Though all three classes implement java.util.Map interface and follows general contract of a Map interface, defined in terms of [equals() and hashCode()](http://javarevisited.blogspot.sg/2015/01/why-override-equals-hashcode-or-tostring-java.html) method, they also have several differences in terms of Ordering, Sorting, permitting null elements, Iteration, Performance, Speed and internal implementation. Let's have a quick look on each of these properties.

**Ordering and Sorting**

HashMap doesn't provide any ordering guarantee for entries, which means, you can not assume any order while [iterating over keys and values of HashMap](http://java67.blogspot.sg/2013/08/best-way-to-iterate-over-each-entry-in.html). This behavior of HashMap is similar to Hashtable while other two Map implementation provides ordering guarantee.  
  
LinkedHashMap can be used to maintain insertion order, on which keys are inserted into Map or it can also be used to maintain an access order, on which keys are accessed. This provides LinkedHashMap an edge over HashMap without compromising too much performance.  
  
TreeMap provides you complete control over sorting elements by passing [custom Comparator](http://javarevisited.blogspot.sg/2014/01/java-comparator-example-for-custom.html) of your choice, but with the expense of some performance. Since entries are stored in a tree-based data structure, it provides lower performance than HashMap and LinkedHashMap.

**Null keys and Values**

HashMap allows one null key and multiple null values. It keeps null key based entries on index[0] on an internal bucket. If you look at the put() method of HashMap, you can see, it doesn't throw [NullPointerException for null keys](http://javarevisited.blogspot.sg/2012/06/common-cause-of-javalangnullpointerexce.html). Since LinkedHashMap is a subclass of HashMap, it also allows null keys and values.  
  
On the other hand, TreeMap, which sorts elements in natural order doesn't allow null keys because compareTo() method throws NullPointerException if compared with null. If you are using TreeMap with [user defined Comparator](http://java67.blogspot.sg/2014/11/java-8-comparator-example-using-lambda-expression.html) than it depends upon the implementation of compare() method.

**Iterators**

Iterators returned by all these Map's collection view methods e.g. values() or keySet() is [fail-fast iterators](http://java67.blogspot.sg/2015/06/what-is-fail-safe-and-fail-fast-iterator-in-java.html), which means they will throw ConcurrentModificatoinException if Collection is modified structurally once Iteration begins, except by using remove() method of Iterator.  
  
By the way, it's worth remembering that apart from adding or removing more mappings, it can also be any operation which affects iteration order of LinkedHashMap. In access-ordered LinkedHashMap, even querying the Map with get() method is a structural modification, because it changes the iteration order, on the other hand updating the value in an insertion-ordered linked hash map is not a structural modification.  
  
Finally, the fail-fast behavior is not guaranteed, and they throw [ConcurrentModificationException](http://java67.blogspot.com/2015/10/how-to-solve-concurrentmodificationexception-in-java-arraylist.html) on the best-effort basis, which means do not write code, which depends upon this behavior. It should only be used to detect programming bugs.

**Performance and Speed**

Since HashMap is a barebone implementation of java.util.Map interface, it provides constant time performance for the get() and put() operation, where **put()** method is used to store entries (key-value pairs) and **get()** is used to retrieve a value based on a key.  
  
BTW, constant time performance is only provided if mappings are distributed uniformly across bucket location. In the real world, you always have collision and HashMap handles collision by using a linked list to store collided elements. This can reduce worst case performance of HashMap up to O(n).  
  
To mitigate the above performance issue, JDK 8 has introduced balanced tree instead of linked list in case of frequent collision in HashMap. It internally switches to balanced tree from linked list if there are more than 8 entries in one bucket. See [how does HashMap handles collisions in Java](http://javarevisited.blogspot.com/2016/01/how-does-java-hashmap-or-linkedhahsmap-handles.html) for more details.  
  
Worth noting is that this behavior is only applicable to HashMap, LinkedHashMap, and ConcurrentHashMap, Hashtable is left behind to preserve its legacy iteration order as many legacy Java application relies on that and this changes that order. This is also a good example of why you should not rely on undocumented features of JDK e.g. iteration order of HashMap because they can change in future.  
  
but HashMap is certainly faster than Hashtable because it's not synchronized. [Iteration over Map](http://java67.blogspot.sg/2014/05/3-examples-to-loop-map-in-java-foreach.html) is directly proportional to the "capacity" + "size" of HashMap, that's why it's important to set the initial capacity high enough if iteration performance is important. You can further use **initial capacity** and **load factor** to fine tune your HashMap performance, to avoid rehashing of HashMap.  
  
