**Collections**

#### **Difference between hashmap and concurrent hashmap**

<http://javarevisited.blogspot.in/2011/04/difference-between-concurrenthashmap.html>

#### **Data structure used by arraylist**

ArrayList uses an Array of Object to store the data internally.

When you initialize an arraylist, an array of size 10 (default capacity) is created and any element added to the arrayList is actually added to this array. 10 is the default size and it can be passed as a parameter while initializing the arrayList.

When adding a new element, if the array is full, then a new array of double the initial size is created and the last array is copied to this new array, so that now there is empty spaces for the new element to be added.

Since, the underlying data-structure used is an array, it is fairly easy to add a new element to the arrayList as it is added to the end of the list. When an element is to be added anywhere else, say the beginning, then all the elements shall have to move one position to the right to create an empty space at the beginning for the new element to be added. This process is time-consuming (linear-time). But the Advantage of ArrayList is that retrieving an element at any position is very fast (constant-time), as underlying it is simply using an array of objects.

#### **What is Collection ? What is a Collections Framework ? What are the benefits of Java Collections Framework ?**

**Collection :** A collection (also called as container) is an object  that groups multiple elements into a single unit.

**Collections Framework :**Collections framework provides unified architecture for manipulating and representing collections.

**Benefits of Collections Framework :**

1. Improves program quality and speed

2. Increases the chances of reusability of software

3. Decreases programming effort.

#### **What is the root interface in collection hierarchy ?**

Root interface in collection hierarchy is **Collection interface .**Few interviewer may argue that

Collection interface extends **Iterable interface**. So iterable should be the root interface. But you should reply iterable interface present in java.lang package not in java.util package .It is clearly mentioned in [Oracle Collection  docs](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html) , that Collection interface is a member of the Java Collections framework.  For [Iterable interface Oracle doc](https://docs.oracle.com/javase/7/docs/api/java/lang/Iterable.html) , iterable interface is not mentioned as a part of the Java Collections framework .So if the question includes collection hierarchy, then you should answer the question as Collection interface (which is found in java.util package).

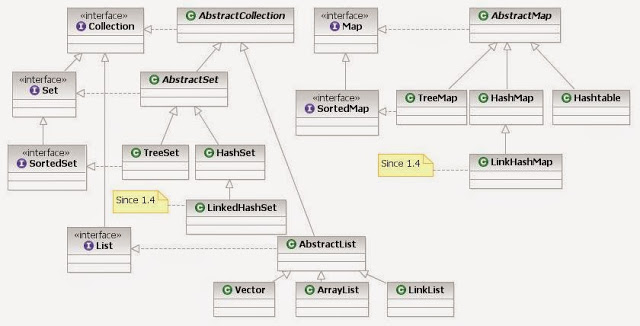
#### **What is the difference between Collection and Collections ?**

Collection is  an interface while Collections is a java class , both are present in java.util package and  part of java collections framework.

#### **Which collection classes are synchronized or thread-safe ?**

Stack, Properties , Vector and Hashtable can be used in multi threaded environment because they are synchronized classes (or thread-safe).

#### **Name the core Collection interfaces?**

[](http://1.bp.blogspot.com/-ifC30f-ZZ7M/VU7wWRwqo3I/AAAAAAAAAmQ/b1CIhOP5NU8/s1600/Java_collection_framework.jpg)

The list of core collection interfaces are: just mention the important ones

Important: Collection, Set, Queue, List, and Map

Other interface also in the list:  SortedSet, SortedMap, Deque, ListIterator etc.

#### **What is the difference between List and Set?**

Set contain only unique elements while List can contain duplicate elements.

Set is unordered while List is ordered. List maintains the order in which the objects are added.

#### **What is the difference between Map and Set?**

Map object has unique keys each containing some value, while Set contain only unique values.

#### **What are the classes implementing List and Set interface?**

***Class implementing List interface:***  ArrayList, Vector, LinkedList

***Class implementing Set interface:***HashSet, TreeSet

#### **What is an iterator?**

Iterator is an interface. It is found in java.util package. It provides methods to iterate over any Collection.

#### **What is the difference between Iterator and Enumeration?**

Between Enumeration and Iterator, Enumeration is older and it is there from JDK1.0, while iterator was introduced later. Iterator can be used with ArrayList, HashSet and other collection classes.  Another similarity between Iterator and Enumeration in Java is that functionality *of Enumeration interface is duplicated by the Iterator interface*.

Only major difference between Enumeration and iterator is Iterator has a remove() method while Enumeration does not. Enumeration acts as Read-only interface, because it has the methods only to traverse and fetch the objects, whereas by using Iterator we can manipulate the objects like adding and removing the objects from collection e.g. Arraylist.

In addition, **Iterator**is more secure and safe as compared to **Enumeration**because it does not allow other thread to modify the collection object while some thread is iterating over it and throws **ConcurrentModificationException.**This is by far most important fact for me for deciding between Iterator vs Enumeration in Java.

In Summary, both Enumeration and Iterator will give successive elements, but Iterator is new and improved version where method names are shorter, and has new method called remove. Here is a short comparison:

**Enumeration**

hasMoreElement()

nextElement()

N/A

**Iterator**

hasNext()

next()

remove()

So **Enumeration** is used whenever we want to make Collection objects as Read-only.

#### **Which design pattern followed by Iterator?**

It follows iterator design pattern. Iterator design pattern provides us to navigate through the collection of objects by using a common interface without letting us know about the underlying implementation.

Enumeration is an example of Iterator design pattern.

#### **Which methods you need to override to use any object as key in Hash Map?**

HashMap in Java works on hashing principle. It is a data structure that 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, hashcode() method of the key object is called so that 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 you want to retrieve the object, you call [the get() method](http://java67.blogspot.com/2013/06/how-get-method-of-hashmap-or-hashtable-works-internally.html) and again pass the key object. This time again key object generate same hash code (it is 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 your value object which you have stored earlier. Things get little [tricky](http://java67.blogspot.com/2012/09/top-10-tricky-java-interview-questions-answers.html) when collisions occur. It is easy to answer this question if you have read good books on data structure and algorithms like [this](http://www.amazon.com/Data-Structures-Algorithm-Analysis-Edition/dp/0132576279?tag=javamysqlanta-20)one. If you know how hash table data structure works then this is a piece of cake.

