# [How to Create Read Only List, Map and Set in Java – UnModifiable Collection Example](https://javarevisited.blogspot.com/2012/07/create-read-only-list-map-set-example-java.html)

**Read-only List, Map and Set in Java**

A read-only List means a List where you can not perform modification operations like add, remove or set. You can only read from the List by using get method or by using [Iterator](http://javarevisited.blogspot.sg/2011/10/java-iterator-tutorial-example-list.html) of List, This kind of List is good for a certain requirement where parameters are final and can not be changed. In Java, you can use Collections.unModifiableList() method to create read-only List , Collections.unmodifiableSet() for creating read-only Set like [read-only HashSet](http://javarevisited.blogspot.sg/2011/09/difference-hashmap-vs-hashset-java.html) and similarly creating a read-only [Map in Java](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html), as shown in below example. Any modification in the read-only List will result in java.lang.UnSupportedOperationException in Java.

This read-only List example is based on Java 5 generics but also applicable to other Java versions like JDK 1.4 or JDK 1.3, just remove [Generics](http://javarevisited.blogspot.sg/2011/09/generics-java-example-tutorial.html) code i.e. angle bracket which is not supported prior to Java 5. One common mistake programmer makes is that assuming fixed size List and read-only List as same.   
  
As shown in our [3 example of converting Array to Array List](http://javarevisited.blogspot.sg/2011/06/converting-array-to-arraylist-in-java.html), we can use Arrays.asList() method to [create and initialize List at the same line](http://javarevisited.blogspot.sg/2011/06/converting-array-to-arraylist-in-java.html). List implementation returned by this method is a fixed size and it doesn’t allow adding or removal of element but it is not read-only because you can update objects by calling set(index) method. How to make a collection read-only is also a [popular Java collection interview question](http://javarevisited.blogspot.sg/2011/11/collection-interview-questions-answers.html), which makes this Java collection tutorial even more important.

## Read only List, Set and Map Example - Java

Here is sample Java program which demonstrate method of creating read only List, [Set](http://javarevisited.blogspot.sg/2012/06/hashset-in-java-10-examples-programs.html) and Map in Java. You can make any [List, Set](http://javarevisited.blogspot.sg/2012/04/difference-between-list-and-set-in-java.html) or Map implementation as read only by following code example. Just remember that Collections.unModifiableList() , Collections.unModifiableSet() and Collections.unModifiableMap() [method in Java](http://javarevisited.blogspot.sg/2011/12/method-overloading-vs-method-overriding.html).   
  
[Example to create read only List Set and Map in Java](http://javarevisited.blogspot.sg/2011/08/enum-in-java-example-tutorial.html)On read-only List add, remove and set operation is not permitted, on read only Set you can not add or remove elements and in read only Map you can not put new entries or update existing entries. You can use same methods to convert any List implementation like [ArrayList](http://javarevisited.blogspot.sg/2011/05/example-of-arraylist-in-java-tutorial.html), LinkedList or Vector to read only ArrayList, LinkedList and [Vector in Java](http://javarevisited.blogspot.sg/2011/09/difference-vector-vs-arraylist-in-java.html).

**package** example;  
  
**import** java.util.ArrayList;  
**import** java.util.Collections;  
**import** java.util.HashMap;  
**import** java.util.HashSet;  
**import** java.util.List;  
**import** java.util.Map;  
**import** java.util.Set;

/\*\*  
 \* Java program to **create read only List, Set and Map in Java**. You can first create

 \* a List or Set and than make it unmodifiable or read-only by

 \* using  Collections.unmodifiableList() or Collections.unmodifiableSet() method.  
 \*  
 \* @author Javin Paul  
 \*/  
**public** **class** ReadOnlyListSetMap {  
   
    **public** **static** **void** main(**String** args[]) {              
         
        *// creating List in Java*  
        **List**<**String**> contents = **new** **ArrayList**<**String**>();  
       
        *// initializing List in Java*  
        contents.add("Example");  
        contents.add("Tutorial");  
        contents.add("Program");  
       
       
        *// Currently This List is not read only, you can add or remove elements from List*  
        contents.add("Tips"); *//should not be allowed if List is read only.*  
       
        **System**.err.println("normal List in Java : " + contents);  
       
        *//creating readonly List from contents*  
        contents = **Collections**.unmodifiableList(contents);  
       
        *//java.lang.UnsupportedOperationException -- no modification in read only list*  
       
       *//not allowed as it is read-only List in Java*  
       contents.add("Can I add object into read only List - No");  
  
         
        contents.remove("Example"); *//remove not allowed in read only list*  
       
        *//java.lang.UnSupportedOperation - List update not allowed*  
        contents.set(0, "Can I override or set object in read-only Set - No");  
  
       
        *//Creating read only Set in Java*  
        *//similar to read-only List you can also create a Set which is read only*  
        *//i.e. addition , removal and modification operation is not permitted on list*  
       
       
        *//Creating a Set based on contents of List*  
        **Set**<**String**> readOnlySet = **new** **HashSet**<**String**>(contents);  
       
        **System**.out.println("original Set in Java : " + readOnlySet);  
  
        *//Set is not yet read-only you can still add elements into Set*  
        readOnlySet.add("Override");  
  
        **System**.out.println("Set before making read only : " + readOnlySet);  
       
        *//making Set readonly in Java - no add remove or set operation permitted*  
        readOnlySet = **Collections**.unmodifiableSet(readOnlySet);  
       
        *//trying to add element in read only Set - java.lang.UnSupportedOperationException*  
        readOnlySet.add("You can not add element in read Only Set");  
       
        *//trying to remove element from read only set*  
        readOnlySet.remove("Example"); *//you can not remove elements from read only Set*  
       
       
       
        *// Creating read only Map in Java*  
        *// Similar to List and Set you can also create read only or unmodifiable Map in Java*  
        *// add , remove and override is not allowed on read only Map in Java*  
       
        **Map**<**String**, **String**> contries = **new** **HashMap**<**String**, **String**>();        
        contries.put("India", "New Delhi");  
       
        *//Map is not read only yet, you can still add entries into*  
        contries.put("UK", "London");  
       
        **System**.out.println("Map in Java before making read only: " + contries);  
       
        *//Making Map read only in Java*  
        **Map** readOnlyMap = **Collections**.unmodifiableMap(contries);  
         
        *//you can not put a new entry in read only Map in Java*  
        readOnlyMap.put("USA", "Washington"); *//java.lang.UnSupportedOperation*  
         
        *//you can not remove keys from read only Map in Java*  
        readOnlyMap.remove("UK"); *//java.lang.UnSupportedOperation*  
       
    }  
   
}

That’s all on**how to create read-only List, Set and Map in Java**. Its very easy to make any List implementation read only by wrapping it with Collections.unModifiableList(). Use same technique to convert any other Collection into read-only in Java.

**Q. How HashMap works?**

**Ans**. **HashMap in Java works on hashing principles**. It is a data structure which allows us to store object and retrieve it in constant time O(1) provided we know the key. In hashing, hash functions are used to link key and value in HashMap.

**How Hashmap Calculates the Index of a Bucket in Java**

To understand how HashMap works internally in Java, we must know about how the HashMap **calculates the index of the bucket**. Until now, we know the internal structure of HashMap, that HashMap maintains an **array of the bucket.** But when we store or retrieve any key-value pair, HashMap calculates the **index of the bucket** for each and every operation. The Key object is used to calculate the index of the bucket. By using this key, the hash value is calculated using the **hash(key)** private method of HashMap.

Note: **hash(key)** the method is a private method of HashMap that returns the hash value of the key, also if the hash value is too large then converts it into a smaller hash value.

But what will happen, if the hash value of Key Object returns the integer that is greater than the size of the array i.e., hash(key) > n, then ArrayOutOfBoundsException could be raised. To handle this situation, HashMap reduces the hash value between 0 and n-1 using an expression :

Index Calculating Expression:

index = hash(key) & (n-1)

Now, this index value is generated is used by HashMap to find bucket location and can never generate any Exception as the index value always from 0 to n-1.

What put() Method Does

Let’s note down the internal working of put method in hashmap.

1. First of all, the key object is checked for null.

If the key is null, the value is stored in table[0] position, because hashcode for null is always 0.

1. Then on the next step, a hash value is calculated using the key’s hash code by calling its **hashCode() method.** This hash value is used to calculate the index in the array for storing the Entry objects. JDK designers well assumed that there might be some poorly written hashCode() functions that can return very high or low hash code value. To solve this issue, they introduced another hash() function and passed the object’s hash code to this hash() function to bring hash value in the range of array index size.

Now the **indexFor(hash, table.length) the function is called** to calculate the exact index position for storing the Entry object.

How Collisions Are Resolved

Here comes the main part. Now, as we know that two unequal objects can have the same hash code value, how two different objects will be stored in the same array location called a bucket.

**The answer is LinkedList**. If we remember, the Entry class had an attribute "next." This attribute always points to the next object in the chain. This is exactly the behavior of the LinkedList.

So, in case of collision, Entry objects are stored in a linked list form. When an Entry object needs to be stored in a particular index, HashMap checks whether there is already an entry. If there is no entry already present, the entry object is stored in this location.

If there is already an object sitting on a calculated index, its next attribute is checked. If it is null, and the current entry object becomes the next node in LinkedList. If the next variable is not null, the procedure is followed until the next is evaluated as null.

What if we add another value object with the same key as entered before? Logically, it should replace the old value. How it is done? Well, after determining the index position of the Entry object, while iterating over LinkedListd on the calculated index, HashMap calls equals method on the key object for each entry object.

All these entry objects in LinkedList will have similar hashcode but equals() method will test for true equality. If key.equals(k) will be true then both keys are treated as the same key object. This will cause the replacement of value objects inside the entry object only.

HashMap in Java works on hashing principles. It is a data structure which allows us to store object and retrieve it in constant time O(1) provided we know the key. In hashing, hash functions are used to link key and value in HashMap. Objects are stored by calling put(key, value) method of HashMap and retrieved by calling get(key) method. When we call put method, the hashcode() method of the key object is called so that the hash function of the map can find a bucket location to store value object, which is actually an index of the internal array, known as the table.  
HashMap internally stores mapping in the form of Map.Entry object which contains both key and value object.  
When we want to retrieve the object, we call the get() method and again pass the key object. This time again key objects generate the same hash code (it's mandatory for it to do so to retrieve the object and that's why HashMap keys are immutable e.g. String) and we end up at same bucket location. If there is only one object then it is returned and that's our value object which we have stored earlier.  
Since the internal array of HashMap is of fixed size, and if we keep storing objects, at some point in time hash function will return the same bucket location for two different keys, this is called **collision** in HashMap. In this case, a linked list is formed at that bucket location and a new entry is stored as the next node.  
If we try to retrieve an object from this linked list, we need an extra check to search the correct value, this is done by equals() method. Since each node contains an entry, HashMap keeps comparing entry's key object with the passed key using equals() and when it returns true, Map returns the corresponding value.  
Since searching inlined list is O(n) operation, in worst case hash collision reduces a map to linked list. This issue is recently addressed in Java 8 by replacing the linked list to the tree to search in O(logN) time.

**Q. Have we used HashMap before or  What is HashMap? Why do we use it**

**Ans**. HashMap works on the principle of hashing, we have put(key, value) and get(key) method for storing and retrieving Objects from HashMap. When we pass Key and Value object  to put() method on Java HashMap, HashMap implementation calls **hashCode** method on Key object and applies returned hashcode into its own hashing function to find a bucket location for storing Entry object, important point to mention is that HashMap in Java stores both key and value object as Map.Entry in a bucket which is essential to understand the retrieving logic.   
If people fail to recognize this and say it only stores Value in the bucket they will fail to explain the retrieving logic of any object stored in Java HashMap

**Q. What will happen if two different objects have the same hashcode?**

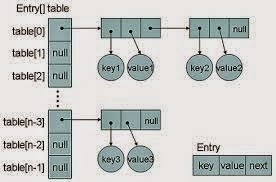
Now from here onwards real confusion starts, sometime candidate will say that since hashcode is equal, both objects are equal and HashMap  will throw exception or not store them again etc, Then we might want to remind them about equals() and hashCode() contract  that two unequal objects in Java can have same hashcode. Some will give up at this point and few will move ahead and say "Since hashcode is same, bucket location would be same and collision will occur in HashMap Since HashMap uses LinkedList to store object, this entry (object of Map.Entry comprise key and value )  will be stored in LinkedList. Great this answer make sense though there are many collision resolution methods available  like linear probing and chaining, this is simplest and HashMap in Java does follow this. But story does not end here and interviewer asks.

**Q.How will we retrieve Value object  if two Keys will have the same hashcode?**

Interviewee will say we will call get() method and then HashMap uses Key Object's hashcode to find out bucket location and retrieves Value object but then we need to remind him that there are two Value objects are stored in same bucket , so they will say about traversal in LinkedList until we find the value object , then we ask how do we identify value object because we don't  have value object to compare ,Until they know that HashMap  stores both Key and Value in LinkedList node or as Map.Entry they won't be able to resolve this issue and will try and fail.

But those bunch of people who remember this key information will say that after finding bucket location, we will call keys.equals() method to identify a correct node in LinkedList and return associated value object for that key in Java HashMap.

In many cases interviewee fails at this stage because they get confused between hashCode() and equals() or keys and values object in Java HashMap  which is pretty obvious because they are dealing with the hashcode() in all previous questions and equals() come in picture only in case of retrieving value object from HashMap in Java. Some good developer point out here that using immutable, final object with proper equals() and hashcode() implementation would act as perfect Java HashMap  keys and improve the performance of Java HashMap  by reducing collision. Immutability also allows caching their hashcode of different keys which makes overall retrieval process very fast and suggest that String and various wrapper classes e.g. Integer very good keys in Java HashMap.



Now if we clear this entire Java HashMap interview,  We will be surprised by this very interesting question

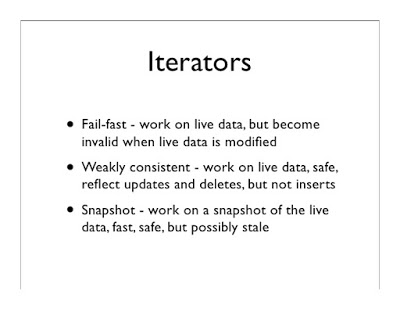
**What is fail safe and fail fast Iterator in Java?**

Java Collections supports two types of Iterator, fail safe and fail fast. The main distinction between a fail-fast and fail-safe Iterator is whether or not the underlying collection can be modified while its begin iterated. If you have used Collection like ArrayList then you know that when you iterate over them, no other thread should modify the collection. If Iterator detects any structural change after iteration has begun e.g adding or removing a new element then it throws ConcurrentModificationException,  this is known as fail-fast behavior and these iterators are called fail-fast iterator because they fail as soon as they detect any modification . Though it's not necessary that iterator will throw this exception when multiple threads modified it simultaneously. it can happen even with the single thread when you try to remove elements  by using ArrayList's remove() method instead of Iterator's remove method, as discussed in my earlier post, 2 ways to remove objects from ArrayList.  
  
Most of the Collection classes from Java 1.4 e.g. Vector, ArrayList, HashMap, HashSet has fail-fast iterators. The other type of iterator was introduced in Java 1.5 when concurrent collection classes e.g. ConcurrentHashMap, CopyOnWriteArrayList and CopyOnWriteArraySet was introduced.  
  
These iterator uses a view of original collection for doing iteration and that's why they doesn't throw ConcurrentModificationException even when original collection was modified after iteration has begun.  This means you could iterate and work with stale value, but this is the cost you need to pay for fail-safe iterator and this feature is clearly documented  
  
**Q. Difference between Fail Safe and Fail Fast Iterator in Java**

**Ans**. In order to best understand difference between these two iterators you need to try out examples with both traditional collections like ArrayList and concurrent collections like CopyOnWriteArrayList. Nevertheless let's first see some key differences one at a time :  
  
1) Fail-fast Iterator throws ConcurrentModfiicationException as soon as they detect any structural change in collection during iteration, basically which changes the modCount variable hold by Iterator. While fail-fast iterator doesn't throw ConcurrentModfiicationException.  
  
2) **Fail-fast iterator traverse over original collection** class while **fail-safe iterator traverse over a copy or view of original collection**. That's why they don't detect any change on original collection classes and this also means that you could operate with stale value.  
  
3) Iterators from Java 1.4 Collection classes e.g. ArrayList, HashSet and Vector are **fail-fast** while Iterators returned by concurrent collection classes e.g. CopyOnWriteArrayList or CopyOnWriteArraySet are **fail-safe**.  
  
4) Iterator returned by synchronized Collection are fail-fast while iterator returned by concurrent collections are fail-safe in Java.  
  
5) Fail fast iterator works in live data but become invalid when data is modified while fail-safe iterator are weekly consistent.

**When to use fail fast and fail-safe Iterator:**

**Use fail-safe** iterator when you are not bothered about Collection to be modified during iteration, as fail-fast iterator will not allow that. Unfortunate you can't choose fail safe or fail-fast iterator, it depends on upon which Collection class you are using. Most of the JDK 1.4 Collections e.g. HashSet, Vector, ArrayList has fail-fast Iterator and only Concurrent Collections introduced in JDK 1.5 e.g. CopyOnWriteArrayList and CopyOnWriteArraySet supports fail safe Iteration. **Also, if you want to remove elements during iteration please use iterator's remove() method and don't use remove method provided by Collection classes e.g. ArrayList or HashSet because that will result in ConcurrentModificationException.**

[](https://2.bp.blogspot.com/-VHrVmhrw9VU/VXB08P-aHvI/AAAAAAAAC3U/Xs2dkl_U-n0/s1600/Fail%2Bsafe%2Bvs%2BFail%2Bfast%2BITerator%2Bjava.jpg)

Keep in mind that when you work with concurrent collection classes like **ConcurrentHashMap you work with fail-safe iterator, which will not throw ConcurrentModificationException** but not necessarily be holding the most updated view of underlying Collection.

