

**What is difference between HashMap and ConcurrentHashMap?**

**Answer: -**

* HashMap is not Thread-safe where as ConcurrentHashMap is Thread-safe.
* In HashMap, null values are allowed for key and values, whereas in ConcurrentHashMap null value is not allowed for key and value, otherwise we will get Run-time exception saying **NullPointerException**.
* While one thread is Iterating the HashMap object, if other thread tries to add/modify the contents of Object then we will get Run-time exception saying **ConcurrentModificationException**. Whereas in ConcurrentHashMap we won't get any exception while performing any modification at the time of Iteration.
* HashMap performance is relatively high because it is non-synchronized in nature and any number of threads can perform simultaneously. But ConcurrentHashMap performance is low sometimes because sometimes Threads are required to wait on ConcurrentHashMap.
* HashMap can be synchronized by using **synchronizedMap(HashMap) method** .By using this method we get a HashMap object which is equivalent to the HashTable object . So, every modification is performed on Map is locked on Map object. ConcurrentHashMap synchronizes or locks on the certain portion of the Map. To optimize the performance of ConcurrentHashMap , Map is divided into different partitions depending upon the Concurrency level . So that we do not need to synchronize the whole Map Object.

**What’s the difference between ConcurrentHashMap and Collections.synchronizedMap(Map)?**

**Answer: -** The Map object is an associative container that store elements, formed by a combination of a uniquely identify key and a mapped value. If you have very highly concurrent application in which you may want to modify or read key value in different threads, then it’s ideal to use ConcurrentHashMap. Best example is Producer Consumer which handles concurrent read/write.

**So, what does the thread-safe Map means?**

If multiple threads access a HashMap concurrently, and at least one of the threads modifies the map structurally, it must be synchronized externally to avoid an inconsistent view of the contents. There are two ways we could synchronized HashMap

* Java Collections synchronizedMap() method
* Use ConcurrentHashMap

//Hashtable

Map<String, String> normalMap = new Hashtable<String, String>();

//synchronizedMap

synchronizedHashMap = Collections.synchronizedMap(new HashMap<String, String>());

//ConcurrentHashMap

concurrentHashMap = new ConcurrentHashMap<String, String>();

### **ConcurrentHashMap**

* You should use ConcurrentHashMap when you need very high concurrency in your project.
* It is thread safe without synchronizing the whole map.
* Reads can happen very fast while write is done with a lock.
* There is no locking at the object level.
* The [locking](https://crunchify.com/how-to-create-your-own-non-blocking-queue-in-java-same-as-evictingqueue/) is at a much finer granularity at a hashmap bucket level.
* ConcurrentHashMap doesn’t throw a ConcurrentModificationException if one thread tries to modify it while another is iterating over it.
* ConcurrentHashMap uses multitude of locks.

### **SynchronizedHashMap**

* Synchronization at [Object level](https://crunchify.com/java-hashmap-containskeyobject-key-and-containsvalueobject-value-check-if-key-exists-in-map/).
* Every read/write operation needs to acquire lock.
* Locking the entire collection is a performance overhead.
* This essentially gives access to only one thread to the entire map & blocks all the other threads.
* It may cause contention.
* SynchronizedHashMap returns Iterator, which fails-fast on concurrent modification.

**ConcurrentHashMap:** It allows concurrent access to the map. Part of the map called *Segment (internal data structure)* is only getting locked while adding or updating the map. So ConcurrentHashMap allows concurrent threads to read the value without locking at all. This data structure was introduced to improve performance.

**Concurrency-Level:** Defines the number which is an estimated number of concurrently updating threads. The implementation performs internal sizing to try to accommodate these many threads.

**Load-Factor:** It's a threshold, used to control resizing.

**Initial Capacity:** The implementation performs internal sizing to accommodate these many elements.

**What is difference between ConcurrentHashMap and ConcurrentSkipListMap?**

**Answer: -**

**ConcurrentHashMap:-** ConcurrentHashMap does not guarantee\* the runtime of its operations as part of its contract. It also allows tuning for certain load factors (roughly, the number of threads concurrently modifying it).

**ConcurrentSkipListMap : -**ConcurrentSkipListMap, on the other hand, guarantees average O(log(n)) performance on a wide variety of operations. It also does not support tuning for concurrency's sake. ConcurrentSkipListMap also has a number of operations that ConcurrentHashMap doesn't: ceilingEntry/Key, floorEntry/Key, etc. It also maintains a sort order, which would otherwise have to be calculated (at notable expense) if you were using a ConcurrentHashMap.

Basically, different implementations are provided for different use cases. If you need quick single key/value pair addition and quick single key lookup, use the HashMap. If you need faster in-order traversal, and can afford the extra cost for insertion, use the **SkipListMap**.

### What is difference between HashMap and TreeMap?

|  |  |
| --- | --- |
| **HashMap** | **TreeMap** |
| 1) HashMap can contain one null key. | TreeMap cannot contain any null key. |
| 2) HashMap maintains no order. | TreeMap maintains ascending order. |

### TreeMap vs HashMap

TreeMap and [HashMap](https://www.journaldev.com/11560/java-hashmap) both implements Map interface and part of collection framework. Let’s look at some of the differences between TreeMap vs HashMap.

1. TreeMap entries are sorted in natural ordering of keys whereas HashMap doesn’t store entries in any order.
2. TreeMap doesn’t allow null key whereas we can have one null key in HashMap.
3. Since TreeMap stores entries in sorted way, it’s a bit slower that HashMap in storing and retrieving objects.
4. TreeMap uses Red-Black tree based NavigableMap implementation whereas HashMap uses hashing algorithm implementation.
5. TreeMap implements NavigableMap, so you get some extra features that are not present in HashMap. For example – submap, first key, last key, head map, tail map etc.

