Hashtable

Hashtable is basically a datastructure to retain values of key-value pair.

- It didn't allow null for both key and value. You will get NullPointerException if you add null value.
- It is synchronized. So it comes with its cost. Only one thread can access in one time

```
Hashtable<Integer,String> cityTable = new Hashtable<Integer,String
cityTable.put(1, "Lahore");
cityTable.put(2, "Karachi");
cityTable.put(3, null); /* NullPointerExcxeption at runtime*/

System.out.println(cityTable.get(1));
System.out.println(cityTable.get(2));
System.out.println(cityTable.get(3));
```

HashMap

Like Hashtable it also accepts key value pair.

- It allows null for both key and value
- It is unsynchronized. So come up with better performance

```
HashMap<Integer,String> productMap = new HashMap<Integer,String>();
productMap.put(1, "Keys");
productMap.put(2, null);
```

HashSet

HashSet does not allow duplicate values. It provides add method rather put method. You also use its contain method to check whether the object is already available in HashSet. HashSet can be used where you want to maintain a unique list.

```
HashSet<String> stateSet = new HashSet<String>();
stateSet.add ("CA");
stateSet.add ("WI");
stateSet.add ("NY");

if (stateSet.contains("PB")) /* if CA, it will not add but shows fol
     System.out.println("Already found");
else
    stateSet.add("PB");
```

Difference between TreeMap, TreeSet, ArrayList, HashMap, HashSet and TreeSet

HashSet:

- Class offers constant time performance for the basic operations (adds, remove, contains and size).
- it does not guarantee that the order of elements will remain constant over time
- Iteration performance depends on the 'initial capacity' and the 'load factor' of the HashSet.
    - It's quite safe to accept default load factor but you may want to specify an initial capacity that's about twice the size to which you expect the set to grow.
- **HashSet is much faster than TreeSet (constant-time versus log-time for most operations like add, remove and contains) but offers no ordering guarantees like TreeSet.**

TreeSet:

- guarantees log(n) time cost for the basic operations (add, remove and contains)
- guarantees that elements of set will be sorted [ascending, natural, or the one specified by you via it's constructor]
- doesn't offer any tuning parameters for iteration performance
- offers few handy methods to deal with the ordered set like first(), last(), headSet(), and tailSet() etc

Important points:

- Both guarantee duplicate-free collection of elements
- It is generally faster to add elements to the HasSet and then convert the collection to a TreeeSet for a duplicate-free sorted traversal.
- None of these implementation are synchronized. That is if multiple threads access a set concurrently, and at least one of the threads modifies the set, it must be synchronized externally.
- **LinkedHashSet** is in some sense intermediate between HashSet and TreeSet. Implemented as a hash table with a linked list running through it, however **it provides insertion-ordered iteration which is not same as sorted traversal guaranteed by TreeSet**

So choice of usage depends entirely on your needs but I feel that even if you need a ordered collection then you should still prefer HashSet to create the Set and then convert it into TreeSet.

- e.g. Set s = new TreeSet(hashSet);

NOTE:

- **TreeMap** lets you access the elements in your collection by key, or sequentially by key. It has considerably more overhead than **ArrayList** or **HashMap**.
- ***Use HashMap when you don't need sequential access, just lookup by key.***
- ***Use an ArrayList and use Arrays. Sort if you just want the elements in order.***
- ***TreeMap keeps the elements in order at all times. With ArrayList you just sort when you need to.***

1. **What is Java Collections API?**

   Java Collections framework API is a unified architecture for representing and manipulating collections. The API contains Interfaces, Implementations & Algorithm to help java programmer in everyday programming. In nutshell, this API does 6 things at high level

   - Reduces programming efforts. - Increases program speed and quality.
   - Allows interoperability among unrelated APIs.
   - Reduces effort to learn and to use new APIs.
   - Reduces effort to design new APIs.
   - Encourages & Fosters software reuse.

   To be specific, There are six collection java interfaces. The most basic interface is Collection. Three interfaces extend Collection: Set, List, and SortedSet. The other two collection interfaces, Map and SortedMap, do not extend Collection, as they represent mappings rather than true collections.

2. **What is an Iterator?**

   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.

3. **What is the difference between java.util.Iterator and java.util.ListIterator?**

   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.

4. **What is HashMap and Map?**

   Map is Interface which is part of Java collections framework. This is to store Key Value pair, and Hashmap is class that implements that using hashing technique.