Since the internal array of HashMap is of fixed size, and if you keep storing objects, at some point of time hash function will return 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 next node.

If we try to retrieve an object from this linked list, we need an extra check to search 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 return true, Map returns the corresponding value.

Since searching inline list is O(n) operation, in worst-case hash collision reduce a map to linked list. This issue is recently addressed in Java 8 by replacing linked list to the tree to search in O(logN) time. By the way, you can easily verify how HashMap works by looking at the code of HashMap.java in your Eclipse IDE if you know [how to attach source code of JDK in Eclipse](http://javarevisited.blogspot.com/2012/12/how-to-attach-source-in-eclipse-Jar-JDK-debugging.html).

How HashMap works in Java or sometimes how does get method work in HashMap is a very common question on Java interviews nowadays. Almost everybody who worked in Java knows about HashMap, where to use HashMap and difference between Hashtable and HashMap then why this interview question becomes so special? Because of the depth it offers.

It has become very [popular Java interview question](http://javarevisited.blogspot.com/2015/10/133-java-interview-questions-answers-from-last-5-years.html) in almost any senior or mid-senior level Java interviews. Investment banks mostly prefer to ask this question and sometimes even ask you to implement your own HashMap based upon your coding aptitude. The introduction of [ConcurrentHashMap](http://javarevisited.blogspot.co.uk/2013/02/concurrenthashmap-in-java-example-tutorial-working.html) and other concurrent collections has also made these questions as starting point to delve into a more advanced feature. Let’s start the journey.

**How HashMap Internally Works in Java**

Questions start with simple statement:

**Have you used HashMap before**or**what is HashMap? Why do you use it?**

Almost everybody answers this with yes and then interviewee keep talking about common facts about HashMap like HashMap accept null while Hashtable does not, [HashMap is not synchronized](http://javarevisited.blogspot.com/2010/10/difference-between-hashmap-and.html), HashMap is fast and so on along with basics like its stores key and value pairs etc. This shows that person has usedHashMap and quite familiar with the functionality it offers, but interview takes a sharp turn from here and next set of follow-up questions gets more detailed about fundamentals involved with HashMap in Java. Interviewer strike back with questions like:

**Do you Know how HashMap works in Java** or **How does get () method of HashMap works in Java**

And then you get answers like,  I don't bother its standard Java API, you better look code on Java source or Open JDK; I can find it out in Google at any time etc. But some interviewee definitely answers this and will say **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](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html)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. This answer is very much acceptable and does make sense that interviewee has a fair bit of knowledge on how hashing works and how HashMap  works in Java. But this is just start of story and confusion increases when you put interviewee on scenarios faced by Java developers on day by day basis. Next question could be about collision detection and collision resolution in Java HashMap  e.g.

**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 you might want to remind them about [equals() and hashCode() contract](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html)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](http://javarevisited.blogspot.sg/2012/02/difference-between-linkedlist-vs.html). 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

**How will you 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 you need to remind him that there are two Value objects are stored in same bucket , so they will say about [traversal in LinkedList](http://javarevisited.blogspot.sg/2010/10/how-do-you-find-length-of-singly-linked.html)until we find the value object , then you ask *how do you identify value object because you 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. Perfect this is the correct answer.

In many cases interviewee fails at this stage because they get confused between[hashCode()](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html) 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](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) 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](http://javarevisited.blogspot.sg/2011/07/string-vs-stringbuffer-vs-stringbuilder.html)and various wrapper classes e.g. Integer very good keys in Java HashMap.

Machine generated alternative text:


Now if you clear this entire Java HashMap interview,  You will be surprised by this very interesting question "**What happens On HashMap in Java if the size of the HashMap  exceeds a given threshold defined by load factor ?"**. Until you know how HashMap  works exactly you 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](http://javarevisited.blogspot.sg/2011/05/example-of-arraylist-in-java-tutorial.html),  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.

If you manage to answer this question on HashMap in Java you will be greeted by **"do you see any problem with resizing of HashMap  in Java"**, you might not be able to pick the context and then he will try to give you 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](http://javarevisited.blogspot.sg/2012/02/what-is-race-condition-in.html) exists while resizing HashMap in Java, if two [thread](http://javarevisited.blogspot.sg/2011/02/how-to-implement-thread-in-java.html)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 you will end up with an infinite loop. Though this point, you can potentially argue that what the hell makes you think to use HashMap  in multi-threaded environment to interviewer :)

**Some more Hashtable and HashMap Questions**

Few more question on HashMap in Java which is contributed by readers of Javarevisited blog:

**1) 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](http://javarevisited.blogspot.sg/2010/10/why-string-is-immutable-in-java.html), 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](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html), If you can  keep your hashCode same by only making certain fields final, then you 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.

**2) Can we use any custom object as a key in HashMap?**

This is an extension of previous questions. Of course you 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](http://javarevisited.blogspot.sg/2011/12/how-to-traverse-or-loop-hashmap-in-java.html). If the custom object is Immutable than this will be already taken care because you can not change it once created.

**3) Can we use ConcurrentHashMap in place of Hashtable?**

This is another question which getting popular due to increasing popularity of ConcurrentHashMap. 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 asHashtable and can be used in place of it, but Hashtable provides stronger thread-safety than ConcurrentHashMap. See my post [difference between Hashtable and ConcurrentHashMap](http://javarevisited.blogspot.sg/2011/04/difference-between-concurrenthashmap.html) for more details.

