1 The JCF

As in many languages, Java provides several different data structures to us in a library. This allows us to quickly develop solutions without having to write common data structures and be confident in the implementation. So far we have seen `ArrayList` and perhaps `LinkedList`. In Java, these classes do not simply stand alone. Rather, they are part of an overall library called the Java Collections Framework. This week, we will learn not only the various collections in the JCF, but how they are related in an object-oriented way. The idea behind having a framework is manifold:

- Consistent naming of methods (easier to remember)
- Interoperability of classes
- Well-tested and efficient implementations provided

Therefore, the JCF contains:

- Interfaces (collection definitions and others)
- Abstract collections (partially-implemented lists, sets, maps)
- Concrete collections (lists, sets, maps)
- Algorithms commonly used with collections (sort, shuffle, …)
- Some other infrastructure (discussed in detail next week!)

To see one reason why putting these classes into an overall framework is beneficial, let us look at the constructors available for an ArrayList. In addition to the default constructor `ArrayList()` and one that gives an initial size `ArrayList(int)`, there is also `ArrayList(Collection<? extends E>)`. What is `Collection`, and what is it doing there?

1.1 Collection

`Collection<E>` is an interface that all lists and sets (i.e. linear collections) implement. The goal of the JCF here is to enable flexible use of the collections, and allow for code reuse within the JCF, through the use of a single interface. Thus, by providing this one constructor, the `ArrayList` can be constructed using any other List or Set in the JCF!
Actually, even Collection is not at the highest level of the JCF inheritance hierarchy — that distinction goes to the interface Iterable<E>, which Collection inherits from. Therefore, every Collection can be iterated through, which is what allows this constructor to operate — the constructor simply iterates through the given collection, in whatever way the collection allows, and adds all its contents to the ArrayList being constructed.

Collection actually specifies a number of methods that all the concrete collection classes implement, such as size(), contains(Object), and iterator() (which returns an Iterator that is used to walk through the collection)\(^1\). By putting these methods in an interface, it also eases the burden on us, the users of the JCF, since we only have to remember one set of method names and all the collections will follow the same basic structure. Collection also includes methods such as add(E) and remove(Object) that are specified as “optional”! The reason for this is that the JCF would like all collections to use the same method signatures for these methods as well, but also allows the creation of unmodifiable collections. That is, we are allowed to create a collection class that does not implement these methods — or, to be more accurate, implements them by doing nothing except throwing an exception, since the compiler will not just let us ignore them and still claim to implement Collection.

OK, now let’s take a look at the class diagram of the JCF and move on to some other concrete classes that we would like to use. Note that this document does not explain all (or even a majority) of the methods available even in the common classes; it explains some of the most critical ones, but you should consult the Javadocs for more details.

---

\(^1\) There are many other methods in the interface, check out the Javadoc!
1.2 List classes

The JCF provides an interface List and two main concrete implementations, ArrayList and LinkedList. We have already seen these, and the concept of lists in general, so there is little surprising (and little left to learn!) about these classes.

As you may recall, although they appear identical from the outside, these two implementations have very different expected time complexity for basic operations. This exhibits both the positive and negative of a basic O-O design concept: encapsulation. Encapsulation refers to the idea that the implementation of a class can and should be hidden from the users of that class — only the interface (whether a Java interface or simply the public methods of the class) needs to be exposed. The positive is that as a developer, you can modify the implementation without worrying about your users, and that as a user, you often do not care about the implementation. The problem is that sometimes you do care, and so the List interface is careful to explain how its methods are expected to behave. For example, not only is a list expected to remain in a particular order, but List.add(E) promises to add to the end of the list. This is not required by the Java compiler, merely a contract between the class developers and users. Beyond simple behaviors, the specific implementing classes provide their own contracts. LinkedList methods to add and remove from either end of the list will operate in O(1) time, while removal from the middle is O(n). For ArrayList, the Javadoc is more specific yet: “The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.”

Because both classes implement the List interface, it is possible (and often desirable) to declare your variables of type List rather than one specific type, for example:

\[
\text{List\<Student\>}\ \text{myClass} = \text{new LinkedList\<Student\>()};
\]

This way, Java will only allow us to call List methods on myClass, so if we decide later to change this to an ArrayList, we can do so without changing any of the method calls. We are then open to using new implementations later on, for example if we notice that our usage pattern would be more suited to a different implementation. Also, you may note in the class diagram that in fact the classes both inherit from an abstract class AbstractList. This includes some simple implementation of iteration and the like that may be useful for any concrete implementation (note that many non-abstract methods in this class simply throw an Exception!).

