**Java Collection Interview**

**Q:- What is difference between ArrayList and vector?**

**Ans:-**

* **Synchronization** - ArrayList is not thread-safe whereas Vector is thread-safe. In Vector class each method like add(), get(int i) is surrounded with a synchronized block and thus making Vector class thread-safe.
* **Data growth** - Internally, both the ArrayList and Vector hold onto their contents using an Array. When an element is inserted into an ArrayList or a Vector, the object will need to expand its internal array if it runs out of room. A Vector defaults to doubling the size of its array, while the ArrayList increases its array size by 50 percent.
* ArrayList is way faster than Vector. Since Vector is synchronized and thread-safe it pays price of synchronization which makes it little slow. On the other hand ArrayList is not synchronized and fast which makes it obvious choice in a single-threaded access environment.
* Legacy: Another point worth to remember is Vector is one of those classes which comes with JDK 1.0 and initially not part of Collection framework but in later version it's been re-factored to implement List interface so that it could become part of collection framework.

Each JVM has a heap size of 1.2 GB?

**Q:- How can Arraylist be synchronized without using Vector?**

**Ans:-** Arraylist can be synchronized using:

        Collection.synchronizedList(List list)

Other collections can be synchronized:

        Collection.synchronizedMap(Map map)

Collection.synchronizedCollection(Collection c)

**Q:- If an Employee class is present and its objects are added in an arrayList. Now I want the list to be sorted on the basis of the employeeID of Employee class. What are the steps?**

**Ans:-**

1. Implement Comparable interface for the Employee class and override the compareTo(Object obj) method in which compare the employeeID.

2. Now call Collections.sort() method and pass list as an argument.

**Q:- How about when Employee class is a jar file and you do not have access to its source code?**

**Ans:-** Since Comparable interface cannot be implemented, create Comparator and override the compare(Object obj, Object obj1) method .

Call Collections.sort() on the list and pass Comparator as an argument.

**Q:- What is difference between HashMap and HashTable?**

**Ans:-**

1. Both collections implements Map.

2. Both collections store value as key-value pairs.

**The key differences between the two are**

1. HashMap is not synchronized in nature but HashTable is.

2. Iterator in the HashMap is fail-safe while the enumerator for the HashTable isn't fail-safe. If the HashTable is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a concurrentModificationException.

3. HashMap permits null values and only one null key, while HashTable doesn't allow key or value as null.

4. HashMap does not guarantee that the order of the map will remain constant over time. But one of HashMap's subclasses is LinkedHashMap, so in the event that you'd want predictable iteration order (which is insertion order by default), you can easily swap out the HashMap for a LinkedHashMap. This wouldn't be as easy if you were using Hashtable.

**Q:- What is the prominent implementation of List interface?**

**Ans:-** The classes that implement List interface:

**1. ArrayList:** It is a resizable array implementation. The size of the ArrayList can be increased dynamically also operations like add, remove and get can be formed once the object is created. It also ensures that the data is retrieved in the manner it was stored. The ArrayList is not thread-safe.

**2. Vector:** It is thread-safe implementation of ArrayList. The methods are wrapped around a synchronized block.

**3. LinkedList:** the LinkedList also implements Queue interface and provide FIFO(First In First Out) operation for add operation. It is faster if than ArrayList if it performs insertion and deletion of elements from the middle of a list.

**Q:- Which all classes implement Set interface?**

**Ans:-**

**1- SortedSet:-** It is an interface which extends Set. A the name suggest , the interface allows the data to be iterated in the ascending order or sorted on the basis of Comparator or Comparable interface. All elements inserted into the interface must implement Comparable or Comparator interface.

**2- TreeSet:-** It is the implementation of SortedSet interface. This implementation provides guaranteed log(n) time cost for the basic operations (add, remove and contains). The class is not synchronized.

**3- HashSet:-** This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element. This class offers constant time performance for the basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets

**Q:- What is difference between List and a Set?**

**Ans:-**

1. List can contain duplicate values but Set doesn’t allow. Set allows only to unique elements.

2. List allows retrieval of data to be in same order in the way it is inserted but Set doesn’t ensures the sequence in which data can be retrieved.(Except HashSet)

**Q:- What is difference between Arrays and ArrayList ?**

**Ans:-**

1- Arrays are created of fix size whereas ArrayList is of not fix size. It means that once array is declared as:

int [] intArray= new int[6];

intArray[7]   // will give ArraysOutOfBoundException.

Also the size of array cannot be incremented or decremented. But with ArrayList the size is variable. Once the array is created elements cannot be added or deleted from it. But with

2- ArrayList the elements can be added and deleted at runtime.

List list = new ArrayList();

list.add(1);

list.add(3);

list.remove(0) // will remove the element from the 1st location.

3- ArrayList is one dimensional but array can be multidimensional.

int[][][] intArray= new int[3][2][1];   // 3 dimensional array

To create an array the size should be known or initialized to some value. If not initialized carefully there could me memory wastage. But arrayList is all about dynamic creation and there is no wastage of memory.

**Q:- When to use ArrayList or LinkedList ?**

**Ans:-**

1.  Adding new elements is pretty fast for either type of list.

2.  For the ArrayList, doing random lookup using "get" is fast, but for LinkedList, it's slow. It's slow because there's no efficient way to index into the middle of a linked list.

3. When removing elements, using ArrayList is slow. This is because all remaining elements in the underlying array of Object instances must be shifted down for each remove operation. But here LinkedList is fast, because deletion can be done simply by changing a couple of links. So an ArrayList works best for cases where you're doing random access on the list, and a LinkedList works better if you're doing a lot of editing in the middle of the list.

**Q:- Consider a scenario. If an ArrayList has to be iterated to read data only, what are the possible ways and which is the fastest?**

**Ans:-** It can be done in two ways, using for loop or using iterator of ArrayList. The first option is faster than using iterator. Because value stored in array list is indexed access. So while accessing the value is accessed directly as per the index.

**Q:- Now another question with respect to above question is if accessing through iterator is slow then why do we need it and when to use it.**

**Ans:-** For loop does not allow the updation in the array(add or remove operation) inside the loop whereas Iterator does. Also Iterator can be used where there is no clue what type of collections will be used because all collections have iterator.

**Q:- Which design pattern Iterator follows?**

**Ans:-** It follows Iterator design pattern. Iterator Pattern is a type of behavioral pattern. The Iterator pattern is one, which allows you to navigate through a collection of data using a common interface without knowing about the underlying implementation. Iterator should be implemented as an interface. This allows the user to implement it anyway it's easier for him/her to return data. The benefits of Iterator are about their strength to provide a common interface for iterating through collections without bothering about underlying implementation.

Example of Iteration design pattern - Enumeration The class java.util.Enumeration is an example of the Iterator pattern. It represents and abstract means of iterating over a collection of elements in some sequential order without the client having to know the representation of the collection being iterated over. It can be used to provide a uniform interface for traversing collections of all kinds.

**Q:- Is it better to have a HashMap with large number of records or n number of small hashMaps?**

**Ans:-** It depends on the different scenario one is working on:

1) If the objects in the hashMap are same then there is no point in having different hashmap as the traverse time in a hashmap is invariant to the size of the Map.

2) If the objects are of different type like one of Person class, other of Animal class etc then also one can have single hashmap but different hashmap would score over it as it would have better readability.

**Q:- Why is it preferred to declare:**

List<String> list = new ArrayList<String>();

instead of

ArrayList<String> = new ArrayList<String>();

**Ans:-** It is preferred because:

If later on code needs to be changed from ArrayList to Vector then only at the declaration place we can do that.

