Java Collections

JDK Collections Interfaces and Implementations

You might think that the collections framework was always a part of the JDK. Surprise, surprise, that's not true. The Java Collections API is one of the most fundamentals parts of the JDK, or any programming language for that matter, but it's also one of the hardest aspects of programming to get right. On top of that it's really challenging to make the collections library simple, coherent, and easy to use (I'm looking at you, Scala).

So the Java collections library was introduced to Java in the Java 1.2 in 1998 and has stuck with us since then. Since backwards compatibility is one of the core values of the Java platform, collections haven't changed a lot since then. However, recently in the Java 8 release, the language evolved enough to provide tools to enhance the collection interfaces without breaking backwards compatibility.

All in all, Java collections is a comprehensive library that contains the utilities to tackle almost any container-like data structure you might need. Let's dive into which types of collections are available and outline what makes them special and why you might want to use them.

Short Description of the Most Used Java Collections

There are two main approaches to choosing the collection type you need in your code. The first is simple enough, but not always the most effective:

if it fits into an ArrayList, use ArrayList, otherwise you probably need a HashMap.

The other includes having an understanding of the operations you will be needing from the collection, the performance you would expect from them and what kind of data you intend to put into the collection. While the advice above can take you quite far, it's much better to understand the options available to help you make an informed decision.

Java 8 Collections You Need to Master

There are 4 main types of Java 8 collections available. Naturally, the implementations for each type can be found under the corresponding interfaces. Before we start digging into these, there are 2 main umbrella interfaces you want to know about: Iterable and Collection. Typically you rarely need to think about these, but it's important to at least understand them. The iterable interface allows one to obtain an iterator and traverse the sequence of elements by calling the next() method. It also makes it possible to iterate through elements with the syntactic sugar of the "for-each" loop:

for(E e: iterable)

The collection interface extends the Iterable and represents an iterable group of elements which can be added, removed, or checked for presence in the collection. However, the implementations usually implement the more specific interfaces that are designed with the collection implementation in mind. Here are the most used collection interfaces.

Java Collections: List

A List is an ordered collection of elements. Some languages call that a sequence, or you can think of if it as an array of varying length. You can add elements to it, in the middle of it, access and replace the elements using an index.

Not surprisingly, the list is one of the most common collections used. However, there's almost no variety in the used implementations required. Yeah, in school you learned about the dreaded linked lists and how to implement them and how awesome it is to have access to the head and the tail of it. In practice, you are better using the ArrayList 99% of the time.

The reason for that is quite simple, it consumes less memory, it is friendlier to the caches, and it is in general faster than the LinkedList from the JDK. In fact, you should try not to use a LinkedList.

If you foresee that you'll need to establish concurrent access to a list, you'll need to worry about the synchronization yourself. Typically, that means that either you will use a CopyOnWriteArrayList which is thread-safe and immutable. If your data can be mutated by another thread, but the performance of the reading accesses is much more important. CopyOnWriteArrayList is your list of choice.

Java Collections: Set

A collection that contains no duplicate elements is a set. Just like the mathematical set, the interface denotes a collection that holds elements and basically answers a single question: is a given element contained in the set.

Set doesn't necessarily specify the order of elements when you iterate them, but all set implementations are created with the performance of the contains method in mind.

There are two main Set implementations that you should know about: HashSet and TreeSet. HashSet hashes the elements and distributes them into buckets by the hash value. Tree set is backed by a balanced tree, which makes it ordered and navigable (so you can ask what for the previous and next elements by value). TreeSet operations will have worse complexity compared to the HashSet as a result, but bear in mind that the operations still take sublinear time of the set size, which means that for realistic values of the set sizes, it's still quite a short time.

However, typically use-cases for sets do not require navigating from element to element, so the HashSet is the goto implementation you'll tend to use. And it's a good one. Just remember to correctly implement the hashCode() and equals() methods for your elements.

