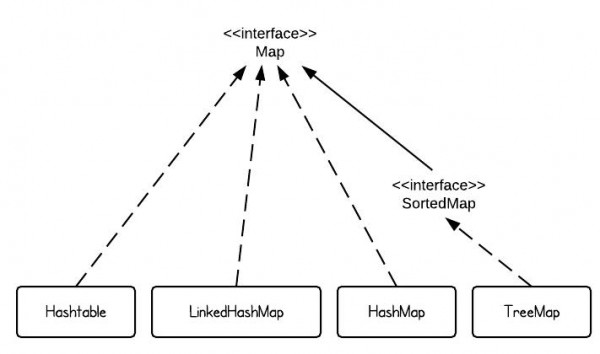
[HashMap vs. TreeMap vs. Hashtable vs. LinkedHashMap](http://www.programcreek.com/2013/03/hashmap-vs-treemap-vs-hashtable-vs-linkedhashmap/)

Map is one of the most important data structures in Java. In this post, I will illustrate how to use different types of maps, such as HashMap, TreeMap, HashTable and LinkedHashMap.

1. Map Overview



There are 4 commonly used implementations of Map in Java SE - HashMap, TreeMap, Hashtable and LinkedHashMap. If we use only one sentence to describe each implementation, it would be the following:

* HashMap is implemented as a hash table, and there is no ordering on keys or values.
* TreeMap is implemented based on red-black tree structure, and it is ordered by the key.
* LinkedHashMap preserves the insertion order
* Hashtable is synchronized, in contrast to HashMap. It has an overhead for synchronization.

This is the reason that HashMap should be used if the program is thread-safe.

2. HashMap

If the key of a HashMap is a self-defined object, then the equals() and hashCode() contract need to be followed.

|  |
| --- |
| **class** Dog {  String color;    Dog(String c) {  color = c;  }  **public** String toString(){  **return** color + " dog";  }  }    **public** **class** TestHashMap {  **public** **static** **void** main(String[] args) {  HashMap<Dog, Integer> hashMap = **new** HashMap<Dog, Integer>();  Dog d1 = **new** Dog("red");  Dog d2 = **new** Dog("black");  Dog d3 = **new** Dog("white");  Dog d4 = **new** Dog("white");    hashMap.put(d1, 10);  hashMap.put(d2, 15);  hashMap.put(d3, 5);  hashMap.put(d4, 20);    *//print size*  System.out.println(hashMap.size());    *//loop HashMap*  **for** (Entry<Dog, Integer> entry : hashMap.entrySet()) {  System.out.println(entry.getKey().toString() + " - " + entry.getValue());  }  }  } |

Output:

4

white dog - 5

black dog - 15

red dog - 10

white dog - 20

Note here, we add "white dogs" twice by mistake, but the HashMap accepts it. This does not make sense, because now we are confused with how many white dogs are really there.

The Dog class should be defined as follows:

|  |
| --- |
| **class** Dog {  String color;    Dog(String c) {  color = c;  }    **public** **boolean** equals(Object o) {  **return** ((Dog) o).color.equals(**this**.color);  }    **public** **int** hashCode() {  **return** color.length();  }    **public** String toString(){  **return** color + " dog";  }  } |

Now the output is:

3

red dog - 10

white dog - 20

black dog - 15

The reason is that HashMap doesn't allow two identical elements. By default, the hashCode() and equals() methods implemented in the Object class are used. The default hashCode() method gives distinct integers for distinct objects, and the equals() method only returns true when two references refer to the same object. Check out [the hashCode() and equals() contract](http://www.programcreek.com/2011/07/java-equals-and-hashcode-contract/) if this is not obvious to you.

Check out the [most frequently used methods for HashMap](http://www.programcreek.com/2013/04/frequently-used-methods-of-java-hashmap/), such as iteration, print, etc.

