Java Collections Interview Questions Answers

- **What is Java Collections Framework? List out some benefits of Collections framework?**

Collections are used in every programming language and initial java release contained few classes for collections: **Vector**, **Stack**, **Hashtable**, **Array**. But looking at the larger scope and usage, Java 1.2 came up with Collections Framework that group all the collections interfaces, implementations and algorithms. Java Collections have come through a long way with usage of Generics and Concurrent Collection classes for thread-safe operations. It also includes blocking interfaces and their implementations in java concurrent package. Some of the benefits of collections framework are:

- Reduced development effort by using core collection classes rather than implementing our own collection classes.
- Code quality is enhanced with the use of well tested collections framework classes.
- Reduced effort for code maintenance by using collection classes shipped with JDK.
- Reusability and Interoperability

- **What is the benefit of Generics in Collections Framework?**

Java 1.5 came with Generics and all collection interfaces and implementations use it heavily. Generics allow us to provide the type of Object that a collection can contain, so if you try to add any element of other type it throws compile time error. This avoids ClassCastException at Runtime because you will get the error at compilation. Also Generics make code clean since we don't need to use casting and instanceof operator. It also adds up to runtime benefit because the bytecode instructions that do type checking are not generated.

- **What are the basic interfaces of Java Collections Framework?**

Collection is the root of the collection hierarchy. A collection represents a group of objects known as its elements. The Java platform doesn't provide any direct implementations of this interface.

Set is a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the deck of cards.

List is an ordered collection and can contain duplicate elements. You can access any element from it's index. List is more like array with dynamic length.

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

Some other interfaces are `Queue`, `Dequeue`, `Iterator`, `SortedSet`, `SortedMap` and `ListIterator`.

- **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.

- **What is an Iterator?**

Iterator interface provides methods to iterate over any Collection. We can get iterator instance from a Collection using iterator method. Iterator takes the place of Enumeration in the Java Collections Framework. Iterators allow the caller to remove elements from the underlying collection during the iteration.

- **What is difference between Enumeration and Iterator interface?**

Enumeration is twice as fast as Iterator and uses very less memory. Enumeration is very basic and fits to basic needs. But Iterator is much safer as compared to Enumeration because it always denies other threads to modify the collection object which is being iterated by it.
Iterator takes the place of Enumeration in the Java Collections Framework. Iterators allow the caller to remove elements from the underlying collection that is not possible with Enumeration. Iterator method names have been improved to make it’s functionality clear.

- **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.

- **Why Iterator don’t have a method to get next element directly without moving the cursor?**

It can be implemented on top of current Iterator interface but since it’s use will be rare, it doesn’t make sense to include it in the interface that everyone has to implement.

- **What is different between Iterator and ListIterator?**

- We can use Iterator to traverse Set and List collections whereas ListIterator can be used with Lists only.
- Iterator can traverse in forward direction only whereas ListIterator can be used to traverse in both the directions.
- ListIterator inherits from Iterator interface and comes with extra functionalities like adding an element, replacing an element, getting index position for previous and next elements.

- **What are different ways to iterate over a list?**

We can iterate over a list in two different ways – using iterator and using for-each loop.

```
01 List<String> strList = new ArrayList<>();
02 //using for-each loop
03 for(String obj : strList){
04     System.out.println(obj);
05 }
06 //using iterator
07 Iterator<String> it = strList.iterator();
08 while(it.hasNext()){
09     String obj = it.next();
10     System.out.println(obj);
11 }
```

Using iterator is more thread-safe because it makes sure that if underlying list elements are modified, it will throw `ConcurrentModificationException`.

- **What do you understand by iterator fail-fast property?**

Iterator fail-fast property checks for any modification in the structure of the underlying collection everytime we try to get the next element. If there are any modifications found, it throws `ConcurrentModificationException`. All the implementations of Iterator in Collection classes are fail-fast by design except the concurrent collection classes like ConcurrentHashMap and CopyOnWriteArrayList.

- **What is difference between fail-fast and fail-safe?**

Iterator fail-safe property work with the clone of underlying collection, hence it's not affected by any modification in the collection. By design, all the collection classes in `java.util` package are fail-fast whereas collection classes in `java.util.concurrent` are fail-safe. Fail-fast iterators throw ConcurrentModificationException whereas fail-safe iterator never throws ConcurrentModificationException. Check this post for CopyOnWriteArrayList Example.

- **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.
Check this post for ConcurrentHashMap Example.