### **When to use TreeMap in Java?**

**Answer: -**Most of the time HashMap will be enough to use as Map implementation in your program. But if you have some special requirements related to sorting, finding next lower and higher key, work on a submap then you can go for TreeMap.

**What is difference between LinkedHashMap and HashMap?**

**Answer: -** Main difference between HashMap and LinkedHashMap is that LinkedHashMap maintains insertion order of keys, order in which keys are inserted in to LinkedHashMap. On the other hand, HashMap doesn't maintain any order or keys or values.

In terms of Performance there is not much difference between HashMap and LinkedHashMap but yes LinkedHashMap has more memory foot print than HashMap to maintain doubly LinkedList which it uses to keep track of insertion order of keys.

LinkedHashMap actually extends HashMap and implements Map interface.

**What is difference between HashMap, LinkedHashMap and TreeMap?**

**Answer: -** All three classes implement the Map interface and offer mostly the same functionality. The most important difference is the order in which iteration through the entries will happen: -

* **HashMap** makes absolutely no guarantees about the iteration order. It can (and will) even change completely when new elements are added.
* **TreeMap** will iterate according to the "natural ordering" of the keys according to their compareTo() method (or an externally supplied Comparator). Additionally, it implements the SortedMap interface, which contains methods that depend on this sort order.
* **LinkedHashMap** will iterate in the order in which the entries were put into the map.

**How ConcurrentHashMap Works Internally in Java?**

**Answer: -** A ConcurrentHashMap is divided into number of segments.

A ConcurrentHashMap has internal final class called Segment so we can say that ConcurrentHashMap is internally divided in segments of size 32, so at max 32 threads can work at a time. It means each thread can work on a each segment during high concurrency and at most 32 threads can operate at max which simply maintains 32 locks to guard each bucket of the ConcurrentHashMap.

|  |
| --- |
| The definition of Segment is as below:  /\*\* Inner Segment class plays a significant role \*\*/  protected static final class Segment {  protected int count;  protected synchronized int getCount() {  return this.count;  }  protected synchronized void synch () {}  }  /\*\* Segment Array declaration \*\*/  public final Segment [] segments = new Segment[32]; |

As we all know that Map is a kind of data structure which stores data in key-value pair which is array of inner class Entry, see as below:

|  |
| --- |
| static class Entry implements Map.Entry {  protected final Object key;  protected volatile Object value;  protected final int hash;  protected final Entry next;  Entry (int hash, Object key, Object value, Entry next) {  this.value = value;  this.hash = hash;  this.key = key;  this.next = next;  }  // Code goes here like getter/setter  } |

And ConcurrentHashMap class has an array defined as below of type Entry class:

protected transient Entry [] table;

This Entry array is getting initialized when we are creating an instance of ConcurrentHashMap, even using a default constructor called internally as below:

|  |
| --- |
| public ConcurrentHashMap(int initialCapacity, float loadFactor) {  //Some code  int cap = getCapacity();  this.table = newTable(cap); // here this.table is Entry[] table  }  protected Entry[] newTable(int capacity) {  this.threshold = ((int)(capacity \* this.loadFactor / 32.0F) + 1);  return new Entry[capacity];  } |

Here, threshold is getting initialized for re-sizing purpose.

## **Inserting (Put) Element in ConcurrentHashMap:**

Most important thing to understand the put method of ConcurrentHashMap, that how ConcurrentHashMap works when we are adding the element. As we know put method takes two arguments both of type Object as below:

put(Object key, Object value)

So it wil 1st calculate the hash of key as below:

|  |
| --- |
| int hashVal = hash(key);  static int hash(Object x) {  int h = x.hashCode();  return (h << 7) - h + (h >>> 9) + (h >>> 17);  } |

After getting the hashVal we can decide the Segment as below:

Segment seg = segments[(hash & 0x1F)]; // segments is an array defined above   
   
Since it's all about concurrency, we need synchronized block on the above Segment as below:

|  |
| --- |
| synchronized (seg) {  // code to add  int index = hash & table.length - 1; // hash we have calculated for key and table is Entry[] table  Entry first = table[index];  for (Entry e = first; e != null; e = e.next) {  if ((e.hash == hash) && (eq(key, e.key))) { // if key already exist means updating the value  Object oldValue = e.value;  e.value = value;  return oldValue;  }  }  Entry newEntry = new Entry(hash, key, value, first); // new entry, i.e. this key not exist in map  table[index] = newEntry; // Putting the Entry object at calculated Index  } |

## **Size of ConcurrentHashMap**

Now when we are asking for size() of the ConcurrentHashMap the size comes out as below:

|  |
| --- |
| for (int i = 0; i < this.segments.length; i++) {  c += this.segments[i].getCount(); //here c is an integer initialized with zero  } |

## **Getting (get) Element From ConcurrentHashMap**

When we are getting an element from ConcurrentHashMap we are simply passing key and hash of key is getting calculated. The defintion goes something like as below:

|  |
| --- |
| public Object get(Object key){  //some code here  int index = hash & table.length - 1; //hash we have calculated for key and calculating index with help of hash  Entry first = table[index]; //table is Entry[] table  for (Entry e = first; e != null; e = e.next) {  if ((e.hash == hash) && (eq(key, e.key))) {  Object value = e.value;  if (value == null) {  break;  }  return value;  }  }  //some code here  } |

**Note:** No need to put any lock when getting the element from ConcurrentHashMap.

## **Removing Element From ConcurrentHashMap**

Now question is how remove works with ConcurrentHashMap, so let us understand it. Remove basically takes one argument 'Key' as an argument or takes two argument 'Key' and 'Value' as below:

|  |
| --- |
| Object remove(Object key);  boolean remove(Object key, Object value); |