Personally, I like this question because of its depth and number of concept it touches indirectly if you look at questions asked during interview this HashMap  questions has verified

* The concept of hashing
* Collision resolution in HashMap
* Use of equals () and hashCode () and their importance in HashMap?
* The benefit of the immutable object?
* Race condition on HashMap  in Java
* Resizing of Java HashMap

**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?**

The null key is handled specially in HashMap, there are two separate methods for that putForNullKey(V value) and getForNullKey(). Later is 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**;  
 }

In terms of usage, Java HashMap is very versatile and I have mostly used HashMap as cache in an electronic trading application I have worked. Since finance domain used Java heavily and due to performance reason we need caching HashMap and ConcurrentHashMap  comes as very handy there. You can also check following articles from Javarevisited to learn more about HashMap and Hashtable in Java:

#### **What is the difference between Queue and Stack ?**

Queue is a data structure which is based on FIFO ( first in first out ) property . An example of Queue in real world is buying movie tickets in the multiplex or cinema theaters.

Stack is a data structure which is based on LIFO (last in first out) property . An example of Stack in real world is  insertion or removal of CD  from the CD case.

#### **How to reverse the List in Collections ?**

There is a built in reverse method in Collections class . reverse(List list) accepts list as parameter.

**Collections.reverse(listobject);**

#### **How to convert the array of strings into the list ?**

Arrays class of java.util package contains the method asList() which accepts the array as parameter.

So,

**String[]  wordArray =  {"Love Yourself"  , "Alive is Awesome" , "Be in present"};**

**List wordList =  Arrays.asList(wordArray);**

#### **What is the difference between ArrayList and Vector ?**

1. Synchronization and Thread-Safe

Vector is synchronized while ArrayList is not synchronized . Synchronization and thread safe means at a time only one thread can access the code .In Vector class all the methods are synchronized .Thats why the Vector object is already synchronized when it is created .

2. Performance

Vector is slow as it is thread safe . In comparison ArrayList is fast as it is non synchronized . Thus in ArrayList two or more threads can access the code at the same time , while Vector is limited to one thread at a time.

3. Automatic Increase in Capacity

A Vector defaults to doubling size of its array . While when you insert an element into the ArrayList , it increases

its Array size by 50% .By default ArrayList size is 10 . It checks whether it reaches the last element then it will create the new array ,copy the new data of last array to new array ,then old array is garbage collected by the Java Virtual Machine (JVM) .

4. Set Increment Size

ArrayList does not define the increment size . Vector defines the increment size .

You can find the following method in Vector Class

public synchronized void setSize(int i) { //some code }up

There is no setSize() method or any other method in ArrayList which can manually set the increment size.

5. Enumerator

Other than Hashtable ,Vector is the only other class which uses both Enumeration and Iterator .While ArrayList can only use Iterator for traversing an ArrayList .

6. Introduction in Java

java.util.Vector class was there in java since the very first version of the java development kit (jdk).

java.util.ArrayList was introduced in java version 1.2 , as part of Java Collections framework . In java version 1.2 , Vector class has been refactored to implement the List Inteface .

#### **What is the difference between HashMap and Hashtable ?**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Hashtable** | **HashMap** | **ConcurrentHashMap** |
| **Legacy Class** | Yes, introduced in 1.0 | No, Introduced in 1.2 | No, Introduced in 1.5 with java.util.concurrent package |
| **Thread-safe** | Yes, use synchronized keyword in all methods to provide thread safety. | No, only suitable for single threaded enviornment | Yes, use java.concurrent.ReenterantLock to avoid concurrent access |
| **Performance** | slow, any operation blocks whole Map. So a new operation has to wait for finish of previous one. | Fast, as no locks are used | Faster than Hashtable, as locks are applied on different segments and any two operations across different segments are parallel. |
| **Iterators** | Fail Fast. | Fail Fast | Fail Safe |
| **Null key/values** | No | Yes | No |

**ConcurrentHashMap vs Hashtable vs Synchronized Map**

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 you can wrap any Map to create it's synchronized version by usingCollections.synchronizedMap() method.

On the other hand, 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](http://javarevisited.blogspot.sg/2013/02/concurrenthashmap-in-java-example-tutorial-working.html) 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.

In this tutorial, we will look on ConcurrentHashMap, Hashtable, HashMap and synchronized Map and see the difference between ConcurrentHashMap and Hashtable and synchronized Map in Java. We have already discussed some key [difference between HashMap and Hashtable in Java](http://javarevisited.blogspot.com/2010/10/difference-between-hashmap-and.html) in this blog and those will also help you to answer this question during interviews.

**Why need ConcurrentHashMap and CopyOnWriteArrayList**

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 prevent [ConcurrentModificationException](http://javarevisited.blogspot.sg/2011/10/java-iterator-tutorial-example-list.html).

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 you might use HashMap or ArrayList, but are designed to optimize specific common situations. Many concurrent applications will benefit from their use.

**Difference between ConcurrentHashMap and Hashtable**

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.

**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. Here are some of the common differences between ConcurrentHashMap and synchronized map in Java

ConcurrentHashMap does not allow null keys or null values while synchronized HashMap allows one null key.