**Q. What happens On HashMap in Java if the size of the HashMap  exceeds a given threshold defined by load factor ?".**

**Ans**. Until we know how HashMap  works exactly we won't be able to answer this question. If the size of the Map exceeds a given threshold defined by load-factor e.g. if the load factor is .75 it will act to **re-size the map once it filled 75%.** Similar to other collection classes like **ArrayList,  Java HashMap re-size itself by creating a new bucket array of size twice of the previous size** **of HashMap and then start putting every old element into that new bucket array**. This process is called **rehashing** because it also applies the hash function to find new bucket location.

**Q. we see any problem with resizing of HashMap  in Java" ?**

**Ans**. we might not be able to pick the context and then he will try to give we hint about multiple thread accessing the Java HashMap and potentially looking for race condition on HashMap  in Java. **So the answer is Yes there is potential race condition exists while resizing HashMap in Java,** **if two thread at the same time found that now HashMap needs resizing and they both try to resizing**. on the process of resizing of HashMap in Java, the element in the bucket which is stored in linked list get reversed in order during their migration to new bucket **because Java HashMap  doesn't append the new element at tail instead it append new element at the head to avoid tail traversing.** If race condition happens then we will end up with an infinite loop. Though this point, we can potentially argue that what the hell makes we think to use HashMap  in multi-threaded environment to interviewer :

**Q. Can we use ConcurrentHashMap in place of Hashtable?**

**Ans**. Since we know **Hashtable** is **synchronized** but ConcurrentHashMap provides better concurrency by **only locking portion of map determined by concurrency level.** **ConcurrentHashMap** is certainly introduced as Hashtable and can be used in place of it, but Hashtable provides stronger thread-safety than ConcurrentHashMap.

**Q. What will happen if two different HashMap key objects have the same hashcode?**

They will be stored in the same bucket but no next node of linked list. And **keys equals () method will be used to identify the correct key-value pair in HashMap**.

**Q.) Why String, Integer and other wrapper classes are considered good keys?**

String, Integer and other wrapper classes are natural candidates of HashMap key, and String is most frequently used key as well **because String is immutable and final, and overrides equals and hashcode() method**. Other wrapper class also shares similar property. Immutability is required, in order to prevent changes on fields used to calculate hashCode() **because if key object returns different hashCode during insertion and retrieval than it won't be possible to get an object from HashMap.**   
Immutability is best as it offers other advantages as well like thread-safety, If we can  keep our hashCode same by only making certain fields final, then we go for that as well. Since equals() and hashCode() method is used during retrieval of value object from HashMap, it's important that key object correctly override these methods and follow contact. If unequal object returns different hashcode than chances of collision will be less which subsequently improve the performance of HashMap.

**Q. Can we use any custom object as a key in HashMap?**

**Ans**. This is an extension of previous questions. Of course **we can use any Object as key in Java HashMap provided it follows equals and hashCode contract** and **its hashCode should not vary once the object is inserted into Map**. If the **custom object is Immutable** than this will be already taken care because we can not change it once created.

**Q.How null key is handled in HashMap? Since equals() and hashCode() are used to store and retrieve values, how does it work in case of the null key?  
Ans** The null key is handled specially in HashMap, **there are two separate methods** for that **putForNullKey**(V value) and **getForNullKey**(). Later is an offloaded version of get() to look up null keys**.  Null keys always map to index 0**.  
This null case is split out into separate methods for the sake of performance in the two most commonly used operations (get and put), but incorporated with conditionals in others. **In short, equals() and hashcode() method are not used in case of null keys in HashMap.**  
here is how nulls are retrieved from HashMap  
  
   private V **getForNullKey**() {

if (size == 0) {

return null;

}

for (Entry<K,V> e = table[0]; e != null; e = e.next) {

if (e.key == null)

return e.value;

}

return null;

}

**HashMap Changes in JDK 1.7 and JDK 1.8**

There is some performance improvement done on HashMap and ArrayList from JDK 1.7, which reduces memory consumption. Due to this empty Map are lazily initialized and will cost we less memory. Earlier, when we create HashMap e.g. new HashMap() it automatically creates an array of default length e.g. 16.   
After some research, the Java team found that most of this Map are temporary and never use that many elements, and only end up wasting memory. Also, From JDK 1.8 onwards HashMap has introduced an improved strategy to deal with a high collision rate.   
Since a poor hash function e.g. which always return the location of the same bucket, can turn a HashMap into linked list, i.e. converting get() method to perform in O(n) instead of O(1) and someone can take advantage of this fact, Java now internally replace linked list to a binary tree once certain threshold is breached. This ensures performance or order O(log(n)) even in the worst case where a hash function is not distributing keys properly.

**Q. “How HashMap works?“,**

**Ans**: I simply answer: “**On principle of Hashing**“. As simple as it is. Now before answering it, one must be very sure to know at least basics of Hashing. Right??

**Q. What is Hashing?**

**Ans**. Hashing in its simplest form, is a way to assigning a unique code for any variable/object after applying any formula/algorithm on its properties.

A true hash function must follow this rule –

“Hash function should return the same hash code each and every time when the function is applied on same or equal objects. In other words, two equal objects must produce the same hash code consistently.”

All objects in Java inherit a default implementation of hashCode() function defined in Object class. This function produces hash code by typically converting the internal address of the object into an integer, thus producing different hash codes for all different objects.

**Q. Entry class in HashMap ?**

**Ans**: A map by definition is : “An object that maps keys to values”. Very easy.. right?

So, there must be some mechanism in HashMap to store this key-value pair. The answer is YES**. HashMap has an nested static class Entry**, which looks like this:

|  |
| --- |
| **Entry class** |
| **static class Entry<K ,V> implements Map.Entry<K, V>**  {      final K key;      V value;      Entry<K ,V> next;      final int hash;      ...//More code goes here  } |

Surely Entry class has key and value mapping stored as attributes. key has been marked as final and two more fields are there: **next and hash.**

**Q. How HashMap.put() methods works internally**

Before going into put() method’s implementation, it is very important to learn that instances of Entry class are stored in an array. HashMap class defines this variable as:

|  |
| --- |
| entry array |
| /\*\*   \* The table, resized as necessary. Length MUST Always be a power of two.   \*/  transient Entry[] table; |

Now look at code implementation of put() method:

|  |
| --- |
| put() method |
| /\*\*  \* Associates the specified value with the specified key in this map. If the  \* map previously contained a mapping for the key, the old value is  \* replaced.  \*  \* @param key  \*            key with which the specified value is to be associated  \* @param value  \*            value to be associated with the specified key  \* @return the previous value associated with <tt>key</tt>, or <tt>null</tt>  \*         if there was no mapping for <tt>key</tt>. (A <tt>null</tt> return  \*         can also indicate that the map previously associated  \*         <tt>null</tt> with <tt>key</tt>.)  \*/  **public V put(K key, V value)** {      if (key == null)      return **putForNullKey**(value);      int hash = **hash**(key.**hashCode**());      int i = **indexFor**(hash, table.length);      for (Entry<K , V> e = table[i]; e != null; e = e.next) {          Object k;          if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {              V oldValue = e.value;              e.value = value;              e.recordAccess(this);              return oldValue;          }      }        modCount++;      addEntry(hash, key, value, i);      return null;  } |

First of all, the key object is checked for null. If the key is null, the value is stored in table[0] position. Because hashcode for null is always 0.

Then on next step, a hash value is calculated using the key’s hash code by calling its **hashCode**() **method**. This hash value is used to calculate the index in the array for storing Entry object. JDK designers well assumed that there might be some poorly written **hashCode**() functions that can return very high or low hash code value. To solve this issue, they introduced another **hash**() function and passed the object’s hash code to this hash() function to bring hash value in the range of array index size.

Now **indexFor(hash, table.length)**function is called to calculate exact index position for storing the Entry object.

**Q. How collisions are resolved ?**

**Ans**. **The answer is LinkedList**. If we remember, Entry class had an attribute "next". This attribute always points to the next object in the chain. This is exactly the behavior of LinkedList.

So, in case of collision, Entry objects are stored in linked list form. When an Entry object needs to be stored in particular index, HashMap checks whether there is already an entry?? If there is no entry already present, the entry object is stored in this location.

If there is already an object sitting on calculated index, its next attribute is checked. If it is null, and current entry object becomes next node in linkedlist. If next variable is not null, procedure is followed until next is evaluated as null.

What if we add the another value object with same key as entered before. Logically, it should replace the old value. How it is done? Well, after determining the index position of Entry object, while iterating over linkedist on calculated index, HashMap calls **equals method on key object for each entry object.**

All these entry objects in linkedlist will have similar hashcode but equals() method will test for true equality. If key.equals(k) will be true then both keys are treated as same key object. This will cause the replacing of value object inside entry object only.

**Q. How HashMap.get() methods works internally?**

**Ans** . we already should know that the way key uniqueness is determined in put() method , same logic is applied in get() method also. The moment hashmap identify the exact match for the key object passed as an argument, it simply returns the value object stored in current Entry object. If no match is found, get() method returns null.

|  |
| --- |
| get() method |
| /\*\*  \* Returns the value to which the specified key is mapped,  or {@code null}if this map contains no mapping for the key.  \* <p>  \* More formally, if this map contains a mapping from a key {@code k} to a  \* value {@code v} such that {@code (key==null ? k==null :  \* key.equals(k))}, then this method returns {@code v}; otherwise it returns  \* {@code null}. (There can be at most one such mapping.)  \*  \* </p><p>  \* A return value of {@code null} does not <i>necessarily</i> indicate that  \* the map contains no mapping for the key; it's also possible that the map  \* explicitly maps the key to {@code null}. The {@link #containsKey  \* containsKey} operation may be used to distinguish these two cases.  \*  \* @see #put(Object, Object)  \*/  **public V get(Object key) {**      if (key == null)      return getForNullKey();      int hash = hash(**key.hashCode());**      for (Entry<K , V> e = table[**indexFor**(hash, table.length)]; e != null; e = e.next) {          Object k;          if (e.hash == hash && ((k = e.key) == key || key.equals(k)))              return e.value;      }      return null;  } |

Above code is same as put() method till if (e.hash == hash && ((k = e.key) == key || key.equals(k))), after this simply value object is returned.

**Key notes on internal working of HashMap**

1.Data structure to **store entry objects is an array named table of type Entry**.

A particular index location in array is referred as bucket, because it can hold the first element of a linkedlist of entry objects.

**Key object’s hashCode() is required to calculate the index location** of Entry object.

**Key object’s equals() method is used to maintain uniqueness of keys in map.**

**Value object’s hashCode() and equals() method are not used in** **HashMap’s get() and put() methods**.

**Hash code for null keys is always zero,** and such entry object is always stored in zero index in Entry[].

**Q. HashMap improvements in Java 8**

**Ans .**As part of the work for JEP 180, there is a performance improvement for HashMap objects where there are lots of collisions in the keys **by using balanced trees rather than linked lists to store map entries**. **The principal idea is that once the number of items in a hash bucket grows beyond a certain threshold, that bucket will switch from using a linked list of entries to a balanced tree.** In the case of high hash collisions, this will improve worst-case performance from O(n) to O(log n).

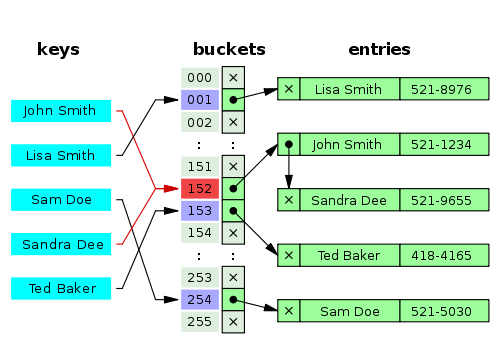
Basically when a bucket becomes too big (currently: TREEIFY\_THRESHOLD = 8), HashMap dynamically replaces it with an ad-hoc implementation of the treemap. This way rather than having pessimistic O(n) we get much better O(log n).

Bins (elements or nodes) of TreeNodes may be traversed and used like any others, but additionally support faster lookup when overpopulated. However, since the vast majority of bins in normal use are not overpopulated, checking for the existence of tree bins may be delayed in the course of table methods.

Tree bins (i.e., bins whose elements are all TreeNodes) are ordered primarily by hashCode, but in the case of ties, if two elements are of the same “class C implements Comparable<C>“, type then their compareTo() method is used for ordering.

Because TreeNodes are about twice the size of regular nodes, we use them only when bins contain enough nodes. And when they become too small (due to removal or resizing) they are converted back to plain bins (currently: UNTREEIFY\_THRESHOLD = 6). In usages with well-distributed user hashCodes, tree bins are rarely used.

**Q. What is bucket ?  
Ans**. A bucket is used to store key value pairs . A bucket can have multiple key-value pairs . In hash map, bucket used simple linked list to store objects .



**Q. How will we measure the performance of HashMap?  
Ans**.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( HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.), 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.  
In HashMap class, the default value of load factor is (.75) .

**Q. What is the time complexity of Hashmap get() and put() method ?  
Ans**. The hashmap implementation provides constant time performance for (get and put) basic operations.  
i.e the complexity of get() and put() is O(1) , assuming the hash function disperses the elements properly among the buckets.

**Q. How HashMap works in Java 8**

**Ans**. In java 8 there is a lot of changes in the inner representation of HashMap. The implementation of HashMap went from 1k lines of code in java 7 to 2k lines of code in java 8. In java 8, Node class contains the exact same information as the Entry class i.e Node class contains ( hash , key, value, bucketindex).  
Here is the implementation of the Node class in java 8.

static **class** **Node**<K,V> implements Map.Entry<K,V> {

final int hash;

final K key;

V value;

Node<K,V> next;

Node(int hash, K key, V value, Node<K,V> next) {

this.hash = hash;

this.key = key;

this.value = value;

this.next = next;

}

**Q. What’s the big difference from java 7 ?  
Ans**. TreeNode class extends Node through LInkedHashMap.Entry<K,V>. In other words ,  
TreeNode indirectly extends Node class. A TreeNode internally implements Red-Black tree structure. It stores more information than Node class so that it can perform get(),add() or delete() operations in O(log(n)). As we know Node class contains **(hash,key,value,bucketindex**) where as TreeNode class contains following list of data.

static final **class** **TreeNode**<K,V> extends LinkedHashMap.Entry<K,V> {

TreeNode<K,V> parent; // red-black tree links

TreeNode<K,V> left;

TreeNode<K,V> right;

TreeNode<K,V> prev; // needed to unlink next upon deletion

boolean red;

TreeNode(int hash, K key, V val, Node<K,V> next) {

super(hash, key, val, next);

}

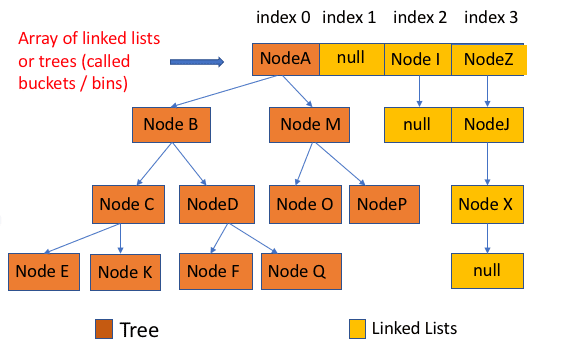
**Q. What are Red Black Trees and Why they are used?  
Ans.** Red-Black trees are self-balancing **binary search trees**. Red-black tree makes sure that the length of the binary search trees is always log(n) despite new addition or removal of nodes. The main advantage of using Red-black tree structure is in a case where many entries are in the same bucket. For search operation,in java 7,it will take O(n) with a linked list. **While in java 8 , the same search operation in a tree will cost O(log(n)).**  
**Drawbacks** : Tree really takes more space than the linked list.  
By Inheritance, bucket can contain both Node(Entry object) and TreeNode(Red-black tree).  
Oracle java developers decided to use both data structures and following rules are applied.  
**1**. If for a given bucket , there are more than 8 Nodes, the linked list is converted into a  
red-black tree. This is represented by the following code in HashMap class :

static final int TREEIFY\_THRESHOLD = 8;

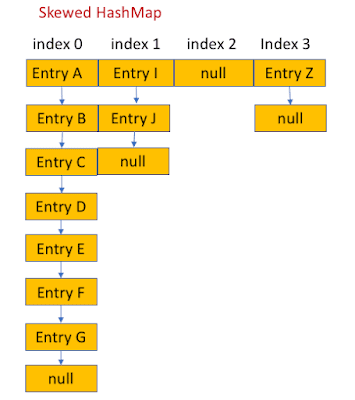
2. If for a given bucket , there are less than 6 nodes, then the tree can be converted  
into a linkedlist.

static final int UNTREEIFY\_THRESHOLD = 6;

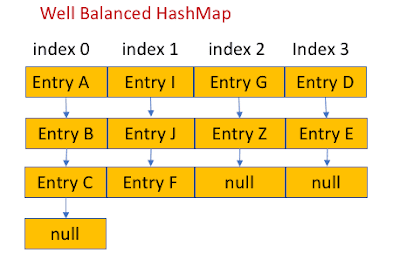
The below Java 8 HashMap image shows both trees(at bucket 0) and linkedlists (at bucket 1,2 and 3). Bucket 0 is a Tree because it contains at least 8 nodes.