TreeMap is  so it's costlier than HashMap if the order is not concerned. Since TreeMap is based on tree data structure (based upon Red-Black tree), it provides the log(n) time for the get(), put(), containsKey() and remove() operation, Algorithms are based upon those given in Cormen, Leiserson, and Rivest's [*Introduction to Algorithms*](http://www.amazon.com/Introduction-Algorithms-Edition-Thomas-Cormen/dp/0262033844?tag=javamysqlanta-20).

LinkedHashMap is a trade-off between two, like HashMap it also provides constant time performance for add, contains and remove, though it's slightly slower than HashMap, to maintain linked list. By the way, looping over Map in the case of LinkedHashMap is slightly faster than HashMap because the time required is proportional to size only. So if you need insertion order or *access order*, consider using LinkedHashMap over TreeMap in Java.

**Thread-safety and Synchronization**

All three Map implementation are [not thread-safe](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html), which means you can not use them safely in a multi-threaded application. Though you can synchronize them externally by using Collections.synchronizedMap(Map map) method. Alternatively, you can also use their concurrent counterpart e.g. ConcurrentHashMap which is also a better choice than HashMap in a concurrent Java application.  
  
When using synchronized Map e.g. synchronized LinkedHashMap or SortedMap, you must do at the time or creating the map to prevent accidental non-synchronized access. You can use the following idiom to create Synchronized Map in Java:  
  
**Synchronized LinkedHashMap**

Map<Integer, Integer> numbers **=** Collections**.**synchronizedMap(**new** LinkedHashMap<>());

**Synchronized TreeMap**

SortedMap<Integer, String> sorted **=** Collections**.**synchronizedSortedMap(**new** TreeMap<>());

Remember to use Collections.synchronizedMap() for synchronizing HashMap, LinkedHashMap and Collections.synchronizedSortedMap() method for synchronizing TreeMap. If you are not comfortable then see this guide on [how to synchronize HashMap in Java](http://java67.blogspot.sg/2015/02/how-to-synchronize-hashmap-in-java-with.html).

**Internal Implementation**

TreeMap is Red-Black tree based NavigableMap implementation while HashMap is internally backed by an array. It uses index[0] to store entries corresponding to null keys. In fact, questions related to the inner working of HashMap is very popular in Java, for example, [How does get() method of HashMap works internally](http://java67.blogspot.sg/2013/06/how-get-method-of-hashmap-or-hashtable-works-internally.html) is one of the frequently used questions to Senior Java developers.  
  
On the other hand, LinkedHashMap extends HashMap and uses linked list to provide insertion order guarantee. It uses doubly-linked list running through all of its entries, which can also be used to maintain access-order. Remember, insertion order is not affected if a key is re-inserted into LinkedHashMap, but access order is affected if LinkedHashMap is created to maintain access-order.  
  
TreeMap is internally based upon Red-Black Tree and NavigableMap, introduced in JDK 6. The Red-Black tree is used to maintain the sorting order imposed by Comparable or Comparator, provided at the time of creation.  TreeMap provides guaranteed log(n) time cost for the get, put, containsKey and remove operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's [*Introduction to Algorithms*](http://www.amazon.com/Introduction-Algorithms-Edition-Thomas-Cormen/dp/0262033844?tag=javamysqlanta-20).

**When to use LinkedHashMap, TreeMap, and HashMap**

You can use a LinkedHashMap when you need to keep your mappings in either **insertion order** or **access order**. LinkedHashMap by default keeps elements in the order, on which they are inserted, and this order is reflected when you [traverse over LinkedHashMap](http://javarevisited.blogspot.sg/2011/12/how-to-traverse-or-loop-hashmap-in-java.html), but it also provides a constructor, which allows you to keep entries in *access order*, the. order in which they are accessed. One of the clever use of Java LinkedHashMap is to use it as Least Recently Use or **LRU Cache**.  
  
**TreeMap** is your go to map implementation if you want to keep keys  in a sorted order, either in their natural order defined by Comparable interface or a custom order imposed by Comparator interface, though it's worth remembering that your compareTo() or compare() method must be [consistent with equals() method](http://java67.blogspot.sg/2013/04/example-of-overriding-equals-hashcode-compareTo-java-method.html), because Map interface is defined in terms of equals and TreeMap uses compareTo for comparing keys. So if keys compare() or compareTo() implementation is not consistent, then it will fail to obey Map's general contract.  
  
**HashMap** is your general purpose hashing based collection, whenever you need to use a hash table data structure in Java to store key-value pairs, the first choice goes to HashMap in a single threaded environment. If you happened to use a Map in a multi-threaded environment consider using [Hashtable, synchronized HashMap or ConcurrentHashMap](http://javarevisited.blogspot.sg/2011/04/difference-between-concurrenthashmap.html) from Java Collection Framework.  
  