1.3 Set classes

The other major type of data structure that implements the Collection interface is the set. A set is a data structure that contains a bunch of items and does not allow duplicates (much as a mathematical set). Since Java’s sets are collections, they implement addition, removal, contains(), as well as various constructors. Also, like all collections, they are defined with a generic type parameter, so you might have a declaration like

\[
\text{Set\<Vehicle\>}\ \text{myGarage} = \text{new HashSet\<Vehicle\>()};
\]
Unlike Python, the JCF provides two very different implementations of the set concept. In fact, these are different enough that in addition to the Set interface (a sub-interface of Collection) that they both implement, there is a separate interface for one of the implementations. This interface is called SortedSet, which may give you an idea what its implementation does!

The two main concrete set classes in the JCF are HashSet and TreeSet. HashSet uses an underlying hash table to allow constant-time addition, removal and membership checking in the set. However, as you may recall about hash tables, the order of the items in the set is arbitrary (or rather, the set itself decides how to order them). The TreeSet, which also implements the SortedSet interface, keeps the items within in their natural order. That is, numbers will be in numerical order, strings in alphabetical order, and so on. This is all managed by the TreeSet itself (how do you think it does this?), so that again, you can simply call the add, remove and contains (or other) methods common to all collections. Note that the SortedSet interface also includes methods such as first() which returns the smallest element in the set, last() which returns the largest element, and subSet() which gives you the elements between two specified values. Clearly a HashSet cannot do any of these in an efficient manner, so it does not try to implement this interface.

1.4 Map classes

The last major data structures available in the JCF are maps, which are equivalent in concept to the dictionary in Python. Since maps have both keys and values, they are defined in Java with two generic type parameters, for example:

```java
Map<String,Integer> phoneBook = new HashMap<String,Integer>();
```

Much like sets (in fact, maps and sets use much of the same underlying code base in the JCF, good design idea!), there are two types of maps, sorted and unsorted. The implementation of an unsorted map is in the class HashMap, while the implementation of a sorted map (and the SortedMap interface) is in the class TreeMap. In both cases, the syntax for adding elements and looking up keys is the same. To add a key, value pair, call put(key, value). Note that if the key was already present in the map, this will replace the old value with the new one, but it will also return the old value! To look something up in the map, call get(key), which returns a value or null if the value is not in the map. Note that null could be the value for a key that is in the map, so you can also use containsKey(key) to find out if a key is present in the map.

Finally, note that Map does not inherit from Collection. This is because (among other reasons) Collection has a single generic type whereas Map has two. However, the keys of a map (obtainable via the function Map.keySet() ) are a Set, and therefore a Collection! However, the designers of the JCF did use many of the same method names in Map, such as size(), clear() and isEmpty() to make things easier to remember.

1.5 The Collections class — not to be confused with the Collection interface!

Along with common data structures, the JCF also provides common algorithms that are used with these data structures, such as sorting. However, rather than putting these functions
scattered (or repeated!) across the JCF, they are collected in one class. The Collections class cannot be instantiated; that is, you cannot construct objects of this class. Rather, you simply call the static methods and pass in a relevant collection. Note that some methods in Collections operate on any collection, while others require a list or a map.

For example, sort(List<T extends Comparable<T>) takes a list, and destructively sorts it (that is, it sorts the list you give it rather than making a new list. Note that the List you pass in must contain Comparable objects. reverse() and shuffle() also only take lists, which makes sense in that sets either do not have a defined order, or are maintained in a sorted order. On the other hand, max() and min() take any Collection of comparable objects and return the largest (or smallest) element in the collection. Again, other algorithms are available, if you think the algorithm you need might have already been written in the JCF, check the Javadoc before implementing it yourself.

2 O-O design principles

Along the way, we have seen some various principles of object-oriented software design. There are many more that practitioners advocate. Some of these are uncontroversial, though always good to keep in mind, while others are not so cut-and-dried.

Some principles to keep in mind:

• Make sure your function and parameter names are meaningful.

• Follow a consistent coding style.

• Program to the interface (when applicable).

• All public methods of a class should leave the object in a well-formed state.

• Just because two classes have similar sets of methods does not mean they should share an interface; they should also have similar behavior within their common methods.

• Use interfaces and inheritance with polymorphism to avoid unnecessary conditionals (if statements).

Some things to consider both sides of:

• Cache or recompute data? If a method of a class returns data that can be computed from other instance data (e.g. the size of a linked list), should it be recomputed whenever the method is called (might be slow) or stored in a separate instance variable (and must be updated in each other method that might affect it)?

• When you have subclasses of a particular class, what does it mean for a subclass object and a superclass object to be equal? Are they equal if the subclass object has all the same instance data as the superclass object? What happens when you call super.equals(sub)?

• Functions (those that return a value) should not modify state; those that modify state should not also return a value. Is this always obeyed in the JCF? If not, are there good reasons for this?