Now let us understand how this works internally. The method remove (Object key) internally calls remove (Object key, Object value) where it passed 'null' as a value. Since we are going to remove an element from a Segment, we need a lock on the that Segment.

|  |
| --- |
| Object remove(Object key, Object value) {  Segment seg = segments[(hash & 0x1F)]; //hash we have calculated for key  synchronized (seg) {  Entry[] tab = this.table; //table is Entry[] table  int index = hash & tab.length - 1; //calculating index with help of hash  Entry first = tab[index]; //Getting the Entry Object  Entry e = first;  while(true) {  if ((e.hash == hash) && (eq(key, e.key))) {  break;  }  e = e.next;  }  Object oldValue = e.value;  Entry head = e.next;  for (Entry p = first; p != e; p = p.next) {  head = new Entry(p.hash, p.key, p.value, head);  }  table[index] = head;  seg.count -= 1;  }  return oldValue;  } |

Hope this will give you a clear understanding of the internal functionality of ConcurrentHashMap.

**SortedMap Interface**

The java.util.SortedMap interface is a subtype of the java.util.Map interface, with the addition that the elements stored in the map are sorted internally.

The order of the sorting is either the natural sorting order of the elements (if they implement java.lang.Comparable), or the order determined by a Comparator that you can give to the SortedSet.

By default the elements are iterated in ascending order, starting with the "smallest" and moving towards the "largest". But it is also possible to iterate the elements in descending order using the method TreeMap.descendingKeySet().

The Java Collections API only has one implementation of the SortedMap interface - the java.util.TreeMap class. The java.util.concurrent package also has an implementation of this interface, but I will leave the concurrency utilities out of this trail.

The main characteristic of a SortedMap is that, it orders the keys by their natural ordering, or by a specified comparator. So, consider using a [TreeMap](https://www.geeksforgeeks.org/hashmap-treemap-java/) when you want a map that satisfies the following criteria:

* null key or null value are not permitted.
* The keys are sorted either by natural ordering or by a specified comparator.

**Methods** of SortedMap:

* [subMap(K fromKey, K toKey)](https://www.geeksforgeeks.org/sortedmap-submap-method-in-java/): Returns a view of the portion of this Map whose keys range from fromKey, inclusive, to toKey, exclusive.
* headMap(K toKey): Returns a view of the portion of this Map whose keys are strictly less than toKey.
* [tailMap(K fromKey)](https://www.geeksforgeeks.org/sortedmap-tailmap-method-in-java/): Returns a view of the portion of this Map whose keys are greater than or equal to fromKey.
* [firstKey()](https://www.geeksforgeeks.org/sortedmap-firstkey-method-in-java/): Returns the first (lowest) key currently in this Map.
* [lastKey()](https://www.geeksforgeeks.org/sortedmap-lastkey-method-in-java/): Returns the last (highest) key currently in this Map.
* [comparator()](https://www.geeksforgeeks.org/sortedmap-comparator-method-in-java-with-examples/): Returns the Comparator used to order the keys in this Map, or null if this Map uses the natural ordering of its keys.
* [values ()](https://www.geeksforgeeks.org/sortedmap-values-method-in-java-with-examples/): Returns a Collection view of the values contained in this map.
* [keySet ()](https://www.geeksforgeeks.org/sortedmap-keyset-method-in-java-with-examples/): Returns a Set view of the keys contained in this map.
* [entrySet ()](https://www.geeksforgeeks.org/sortedmap-entryset-method-in-java-with-examples/): Returns a Set view of the mappings contained in this map.

**ConcurrentMap Interface**

* A java.util.concurrent.ConcurrentMap interface is a sub-interface of Map interface.
* ConcurrentMap is capable of handling concurrent access.
* The ConcurrentMap provides some extra methods apart from what it inherits from the Super-Interface i.e. java.util.Map.
* A ConcurrentMap providing thread safety and atomicity guarantees.

**Specific methods of ConcurrentMap interface: -**

* **Object putIfAbsent(K key, V value)**: If the specified key is not already associated with a value, associate it with the given value.
* **boolean remove (Object key, Object value):** Removes the entry for a key only if currently mapped to a given value.
* **boolean replace (K key, V oldValue, V newValue):** Replaces the entry for a key only if currently mapped to a given value.

**ConcurrentMap’s improved Methods in Java 8: -**

* **computeIfAbsent():** If the value is thread safe and can be safely updated outside the method, or you intend to synchronize on the value whilst updating it, or if you just want to be certain of getting a new or existing value without having to check for null.
* **Compute ():** If the value is not thread safe and must be updated inside the method with a remapping function to ensure the entire operation is atomic. This gives you the most control over the computation, but also the responsibility to handle the possibility that there is no existing value inside your remapping function.
* **Merge ():** Like compute (), you provide a remapping function to be performed on the existing value — if any. You also supply an initial value to be used if there is no existing value. This is a convenience because, with compute (), you have to, in the remapping function, handle the possibility of there being no existing value. Here, though, you do not have access to the key in the remapping function, unlike with compute ().

**NavigableMap Interface**

The java.util.NavigableMap interface is a subtype of the java.util.SortedMap interface. It has a few extensions to the SortedSet which makes it possible to navigate the map.The java.util package only has one implementation of the NavigableMap interface: java.util.TreeMap. And another at java.util.concurrent. ConcurrentSkipListMap

NavigableMap provides convenient navigation method like lowerKey, floorKey, ceilingKey and higherKey, and along with these popular navigation method it also provide ways to create a Sub Map from existing Map in Java e.g. headMap whose keys are less than specified key, tailMap whose keys are greater than specified key and a subMap which is strictly contains keys which falls between toKey and fromKey.