The most important one – If a function is declared such that it takes list. E.g void showDetails(List list);

When the parameter is declared as List to the function it can be called by passing any subclass of List like ArrayList,Vector,LinkedList making the function more flexible

**Q:- What is difference between iterator access and index access?**

**Ans:-** Index based access allow access of the element directly on the basis of index. The cursor of the data structure can directly goto the 'n' location and get the element. It does not traverse through n-1 elements.

In Iterator based access, the cursor has to traverse through each element to get the desired element. So to reach the 'nth' element it need to traverse through n-1 elements.

Insertion, update or deletion will be faster for iterator based access if the operations are performed on elements present in between the data structure.

Insertion, update or deletion will be faster for index based access if the operations are performed on elements present at last of the data structure.

Traversal or search in index based data structure is faster.

ArrayList is index access and LinkedList is iterator access.

**Q:- How to sort list in reverse order?**

**Ans:-** To sort the elements of the List in the reverse natural order of the strings, get a reverse Comparator from the Collections class with reverseOrder(). Then, pass the reverse Comparator to the sort() method.

List list = new ArrayList();

Comparator comp = Collections.reverseOrder();

Collections.sort(list, comp)

**Q:- Can a null element added to a Treeset or HashSet?**

**Ans:-** A null element can be added only if the set contains one element because when a second element is added then as per set definition a check is made to check duplicate value and comparison with null element will throw NullPointerException.

HashSet is based on hashMap and can contain null element.

**Q:- How to sort list of strings - case insensitive?**

**Ans:-** using Collections.sort(list, String.CASE\_INSENSITIVE\_ORDER);

**Q:- How to make a List (ArrayList,Vector,LinkedList) read only?**

**Ans:-** A list implementation can be made read only using Collections.unmodifiableList(list). This method returns a new list. If a user tries to perform add operation on the new list; UnSupportedOperationException is thrown.

**Q:- What is ConcurrentHashMap?**

**Ans:-** A concurrentHashMap is thread-safe implementation of Map interface. In this class put and remove method are synchronized but not get method. This class is different from Hashtable in terms of locking; it means that hashtable use object level lock but this class uses bucket level lock thus having better performance.

**Q:- Which is faster to iterate LinkedHashSet or LinkedList?**

**Ans:-** LinkedList.

**Q:- Which data structure HashSet implements?**

**Ans:-** HashSet implements hashmap internally to store the data. The data passed to hashset is stored as key in hashmap with null as value.

**Q:- Arrange in the order of speed –**

**HashMap, HashTable, Collections.synchronizedMap, concurrentHashmap**

**Ans:-** HashMap is fastest, ConcurrentHashMap,Collections.synchronizedMap,HashTable.

**Q:- What is identityHashMap?**

**Ans:-** The IdentityHashMap uses == for equality checking instead of equals(). This can be used for both performance reasons, if you know that two different elements will never be equals and for preventing spoofing, where an object tries to imitate another.

**Q:- What is WeakHashMap?**

**Ans:-** A hashtable-based Map implementation with weak keys. An entry in a 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, that is, made finalizable, finalized, and then reclaimed. When a key has been discarded its entry is effectively removed from the map, so this class behaves somewhat differently than other Map implementations.

**Q:- What is an Iterator?**

**Ans:-** Some of the collection classes provide traversal of their contents via a java.util.Iterator interface. This interface allows you to walk through a collection of objects, operating on each object in turn. Remember when using Iterators that they contain a snapshot of the collection at the time the Iterator was obtained; generally it is not advisable to modify the collection itself while traversing an Iterator.

**Q:- What is the difference between java.util.Iterator and java.util.ListIterator?**

**Ans:-** Iterator : Enables you to traverse through a collection in the forward direction only, for obtaining or removing elements.

ListIterator : extends Iterator, and allows bidirectional traversal of list and also allows the modification of elements.

**Q:-What is fail-fast property?**

**Ans:-** At high level - Fail-fast is a property of a system or software with respect to its response to failures. A fail-fast system is designed to immediately report any failure or condition that is likely to lead to failure. Fail-fast systems are usually designed to stop normal operation rather than attempt to continue a possibly-flawed process. When a problem occurs, a fail-fast system fails immediately and visibly. Failing fast is a non-intuitive technique: "failing immediately and visibly" sounds like it would make your software more fragile, but it actually makes it more robust. Bugs are easier to find and fix, so fewer go into production. In Java, Fail-fast term can be related to context of iterators. If an iterator 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.

**Q:-Why doesn't Collection extend Cloneable and Serializable?**

**Ans:-** Many Collection implementations (including all of the ones provided by the JDK) will have a public clone method, but it would be mistake to require it of all Collections. For example, what does it mean to clone a Collection that's backed by a terabyte SQL database? Should the method call because the company to requisition a new disk farm? Similar arguments hold for serializable. If the client doesn't know the actual type of a Collection, it's much more flexible and less error prone to have the client decide what type of Collection is desired, create an empty Collection of this type, and use the addAll method to copy the elements of the original collection into the new one. Note on Some Important Terms

* 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 while others will wait for lock to be released.
* Fail-fast is relevant from the context of iterators. If an iterator 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.

**Q:-How can we make Hashmap synchronized?**

**Ans:** HashMap can be synchronized by Map m = Collections.synchronizedMap(hashMap);

**Q:-Where will you use Hashtable and where will you use HashMap?**

**Ans:** There are multiple aspects to this decision:

1. The basic difference between a Hashtable and an HashMap is that, Hashtable is synchronized while HashMap is not. Thus whenever there is a possibility of multiple threads accessing the same instance, one should use Hashtable. While if not multiple threads are going to access the same instance then use HashMap. Non synchronized data structure will give better performance than the synchronized one.

2. If there is a possibility in future that - there can be a scenario when you may require to retain the order of objects in the Collection with key-value pair then HashMap can be a good choice. As one of HashMap's subclasses is LinkedHashMap, so in the event that you'd want predictable iteration order (which is insertion order by default), you can easily swap out the HashMap for a LinkedHashMap. This wouldn't be as easy if you were using Hashtable. Also if you have multiple thread accessing you HashMap then Collections.synchronizedMap() method can be leveraged. Overall HashMap gives you more flexibility in terms of possible future changes.

**Q:- What is the Difference between Enumeration and Iterator interface?**

**Ans:** Enumeration and Iterator are the interface available in java.util package. The functionality of Enumeration interface is duplicated by the Iterator interface. New implementations should consider using Iterator in preference to Enumeration. Iterators differ from enumerations in following ways:

* Enumeration contains 2 methods namely hasMoreElements() & nextElement() whereas Iterator contains three methods namely hasNext(), next(),remove().
* Iterator adds an optional remove operation, and has shorter method names. Using remove() we can delete the objects but Enumeration interface does not support this feature.
* Enumeration interface is used by legacy classes. Vector.elements() & Hashtable.elements() method returns Enumeration. Iterator is returned by all Java Collections Framework classes. java.util.Collection.iterator() method returns an instance of Iterator.

