**Java Collections**

**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.

### ****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.

### ****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.

### ****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.

### ****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.

### ****What is the difference between fail-fast and fail-safe iterators?****

**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.

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

When your class implements [Comparable](http://docs.oracle.com/javase/6/docs/api/java/lang/Comparable.html), the compareTo method of the class is defining the "natural" ordering of that object. That method is contractually obligated (though not demanded) to be in line with other methods on that object, such as a 0 should always be returned for objects when the .equals() comparisons return true.

A [Comparator](http://docs.oracle.com/javase/6/docs/api/java/util/Comparator.html) is its own definition of how to compare two objects, and can be used to compare objects in a way that might not align with the natural ordering.