3. TreeMap

A TreeMap is sorted by keys. Let's first take a look at the following example to understand the "sorted by keys" idea.

|  |
| --- |
| **class** Dog {  String color;    Dog(String c) {  color = c;  }  **public** **boolean** equals(Object o) {  **return** ((Dog) o).color.equals(**this**.color);  }    **public** **int** hashCode() {  **return** color.length();  }  **public** String toString(){  **return** color + " dog";  }  }    **public** **class** TestTreeMap {  **public** **static** **void** main(String[] args) {  Dog d1 = **new** Dog("red");  Dog d2 = **new** Dog("black");  Dog d3 = **new** Dog("white");  Dog d4 = **new** Dog("white");    TreeMap<Dog, Integer> treeMap = **new** TreeMap<Dog, Integer>();  treeMap.put(d1, 10);  treeMap.put(d2, 15);  treeMap.put(d3, 5);  treeMap.put(d4, 20);    **for** (Entry<Dog, Integer> entry : treeMap.entrySet()) {  System.out.println(entry.getKey() + " - " + entry.getValue());  }  }  } |

Output:

Exception in thread "main" java.lang.ClassCastException: collection.Dog cannot be cast to java.lang.Comparable

at java.util.TreeMap.put(Unknown Source)

at collection.TestHashMap.main(TestHashMap.java:35)

Since TreeMaps are sorted by keys, the object for key has to be able to compare with each other, that's why it has to implement Comparable interface. For example, you use String as key, because String implements Comparable interface.

Let's change the Dog, and make it comparable.

|  |
| --- |
| **class** Dog **implements** Comparable<Dog>{  String color;  **int** size;    Dog(String c, **int** s) {  color = c;  size = s;  }    **public** String toString(){  **return** color + " dog";  }    @Override  **public** **int** compareTo(Dog o) {  **return** o.size - **this**.size;  }  }    **public** **class** TestTreeMap {  **public** **static** **void** main(String[] args) {  Dog d1 = **new** Dog("red", 30);  Dog d2 = **new** Dog("black", 20);  Dog d3 = **new** Dog("white", 10);  Dog d4 = **new** Dog("white", 10);    TreeMap<Dog, Integer> treeMap = **new** TreeMap<Dog, Integer>();  treeMap.put(d1, 10);  treeMap.put(d2, 15);  treeMap.put(d3, 5);  treeMap.put(d4, 20);    **for** (Entry<Dog, Integer> entry : treeMap.entrySet()) {  System.out.println(entry.getKey() + " - " + entry.getValue());  }  }  } |

Output:

red dog - 10

black dog - 15

white dog - 20

It is sorted by key, i.e., dog size in this case.

If "Dog d4 = new Dog("white", 10);" is replaced with "Dog d4 = new Dog("white", 40);", the output would be:

white dog - 20

red dog - 10

black dog - 15

white dog - 5

The reason is that TreeMap now uses compareTo() method to compare keys. Different sizes make different dogs!

4. Hashtable

From Java Doc:  
The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.

5. LinkedHashMap

LinkedHashMap is a subclass of HashMap. That means it inherits the features of HashMap. In addition, the linked list preserves the insertion-order.

Let's replace the HashMap with LinkedHashMap using the same code used for HashMap.

|  |
| --- |
| **class** Dog {  String color;    Dog(String c) {  color = c;  }    **public** **boolean** equals(Object o) {  **return** ((Dog) o).color.equals(**this**.color);  }    **public** **int** hashCode() {  **return** color.length();  }    **public** String toString(){  **return** color + " dog";  }  }    **public** **class** TestHashMap {  **public** **static** **void** main(String[] args) {    Dog d1 = **new** Dog("red");  Dog d2 = **new** Dog("black");  Dog d3 = **new** Dog("white");  Dog d4 = **new** Dog("white");    LinkedHashMap<Dog, Integer> linkedHashMap = **new** LinkedHashMap<Dog, Integer>();  linkedHashMap.put(d1, 10);  linkedHashMap.put(d2, 15);  linkedHashMap.put(d3, 5);  linkedHashMap.put(d4, 20);    **for** (Entry<Dog, Integer> entry : linkedHashMap.entrySet()) {  System.out.println(entry.getKey() + " - " + entry.getValue());  }  }  } |

Output is:

red dog - 10

black dog - 15

white dog - 20

The difference is that if we use HashMap the output could be the following - the insertion order is not preserved.

red dog - 10

white dog - 20

black dog - 15