**Q:- Why Java Vector class is considered obsolete or unofficially deprecated? or Why should I always use ArrayList over Vector?**

**Ans:** You should use ArrayList over Vector because you should default to non-synchronized access. Vector synchronizes each individual method. That's almost never what you want to do. Generally you want to synchronize a whole sequence of operations. Synchronizing individual operations is both less safe (if you iterate over a Vector, for instance, you still need to take out a lock to avoid anyone else changing the collection at the same time) but also slower (why take out a lock repeatedly when once will be enough)? Of course, it also has the overhead of locking even when you don't need to. It's a very flawed approach to have synchronized access as default. You can always decorate a collection using Collections.synchronizedList - the fact that Vector combines both the "resized array" collection implementation with the "synchronize every operation" bit is another example of poor design; the decoration approach gives cleaner separation of concerns. Vector also has a few legacy methods around enumeration and element retrieval which are different than the List interface, and developers (especially those who learned Java before 1.2) can tend to use them if they are in the code. Although Enumerations are faster, they don't check if the collection was modified during iteration, which can cause issues, and given that Vector might be chosen for its synchronization - with the attendant access from multiple threads, this makes it a particularly pernicious problem. Usage of these methods also couples a lot of code to Vector, such that it won't be easy to replace it with a different List implementation. Despite all above reasons Sun may never officially deprecate Vector class. (Read details Deprecate Hashtable and Vector)

**Q:- What is an enumeration?**

**Ans:** An enumeration is an interface containing methods for accessing the underlying data structure from which the enumeration is obtained. It is a construct which collection classes return when you request a collection of all the objects stored in the collection. It allows sequential access to all the elements stored in the collection.

**Q:- What is the importance of hashCode() and equals() methods? How they are used in Java?**

**Ans:** The java.lang.Object has two methods defined in it. They are -

public boolean equals(Object obj) , public int hashCode().

These two methods are used heavily when objects are stored in collections. There is a contract between these two methods which should be kept in mind while overriding any of these methods. The Java API documentation describes it in detail.

The hashCode() method returns a hash code value for the object. This method is supported for the benefit of hashtables such as those provided by java.util.Hashtable or java.util.HashMap. The general contract of hashCode is: Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application. If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result. It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables. As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects.

The equals(Object obj) method indicates whether some other object is "equal to" this one. The equals method implements an equivalence relation on non-null object references: It is reflexive: for any non-null reference value x, x.equals(x) should return true. It is symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true. It is transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true. It is consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified. For any non-null reference value x, x.equals(null) should return false. The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true). Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes. A practical Example of hashcode() & equals():This can be applied to classes that need to be stored in Set collections. Sets use equals() to enforce non-duplicates, and HashSet uses hashCode() as a first-cut test for equality. Technically hashCode() isn't necessary then since equals() will always be used in the end, but providing a meaningful hashCode() will improve performance for very large sets or objects that take a long time to compare using equals().

**Q:-What is the difference between Sorting performance of Arrays.sort() vs Collections.sort() ? Which one is faster? Which one to use and when?**

**Ans:** Many developers are concerned about the performance difference between java.util.Array.sort() java.util.Collections.sort() methods. Both methods have same algorithm the only difference is type of input to them. Collections.sort() has a input as List so it does a translation of List to array and vice versa which is an additional step while sorting. So this should be used when you are trying to sort a list. Arrays.sort is for arrays so the sorting is done directly on the array. So clearly it should be used when you have a array available with you and you want to sort it.

**Q:- What is java.util.concurrent BlockingQueue? How it can be used?**

**Ans:-** Java has implementation of BlockingQueue available since Java 1.5. Blocking Queue interface extends collection interface, which provides you power of collections inside a queue. Blocking Queue is a type of Queue that additionally 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. A typical usage example would be based on a producer-consumer scenario. Note that a BlockingQueue can safely be used with multiple producers and multiple consumers. An ArrayBlockingQueue is a implementation of blocking queue with an array used to store the queued objects. The head of the queue is that element that has been on the queue the longest time. The tail of the queue is that element that has been on the queue the shortest time. New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue. ArrayBlockingQueue requires you to specify the capacity of queue at the object construction time itself. Once created, the capacity cannot be increased. This is a classic "bounded buffer" (fixed size buffer), in which a fixed-sized array holds elements inserted by producers and extracted by consumers. Attempts to put an element to a full queue will result in the put operation blocking; attempts to retrieve an element from an empty queue will be blocked.

**Q:- Set & List interface extend Collection, so Why doesn't Map interface extend Collection?**

**Ans:-** Though the Map interface is part of collections framework, it does not extend collection interface. This is by design, and the answer to this questions is best described in Sun's FAQ Page: This was by design. We feel that mappings are not collections and collections are not mappings. Thus, it makes little sense for Map to extend the Collection interface (or vice versa). If a Map is a Collection, what are the elements? The only reasonable answer is "Key-value pairs", but this provides a very limited (and not particularly useful) Map abstraction. You can't ask what value a given key maps to, nor can you delete the entry for a given key without knowing what value it maps to. Collection could be made to extend Map, but this raises the question: what are the keys? There's no really satisfactory answer, and forcing one leads to an unnatural interface. Maps can be viewed as Collections (of keys, values, or pairs), and this fact is reflected in the three "Collection view operations" on Maps (keySet, entrySet, and values). While it is, in principle, possible to view a List as a Map mapping indices to elements, this has the nasty property that deleting an element from the List changes the Key associated with every element before the deleted element. That's why we don't have a map view operation on Lists.

**Q:-What is the difference between ArrayList and LinkedList? (ArrayList vs LinkedList.)**

**Ans:-** java.util.ArrayList and java.util.LinkedList are two Collections classes used for storing lists of object references Here are some key differences:

1. ArrayList uses primitive object array for storing objects whereas LinkedList is made up of a chain of nodes. Each node stores an element and the pointer to the next node. A singly linked list only has pointers to next. A doubly linked list has a pointer to the next and the previous element. This makes walking the list backward easier.
2. ArrayList implements the RandomAccess interface, and LinkedList does not. The commonly used ArrayList implementation uses primitive Object array for internal storage. Therefore an ArrayList is much faster than a LinkedList for random access, that is, when accessing arbitrary list elements using the get method. Note that the get method is implemented for LinkedLists, but it requires a sequential scan from the front or back of the list. This scan is very slow. For a LinkedList, there's no fast way to access the Nth element of the list.
3. Adding and deleting at the start and middle of the ArrayList is slow, because all the later elements have to be copied forward or backward. (Using System.arrayCopy()) Whereas Linked lists are faster for inserts and deletes anywhere in the list, since all you do is update a few next and previous pointers of a node.
4. Each element of a linked list (especially a doubly linked list) uses a bit more memory than its equivalent in array list, due to the need for next and previous pointers.
5. ArrayList may also have a performance issue when the internal array fills up. The arrayList has to create a new array and copy all the elements there. The ArrayList has a growth algorithm of (n\*3)/2+1, meaning that each time the buffer is too small it will create a new one of size (n\*3)/2+1 where n is the number of elements of the current buffer. Hence if we can guess the number of elements that we are going to have, then it makes sense to create a arraylist with that capacity during object creation (using construtor new ArrayList(capacity)). Whereas LinkedLists should not have such capacity issues.

**Q:- What is performance of various Java collection implementations/algorithms? What is Big 'O' notation for each of them?**

**Ans:-** Each java collection implementation class have different performance for different methods, which makes them suitable for different programming needs.

**Performance of Map interface implementations**

**Hashtable**

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

**HashMap**

This implementation provides constant-time [ Big O Notation is O(1) ] performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

**TreeMap**

The TreeMap implementation provides guaranteed log(n) [ Big O Notation is O(log N) ] time cost for the containsKey, get, put and remove operations.

LinkedHashMap

A linked hash map has two parameters that affect its performance: initial capacity and load factor. They are defined precisely as for HashMap. Note, however, that the penalty for choosing an excessively high value for initial capacity is less severe for this class than for HashMap, as iteration times for this class are unaffected by capacity.

**Performance of Set interface implementations**

**HashSet**

The HashSet class offers constant-time [ Big O Notation is O(1) ] performance for the basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets. Iterating over this set requires time proportional to the sum of the HashSet instance's size (the number of elements) plus the "capacity" of the backing HashMap instance (the number of buckets). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

**TreeSet**

The TreeSet implementation provides guaranteed log(n) time cost for the basic operations (add, remove and contains).