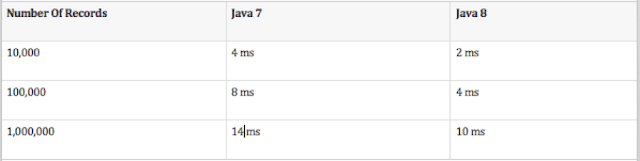
**Performance Issue in Java 8**  
In the best case scenario, put() and get() operations have **a O(1) time complexity**. But if we do not provide efficient hash function of the key , then we might end up with very slow get() and put() operations.  
The good performance of the get() and put() operations depend on the repartition of the data into the different indexes of the bucket.  
If the hash function of our key is poorly designed, then we will have a skew repartition (capacity of the bucket becomes irrelevant). All the get() and put() operations that use the biggest linked lists of entry will be really slow. It is due to the reason as the get() and put() operation need to iterate the entire lists. In the worst case scenario (when hash function of the key is poorly designed and all the entry objects are in the same buckets), we would end up **with O(n) time complexity.**  
Below are the images showing skewed HashMap and well balanced HashMap. In the case of skewed HashMap . the put() and get() operations on the bucket 0 are costlier. Getting the Entry F will cost 5 iterations.



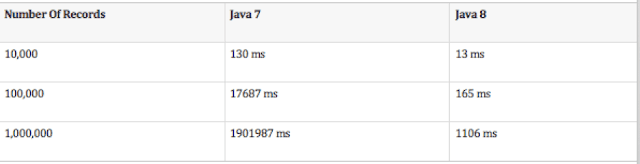
In the case of well balanced HashMap , getting the Entry F will cost 2 iterations.



Interestingly, both above HashMaps store the same amount of data and have the same bucket size.  
**Q. Why there is difference between skewedHashMap and well balanced HashMap entries  
Ans.** The only difference is the hash function of the key that distributes the Entries in the buckets.  
Testing : Performance of HashMap Java 7 vs Java 8 :  
 1. get() method with proper hashCode() logic

[](https://2.bp.blogspot.com/-sL7Ged_Ncw0/XEYCKPtPxfI/AAAAAAAAA8s/GNoH6UYYQogsrTwWhQrA5RRl7cQm-3HjwCLcBGAs/s1600/HashMap%2Bget%2Bperformance%2Bjava7%2Bvs%2Bjava8.png)

 1. get() method with poorly designed hashCode() logic

[](https://2.bp.blogspot.com/-HacBaiunnAg/XEYCjwmeEhI/AAAAAAAAA80/q_fzv9xyvR8ewk-tqxNDK-16TTNXC65mQCLcBGAs/s1600/HashMap%2Bget%2Bperformance.png)

When using HashMap, our goal is to write a hash function for our keys that spreads the keys into most number of possible buckets. To do that, we need to avoid hash collisions.  
**Memory Overhead in Java 8 and Java 7**  
With the java 8 implementation of HashMap, it is quite difficult to get the memory usage because a Node object can contain the same data as an Entry object or same data plus a boolean and 4 more references (if it’s TreeNode).  
If all the nodes are Nodes then memory overhead in Java 8 will be same as the Java 7 HashMap.  
The worst case scenario , if all the nodes in the HashMap are TreeNodes , then the memory overhead of a Java 8 HashMap becomes :

N \* sizeOf(integer) + N \* sizeOf(boolean) + sizeOf(reference)\* (9\*N+CAPACITY)

**Resizing overhead in Java 8 and Java 7**  
If we need to store large amount of data into the HashMap then we should create a  
HashMap with an initial capacity close to our expected volume.  
If we missed that , the HashMap will take the default size of 16 and load factor of 0.75. The first 11 put() operations will be fast but the 12th (16\*0.75) will resize the bucket with a new capacity of 32. The 13th to 23rd will be fast but 24th (0.75 \* 32) will again recreate a costly new representation that doubles the initial size of the bucket. The initial resizing operation will appear at 48th , 96th ,192nd , 384th call of put() operation.  
At low volume of data the full recreation of the bucket is fast but at high volume of data it can take seconds to minutes.  
**Note** : By setting the expected size initially, we can avoid these costly operations.  
**Drawback** : If we initialize HashMap with size 2^32 but we are using only 2^29 buckets then we will waste a lot of memory.

If our application demands faster insertion and faster retrieval then HashMap is the ultimate choice. While selecting the data structure, we must keep two things in our mind. First one is that the data structure must give better performance while inserting the new elements and second one is that it should give even more better performance while searching for an element. Because insertion and retrieval are two operations which we perform very frequently in our applications. These things will matter even more when we are handling the big data. HashMap is the most sought after data structure when we are handling the big data with more preference to insertion and retrieval operations.

HashMap is the most used data structure in java because it gives almost constant time performance of O(1) for put and get operations irrespective of how big is the data. As we already know, HashMap stores the data in the form of key-value pairs. In this post, we will see how HashMap works internally in java and how it stores the elements to give O(1) performance for put and get operations.

HashMap Internal Structure :

HashMap stores the data in the form of key-value pairs. Each key-value pair is stored in an object of Entry<K, V> class. Entry<K, V> class is the static inner class of HashMap which is defined like below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | static class Entry<K,V> implements Map.Entry<K,V>  {          final K key;          V value;          Entry<K,V> next;          int hash;            //Some methods are defined here  } |

As we see, this inner class has four fields. key, value, next and hash.

**key** : It stores the key of an element and its final.

**value** : It holds the value of an element.

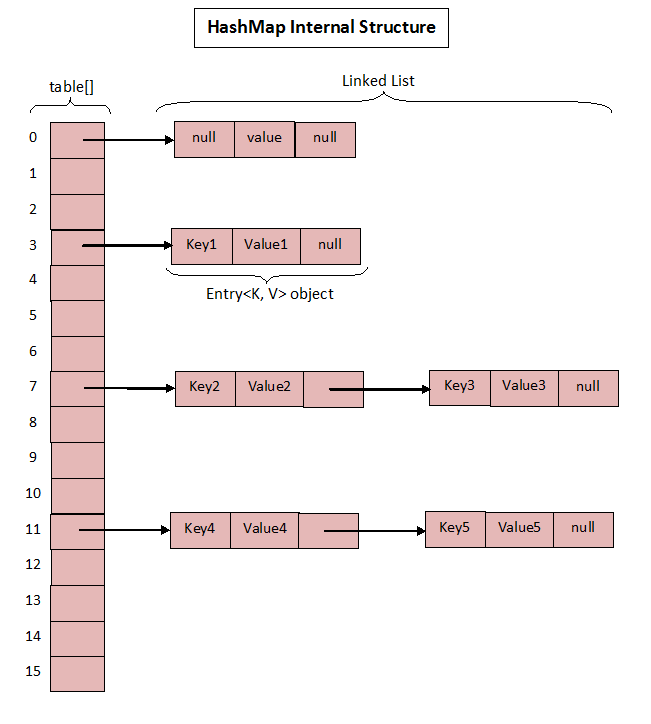
**next** : It holds the pointer to next key-value pair. This attribute makes the key-value pairs stored as a linked list.

**hash** : It holds the hashcode of the key.

These Entry objects are stored in an array called table[]. This array is initially of size 16. It is defined like below.

|  |  |
| --- | --- |
| 1  2  3  4 | /\*\*       \* The table, resized as necessary. Length MUST Always be a power of two.       \*/      transient Entry<K,V>[] table; |

To summarize the whole HashMap structure, each key-value pair is stored in an object of Entry<K, V> class. This class has an attribute called next which holds the pointer to next key-value pair. This makes the key-value pairs stored as a linked list. All these Entry<K, V> objects are stored in an array called table[]. The below image best describes the HashMap structure.



The above image roughly shows how the HashMap stores its elements. Internally it uses an array of Entry<K, V> class called table[] to store the key-value pairs. But how HashMap allocates slot in table[] array to each of its key-value pair is very interesting. It doesn’t inserts the objects as we put them into HashMap i.e first element at index 0, second element at index 1 and so on. Instead it uses the hashcode of the key to decide the index for a particular key-value pair. It is called Hashing.

**Q.What Is Hashing?**

Ans : The whole HashMap data structure is based on the principle of Hashing. **Hashing** is nothing but the function or algorithm or method which when applied on any object/variable returns an unique integer value representing that object/variable. This unique integer value is called hash code. Hash function or simply hash said to be the best if it returns the same hash code each time it is called on the same object. Two objects can have same hash code.

Whenever we insert new key-value pair using put() method, HashMap blindly doesn’t allocate slot in the table[] array. Instead it calls hash function on the key. HashMap has its own hash function to calculate the hash code of the key. This function is implemented so that it overcomes poorly implemented hashCode() methods. Below is implementation code of hash().

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | /\*\*  Retrieve object hash code and applies a supplemental hash function to the  result hash, which defends against poor quality hash functions.  This is  critical because HashMap uses power-of-two length hash tables, that  otherwise encounter collisions for hashCodes that do not differ  in lower bits. Note: Null keys always map to hash 0, thus index 0.       \*/      final int **hash**(Object k) {          int h = 0;          if (useAltHashing) {              if (k instanceof String) {                  return sun.misc.Hashing.stringHash32((String) k);              }              h = hashSeed;          }            h ^= k.hashCode();            // This function ensures that hashCodes that differ only by          // constant multiples at each bit position have a bounded          // number of collisions (approximately 8 at default load factor).          h ^= (h >>> 20) ^ (h >>> 12);          return h ^ (h >>> 7) ^ (h >>> 4);      } |

After calculating the hash code of the key, it calls indexFor() method by passing the hash code of the key and length of the table[] array. This method returns the index in the table[] array for that particular key-value pair.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | /\*\*       \* Returns index for hash code h.       \*/      static int indexFor(int h, int length) {          return h & (length-1);      } |

Now, let’s see how put() method works in detail.

How put() method works?

Below is the code implementation of put() method in the HashMap class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | /\*\*       \* Associates the specified value with the specified key in this map.       \* If the map previously contained a mapping for the key, the old       \* value is replaced.       \*       \* @param key key with which the specified value is to be associated       \* @param value value to be associated with the specified key       \* @return the previous value associated with <tt>key</tt>, or       \*         <tt>null</tt> if there was no mapping for <tt>key</tt>.       \*         (A <tt>null</tt> return can also indicate that the map       \*         previously associated <tt>null</tt> with <tt>key</tt>.)       \*/      public V put(K key, V value) {          if (key == null)              return putForNullKey(value);          int hash = hash(key);          int i = indexFor(hash, table.length);          for (Entry<K,V> e = table[i]; e != null; e = e.next) {              Object k;              if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {                  V oldValue = e.value;                  e.value = value;                  e.recordAccess(this);                  return oldValue;              }          }            modCount++;          addEntry(hash, key, value, i);          return null;      } |

Let’s see how this code works step by step.

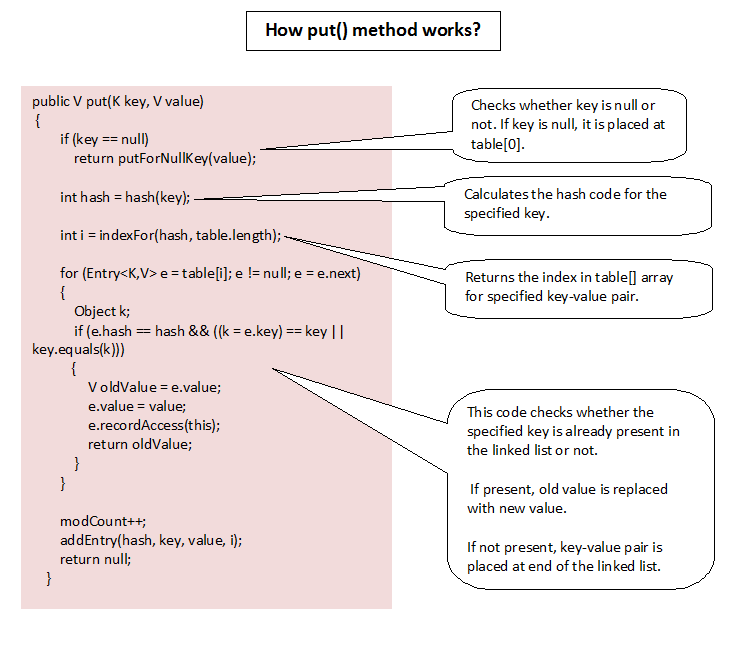
**Step 1 :** First checks whether the key is null or not. If the key is null, it calls **putForNullKey() method. table[0]** is always reserved **for null key**. Because, hash code of null is 0.

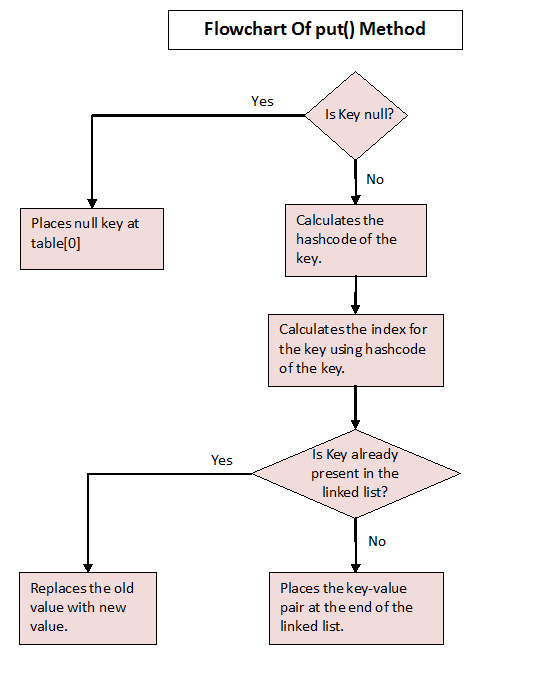
**Step 2 :** **If the key is not null**, then it calculates the hash code of the key by **calling hash() method.**

**Step 3 :** **Calls indexFor() method** by passing the hash code calculated in step 2 and length of the table[] array. This method returns index in table[] array for the specified key-value pair.

**Step 4 :** After getting the index**, it checks all keys present in the linked list at that index** ( or bucket). If the key is already present in the linked list, it replaces the old value with new value.

**Step 5 :** If the key is not present in the linked list, it appends the specified key-value pair at the end of the linked list.





How get() method Works?

Let’s see how get() method has implemented.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | /\*\*  \* Returns the value to which the specified key is mapped, or {@code null}  \* if this map contains no mapping for the key.  \*  \*    \* More formally, if this map contains a mapping from a key {@code k} to a  \* value {@code v} such that {@code (key==null ? k==null :  \* key.equals(k))}, then this method returns {@code v}; otherwise it returns  \* {@code null}. (There can be at most one such mapping.)  \*  \*    \* A return value of {@code null} does not <i>necessarily</i> indicate that  \* the map contains no mapping for the key; it's also possible that the map  \* explicitly maps the key to {@code null}. The {@link #containsKey  \* containsKey} operation may be used to distinguish these two cases.  \*  \* @see #put(Object, Object)  \*/  public V get(Object key) {      if (key == null)      return getForNullKey();      int hash = hash(key.hashCode());      for (Entry<K , V> e = table[indexFor(hash, table.length)]; e != null; e = e.next) {          Object k;          if (e.hash == hash && ((k = e.key) == key || key.equals(k)))              return e.value;      }      return null;  } |

Step 1 : First checks whether specified key is null or not. If the key is null, it calls getForNullKey() method.

Step 2 : If the key is not null, hash code of the specified key is calculated.

Step 3 : indexFor() method is used to find out the index of the specified key in the table[] array.

Step 4 : After getting index, it will iterate though linked list at that position and checks for the key using equals() method. If the key is found, it returns the value associated with it. otherwise returns null.

**Q. Difference between ConcurrentHashMap, Hashtable and Synchronized Map in Java**

**Ans**. Though all three collection classes are thread-safe and can be used in multi-threaded, concurrent Java application, there is a significant difference between them, which arise from the fact that how they achieve their thread-safety.

**Hashtable** is a legacy class from JDK 1.1 itself, which uses **synchronized** methods to achieve thread-safety. All methods of Hashtable are synchronized which makes them quite slow due to contention if a number of thread increases.

**Synchronized Map** is also not very different than Hashtable and provides similar performance in concurrent Java programs. The only difference between Hashtable and Synchronized Map is that later is not a legacy and we can wrap any Map to create it's synchronized version by using **Collections.synchronizedMap() method.**  
**ConcurrentHashMap** is specially designed for concurrent use i.e. more than one thread. By default it simultaneously allows 16 threads to read and write from Map without any external synchronization. It is also very scalable because of stripped locking technique used in the internal implementation of ConcurrentHashMap class. Unlike Hashtable and Synchronized Map, it never locks whole Map, instead, it divides the map into segments and locking is done on those. Though it performs better if a number of reader threads are greater than the number of writer threads.