A NavigableMap may be accessed and traversed in either ascending or descending key order.

**Methods** of **NavigableMap**:

* **lowerKey(Object key)** : Returns the greatest key strictly less than the given key, or if there is no such key.
* **floorKey(Object key)** : Returns the greatest key less than or equal to the given key, or if there is no such key.
* **ceilingKey(Object key)** : Returns the least key greater than or equal to the given key, or if there is no such key.
* **higherKey(Object key)** : Returns the least key strictly greater than the given key, or if there is no such key.
* **descendingMap()** : Returns a reverse order view of the mappings contained in this map.
* **headMap(object toKey, boolean inclusive)** : Returns a view of the portion of this map whose keys are less than (or equal to, if inclusive is true) toKey.
* **subMap(object fromKey, boolean fromInclusive, object toKey, boolean toInclusive)** : Returns a view of the portion of this map whose keys range from fromKey to toKey.
* **tailMap(object fromKey, boolean inclusive)** : Returns a view of the portion of this map whose keys are greater than (or equal to, if inclusive is true) fromKey.

**ConcurrentNavigableMap Interface**

* A java.util.concurrent.ConcurrentNavigableMap interface is a subinterface of ConcurrentMap interface.
* ConcurrentNavigableMap is a NavigableMap which provides navigation methods that returns the closest match for given search targets with a concurrent access support for its **submaps**.
* The submaps are the maps returned by various methods like headMap(K toKey), tailMap(K fromKey) and subMap(K fromKey, K toKey).

## tailMap()

The tailMap(T fromKey) method returns a view of the map containing the keys which are greater than or equal to the given fromKey.

If you make changes to the original map, these changes are reflected in the tail map.

Here is an example illustrating the use of the tailMap() method:

ConcurrentNavigableMap map = new ConcurrentSkipListMap();

map.put("1", "one");

map.put("2", "two");

map.put("3", "three");

ConcurrentNavigableMap tailMap = map.tailMap("2");

The tailMap will contain the keys "2" and "3" because these two keys are greather than or equal to the given key, "2".

See the JavaDoc for more specific details of how this method works, and how its overloaded versions work.

## subMap()

The subMap() method returns a view of the original map which contains all keys from (including), to (excluding) two keys given as parameters to the method. Here is an example:

ConcurrentNavigableMap map = new ConcurrentSkipListMap();

map.put("1", "one");

map.put("2", "two");

map.put("3", "three");

ConcurrentNavigableMap subMap = map.subMap("2", "3");

The returned submap contains only the key "2", because only this key is greater than or equal to "2", and smaller than "3".

## More Methods

The ConcurrentNavigableMap interface contains a few more methods that might be of use. For instance:

* descendingKeySet()
* descendingMap()
* navigableKeySet()

**ConcurrentHashMap Class**

java.util.concurrent.ConcurrentHashMap Class implements ConcurrentMap as well as Serializable interface also. ConcureentHashMap is enhancement of HashMap as we know that while dealing with Threads in our application HashMap is not a good choice because performance wise HashMap is not up to the mark.

**Key points of ConcurrentHashMap:**

* The underlined data structure for ConcurrentHashMap is Hashtable.
* ConcurrentHashMap class is thread-safe i.e. multiple thread can operate on a single object without any complications.
* At a time any number of threads are applicable for read operation without locking the ConcurrentHashMap object which is not there in HashMap.
* In ConcurrentHashMap, the Object is divided into number of segments according to the concurrency level.
* Default concurrency-level of ConcurrentHashMap is 16.
* In ConcurrentHashMap, at a time any number of threads can perform retrieval operation but for updation in object, thread must lock the particular segment in which thread want to operate. This type of locking mechanism is known as **Segment locking or bucket locking**. Hence at a time 16 updation operations can be performed by threads.
* null insertion is not possible in ConcurrentHashMap as key or value.

**Constructors of ConcurrentHashMap:**

1. **ConcurrentHashMap m=new ConcurrentHashMap();**:Creates a new, empty map with a default initial capacity (16), load factor (0.75) and concurrencyLevel (16).
2. **ConcurrentHashMap m=new ConcurrentHashMap(int initialCapacity);**:Creates a new, empty map with the specified initial capacity, and with default load factor (0.75) and concurrencyLevel (16).
3. **ConcurrentHashMap m=new ConcurrentHashMap(int initialCapacity, float loadFactor);**:  
   Creates a new, empty map with the specified initial capacity and load factor and with the default concurrencyLevel (16).
4. **ConcurrentHashMap m=new ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel);**:Creates a new, empty map with the specified initial capacity, load factor and concurrency level.
5. **ConcurrentHashMap m=new ConcurrentHashMap(Map m);**:Creates a new map with the same mappings as the given map.

**Pitfalls: -**

* Retrieval operations generally do not block in *ConcurrentHashMap* and could overlap with update operations. So, for better performance, they only reflect the results of the most recently completed update operations.
* results of aggregate status methods including *size*, *isEmpty*, and *containsValue* are typically useful only when a map is not undergoing concurrent updates in other threads, if concurrent updates are under strict control, aggregate status would still be reliable. Although these **aggregate status methods do not guarantee the real-time accuracy, they may be adequate for monitoring or estimation purposes**. Note that usage of *size()* of *ConcurrentHashMap* should be replaced by *mappingCount()*, for the latter method returns a *long* count, although deep down they are based on the same estimation.
* hashCode matters: note that using many keys with exactly the same hashCode() is a sure way to slow down a performance of any hash table. To ameliorate impact when keys are Comparable, ConcurrentHashMap may use comparison order among keys to help break ties. Still, we should avoid using the same hashCode() as much as we can.
* iterators are only designed to use in a single thread as they provide weak consistency rather than fast-fail traversal, and they will never throw *ConcurrentModificationException.*
* the default initial table capacity is 16, and it’s adjusted by the specified concurrency level.
* caution on remapping functions: though we can do remapping operations with provided *compute* and *merge\** methods, we should keep them fast, short and simple, and focus on the current mapping to avoid unexpected blocking.
* keys in *ConcurrentHashMap* are not in sorted order, so for cases when ordering is required, *ConcurrentSkipListMap* is a suitable choice.

