# Map Notes

A Map is an interface that uses a key to point to values. The two classes we will study using the Map interface is TreeMap and HashMap. There are many other implementing classes but we will concentrate on these two classes. Deep down, a map is a natural evolution of parallel arrays. The Map has a key set which is composed of the Set interface. This is one of the ways we distinguish a TreeMap from a HashMap. The TreeMap uses the TreeSet and the HashMap uses the HashSet. Each key in the set is attributed a corresponding value. Because the key is a Set, then every key has to be unique. The values that correspond to the key, however, does not have to be unique. The values could be any Object. It can be an Integer, it can be a String, it can be and ArrayList, or it can be a Set or any other object type.

Create a file called ExampleTrees and write the following:

**import** java.util.Map;

**import** java.util.TreeMap;

**import** java.util.HashMap;

**import** java.util.\*;

**import** **static** java.lang.System.\*;

**public** **class** ExampleTrees {

**public** **static** **void** main(String args[])

{

Scanner input = **new** Scanner(System.***in***);

//Map<String,String> x = new Map<String,String>();

TreeMap<Integer,String> y = **new** TreeMap<Integer,String>();

HashMap<String,ArrayList<Integer>> z =

**new** HashMap<String, ArrayList<Integer>>();

}

}

In the opening example, the programmer needs to import the Map class from java.util in order to use Map. The programmer needs to import the TreeMap class and the HashMap class from java.util in order to use the respective classes: TreeMap and HashMap. When the programmer has to declare a Map, it must be instantiated as a TreeMap or a HashMap because the Map interface does not have a constructor. It would be fine to declare y or z as a Map as long as the instance was created as either a TreeMap or a HashMap. Notice that the generics should be declared. If the programmer doesn’t declare what the generics are, then the program will assume <Object, Object>. When the Map is declared, the first generic is a key and the second generic is a value <key, value>. The key generic will automatically be created as a TreeSet or a HashSet, depending on the map. But the value may need to be instantiated, especially if the value is a Collection.

Add the following to the code:

TreeMap<Integer, Integer> map1 = **new** TreeMap<Integer,Integer>();

Map<String, Set<String>> map2 =

**new** TreeMap<String, Set<String>>();

z.put("Arsenal", **new** ArrayList<Integer>());

map1.put(1, 2);

map2.put("Houston", **new** HashSet<String>());

***out***.println(z);

***out***.println(map1);

***out***.println(map2);

The output would be

{Arsenal=[]}

{1=2}

{Houston=[]}

Several things occur in this example. First of all, map does not have an add method, but rather has a put method

void put(K key, V Value)

where K and V are the data types in the declared Map<K, V>. There is also another useful method called putAll which will copy a map onto another map.

void putAll(Map<? extends K, ? extends V> m)

and another put method that will add a linked K, V as long as that K does not already exist in the current map.

void putIfAbsent(K key, V Value)

Furthermore, please noticed that the value will not be instantiated unless the programmer instantiates it themselves. So for the map z, there is no automatic ArrayList. The programmer will have to create an ArrayList to place into the value, or use an already created ArrayList to place into the value.

Map has a toString() associated to it, as all objects must. But this toString is overloaded to list each key to value relationship as {key = value, key = value, …, key = value}.

Add the following to the code:

String k = "Cowboys,Rangers,Stars,Cowboys,Mavericks,Mustangs,Texans";

String city = "Dallas";

String list[] = k.split(",");

**for**(String x:list)

{

**if**(!map2.containsKey(city))

map2.put(city,**new** HashSet<String>());

Set<String> temp = map2.get(city);

temp.add(x);

}

***out***.println(map2);

The output would be

{Arsenal=[]}

{1=2}

{Houston=[]}

{Dallas=[Texans, Mustangs, Cowboys, Stars, Mavericks, Rangers], Houston=[]}

Here as a good example of how to add everything into a map for one key. Looking at the code, it is the programmer’s intention to place every value of k into map2 under Dallas. At the time, there is no key in map2 called “Dallas”. Sometimes, a programmer is not sure there is an associated key or if that key is pointing to null. So the programmer writes an if statement seeing if the key is contained in it, if it is not, then the programmer places a key of that value into map2 and creates an instance of a the HashSet associated with that key. Another thing the programmer could have done, was to see if the key pointing to a null in the map, and if it was, then the programmer would have created a HashSet instance and associate it with the key in the map. Notice that once the HashSet is created, the programmer calls the set using get and it returns the actual HashSet. Now the programmer can alter the HashSet all they want to, because it is the actual value and will permanently change the value in map2.

In these examples we saw

Boolean containsKey(Object key)

This method will check to see if the key exists in the Set. We also saw

V get(Object key)

This method will return the value associated with the key. It will return a null if the key is not mapped. There are other methods that can be very useful for the programmer.

void clear() will remove all the mapping in the Map.

Boolean containsValue(Object value) will return true if the map has the value mapped in the map. Since this is the value portion of the map, the value could appear multiple times in the map.

Boolean equals(Object o) can compare one map to another map.

Boolean isEmpty() which will return if there is any mapping in the map at all.

Set<K> keySet() which is an important method. This method returns the Set of keys so that the programmer could use an iterator to go to every element in the key of the map.

V remove(Object key) which removes the key from the keyset and returns the associated value.

Boolean remove(Object key, Object value) which will that specific combination of key and value, do not confuse this with finding a key that has a value inside of a mapped Collection. This is the specific removal of a key and object only if they are uniquely mapped together in the map.

Boolean replace(K key, V oldValue, V newValue) which will only replace the mapped value of key if it is the same as the oldValue listed, otherwise no replacement occurs.

int size() will return the size of the Set

Collection<V> values() will return a Collection of the values in the map.

Add the following to the code:

***out***.println(map2);

z.put("Liverpool", *nullStanding*());

z.put("Manchester City", *nullStanding*());

z.put("Arsenal", *nullStanding*());

***out***.println(z);

ArrayList<Integer> temp = z.get("Liverpool");

temp.set(0, 8);

temp = z.get("Manchester City");

temp.set(0, 6);

temp.set(1, 1);

temp.set(2, 1);

temp = z.get("Arsenal");

temp.set(0, 5);

temp.set(1, 0);

temp.set(2, 3);

***out***.println(z);

temp = z.get("Arsenal");

temp.set(0, temp.get(0)+1);

temp = z.get("Liverpool");

temp.set(2, temp.get(2)+1);

***out***.println(z);

}

**public** **static** ArrayList<Integer> nullStanding()

{

ArrayList<Integer> temp = **new** ArrayList<>();

temp.add(0);

temp.add(0);

temp.add(0);

**return** temp;

}

}

In this final scenario, I wrote a rudimentary map that would hold the win-tie-loss record of the premier league table. Please note, that when I use a Collection for the value set, then all I need to do is get the Collection from the map and any manipulation that I do will change the value set as long as I don’t dereference away from the Collection. If I had a value composed of a Non-Collection, then I would have had to have put the value I created back into the map, effectively erasing the mapping of the current key.