Since LinkedHashMap solved the problem of chaotic ordering provided by Hashtable and HashMap, without incurring the high cost associated with TreeMap, you can also use LinkedHashMap to create a copy of a Map in Java, as shown in below example.

**An example of using LinkedHashMap, TreeMap and HashMap in Java**

Let's see an example of how to use these Map implementations. In this example, we will use HashMap to create a general purpose Cache, TreeMap to create a sorted Cache and we will use LinkedHashMap for copying a Map (cache) and maintaining orders in the original Map.

import java.util.Collections;

import java.util.HashMap;

import java.util.LinkedHashMap;

import java.util.Map;

import java.util.SortedMap;

import java.util.TreeMap;

/\*\*

\* Java Program to demonstrate How to use LinkedHashMap, TreeMap and HashMap.

\* It shows that HashMap doesn't guarantee any order, TreeMap keeps them in

\* sorted order determined by default using Comparable or explicit Comparator

\* provided by client, and LinkedHashMap also keep mapping in order they

\* are added or accessed.,

\*

\* @author Javin Paul

\*/

public class MapTest {

public static void main(String args[]){

//Using HashMap as general purpose single threaded cache

Map<Integer, String> cache = new HashMap<>();

cache.put(1, "Stuart");

cache.put(2, "Steven");

cache.put(3, "James");

cache.put(4, "Ian");

System.out.printf("Name of Employee with id %d is %s %n", 1, cache.get(1));

System.out.println("Order of Entries in HashMap - Not guaranteed");

System.out.println(cache);

//Using TreeMap to create a sorted cache, sorting keys on reverse order

SortedMap<Integer, String> sortedCache = new TreeMap<>(Collections.reverseOrder());

sortedCache.putAll(cache);

System.out.println("Order of Entries in TreeMap - Sorted in reverse order");

System.out.println(sortedCache);

//Using LinkedHashMap to create copy of a Map in Java

Map<Integer, String> copy = new LinkedHashMap<>(sortedCache);

System.out.println("Order of Entries in a copy Map created by LinkedHashMap");

System.out.println(copy);

}

}

**Output:**

Name of Employee with id 1 is Stuart

Order of Entries in HashMap - Not guaranteed

{1=Stuart, 2=Steven, 3=James, 4=Ian}

Order of Entries in TreeMap - Sorted in reverse order

{4=Ian, 3=James, 2=Steven, 1=Stuart}

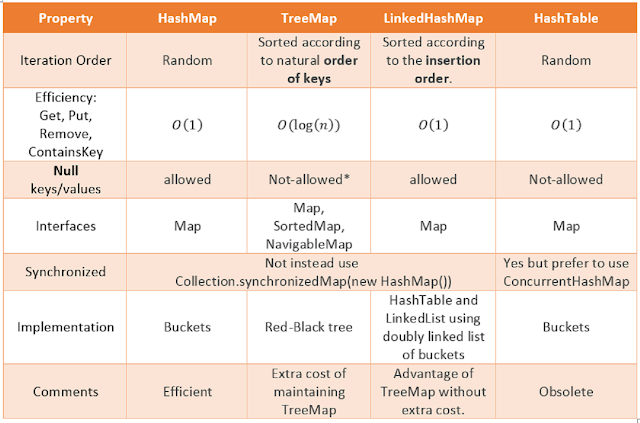
Order of Entries in a copy Map created by LinkedHashMap

{4=Ian, 3=James, 2=Steven, 1=Stuart}

You can see that TreeMap has sorted mappings in reverse order, because of reverse Comparator provided to it. Also, LinkedHashMap has created a copy of TreeMap and order of entries are retained.

**Summary**

Here is the summary of differences between HashMap, LinkedHashMap, and TreeMap in Java:

[](https://2.bp.blogspot.com/--9mLb1kGbjU/VdsKlyusXwI/AAAAAAAADo0/tYU-8u-qLmU/s1600/Difference%2Bbetween%2BHashMap%252C%2BTreeMap%252C%2BLinkedHashMap%2Band%2Bhashtable%2Bin%2BJava.png)

That's all on the **difference between LinkedHashMap, TreeMap, and HashMap in Java**. Though all three are Map implementation, they have a different purpose and used accordingly. Use LinkedHashMap, if you need to maintain insertion or access order of mappings e.g. in LRU Cache. Use TreeMap, if you need to maintain mappings in a sorted order, either in their natural order or a custom order defined by Comparator and use HashMap for all your general purpose hashing based collection requirement. HashMap allows you to retrieve an object in O(1) time if you know the key.