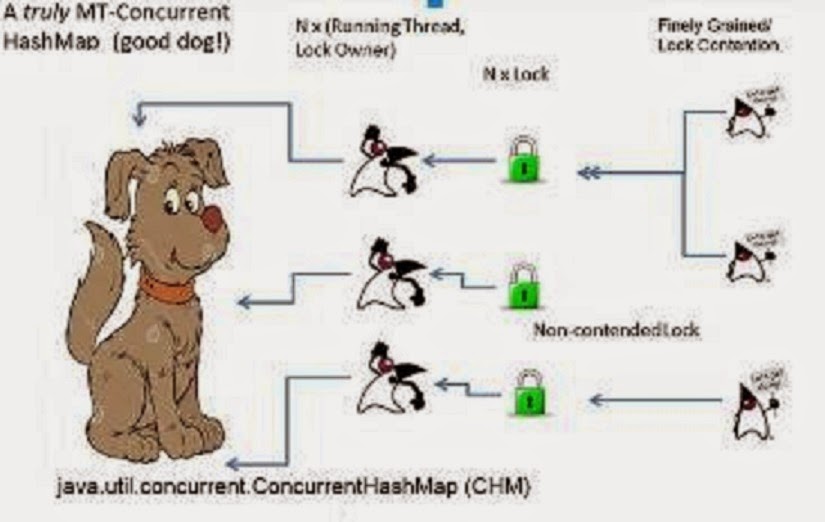
To be frank, Collections classes are the heart of Java API though I feel using them judiciously is an art. It's my personal experience where I have improved the performance of Java application by using ArrayList where legacy codes were unnecessarily using Vector etc. Prior Java 5, One of the major drawback of Java Collection framework was a lack of scalability.  
In multi-threaded Java application synchronized collection classes like Hashtable and Vector quickly becomes the bottleneck; to address scalability JDK 1.5 introduces some good concurrent collections which are highly efficient for high volume, low latency system electronic trading systems In general those are the backbone for Concurrent fast access to stored data..  
**Q. Why need ConcurrentHashMap and CopyOnWriteArrayList**

**Ans** .The **synchronized collections classes**, **Hashtable**, and **Vector**, and the **synchronized** **wrapper classes**, **Collections.synchronizedMap()** and **Collections.synchronizedList(),** provide a basic conditionally thread-safe implementation of Map and List. However, several factors make them unsuitable for use in highly concurrent applications, for example, their single collection-wide lock is an impediment to scalability and it often becomes necessary to lock a collection for a considerable time during iteration to revent ConcurrentModificationException.  
  
**ConcurrentHashMap** and **CopyOnWriteArrayList** implementations provide much **higher** **concurrency** while preserving thread safety, with some minor compromises in their promises to callers. ConcurrentHashMap and CopyOnWriteArrayList are not necessarily useful everywhere we might use HashMap or ArrayList, but are designed to optimize specific common situations. Many concurrent applications will benefit from their use.

**Q. Difference between ConcurrentHashMap and Hashtable**

**Ans** .So what is the difference between Hashtable and ConcurrentHashMap, both can be used in the multithreaded environment but once the size of Hashtable becomes considerable large performance degrade because for iteration it has to be locked for a longer duration.

Since ConcurrentHashMap introduced the **concept of segmentation**, how large it becomes only **certain part of it get locked** to provide thread safety so many other readers can still access map without waiting for iteration to complete.   
In Summary, **ConcurrentHashMap** only locked certain portion of Map while **Hashtable** locks full map while doing iteration. This will be clearer by looking at this diagram which explains the internal working of ConcurrentHashMap in Java.

[](https://3.bp.blogspot.com/-f2O3cncdnjk/VJ5UDxij_FI/AAAAAAAACSU/ut1ybg3WR2E/s1600/ConcurrentHashMap%2Bin%2BJava%2Bwith%2BExample.png)

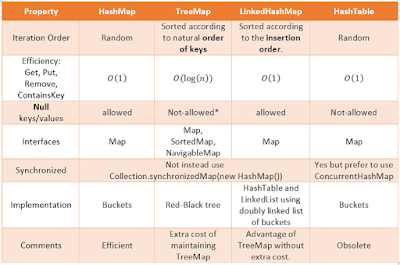
The difference between ConcurrentHashMap and Collections.synchronizedMap

**ConcurrentHashMap** is designed for concurrency and improve performance while HashMap which is non-synchronized by nature can be **synchronized** by applying a wrapper using synchronized Map.   
**ConcurrentHashMap** does not allow null keys or null values while **synchronized** HashMap allows one null key.

**Q. What is difference between HashMap and Hashtable in Java?**

**Ans**. Though both Hashtable and HashMap are data-structure based upon hashing and implementation of Map interface, the main difference between them is that HashMap is not thread-safe but **Hashtable is thread-safe**. This means we cannot use HashMap in a multi-threaded Java application without external synchronization. Another difference is **HashMap allows one null key and null values** but **Hashtable doesn't allow null key or values**. Also, the thread-safety of the hashtable is achieved using internal synchronization, which makes it slower than HashMap.  
Hashtable is a legacy Collection class and it's there in Java API for a long time but it got refactored to implement Map interface in Java 4 and from there Hashtable became part of the Java Collection framework.  
  
1.The HashMap class is roughly equivalent to Hashtable, except that it is non-synchronized and permits nulls. (**HashMap allows null values as key** and value whereas **Hashtable doesn't allow nulls)**.

2. One of the major differences between HashMap and Hashtable is that **HashMap is non-synchronized** whereas **Hashtable is synchronized**, which means Hashtable is thread-safe and can be shared between multiple threads but HashMap can not be shared between multiple threads without proper synchronization. Java 5 introduces ConcurrentHashMap which is an alternative of Hashtable and provides better scalability than Hashtable in Java.



**3**. Another significant difference between HashMap vs Hashtable is that Iterator in the **HashMap is  a fail-fast iterator**  **while the enumerator for the Hashtable is not** and throw ConcurrentModificationException if any other Thread modifies the map structurally  by adding or removing any element except Iterator's own remove() method. **But this is not a guaranteed behavior and will be done by JVM on best effort**. This is also an important difference between Enumeration and Iterator in Java.  
**4**. One more notable difference between Hashtable and HashMap is that because of thread-safety and synchronization Hashtable is much slower than HashMap if used in Single threaded environment. So if we don't need synchronization and HashMap are only used by one thread, it outperforms Hashtable in Java.  
**5**. HashMap does not guarantee that the order of the map will remain constant over time.  
HashMap and Hashtable : note on Some Important Terms

**1**)Synchronized means only one Thread can modify a hash table at one point of time. Basically, it means that any thread before performing an update on a **Hashtable** will have to **acquire a lock on the object whil**e others will wait for the lock to be released.

2) Fail-safe is relevant from the context of iterators. If an Iterator or ListIterator has been created on a collection object and some other thread tries to modify the collection object "structurally", a concurrent modification exception will be thrown. It is possible for other threads though to invoke "set" method since it doesn't modify the collection "structurally". However, if prior to calling "set", the collection has been modified structurally, "IllegalArgumentException" will be thrown.

**3**) Structurally modification means deleting or inserting element which could effectively change the structure of the map.

HashMap can be synchronized by

Map m = **Collections.synchronizeMap(hashMap);**

In Summary, there are significant differences between Hashtable and HashMap in Java e.g. thread-safety and speed and based upon that only use Hashtable if we absolutely need thread-safety if we are running Java 5 consider using ConcurrentHashMap in Java.

**Q. How to Sort HashMap in Java based on Keys and Values**

**Ans**. HashMap is not meant to keep entries in sorted order, but if we have to sort HashMap based upon keys or values, we can do that in Java. **Sorting HashMap on keys is quite easy**, all we need **to do is to create a TreeMap** by copying entries from HashMap. TreeMap is an implementation of SortedMap and keeps keys in their natural order or a custom order specified by Comparator provided while creating TreeMap. This means we can process entries of HashMap in a sorted order but we cannot pass a HashMap containing mappings in a specific order, this is just not possible because HashMap doesn't guarantee any ordering.

On other hand, **sorting HashMap by values** is rather complex because there is no direct method to support that operation. We need to write code for that. In order to sort HashMap by values we can **first create a Comparator,** which can compare two entries based on values.

Then **get the Set of entries from Map**, **convert Set to List** and use **Collections.sort(List) method to sort our list of entries by values by passing our customized value comparator**. This is similar of how we sort an ArrayList in Java. Half of the job is done by now. Now create a new LinkedHashMap and add sorted entries into that. **Since LinkedHashMap guarantees insertion order of mappings,** **we will finally have a Map where contents are sorted by values.**

Steps to sort HashMap by values :-

One difference between sorting HashMap by keys and values is that it can **contain duplicate values** by **not duplicate keys**. We cannot use TreeMap here because it only sort entries by keys. In this case we need to :

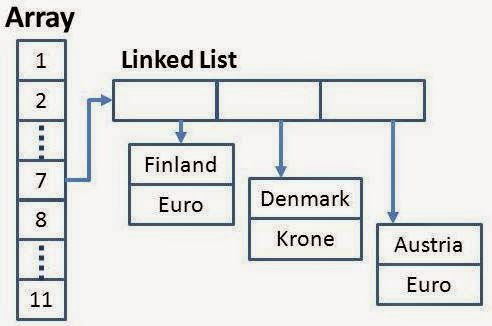
* Get all entries by **calling entrySet() method of Map**
* Create a **custom Comparator to sort entries** based upon values
* Convert **entry set to list**
* **Sort entry list by using Collections.sort()** method by passing our value comparator
* **Create a LinkedHashMap** by adding entries in sorted order.

Steps to sort HashMap by keys:-

There are two ways to sort HashMap by keys, **first by using TreeMap** and **second by using LinkedHashMap**. If we want to sort using TreeMap then it's simple, just create a TreeMap by copying content of HashMap.

On the other hand, if we want to create a LinkedHashMap then we first need to get key set, convert that **Set to List, sort that List** and **then add them into** **LinkedHashMap** in same order. Remember HashMap can contain one null key but duplicate keys are not allowed.

HashMap Sorting by Keys and Values in Java Example

[](https://pluralsight.pxf.io/c/1193463/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fjava-fundamentals-collections)

Here is our sample Java program to sort a HashMap first by keys and then by values. This program is divided into two part, first part sorts HashMap by keys and second part sorts it by values. Second part is more tricky then first part as there is no native Map implementation which supports any order for values. In order to sort a HashMap by values we had to create our own Comparator implementation which compares each entries by values to arrange them in a particular order. We can see that our valueComparator overrides comapre() method and accepts two entries. Later it retrieves values from those entries and compare them and return result. Since there is no method in Java Collection API to sort Map, we need to use Collections.sort() method which accepts a List. This involves creating a temporary ArrayList with entries for sorting purpose and then again copying entries from sorted ArrayList to a new LinkedHashMap to keep them in sorted order. Finally we create a HashMap from that LinkedHashMap, which is what we needed.

import java.text.ParseException;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.HashMap;

import java.util.LinkedHashMap;

import java.util.List;

import java.util.Map.Entry;

import java.util.Set;

import java.util.TreeMap;

/\*\*

\* How to sort HashMap in Java by keys and values.

\* HashMap doesn't guarantee any order, so we cannot rely on it, even if

\* it appear that it storing entries in a particular order, because

\* it may not be available in future version e.g. earlier HashMap stores

\* integer keys on the order they are inserted but from Java 8 it has changed.

\*

\* @author WINDOWS 8

\*/

public class HashMapSorting{

public static void main(String args[]) throws ParseException {

// let's create a map with Java releases and their code names

HashMap<String, String> codenames = new HashMap<String, String>();

codenames.put("JDK 1.1.4", "Sparkler");

codenames.put("J2SE 1.2", "Playground");

codenames.put("J2SE 1.3", "Kestrel");

codenames.put("J2SE 1.4", "Merlin");

codenames.put("J2SE 5.0", "Tiger");

codenames.put("Java SE 6", "Mustang");

codenames.put("Java SE 7", "Dolphin");

System.out.println("HashMap before sorting, random order ");

Set<Entry<String, String>> entries = codenames.entrySet();

for(Entry<String, String> entry : entries){

System.out.println(entry.getKey() + " ==> " + entry.getValue());

}

// Now let's sort HashMap by keys first

// all we need to do is create a TreeMap with mappings of HashMap

// TreeMap keeps all entries in sorted order

**TreeMap<String, String> sorted = new TreeMap<>(codenames);**

**Set**<Entry<String, String>> mappings = **sorted.entrySet();**

System.out.println("HashMap after sorting by keys in ascending order ");

for(Entry<String, String> mapping : mappings){

System.out.println(mapping.getKey() + " ==> " + mapping.getValue());

}

// Now let's sort the HashMap by values

// there is no direct way to sort HashMap by values but we

// can do this by writing our own comparator, which takes

// Map.Entry object and arrange them in order increasing

// or decreasing by values.

Comparator<Entry<String, String>> **valueComparator** = new Comparator<Entry<String,String>>() {

@Override

public int **compare**(Entry<String, String> e1, Entry<String, String> e2) {

String v1 = e1.getValue();

String v2 = e2.getValue();

return v1.compareTo(v2);

}

};

// Sort method needs a List, so let's first convert **Set to List** in Java

List<Entry<String, String>> **listOfEntries** = new ArrayList<Entry<String, String>>(**entries**);

// sorting HashMap by values using comparator

Collections.sort(**listOfEntries**, **valueComparator**);

LinkedHashMap<String, String> **sortedByValue = new LinkedHashMap<String, String>(listOfEntries.size());**

// copying entries from List to Map

for(Entry<String, String> entry : listOfEntries){

sortedByValue.put(entry.getKey(), entry.getValue());

}

System.out.println("HashMap after sorting entries by values ");

Set<Entry<String, String>> entrySetSortedByValue = sortedByValue.entrySet();

for(Entry<String, String> mapping : entrySetSortedByValue){

System.out.println(mapping.getKey() + " ==> " + mapping.getValue());

} }}

Output:

HashMap before sorting, random order

Java SE 7 ==> Dolphin

J2SE 1.2 ==> Playground

Java SE 6 ==> Mustang

J2SE 5.0 ==> Tiger

J2SE 1.3 ==> Kestrel

J2SE 1.4 ==> Merlin

JDK 1.1.4 ==> Sparkler

HashMap after sorting by keys in ascending order

J2SE 1.2 ==> Playground

J2SE 1.3 ==> Kestrel

J2SE 1.4 ==> Merlin

J2SE 5.0 ==> Tiger

JDK 1.1.4 ==> Sparkler

Java SE 6 ==> Mustang

Java SE 7 ==> Dolphin

HashMap after sorting entries by values

Java SE 7 ==> Dolphin

J2SE 1.3 ==> Kestrel

J2SE 1.4 ==> Merlin

Java SE 6 ==> Mustang

J2SE 1.2 ==> Playground

JDK 1.1.4 ==> Sparkler

J2SE 5.0 ==> Tiger

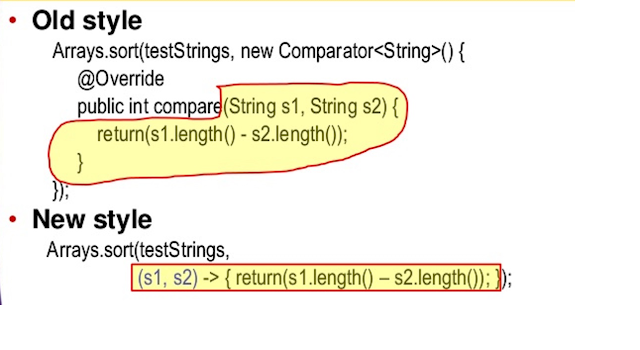
That's all about how to sort HashMap by keys and values in Java. Remember, HashMap is not intended to keep entries in sorted order, so if we have requirement to always keep entries in a particular order, don't use HashMap instead use TreeMap or LinkedHashMap. This method should only be used to cater adhoc needs where we receive a HashMap from some part of legacy code and we have to sort it first to process entries. If we have control of creating the Map initially prefer the right implementation of Map then just HashMap.

In order to sort an ArrayList of objects, we need two things, first a class to provider ordering and a method to provide sorting. If we know about ordering and sorting in Java then we know that Comparable and Comparator class is used to provide the ordering for objects. The Comparable interface provides natural order e.g. lexicographic order of String or name for Employees, while Comparator is used to provide custom order. It gives we the flexibility to sort our objects on the parameter we want e.g. we can sort a list of Coupon objects on the percentage discount, expiry dates, or based upon the total cost.  
  
Once we have got ordering for our object, we can use Collections.sort() method to sort our ArrayList of objects. This method accepts an ArrayList and sorts them in place. Internally it uses MergeSort to sort our list of objects. This method is a good example of strategy pattern because it uses the Comparator we provide to sort our objects, which means we can sort the same list of objects into different order by just changing the Comparator.  
For example, We can sort an ArrayList of Objects in descending order by just reversing the order of our Comparator. The JDK API provides another convenient **method Collections.reverseOrder(Comparator c**) which return a comparator with the opposite order of given Comparator.  
Btw, this is not the only way to sort an ArrayList of objects in Java. If we want we can use Comparable to sort in the natural order and in the decreasing order of natural order imposed by Comparator. the JDK 8 release also added a couple of methods on both java.util.Comparator class and java.util.List to make sorting easier.  
For example, we can also use List.sort() method to sort a List of objects. This is similar to Collections.sort() and we can use it to sort a List of objects using both Comparator and Comparable. This method accepts a Comparator and sort elements based upon that, but if we want to sort on the natural order, just don't supply a Comparator and pass null.  
The List.sort() of Java 8 will then sort the list on order imposed by Comparable.

Sorting a List of Objects using Comparator

In this article, I'll show we how to sort a List of objects in both ascending and descending order by using Comparator. I have a domain object called Course, which contains title and price of the course, a title is String and fee are long value. In order to sort the list of courses e.g. REST With Spring, Learn Spring Security, and Introduction to Spring MVC 4, some of my recommended course to learn Spring, I have created two Comparators, a title comparator which sorts based upon title and a fee comparator which sorts based upon fee. These Comparator implementation classes use Java 8 lambdas to implement compare() method as shown here, and sort the list of objects into ascending order of title and fee.  
In order to sort in the reverse order i.e. descending order, we don't need to create a separator Comparator, instead, we just need to reverse the order of existing comparator using Collections.reverseOrder(Comparator c) method. If we are using Java 8, then we can also use the reversed() method of java.util.Comparator which returns a comparator that imposes the reverse ordering of this comparator.  
In fact, JDK 8 has added several new methods to facilitate sophisticated sorting in Java 8 e.g. comparing() and thenComparing() method to chain multiple comparators.

