Java Collections Interview Questions

1. Introduction

Java Collections is a topic often brought up on technical interviews for Java developers. This article reviews some important questions that are asked most often and may be tricky to get right.

2. Questions

Q1. Describe the Collections type hierarchy. What are the main interfaces, and what are the differences between them?

The ***Iterable*** interface represents any collection that can be iterated using the *for-each* loop. The ***Collection*** interface inherits from *Iterable* and adds generic methods for checking if an element is in a collection, adding and removing elements from the collection, determining its size etc.

The ***List***, ***Set***, and ***Queue*** interfaces inherit from the *Collection* interface.

List is an ordered collection, and its elements can be accessed by their index in the list.

Set is an unordered collection with distinct elements, similar to the mathematical notion of a set.

Queue is a collection with additional methods for adding, removing and examining elements, useful for holding elements prior to processing.

Map interface is also a part of the collection framework, yet it does not extend *Collection*. This is by design, to stress the difference between collections and mappings which are hard to gather under a common abstraction. The *Map* interface represents a key-value data structure with unique keys and no more than one value for each key.

Q2. Describe various implementations of the *Map* interface and their use case differences.

One of the most often used implementations of the *Map* interface is the ***HashMap***. It is a typical hash map data structure that allows accessing elements in constant time, or O(1), but **does not preserve order and is not thread-safe**.

To preserve insertion order of elements, you can use the ***LinkedHashMap*** class which extends the *HashMap* and additionally ties the elements into a linked list, with foreseeable overhead.

The ***TreeMap*** class stores its elements in a red-black tree structure, which allows accessing elements in logarithmic time, or O(log(n)). It is slower than the *HashMap* for most cases, but it allows keeping the elements in order according to some *Comparator*.

The **ConcurrentHashMap** is a thread-safe implementation of a hash map. It provides full concurrency of retrievals (as the *get* operation does not entail locking) and high expected concurrency of updates.

The ***Hashtable*** class has been in Java since version 1.0. It is not deprecated but is mostly considered obsolete. It is a thread-safe hash map, but unlike *ConcurrentHashMap*, all its methods are simply *synchronized*, which means that all operations on this map block, even retrieval of independent values.

Q3. Explain the difference between *LinkedList* and *ArrayList.*

ArrayList is an implementation of the *List* interface that is based on an array. *ArrayList*internally handles resizing of this array when the elements are added or removed. You can access its elements in constant time by their index in the array. However, inserting or removing an element infers shifting all consequent elements which may be slow if the array is huge and the inserted or removed element is close to the beginning of the list.

LinkedList is a doubly-linked list: single elements are put into *Node* objects that have references to previous and next *Node*. This implementation may appear more efficient than *ArrayList* if you have lots of insertions or deletions in different parts of the list, especially if the list is large.

In most cases, however, *ArrayList* outperforms *LinkedList*. Even elements shifting in *ArrayList*, while being an O(n) operation, is implemented as a very fast *System.arraycopy()* call. It can even appear faster than the *LinkedList*'s O(1) insertion which requires instantiating a *Node*object and updating multiple references under the hood. *LinkedList* also can have a large memory overhead due to a creation of multiple small *Node* objects.

Q4. What is the difference between *HashSet* and *TreeSet*?

Both ***HashSet*** and ***TreeSet*** classes implement the *Set* interface and represent sets of distinct elements. Additionally, *TreeSet* implements the *NavigableSet* interface. This interface defines methods that take advantage of the ordering of elements.

HashSet is internally based on a *HashMap*, and *TreeSet* is backed by a *TreeMap* instance, which defines their properties: *HashSet* does not keep elements in any particular order. Iteration over the elements in a *HashSet* produces them in a shuffled order. *TreeSet*, on the other hand, produces elements in order according to some predefined *Comparator*.

Q5. How is *HashMap* implemented in Java? How does its implementation use *hashCode* and *equals* methods of objects? What is the time complexity of putting and getting an element from such structure?

The *HashMap* class represents a typical hash map data structure with certain design choices.

The *HashMap* is backed by a resizable array that has a size of power-of-two. When the element is added to a *HashMap*, first its *hashCode* is calculated (an *int* value). Then a certain number of lower bits of this value are used as an array index. This index directly points to the cell of the array (called a bucket) where this key-value pair should be placed. Accessing an element by its index in an array is a very fast O(1) operation, which is the main feature of a hash map structure.

A *hashCode* is not unique, however, and even for different *hashCodes*, we may receive the same array position. This is called a collision. There is more than one way of resolving collisions in the hash map data structures. In Java's *HashMap*, each

bucket actually refers not to a single object, but to a red-black tree of all objects that landed in this bucket (prior to Java 8, this was a linked list).

So when the *HashMap* has determined the bucket for a key, it has to traverse this tree to put the key-value pair in its place. If a pair with such key already exists in the bucket, it is replaced with a new one.

To retrieve the object by its key, the *HashMap* again has to calculate the *hashCode* for the key, find the corresponding bucket, traverse the tree, call *equals* on keys in the tree and find the matching one.