- **Why there are no concrete implementations of Iterator interface?**

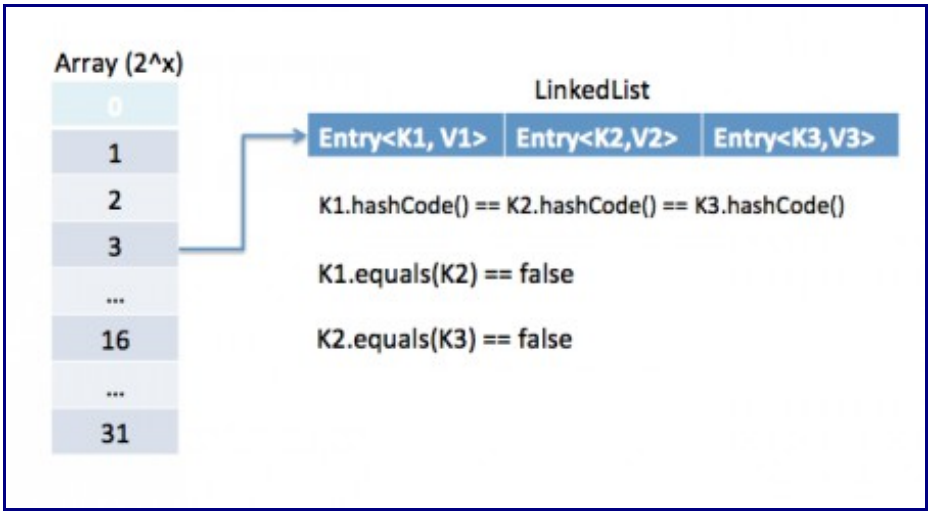
Iterator interface declare methods for iterating a collection but it's implementation is responsibility of the Collection implementation classes. Every collection class that returns an iterator for traversing has it's own Iterator implementation nested class.
This allows collection classes to chose whether iterator is fail-fast or fail-safe. For example ArrayList iterator is fail-fast whereas CopyOnWriteArrayList iterator is fail-safe.

- **What is UnsupportedOperationException?**

`UnsupportedOperationException` is the exception used to indicate that the operation is not supported. It's used extensively in JDK classes, in collections framework `java.util.Collections.UnmodifiableCollection` throws this exception for all `add` and `remove` operations.

- **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 `get`methods.When we call `put`method 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. Below image will explain these detail clearly.



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 the importance of hashCode() and equals() methods?**

HashMap uses Key object hashCode() and equals() method to determine the index to put the key-value pair. These methods are also used when we try to get value from HashMap. If these methods are not implemented correctly, two different Key's might produce same hashCode() and equals() output and in that case rather than storing it at different location, HashMap will consider them same and overwrite them.Similarly all the collection classes that doesn't store duplicate data use hashCode() and equals() to find duplicates, so it's very important to implement them correctly. The implementation of equals() and hashCode() should follow these rules.

- If `o1.equals(o2)`, then `o1.hashCode() == o2.hashCode()`should always be `true`.
- If `o1.hashCode() == o2.hashCode` is true, it doesn't mean that `o1.equals(o2)` will be `true`.

Can we use any class as Map key?

We can use any class as Map Key, however following points should be considered before using them.

- If the class overrides equals() method, it should also override hashCode() method.
- The class should follow the rules associated with equals() and hashCode() for all instances. Please refer earlier question for these rules.
- If a class field is not used in equals(), you should not use it in hashCode() method.
- Best practice for user defined key class is to make it immutable, so that hashCode() value can be cached for fast performance. Also immutable classes make sure that hashCode() and equals() will not change in future that will solve any issue with mutability.
  For example, let's say I have a class `MyKey` that I am using for HashMap key.

```
01 //MyKey name argument passed is used for equals() and hashCode()
02 MyKey key = new MyKey('Pankaj'); //assume hashCode=1234
03 myHashMap.put(key, 'Value');
04
05 // Below code will change the key hashCode() and equals()
06 // but it's location is not changed.
07 key.setName('Amit'); //assume new hashCode=7890
08
09 //below will return null, because HashMap will try to look for key
10 //in the same index as it was stored but since key is mutated,
11 //there will be no match and it will return null.
12 myHashMap.get(new MyKey('Pankaj'));
```

  This is the reason why String and Integer are mostly used as HashMap keys.

- **What are different Collection views provided by Map interface?**

Map interface provides three collection views:

- **Set** keySet(): Returns a Set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator's own remove operation), the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Set.remove, removeAll, retainAll, and clear operations. It does not support the add or addAll operations.
- **Collection** values(): Returns a Collection view of the values contained in this map. The collection is backed by the map, so changes to the map are reflected in the collection, and vice-versa. If the map is modified while an iteration over the collection is in progress (except through the iterator's own remove operation), the results of the iteration are undefined. The collection supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Collection.remove, removeAll, retainAll and clear operations. It does not support the add or addAll operations.
- **Set<Map.Entry<K, V>> entrySet()**: Returns a Set view of the mappings contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator's own remove operation, or through the setValue operation on a map entry returned by the iterator) the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Set.remove, removeAll, retainAll and clear operations. It does not support the add or addAll operations.

- **What is difference between HashMap and Hashtable?**

HashMap and Hashtable both implements Map interface and looks similar, however there are following difference between HashMap and Hashtable.

- HashMap allows null key and values whereas Hashtable doesn't allow null key and values.
- Hashtable is synchronized but HashMap is not synchronized. So HashMap is better for single threaded environment, Hashtable is suitable for multi-threaded environment.
- `LinkedHashMap` was introduced in Java 1.4 as a subclass of HashMap, so incase you want iteration order, you can easily switch from HashMap to LinkedHashMap but that is not the case with Hashtable whose iteration order is unpredictable.
- HashMap provides Set of keys to iterate and hence it's fail-fast but Hashtable provides Enumeration of keys that doesn't support this feature.
- Hashtable is considered to be legacy class and if you are looking for modifications of Map while iterating, you should use ConcurrentHashMap.