Here is our Java program which will sort the list of courses using custom comparators. In this article, I have created two implementations of java.util.Comparator interface, one called TitleComparator, which sort the list of courses on their title and other, called FeeComparator which sorts the list of courses on their price.  
I have also used new List.sort() method of JDK 8 for sorting. Though, we can use Collection.sort() if we are not using JDK 8, just replace the List.sort() with Collections.sort() and the program should work fine.  
Also, I have used the lambda expression to implement our Comparators as oppose to Anonymous class which we used earlier. We can see our Comparator implementation has become quite simple and we can write them in just one line.



Java Program to sort an ArrayList using Comparator  
import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.Iterator;

import java.util.List;

import java.util.stream.Collectors;

/\*

\* Java Program to sort an ArrayList with objects using Comparator

\*/

public class Main {

public static void main(String[] args) {

// sorting an ArrayList of object using Comparator

Course restWithSpring = new Course("REST with Spring", 99);

Course learnSpringSecurity = new Course("Learn Spring Security", 110);

Course introToSpringMVC4 = new Course("Introduction to Spring MVC 4", 0);

List<Course> **listOfCourses** = new ArrayList<>();

listOfCourses.add(restWithSpring);

listOfCourses.add(learnSpringSecurity);

listOfCourses.add(introToSpringMVC4);

// let's sort this list of course by title first using Comparator

Comparator<Course> titleComparator = (c1, c2) -> c1.title().compareTo(c2.title());

Comparator<Course> feeComparator = (c1, c2) -> (int) (c1.fee() - c2.fee());

// printing ArrayList before sorting

System.out.println("unsorted list: " + listOfCourses);

// sorting list of objects using comparator - using title on ascending order

listOfCourses**.sort(titleComparator);**

// printing ArrayList after sorting in ascending order

System.out.println("sorted list in ascending order of title: " + listOfCourses);

// sorting arraylist of objects using comparator - using fee on ascending order

listOfCourses.**sort(feeComparator**);

// printing ArrayList after sorting in ascending order

System.out.println("sorted list in ascending order of fee: " + listOfCourses);

// sorting array list in descending order of title

listOfCourses.**sort(Collections.reverseOrder(titleComparator**));

System.out.println("sorted list in descending order of title: " + listOfCourses);

// sorting arraylist in descending order of fee

listOfCourses.sort(Collections.reverseOrder(feeComparator));

System.out.println("sorted list in descending order of fee: " + listOfCourses);

}

}

class Course{

String title;

long fee;

public Course(String title, long fee){

this.title = title;

this.fee = fee;

}

public String title(){

return title;

}

public long fee(){

return fee;

}

@Override

public String toString() {

return String.format(title + "@ " + fee);

}

}

Output

unsorted list: [REST with Spring@ 99, Learn Spring Security@ 110,

Introduction to Spring MVC 4@ 0]

sorted list in ascending order of title: [Introduction to Spring MVC 4@ 0,

Learn Spring Security@ 110, REST with Spring@ 99]

sorted list in ascending order of fee: [Introduction to Spring MVC 4@ 0,

REST with Spring@ 99, Learn Spring Security@ 110]

sorted list in descending order of title: [REST with Spring@ 99,

Learn Spring Security@ 110, Introduction to Spring MVC 4@ 0]

sorted list in descending order of fee: [Learn Spring Security@ 110,

REST with Spring@ 99, Introduction to Spring MVC 4@ 0]  
From the output, it's clear that our sorting works as expected. In the ascending order of title, Introduction to Spring MVC 4 comes first because it starts with the letter "I", while other starts with letter "L" and letter "R". In the descending order, the "REST with Spring" comes first because of the same reason. While, in the ascending order of fee, again, Introduction to Spring MVC 4 comes first because it is the least expensive out of these three courses.  
  
That's all about how to sort an ArrayList of Objects using Comparator in Java. We have learned to sort ArrayList in both ascending and descending order using Comparator. As I said, we don't need to create two comparators, instead, we just create a comparator to sort the list of objects based upon any attribute we want and then use the Collections.reverseOrder() method to reverse the order imposed by Comparator. It accepts a comparator and then sorts the elements in the array list in the reverse order of that comparator. This way we can sort the list of objects in descending order.

**Q. Difference between ArrayList and** **HashSet in Java**

**Ans**. Main difference between ArrayList and HashSet is that one is a List implementation while **HashSet** is a Set implementation. It means all the differences between a List data structure and a Set data structure also applies to this pair.

For example, **List implementations are ordered**, it store element in the order they were added, while **Set implementation doesn't provide such guarantee.**

Similarly, since **List provides Random access**, we can access any element directly if we know the **index**, **but Set doesn't provide such facility**. We need to Iterate through whole collection to get access of any elements.

Similarities ArrayList and HashSet

1) Both ArrayList and HashSet are non synchronized collection class and not meant to be used in multi-threading and concurrent environment.

We can  make ArrayList and HashSet synchronized by using **Collections.synchroinzedCollection()** just like we make ArrayList and HashSet read only other day.

2) Both ArrayList and HashSet can be traversed using Iterator. This is in fact a preferred way if we want to perform operation on all elements.

3) Iterator of ArrayList and HashSet both are fail-fast, i.e. they will throw ConcurrentModificationException if ArrayList or HashSet is modified structurally once Iterator has been created.

**Difference between ArrayList vs HashSet in Java**

Here are couple of differences between ArrayList and HashSet in Java:

**1**) First and most important difference between ArrayList and HashSet is that **ArrayList implements List interface** while **HashSet implements Set interface** in Java.

**2**) Another difference between ArrayList and HashSet is that **ArrayListallow duplicates** while **HashSet does not allow duplicates**. This is the side effect of fist difference and property of implementing List and Set interface.

**3**) Third difference between ArrayList and HashSet is that **ArrayList is an ordered collection and maintains insertion order of elements** while **HashSet is an unordered collection and doesn't maintain any order.**

**4**) Fourth difference between ArrayList and HashSet is that **ArrayList is backed by an Array** while **HashSet is backed by an HashMap instance**.

**5**) Fifth difference between HashSet and **ArrayList is that its index based we can retrieve object by calling get(index) or remove objects by calling remove(index)** while HashSet is completely object based. HashSet also doesn't provide get() method.

**Q. Difference between TreeSet and TreeMap in Java** ?  
**Ans**. Main Difference between TreeMap and TreeSet is that TreeMap is an implementation of Map interface while TreeSet is an implementation of Set interface.

Similarities between TreeMap and TreeSet in Java

Here is a list of similarities between TreeMap and TreeSet in Java:

1) **Both TreeMap and TreeSet are sorted data structure,** which means they keep there element in predefined Sorted order. Sorting order can be natural sorting order defined by Comparable interface or custom sorting Order defined by Comparator interface. **Both TreeMap and TreeSet has overloaded constructor which accept a Comparator**, if provided all elements inside TreeSet or TreeMap will be compared and Sorted using this Comparator.

2) Both TreeSet and TreeMap implements base interfaces e.g. **TreeSet implements Collection and Set interface** so that they can be passed to method where a Collection is expected and **TreeMap implements java.util.Map interface**, which means we can pass it when a Map is expected.  
3) **TreeSet is practically implemented using TreeMap instance, similar to HashSet which is internally backed by HashMap instance.** See my post Internal Implementation of HashSet to learn more.

4) **Both TreeMap and TreeSet are non synchronized Collection**, hence can not be shared between multiple threads. We can make both TreeSet and TreeMap synchronized by wrapping them into Synchronized collection by calling Collections.synchroinzedMap() method.

5) **Iterator returned by TreeMap and TreeSet are fail-fast, means they will throw ConcurrentModificationException** when TreeMap or TreeSet is modified structurally once Iterator is created. this fail-fast behavior is not guaranteed but works in best effort.

6) **Both TreeMap and TreeSet are slower than there Hash counter part** like HashSet and HashMap and instead of providing constant time performance for add, remove and get operation they provide performance in O(log(n)) order.

**differences between TreeSet vs TreeMap in Java:**

1) Major difference between TreeSet and TreeMap is that **TreeSet implements Set interface** while **TreeMap implements Map interface** in Java.

2) Second difference between TreeMap and TreeSet is the way they store objects. **TreeSet stores only one object** while **TreeMap uses two objects called key and Value**. objects **in TreeSet are sorted** while **keys in TreeMap remain in sorted Order.**

3) Third difference between TreeSet and TreeMap is that, **TreeSet  implements NavigableSet** **while TreeMap  implements NavigableMap** in Java.

4) Fourth difference is that **duplicate objects are not allowed in TreeSet** **but duplicates values are allowed in TreeMap.**

**Q. Difference between HashMap and LinkedHashMap in Java**

**Ans**. HashMap and LinkedHashMap are two of the most common used Map implementation in Java. 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**. Some time we notice that HashMap also returns elements in order e.g. before Java 8 when we use Integer key and then iterate over Map, we would see it returning entries in a particular order, but those are not guaranteed. Any code which is dependent upon ordering provided by HashMap will likely to break in future release when those behavior changes.

LinkedHashMap and HashMap in Java - **Similarities**

1) 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.  
2) Iterator returned by HashMap and LinkedHashMap are fail-fast in nature.  
3) Performance of HashMap and LinkedHashMap are similar also.

**Difference** between LinkedHashMap and HashMap in Java

1) **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.  
2) **Re-entering a mapping, doesn't alter insertion order of LinkedHashMap**. For example, if we 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.  
3) **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.  
4) **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.  
5) Iterator of LinkedHashMap returns elements in the order e.g. either insertion order or access order.  
6)  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 we 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.

**Q.Difference between HashSet and TreeSet in Java**

**Ans**. Probably the most important difference between HashSet and TreeSet is the **performance**. **HashSet is faster than TreeSet**which means if we need performance use HashSet but HashSet doesn't provide any kind of ordering so if we need ordering then we need to switch to TreeSet which provides sorting of keys. Sorting can be natural order defined by a Comparable interface or any particular order defined by Comparator interface in Java.

**Common** in HashSet and TreeSet in Java:

1)Both HashSet and TreeSet implements java.util.Set interface which means they follow contract of Set interface and doesn't allow any duplicates.

2)Both HashSet and TreeSet are not thread-safe and not synchronized. Though we can make them synchronized by using Collections.synchronizedSet() method.

3) The third similarity between TreeSet and HashSet is that Iterator of both classes is fail-fast in nature. They will throw ConcurrentModificationException if Iterator is modified once Iterator is created. this is not guaranteed and application code should not rely on this code but Java makes best effort to fail as soon as it detects a structural change in underlying Set.

Differences between HashSet vs TreeSet in Java.

**1**) The first major difference between HashSet and TreeSet is performance. HashSet is faster than TreeSet and HashSet  should be preferred choice if sorting of elements is not required. TreeSet is internally backed by a Red-Black tree. The performance difference comes from the underlying data structure used by TreeSet and HashSet i.e. a tree and a hash table. Adding an element of a tree is slower than adding it to a hash table but it is still much faster than adding it into the right place in the linked list or array. If the tree contains n elements, then an average log2N comparisons are required to find the correct position for a new element. For example, if the tree contains 1000 elements than adding a new element requires about 10 comparisons.  
**2**) Second difference between HashSet and TreeSet is that HashSet allows null object but TreeSet doesn't allow null Object and throw NullPointerException, Why, because TreeSet uses compareTo() method to compare keys and compareTo() will throw java.lang.NullPointerException as shown in below example :

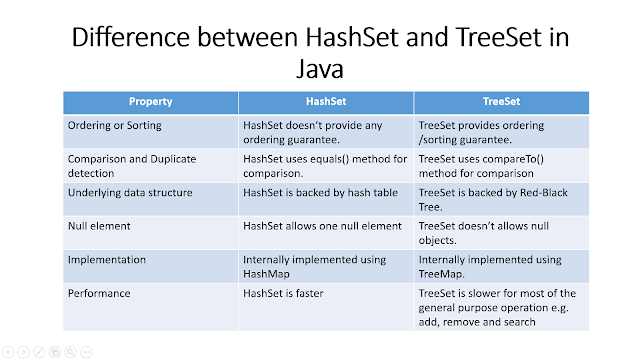
HashSet<String> hashSet = new HashSet<String>();  
hashSet.add("Java");  
hashSet.add(null);  
         
TreeSet<String> treeSet = new TreeSet<String>();  
treeSet.add("C++");  
treeSet.add(null); //Java.lang.NullPointerException  
Output:  
Exception in thread "main" java.lang.NullPointerException  
        at java.util.TreeMap.put(TreeMap.java:541)  
        at java.util.TreeSet.add(TreeSet.java:238)  
        at test.CollectionTest.main(CollectionTest.java:27)  
Java Result: 1

**3**) Another significant difference between HashSet and TreeSet is that HashSet is backed by HashMap while TreeSet is backed by TreeMap in Java.

**4**) One more difference between HashSet and TreeSet which is worth remembering is that HashSet uses equals() method to compare two objects in Set and for detecting duplicates while TreeSet uses compareTo() method for the same purpose. if equals() and compareTo() are not consistent, i.e. for two equal object equals should return true while compareTo() should return zero then it will break the contract of Set interface and will allow duplicates in Set implementations like TreeSet

**5**) Now the most important difference between HashSet and TreeSet is ordering. HashSet doesn't guarantee any order while TreeSet maintains objects in the Sorted order defined by either Comparable or Comparator method in Java.

Here is a nice summary slide of key differences between TreeSet and HashSet in Java, which compares both of these collections on ordering, sorting, performance, underlying data structure, the method used for duplicate detection and how they are implemented in JDK.

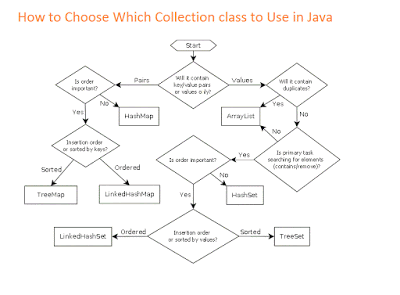


**Q. When to use ArrayList vs LinkedList in Java**

**Ans**. Being List implementation both ArrayList and LinkedList are ordered, the index-based and allows duplicate.

The main **difference** between ArrayList vs LinkedList is that ArrayList is backed by an array while LinkedList is based upon linked list data structure, which makes the performance of add(), remove(), contains() and iterator() different for both ArrayList and LinkedList.

Common between ArrayList and LinkedList in Java :  
**1**) Both ArrayList and LinkedList are an implementation of List interface, which means we can pass either ArrayList or LinkedList if a method accepts the java.util.List interface.



**2**) Both ArrayList and LinkedList are not synchronized, which means we can not share them between multiple threads without external synchronization. See here to know How to make ArrayList synchronized in Java.  
**3**) ArrayList and LinkedList are ordered collection e.g. they maintain insertion order of elements i.e. the first element will be added to the first position.  
**4**) ArrayList and LinkedList also allow duplicates and null, unlike any other List implementation e.g. Vector.  
**5**) An iterator of both LinkedList and ArrayList are fail-fast which means they will throw ConcurrentModificationException if a collection is modified structurally once the Iterator is created. They are different than CopyOnWriteArrayList whose Iterator is fail-safe.

Difference between LinkedList and ArrayList in Java

**1**) The first difference between ArrayList and LinkedList comes with the fact that ArrayList is backed by Array while LinkedList is backed by LinkedList. This will lead to further differences in performance.  
2) Another difference between ArrayList and LinkedList is that apart from the List interface, LinkedList also implements the Deque interface, which provides first in first out operations for add() and poll() and several other Deque functions.

Also, LinkedList is implemented as a doubly-linked list and for index-based operation, navigation can happen from either end .  
3) Adding an element in ArrayList is O(1) operation if it doesn't trigger re-size of Array, in which case it becomes O(log(n)), On the other hand, appending an element in LinkedList is O(1) operation, as it doesn't require any navigation.

4) In order to remove an element from a particular index e.g. by calling remove(index), ArrayList performs a copy operation which makes it close to O(n) while LinkedList needs to traverse to that point which also makes it O(n/2), as it can traverse from either direction based upon proximity.  
5.Iteration is the O(n) operation for both LinkedList and ArrayList where n is a number of an element.  
6) The get(index) operation is O(1) in ArrayList while its O(n/2) in LinkedList, as it needs to traverse till that entry. Though, in Big O notation O(n/2) is just O(n) because we ignore constants there.

**7**) Memory LinkedList uses a wrapper object, Entry, which is a static nested class for storing data and two nodes next and previous while ArrayList just stores data in Array.

So memory requirement seems less in the case of ArrayList than LinkedList except for the case where Array performs the re-size operation when it copies content from one Array to another. If Array is large enough it may take a lot of memory at that point and trigger Garbage collection, which can slow response time.  
From all the above differences between ArrayList vs LinkedList, It looks ArrayList is the better choice than LinkedList in almost all cases, except when we do a frequent add() operation than remove(), or get().

It's easier to modify a linked list than ArrayList, especially if we are adding or removing elements from start or end because the linked list internally keeps references of those positions and they are accessible in O(1) time.

In other words, we don't need to traverse through the linked list to reach the position where we want to add elements, in that case, addition becomes O(n) operation. For example, inserting or deleting an element in the middle of a linked list.

**Q) Difference between poll() and remove() method?**Both poll() and remove() take out the object from the Queue but if poll() fails then it returns null but if remove fails it throws Exception.  
**Q) Which kind of tree is used to implement TreeMap in Java?**A Red Black tree is used to implement TreeMap in Java.  
**Q) Can I write my own container class and use it in the for-each loop?  
Yes**, we can write our own container class. We need to implement the Iterable interface if we want to loop over advanced for loop in Java, though. If we implement Collection then we by default get that property.  
**Q) Is it possible for two unequal objects to have the same hashcode?  
Yes**, two unequal objects can have same hashcode that's why collision happen in a hashmap.  
the equal hashcode contract only says that two equal objects must have the same hashcode it doesn't say anything about the unequal object.  
**Q) Can two equal object have the different hash code?  
No**, that is not possible according to hash code contract.