**ConcurrentSkipListMap Class**

java.util.concurrent.ConcurrentSkipListMap Class implements ConcurrentNavigableMap, Cloneable and Serializable interfaces. ConcurrentSkipListMap can be seen a scalable concurrent version of TreeMap. In practice, there’s no concurrent implementation of the red-black tree in Java. A concurrent variant of SkipLists is implemented in ConcurrentSkipListMap, providing an expected average log(n) time cost for the containsKey, get, put and remove operations and their variants. In addition to TreeMap‘s features, key insertion, removal, update and access operations are guaranteed with thread-safety.

* A scalable concurrent ConcurrentNavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.
* This class implements a concurrent variant of SkipLists providing expected average log(n) time cost for the containsKey, get, put and remove operations and their variants. Insertion, removal, update, and access operations safely execute concurrently by multiple threads.
* Iterators and spliterators are weakly consistent.
* Ascending key ordered views and their iterators are faster than descending ones.
* All Map.Entry pairs returned by methods in this class and its views represent snapshots of mappings at the time they were produced. They do not support the Entry.setValue method. (Note however that it is possible to change mappings in the associated map using put, putIfAbsent, or replace, depending on exactly which effect you need.)
* Beware that, unlike in most collections, the size method is not a constant-time operation. Because of the asynchronous nature of these maps, determining the current number of elements requires a traversal of the elements, and so may report inaccurate results if this collection is modified during traversal. Additionally, the bulk operations putAll, equals, toArray, containsValue, and clear are not guaranteed to be performed atomically. For example, an iterator operating concurrently with a putAll operation might view only some of the added elements.
* This class and its views and iterators implement all of the optional methods of the Map and Iterator interfaces. Like most other concurrent collections, this class does not permit the use of null keys or values because some null return values cannot be reliably distinguished from the absence of elements.

# TreeMap Class.

The TreeMap in Java is used to implement Map interface and NavigableMap along with the Abstract Class. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used. This proves to be an efficient way of sorting and storing the key-value pairs. The storing order maintained by the treemap must be consistent with equals just like any other sorted map, irrespective of the explicit comparators. The treemap implementation is not synchronized in the sense that if a map is accessed by multiple threads, concurrently and at least one of the threads modifies the map structurally, it must be synchronized externally. Some important features of the treemap are:

* This class is a member of Java Collections Framework.
* The class implements Map interfaces including NavigableMap, SortedMap and extends AbstractMap
* TreeMap in Java does not allow null keys (like Map) and thus a NullPointerException is thrown. However, multiple null values can be associated with different keys.
* All Map.Entry pairs returned by methods in this class and its views represent snapshots of mappings at the time they were produced. They do not support the Entry.setValue method.
* Java TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
* Java TreeMap contains only unique elements.
* Java TreeMap cannot have a null key but can have multiple null values.
* Java TreeMap is non synchronized.
* Java TreeMap maintains ascending order.
* Apart from implementing Map interface, Java TreeMap also implements NavigableMap and indirectly implements SortedMap interface. TreeMap also extends AbstractMap class.
* TreeMap entries are sorted in the natural ordering of its keys. It also provides a constructor to provide Comparator to be used for ordering. So if you are using any class as key, make sure it’s implementing Comparable interface for natural ordering. Check out java collections interview questions to understand the importance of these methods.
* Java TreeMap implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations.
* TreeMap is not synchronized and hence not thread-safe. For multithreaded environments, you can get a wrapped synchronized using Collections.synchronizedSortedMap method.
* TreeMap methods to get keyset and values return Iterator that are fail-fast in nature, so any concurrent modification will throw ConcurrentModificationException.
* TreeMap in java doesn’t allow null keys, however you can have multiple null values associated with different keys.

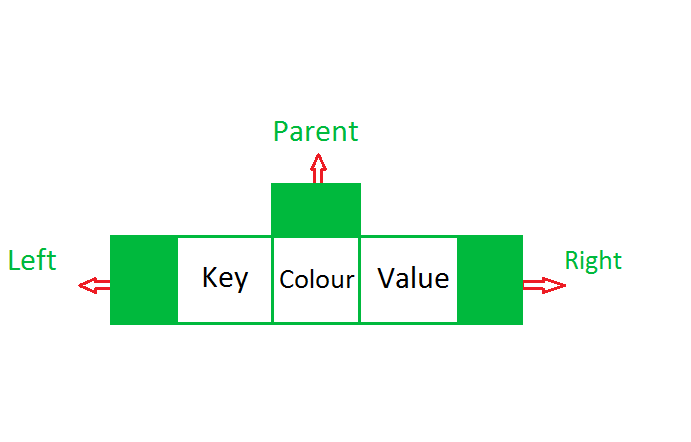
**Performance factors:**  
**TreeMap is not synchronized and thus is not thread-safe. For multithreaded environments, accidental unsynchronized access to the map is prevented by:**

SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));

**Internal structure:** The methods in TreeMap while getting keyset and values, return Iterator that are fail-fast in nature, thus any concurrent modification will throw ConcurrentModificationException.