#### **What is CopyOnWriteArrayList in Java ?**

**CopyOnWriteArrayList vs Array List in Java**

CopyOnWriteArrayList is a concurrent Collection class introduced in Java 5 Concurrency API along with its popular cousin [ConcurrentHashMap](http://java67.blogspot.sg/2012/08/difference-between-hashmap-and-concurrentHashMap-java-collection.html) in Java. CopyOnWriteArrayList implements List interface like [ArrayList](http://javarevisited.blogspot.sg/2011/05/example-of-arraylist-in-java-tutorial.html), [Vector](http://javarevisited.blogspot.sg/2011/09/difference-vector-vs-arraylist-in-java.html) and [LinkedList](http://javarevisited.blogspot.sg/2012/02/difference-between-linkedlist-vs.html) but its a thread-safe collection and it achieves its [thread-safety](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html) in a slightly different way than Vector or other thread-safe collection class. As name suggest CopyOnWriteArrayListcreates copy of underlying [ArrayList](http://javarevisited.blogspot.sg/2011/06/converting-array-to-arraylist-in-java.html) with every mutation operation e.g. add or set. Normally CopyOnWriteArrayList is very expensive because it involves **costly Array copy** with every write operation but its very efficient if you have a [List](http://javarevisited.blogspot.sg/2012/04/difference-between-list-and-set-in-java.html) where Iteration outnumber mutation e.g. you mostly need to [iterate the ArrayList](http://java67.blogspot.sg/2012/08/how-to-traverse-iterate-or-loop-ArrayList-in-java-example-tutorial.html) and don't modify it too often. Iterator of CopyOnWriteArrayList is [fail-safe](http://javarevisited.blogspot.sg/2012/02/fail-safe-vs-fail-fast-iterator-in-java.html) and doesn't throwConcurrentModificationException even if underlying CopyOnWriteArrayList is modified once Iteration begins because Iterator is operating on separate copy of ArrayList. Consequently all the updates made on CopyOnWriteArrayList is not available to Iterator. In this Java Collection tutorial we will see *What is CopyOnWriteArrayList in Java*, *Difference between ArrayList and CopyOnWriteArrayList in Java* and One simple Java program example on How to use CopyOnWriteArrayList in Java.

Difference between CopyOnWriteArrayList and ArrayList in Java.

In last section we have seen What is CopyOnWriteArrayList in Java and How it achieves [thread-safety](http://javarevisited.blogspot.sg/2012/03/simpledateformat-in-java-is-not-thread.html) by creating a separate [copy of List](http://java67.blogspot.sg/2012/07/copy-elements-from-list-to-set-in-java-collection-example.html) for each write operation. Now let's see Some **difference between ArrayList and CopyOnWriteArrayList in Java** , which is another implementation of List interface :

1) First and foremost difference between CopyOnWriteArrayList and ArrayList in Java is that CopyOnWriteArrayList is a [thread-safe collection](http://javarevisited.blogspot.sg/2011/04/difference-between-concurrenthashmap.html) while ArrayList is not thread-safe and can not be used in multi-threaded environment.

2) Second difference between ArrayList and CopyOnWriteArrayList is that [Iterator of ArrayList is fail-fast](http://javarevisited.blogspot.sg/2012/02/fail-safe-vs-fail-fast-iterator-in-java.html) and throw ConcurrentModificationException once detect any modification in List once iteration begins but Iterator of CopyOnWriteArrayList is fail-safe and doesn't throw ConcurrentModificationException.

3) Third difference between CopyOnWriteArrayList vs ArrayList is that [Iterator](http://javarevisited.blogspot.sg/2011/10/java-iterator-tutorial-example-list.html) of former doesn't support remove operation while Iterator of later supports remove() operation.

**CopyOnWriteArrayList Example in Java**

Here is a complete code Example of CopyOnWriteArrayList which demonstrate that [Iterator of CopyOnWriteArrayList](http://javarevisited.blogspot.sg/2011/10/java-iterator-tutorial-example-list.html) doesn't support remove() operation.

**import** java.util.Iterator;

**import** java.util.concurrent.CopyOnWriteArrayList;

/\*\*

 \*

 \* Java program to demonstrate What is CopyOnWriteArrayList in Java,

 \* Iterator of CopyOnWriteArrayList

 \* doesn’t support add, remove or any modification operation.

 \*

 \* @author Java67

 \*/

**public** **class** CopyOnWriteArrayListExample{

**public** **static** **void** main(**String** args[]) {

**CopyOnWriteArrayList**<**String**> threadSafeList = **new** **CopyOnWriteArrayList**<**String**>();

        threadSafeList.add("Java");

        threadSafeList.add("J2EE");

        threadSafeList.add("Collection");

*//add, remove operator is not supported by CopyOnWriteArrayList iterator*

**Iterator**<**String**> failSafeIterator = threadSafeList.iterator();

        while(failSafeIterator.hasNext()){

**System**.out.printf("Read from CopyOnWriteArrayList : %s %n", failSafeIterator.next());

            failSafeIterator.remove(); *//not supported in CopyOnWriteArrayList in Java*

        }

    }

}

Output:

Read from **CopyOnWriteArrayList** : Java

Read from **CopyOnWriteArrayList** : J2EE

Read from **CopyOnWriteArrayList** : **Collection**

If we uncomment, commented code in this [Java program](http://javarevisited.blogspot.sg/2011/11/run-java-program-from-command-prompt.html) which modifies CopyOnWriteArrayList using Iterator then we will get following [Exception](http://javarevisited.blogspot.sg/2011/12/checked-vs-unchecked-exception-in-java.html):

Read from CopyOnWriteArrayList : Java

Exception in thread "main" java.lang.UnsupportedOperationException

        at java.util.concurrent.CopyOnWriteArrayList$COWIterator.remove(CopyOnWriteArrayList.java:1004)

        at test.CollectionTest.main(CollectionTest.java:29)

Java Result: 1

That's all on What is CopyOnWriteArrayList, **Difference between CopyOnWriteArrayList and ArrayList in Java** and an Exmaple of CopyOnWriteArrayList. In Summary use CopyOnWriteArrayList if you mostly require to Iterate over list without modifying it.

#### **What is the difference between Iterator and ListIterator.**

|  |  |
| --- | --- |
| ITERATOR | LISTITERATOR |
| Can do remove operation only on elements | Can remove, add and replace elements |
| Method is remove() | Methods are remove(), add() and set() |
| iterator() method returns an object of Itertor | listIterator() method returns an object of ListItertor |
| iterator() method is available for all collections. That is, Iterator can be used for all collection classes | listIterator() method is available for those collections that implement List interface. That is, descendants of List interface only can use ListIterator |

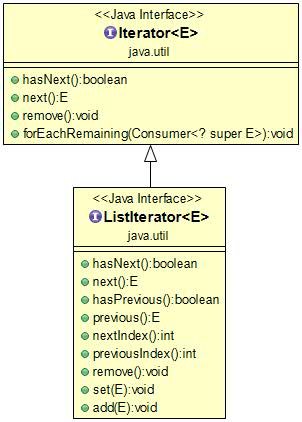
**Java ListIterator**

Like Iterator, ListIterator is a Java Iterator, which is used to iterate elements one-by-one from a List implemented object.

