Java Collections Framework

Introduction

In this article from my free Java 8 course, you will be given a high-level introduction of the Java Collections Framework (JCF). The term 'Collection' has several meanings, unfortunately. To clear things up, we will first discuss the word's meanings upfront.

'Collection' can refer to:

- its day-to-day meaning as *a compilation or group of things*.
- the collection of interfaces and classes that make up the Java Collections Framework.
- some data structure like a box or container that can hold a group of objects like an array.
- the `java.util.Collection` interface, one of the two main JCF interfaces.
- `java.util.Collections`, a utility class which can help to modify or operate on Java collections.

This piece is based on Chapter 11 of the OCA/OCP Study Guide, a book packed with knowledge on Java programming. As a great fan of the authors, Kathy Sierra and Bert Bates, I recommend you read the book even if you don't plan on being a certified Java programmer.

What is the Java Collections Framework, from a high-level perspective? First of all, it is in fact a library, a toolbox of generic interfaces and classes. This toolbox contains various collection interfaces and classes that serve as a more powerful, object-oriented alternative to arrays. Collection-related utility interfaces and classes also make for better ease of use.

Overview

In this section, we will be going into more detail as we delve into the interface and class hierarchy for collections. Unlike arrays, all collections can dynamically grow or shrink in size. As I said before, collections hold groups of objects. A *map* can store strongly-related *pairs* of objects together, each pair being made up of a *key* and a *value*. A value does **not** have a specific position in a map, but can be retrieved using the key it is paired with. Don't worry if this is too much to take in right now as we will take a more detailed look later on.

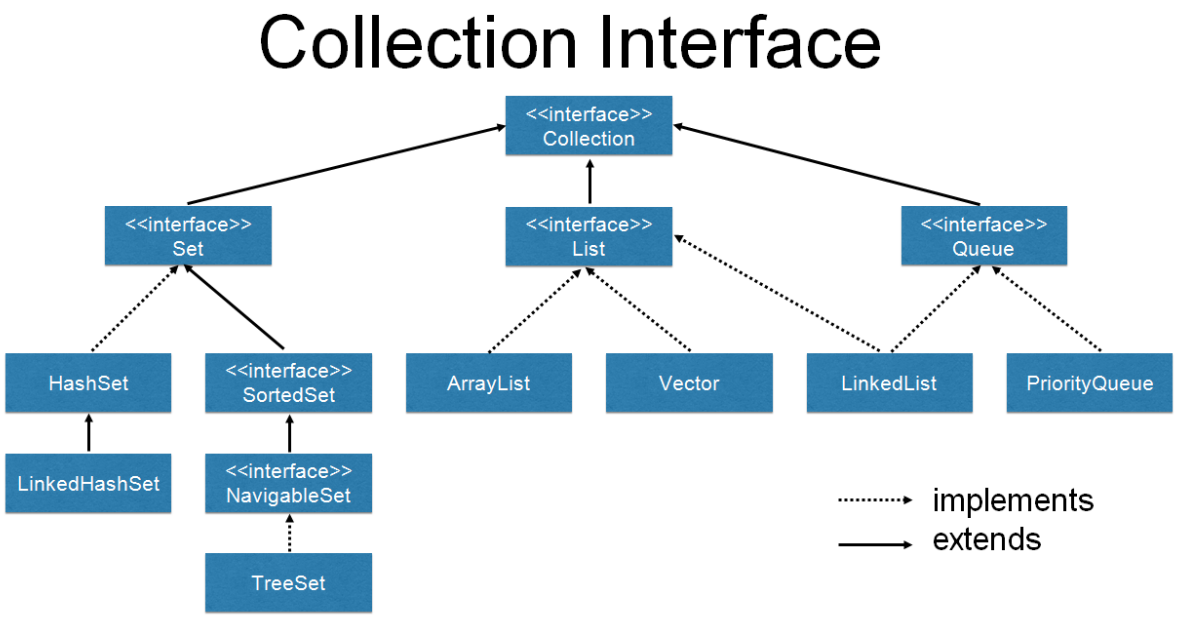

Figure 1, the Collection hierarchy

Figure 1 shows the hierarchy of classes and interfaces extending or implementing the `Collection` interface – it would be useful to at least familiarize yourself with the names listed there. The `Collection` interface sits on top of a number of sub-interfaces and implementing classes. A `Collection` can hold a group of objects in different ways which `Set`, `List` and `Queue` provide. A `Set` is defined as a group of unique objects. What is considered to be unique is defined by the `equals` method of the object type it holds. In other words, a `Set` **cannot** hold two equal objects. A `List` is defined as a sequence of objects. In contrast to a `Set`, a `List` can contain duplicate entries. It also keeps its elements in the order in which they were inserted. A `Queue` has two sides. Entries are added to its tail end while entries are removed from the top or the head. This is often described “first-in, first-out” (FIFO), which is oftentimes much like waiting in line in real life, i.e. the first person queuing up is the first person to leave the queue.