TreeMap is based upon tree data structure. Each node in the tree has,

* 3 Variables (*K key=Key, V value=Value, boolean color=Color*)
* 3 References (*Entry left = Left, Entry right = Right, Entry parent = Parent*)



**HashMap class.**

Java.util.HashMap is Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

This implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

An instance of HashMap has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is *rehashed* (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the HashMap class, including get and put). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.

If many mappings are to be stored in a HashMap instance, creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table. Note that using many keys with the same hashCode() is a sure way to slow down performance of any hash table. To ameliorate impact, when keys are [Comparable](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html), this class may use comparison order among keys to help break ties.

**Note that this implementation is not synchronized.** If multiple threads access a hash map concurrently, and at least one of the threads modifies the map structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the [Collections.synchronizedMap](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#synchronizedMap-java.util.Map-) method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

Map m = Collections.synchronizedMap(new HashMap(...));

The iterators returned by all of this class's "collection view methods" are *fail-fast*: if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a [ConcurrentModificationException](https://docs.oracle.com/javase/8/docs/api/java/util/ConcurrentModificationException.html). Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

**Few important features of HashMap are:**

* HashMap is a part of java.util package.
* HashMap extends an abstract class AbstractMap which also provides an incomplete implementation of Map interface.
* It also implements [Cloneable](https://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html) and [Serializable](https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html) interface. K and V in the above definition represent Key and Value respectively.
* HashMap doesn’t allow duplicate keys but allows duplicate values. That means A single key can’t contain more than 1 value but more than 1 key can contain a single value.
* HashMap allows null key also but only once and multiple null values.
* This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time. It is roughly similar to HashTable but is unsynchronized.
* Java HashMap class contains values based on the key.
* Java HashMap class contains only unique keys.
* Java HashMap class may have one null key and multiple null values.
* Java HashMap class is non synchronized.
* Java HashMap class maintains no order.
* The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.
* Java HashMap allows null key and null values.
* HashMap is not an ordered collection. You can iterate over HashMap entries through keys set but they are not guaranteed to be in the order of their addition to the HashMap.
* HashMap is almost similar to Hashtable except that it’s unsynchronized and allows null key and values.
* HashMap uses it’s inner class Node<K,V> for storing map entries.
* HashMap stores entries into multiple singly linked lists, called buckets or bins. Default number of bins is 16 and it’s always power of 2.
* HashMap uses hashCode() and equals() methods on keys for get and put operations. So HashMap key object should provide good implementation of these methods. This is the reason immutable classes are better suitable for keys, for example String and Interger.
* Java HashMap is not thread safe, for multithreaded environment you should use ConcurrentHashMap class or get synchronized map using Collections.synchronizedMap() method.

# **LinkedHashMap Class**

# Java.util.LinkedHashMap is just like [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/) with an additional feature of maintaining an order of elements inserted into it. HashMap provided the advantage of quick insertion, search and deletion but it never maintained the track and order of insertion which the LinkedHashMap provides where the elements can be accessed in their insertion order. Few important features of LinkedHashMap are as follows:

* A LinkedHashMap contains values based on the key. It implements the Map interface and extends HashMap class.
* It contains only unique elements.
* It may have one null key and multiple null values.
* It is same as HashMap with additional feature that it maintains insertion order. For example, when we ran the code with HashMap, we got different oder of elements.
* Java LinkedHashMap maintains insertion order.
* The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.
* Default ordering provided by LinkedHashMap is the order on which key is inserted, known as insertion order, but LinkedHashMap can be created with another ordering called access order, which is defined by accessing entries.
* Re-entering a mapping, doesn't alter insertion order of LinkedHashMap. For example, if you already have mapping for a key, and want to update it's value by calling put(key, newValue), insertion order of LinkedHashMap will remain same.
* Access order is affected by calling get(key), put(key, value) or putAll(). When a particular entry is accessed, it moves towards end of the doubly linked list, maintained by LinkedHashMap.
* LinkedHashMap can be used to create LRU cache in Java. Since in LRU or Least Recently Used Cache, oldest non accessed entry is removed, which is the head of the doubly linked list maintained by LinkedHashMap.
* Iterator of LinkedHashMap returns elements in the order e.g. either insertion order or access order.
* LinkedHashMap also provides a method called removeEldestEntry(), which is protected and default implementation return false. If overridden, an implementation can return true to remove oldest entry, when a new entry is added.
* Given the insertion order guarantee of LinkedHashMap, Its a good compromise between HashMap and TreeMap in Java because with TreeMap you get increased cost of iteration due to sorting and performance drops on to log(n) level from constant time. That's all about difference between LinkedHashMap and HashMap in Java.

**There are lot of similarity between LinkedHashMap and HashMap in Java, as they both implement Map interface. let's have a look: -**

* Both LinkedHashMap and HashMap are not synchronized and subject to race condition if shared between multiple threads without proper synchronization. Use Collections.synchronizedMap() for making them synchronized.
* Iterator returned by HashMap and LinkedHashMap are fail-fast in nature.
* Performance of HashMap and LinkedHashMap are similar also.

# **Java.util.Hashtable Class**

Hashtable class extends Dictionary class and implements the Map interface. This class implements a hash table, which maps keys to values. Any non-null object can be used as a key or as a value.

To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method.

An instance of Hashtable has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of *buckets* in the hash table, and the *initial capacity* is simply the capacity at the time the hash table is created. Note that the hash table is *open*: in the case of a "hash collision", a single bucket stores multiple entries, which must be searched sequentially. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. The initial capacity and load factor parameters are merely hints to the implementation. The exact details as to when and whether the rehash method is invoked are implementation-dependent.