**Q. How to override hashcode in Java example**

Equals and hashcode methods are two primaries but yet one of the most important methods for java developers to be aware of. Java intends to provide equals and hashcode for every class to test equality and to provide a hash or digest based on the content of the class. The importance of hashcode increases when we use the object in different collection classes which works on hashing principle e.g. hashtable and hashmap. A well-written hashcode method can improve performance drastically by distributing objects uniformly and avoiding a collision.  
**General Contracts for hashCode() in Java**

1) If two objects are equal by equals() method then there hashcode returned by hashCode() method **must be the same**.

2) Whenever the hashCode() method is invoked on the same object more than once within a single execution of the application, hashCode() must return the same integer provided no information or fields used in equals and hashcode is modified. This integer is not required to be the same during multiple executions of application though.

3) If two objects are not equaled by equals() method it is not require that there hashcode must be different. Though it’s always good practice to return different hashCode for unequal object. Different hashCode for distinct object can improve performance of hashmap or hashtable by reducing collision.

To better understand concept of equals and hashcode and what happens if we don’t override them properly I would recommend understanding of How HashMap works in Java

**Overriding hashCode method in Java**

We will follow step by step approach for overriding hashCode method. This will enable us to understand the concept and process better.

1) Take a prime hash e.g. 5, 7, 17 or 31 (prime number as hash, results in distinct hashcode for distinct object)

2) Take another prime as multiplier different than hash is good.

3) Compute hashcode for each member and add them into final hash. Repeat this for all members which participated in equals.

4) Return hash

  Here is an example of hashCode() method

   @Override

    public int hashCode() {

        int hash = 5;

        hash = 89  hash + (this.name != null ? this.name.hashCode() : 0);

        hash = 89  hash + (int) (this.id ^ (this.id >>> 32));

        hash = 89  hash + this.age;

        return hash;

    }

It’s always good to check null before calling hashCode() method on members or fields to avoid NullPointerException, if member is null than return zero. Different data types has different way to compute hashCode.Integer members are simplest we just add there value into hash, for other numeric data-type are converted into int and then added into hash. Joshua bloach has full tables on this. I mostly relied on IDE for this.

Better way to override equals and hashCode

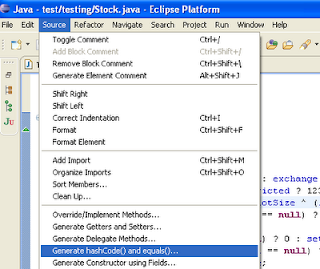
In my opinion, a better way to override both equals and hashcode methods should be left to IDE. I have seen Netbeans and Eclipse and found that both have excellent support of generating code for equals and hashcode and their implementations seem to follow all best practices and requirement e.g. null check, instanceof check, etc and also frees we to remember how to compute hashcode for different data-types.

Let’s see how we can override the hashcode method in Netbeans and Eclipse.

In Netbeans

1) Write wer Class.

2) Right click + insert code + Generate equals() and hashCode().

[](http://javarevisited.blogspot.com/2011/08/what-is-polymorphism-in-java-example.html)

In Eclipse

1) Write to our Class.

2) Go to Source Menu + Generate hashCode() and equals()

1. Whenever we override equals method, hashcode should be overridden to be in compliance with equals hashcode contract.

2. hashCode() is declared in Object class and return type of hashcode method is int and not long.

3. For an immutable object, we can cache the hashcode once generated for improved performance.

4. Test our hashcode method for equals hashcode compliance.

5. If we don't override hashCode() method properly our Object may not function correctly on hash-based collection e.g. HashMap, Hashtable or HashSet.

Things to remember while overriding hashcode in Java

A complete example of equals and hashCode

public class Stock {

       private String symbol;

       private String exchange;

       private long lotSize;

       private int tickSize;

       private boolean isRestricted;

       private Date settlementDate;

       private BigDecimal price;

       @Override

       public int hashCode() {

              final int prime = 31;

              int result = 1;

              result = prime \* result

                           + ((exchange == null) ? 0 : exchange.hashCode());

              result = prime \* result + (isRestricted ? 1231 : 1237);

              result = prime \* result + (int) (lotSize ^ (lotSize >>> 32));

              result = prime \* result + ((price == null) ? 0 : price.hashCode());

              result = prime \* result

                           + ((settlementDate == null) ? 0 : settlementDate.hashCode());

              result = prime \* result + ((symbol == null) ? 0 : symbol.hashCode());

              result = prime \* result + tickSize;

              return result;

       }

       @Override

       public boolean equals(Object obj) {

              if (this == obj) return true;

              if (obj == null || this.getClass() != obj.getClass()){

                     return false;

              }

              Stock other = (Stock) obj;

return

this.tickSize == other.tickSize && this.lotSize == other.lotSize &&

this.isRestricted == other.isRestricted &&

(this.symbol == other.symbol|| (this.symbol != null && this.symbol.equals(other.symbol)))&&

(this.exchange == other.exchange|| (this.exchange != null && this.exchange.equals(other.exchange))) &&

(this.settlementDate == other.settlementDate|| (this.settlementDate != null && this.settlementDate.equals(other.settlementDate))) &&

(this.price == other.price|| (this.price != null && this.price.equals(other.price)));

                 }

}

Writing equals and hashcode using Apache Commons EqualsBuilder and HashCodeBuilder

EqualsBuilder and HashCodeBuilder from Apache commons are a much better way to override equals and hashcode method, at least much better than ugly equals, hashcode generated by Eclipse. I have written the same example by using HashCodebuilder and EqualsBuilder and now we can see how clear and concise they are.

    @Override

    public boolean equals(Object obj){

        if (obj instanceof Stock) {

            Stock other = (Stock) obj;

            EqualsBuilder builder = new EqualsBuilder();

            builder.append(this.symbol, other.symbol);

            builder.append(this.exchange, other.exchange);

            builder.append(this.lotSize, other.lotSize);

            builder.append(this.tickSize, other.tickSize);

            builder.append(this.isRestricted, other.isRestricted);

            builder.append(this.settlementDate, other.settlementDate);

            builder.append(this.price, other.price);

            return builder.isEquals();

        }

        return false;

    }

    @Override

    public int hashCode(){

        HashCodeBuilder builder = new HashCodeBuilder();

        builder.append(symbol);

        builder.append(exchange);

        builder.append(lotSize);

        builder.append(tickSize);

        builder.append(isRestricted);

        builder.append(settlementDate);

        builder.append(price);

        return builder.toHashCode();

    }

    public static void main(String args[]){

        Stock sony = new Stock("6758.T", "Tkyo Stock Exchange", 1000, 10, false, new Date(), BigDecimal.valueOf(2200));

        Stock sony2 = new Stock("6758.T", "Tokyo Stock Exchange", 1000, 10, false, new Date(), BigDecimal.valueOf(2200));

        System.out.println("Equals result: " + sony.equals(sony2));

        System.out.println("HashCode result: " + (sony.hashCode()== sony2.hashCode()));

    }

The only thing to concern is that it adds a dependency on apache-commons jar, most people use it but if we are not using than we need to include it for writing equals and hashcode method.

**Q. Difference between Comparator and Comparable in Java -**

**Ans**. Comparator and Comparable are two interfaces in Java API, which is used to compare two objects in Java. Though both are used for comparison there are some differences between them, a major difference between Comparable and Comparator is that former is used to define the natural ordering of object e.g. lexicographic order for java.lang.String, while later is used to define any alternative ordering for an object.  The main usage of java.lang.Comparable and java.util.Comparator interface is for sorting a list of objects in Java. For example to sort a list of Employee by their Id, we can use Comparable interface and provide additional sorting capability, we can define multiple comparators e.g. AgeComparator to compare the age of the employee, SalaryComparator to compare the salary of employees etc.  This brings another important difference between Comparator and Comparable interface in Java, we can have only one ordering via Comparable e.g. natural ordering while we can define multiple Comparator for alternative ordering as discussed above.  
  
Coming to Interviews, this question is very common in 2 to 3 years experience Java interviews, and we just can't afford to not prepare this. It's definitely possible to achieve years of experience in Java, without writing our own Comparator or Comparable, especially if we are not doing active development or coding, but even though, we must know basics e.g. equals and hashcode, compareTo() and compare().  
  
In this article, we will see some notable difference between Comparator vs Comparable in Java from interview perspective. If we are preparing for Java Interview, we may like to see this nice book, Java Programming Interview Exposed by Markham, an excellent book with Java interview questions for both intermediate and experienced Java programmers.  
**Comparator vs Comparable in Java**

During interviews, we can face this question differently, if we are giving telephonic round then it's mostly fact based i.e. we need to mention key points about both interfaces, while in face to face interviews or during written test, we might be asked to write code to sort objects using Comparator and Comparable e.g. sorting employee by name or branch. I have already discussed the second part of this question here and we will only see fact based differences in this post.  
  
1. Comparator interface is in java.util package, which implies it's a utility class, while Comparable interface is kept on java.lang package, which means it's essential for Java objects.  
  
2. Based on syntax, one difference between Comparable and Comparator in Java is that former gives us compareTo(Object toCompare), which accepts an object, which now uses Generics from Java 1.5 onwards, while Comparator defines compare(Object obj1, Object obj2) method for comparing two object.  
  
3. Continuing from the previous difference between Comparator vs Comparable later is used to compare current object, represented by this keyword, with another object, while Comparator compares two arbitrary objects passed to compare() method in Java.  
  
4. One of the key difference between Comparator and Comparable interface in Java is that, We can only have one compareTo() implementation for an object, while we can define multiple Comparator for comparing objects on different parameters e.g. for an Employee object, we can use compareTo() method to compare Employees on id,  known as natural ordering, while multiple compare() method to order employee on age, salary, name and city. It's also a best practice to declare Comparator as nested static classes in Java, because they are closely associated with objects they compare.  
  
5. Many Java classes, which make uses of Comparator and Comparable defaults to Comparable and provided overloaded method to work with arbitrary Comparator instance e.g. Collections.sort() method, which is used to sort Collection in Java has two implementation, one which sort object based on natural order i.e. by using java.lang.Comparable and other which accepts an implementation of java.util.Comparator interface.  
  
6. One more key thing, which is not a difference but worth remembering is that both compareTo() and compare() method in Java must be consistent with equals() implementation i.e. if two methods are equal by equals() method than compareTo() and compare() must return zero. Failing to adhere this guideline, our object may break invariants of Java collection classes which rely on compare() or compareTo() e.g. TreeSet and TreeMap.  
That's all on difference between Comparator and Comparable in Java. Always remember that java.lang.Comparable is used to define natural ordering of an object, while java.util.Comparator can be used to define any ordering based upon our requirements. We can define only one ordering with Comparable but can have multiple Comparators. I strongly suggest to look my example of comparator vs comparable to gain more insight on how to use really use it in code. From Java interview point of view, I think, we are good to go if we know this much details.

**Q.Why Override equals, hashcode and toString method in Java**

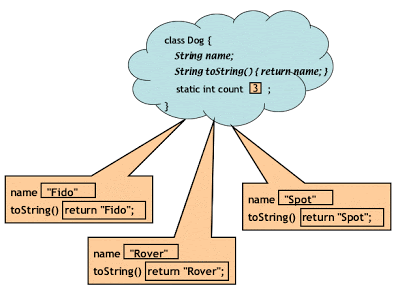
A couple of questions, which are often asked to me was why do we need to override equals() and hashcode() method, Why should I implement toString(), What will happen if I don't override them or in a different way, I have never overridden equals and hashcode and not faced any problem, why should I override them now. We guessed it correct, almost all of these questions come from beginners, who have either taken some Java programming classes or started learning Java by their own.  
Though the concept of equality is something we cannot ignore, correct understanding of equals(), hashcode(), toString(), and some other method from java.lang.Object class, often goes unnoticed, at least until they self-realize it or Interviewer forces them to explore that part.  
I have already written a couple of articles on equals and hashcode, like equals vs ==,  hashcode tips, equals, and hashcode interview questions and one of my personal favorite, 5 tips to override equals in Java, which already touches this subject but I thought to explain it clearly here for once and all.  
And, If we are new to Java world then I also recommend we go through The Complete Java MasterClass on Udemy to learn Java in a better and more structured way. This is one of the best and up-to-date courses to learn Java online.

**Q. Why we should override equals or hashcode**

From the face, we can guess that equals() are used to check if two objects are equal or not. Now, this equality can be defined in two ways, identity equality and logical equality, as I explained in equals vs == post, it's the logical equality, which is taken care of by equals method.  
If we are doing Java programming than we probably know that every class in Java implicitly inherit from java.lang.Object, and from there every object inherit equals() and hashcode(). There default implementation is in line with == operator, i.e. equals() provide identity equality and return true if reference variable pointing to the same object.  
Now, if we don't need logical equality, then we don't need to override equals, but the problem is we will need it. All our domain object e.g. Order, Trade, Message can be compared to each other and we need a logical comparison.  
One of the popular examples is java.lang.String class, which needs logical comparison i.e. character-based comparison. If two String object contains the same characters in the same order they are considered equals, which is what we need in many programming tasks.  
Similarly, all domain object has equality defined, but the true need for equals and hashcode arise, when we use them as key in hash-based collection e.g. Hashtable or HashMap. These collection classes rely on rules of  Java programming around equals and hashcode to work according to their specification, popularly known as an equals-hashcode contract.  
  
According to which, we must override hashcode if we are overriding equals and vice-versa. The problem is that this is not enforced by the compiler, and if we make such a mistake, our program will not work properly.  
For example, any object which doesn't follow equals and hashcode contract, if used as a key in HashMap, we may not be able to retrieve object again, see how HashMap works internally in Java for more details.  
In short, we need to override equals and hashcode, if we are writing a domain object, or we want to store them in the hash-based collection. Once we understand why we should override equals and hashcode, and when we should do that, it's easy to actually do that. See my post 5 tips to override equals and hashcode in Java for more details.

**Q. Why we need to override the toString method**

**Ans** .We should override toString() method for all domain object, because whenever we print them using logger or System.out.println() statements, there toString() method is called. Since default implementation of toString() is not very helpful, and only print classname@hashcode e.g. com.test.User@1033203. If we print some useful information, e.g. Arun, 1022320,  it will only help we during debugging and troubleshooting.  
  
Now there are multiple ways to override toString() in Java, see that link for some easy and productive way. For example, if we print array in Java we will not see any meaningful value, because it doesn't override toString() method, but we can still print arrays by using Arrays.toString() method. This will now show we elements stored in an array, instead of just type of array and it's hashcode.

[](https://3.bp.blogspot.com/-1uQ70MeLtn4/VLPGu_hXpbI/AAAAAAAACWI/NzvZr7W4_TM/s1600/toString%2Bin%2BJava.gif)

I hope this helps we to understand the significance of equals, hashcode, and toString method in Java. In fact all methods of java.lang.Object are worth reading. These are fundamental concepts and solid knowledge of this will only be going to help we, both during the Interview and on Job.  
  
Time spent on understanding java.lang, java.util and java.io are the best investments in learning Java. Always remember to override hashcode() if we are overriding equals() method and vice-versa.  
  
Failing to do so is not a compile-time error but can create really subtle bugs which can take hours to debug and solve, for example, our HashMap reduced to linked list due to frequent collision, we not able to retrieve object put on HashMap, etc.

**Q. Difference between Set, List and Map in Java - Interview question**

**Ans**. Set, List and Map are three important interfaces of Java collection framework and Difference between Set, List, and Map in Java is one of the most frequently asked Java Collection interview question. Sometimes this question is asked as When to use List, Set and Map in Java. Clearly, the interviewer is looking to know that whether we are familiar with fundamentals of Java collection framework or not. In order to decide when to use List, Set or Map, we need to know what are these interfaces and what functionality they provide. List in Java provides ordered and indexed collection which may contain duplicates.   
  
The Set interface provides an unordered collection of unique objects, i.e. Set doesn't allow duplicates, while Map provides a data structure based on key-value pair and hashing.   
  
All three List, Set, and Map are interfaces in Java and there are many concrete implementations of them are available in Collection API. ArrayList and LinkedList are two most popular used List implementation while LinkedHashSet, TreeSet, and HashSet are frequently used Set implementation.   
  
In this Java article, we will see the difference between Map, Set, and List in Java and learn when to use List, Set or Map.

**Set vs List vs Map in Java**

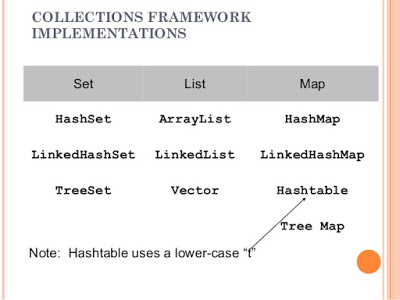
As I said Set, List and Map are interfaces, which defines core contract e.g. a Set contract says that it can not contain duplicates. Based on our knowledge of List, Set and Map let's compare them on different metrics.

Duplicate Objects

The main difference between List and Set interface in Java is that List allows duplicates while Set doesn't allow duplicates. All implementation of Set honor this contract.   
  
While a Map holds two objects per Entry e.g. a key and a value and It may contain duplicate values but keys are always unique. See here for more difference between List and Set data structure in Java.

Order

Another key difference between List and Set is that List is an ordered collection, List's contract maintains insertion order or element. Set is an unordered collection, we get no guarantee on which order element will be stored.   
  
