**Question 1: What’s wrong using HashMap in the multi-threaded environment? When does the get() method go to an infinite loop?** ([answer](http://java67.blogspot.com/2013/06/how-get-method-of-hashmap-or-hashtable-works-internally.html))

Well, nothing is wrong, depending on how we use it. For example, if we initialize the HashMap just by one thread and then all threads are only reading from it, then it’s perfectly fine. One example of this is a Map which contains configuration properties.

The real problem starts when at-least one of that thread is updating HashMap i.e. adding, changing or removing any key value pair. Since put() operation can cause re-sizing and which can further lead to infinite loop, that’s why either you should use [Hashtable](http://javarevisited.blogspot.com/2012/01/java-hashtable-example-tutorial-code.html) or [ConcurrentHashMap](http://javarevisited.blogspot.com/2013/02/concurrenthashmap-in-java-example-tutorial-working.html), later is better.

**Question 2. Does overriding the hashCode() method have any performance implication?** ([answer](http://java67.blogspot.com/2013/04/example-of-overriding-equals-hashcode-compareTo-java-method.html))

This is a good question and open to all, as per my knowledge a poor hash code function will result in the frequent collision in HashMap which eventually increases the time for adding an object into Hash Map.

From Java 8 onwards though, collision will not impact performance as much as it does in earlier versions, because after a threshold the linked list will be replaced by the binary tree, which will give you O(logN) performance in the worst case, as compared to O(n) of linked list.

**Question 3: Do all properties of an Immutable Object need to be final?** ([answer](http://javarevisited.blogspot.com/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html))

Not necessarily, as stated above you can achieve same functionality by making the member non-final but private and not modifying them except in a constructor. Don’t provide setter methods for them and if it is a mutable object, then don’t ever leak any reference for that member.

Remember making a reference variable final, only ensures that it will not be reassigned to a different value, but you can still change individual properties of object, pointed by that reference variable. This is one of the key point, Interviewer like to hear from candidates.

**Executor.execute() vs ExecutorService.submit() method**

1) Both submit() and execute() methods are used to submit a task to [Executor framework](http://javarevisited.blogspot.com/2013/07/how-to-create-thread-pools-in-java-executors-framework-example-tutorial.html) for asynchronous execution.  
  
2) Both submit() and execute() can accept a Runnable task.  
  
3) You can access submit() and execute() from the ExecutorService interface because it also extends the Executor interface which declares the execute() method.  
  
Apart from the fact that submit() method can return output and execute() cannot, following are other notable differences between these two key methods of Executor framework of Java 5.  
  
1) The submit() can accept both [Runnable](http://java67.blogspot.com/2016/01/7-differences-between-extends-thread-vs-implements-Runnable-java.html) and [Callable](http://javarevisited.blogspot.com/2015/06/how-to-use-callable-and-future-in-java.html) task but execute() can only accept the Runnable task.  
  
2) The submit() method is declared in ExecutorService interface while execute() method is declared in the Executor interface.  
  
3) The return type of submit() method is a Future object but return type of execute() method is void.

Read more: <https://javarevisited.blogspot.com/2016/04/difference-between-ExecutorServie-submit-vs-Executor-execute-method-in-Java.html#ixzz5uIul0vYu>