Generally, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the time cost to look up an entry (which is reflected in most Hashtable operations, including get and put).

The initial capacity controls a tradeoff between wasted space and the need for rehash operations, which are time-consuming. No rehash operations will *ever* occur if the initial capacity is greater than the maximum number of entries the Hashtable will contain divided by its load factor. However, setting the initial capacity too high can waste space.

If many entries are to be made into a Hashtable, creating it with a sufficiently large capacity may allow the entries to be inserted more efficiently than letting it perform automatic rehashing as needed to grow the table.

### Points to remember

* A Hashtable is an array of a list. Each list is known as a bucket. The position of the bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.
* Java Hashtable class contains unique elements.
* Java Hashtable class doesn't allow null key or value.
* Java Hashtable class is synchronized.
* The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.
* It is similar to HashMap but is synchronized.
* Hashtable stores key/value pair in hash table.
* In Hashtable we specify an object that is used as a key, and the value we want to associate to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

**When to use Hashtable**

**Answer: -** Let’s say we have a dictionary, where each word has its definition. Also, we need to get, insert and remove words from the dictionary quickly. Hence, Hashtable (or HashMap) makes sense. Words will be the keys in the Hashtable, as they are supposed to be unique. Definitions, on the other hand, will be the values.

***Hashtable* and *HashMap* provide very similar functionality.**

Both of them provide:

* Fail-fast iteration
* Unpredictable iteration order

But there are some differences too:

* *HashMap* doesn’t provide any *Enumeration, while Hashtable* provides not fail-fast *Enumeration*
* *Hashtable* doesn’t allow *null* keys and *null* values, while *HashMap* do allow one *null* key and any number of *null* values
* *Hashtable*‘s methods are synchronized while *HashMaps*‘s methods are not

**Java 8 has introduced new methods which help make our code cleaner: -**

* getOrDefault()
* putIfAbsent()
* boolean remove ()
* replace ()
* computeIfAbsent()
* computeIfPresent()
* compute ()
* merge ()
* oreach()
* replaceAll()

# **EnumMap class**

Java.util.EnumMap is specialized Map implementation designed and optimized for using Java Enum as key. Since enum can represent a type (like class or interface) in Java and it can also override equals() and hashCode() , It can be used inside HashMap or any other collection but using EnumMap brings implementation specific benefits which are done for enum keys, In short EnumMap is optimized Map implementation exclusively for enum keys . As per Javadoc Enum is implemented using Arrays and common operations result in constant time. So, if you are thinking of an high-performance Map, EnumMap could be a decent choice for enumeration data.

All of the keys in an enum map must come from a single enum type that is specified, explicitly or implicitly, when the map is created. Enum maps are represented internally as arrays. This representation is extremely compact and efficient.

Enum maps are maintained in the *natural order* of their keys (the order in which the enum constants are declared). This is reflected in the iterators returned by the collections views ([keySet()](https://docs.oracle.com/javase/8/docs/api/java/util/EnumMap.html#keySet--), [entrySet()](https://docs.oracle.com/javase/8/docs/api/java/util/EnumMap.html#entrySet--), and [values()](https://docs.oracle.com/javase/8/docs/api/java/util/EnumMap.html#values--)).

Iterators returned by the collection views are *weakly consistent*: they will never throw [ConcurrentModificationException](https://docs.oracle.com/javase/8/docs/api/java/util/ConcurrentModificationException.html) and they may or may not show the effects of any modifications to the map that occur while the iteration is in progress.

Null keys are not permitted. Attempts to insert a null key will throw [NullPointerException](https://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html). Attempts to test for the presence of a null key or to remove one will, however, function properly. Null values are permitted.

Like most collection implementations EnumMap is not synchronized. If multiple threads access an enum map concurrently, and at least one of the threads modifies the map, it should be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the enum map. If no such object exists, the map should be "wrapped" using the [Collections.synchronizedMap(java.util.Map<K, V>)](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#synchronizedMap-java.util.Map-) method. This is best done at creation time, to prevent accidental unsynchronized access

It extends AbstractMap and implements [Map](https://www.geeksforgeeks.org/map-interface-java-examples/) Interface in Java.

**Few important features of EnumMap are as follows:**

* EnumMap class is a member of the [Java Collections Framework](https://www.geeksforgeeks.org/collections-in-java-2/) & is not synchronized.
* EnumMap is ordered collection and they are maintained in the natural order of their keys (natural order of keys means the order on which enum constant are declared inside enum type)
* It’s a high-performance map implementation, much faster than [HashMap](https://www.geeksforgeeks.org/hashmap-treemap-java/).
* All keys of each EnumMap instance must be keys of a single [enum](https://www.geeksforgeeks.org/enum-in-java/) type.
* EnumMap doesn’t allow null key and throw NullPointerException, at same time null values are permitted.
* All keys used in EnumMap must be from same Enum type which is specified while creating EnumMap in Java. For example, if you cannot use different enum instances from two different enum.
* EnumMap is ordered collection and they are maintained in the natural order of their keys (natural order of keys means the order on which enum constant are declared inside enum type). you can verify this while Iterating over an EnumMap in Java.
* Iterators of EnumMap are fail-fast Iterator, much like of ConcurrentHashMap and doesn't throw ConcurrentModificationException and may not show effect of any modification on EnumMap during Iteration process.
* You cannot insert null keys inside EnumMap in Java. EnumMap doesn't allow null key and throw NullPointerException, at same time null values are permitted.
* EnumMap is not synchronized and it has to be synchronized manually before using it in a concurrent or multi-threaded environment. like synchronized Map in Java you can also make EnumMap synchronized by using Collections.synchronizedMap() method and as per javadoc this should be done while creating EnumMap in java to avoid accidental non synchronized access.
* EnumMap is likely give better performance than HashMap in Java. So, prefer EnumMap if you are going to use enum keys.