* It is available since Java 1.2.
* It extends Iterator interface.
* It is useful only for List implemented classes.
* Unlike Iterator, It supports all four operations: CRUD (CREATE, READ, UPDATE and DELETE).
* Unlike Iterator, It supports both Forward Direction and Backward Direction iterations.
* It is a Bi-directional Iterator.
* It has no current element; its cursor position always lies between the element that would be returned by a call to previous() and the element that would be returned by a call to next().

**Java ListIterator Class Diagram**

In Java, ListIterator is an interface in Collection API. It extends Iterator interface. To support Forward and Backward Direction iteration and CRUD operations, it has the following methods. We can use this Iterator for all List implemented classes like ArrayList, CopyOnWriteArrayList, LinkedList, Stack, Vector etc.



**Java ListIterator Methods**

Java ListIterator interface has the following Methods.

* void add(E e): Inserts the specified element into the list.
* boolean hasNext(): Returns true if this list iterator has more elements when traversing the list in the forward direction.
* boolean hasPrevious(): Returns true if this list iterator has more elements when traversing the list in the reverse direction.
* E next(): Returns the next element in the list and advances the cursor position.
* int nextIndex(): Returns the index of the element that would be returned by a subsequent call to next().
* E previous(): Returns the previous element in the list and moves the cursor position backwards.
* int previousIndex(): Returns the index of the element that would be returned by a subsequent call to previous().
* void remove(): Removes from the list the last element that was returned by next() or previous().
* void set(E e): Replaces the last element returned by next() or previous() with the specified element.

#### **What is the difference between Array and ArrayList in Java?**

**1. Resizable :**   Array is static in size that is fixed length data structure, One can not change the length after creating the Array object.ArrayList is dynamic in size . Each ArrayList object  has instance variable *capacity*which indicates the size of the ArrayList. As elements are added to an ArrayList its capacity grows automatically.

**2. Performance :** Performance of Array and ArrayList depends on the operation you are performing :

*resize() opertation :* Automatic resize of ArrayList will slow down the performance as it will use temporary array to copy elements from the old array to new array.

ArrayList is internally backed by Array during resizing  as it calls the native implemented method System.arrayCopy(src,srcPos,dest,destPos,length) .

*add() or get() operation :* adding an element or retrieving an element from the array or arraylist object has almost same  performance , as for ArrayList object these operations  run in constant time.

**3. Primitives :**  ArrayList can not contains primitive data types (like int , float , double) it can only contains Object while Array can contain both primitive data types as well as objects.

One get a misconception that we can store primitives(int,float,double) in ArrayList , but it is not true

Suppose we have ArrayList object ,

ArrayList  arraylistobject = new ArrayList();

arraylistobject.add(**23**);  // try to add 23 (primitive)

JVM through Autoboxing(converting primitives to equivalent objects internally) ensures that only objects are added to the arraylist object.

thus , above step internally works like this :

arraylistobject.add( **new Integer(23)**);

// Converted int primitive to Integer object and added to arraylistobject

**4. Iterating the values :** We can use iterator  to iterate through ArrayList . The iterators returned by the ArrayList class's iterator and listiterator method are [fail-fast](http://javahungry.blogspot.ca/2014/04/fail-fast-iterator-vs-fail-safe-iterator-difference-with-example-in-java.html).  We can use for loop or for each loop to iterate through array .

**5. Type-Safety :**In Java , one can ensure Type Safety through Generics. while Array is a homogeneous data structure , thus it will contain objects of specific class or primitives of specific  data type. In array if one try to store the different data type other than the specified while creating the array object , ArrayStoreException is thrown.

for example :

String temp[] =  new String[2];  // creates a string array of size 2

temp[0] = new Integer(12); // throws ArrayStoreException, trying to add Integer object in String[]

**6. Length :**Length of the ArrayList is provided by the size() method while Each array object has the length variable which returns the length of the array.

for example :

Integer arrayobject[] = new Integer[3];

arraylength= arrayobject.length   ;  //uses arrayobject length variable

ArrayList  arraylistobject = new ArrayList();

arraylistobject.add(12);

arraylistobject.size();   //uses arraylistobject size method

**7. Adding elements :** We can insert elements into the arraylist object using the add() method while  in array we insert elements using the assignment operator.

for example :

Integer addarrayobject[] = new Integer[3];

addarrayobject[0]= new Integer(8)   ;  //new object is added to the array object

**8. Multi-dimensional :**Array can be multi dimensional , while ArrayList is always single dimensional.

example of multidimensional array:

Integer addarrayobject[][] = new Integer[3][2];

addarrayobject[0][0]= new Integer(8)

**Example of Array and ArrayList**

**import java.util.ArrayList**;  
**import java.util.Iterator**;

**public class ArrayArrayListExample** {  
   
 **public static void main**(String[] args) {

// ArrayList Example

 ArrayList<String> arrlistobj = **new** ArrayList<String>();  
 arrlistobj.add("Alive is awesome");  
 arrlistobj.add("Love yourself");

  Iterator it = arrlistobj.iterator();

System.out.print("*ArrayList object output :*");

  while(it.hasNext())

  System.out.print(it.next() + " ");

  // Array Example

String[] arrayobj = new String[3];  
 arrayobj[0]= "Love yourself";   
 arrayobj[1]= "Alive is awesome";

arrayobj[2]= "Be in Present";

System.out.print("*Array object output :*");

  for(int i=0; i < arrayobj.length ;i++)

  System.out.print(arrayobj[i] + " ");

 }  
}

**Output :***ArrayList object output :****{ Alive is awesome , Love yourself }***

*Array object output :****{ Love yourself , Alive is awesome, Be in present}***

**Similarities Between Array and ArrayList**

**1. add and get method :**Performance of Array and ArrayList are similar for the add and get operations .Both operations runs in constant time.