Java Collections: Map

Perhaps the most used collection type of all time -- an object that maps keys to values, an associative array, a table: the map. Maps are the most versatile collection type because the association of keys to values is really what computers are all about: a dictionary, mapping object properties to their values just like javascript does, mapping filenames to their contents and metadata, mapping usernames to password hashes, session attributes, product cart items, game high scores. Wherever you look you'll find a use case for a map.

Naturally there are different implementations that are tweaked to provide different performance trade-offs. But the default goto implementation of the map interface is undoubtedly the infamous HashMap.

The Hashmap implementation depends on the Key objects implementing the hashCode() and equals() methods correctly, so always take care of these. In return the HashMap promises you the almost constant time performance which is as amazing as you can get out of any data structure and scales amazingly well.

If you need the navigation between elements or you'd need to deal with unhashable element types, you can use a TreeMap. Using a balanced tree, just like the sets above, treemap scales appropriately and gives you a chance to iterate the contents in a sorted order.

Now we don't want to dive into all the Java Collection API methods for maps. You can put and get values, as you'd expect. But we would love to point out one super amazing method that has been available to us since Java 8: **computeIfAbsent**.

```
default V computeIfAbsent(K key,
                          Function<? super K,? extends V> mappingFunction)
```

What computeIfAbsent does is give you an opportunity to specify how to obtain the value for the key, if it's not in the collection yet. Essentially it checks if the given key is contained in the map. If it happens to be so, you get the corresponding value back. If not, the mapping function is executed and the resulting values is put into the map and returned to you.

That's more or less how caches work. Amazing, right?

Java Collections: Queue

A queue is a collection designed to hold elements prior to processing, the elements wait in line to be consumed and can be added to the tail of the queue. Queues are those fundamental pieces of software that tie components together. When components are communicating in a system, there's typically a queue of messages sitting between them.

Queues are a special sort of collection, because are mostly operated using the Queue interface, rather than methods inherited from the Collection interface. To add an element to a queue, use:

```
E e;
Queue q = …
q.offer(e); // try to put the element into the queue, if the queue is full, do nothing
q.poll(); // remove the head of the queue
q.peek(); // return the head of the queue without modifying it.
```

There are also corresponding methods that throw exceptions if operations don't succeed. But all in all, there are three main operations one performs on a queue: enqueue, deque and peek. Note that

you can get an iterator from the queue and process the elements similar to any other collection. It would even work for some implementations of the queue interface, but not for all.

These are the typical classes that you might use that implement the queue interface:

- ArrayDeque - a good general purpose battle-tested implementation of a queue. ArrayDeque is backed by a resizeable array that can grow to accommodate additional elements. Most ArrayDeque operations run in amortized constant time.
- PriorityQueue - a queue of sorted elements. The sorting is achieved either naturally or determined by a provided comparator, but the head of the queue is always the minimal element. Note that a priority queue won't change its order if you mutate the elements inside it. If you're curious about the performance, the mutating operations run in logarithmic time of the queue size, peeking and such -- in constant, and the search is linear.

Java Collection Utilities

Now we've talked about all the most common collection types you'll encounter, it's time to dig into the other parts of the Java collections framework. There's an amazing class, called Collections, that provides handy utility methods that you should be aware of. First of all let's talk about the collection wrappers. Any collection type can have additional requirements to its functionality, these orthogonal qualities are implemented with the wrappers. The original collection is wrapped into another one that delegates the element handling logic to it.

The prime example of some functionality that is suitable for a wrapper is making a collection immutable. Indeed, you just need to ignore all the mutating operations like: put, insert, and so on. And all the non-mutating operations can be easily delegated to the actual collection implementations.

