
List keyList = new ArrayList(map.keySet());
// value list
List valueList = new ArrayList(map.valueSet());
// key-value list
List entryList = new ArrayList(map.entrySet());
```

1. Iterate over each Entry in a Map

Iterating over every pair of key-value is the most basic operation to traverse a map. In Java, such pair is stored in the map entry called **Map.Entry**. **Map**.*entrySet()* returns a key-value set, therefore the most efficient way of going through every entry of a map is

```
for(Entry entry: map.entrySet()) {
  // get key
  K key = entry.getKey();
  // get value
  V value = entry.getValue();
}
```

Iterator can also be used, especially before JDK 1.5

```
Iterator itr = map.entrySet().iterator();
while(itr.hasNext()) {
  Entry entry = itr.next();
  // get key
  K key = entry.getKey();
  // get value
  V value = entry.getValue();
}
```

2. Sort a Map on the keys

Sorting a map on the keys is another frequent operation. One way is to put **Map.Entry** into a list, and sort it using a comparator that sorts the value.

```
List list = new ArrayList(map.entrySet());
Collections.sort(list, new Comparator() {

  @Override
  public int compare(Entry e1, Entry e2) {
    return e1.getKey().compareTo(e2.getKey());
  }

});
```

The other way is to use **SortedMap**, which further provides a total ordering on its keys. Therefore all keys must either implement **Comparable** or be accepted by the comparator.
One implementing class of **SortedMap** is **TreeMap**. Its constructor can accept a comparator. The following code shows how to transform a general map to a sorted map.

```
SortedMap sortedMap = new TreeMap(new Comparator() {

  @Override
  public int compare(K k1, K k2) {
    return k1.compareTo(k2);
  }

});
sortedMap.putAll(map);
```

3. Sort a Map on the values

Putting the map into a list and sorting it works on this case too, but we need to compare**Entry**.*getValue()* this time. The code below is almost same as before.

```
List list = new ArrayList(map.entrySet());
Collections.sort(list, new Comparator() {

  @Override
  public int compare(Entry e1, Entry e2) {
    return e1.getValue().compareTo(e2.getValue());
  }

});
```

We can still use a sorted map for this question, but only if the values are unique too. Under such condition, you can reverse the key=value pair to value=key. This solution has very strong limitation therefore is not really recommended by me.

4. Initialize a static/immutable Map

When you expect a map to remain constant, it's a good practice to copy it into an immutable map. Such defensive programming techniques will help you create not only safe for use but also safe for thread maps.

To initialize a static/immutable map, we can use a static initializer (like below). The problem of this code is that, although **map** is declared as **static final**, we can still operate it after initialization, like `Test.map.put(3,"three");`. Therefore it is not really immutable. To create an immutable map using a static initializer, we need an extra anonymous class and copy it into a unmodifiable map at the last step of initialization. Please see the second piece of code. Then, an**UnsupportedOperationException** will be thrown if you run `Test.map.put(3,"three");`.

```
public class Test {

  private static final Map map;
  static {
    map = new HashMap();
    map.put(1, "one");
    map.put(2, "two");
  }
}
```

```
public class Test {

  private static final Map map;
  static {
    Map aMap = new HashMap();
    aMap.put(1, "one");
    aMap.put(2, "two");
    map = Collections.unmodifiableMap(aMap);
  }
}
```

Guava libraries also support different ways of intilizaing a static and immutable collection. To learn more about the benefits of Guava's immutable collection utilities, see *Immutable Collections Explained in Guava User Guide*.

5. Difference between HashMap, TreeMap, and Hashtable

There are three main implementations of **Map** interface in Java: **HashMap**, **TreeMap**, and**Hashtable**. The most important differences include:

1. **The order of iteration**. **HashMap** and **Hashtable** make no guarantees as to the order of the map; in particular, they do not guarantee that the order will remain constant over time. But**TreeMap** will iterate the whole entries according the "natural ordering" of the keys or by a comparator.
2. **key-value** permission. **HashMap** allows *null* key and *null* values (Only one null key is allowed because no two keys are allowed the same). **Hashtable** does not allow *null* key or*null* values. If **TreeMap** uses natural ordering or its comparator does not allow *null* keys, an exception will be thrown.
3. **Synchronized**. Only **Hashtable** is synchronized, others are not. Therefore, "if a thread-safe implementation is not needed, it is recommended to use **HashMap** in place of **Hashtable**."

A more complete comparison is

```
                | HashMap | Hashtable | TreeMap

-----------------------------------------------------

iteration order  | no      | no        | yes

null key-value   | yes-yes | no-no   | no-yes

synchronized     | no      | yes       | no

time performance | O(1)    | O(1)      | O(log n)

implementation   | buckets | buckets   | red-black tree
```

Read more about HashMap vs. TreeMap vs. Hashtable vs. LinkedHashMap.

6. A Map with reverse view/lookup

Sometimes, we need a set of key-key pairs, which means the map's values are unique as well as keys (one-to-one map). This constraint enables to create an "inverse lookup/view" of a map. So we can lookup a key by its value. Such data structure is called **bidirectional map**, which unfortunetely is not supported by JDK.
Both Apache Common Collections and Guava provide implementation of bidirectional map, called **BidiMap** and **BiMap**, respectively. Both enforce the restriction that there is a 1:1 relation between keys and values.

7. Shallow copy of a Map

Most implementation of a map in java, if not all, provides a constructor of copy of another map. But the copy procedure is **not synchronized**. That means when one thread copies a map, another one may modify it structurally. To [prevent accidental unsynchronized copy, one should use **Collections**.*synchronizedMap()* in advance.

```
Map copiedMap = Collections.synchronizedMap(map);
```

Another interesting way of shallow copy is by using *clone()* method. However it is **NOT** even recommended by the designer of Java collection framework, Josh Bloch. In a conversation about "Copy constructor versus cloning", he said

> *I often provide a public clone method on concrete classes because people expect it. ... It's a shame that Cloneable is broken, but it happens. ... Cloneable is a weak spot, and I think people should be aware of its limitations.*

For this reason, I will not even tell you how to use *clone()* method to copy a map.

8. Create an empty Map

If the map is immutable, use

```
map = Collections.emptyMap();
```

Otherwise, use whichever implementation. For example

```
map = new HashMap();
```

THE END