**2. Duplicate elements :**Both array and arraylist can contain duplicate elements.

**3. Null Values :**Both can store null values and uses index to refer to their elements.

**4. Unordered :**  Both does not guarantee ordered  elements.

|  |  |  |
| --- | --- | --- |
|  | **Array** | **ArrayList** |
| Resizable | No | Yes |
| Primitives | Yes | No |
| Iterating values | for, for each | Iterator , for each |
| Length | length variable | size method |
| Performance | Fast | Slow in comparision |
| Multidimensional | Yes | No |
| Add Elements | Assignment operator | add method |

#### **What is the difference between HashSet and TreeSet ?**

Main differences between HashSet and TreeSet are :

a.  HashSet maintains the inserted elements in random order while TreeSet maintains elements in the sorted order

b. HashSet can store null object while TreeSet can not store null object.

find detailed explanation here [TreeSet vs HashSet in Java](http://javahungry.blogspot.co.uk/2014/03/difference-between-hashset-and-treeset-similarities-and-example.html)

#### **Write java code showing insertion,deletion and retrieval of HashMap object ?**

import java.util.Map;

import java.util.HashMap;

import java.util.Iterator;

import java.util.Set;

public class HashMapDemo{

public static void main(String[] args){

Map hashmap=new HashMap();

hashmap.put(1001,"Jerry");

hashmap.put(1005,"12");

hashmap.put(1003,"Mary");

hashmap.put(1002,"jack");

hashmap.put(1004,"Jerry");

Set set=hashmap.keySet();

Iterator iterator = set.iterator();

while (iterator.hasNext()){

Integer key=(Integer)iterator.next();

System.out.print(key + " = ");

System.out.println((String)hashmap.get(key) + " and : ");

}

}

}

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

This is also one of the most popular java collections interview question . Make sure this question is in your to do list before appearing for the interview .

Main differences between HashMap and ConcurrentHashMap are :

a. HashMap is not synchronized while ConcurrentHashMap is synchronized.

b. HashMap can have one null key and any number of null values while ConcurrentHashMap does not allow null keys and null values .

find detailed explanation here [ConcurrentHashMap vs HashMap in Java](http://javahungry.blogspot.co.uk/2014/02/hashmap-vs-concurrenthashmap-java-collections-interview-question.html)

#### **Arrange the following in the ascending order (performance):**

**HashMap , Hashtable , ConcurrentHashMap and Collections.SynchronizedMap**

Hashtable  <  Collections.SynchronizedMap  <  ConcurrentHashMap  <  HashMap

#### **How HashMap works in Java ?**

This is one of the most important question for java developers. HashMap  works on the principle of Hashing . Find detailed information here to understand [what is hashing and how hashmap works in java](http://javahungry.blogspot.co.uk/2013/08/hashing-how-hash-map-works-in-java-or.html) .

#### **What is the difference between LinkedList and ArrayList in Java ?**

Different between arraylist and linked list

|  |  |
| --- | --- |
| ArrayList | LinkedList |
| 1) ArrayList internally uses dynamic array to store the elements. | LinkedList internally uses doubly linked list to store the elements. |
| 2) Manipulation with ArrayList is slow because it internally uses array. If any element is removed from the array, all the bits are shifted in memory. | Manipulation with LinkedList is faster than ArrayList because it uses doubly linked list so no bit shifting is required in memory. |
| 3) ArrayList class can act as a list only because it implements List only. | LinkedList class can act as a list and queue both because it implements List and Deque interfaces. |
| 4) ArrayList is better for storing and accessing data. | LinkedList is better for manipulating data. |

#### **What are Comparable and Comparator interfaces ? List the difference between them ?**

Comparator and Comparable

|  |  |  |
| --- | --- | --- |
| Parameter | Comparable | Comparator |
| Sorting logic | Sorting logic must be in same class whose objects are being sorted. Hence this is called natural ordering of objects | Sorting logic is in separate class. Hence we can write different sorting based on different attributes of objects to be sorted. E.g. Sorting using id,name etc. |
| Implementation | Class whose objects to be sorted must implement this interface.e.g Country class needs to implement comparable to collection of country object by id | Class whose objects to be sorted do not need to implement this interface.Some other class can implement this interface. E.g.-CountrySortByIdComparator class can implement Comparator interface to sort collection of country object by id |
| Sorting method | int compareTo(Object o1)  This method compares this object with o1 object and returns a integer.Its value has following meaning  1. positive – this object is greater than o1  2. zero – this object equals to o1  3. negative – this object is less than o1 | int compare(Object o1,Object o2)  This method compares o1 and o2 objects. and returns a integer.Its value has following meaning.  1. positive – o1 is greater than o2  2. zero – o1 equals to o2  3. negative – o1 is less than o1 |
| Calling method | Collections.sort(List)  Here objects will be sorted on the basis of CompareTo method | Collections.sort(List, Comparator)  Here objects will be sorted on the basis of Compare method in Comparator |
| Package | Java.lang.Comparable | Java.util.Comparator |

#### **Why Map interface does not extend the Collection interface in Java Collections Framework ?**

One liner answer : **Map interface is not compatible with the Collection interface.**

Explanation : Since Map requires key as well as value , for example , if we want to add key-value pair then we will use put(Object key , Object value) . So there are two parameters required to add element to the HashMap object  . In Collection interface add(Object o) has only one parameter.

The other reasons are Map supports valueSet , keySet as well as other appropriate methods which have just different views from the Collection interface.

#### **When to use ArrayList and when to use LinkedList in application?**

ArrayList has constant time search operation O(1) .Hence, ArrayList is preferred when there are more get() or search operation .

Insertion , Deletion operations take constant time O(1) for LinkedList. Hence, LinkedList is preferred when there are more insertions or deletions involved in the application.

#### **Write the code for iterating the list in different ways in java ?**

There are two ways to iterate over the list in java :

a. using Iterator

b. using for-each loop

Coding part : Do it  yourself (DIY) , in case of any doubts or difficulty please mention in the comments .

