[What is Java Collections API?](file:///D:\\collecxtion\\1st.htm" \l "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.

## [What is an Iterator?](file:///D:\\collecxtion\\1st.htm" \l "What-is-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. Generally it is not advisable to modify the collection itself while traversing an Iterator.

## [What is the difference between java.util.Iterator and java.util.ListIterator?](file:///D:\\collecxtion\\1st.htm" \l "Iterator-vs-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.

Difference between HashMap and HashTable?

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.

## [What does synchronized means in Hashtable context?](file:///D:\\collecxtion\\1st.htm" \l "synchronized-Hashtable)

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.

## [How can we make Hashmap synchronized?](file:///D:\\collecxtion\\1st.htm" \l "Hashmap-synchronized)

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

[Where will you use Hashtable and where will you use HashMap?](file:///D:\\collecxtion\\1st.htm" \l "Hashtable-vs-Hashmap-use)

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.

[Difference between Vector and ArrayList? What is the Vector class?](file:///D:\\collecxtion\\1st.htm" \l "Vector-vs-ArrayList)

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.

[What is the Difference between Enumeration and Iterator interface?](file:///D:\\collecxtion\\1st.htm" \l "Difference-between-Enumeration-Iterator)

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.

[Why Java Vector class is considered obsolete or unofficially deprecated? or Why should I always use ArrayList over Vector?](file:///D:\\collecxtion\\1st.htm" \l "Why-vector-class-deprecated)

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](http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6201870))

[What is an enumeration?](file:///D:\\collecxtion\\1st.htm" \l "What-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.

## [Set & List interface extend Collection, so Why doesn't Map interface extend Collection?](file:///D:\\collecxtion\\1st.htm" \l "17)

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.

[Which implementation of the List interface provides for the fastest insertion of a new element into the middle of the list?](file:///D:\\collecxtion\\1st.htm" \l "18)

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.

[What is the difference between ArrayList and LinkedList? (ArrayList vs LinkedList.)](file:///D:\\collecxtion\\1st.htm" \l "19)

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.

[Where will you use ArrayList and Where will you use LinkedList? Or Which one to use when (ArrayList / LinkedList).](file:///D:\\collecxtion\\1st.htm" \l "20)

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.

What are the advantages of ArrayList over arrays ?

Some of the advantages ArrayList has over arrays are:

* It can grow dynamically
* It provides more powerful insertion and search mechanisms than arrays.

How to obtain Array from an ArrayList ?

* Array can be obtained from an ArrayList using ***toArray()*** method on ArrayList.
* List arrayList = new ArrayList();  
   arrayList.add(â€¦
* ObjectÂ  a[] = **arrayList.toArray()**;

Why are Iterators returned by ArrayList called Fail Fast ?

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.

 What is the Set interface?

The Set interface provides methods for accessing the elements of a finite mathematical set. Sets do not allow duplicate elements.

What is the List interface?

The List interface provides support for ordered collections of objects.

What is the difference between set and list?

**Set** stores elements in an unordered way but does not contain duplicate elements, where as **List** stores elements in an ordered way but may contain duplicate elements.

What is map interface in a java?

**Map** is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value. This Map permits null value

What is a vector in Java?

**Vector**implements a dynamic array. It is similar to ArrayList, but with two differences:  Vector is synchronized, and it contains many legacy methods that are not part of the collections framework.

What is the Properties class?

The properties class is a subclass of Hashtable that can be read from or written to a stream. It also provides the capability to specify a set of default values to be used.

What is difference between HashMap and HashSet?

**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.

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

What is the Difference between the Iterator  and ListIterator?

**Iterator** : Iterator takes the place of Enumeration in the Java collections framework. One can traverse throughr the the collection with the help of iterator in forward direction only and Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics

**ListIterator:** An iterator for lists that allows one to traverse the list in either direction.modify the list during iteration, and obtain the iterator's current position in the list. A ListIterator has no current element. its cursor position always lies between the element that would be returned by a call to previous() and the element that would be returned by a call to next(). In a list of length n, there are n+1 valid index values, from 0 to n, inclusive.

What is difference between Synchronized Collection andConcurrentCollection?

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.

When do you use ConcurrentHashMap in Java?