**What are the important use cases of IdentityHashMap?**

**Answer: -** A typical use of this class is topology-preserving object graph transformations, such as serialization or deep-copying. To perform such a transformation, a program must maintain a "node table" that keeps track of all the object references that have already been processed. The node table must not equate distinct objects even if they happen to be equal. Another typical use of this class is to maintain proxy objects. For example, a debugging facility might wish to maintain a proxy object for each object in the program being debugged.

Whenever you want your keys not to be compared by equals but by == you would use an IdentityHashMap. This can be very useful if you're doing a lot of reference-handling but it's limited to very special cases only.

**What is difference between IdentityHashMap and HashMap?**

**Answer: -**

* The main difference between HashMap vs IdentityHashMap is that IdentityHashMap uses equality operator "==" for comparing keys and values inside Map while HashMap uses equals method for comparing keys and values.
* Unlike HashMap, who uses hashcode to find bucket location, IdentityHashMap also doesn't use hashCode() instead it uses System.identityHashCode(object).
* Another key difference between IdentityHashMap and HashMap in Java is Speed. Since IdentityHashMap doesn't use equals() its comparatively faster than HashMap for object with expensive equals() and hashCode().
* One more difference between HashMap and IdentityHashMap is Immutability of the key. One of the basic requirement to safely store Objects in HashMap is keys needs to be immutable, IdentityHashMap doesn't require keys to be immutable as it is not relied on equals and hashCode.
* Initial capacity of HashMap is 16 by default .Initial capacity of IdentityHashMap is 21 by default.

# **IdentityHashMap class**

Java.util.IdentityHashMap implements Map, Serializable and Clonable interfaces and extends AbstractMap class.  
This class is not a general-purpose Map implementation. While this class implements the [Map](https://www.geeksforgeeks.org/map-interface-java-examples/) interface, it intentionally violates Map’s general contract, which mandates the use of the equals method when comparing objects.

This class is used when the user requires the objects to be compared via reference.

It is similar to HashMap except that it uses reference equality when comparing the elements.

This class is not a general-purpose Map implementation. While this class implements the Map interface, it intentionally violates Map's general contract, which mandates the use of the equals method when comparing objects.

This class is designed for use only in rare cases wherein reference-equality semantics are required. This class provides constant-time performance for the basic operations (get and put), assuming the system identity hash function (System.identityHashCode(Object)) disperses elements properly among the buckets.

This class has one tuning parameter (which affects performance but not semantics): expected maximum size. This parameter is the maximum number of key-value mappings that the map is expected to hold.

## **WeakHashMap Class**

java.util.WeakHashMap is an implementation of the Map interface that stores only weak references to its keys. Storing only weak references allows a key-value pair to be garbage-collected when its key is no longer referenced outside of the WeakHashMap.

This class provides the easiest way to harness the power of weak references. It is useful for implementing "registry-like" data structures, where the utility of an entry vanishes when its key is no longer reachable by any thread.

The WeakHashMap functions identically to the HashMap with one very important exception: if the Java memory manager no longer has a strong reference to the object specified as a key, then the entry in the map will be removed.

Both null values and the null key are supported. This class has performance characteristics similar to those of the HashMap class and has the same efficiency parameters of initial capacity and load factor. Like most collection classes, this class is not synchronized. A synchronized WeakHashMap may be constructed using the Collections.synchronizedMap method. Weak Reference − if the only references to an object are weak references, the garbage collector can reclaim the object's memory at any time. It doesn't have to wait until the system runs out of memory. Usually, it will be freed the next time the garbage collector runs. This class is a member of the Java Collections Framework.

**When would you use a WeakHashMap or a WeakReference?**

**Answer: -** One problem with strong references is caching, particular with very large structures like images. Suppose you have an application which has to work with user-supplied images, like the web site design tool I work on. Naturally you want to cache these images, because loading them from disk is very expensive and you want to avoid the possibility of having two copies of the (potentially gigantic) image in memory at once.

Because an image cache is supposed to prevent us from reloading images when we don't absolutely need to, you will quickly realize that the cache should always contain a reference to any image which is already in memory. With ordinary strong references, though, that reference itself will force the image to remain in memory, which requires you to somehow determine when the image is no longer needed in memory and remove it from the cache, so that it becomes eligible for garbage collection. You are forced to duplicate the behavior of the garbage collector and manually determine whether or not an object should be in memory.

**Difference between HashMap and WeakHashMap?**

**Answer: -**

|  |  |  |
| --- | --- | --- |
|  | **HashMap** | **WeakHashMap** |
| Entry object Garbage Collected | No ,even if key object is null | Yes |
| Key Object Reference | Strong | Weak |
| Automatic Size decrease | No | Yes |
| Clone method | Yes | No |
| Serialize and Deserialize objects | Yes | No |

**Similarities between HashMap and WeakHashMap in Java**

**Answer: -**

* Null key and Null values: Both classes (i.e WeakHashMap and HashMap ) permit null key and null values .
* Performance: WeakHashMap class has the similar characteristics as of the HashMap class and has the same efficiency parameters of initial capacity and load factor.
* Not Synchronized: Both classes are not synchronized. Collections.synchronizedMap() can be used to synchronize both HashMap and WeakHashMap class.
* Iterators returned by iterator method: The iterators returned by the iterator method of HashMap and WeakHashMap are fail-fast iterators. I have already discussed fail-safe vs fail-fast iterator with example.