The Collections class provides a set of methods that give you just that: unmodifiable collections:

`static <T> Collection<T>`	**`unmodifiableCollection(Collection<? extends T> c)`** Returns an unmodifiable view of the specified collection.
`static <T> List<T>`	**`unmodifiableList(List<? extends T> list)`** Returns an unmodifiable view of the specified list.
`static <K,V> Map<K,V>`	**`unmodifiableMap(Map<? extends K,? extends V> m)`** Returns an unmodifiable view of the specified map.
`static <K,V> NavigableMap<K,V>`	**`unmodifiableNavigableMap(NavigableMap<K,? extends V> m)`** Returns an unmodifiable view of the specified navigable map.
`static <T> NavigableSet<T>`	**`unmodifiableNavigableSet(NavigableSet<T> s)`** Returns an unmodifiable view of the specified navigable set.
`static <T> Set<T>`	**`unmodifiableSet(Set<? extends T> s)`** Returns an unmodifiable view of the specified set.
`static <K,V> SortedMap<K,V>`	**`unmodifiableSortedMap(SortedMap<K,? extends V> m)`** Returns an unmodifiable view of the specified sorted map.
`static <T> SortedSet<T>`	**`unmodifiableSortedSet(SortedSet<T> s)`** Returns an unmodifiable view of the specified sorted set.

Another useful type of wrapper available is the synchronized collections wrapper. Indeed, most of the collection implementations do not implement the synchronization themselves. What would you do if you need to use HashSet concurrently from multiple threads? That's right, you call the corresponding wrapper method and use the returned set.

Collections.synchronizedSet(new HashSet());

It's important to remember that the wrapper does not change the implementation of the underlying collection. That is if you obtain a reference to the wrapped hash set and use that concurrently there are no guarantees.

Here's a short list of the useful methods in the Collections class that typically are used more than others:

- binarySearch - a fast way to find an element in a sorted List.
- frequency - tells you how many times an element is encountered in a collection
- min / max - returns the smallest / largest element of the collection
- reverseOrder - provides you with a Comparator to sort elements in the descending order
- singleton - wraps an object into the Set containing that object, the similar methods are available for the List and Map classes

Other Java Collections Libraries

There are more questions that you can ask of your collection implementation and obviously there are more answers than just the stock implementations you can find in the JDK. Here are some libraries that we think are amazing and give you more functionality for the collection creation and manipulation and more collection types.

- Guava - *Google Core Libraries for Java 6+*. Perhaps the default third party collection library for Java projects. Contains a magnitude of convenient methods for creating collection, like fluent builders, as well as advanced collection types.
- Eclipse Collections - *Features you want with the collections you need*. Previously known as gs-collections, this library includes almost any collection you might need: primitive type collections, multimaps, bidirectional maps and so on.
- Fastutil - *Fast & compact type-specific collections for Java*. Great default choice for collections of primitive types, like int or long. Also handles big collections with more than 2^31 elements.
- JCTools - *Java Concurrency Tools for the JVM*. If you work on high throughput concurrent applications and need a way to increase your performance, check out JCTools. It contains lots of queues for all kinds of single / multiple producer / consumer environments and other useful classes.

Sure, this is not a definitive list, but these are the essential libraries that you should know about. You might be using Guava in your project already; at least once you have thought about how awesome would it be to have the collections for primitives, and so on. So check these out!

Improvements Java Collections

It's not easy to change something as fundamental as the collections library in the JDK. However, the Java 8 release had implemented a major enhancements to the collections library. Java 8

collections default methods made it possible to evolve the collection interfaces without breaking all of the implementations.

The progress doesn't stop there. The upcoming Java 9 release contains the JEP 269: Convenience Factory Methods for Collections, the proposal to include the methods for easier collection creation. Finally, we would be able to specify the collections with a simpler syntax like:

```
List.of(a, b, c);

Set.of(d, e, f, g);

Map.ofEntries(
   entry(k1, v1),
   entry(k2, v2),
   entry(k3, v3));
```

That would be really sweet.

Final Thoughts

We've talked a lot about Java collections and different implementations of the collection interfaces you can find in the java.util package. Sets, Lists and Maps are your friends and you'll need to apply quite a bit of imagination to come up with a program that doesn't need the Java collections API. We touched on the subject of collection wrappers and how to make collections immutable and synchronize them for concurrent access.