Though some of the Set implementation e.g. LinkedHashSet maintains order. Also SortedSet and SortedMap e.g. TreeSet and TreeMap maintain a sorting order, imposed by using Comparator or Comparable.

[](https://pluralsight.pxf.io/c/1193463/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fjava-8-lambda-expressions-collections-streams)

Null elements

The list allows null elements and we can have many null objects in a List because it also allowed duplicates. Set just allow one null element as there is no duplicate permitted while in Map we can have null values and at most one null key.   
  
Worth noting is that Hashtable doesn't allow null key or values but HashMap allows null values and one null key.  This is also the main difference between these two popular implementations of Map interface, aka HashMap vs Hashtable.

Useful implementations  
Most popular implementations of List interface in Java are ArrayList, LinkedList, and Vector class. ArrayList is more general purpose and provides random access with index, while LinkedList is more suitable for frequently adding and removing elements from List.  
Vector is synchronized counterpart of ArrayList. On the other hand, most popular implementations of the Set interface are HashSet, LinkedHashSet, and TreeSet. First one is general purpose Set which is backed by HashMap, see how HashSet works internally in Java for more details.  
It also doesn't provide any ordering guarantee but LinkedHashSet does provide ordering along with uniqueness offered by the Set interface.  
Third implementation TreeSet is also an implementation of SortedSet interface, hence it keeps elements in a sorted order specified by compare() or compareTo() method.  
Now the last one, most popular implementation of Map interface is HashMap, LinkedHashMap, Hashtable, and TreeMap.  
First one is the non-synchronized general purpose Map implementation while Hashtable is its synchronized counterpart, both doesn' provide any ordering guarantee which comes from LinkedHashMap. Just like TreeSet, TreeMap is also a sorted data structure and keeps keys in sorted order (see Java Fundamentals: Collections)

When to use List, Set and Map in Java

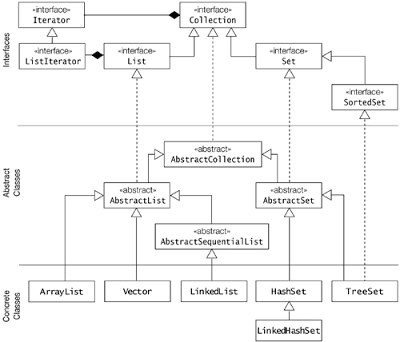
Based upon our understanding of the difference between Set, List and Map we can now decide when to use List, Set or Map in Java.

1) If we need to access elements frequently by using the index than List is a way to go. Its implementation e.g. ArrayList provides faster access if we know index.

2) If we want to store elements and want them to maintain an order on which they are inserted into a collection then go for List again, as List is an ordered collection and maintain insertion order.

3) If we want to create a collection of unique elements and don't want any duplicate than choosing any Set implementation e.g. HashSet, LinkedHashSet or TreeSet.   
All Set implementation follow there general contract e.g. uniqueness but also add addition feature e.g. TreeSet is a SortedSet and elements stored on TreeSet can be sorted by using Comparator or Comparable in Java. LinkedHashSet also maintains insertion order.

4) If we store data in form of key and value than Map is the way to go. We can choose from Hashtable, HashMap, TreeMap based upon our subsequent need. In order to choose between first two see the difference between HashSet and HashMap in Java.

[](https://click.linksynergy.com/fs-bin/click?id=JVFxdTr9V80&subid=0&offerid=323058.1&type=10&tmpid=14538&RD_PARM1=https%3A%2F%2Fwww.udemy.com%2Fjava-the-complete-java-developer-course%2F)

That's all on the difference between Set, List, and Map in Java. All three are the most fundamental interface of Java Collection framework and any Java developer should know there distinguish feature and given a situation should be able to pick right Collection class to use.  We can also further see, Java Fundamentals: Collections, a free course form Pluralsight by Richard Warburton. Pluralsight offers a 10-day free trial period, in which we can access many more advanced Java courses.  
It's also good to remember the difference between there implementation e.g. When to use ArrayList and LinkedList, HashMap vs Hashtable or When to use Vector or ArrayList etc. Collection API is huge and it's difficult to know every bits and piece but at the same time, there is no excuse for not knowing fundamentals like the difference between Set, List, and Map in Java.

**Q. When to use ArrayList vs LinkedList in Java**

**Ans.** ArrayList and LinkedList are two popular concrete implementations of List interface from Java's popular Collection framework. Being List implementation both ArrayList and LinkedList are **ordered**, the **index-based** and allows duplicate.

The main difference between ArrayList vs LinkedList is that ArrayList is backed by an array while LinkedList  is based upon linked list data structure, which makes the performance of add(), remove(), contains() and iterator() different for both ArrayList and LinkedList.

**Q. How HashSet Internally Works in Java?**

**Ans.** Not many programmer know that **HashSet is internally implemented using HashMap** in Java, so if we know How HashMap works internally in Java, more likely we can figure out how HashSet works in Java. But, now a curious Java developer can question that, how come HashSet uses HashMap, because we need a key value pair to use with Map, while in HashSet we only store one object.   
Since HashSet implements Set interface it needs to guarantee uniqueness and this is achieved by storing elements as keys with same value always. Things gets clear by checking HashSet.java from JDK source code.  
  
All we need to look at is, how elements are stored in HashSet and how they are retrieved from HashSet. Since HashSet doesn't provide any direct method for retrieving object e.g. get(Key key) from HashMap or get(int index) from List, **only way to get object from HashSet is via Iterator**. See here for code example of iterating over HashSet in Java.  
When we create an object of **HashSet in Java, it internally create instance of backup Map with default initial capacity 16** and **default load factor 0.75** as shown below :  
  
/\*\*

\* Constructs a new, empty set; the backing <tt>HashMap</tt> instance has

\* default initial capacity (16) and load factor (0.75).

\*/

**public HashSet() {**

**map = new HashMap<>();**

**}**

Now let's see the code for add() and iterate() method from java.util.HashSet in Java to understand how HashSet works internally in Java.  
  
How Object is stored in HashSet

As we can see below, **a call to add(Object) is delegate to put(Key, Value) internally**, **where Key is the object we have passed and value is another object,  called PRESENT, which is a constant in java.util.HashSet as shown below :**

**private transient HashMap<E,Object> map;**

// Dummy value to associate with an Object in the backing Map

**private static final Object PRESENT = new Object();**

public boolean add(E e) {

return map.put(e, PRESENT)==null;

}

**Since PRESENT is a constant, for all keys we have same value in backup HashMap called map.**  
  
**How Object is retrieved from HashSet**  
Now let's see the code for getting iterator for traversing over HashSet in Java. iterator() method from java.util.HashSet class returns iterator for backup Map returned by map.keySet().iterator() method.  
  
       /\*\*

\* Returns an iterator over the elements in this set. The elements

\* are returned in no particular order.

\*

\* @return an Iterator over the elements in this set

\* @see ConcurrentModificationException

\*/

**public Iterator<E> iterator() {**

return **map.keySet().iterator();**

}

How to use HashSet in Java

Using HashSet in Java is very simple, don't think it is Map but think more like Collection i.e. add elements by **using add() method**, check its return value to see if object already existed in HashSet or not. Similarly use iterator for retrieving element from HashSet in Java. We can also use **contains() method** to check if any object already exists in HashSet or not. This method use equals() method for comparing object for matching. We can also use **remove() method to remove** object from HashSet. Since element of HashSet is used as key in backup HashMap, they must implement equals() and hashCode() method. Immutability is not requirement but if its immutable then we can assume that object will not be changed during its stay on set. Following example demonstrate basic usage of HashSet in Java, for more advanced example, we can check this tutorial.

import java.util.HashSet;

import java.util.Iterator;

/\*\*

\* Java Program to demonstrate How HashSet works internally in Java.

\* @author http://java67.blogspot.com

\*/

public class HashSetDemo{

public static void main(String args[]) {

HashSet<String> supportedCurrencies = new HashSet<String>();

// adding object into HashSet, this will be translated to put() calls

supportedCurrencies.add("USD");

supportedCurrencies.add("EUR");

supportedCurrencies.add("JPY");

supportedCurrencies.add("GBP");

supportedCurrencies.add("INR");

supportedCurrencies.add("CAD");

// retrieving object from HashSet

Iterator<String> itr = supportedCurrencies.iterator();

while(itr.hasNext()){

System.out.println(itr.next());

}

}

}

Output

JPY

EUR

INR

USD

CAD

GBP

That's all about How HashSet is implemented in Java and How HashSet works internally. As I said, If we how HashMap internally in Java, we can explain working of HashSet provided,  we know it uses same values for all keys. Remember to override equals() and hashCode() for any object we are going to store in HashSet, since our object is used as key in backup Map, it must override those method. Make our object Immutable or effective immutable if possible.

**Q. How to Remove Objects from Collection or ArrayList in Java while Traversing - Iterator remove() method Example**

**Ans**. Because of every java.util.Collection implementation e.g**. List or Set has remove() method to delete a particular object**, which can be used to remove elements from any Collection e.g. ArrayList, LinkedList, or Vector. Well, this is where things go wrong and interviewers are interested to see, whether we can point about remove() method from Iterator or not.  
The answer to this question is as simple as that, **we should be using Iterator's remove() method to delete any object from Collection we are iterating,** but this is not the end of this question.  
Most likely we will be asked to explain, what is a difference in removing object using remove() method of Collection over remove() method of Iterator and why one should use over other?  
The reason is **ConcurrentModificationException**, **if we use remove() method of List, Set, or basically from any Collection to delete object while iterating, it will throw ConcurrentModificationException.**Though **remove() method of java.util.Collection works fine to remove individual objects, they don't work well when we are iterating over a collection**.  Let's see a code example to clear doubts  
In the below code, we have a list of exchanges and we are removing exchanges which are closed at the moment. Code is as simple as it could be and it's hard to find anything wrong on it by just looking, but things will be different when we run it.  
We will be hit by ConcurrentModificationException, as soon as we run, because here are we are using remove() method of ArrayList to remove objects, instead of Iterator's remove method. In order to fix ConcurrentModificationException, just use remove() method of java.util.Iteator class.

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

/\*\*

  \* Java program to demonstrate how to remove object form List and differnece

  \* between Iterator's remove() method and Colection's remove() method in Java

  \*

  \* @author http://javarevisited.blogspot.com

 \*/

public class ObjectRemovalTest {

    public static void main(String args[]) {

       List markets = new ArrayList();

       StockExchange TSE = new StockExchange(){

            @Override

            public boolean isClosed() {

                return true;

            }

       };

       StockExchange HKSE = new StockExchange(){

            @Override

            public boolean isClosed() {

                return true;

            }

       };

       StockExchange NYSE = new StockExchange(){

            @Override

            public boolean isClosed() {

                return false;

            }

       };

       markets.add(TSE);

       markets.add(HKSE);

       markets.add(NYSE);

       Iterator itr = markets.iterator();

       while(itr.hasNext()){

           StockExchange exchange = itr.next();

           if(exchange.isClosed()){

               markets.remove(exchange); //Use itr.remove() method

           }

       }

    }

}

interface StockExchange{

    public boolean isClosed();

}

Output

Exception in thread "main" java.util.ConcurrentModificationException

        at java.util.AbstractList$Itr.checkForComodification(AbstractList.java:372)

        at java.util.AbstractList$Itr.next(AbstractList.java:343)

        at ObjectRemovalTest.main(StringReplace.java:63)

Java Result: 1

to be frank, even with the use of modern IDE like Eclipse, we may code it wrong, and end up confusing ourself when we see ConcurrentModificationException because that sometimes mislead programmers. It looks that may be another thread is modifying collection, and with this thought, we won't look the code we are using for traversing ArrayList.  
  
That's why sometime interviewer present code and ask we to find bugs on it, or they may simply ask we to write code for removing objects from Collection while traversing over them. **Always remember to use Iterator's remove() method for removing objects from Collection in Java.**

**Q. How to Sort List into Ascending and Descending Order in Java?**

**Ans.** Sorting List, Set and ArrayList in Java on ascending and descending order is very easy, We just need to know correct API method to do that. **Collections.sort()  method** will sort the collection passed to it,  doesn't return anything just sort the collection itself.  Sort() method of Collections class in Java is overloaded where another version takes a Comparator and sort all the elements of Collection on order defined by Comparator.If we  don't pass any Comparator than object will be sorted based upon there natural order like String will be sorted alphabetically or lexicographically. Integer will be sorted numerically etc. Default sorting order for an object is ascending order like Integer will be sorted  from low to high while descending order is just opposite. Collections.reverseOrder() returns a Comparator which will be used for sorting Object in descending order.

**Q. Sorting List in ascending and descending order - Example**

**Ans**. Here full code example of how to sort List in Java on both ascending and descending order in Java. Its not difficult just remember that Collections.sort() is a key method which sort objects of Collection. We can pass it Comparator for defining sorting order. If no comparator passed than Object must implement Comparable interface in order to be sorted.

package example;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.Collections;  
import java.util.List;  
  
/\*\*  
 \* Java program Example of sorting List, ArrayList and Set on ascending and descending order

 \* in Java.  
 \* Collections.sort() method is used for sorting collection and Collections.reverseOrder()  
 \* will sort elements of Collections in descending order.  
 \* @author Javin Paul  
 \*/  
public class CollectionSortingExample{  
       public static void main(String args[]) throws InterruptedException{              
     
        // Sort List and Set in ascending order  
       // Sorting List in Java in ascending order in Java  
        List<Integer> list = Arrays.asList( 1, 2, 5, 9, 3, 6, 4, 7, 8);  
         
        System.out.println("Unsorted List in Java: " + list);

Unsorted List in Java: [1, 2, 5, 9, 3, 6, 4, 7, 8]  
  
         // Sorting List into Java now, Collections.sort()  method will sort

        // the collection passed  
        // to it. Doesn't return anything it just sort the collection itself  
  
       **Collections.sort(list); //**sorting collection  
        System.out.println("List in Java sorted in ascending order: " + list);  
        List in Java sorted in ascending order: [1, 2, 3, 4, 5, 6, 7, 8, 9]  
  
        // sorting List in descending order in Java, Collections.sort() method can be used  
        // to sort all element in descending order if we pass it comparator which can  
        // return descending order for elements. here is an example of sorting List  
        // in descending order  in Java  
        // Collection class provides a in built-in comparator for that which can  
        // sort objects in reverse order i..e descending order for Integers or  
        // any other Object in Java  
        Collections.**sort**(list, **Collections.reverseOrder());**  
        System.out.println("Java List sorted in descending order: " + list);  
         
        // Any List implementation in Java like ArrayList, LinkedList  
        // can be sorted in ascending and descending order in Java by following above  
        // example, let's see a quick example for sorting ArrayList and LinkedList  
        // in ascending and descending order in Java  
                
        //Sorting ArrayList in ascending order in Java  
        ArrayList alphabets = new ArrayList();  
        alphabets.add("c");  
        alphabets.add("b");  
        alphabets.add("a");  
        alphabets.add("d");  
         System.out.println("Unsorted ArrayList in Java : " + alphabets);  
         //Sorting ArrayList into **ascending** order  
        **Collections.sort(alphabets);**         System.out.println("Sorted ArrayList in ascending order in Java : " + alphabets);  
         //Sorting ArrayList into **descending** order or reverse order in Java  
**Collections.sort(alphabets, Collections.reverseOrder());**         
        System.out.println("ArrayList sort in descending order in Java : " + alphabets);        
           }    
 }  
  
  
Output:  
Unsorted List in Java: [1, 2, 5, 9, 3, 6, 4, 7, 8]  
List in Java sorted in ascending order: [1, 2, 3, 4, 5, 6, 7, 8, 9]  
Java List sorted in descending order: [9, 8, 7, 6, 5, 4, 3, 2, 1]  
Unsorted ArrayList in Java : [c, b, a, d]  
Sorted ArrayList in ascending order in Java : [a, b, c, d]  
ArrayList sort in descending order in Java : [d, c, b, a]

As shown in this sort example of ArrayList and Set, We can sort LinkedList on ascending and descending order in Java. Since Collections.sort() method is applicable to Collection interface, any Collection implementation like Set, HashSet can also be sorted in ascending, descending or reverse order of elements. Sorting Array in Java is completely different than sorting collection in Java which we will see in next tutorial. Now we learned how to sort Collection in Java including frequently used collection like ArrayList, List , Set and HashSet.

**Q. How to Print Array with elements in Java?**

We cannot print array elements directly in Java, we need to use Arrays.toString() or Arrays.deepToString() to print array elements. Use toString() if we want to print one-dimensional array and use deepToString() method if we want to print two-dimensional array. Have we tried printing array in Java before? What did we do? just passed an array to println() method and expecting it prints its elements? Me too, but surprisingly array despite being Object and providing a length field, doesn't seem overriding toString() method from java.lang.Object class. All it prints is type@somenumber. This is not at all useful for anyone who is interested in seeing whether an array is empty or not, if not then what elements it has etc.  
  
To print Java array in a meaningful way, we don't need to look further because our very own Collection framework provides lots of array utility methods in java.util.Arrays class. Here we have toString() and deepToString() method to print array in Java.  
These methods are overloaded, much like System.out.println() method to accept all primitive types, which means a different method is called if we pass boolean array, and a different one is called when we print integer array.  
Same is true with deepToString(), which is used to print two dimensional array in Java. In this Java array tutorial, we will see examples of printing string array, integer array, byte array and a two dimensional array in Java. Rest of them are like that, which means by following these examples, we should be able to print boolean, char, short, float, double and long array by our own.  
**Q. How to Print int array in Java**

In order to print integer array, all we need to do is call Arrays.toString(int array) method and pass our integer array to it. This method will take care of printing content of our integer array, as shown below. If we directly pass int array to System.out.println(), we will only see type of array and a random number.

int[] primes = {5, 7, 11, 17, 19, 23, 29, 31, 37};

System.out.println("Prime numbers : " + primes);

System.out.println("Real prime numbers : " + Arrays.toString(primes)); //Ok

Output:

Prime numbers : [I@5eb1404f

Real prime numbers : [5, 7, 11, 17, 19, 23, 29, 31, 37]

**Q.How to Print byte array in Java**

printing byte array is no different than printing int array, as Arrays class provides an overloaded method toString(byte[] bytes) to print contents of byte array in Java, as shown below. By the way, if we want to print byte array as Hex String then see this link.