The `Set` Interface

`HashSet`, `LinkedHashSet` and `TreeSet`

`HashSet`, `LinkedHashSet` and `TreeSet` are implementations of `Set`, located around the left end of the `Collection` interface hierarchy in Figure 1. `HashSet` is the default implementation used in most cases. `LinkedHashSet` is like a combination of `HashSet` and `List` in that it does not allow duplicate entries as with `Set`s, but traverses its elements in the order they were inserted, like a `List` would do.

TreeSet will constantly keep all its elements in some sorted order. Keep in mind, however, that there is no such thing as a *free lunch* and that every added feature comes at a certain cost.

`SortedSet` and `NavigableSet`

After looking at three classes implementing `Set`, let's also take a look at the two sub-interfaces we haven't talked about yet. As the name implies, `SortedSet` is a `Set` with the property that it is always sorted. The `NavigableSet` interface, added with Java 6, allows us to navigate through the sorted list, providing methods for retrieving the next element greater or smaller than a given element of the `Set`.

The `List` Interface

`ArrayList` and `LinkedList`

`ArrayList` is the default implementation for `List`, located to the middle of the collection hierarchy in Figure 1. Like any `List` implementation, it does allow duplicate elements and iteration in the order of insertion. As it is based on arrays, it is very fast to iterate and read from, but very slow to add or remove an element at random positions, as it has to rebuild the underlying array structure. In contrast, `LinkedList` makes it easy to add or remove elements at any position in the list while being slower to read from at random positions.

`Vector`

As a side note, we shortly consider `java.util.Vector`, a class that has been around since JDK 1, before the Collections Framework which was added with Java 2. Long story short, its performance is suboptimal, so no new code should ever have to use it. An `ArrayList` or `LinkedList` simply does a better job.

The `Queue` Interface

Lastly, we take a look at the classes implementing Queue. Another thing to mention about `LinkedList` is that while it implements `List`, it actually also implements `Queue`. It does so based on the fact that its actual implementation as a doubly-linked list makes it quite easy to also implement the `Queue` interface.

`PriorityQueue`

Besides `LinkedList`, another common `Queue` implementation is `PriorityQueue`. It is an implementation that keeps its elements ordered automatically. It has functionality similar to `TreeSet`, except that it allows duplicate entries.

The `Map` Interface

We now take a look at the `Map` interface, one which oddly enough has no relation to the `Collection` interface. A `Collection` operates on one entity, while a `Map` operates on two: a unique key, e.g. a vehicle identification number, and an object related to the key, e.g. a car. To retrieve an object from a `Map`, you would normally use its key. Map is the root of quite a number of interfaces and classes, as depicted on Figure 2.

`Hashtable, Hashmap` and `LinkedHashMap`

The Hashtable class was the first Collection in Java 1 that was based on the hash table data structure. Unfortunately, like `Vector`, the class is deprecated because of its suboptimal performance. We can forget about it and use the other `Map` implementations instead. `HashMap` is the default implementation that you will find yourself using in most cases.

A `Map` usually doesn't make any guarantee as to how it internally stores elements. An exception to this rule, however, is `LinkedHashMap`, which allows us to iterate the map in the order of insertion.