***Advance Level (3+ yrs): Java Collections Interview Questions  and Answers***

#### **How HashSet works internally in java ?ds**

**Set Implementation Internally in Java**

Each and every element in the set is unique .  So that there is no duplicate element in set . So in java if we want to add elements in the set then we write code like this

**public class JavaHungry** {   
 **public static void main**(String[] args)  
 {  
 // TODO Auto-generated method stub  
   
 HashSet<Object> hashset = **new** HashSet<Object>();  
 hashset.add(**3**);  
 hashset.add("Java Hungry");  
 hashset.add("Blogspot");  
 System.out.println("Set is "+hashset);  
 }  
}

*It will print the result* :       Set is [3, Java Hungry, Blogspot]

Now let add duplicate element in the above code

**public class JavaHungry** {  
 **public static void main**(String[] args)  
 {  
 HashSet<Object> hashset = **new** HashSet<Object>();  
 hashset.add(**3**);  
 hashset.add("Java Hungry");  
 hashset.add("Blogspot");  
 hashset.add(**3**); // duplicate elements  
 hashset.add("Java Hungry"); // duplicate elements  
 System.out.println("Set is "+hashset);  
 }  
}

*It will print the result* :       Set is [3, Java Hungry, Blogspot]

Now , what happens internally when you pass duplicate elements in the  add() method of the Set object , It will return false and do not add to the HashSet , as the element is already present .So far so good .

But the main problem arises that how it returns false . So here is the answer

When you open the HashSet implementation of the add() method in Java Apis that is rt.jar , you will find the following code in it

**public class HashSet**<E>  
**extends** AbstractSet<E>  
**implements** Set<E>, Cloneable, java.io.Serializable{  
 **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 HashSet**() {  
 *map =* ***new*** *HashMap<>();*  
 }   
 // SOME CODE ,i.e Other methods in Hash Set   
   
 **public boolean add**(E e) {  
 **return** map.put(e, PRESENT)==**null**;  
 }   
 // SOME CODE ,i.e Other methods in Hash Set  
}

So , we are achieving uniqueness in Set,internally in java  through HashMap . Whenever you create an object of HashSet it will create an object of HashMap as you can see in the italic lines in the above code .

We already discussed   [How HashMap works internally  in java](http://javahungry.blogspot.com/2013/08/hashing-how-hash-map-works-in-java-or.html) .

As we know in HashMap each key is unique . So what we do in the set is that we pass the argument in the add(Elemene E) that is E as a key in the HashMap . Now we need to associate some value to the key , so what Java apis developer did is to pass the Dummy  value that is ( new Object () ) which is referred by Object reference PRESENT .

So , actually when you are adding a line in HashSet like  hashset.add(3)   what java does internally is that it will put that element E here 3 as a key in the HashMap(created during HashSet object creation) and some dummy value that is Object's object is passed as a value to the key .

Now if you see the code of the HashMap put(Key k,Value V) method , you will find something like this

 public V put(K key, V value) {

//Some code

}

The main point to notice in above code is that put (key,value) will return

1.  null , if key is unique and added to the map

2.  Old Value of the key , if key is duplicate

So , in HashSet add() method ,  we check the return value of map.put(key,value) method with null value

i.e.

   public boolean add(E e) {

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

       }

So , if map.put(key,value) returns null ,then

map.put(e, PRESENT)==null      will return true and element is added to the HashSet.

So , if map.put(key,value) returns old value of the key ,then

map.put(e, PRESENT)==null      will return false and element is  not added to the HashSet .

#### **What is CopyOnWriteArrayList ?  How it is different from  ArrayList in Java?**

[CopyOnWriteArrayList](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArrayList.html) is a thread safe variant of ArrayList   in which all mutative operations like add , set are implemented by creating a fresh copy of the underlying array.

It guaranteed not to throw ConcurrentModificationException.

It permits all elements including null. It is introduced in jdk 1.5 .

#### **How HashMap works in Java ?**

We are repeating this question , as it is one of the most important question for java developer.HashMap works on the principle of Hashing . Please find the detailed answer here [hashmap internal working in java](http://javahungry.blogspot.co.uk/2013/08/hashing-how-hash-map-works-in-java-or.html) .

#### **How remove(key) method works in HashMap ?**

This is the new question which is getting popular among java interviewers . We have shared the detailed explanation here [how remove method works internally in java](http://javahungry.blogspot.co.uk/2015/03/how-remove-method-internally-works-in-hashmap-java.html).

#### **What is BlockingQueue in Java Collections Framework?**

[BlockingQueue](https://docs.oracle.com/javase/6/docs/api/java/util/concurrent/BlockingQueue.html) implements the java.util.Queue interface . BlockingQueue supports  operations that wait for the queue to become non-empty when retrieving an element , and wait  for space to become available in the queue when storing an element .

It does not accept null elements.

Blocking queues are primarily designed for the producer-consumer problems.

BlockingQueue implementations are thread-safe and can also be used in inter-thread communications.

This concurrent Collection class was added in jdk 1.5

**All the questions related to HashSet class can be found here** ,

[frequently asked HashSet interview questions](http://javahungry.blogspot.co.uk/2014/04/top-10-hashset-java-interview-questions-collection.html)

#### **What is the difference between Fail- fast iterator and Fail-safe iterator ?**

This is one  of the most popular interview question for the higher experienced java developers .

Main differences between Fail-fast and Fail-safe iterators are :

a. Fail-fast throw ConcurrentModificationException while Fail-safe does not.

b. Fail-fast does not clone the original collection list of objects while Fail-safe creates a copy of the original collection list of objects.

The difference is explained in detail here [fail-safe vs fail-fast iterator in java](http://javahungry.blogspot.co.uk/2014/04/fail-fast-iterator-vs-fail-safe-iterator-difference-with-example-in-java.html).

#### **How ConcurrentHashMap works internally in Java?**

The detailed answer is already explained here  [internal implementation of concurrenthashmap](http://javahungry.blogspot.co.uk/2015/02/how-concurrenthashmap-works-in-java-internal-implementation.html)

#### **How do you use a custom object as key in Collection  classes like HashMap ?**

If one is using the custom object as key then one needs to override equals() and hashCode() method

and one also need to fulfill the contract.