String random = "In Java programming langue, array is object";

byte[] bytes = random.getBytes();

System.out.println("What is inside bytes : " + bytes);

System.out.println("Not visible, check closely .." + Arrays.toString(bytes));

Output

What is inside bytes : [B@31602bbc

Not visible, check closely ..[73, 110, 32, 74, 97, 118, 97, 32, 112, 114, 111, 103, 114, 97, 109, 109, 105, 110, 103, 32, 108, 97, 110, 97, 103, 117, 101, 44, 32, 97, 114, 114, 97, 121, 32, 105, 115, 32, 111, 98, 106, 101, 99, 116]

**Q.How to Print String array in Java**

**Ans.** Printing string array in Java is probably easiest thing to do, because Arrays class has another overloaded version of toString() to accept Object. This method calls toString() of Object to get a printable String. This can also be used to print array of any arbitrary object in Java. User defined object must override toString() method to show something reasonable on console.

String[] **buzzwords** = {"Java", "Android", "iOS", "Scala", "Python"};

System.out.println("Buzzing .." + buzzwords);

System.out.println("Not buzzing? try again : " + **Arrays.toString(buzzwords));**

Output:

Buzzing ..[Ljava.lang.String;@46f5331a

Not buzzing? try again : [Java, Android, iOS, Scala, Python]

**Q. How to Print Two Dimensional array in Java**

**Ans**. Arrays class provides a different method to print two dimensional array in Java, it’s called **toDeepString().** It's capable of printing multi-dimensional array in Java and similar to toDeepEquals() which is used to compare multi-dimensional array in Java. This method is also overloaded and provides 8 + 1 primitive and object versions to accept boolean, byte, short, char, int, long, float, double and Object in Java. Here is an example of how to print two dimensional array in Java.  
String[][] phones = {{"Apple", "iPhone"}, {"Samsung", "Galaxy"}, {"Sony", "Xperia"}};

System.out.println("Hot phones .. " + phones);

System.out.println("Not hot? See again.." + Arrays.deepToString(phones));

Output

Hot phones .. [[Ljava.lang.String;@57398044

Not hot? See again..[[Apple, iPhone], [Samsung, Galaxy], [Sony, Xperia]]

Complete Java Program to Print Array in Java

This is the full Java code of print different types of array in Java. As explained in this article, it prints integer, String, byte and two dimensional array using toString() and deepToString() method of java.util.Arrays class. We can copy paste this program in our Java IDE and run it. Don't need of any third-party libraries.

import java.util.Arrays;

/\*\*

\* Java Program to print arrays in Java. We will learn how to print String, int,

\* byte and two dimensional arrays in Java by using toString() and

\* deepToString() method of Arrays class.

\*

\* @author http://java67.blogspot.com

\*/

public class PrintArrayInJava{

public static void main(String args[]) {

// Example 1 : print int array in Java

int[] primes = {5, 7, 11, 17, 19, 23, 29, 31, 37};

System.out.println("Prime numbers : " + primes); // Not OK

System.out.println("Real prime numbers : " + Arrays.toString(primes)); //Ok

// Example 2 : print String array in Java

String[] buzzwords = {"Java", "Android", "iOS", "Scala", "Python"};

System.out.println("Buzzing .." + buzzwords);

System.out.println("Not buzzing? try again : " + **Arrays.toString(buzzwords)**);

// Example 3 : print two dimensional array in Java

String[][] phones = {{"Apple", "iPhone"}, {"Samsung", "Galaxy"}, {"Sony", "Xperia"}};

System.out.println("Hot phones .. " + phones);

System.out.println("Not hot? See again.." + **Arrays.deepToString(phones));**

// Example 4 : print byte array in Java

String random = "In Java programming langue, array is object";

byte[] bytes = random.getBytes();

System.out.println("What is inside bytes : " + bytes);

System.out.println("Not visible, check closely .." + Arrays.toString(bytes));

}

}

Output:

Prime numbers : [I@5eb1404f

Real prime numbers : [5, 7, 11, 17, 19, 23, 29, 31, 37]

Buzzing ..[Ljava.lang.String;@46f5331a

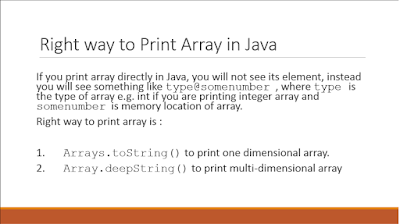
Not buzzing? try again : [Java, Android, iOS, Scala, Python]

Hot phones .. [[Ljava.lang.String;@57398044

Not hot? See again..[[Apple, iPhone], [Samsung, Galaxy], [Sony, Xperia]]

What is inside bytes : [B@31602bbc

Not visible, check closely ..[73, 110, 32, 74, 97, 118, 97, 32, 112, 114, 111, 103, 114, 97, 109, 109, 105, 110, 103, 32, 108, 97, 110, 97, 103, 117, 101, 44, 32, 97, 114, 114, 97, 121, 32, 105, 115, 32, 111, 98, 106, 101, 99, 116]

[](https://4.bp.blogspot.com/-UU6bpwx85jQ/VX0QcmCYJbI/AAAAAAAADDI/C8Tc2r34XTg/s1600/How%2Bto%2Bprint%2BArray%2Belements%2Bin%2BJava.png)

That's all about how to print array in Java. We have learned how to print objects of array, instead of array object, which is nothing but a hashCode. I really hope that Java should add a toString() in Array, instead of providing Arrays.toString(), don't know when they chose other part. Nevertheless, toString() and toDeepString() from java.util.Arrays class is sufficient to print any kind of one dimensional and two dimensional array in Java. Though special care needs to take, while printing byte arrays, which requires byte to be encoded in Hex string.

**Q. Java ArrayList and HashMap Performance Improvement in JDK 7**

**Ans.**   
If we are running on **Java 1.6** or earlier version of **Java 1.7,** we can open the code of java.util.ArrayList and check that, currently empty **ArrayList is initialized with Object array of size 10**.  
  
If we create several temporary lists in our program, which remains uninitialized, due to any reason then we are not only losing memory but also losing performance by giving our garbage collector more work.  
  
The same is true for empty **HashMap**, which was initialized by the **default initial capacity of 16.** These changes are the result of observation made by Nathan Reynolds, and Architect at Oracle, which apparently analyzed 670 Java heap dumps from different Java programs to find out memory hogs.

*Change in ArrayList on Java 7 update 40*

As I said, when we create an empty ArrayList, without specifying any initial capacity i.e. by using new ArrayList(), Java creates an Object array of default size 10 to hold objects. This memory is allocated eagerly, even before we have added any object, which means, if 100K list is created during application runtime, say for storing order details of each order in a transaction processing system, and 10% of them will remain empty then we are going to lose significant memory.

By the way, it's not just memory, it’s also extra work-load for Garbage collector. If we are working in high-frequency trading application development, where every ounce of performance matters or just cares enough for the performance of our Java application, we will appreciate this saving.

Now let's see the actual change :  
java.util.ArrayList code from JDK 1.6.30

Here is the code snippet from java.util.ArrayList class from jdk1.6.30 to create an empty ArrayList :

/\*\*

\* Constructs an empty list with an initial capacity of ten.

\*/

public ArrayList() {

this(10);

}

We can see that it's calling another constructor of java.util.ArrayList with initial capacity 10, which allocates array.

public ArrayList(int initialCapacity) {

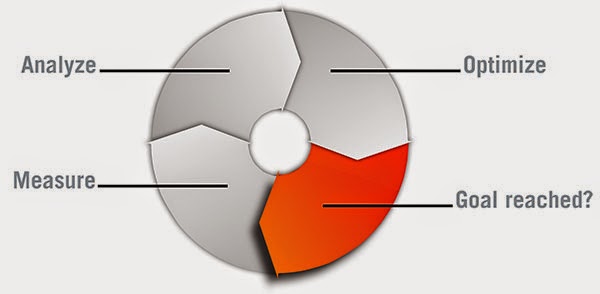
super();

if (initialCapacity < 0)

throw new IllegalArgumentException("Illegal Capacity: "+ initialCapacity);

this.elementData = new Object[initialCapacity];

}

We can see array allocate at last line of the constructor (highlighted by amber).[](https://pluralsight.pxf.io/c/1193463/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Funderstanding-java-vm-memory-management)

java.util.ArrayList code from JDK 1.7.0.\_40

This version of ArrayList has an empty shared array, a static variable that is shared by all instances of the ArrayList class.

/\*\*

\* Shared empty array instance used for empty instances.

\*/

private static final Object[] EMPTY\_ELEMENTDATA = {};

and now look at the change made in no-argument constructor of java.util.ArrayList class

/\*\*

\* Constructs an empty list with an initial capacity of ten.

\*/

public ArrayList() {

super();

this.elementData = EMPTY\_ELEMENTDATA;

}

We can see that, instead of constructor chaining, elementDate is assigned an empty array. This immediately saves memory hogged by an object array of size 10. By the way, how many of we have noticed the same comment "Constructs an empty list with an initial capacity of ten/" in both the version? Yes, they forget to update the comment, and that's one of the reasons, why code comments are bad? they quickly lose relevance, as no compiler is there to verify the correctness of a comment.

Change in **HashMap on JDK 7 updated 40**

A similar change has been made on java.util.HashMap class, earlier it was used to initialized by default size of 16, but now its initialized by an empty table.

java.util.HashMap code from jdk1.6.30

Here is the code for creating empty HashMap in Java 6, we can see that table instance variable is initialized by an Entry array of default initial size 16, highlighted by red :

   /\*\*

\* Constructs an empty <tt>HashMap</tt> with the default initial capacity

\* **(16) and the default load factor (0.75).**

\*/

public HashMap() {

this.loadFactor = DEFAULT\_LOAD\_FACTOR;

threshold = (int) (DEFAULT\_INITIAL\_CAPACITY \* DEFAULT\_LOAD\_FACTOR);

table = new Entry[DEFAULT\_INITIAL\_CAPACITY];

init();

}

java.util.HashMap code from jdk1.7.0.\_40

In this version a special shared empty table has created, it's static final variable, so that all instance of HashMap can share it. Initialization of table is also moved out of constructor to the same line, where table is declared. Here is code snippet from Java 1.7 update 40 :

      /\*\*

\* An empty table instance to share when the table is not inflated.

\*/

static final Entry<?,?>[] EMPTY\_TABLE = {};

/\*\*

\* The table, resized as necessary. Length MUST Always be a power of two.

\*/

transient Entry<K,V>[] table = (Entry<K,V>[]) EMPTY\_TABLE;

This saves memory hogged by an Entry array of size 16. Actual initialization of table is now moved into put(K,V) and putAll(K,V), where inflateTable() method is called to allocate memory, as seen below :

  public V put(K key, V value) {

if (table == EMPTY\_TABLE) {

inflateTable(threshold);

}

.....

}

These change is also documented as part of bug JDK-8011200 - (coll) Optimize empty ArrayList and HashMap, and they have also done a performance test to ensure no side effect on  JDK performance.

That's all about this optimization of empty ArrayList and HashMap in JDK 7, no doubts this is going to save a lot of memory and also reduce garbage collection. Take away from this post is to pay attention on any core library change made on minor Java updates, as we could potentially improve performance of our Java application, just by switching to new JVM. Don't think that because we are not using new features of JDK 7, it's not necessary for we and our project to update to newer Java version. In every Java release, several bugs are fixed and optimizations are done,  and everyone we should take advantage of that.

**Q.What is PriorityQueue or priority queue data structure in Java with Example – Tutorial.**

**Ans.** PriorityQueue is an unbounded Queue implementation in Java, which is based on priority heap. PriorityQueue allows we to keep elements in a particular order, according to there natural order or custom order defined by Comparator interface in Java. Head of priority queue data structure will always contain least element with respect to specified ordering. For example, in this post, we will create a PriorityQueue of Items, which are ordered based upon there price, this will allow us to process Items, starting from lowest price. Priority queue is also very useful in implementing Dijkstra algorithm in Java. We can use to PriorityQueue to keep unsettled nodes for processing.  
One of the key thing to remember about PriorityQueue in Java is that it's Iterator doesn't guarantee any order, if we want to traverse in ordered fashion, better use Arrays.sort(pq.toArray()) method.  
The PriorityQueue is also not synchronized, which means can not be shared safely between multiple threads, instead it's concurrent counterpart PriorityBlockingQueue is thread-safe and should be used in a multithreaded environment.   
Priority queue provides O(log(n)) time performance for common enqueing and dequeing methods e.g. offer(), poll() and add(), but constant time for retrieval methods e.g. peek() and element().

How to use PriorityQueue in Java

Here is one example of using PriorityQueue in Java, as I said earlier, we can use PriorityQueue to consume elements in a particular order, which can be natural ordering or any custom order defined by Comparator provided to PriorityQueue. In this example, we have kept a number of Items in PriorityQueue, whose natural ordering is decided by it's price.   
We can take a look at Item class compareTo method, it's consistent with equals method and also sorts Items based upon there price. I have also create Item as nested static class here. By storing Item in priority queue, we can always retrieve Item with lowest price by using poll() method of Queue interface.

package test;

import java.util.PriorityQueue;

import java.util.Queue;

/\*\*

  \* Java Program to show How to use PriorityQueue in Java. This example also demonstrate

  \* that PriorityQueue doesn't allow null elements and how PriorityQueue keeps elements i.e. ordering.

  \*

  \* @author

 \*/

public class PriorityQueueTest {

    public static void main(String args[]) {

        Queue<Item> items = new PriorityQueue<Item>();

        items.add(new Item("IPone", 900));

        items.add(new Item("IPad", 1200));

        items.add(new Item("Xbox", 300));

        items.add(new Item("Watch", 200));

        System.out.println("Order of items in PriorityQueue");

        System.out.println(items);

        System.out.println("Element consumed from head of the PriorityQueue : " + items.poll());

        System.out.println(items);

        System.out.println("Element consumed from head of the PriorityQueue : " + items.poll());

        System.out.println(items);

        System.out.println("Element consumed from head of the PriorityQueue : " + items.poll());

        System.out.println(items);

        //items.add(null); // null elements not allowed in PriorityQueue - NullPointerException

    }

    private static class Item implements Comparable<Item> {

        private String name;

        private int price;

        public Item(String name, int price) {

            this.name = name;

            this.price = price;

        }

        public String getName() {

            return name;

        }

        public int getPrice() {

            return price;

        }

        @Override

        public boolean equals(Object obj) {

            if (obj == null) {

                return false;

            }

            if (getClass() != obj.getClass()) {

                return false;

            }

            final Item other = (Item) obj;

            if ((this.name == null) ? (other.name != null) : !this.name.equals(other.name)) {

                return false;

            }

            if (this.price != other.price) {

                return false;

            }

            return true;

        }

        @Override

        public int hashCode() {

            int hash = 5;

            hash = 97  hash + (this.name != null ? this.name.hashCode() : 0);

            hash = 97  hash + this.price;

            return hash;

        }

        @Override

        public int compareTo(Item i) {

            if (this.price == i.price) {

                return this.name.compareTo(i.name);

            }

            return this.price - i.price;

        }

        @Override

        public String toString() {

            return String.format("%s: $%d", name, price);

        }

  }

}

Output:

Order of items in PriorityQueue

[Watch: $200, Xbox: $300, IPone: $900, IPad: $1200]

Element consumed from head of the PriorityQueue : Watch: $200

[Xbox: $300, IPad: $1200, IPone: $900]

Element consumed from head of the PriorityQueue : IPone: $900

From the above output, it's clear that PriorityQueue keeps the least value element at the head, where the order is determined using compareTo() method. It doesn't keep all elements in that order though, the only head being least value element is guaranteed.   
This is in fact the main difference between TreeSet and PriorityQueue in Java, the former keeps all elements in a particular sorted order, while priority queue only keeps head in sorted order. Another important point to note is that PriorityQueue  doesn't permit null elements and trying to add null elements will result in NullPointerException, as shown below :

Exception in thread "main" java.lang.NullPointerException

        at java.util.PriorityQueue.offer(PriorityQueue.java:265)

        at java.util.PriorityQueue.add(PriorityQueue.java:251)

        at test.PriorityQueueTest.main(PriorityQueueTest.java:36)

Java Result: 1

Summary

Like any other Collection class, it's worth noting to remember key points about PriorityQueue in Java.

1) PriorityQueue is not synchronized, if thread-safety is requirement use BlockingPriorityQueue.

2) Queue retrieval methods like poll(), peek() and element() access head of Queue, which keeps least element according to specified ordering.

3) Iterator returned by PriorityQueue doesn't offer any ordering traversal guarantee.

4) PriorityQueue doesn't allow null elements, if we try to add null, it will throw java.lang.NullPointerException.

That's all on what is PriorityQueue in Java and How to use them. It's a special class, which can be used in special scenarios, where we need to consume Orders in a particular order, remember BlockingQueue maintains insertion order, but if we want to maintain a custom order, PriorityQueue or BlockingPriorityQueue is a right collection to use. TreeSet also provides similar functionality but doesn't have one short retrieval cum removal method e.g. poll().