**LinkedHashSet**

A linked hash set has two parameters that affect its performance: initial capacity and load factor. They are defined precisely as for HashSet. Note, however, that the penalty for choosing an excessively high value for initial capacity is less severe for this class than for HashSet, as iteration times for this class are unaffected by capacity.

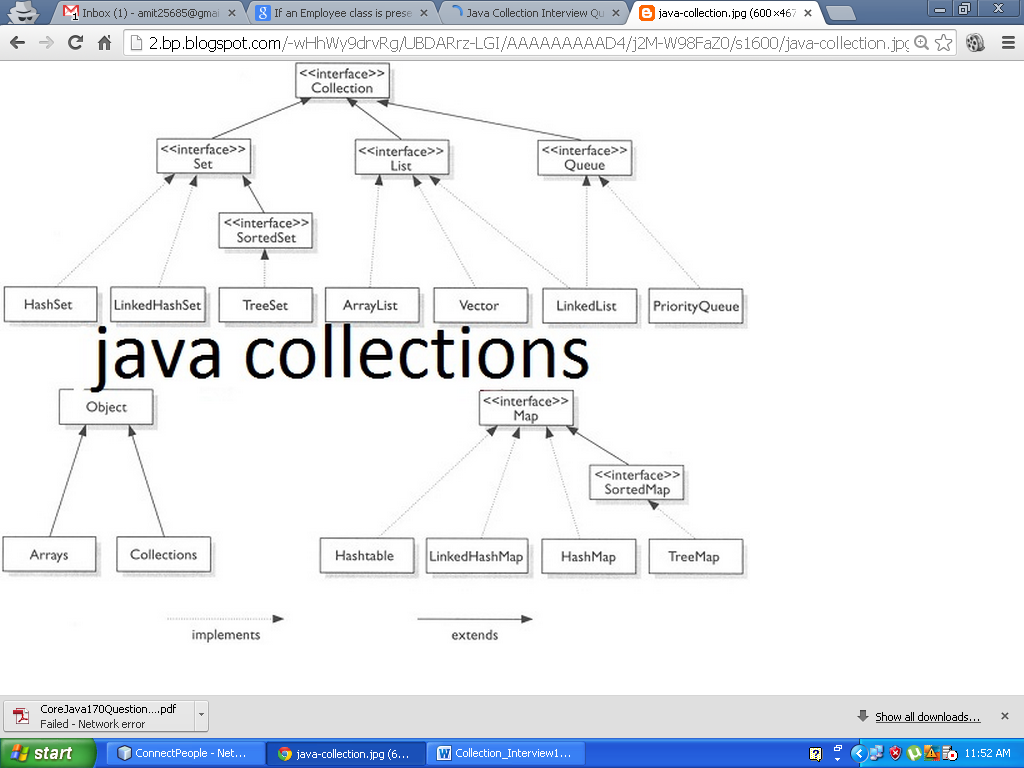
**Performance of List interface implementations**

**LinkedList**

- Performance of get and remove methods is linear time [ Big O Notation is O(n) ] - Performance of add and Iterator.remove methods is constant-time [ Big O Notation is O(1) ]

ArrayList

- The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. [ Big O Notation is O(1) ] - The add operation runs in amortized constant time [ Big O Notation is O(1) ] , but in worst case (since the array must be resized and copied) adding n elements requires linear time [ Big O Notation is O(n) ] - Performance of remove method is linear time [ Big O Notation is O(n) ] - All of the other operations run in linear time [ Big O Notation is O(n) ]. The constant factor is low compared to that for the LinkedList implementation.



**Q:-What are Java Comparators and Comparables?**

**Ans:-** As both names suggest (and you may have guessed), these are used for comparing objects in Java. Using these concepts; Java objects can be sorted according to a predefined order.

Two of these concepts can be explained as follows.

**Comparable**

A comparable object is capable of comparing itself with another object. The class itself must implements the java.lang.Comparable interface in order to be able to compare its instances.

**Comparator**

A comparator object is capable of comparing two different objects. The class is not comparing its instances, but some other class’s instances. This comparator class must implement the java.util.Comparator interface.

Do we need to compare objects? The simplest answer is yes. When there is a list of objects, ordering these objects into different orders becomes a must in some situations. For example; think of displaying a list of employee objects in a web page. Generally employees may be displayed by sorting them using the employee id. Also there will be requirements to sort them according to the name or age as well. In these situations both these (above defined) concepts will become handy.

How to use these?

There are two interfaces in Java to support these concepts, and each of these has one method to be implemented by user.

Those are;

**java.lang.Comparable: int compareTo(Object o1)**

This method compares this object with o1 object. Returned int value has the following meanings.

1.      positive – this object is greater than o1

2.      zero – this object equals to o1

3.      negative – this object is less than o1

java.util.Comparator: int compare(Object o1, Object o2)

This method compares o1 and o2 objects. Returned int value has the following meanings.

1.      positive – o1 is greater than o2

2.      zero – o1 equals to o2

3.      negative – o1 is less than o1

java.util.Collections.sort(List) and java.util.Arrays.sort(Object[]) methods can be used to sort using natural ordering of objects.

java.util.Collections.sort(List, Comparator) and java.util.Arrays.sort(Object[], Comparator) methods can be used if a Comparator is available for comparison.

The above explained Employee example is a good candidate for explaining these two concepts. First we’ll write a simple Java bean to represent the Employee.

*public class Employee {*

*private int empId;*

*private String name;*

*private int age;*

*public Employee(int empId, String name, int age) {*

*// set values on attributes*

*}*

*// getters & setters*

*}*

Next we’ll create a list of Employees for using in different sorting requirements. Employees are added to a List without any specific order in the following class.

*import java.util.\*;*

*public class Util {*

*public static List<Employee> getEmployees() {*

*List<Employee> col = new ArrayList<Employee>();*

*col.add(new Employee(5, "Frank", 28));*

*col.add(new Employee(1, "Jorge", 19));*

*col.add(new Employee(6, "Bill", 34));*

*col.add(new Employee(3, "Michel", 10));*

*col.add(new Employee(7, "Simpson", 8));*

*col.add(new Employee(4, "Clerk",16 ));*

*col.add(new Employee(8, "Lee", 40));*

*col.add(new Employee(2, "Mark", 30));*

*return col;*

*}*

*}*

Sorting in natural ordering

Employee’s natural ordering would be done according to the employee id. For that, above Employee class must be altered to add the comparing ability as follows.

*public class Employee implements Comparable<Employee> {*

*private int empId;*

*private String name;*

*private int age;*

*/\*\**

*\* Compare a given Employee with this object.*

*\* If employee id of this object is*

*\* greater than the received object,*

*\* then this object is greater than the other.*

*\*/*

*public int compareTo(Employee o) {*

*return this.empId - o.empId ;*

*}*

*….*

*}*

The new compareTo() method does the trick of implementing the natural ordering of the instances. So if a collection of Employee objects is sorted using Collections.sort(List) method; sorting happens according to the ordering done inside this method.

We’ll write a class to test this natural ordering mechanism. Following class use the Collections.sort(List) method to sort the given list in natural order.

*import java.util.\*;*

*public class TestEmployeeSort {*

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

*List coll = Util.getEmployees();*

*Collections.sort(coll); // sort method*

*printList(coll);*

*}*

*private static void printList(List<Employee> list) {*

*System.out.println("EmpId\tName\tAge");*

*for (Employee e: list) {*

*System.out.println(e.getEmpId() + "\t" + e.getName() + "\t" + e.getAge());*

*}*

*}*

*}*

EmpId Name  Age

1 Jorge   19

2 Mark 30

3 Michel  10

4 Clerk   16

5 Frank   28

6 Bill 34

7 Simp 8

8    Lee 40

Sorting by other fields

If we need to sort using other fields of the employee, we’ll have to change the Employee class’s compareTo() method to use those fields. But then we’ll lose this empId based sorting mechanism. This is not a good alternative if we need to sort using different fields at different occasions. But no need to worry; Comparator is there to save us.