This is another advanced level collection interview questions in Java which normally asked to check whether interviewer is familiar with optimization done on ConcurrentHashMap or not. ConcurrentHashMap is better suited for situation where you have multiple readers and one

Writer or fewer writers since Map gets locked only during write operation. If you have equaled number of reader and writer than [ConcurrentHashMap](http://javarevisited.blogspot.com/2011/04/difference-between-concurrenthashmap.html) will perform in line of hashtable or synchronized hashMap.

### Why Collection doesn’t extend Cloneable and Serializable interfaces?

Collection interface specifies group of Objects known as elements. How the elements are maintained is left up to the concrete implementations of Collection. For example, some Collection implementations like List allow duplicate elements whereas other implementations like Set don’t. A lot of the Collection implementations have a public clone method. However, it does’t really make sense to include it in all implementations of Collection. This is because Collection is an abstract representation. What matters is the implementation.  
The semantics and the implications of either cloning or serializing come into play when dealing with the actual implementation; so concrete implementation should decide how it should be cloned or serialized, or even if it can be cloned or serialized.  
So mandating cloning and serialization in all implementations is actually less flexible and more restrictive. The specific implementation should make the decision as to whether it can be cloned or serialized.

### Why Map interface doesn’t extend Collection interface?

Although Map interface and it’s implementations are part of Collections Framework, Map are not collections and collections are not Map. Hence it doesn’t make sense for Map to extend Collection or vice versa.  
If Map extends Collection interface, then where are the elements? Map contains key-value pairs and it provides methods to retrieve list of Keys or values as Collection but it doesn’t fit into the “group of elements” paradigm.

### Why there is not method like Iterator.add() to add elements to the collection?

The semantics are unclear, given that the contract for Iterator makes no guarantees about the order of iteration. Note, however, that ListIterator does provide an add operation, as it does guarantee the order of the iteration.

### How to avoid ConcurrentModificationException while iterating a collection?

We can use concurrent collection classes to avoid ConcurrentModificationException while iterating over a collection, for example CopyOnWriteArrayList instead of ArrayList.

### How HashMap works in Java?

HashMap stores key-value pair in Map.Entry static nested class implementation. HashMap works on hashing algorithm and uses hashCode() and equals() method in put and getmethods.When we call putmethod by passing key-value pair, HashMap uses Key hashCode() with hashing to find out the index to store the key-value pair. The Entry is stored in the LinkedList, so if there are already existing entry, it uses equals() method to check if the passed key already exists, if yes it overwrites the value else it creates a new entry and store this key-value Entry.When we call get method by passing Key, again it uses the hashCode() to find the index in the array and then use equals() method to find the correct Entry and return it’s value.

The other important things to know about HashMap are capacity, load factor, threshold resizing. HashMap initial default capacity is 32 and load factor is 0.75. Threshold is capacity multiplied by load factor and whenever we try to add an entry, if map size is greater than threshold, HashMap rehashes the contents of map into a new array with a larger capacity. The capacity is always power of 2, so if you know that you need to store a large number of key-value pairs, for example in caching data from database, it’s good idea to initialize the HashMap with correct capacity and load factor.

### What is Collections Class?

java.util.Collectionsis a utility class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, “wrappers”, which return a new collection backed by a specified collection, and a few other odds and ends.This class contains methods for collection framework algorithms, such as binary search, sorting, shuffling, reverse etc.

### What is Comparable and Comparator interface?

Java provides Comparable interface which should be implemented by any custom class if we want to use Arrays or Collections sorting methods. Comparable interface has compareTo(T obj) method which is used by sorting methods. We should override this method in such a way that it returns a negative integer, zero, or a positive integer if “this” object is less than, equal to, or greater than the object passed as argument.But, in most real life scenarios, we want sorting based on different parameters. For example, as a CEO, I would like to sort the employees based on Salary, an HR would like to sort them based on the age. This is the situation where we need to use Comparator interface because Comparable.compareTo(Object o)method implementation can sort based on one field only and we can’t chose the field on which we want to sort the Object.Comparator interface compare(Object o1, Object o2) method need to be implemented that takes two Object argument, it should be implemented in such a way that it returns negative int if first argument is less than the second one and returns zero if they are equal and positive int if first argument is greater than second one.

### What is difference between Comparable and Comparator interface?

Comparable and Comparator interfaces are used to sort collection or array of objects.Comparable interface is used to provide the natural sorting of objects and we can use it to provide sorting based on single logic.  
Comparator interface is used to provide different algorithms for sorting and we can chose the comparator we want to use to sort the given collection of objects.