Java Container Overview

Based on: http://www.corejavainterviewquestions.com/java-collections-data-structures-interview-questions/

The Java Collection Framework is simply the set of collection types that are included as part of core Java, and with which every serious Java programmer should be familiar. There are four "high-level" types of collection you will normally deal with when writing Java code; sets, lists and queues of various kinds (all of which implement the *Collection* interface, which in turn implements the *Iterable* interface) and maps of various kinds (which implement the *Map* interface). Maps are generally regarded as part of the Java Collection Framework, even though they do not implement the *Collection* interface. One view holds that they are technically not part of the core collections. Each type of collection (or map) exhibits certain characteristics which determine the way it is used. The key features of any collection (or map) are these:

- One or more elements can be added or removed
- It has a size that can be queried
- It may or may not contain duplicates
- It may or may not provide an ordering of its elements
- It may or may not provide positional access (that is, you may or may not be able to access directly the element at a particular location, using an index)

The behavior of the methods that perform these actions varies based on the implementation (as you would expect from an interface). For example, what happens if you call a *remove()* method on a collection for an object that doesn't exist depends on the implementation.

General Description of the Collection Types

Set **(implements *Collection*):** A **set** has no duplicates and no guaranteed order. Because of this, it does not provide positional access. Example implementations include **TreeSet** and **HashSet**.

List **(implements *Collection*):** A **list** may or may not contain duplicates and also guarantees order, allowing positional access. Example implementations include **ArrayList** and **LinkedList**.

Queue **(implements *Collection*):** A **queue** is a collection designed for holding elements prior to processing. Besides the basic *Collection* operations, queues provide additional insertion, extraction, and inspection operations, and each of these methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either `null` or `false`, depending on the operation). The latter form of the insert operation is designed specifically for use with capacity-restricted ***Queue*** implementations.

Map **(does not implement *Collection*):** A **map** is slightly different as it contains key-value pairs as opposed to objects. The keys of a map may not contain duplicates. A map has no guaranteed order and no positional access. Example implementations include **HashMap** and **TreeMap**.

Note: There are, of course, other more specialized types, including the ***Deque*** interface, which is a subinterface of the ***Queue*** interface, and the **ArrayDeque** class, which is recommended for use as a stack (rather than the legacy Java **Stack** class), the **PriorityQueue** class, which implements the ***Queue*** interface. These are all Java-specific. There are also more general data types, such as **trees** of various kinds (**binary search tree**, **expression tree**, and **heap**, for example) as well as **graphs**, none of which are available directly from the standard Java API. These may be obtained either from a third-party library, or by going to the trouble of implementing your very own independent, standalone container.

So … which to choose?

Question: Why would you choose one implementation of one of these interfaces over another, and what are the implications of any particular choice?

Whether you choose a set, list or map will depend on the structure of your data, and the answers to at least the following questions, which should always be asked when you are trying to choose a container for your data:

- What operations (inserting, deleting, searching) are going to be performed most of the time?
- Must elements be stored in some particular order (insertion order, or sorted, for example)?
- Must elements be retrieved in a particular way (by location, or by key, for example)?

And here's some advice that will, of course, depend on the answers to the above questions:

- If you won't have duplicates and you don't need order, then your choice should probably be a set.
- If you need to have ordered data, then a list should be a good choice.
- If you have key-value pairs (such as if you want to associate two different objects or if an object has an obvious identifier), then a map may be the best choice
- If you require FIFO (first-in, first-out) behavior, you want a queue, while if you need LIFO behavior (last-in, first-out) you need a stack.

Also, for large amounts of data, performance is important, so attention needs to be paid to the following questions as well:

- Speed of access (Will elements be accessed directly, sequentially, in some other order?)
- Speed of insertion (Will elements be added at one "end", or "in the middle"?)
- Speed of deletion (Will any "holes" have to be filled when an element is removed?)
- Speed of iteration (How fast can the entire container be traversed?)

Some implementations may offer performance guarantees for certain operations, while others may not. How that performance guarantee is determined will depend on the implementation. This is a matter for discussion under the heading of "algorithm analysis". The bottom line in this context is that if you choose a particular implementation of a particular container type, you will want to know what the performance of your most common operations will be for that container.

Examples

- A collection of colors could be put into a set, since colors have no natural ordering and it would likely be pointless to have duplicate copies of any particular color.
- The batting order of a baseball team would be a good candidate for storage in a list, since you would want to retain the order of the objects.
- A printer queue that prints jobs as they come in would be an example of a standard (first-in, first-out queue), while a print queue that prioritizes jobs by printing the shorter ones first would be an example of a "priority queue".
- A collection of web sessions might be best placed in a map, since the unique session ID would make a good key or reference to each session considered as a complete object.

Collection Implementations

When it comes to implementing a concrete class that implements a particular interface, there are generally four "behind the scenes" data structures that are used:

- arrays (contiguous storage)
- linked nodes (non-contiguous)
- trees (can be contiguous storage or not)
- hash tables (can be contiguous storage or not)

The performance of any particular operation in any particular Java container will depend on which of these underlying data structures has been used in its implementation.