By writing a class that implements the java.util.Comparator interface, you can sort Employees using any field as you wish even without touching the Employee class itself; Employee class does not need to implement java.lang.Comparable or java.util.Comparator interface.

Sorting by name field

Following EmpSortByName class is used to sort Employee instances according to the name field. In this class, inside the compare() method sorting mechanism is implemented. In compare() method we get two Employee instances and we have to return which object is greater.

public class EmpSortByName implements Comparator<Employee>{

public int compare(Employee o1, Employee o2) {

    return o1.getName().compareTo(o2.getName());

}

}

Watch out: Here, String class’s compareTo() method is used in comparing the name fields (which are Strings).

Now to test this sorting mechanism, you must use the Collections.sort(List, Comparator) method instead of Collections.sort(List) method. Now change the TestEmployeeSort class as follows. See how the EmpSortByName comparator is used inside sort method.Watch out: Here, String class’s compareTo() method is used in comparing the name fields (which are Strings).

Now to test this sorting mechanism, you must use the Collections.sort(List, Comparator) method instead of Collections.sort(List) method. Now change the TestEmployeeSort class as follows. See how the EmpSortByName comparator is used inside sort method.

*import java.util.\*;*

*public class TestEmployeeSort {*

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

*List coll = Util.getEmployees();*

*//Collections.sort(coll);*

*//use Comparator implementation*

*Collections.sort(coll, new EmpSortByName());*

*printList(coll);*

*}*

*private static void printList(List<Employee> list) {*

*System.out.println("EmpId\tName\tAge");*

*for (Employee e: list) {*

*System.out.println(e.getEmpId() + "\t" + e.getName() + "\t" + e.getAge());*

*}*

*}*

*}*

Now the result would be as follows. Check whether the employees are sorted correctly by the name String field. You’ll see that these are sorted alphabetically.

EmpId Name Age

6 Bill 34

4 Clerk 16

5 Frank 28

1 Jorge 19

8 Lee 40

2 Mark 30

3 Michel 10

7 Simp 8

Sorting by empId field

Even the ordering by empId (previously done using Comparable) can be implemented using Comparator; following class does that.

public class EmpSortByEmpId implements Comparator<Employee>{

public int compare(Employee o1, Employee o2) {

    return o1.getEmpId().compareTo(o2.getEmpId());

}

}

Explore further

Do not stop here. Work on the followings by yourselves and sharpen knowledge on these concepts.

1.      Sort employees using name, age, empId in this order (ie: when names are equal, try age and then next empId)

2.      Explore how & why equals() method and compare()/compareTo() methods must be consistence.

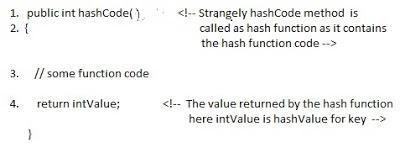
If you have any issues on these concepts; please add those in the comments section and we’ll get back to you.

**Q:- Hashing: How hashmap works in java or How get () method works internally**

**Ans: -** HashMap works on the principle of Hashing.  To understand Hashing, we should understand the three terms first   i.e. Hash Function, Hash Value and Bucket.

**What is Hash Function, Hash Value and Bucket?**  
**hashCode()** function  which returns an integer value is the Hash function. The important point to note that, this method is present in [Object class (Mother of all class)](http://javahungry.blogspot.com/2013/06/object-class-and-methods-in-java-example-explanation.html) .  
This is the code for the hash function(also known as hashCode method) in Object Class :  
**public native int hashCode();**

So summarize the terms in the diagram below:



**What is bucket?**

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.

After understanding the terms we are ready to move next step, **How hash map works in java** or How get() works internally in java.

**Code inside Java Api (HashMap class internal implementation) for HashMap get(Object key)**

**method**

*Public V get(Object key)*

*{*

*if (key ==null)*

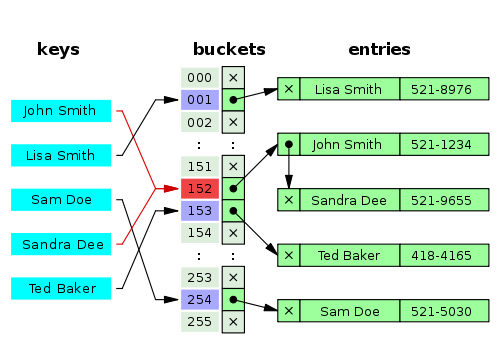
*//Some code*

*int hash = hash(key.hashCode());*

*// if key found in hash table then return value*

*// else return null*

*}*

**Hashmap works on the principle of hashing**   
HashMap get(Key k) method calls hashCode method on the key object and applies returned hash Value to its own static hash function to find a bucket location(backing array) where keys and values are stored in form of nested class called Entry (Map.Entry) . So you have concluded that from the previous line that both key and value is stored in the bucket as a form of Entry object. So thinking that only value is stored in the bucket is not correct and will not give a good impression on the interviewer.  
  
\* Whenever we call get (Key k) method on the HashMap object. First it checks that whether key is null or not.  Note that there can only be one null key in HashMap.   
If key is null, then Null keys always map to hash 0, thus index 0.  
If key is not null then, it will call hashfunction on the key object, see line 4 in above method i.e. key.hashCode(), so after key.hashCode() returns hashValue, line 4 looks like  
  
**4. int hash = hash(hashValue)**  
 , and now ,it applies returned hashValue into its own hashing function .  
  
We might wonder why we are calculating the hashvalue again using hash(hashValue). Answer is ,It defends against poor quality hash functions.  
  
Now step 4 final hash value is used to find the bucket location at which the Entry object is stored. Entry object stores in the bucket like this (hash,key,value,bucketindex) .    
  
**Interviewer: What if when two different keys have the same hashcode?**  
Solution, [equals () method](http://javahungry.blogspot.com/2013/06/difference-between-equals-and-double-equals-method-with-example-java-collections-interview-question.html) comes to rescue. Here candidate gets puzzled. Since bucket is one and we have two objects with the same hashcode .Candidate usually forgets that bucket is a simple linked list.  
  
The bucket is the linked list effectively. It’s not a LinkedList as in a java.util.LinkedList - It's a separate (simpler) implementation just for the map.  
So we traverse through linked list, comparing keys in each entries using keys.equals() until it return true. Then the corresponding entry object Value is returned.  


One of our readers Jammy asked a very good question.   
**When the functions 'equals' traverses through the linked list does it traverses from start to end one by one...in other words brute method. Or the linked list is sorted based on key and then it traverses?**Answer is when an element is added/retrieved, same procedure follows:  
a. Using key.hashCode() [ see above step 4],determine initial hashvalue for the key  
b. Pass intial hashvalue as hashValue  in    hash(hashValue) function, to calculate the final hashvalue.  
c. Final hash value is then passed as a first parameter in the indexFor(int ,int )method .  
    The second parameter is length which is a constant in HashMap Java Api , represented by                             DEFAULT\_INITIAL\_CAPACITY  
The default  value of DEFAULT\_INITIAL\_CAPACITY is 16 in HashMap java api, indexFor(int,int) method  returns the first entry in the appropriate bucket. The linked list in the bucket is then iterated over - (the end is found and the element is added or the key is matched and the value is returned)   
  
Explanation about indexFor(int,int) is below :  
/\*\*

\* Returns index for hash code h.

