**Why two Interface Map and Collection? Why not a single one**

If you look at the respective data structure you can easily guess why Map is not a part of Collection. Each Collection stores a single value where as a Map stores key-value pair. So methods in Collection interface are incompatible for Map interface. For example in Collection we have add(Object o). What would be such implementation in Map. It doesn't make sense to have such a method in Map. Instead we have a put(key,value) method in Map.

Same argument goes for addAll(), remove(), and removeAll() methods. So the main reason is the difference in the way data is stored in Map and Collection. Also if you recall Collection interface implemented Iterable interface i.e. any interface with .iterator() method should return an iterator which must allow us to iterate over the values stored in the Collection. Now what would such method return for a Map? Key iterator or a Value iterator? This does not make sense either.

There are ways in which we can iterate over keys and values stores in a Map and that is how it is a part of Collection framework

**Why doesn't Map extend Collection?**

**This was by design**. We feel that mappings are not collections and collections are not mappings. Thus, it makes little sense for Map to extend the Collection interface (or vice versa).

If a Map is a Collection, what are the elements? The only reasonable answer is "Key-value pairs", but this provides a very limited (and not particularly useful) Map abstraction. You can't ask what value a given key maps to, nor can you delete the entry for a given key without knowing what value it maps to.

Collection could be made to extend Map, but this raises the question: what are the keys? There's no really satisfactory answer, and forcing one leads to an unnatural interface.

Maps can be viewed as Collections (of keys, values, or pairs), and this fact is reflected in the three "Collection view operations" on Maps (keySet, entrySet, and values). While it is, in principle, possible to view a List as a Map mapping indices to elements, this has the nasty property that deleting an element from the List changes the Key associated with every element before the deleted element. That's why we don't have a map view operation on Lists.

**Update:** I think the quote answers most of the questions. It's worth stressing the part about a collection of entries not being a particularly useful abstraction. For example:

Set<Map.Entry<String,String>>

would allow:

set.add(entry("hello", "world"));

set.add(entry("hello", "world 2");

(assuming an entry() method that creates a Map.Entry instance)

Maps require unique keys so this would violate this. Or if you impose unique keys on a Set of entries, it's not really a Set in the general sense. It's a Set with further restrictions.

Arguably you could say the equals()/hashCode() relationship for Map.Entry was purely on the key but even that has issues. More importantly, does it really add any value? You may find this abstraction breaks down once you start looking at the corner cases.

It's worth noting that the HashSet is actually implemented as a HashMap, not the other way around. This is purely an implementation detail but is interesting nonetheless.

The main reason for entrySet() to exist is to simplify traversal so you don't have to traverse the keys and then do a lookup of the key. Don't take it as prima facie evidence that a Map should be a Set of entries (imho).