HashMap has O(1) complexity, or constant-time complexity, of putting and getting the elements. Of course, lots of collisions could degrade the performance to O(log(n)) time complexity in the worst case, when all elements land in a single bucket. This is usually solved by providing a good hash function with a uniform distribution.

When the *HashMap* internal array is filled (more on that in the next question), it is automatically resized to be twice as large. This operation infers rehashing (rebuilding of internal data structures), which is costly, so you should plan the size of your *HashMap*beforehand.

Q6. What is the purpose of the initial capacity and load factor parameters of a *HashMap*? What are their default values?

The *initialCapacity* argument of the *HashMap* constructor affects the size of the internal data structure of the *HashMap*, but reasoning about the actual size of a map is a bit tricky. The *HashMap*‘s internal data structure is an array with the power-of-two size. So the *initialCapacity* argument value is increased to the next power-of-two (for instance, if you set it to 10, the actual size of the internal array will be 16).

The load factor of a *HashMap* is the ratio of the element count divided by the bucket count (i.e. internal array size). For instance, if a 16-bucket *HashMap* contains 12 elements, its load factor is 12/16 = 0.75. A high load factor means a lot of collisions, which in turn means that the map should be resized to the next power of two. So the *loadFactor* argument is a maximum value of the load factor of a map. When the map achieves this load factor, it resizes its internal array to the next power-of-two value.

The *initialCapacity* is 16 by default, and the loadFactor is 0.75 by default, so you could put 12 elements in a *HashMap* that was instantiated with the default constructor,

and it would not resize. The same goes for the *HashSet*, which is backed by a *HashMap* instance internally.

Consequently, it is not trivial to come up with *initialCapacity* that satisfies your needs. This is why the Guava library has *Maps.newHashMapWithExpectedSize()* and *Sets.newHashSetWithExpectedSize()* methods that allow you to build a *HashMap* or a *HashSet* that can hold the expected number of elements without resizing.

Q7. Describe special collections for enums. What are the benefits of their implementation compared to regular collections?

EnumSet and ***EnumMap*** are special implementations of *Set* and *Map* interfaces correspondingly. You should always use these implementations when you're dealing with enums because they are very efficient.

An *EnumSet* is just a bit vector with "ones" in the positions corresponding to ordinal values of enums present in the set. To check if an enum value is in the set, the implementation simply has to check if the corresponding bit in the vector is a "one", which is a very easy operation. Similarly, an *EnumMap* is an array accessed with enum's ordinal value as an index. In the case of *EnumMap*, there is no need to calculate hash codes or resolve collisions.

Q8. What is the difference between fail-fast and fail-safe iterators?

Iterators for different collections are either fail-fast or fail-safe, depending on how they react to concurrent modifications. The concurrent modification is not only a modification of collection from another thread but also modification from the same thread but using another iterator or modifying the collection directly.

Fail-fast iterators (those returned by *HashMap*, *ArrayList*, and other non-thread-safe collections) iterate over the collection's internal data structure, and they throw *ConcurrentModificationException* as soon as they detect a concurrent modification.

Fail-safe iterators (returned by thread-safe collections such as *ConcurrentHashMap*, *CopyOnWriteArrayList*) create a copy of the structure they

iterate upon. They guarantee safety from concurrent modifications. Their drawbacks include excessive memory consumption and iteration over possibly out-of-date data in case the collection was modified.

Q9. How can you use *Comparable* and *Comparator* interfaces to sort collections?

The *Comparable* interface is an interface for objects that can be compared according to some order. Its single method is *compareTo*, which operates on two values: the object itself and the argument object of the same type. For instance, *Integer*, *Long*, and other numeric types implement this interface. *String* also implements this interface, and its *compareTo*method compares strings in lexicographical order.

The *Comparable* interface allows to sort lists of corresponding objects with the *Collections.sort()* method and uphold the iteration order in collections that implement *SortedSet* and *SortedMap*. If your objects can be sorted using some logic, they should implement the *Comparable* interface.

The *Comparable* interface usually is implemented using natural ordering of the elements. For instance, all *Integer* numbers are ordered from lesser to greater values. But sometimes you may want to implement another kind of ordering, for instance, to sort the numbers in descending order. The *Comparator* interface can help here.

The class of the objects you want to sort does not need to implement this interface. You simply create an implementing class and define the *compare* method which receives two objects and decides how to order them. You may then use the instance of this class to override the natural ordering of the *Collections.sort()* method or *SortedSet* and *SortedMap*instances.

As the *Comparator* interface is a functional interface, you may replace it with a lambda expression, as in the following example. It shows ordering a list using a natural ordering (*Integer*‘s *Comparable* interface) and using a custom iterator (*Comparator<Integer>* interface).

```
1  List<Integer> list1 = Arrays.asList(5, 2, 3, 4, 1);
2  Collections.sort(list1);
3  assertEquals(new Integer(1), list1.get(0));
4
5  List<Integer> list1 = Arrays.asList(5, 2, 3, 4, 1);
6  Collections.sort(list1, (a, b) -> b - a);
7  assertEquals(new Integer(5), list1.get(0));
```