\*/

static int indexFor(int h, int length) {

return h & (length-1);

}  
The above function indexFor() works because Java HashMaps always have a capacity, i.e. number of buckets, as a power of 2. Let's work with a capacity of 256,which is 0x100, but it could work with any power of 2. Subtracting 1 from a power of 2 yields the exact bit mask needed to bitwise-and with the hash to get the proper bucket index, of range 0 to length - 1.  
256 - 1 = 255  
0x100 - 0x1 = 0xFF  
E.g. a hash of 257 (0x101) gets bitwise-anded with 0xFF to yield a bucket number of 1.  
  
**Interviewer:    What if  when two  keys are same and have the same hashcode ?**If key needs to be inserted and already inserted hashkey's hashcodes are same, and keys are also same(via reference or using equals() method)  then override the previous key value pair with the current key value pair.  
The other important point to note is that in Map ,Any class(String etc.) can serve as a key if and only if it overrides the equals() and hashCode() method .

**Interviewer:  How will you measure the performance of HashMap?**  
According to [Oracle Java docs](http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html).    
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.

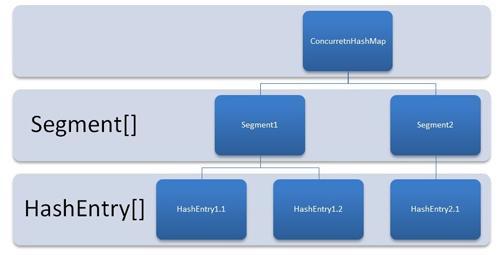
In HashMap class, the default value of load factor is (.75) .

**Q:- What is difference between HashMap and ConcurrentHashMap?**

**Ans:-** To better visualize the ConcurrentHashMap, let it consider as a group of HashMaps. To get and put key-value pairs from hashmap, you have to calculate the hashcode and look for correct bucket location in array of Collection.Entry. Rest you have read on previous related article on how hashmap works.

In concurrentHashMap, the **difference lies in internal structure to store these key-value pairs**. ConcurrentHashMap has an addition concept of segments. It will be easier to understand it you think of one segment equal to one HashMap [conceptually]. A concurrentHashMap is divided into number of segments [default 16] on initialization. ConcurrentHashMap allows similar number (16) of threads to access these segments concurrently so that each thread work on a specific segment during high concurrency.

This way, if when your key-value pair is stored in segment 10; code does not need to block other 15 segments additionally. This structure provides a very high level of concurrency.



**ConcurrentHashMap Internal Structure**

In other words, ConcurrentHashMap uses a multitude of locks, each lock controls one segment of the map. When setting data in a particular segment, the lock for that segment is obtained. So essentially update operations are synchronized.  
When getting data, a volatile read is used without any synchronization. If the volatile read results in a miss, then the lock for that segment is obtained and entry is again searched in synchronized block.

I will go further deeper into this concept in my coming post. I will suggest you to subscribe email updates to get notified.

**Q- What is difference between HashMap and Collections.synchronizedMap(HashMap)?**

**Ans:-** HashMap is non-synchronized and Collections.synchronizedMap() returns a wrapped instance of HashMap which has all get, put methods synchronized.

Essentially, Collections.synchronizedMap() returns the reference of internally created inner-class “SynchronizedMap”, which contains key-value pairs of input HashMap, passed as argument.

This instance of inner class has nothing to do with original parameter HashMap instance and is completely independent.

**Q:- What is difference between ConcurrentHashMap and Collections.synchronizedMap (HashMap)?**

**Ans:-** This one is slightly tougher. Both are synchronized version of HashMap, with difference in their core functionality and internal structure.

As stated above, ConcurrentHashMap is consist of internal segments which can be viewed as independent HashMaps, conceptually. All such segments can be locked by separate threads in high concurrent executions. In this way, multiple threads can get/put key-value pairs from ConcurrentHashMap without blocking/waiting for each other.

In Collections.synchronizedMap(), we get a synchronized version of HashMap and it is accessed in blocking manner. This means if multiple threads try to access synchronizedMap at same time, they will be allowed to get/put key-value pairs one at a time in synchronized manner.

**Q:- What is difference between HashTable and Collections.synchronized(HashMap)?**

**Ans:-** So far you must have got the core idea of the similarities between them. Both are synchronized version of collection. Both have synchronized methods inside class. Both are blocking in nature i.e. multiple threads will need to wait for getting the lock on instance before putting/getting anything out of it.

So what is the difference? Well, NO major difference for above said reasons. Performance is also same for both collections.

Only thing which separates them is the fact HashTable is legacy class promoted into collection framework. It got its own extra features like enumerators.

**Q:- What is difference between random/fixed hashcode() value for key?**

**Ans: -** The impact of both cases (fixed hashcode or random hashcode for keys) will have same result and that is “unexpected behavior“. The very basic need of hashcode in HashMap is to identify the bucket location where to put the key-value pair, and from where it has to be retrieved.

If the hashcode of key object changes every time, the exact location of key-value pair will be calculated different, every time. This way, one object stored in HashMap will be lost forever and there will be very minimum possibility to get it back from map.

For this same reason, key are suggested to be immutable, so that they return a unique and same hashcode each time requested on same key object.

**Q:- How to create immutable class?**

**Ans:-** Java documentation itself has some guidelines identified [in this link](http://classes). We will understand what they mean actually:

**1) Don’t provide “setter” methods — methods that modify fields or objects referred to by fields.**

This principle says that for all mutable properties in your class, do not provide setter methods. Setter methods are meant to change the state of object and this is what we want to prevent here.

**2) Make all fields final and private**

This is another way to increase immutability. Fields declared private will not be accessible outside the class and making them final will ensure the even accidentally you can not change them.

**3) Don’t allow subclasses to override methods**

The simplest way to do this is to declare the class as ***final***. Final classes in java can not be overridden.

**4) Special attention when having mutable instance variables**

Always remember that your instance variables will be either **mutable or immutable**. Identify them and **return new objects with copied content for all mutable objects**. Immutable variables can be returned safely without extra effort.

A more sophisticated approach is to make the constructor ***private*** and construct instances in **factory methods**.

Lets add all above rules and make something concrete class implementation.

import java.util.Date;

/\*\*

\* Always remember that your instance variables will be either mutable or immutable.

\* Identify them and return new objects with copied content for all mutable objects.

\* Immutable variables can be returned safely without extra effort.

\* \*/

public final class ImmutableClass

{

/\*\*

\* Integer class is immutable as it does not provide any setter to change its content

\* \*/

private final Integer immutableField1;

/\*\*

\* String class is immutable as it also does not provide setter to change its content

\* \*/

private final String immutableField2;

/\*\*

\* Date class is mutable as it provide setters to change various date/time parts

\* \*/

private final Date mutableField;

//Default private constructor will ensure no unplanned construction of class

private ImmutableClass(Integer fld1, String fld2, Date date)

{

this.immutableField1 = fld1;

this.immutableField2 = fld2;

this.mutableField = new Date(date.getTime());

}

//Factory method to store object creation logic in single place

public static ImmutableClass createNewInstance(Integer fld1, String fld2, Date date)

{

return new ImmutableClass(fld1, fld2, date);

}

//Provide no setter methods

/\*\*

\* Integer class is immutable so we can return the instance variable as it is

\* \*/

public Integer getImmutableField1() {

return immutableField1;

}

/\*\*

\* String class is also immutable so we can return the instance variable as it is

\* \*/

public String getImmutableField2() {

return immutableField2;

}

/\*\*

\* Date class is mutable so we need a little care here.

\* We should not return the reference of original instance variable.

\* Instead a new Date object, with content copied to it, should be returned.