If you want to store the custom object in the SortedCollections like SortedMap then one needs to make sure that equals() method is consistent to the compareTo() method. If inconsistent , then collection will not follow their contracts ,that is , Sets may allow duplicate elements.

#### **What is hash-collision in Hashtable ? How it was handled in Java?**

In Hashtable , if two different keys have the same hash value then it lead to hash -collision. A bucket of type linkedlist used to hold the different keys of same hash value.

#### **Explain the importance of hashCode() and equals() method ? Explain the contract also ?**

HashMap object uses Key object hashCode() method and equals() method to find out the index to put the key-value pair. If we want to get value from the HashMap same both methods are used . Somehow, if both methods are not implemented correctly , it will result in two keys producing the same hashCode() and equals() output. The problem will arise that HashMap will treat both output same instead of different and overwrite the most recent key-value pair with the previous key-value pair.

Similarly all the collection classes that does not allow the duplicate values use hashCode() and equals() method to find the duplicate elements.So it is very important to implement them correctly.

**Contract of hashCode() and equals() method**

a.If  object1.equals(object2) , then  object1.hashCode() == object2.hashCode() should always be true.

b. If object1.hashCode() == object2.hashCode() is true does not guarantee object1.equals(object2)

#### **What is  WeakHashMap ?**

**WeakHashMap :**

[WeakHashMap](http://docs.oracle.com/javase/7/docs/api/java/util/WeakHashMap.html) is a class present in java.util package similar to IdentityHashMap. It is a Hashtable based implementation of Map interface with weak keys. An entry in WeakHashMap will automatically be removed when its key is no longer in ordinary use. More precisely the presence of a mapping for a given key will not prevent the key from being discarded by the garbage collector.

It permits null keys and null values.

Like most collection classes this class is not synchronized.A synchronized WeakHashMap may be constructed using the Collections.synchronizedMap() method.

Iterators returned by the iterator() method are fail-fast , hence , will throw ConcurrentModificationException.

#### **How will you make Collections readOnly ?**

We can make the Collection readOnly by using the following lines code:

General : Collections.unmodifiableCollection(Collection c)

Collections.unmodifiableMap(Map m)

Collections.unmodifiableList(List l)

Collections.unmodifiableSet(Set s)

#### **What is UnsupportedOperationException?**

This exception is thrown to indicate that the requested operation is not supported.

Example of UnsupportedOperationException:

In other words, if you call add() or remove() method on the readOnly collection . We know readOnly collection can not be modified . Hence , UnsupportedOperationException will be thrown.

#### **Suppose there is an Employee class. We add Employee class objects to the ArrayList. Mention the steps need to be taken , if I want to sort the objects in ArrayList using the employeeId attribute present  in Employee class.**

a. Implement the Comparable interface for the Employee class and now to compare the objects by employeeId we will override the emp1.compareTo(emp2)

b. We will now call Collections class sort method and pass the list as argument , that is ,

     Collections.sort(empList)

If you want to add more java collections interview questions  and answers or in case you have any doubts related to the Java Collections framework , then please mention in the comments.

While passing a Collection as argument to a function, how can we make sure the function will not be able to modify it?

We can create a read-only collection using Collections.unmodifiableCollection(Collection c) method before passing it as argument, this will make sure that any operation to change the collection will throw UnsupportedOperationException.

#### **How to override hashcode in Java example - Tutorial**

Equals and hashcode methods are two primary but yet one of most important methods for java developers to be aware of. Java intends to provide equals and hashcode for every class to test the equality and to provide a hash or digest based on content of class. Importance of hashcode increases when we use the object in different collection classes which works on hashing principle e.g. [hashtable and hashmap](http://javarevisited.blogspot.com/2010/10/difference-between-hashmap-and.html). A well written hashcode method can improve performance drastically by distributing objects uniformly and avoiding collision. In this article we will see**how to correctly override hashcode() method in java with a simple example**. We will also examine important aspect of hashcode contracts in java. This is in continuation of my earlier post on [overriding equals method in Java](http://javarevisited.blogspot.com/2011/02/how-to-write-equals-method-in-java.html), if you haven’t read already I would suggest go through it.

**General Contracts for hashCode() in Java**

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

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

3) If two objects are not equals 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 you don’t override them properly I would recommend understanding of [How HashMap works in Java](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html)

**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**

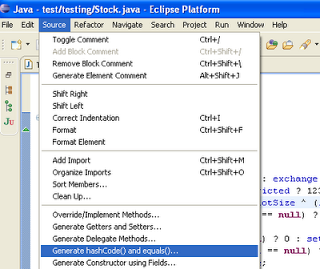
In my opinion better way to **override both equals and hashcode method** should be left to IDE. I have seen Netbeans and Eclipse and found that both has excellent support of generating *code for equals and hashcode* and there implementations seems to follow all best practice and requirement e.g. null check , instanceof check etc and also frees you to remember how to compute hashcode for different data-types.

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

**In Netbeans**

1) Write your 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 your Class.

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

**Things to remember while overriding hashcode in Java**

1. Whenever you override equals method, hashcode should be overridden to be in compliant of equals hashcode contract.

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

3. For immutable object you can cache the hashcode once generated for improved performance.

4. Test your hashcode method for equals hashcode compliance.

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

**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 much better way to override equals and hashcode method, at least much better than ugly equals, hashcode generated by Eclipse. I have written same example by using HashCodebuilder and EqualsBuilder and now you 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**()));

    }

Onlything to concern is that it adds dependency on apache commons jar, most people use it but if you are not using than you need to include it for writing equals and hashcode method.

Read more: <http://javarevisited.blogspot.com/2011/10/override-hashcode-in-java-example.html#ixzz44DkobOLl>

From <<http://javarevisited.blogspot.in/2011/10/override-hashcode-in-java-example.html>>