Chapter 7: Collections

A collection is an object that represents a group of objects. For example, a stack is a collection, as is a set or a vector. Java defines a `Collection` interface that defines the operations that any collection should offer, including operations to add and remove items, and to find out the number of items in the collection.

This interface is very general. For instance, it doesn't specify the order in which items are removed or whether duplicates are allowed. Java defines more specific interfaces that specify these things more fully and add more methods. These include interfaces `Queue`, `List` and `Set`.

7.1. Implementations

So far, we have only discussed interfaces. Each interface has several different implementations, with different characteristics. For example, `HashSet` implements the `Set` interface with a hash table. This allows it to offer constant-time performance for the core methods `add`, `remove`, `size` and `contains`, however, it cannot guarantee the order in which set elements will be seen if you iterate over them. On the other hand, `TreeSet` implements the interface with a balanced tree. As a result, it can make guarantees about order, but it can guarantee only that the core methods will run in logarithmic time.

When you choose an implementation of any kind of collection, you can read the documentation to find out the properties of the various options.

7.2. The Java Collections Framework

We refer to the various interfaces, their implementations, provided static methods that operate on collections, and some additional infrastructure, together as the "Java Collections Framework".

Let's look at a few of the implementations to get a feel for them.

7.2.1. List

Java's `List` is similar to Python's `list` type, in that they grow and shrink as needed. `ArrayList` is one implementation. It offers quick access to elements by index.

```
public static void main(String[] args) {
    // Declare and initialize an ArrayList of Strings:
    ArrayList<String> csc207team = new ArrayList<>();

    csc207team.add("Juanita");
    csc207team.add("Amelie");
    csc207team.add("Abhinav");
    csc207team.add("Menghan");

    // ArrayList has a toString that prints the list nicely.
    System.out.println(csc207team);

    // This isn't permitted, because we specified the elements would
    // be of type String:
```

```
        // csc207team.add(5);

        // We don't need to typecast. The compiler knows that get will return
        // a String, since this is an ArrayList of Strings.
        csc207team.get(0).charAt(0);

        // Some other useful methods. ArrayList has many more: check the
        // documentation!
        System.out.println(csc207team.size());
        System.out.println("Is alex in csc207team? " +
        csc207team.contains("alex"));

        // We can have an ArrayList of any valid Java object type, i.e.,
        // any built-in or user-defined object. But no primitives, so below
    won't work!
        // ArrayList<int> wontWork = new ArrayList<int>();

        // But we can get primitives into an ArrayList by using a wrapper
        // class.
        List<Integer> intList = new ArrayList<>();

        // And we can use autoboxing to avoid having to construct instances
        // of the wrapper class. So rather than say:
        intList.add(new Integer(42));
        // ... we can say just:
        intList.add(42);
        System.out.println(intList);

        // We can also automatically unbox. To get an Integer object from
        // the ArrayList, and get an int from it, we can just say:
        int x = intList.get(0);
        System.out.println(x);
    }
```

All elements of a `List` must be objects, not primitives, and they must be of the same type. However, we can use inheritance to allow mixed types. For instance, if we specify just that all elements are of type `Object` we are placing no restriction at all.

7.2.2. Set

A `Set`, as one would expect, does not allow duplicate elements, and has no notion of elements being in any particular position. `TreeSet` is one implementation.

```
    public static void main(String[] args) {

        // Declare a TreeSet of Strings, and try to add some elements.
        TreeSet<String> s = new TreeSet<>();
        s.add("hello");
        s.add("silly");
        // We can look at the return value of add to see if the operation
```

```
        // succeeded.
        System.out.println(s.add("goodbye!"));
        // We won't be able to add this String a second or third time.
        System.out.println(s.add("silly"));
        System.out.println(s.add("silly"));

        // TreeSet has a toString that prints the set nicely.
        // The elements of the set could come out in any order.
        System.out.println(s);

        // TreeSet implements the Iterable interface, which guarantees that
        // it provides a hasNext and a next method. Here we use it to iterate
        // over our set and assemble a single String with all the words.
        String allWords = "";
        Iterator<String> it = s.iterator();
        while (it.hasNext()) {
            allWords += it.next();
        }
        System.out.println(allWords);

        // Because Treeset implements Iterable, we can instead use an
        // enhanced for-loop.  Much more concise!
        allWords = "";
        for(String word: s) {
            allWords += word;
        }
        System.out.println(allWords);
    }
```

7.2.3. Map

The `Map` interface is similar to Python's `dict` type. As in Python, these map keys to values, and each key can have only one value associated with it. Unlike in Python, keys do not have to be immutable. However, you must really know what you are doing if you mutate an object after it has been added as a key, otherwise, the behaviour of your `Map` will be undefined. See the documentation for further details.

`HashMap` is one implementation of `Map`, based on — you guessed it — a hash table!

```
    public static void main(String[] args) {
        // Declare and initialize a Map from Strings to Integers.
        // Notice that the generic type HashMap requires two arguments:
        // one for the type of the keys, and one for the type of the values.
        Map<String, Integer> myMap = new HashMap<>();

        // We can use autoboxing to get an Integer value into the HashMap.
        myMap.put("csc", 207);
        myMap.put("bio", 120);
        myMap.put("his", 150);
        myMap.put("ant", 100);
        System.out.println(myMap.get("csc"));
```

```
        // "mat" is not a key, so this returns null.
        System.out.println(myMap.get("mat"));

        // HashMap has many other methods -- see the documentation for
        // details.
    }
```

`HashMap` offers constant-time performance for the basic operations (`get` and `put`), which is fantastic. However, this depends upon certain properties of the hash table being maintained as key-value pairs are added and removed. When you construct your `HashMap`, you can control parameters of the data structure that may help ensure these properties are maintained. You'll learn all about hash tables in CSC263, and then will be well-equipped to make good choices here. Something to look forward to!