\* \*/

public Date getMutableField() {

return new Date(mutableField.getTime());

}

@Override

public String toString() {

return immutableField1 +" - "+ immutableField2 +" - "+ mutableField;

}

}

Now its time to test our class:

class TestMain

{

public static void main(String[] args)

{

ImmutableClass im = ImmutableClass.createNewInstance(100,"test", new Date());

System.out.println(im);

tryModification(im.getImmutableField1(),im.getImmutableField2(),im.getMutableField());

System.out.println(im);

}

private static void tryModification(Integer immutableField1, String immutableField2, Date mutableField)

{

immutableField1 = 10000;

immutableField2 = "test changed";

mutableField.setDate(10);

}

}

Output: (content is unchanged)

100 - test - Tue Oct 30 21:34:08 IST 2012

100 - test - Tue Oct 30 21:34:08 IST 2012

As it can be seen that even changing the instance variables using their references does not change their value, so the class is immutable.

**Q:- Working with hashCode and equals methods in java?**

**Ans: -** Sections in this post:

1. Usage of hashCode() and equals()
2. Overriding the default behavior
3. Overriding hashCode() and equals() using Apache Commons Lang
4. Important things to remember
5. Special Attention When Using in ORM

hashCode() and equals() methods have been defined in Object class which is parent class for java objects. For this reason, all java objects inherit a default implementation of these methods.

**Usage of hashCode() and equals()**

hashCode() method is used to get a unique integer for given object. This integer is used for determining the bucket location, when this object needs to be stored in some HashTable like data structure. By default, Object’s hashCode() method returns and integer representation of memory address where object is stored.

equals() method, as name suggest, is used to simply verify the equality of two objects. Default implementation simply check the object references of two objects to verify their equality.

Let’s take an example where your application has Employee object. Let’s create a minimal possible structure of Employee class:

*public class Employee*

*{*

*private Integer id;*

*private String firstname;*

*private String lastName;*

*private String department;*

*public Integer getId() {*

*return id;*

*}*

*public void setId(Integer id) {*

*this.id = id;*

*}*

*public String getFirstname() {*

*return firstname;*

*}*

*public void setFirstname(String firstname) {*

*this.firstname = firstname;*

*}*

*public String getLastName() {*

*return lastName;*

*}*

*public void setLastName(String lastName) {*

*this.lastName = lastName;*

*}*

*public String getDepartment() {*

*return department;*

*}*

*public void setDepartment(String department) {*

*this.department = department;*

*}*

*}*

Above Employee class has some very basic attributes and there accessor methods. Now consider a simple situation where you need to compare two employee objects.

*public class EqualsTest {*

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

*Employee e1 = new Employee();*

*Employee e2 = new Employee();*

*e1.setId(100);*

*e2.setId(100);*

*//Prints false in console*

*System.out.println(e1.equals(e2));*

*}*

*}*

No prize for guessing. Above method will print “false“. But, is it really correct after knowing that both objects represent same employee. In a real time application, this must return true.

To achieve correct behavior, we need to override equals method as below:

*public boolean equals(Object o) {*

*if(o == null)*

*{*

*return false;*

*}*

*if (o == this)*

*{*

*return true;*

*}*

*if (getClass() != o.getClass())*

*{*

*return false;*

*}*

*Employee e = (Employee) o;*

*return (this.getId() == e.getId());*

*}*

Add this method to your Employee class, and EqualsTest will start returning “true“.

So are we done? Not yet. Lets test again above modified Employee class in different way.

*import java.util.HashSet;*

*import java.util.Set;*

*public class EqualsTest*

*{*

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

*{*

*Employee e1 = new Employee();*

*Employee e2 = new Employee();*

*e1.setId(100);*

*e2.setId(100);*

*//Prints 'true'*

*System.out.println(e1.equals(e2));*

*Set employees = new HashSet();*

*employees.add(e1);*

*employees.add(e2);*

*//Prints two objects*

*System.out.println(employees);*

*}*

*}*

Above class prints two objects in second print statement. If both employee objects have been equal, in a Set which stores only unique objects, there must be only one instance inside HashSet, after all both objects refer to same employee. What is it we are missing??

We are missing the second important method hashCode(). As java docs say, if you override equals() method then you must override hashCode() method. So let's add another method in our Employee class.

*@Override*

*public int hashCode()*

*{*

*final int PRIME = 31;*

*int result = 1;*

*result = PRIME \* result + getId();*

*return result;*

*}*

Once above method is added in Employee class, the second statement start printing only single object in second statement, and thus validating the true equality of e1 and e2.

**Q: What is hash-collision in Hashtable and how it is handled in Java?**

Ans: Two different keys with the same hash value. Two different entries will be kept in a single hash bucket to avoid the collision.

**Q: What is difference between HashSet and HashMap?**

Ans: HashSet contains only values whereas HashMap contains entry(key,value).

**Q: What is difference between HashMap and TreeMap?**

**Ans:** HashMap TreeMap

1) HashMap is can contain one null key. 1) TreeMap connot contain any null key.

2) HashMap maintains no order. 2) TreeMap maintains ascending order.

**Q:Can you make List,Set and Map elements synchronized?**

**Ans:** Yes, Collections class provides methods to make List,Set or Map elements as synchronized:

public static List synchronizedList(List l){}

public static Set synchronizedSet(Set s){}

public static SortedSet synchronizedSortedSet(SortedSet s){}

public static Map synchronizedMap(Map m){}

public static SortedMap synchronizedSortedMap(SortedMap m){}

**Q: What is the Dictionary class?**

**Ans:**

The Dictionary class provides the capability to store key-value pairs.

**Q. What is the Vector class?**

**Ans:** The Vector class provides the capability to implement a growable array of objects.

**Q. What is difference between Iterator and Enumeration?**

**Ans.** This is a beginner level collection interview questions and mostly asked during interviews of Junior Java developer up to experience of 2 to 3 years Iterator duplicate functionality of Enumeration with one addition of remove() method and both provide navigation functionally on objects of Collection.Another difference is that Iterator is more safe than Enumeration and doesn't allow another thread to modify collection object during iteration except remove() method and throws ConcurrentModificaitonException. See Iterator vs Enumeraiton in Java for more differences.

**Q. What is difference between Synchronized Collection and ConcurrentCollection?**

**Ans.** Java5 has added several new ConcurrentCollection classes e.g. ConcurrentHashMap, CopyOnWriteArrayList, BlockingQueue etc, which has made Interview questions on Java Collection even trickier. Java Also provided way to get Synchronized copy of collection e.g. ArrayList, HashMap by using Collections.synchronizedMap() Utility function.One Significant difference is that ConccurentCollections has better performance than synchronized Collection because they lock only a portion of Map to achieve concurrency and Synchronization. See Difference between SynchronizedCollection and ConcurrentCollection in Java for more details.

**Q. What is difference between fail-fast and fail-safe Iterators?**

**Ans.** This is relatively new collection interview questions and can become trick if you hear the term fail-fast and fail-safe first time. Fail-fast Iterators throws ConcurrentModificationException when one Thread is iterating over collection object and other thread structurally modify Collection either by adding, removing or modifying objects on underlying collection. They are called fail-fast because they try to immediately throw Exception when they encounter failure. On the other hand fail-safe Iterators works on copy of collection instead of original collection.

**Q Is it better to synchronize critical section of getInstance() method or whole getInstance() method ?**

**Ans.** Answer is critical section because if we lock whole method than every time someone call this method will have to wait even though we are not creating any object.

**Q: How do you traverse through a collection using its Iterator?**

**Ans.** To use an iterator to traverse through the contents of a collection, follow these steps:

* Obtain an iterator to the start of the collection by calling the collectionâ€™s iterator() method.
* Set up a loop that makes a call to hasNext(). Have the loop iterate as long as hasNext() returns true.
* Within the loop, obtain each element by calling next().