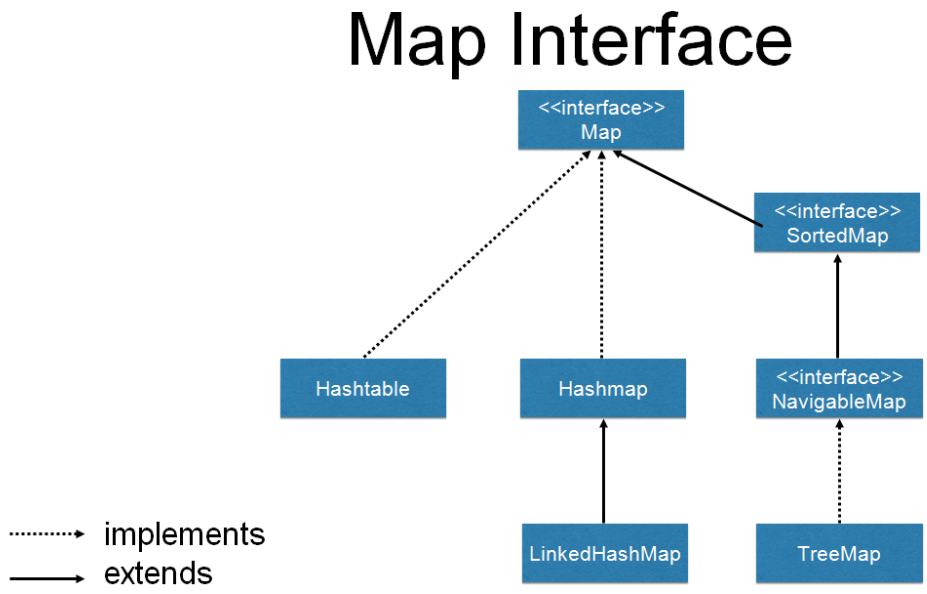

Figure 2, the Map hierarchy

`SortedMap`

Let's look at the interfaces that extend `Map`. As the name implies, `SortedMap` extends `Map` and

defines the contract of a constantly sorted map. `NavigableMap` takes it even further, adding methods to navigate sorted maps. This allows us to get all entries smaller or bigger than some entry, for example. There are actually many similarities between the `Map` and `Set` hierarchies. The reason is that `Set` implementations are actually internally backed by `Map` implementations.

The Bigger Picture

You might have noticed that Java's `Collection` classes often contain data structures based on their name. To choose the best collection for a given situation, you have to compare and match the properties of data structures like `LinkedList`, `Hashtable` or `TreeSet` to the problem at hand. In short, there is no single best option as each one has its own advantages and disadvantages. There is actually a lot more ground to cover on this, as this overview has only shown but a tiny part of the huge scope of `Collection` and `Map` classes. In fact, there are even concurrent containers in the Java Collections Framework which are used for concurrent programming.

Generics

The subject of generics is at least as broad as the Java Collections Framework. In the context of this article, we will only discuss the bare minimum needed to understand the collections framework. It's OK to have a lot of open questions after this brief overview. Everything will be explained one after the other.

```
List<String> myList = new ArrayList<String>(100);
```

Notice the usage of the angle brackets. To the left side, we define a `List` variable `myList` with the `String` parameter in the angle brackets. We tell the compiler that the `myList` variable is to only ever refer to some list that contains `String`s. We then create an object of type `ArrayList` and again tell the compiler that the list is supposed to only contain `String`s. In other words, this is what makes the containers *type-safe*. Also note the use of the `List` type for the variable instead of `ArrayList`. This makes our code more flexible. You will only ever create the object once but you will often end up using it in many places. That being said, when you declare a `List` instead of an `ArrayList`, you get to replace the `ArrayList` with a `LinkedList` later on, and all you had to change was that one line of code.

```
Collection<String> myList = new ArrayList<String>(100);
```

In case you don't really need methods specific to List, you could take it a bit further and use Collection instead. It is a good idea to always use the least specific, smallest interface as possible as a variable type. Note the use of 100 as the ArrayList constructor argument.

In this case, it's a performance optimization. Since ArrayList and all hashtable-based collections internally operate on arrays, when such a collection grows in size, it creates larger arrays on the fly and transfers all contents from the old array to the new one. Although this takes some extra time, modern hardware is so fast that this usually shouldn't be a problem. On the other hand, knowing the exact or even just an approximate size for the collection is better than settling for the default collection sizes. Knowing what data structures Java collections are based on helps give a better understanding of performance in cases like this one. Paying attention to such small details is often the difference between a regular developer and a software craftsman.

```
Map<VIN, Car> myMap = new HashMap<>(100);
```

See how a `Map` is declared and how `HashMap` is constructed above. A map is a relation of one identifying **key element** to one **value element** and both can be of different types. In the example above, `VIN` – the vehicle identification number – is used as the key while a `Car` object is the value. The *type parameters* are added as a comma-separated list in angle-brackets. As of Java 7, if you declare the variable and create the object all in the same line, you can leave the second pair of angle brackets empty as the compiler infers the type of the object from the generic type of the reference variable. The empty angle brackets are called a *diamond operator*, owing its name to how the empty brackets form a diamond shape. What has been discussed so far is just the usage of generic classes, however, where we lock in the concrete type parameters to be used in instantiation. All of this is only possible if some **method**, **interface**, or **class** is defined to be used in a generic way beforehand.