5. **Difference between HashMap and HashTable? Compare Hashtable vs HashMap?**

   Both Hashtable & HashMap provide key-value access to data. The Hashtable is one of the original collection classes in Java (also called as legacy classes). HashMap is part of the new Collections Framework, added with Java 2, v1.2. There are several differences between HashMap and Hashtable in Java as listed below

   - The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls. (HashMap allows null values as key and value whereas Hashtable doesn’t allow nulls).
   - 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.
   - HashMap is non synchronized whereas Hashtable is synchronized.
   - Iterator in the HashMap is fail-fast while the enumerator for the Hashtable isn't. So this could be a design consideration.

6. **What does synchronized means in Hashtable context?**

Synchronized means only one thread can modify a hash table at one point of time. 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.

7. **What is fail-fast property?**

   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.

8. **Why doesn't Collection extend Cloneable and Serializable?**

   From Sun FAQ Page: 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 cause 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.

9. **How can we make Hashmap synchronized?**

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

10. **Where will you use Hashtable and where will you use HashMap?**

    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.

11. **Difference between Vector and ArrayList? What is the Vector class?**

    Vector & ArrayList both classes are implemented using dynamically resizable arrays, providing fast random access and fast traversal. ArrayList and Vector class both implement the List interface. Both the classes are member of Java collection framework, therefore from an API perspective, these two classes are very similar. However, there are still some major differences between the two. Below are some key differences

    - Vector is a legacy class which has been retrofitted to implement the List interface since Java 2 platform v1.2
    - Vector is synchronized whereas ArrayList is not. Even though Vector class is synchronized, still when you want programs to run in multithreading environment using ArrayList with Collections.synchronizedList() is recommended over Vector.
    - ArrayList has no default size while vector has a default size of 10.
    - The Enumerations returned by Vector's elements method are not fail-fast. Whereas ArraayList does not have any method returning Enumerations.

12. **What is the Difference between Enumeration and Iterator interface?**

    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:

    1. Enumeration contains 2 methods namely hasMoreElements() & nextElement() whereas Iterator contains three methods namely hasNext(), next(),remove().
    2. 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.
    3. 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.

13. **Why Java Vector class is considered obsolete or unofficially deprecated? or Why should I always use ArrayList over Vector?**

    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 syncronization - 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)

14. **What is an enumeration?**
    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.
15. **What is the difference between Enumeration and Iterator?**
    The functionality of Enumeration interface is duplicated by the Iterator interface. Iterator has a remove() method while Enumeration doesn't. Enumeration acts as Read-only interface, because it has the methods only to traverse and fetch the objects, where as using Iterator we can manipulate the objects also like adding and removing the objects. So Enumeration is used when ever we want to make Collection objects as Read-only.
16. **Where will you use Vector and where will you use ArrayList?**
    The basic difference between a Vector and an ArrayList is that, vector is synchronized while ArrayList is not. Thus whenever there is a possibility of multiple threads accessing the same instance, one should use Vector. While if not multiple threads are going to access the same instance then use ArrayList. Non synchronized data structure will give better performance than the synchronized one.
17. **What is the importance of hashCode() and equals() methods? How they are used in Java?**
    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().

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

    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.

19. **What is java.util.concurrent BlockingQueue? How it can be used?**
    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.
20. **Set & List interface extend Collection, so Why doesn't Map interface extend Collection?**
    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.
21. **Which implementation of the List interface provides for the fastest insertion of a new element into the middle of the list?**
    a. Vector b. ArrayList c. LinkedList ArrayList and Vector both use an array to store the elements of the list. When an element is inserted into the middle of the list the elements that follow the insertion point must be shifted to make room for the new element. The LinkedList is implemented using a doubly linked list; an insertion requires only the updating of the links at the point of insertion. Therefore, the LinkedList allows for fast insertions and deletions.
22. **What is the difference between ArrayList and LinkedList? (ArrayList vs LinkedList.)**
    java.util.ArrayList and java.util.LinkedList are two Collections classes used for storing lists of object references **Here are some key differences:**

- 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.
- 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.
- 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.
- 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.
- 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.

23. **Where will you use ArrayList and Where will you use LinkedList? Or Which one to use when (ArrayList / LinkedList).**

    Below is a snippet from SUN's site. The Java SDK contains 2 implementations of the List interface - ArrayList and LinkedList. If you frequently add elements to the beginning of the List or iterate over the List to delete elements from its interior, you should consider using LinkedList. These operations require constant-time in a LinkedList and linear-time in an ArrayList. But you pay a big price in performance. Positional access requires linear-time in a LinkedList and constant-time in an ArrayList.

24. **What is performance of various Java collection implementations/algorithms? What is Big 'O' notation for each of them ?**

    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.