**Q.How do you remove elements during Iteration?**

**Ans.** Iterator also has a method remove() when remove is called, the current element in the iteration is deleted.

**Q. What is the difference between Enumeration and Iterator?**

|  |  |
| --- | --- |
| Enumeration | Iterator |
| Enumeration doesn't have a remove() method | Iterator has a remove() method |
| Enumeration acts as Read-only interface, because it has the methods only to traverse and fetch the objects | Can be abstract, final, native, static, or synchronized |

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

**Q. What are the main implementations of the List interface ?**

**Ans.** The main implementations of the List interface are as follows :

* ArrayList : Resizable-array implementation of the List interface. The best all-around implementation of the List interface.
* Vector : Synchronized resizable-array implementation of the List interface with additional "legacy methods."
* LinkedList : Doubly-linked list implementation of the List interface. May provide better performance than the ArrayList implementation if elements are frequently inserted or deleted within the list. Useful for queues and double-ended queues (deques).

**Q.How to obtain Array from an ArrayList ?**

**Ans.** Array can be obtained from an ArrayList using toArray() method on ArrayList.

List arrayList = new ArrayList();  
 arrayList.add(â€¦

ObjectÂ  a[] = arrayList.toArray();

**Q. Why insertion and deletion in ArrayList is slow compared to LinkedList ?**

* ArrayList internally uses and array to store the elements, when that array gets filled by inserting elements a new array of roughly 1.5 times the size of the original array is created and all the data of old array is copied to new array.
* During deletion, all elements present in the array after the deleted elements have to be moved one step back to fill the space created by deletion. In linked list data is stored in nodes that have reference to the previous node and the next node so adding element is simple as creating the node an updating the next pointer on the last node and the previous pointer on the new node. Deletion in linked list is fast because it involves only updating the next pointer in the node before the deleted node and updating the previous pointer in the node after the deleted node.

**Q. Why are Iterators returned by ArrayList called Fail Fast ?**

**Ans.** Because, if list is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove or add methods, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

**Q.What is the Set interface ?**

**Ans.**

* The Set interface provides methods for accessing the elements of a finite mathematical set
* Sets do not allow duplicate elements
* Contains no methods other than those inherited from Collection
* It adds the restriction that duplicate elements are prohibited
* Two Set objects are equal if they contain the same elements

**Q.What is a HashSet ?**

**Ans.**

* A HashSet is an unsorted, unordered Set.
* It uses the hashcode of the object being inserted (so the more efficient your hashcode() implementation the better access performance you’ll get).
* Use this class when you want a collection with no duplicates and you don’t care about order when you iterate through it.

**Q.What is a TreeSet ?**

**Ans.** TreeSet is a Set implementation that keeps the elements in sorted order. The elements are sorted according to the natural order of elements or by the comparator provided at creation time.

**Q.What is an EnumSet ?**

**Ans.**An EnumSet is a specialized set for use with enum types, all of the elements in the EnumSet type that is specified, explicitly or implicitly, when the set is created.

**Q.Difference between HashSet and TreeSet ?**

|  |  |
| --- | --- |
| **HashSet** | **TreeSet** |
| HashSet is under set interface i.e. it  does not guarantee for either sorted order or sequence order. | TreeSet is under set i.e. it provides elements in a sorted  order (acceding order). |
| We can add any type of elements to hash set. | We can add only similar types  of elements to tree set. |

**Q.What is a Map ?**

**Ans.**

* A map is an object that stores associations between keys and values (key/value pairs).
* Given a key, you can find its value. Both keys  and  values are objects.
* The keys must be unique, but the values may be duplicated.
* Some maps can accept a null key and null values, others cannot.

**Q.What are the main Implementations of the Map interface ?**

**Ans.** The main implementations of the List interface are as follows:

* HashMap
* HashTable
* TreeMap
* EnumMap

**Q.What is a TreeMap ?  
Ans.** TreeMap actually implements the SortedMap interface which extends the Map interface. In a TreeMap the data will be sorted in ascending order of keys according to the natural order for the key's class, or by the comparator provided at creation time. TreeMap is based on the Red-Black tree data structure.

**Q.How do you decide when to use HashMap and when to use TreeMap ?  
Ans.** For inserting, deleting, and locating elements in a Map, the HashMap offers the best alternative. If, however, you need to traverse the keys in a sorted order, then TreeMap is your better alternative. Depending upon the size of your collection, it may be faster to add elements to a HashMap, then convert the map to a TreeMap for sorted key traversal.

**Q.Difference between HashMap and Hashtable ?**

|  |  |
| --- | --- |
| **HashMap** | **Hashtable** |
| HashMap lets you have null values as well as one null key. | HashTable  does not allows null values as key and value. |
| The iterator in the HashMap is fail-safe (If you change the map while iterating, you’ll know). | The enumerator for the Hashtable is not fail-safe. |
| HashMap is unsynchronized. | Hashtable is synchronized. |

**Note:** Only one NULL is allowed as a key in HashMap. HashMap does not allow multiple keys to be NULL. Nevertheless, it can have multiple NULL values.

**Q.How does a Hashtable internally maintain the key-value pairs?**

**Ans.** TreeMap actually implements the SortedMap interface which extends the Map interface. In a TreeMap the data will be sorted in ascending order of keys according to the natural order for the key's class, or by the comparator provided at creation time. TreeMap is based on the Red-Black tree data structure.

**Q.What Are the different Collection Views That Maps Provide?**

**Ans.**

Maps Provide Three Collection Views.

* Key Set - allow a map's contents to be viewed as a set of keys.
* Values Collection - allow a map's contents to be viewed as a set of values.
* Entry Set - allow a map's contents to be viewed as a set of key-value mappings.

**Q.What is a KeySet View ?**

**Ans.** KeySet is a set returned by the keySet()method of the Map Interface, It is a set that contains all the keys present in the Map.

**Q.What is a Values Collection View ?**

**Ans.** Values Collection View is a collection returned by the values() method of the Map Interface, It contains all the objects present as values in the map.

**Q.What is an EntrySet View ?**

**Ans.** Entry Set view is a set that is returned by the entrySet() method in the map and contains Objects of type Map. Entry each of which has both Key and Value.

**Q.How do you sort an ArrayList (or any list) of user-defined objects ?  
Ans.** Create an implementation of the java.lang.Comparable interface that knows how to order your objects and pass it to java.util.Collections.sort(List, Comparator).

**Q.What is the Comparable interface ?**

**Ans.** The Comparable interface is used to sort collections and arrays of objects using theCollections.sort() and java.utils.Arrays.sort() methods respectively. The objects of the class implementing the Comparable interface can be ordered.

The Comparable interface in the generic form is written as follows:

interface Comparable<T>

where T is the name of the type parameter.  
  
All classes implementing the Comparable interface must implement the compareTo() method that has the return type as an integer. The signature of the compareTo() method is as follows:

int i = object1.compareTo(object2)

If object1 < object2: The value of i returned will be negative.

If object1 > object2: The value of i returned will be positive.

If object1 = object2: The value of i returned will be zero.

**Q.What are the differences between the Comparable and Comparator interfaces ?**

|  |  |
| --- | --- |
| **Comparable** | **Comparator** |
| It uses the compareTo() method.  int objectOne.compareTo(objectTwo). | t uses the compare() method.  int compare(ObjOne, ObjTwo) |
| It is necessary to modify the class whose instance is going to be sorted. | A separate class can be created in order to sort the instances. |
| Only one sort sequence can be created. | Many sort sequences can be created. |
| It is frequently used by the API classes. | It used by third-party classes to sort instances. |