Writing Generic Code

Listing 1 shows a generically defined interface. In the first line, the interface is defined as one operating on two generic types that have to be specified at a later time. When these types are locked in, the types the interface methods use are automatically specified. If you see one-letter types in code, it could mean that it can be used in a generic way.

```
public interface MyInterface<E, T> {
    E read();

    void process(T object1, T object2);
}
```

Listing 1

Other Utility Interfaces

- `java.util.Iterator`
- `java.lang.Iterable`
- `java.lang.Comparable`
- `java.lang.Comparator`

Listed above are some additional utility interfaces from the Java Collections Framework. They are implemented by classes of the framework or the JDK in general. Additionally, they can also be implemented by your own classes, leveraging the features of and interoperability with the Collections Framework. Strictly speaking, `java.lang.Iterable` is not part of the framework, but more precisely sits on top of it. It is the super-interface of `java.util.Collection`, which means that every class that implements Collection also implements `java.lang.Iterable`.

`java.util.Iterator`

- `boolean hasNext();`
- `E next();`
- `void remove();`

An iterator is an object that acts like a remote control for iterating through things, oftentimes collections. `hasNext()` returns true if a collection has more elements, `next()` returns the next element in the iteration, while `remove()` removes the last element returned by an iterator from its underlying collection.

`java.lang.Iterable`

- `Iterator iterator()`

`Iterable` provides only one method which returns an `Iterator`. Every `Collection` that implements this interface can be used in the **for-each loop,** greatly simplifying the usage of your home-made collections. To avail yourself of the use of the for-each loop for your collection, you only have to execute two simple steps: First, write an `Iterator` for your collection and implement all its methods `hasNext`, `next` and `remove`. Second, implement the `Iterable` interface by adding an `iterator` method that returns an instance of the `Iterator` implementation you wrote in the first step.

`java.lang.Comparable`

- `int compareTo(T o)`

Implementing `Comparable` defines a natural sort order for your entities. The interface contains only one method you need to implement, `compareTo`, which compares your Comparable with **`T o`**, the argument representing another entity of the same type. Return a *negative integer* if the object is less than the given argument **`o`**, *`0`* if the object is equal to the **`o`**, and a positive integer if the object is greater than **`o`**.

What it means for one thing to be lesser or greater than another is for you to define. For numbers, it would easily follow that 1 is smaller than 5. But what about colors? This entirely depends on what *you* believe to be the natural ordering of your entities. When you put `Comparable` objects into a `TreeSet` or `TreeMap` for example, it will use your custom-built `compareTo` method to automatically sort all elements in your collection. As you can see, the Java Collections Framework has been greatly designed with extension in mind, offering a lot of possibilities for you to plug in your own classes.

`java.lang.Comparator`

- `int compare(T o1, T o2)`

This interface is very similar to `Comparable`. It allows you to define additional sorting orders, e.g. a reverse order. The sorting logic is not directly implemented in your entity class. Instead, it is defined in an external sorting strategy class that can optionally be attached to a `Collection` or a sorting method to define alternative sorting orders for your collections of entities. The same rules for the interface contract of `Comparable` apply: return a negative integer if the first argument, `o1`, is less than the second argument `o2`, `0` if both arguments are equal, and a positive integer if `o1` is greater than `o2`.

Collections and Arrays

Last but not least, we take a look at the two utility classes `java.util.Collections` and `java.util.Arrays`. Like a Swiss army knife, both provide static helper methods that greatly enhance the general usefulness of the `Collection` classes. `Collections` offers methods like `sort`, `shuffle`, `reverse`, `search`, `min`, and `max`. `Arrays` is actually quite similar to `Collections` except that it operates on raw arrays, i.e. it allows us to sort or search through arrays, for example.

Thanks for reading!