- **How to decide between HashMap and TreeMap?**

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.

- **What are similarities and difference between ArrayList and Vector?**

ArrayList and Vector are similar classes in many ways.

- Both are index based and backed up by an array internally.
- Both maintains the order of insertion and we can get the elements in the order of insertion.
- The iterator implementations of ArrayList and Vector both are fail-fast by design.
- ArrayList and Vector both allows null values and random access to element using index number.

These are the differences between ArrayList and Vector.

- Vector is synchronized whereas ArrayList is not synchronized. However if you are looking for modification of list while iterating, you should use CopyOnWriteArrayList.
- ArrayList is faster than Vector because it doesn't have any overhead because of synchronization.
- ArrayList is more versatile because we can get synchronized list or read-only list from it easily using Collections utility class.

- **What is difference between Array and ArrayList? When will you use Array over ArrayList?**

Arrays can contain primitive or Objects whereas ArrayList can contain only Objects.
Arrays are fixed size whereas ArrayList size is dynamic.
Arrays doesn't provide a lot of features like ArrayList, such as addAll, removeAll, iterator etc.Although ArrayList is the obvious choice when we work on list, there are few times when array are good to use.

- If the size of list is fixed and mostly used to store and traverse them.
- For list of primitive data types, although Collections use autoboxing to reduce the coding effort but still it makes them slow when working on fixed size primitive data types.
- If you are working on fixed multi-dimensional situation, using [][] is far more easier than List<List<>>

- **What is difference between ArrayList and LinkedList?**

ArrayList and LinkedList both implement List interface but there are some differences between them.

- ArrayList is an index based data structure backed by Array, so it provides random access to it's elements with performance as O(1) but LinkedList stores data as list of nodes where every node is linked to it's previous and next node. So even though there is a method to get the element using index, internally it traverse from start to reach at the index node and then return the element, so performance is O(n) that is slower than ArrayList.
- Insertion, addition or removal of an element is faster in LinkedList compared to ArrayList because there is no concept of resizing array or updating index when element is added in middle.
- LinkedList consumes more memory than ArrayList because every node in LinkedList stores reference of previous and next elements.

- **Which collection classes provide random access of it's elements?**

ArrayList, HashMap, TreeMap, Hashtable classes provide random access to it's elements. Download java collections pdf for more information.

- **What is EnumSet?**

`java.util.EnumSet`is Set implementation to use with enum types. All of the elements in an enum set must come from a single enum type that is specified, explicitly or implicitly, when the set is created. EnumSet is not synchronized and null elements are not allowed. It also provides some useful methods like copyOf(Collection c), of(E first, E… rest) and complementOf(EnumSet s).Check this post for java enum tutorial.

- **Which collection classes are thread-safe?**

Vector, Hashtable, Properties and Stack are synchronized classes, so they are thread-safe and can be used in multi-threaded environment. Java 1.5 Concurrent API included some collection classes that allows modification of collection while iteration because they work on the clone of the collection, so they are safe to use in multi-threaded environment.

- **What are concurrent Collection Classes?**

Java 1.5 Concurrent package (`java.util.concurrent`) contains thread-safe collection classes that allow collections to be modified while iterating. By design iterator is fail-fast and throws ConcurrentModificationException. Some of these classes are `CopyOnWriteArrayList`, `ConcurrentHashMap`, `CopyOnWriteArraySet`.Read these posts to learn about them in more detail.

- Avoid ConcurrentModificationException
- CopyOnWriteArrayList Example
- HashMap vs ConcurrentHashMap

- **What is BlockingQueue?**

`java.util.concurrent.BlockingQueue`is a Queue that supports operations that wait for the queue to become non-empty when retrieving and removing an element, and wait for space to become available in the queue when adding an element.BlockingQueue interface is part of java collections framework and it's primarily used for implementing producer consumer problem. We don't need to worry about waiting for the space to be available for producer or object to be available for consumer in BlockingQueue as it's handled by implementation classes of BlockingQueue.Java provides several BlockingQueue implementations such as ArrayBlockingQueue, LinkedBlockingQueue, PriorityBlockingQueue, SynchronousQueue etc.
Check this post for use of BlockingQueue for producer-consumer problem.

- **What is Queue and Stack, list their differences?**

Both Queue and Stack are used to store data before processing them. `java.util.Queue` is an interface whose implementation classes are present in java concurrent package. Queue allows retrieval of element in First-In-First-Out (FIFO) order but it's not always the case. There is also Deque interface that allows elements to be retrieved from both end of the queue.
Stack is similar to queue except that it allows elements to be retrieved in Last-In-First-Out (LIFO) order.
Stack is a class that extends Vector whereas Queue is an interface.

- **What is Collections Class?**

`java.util.Collections`is 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.

Check this post for use of Comparable and Comparator interface to sort objects.

- **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.

- **How can we sort a list of Objects?**

If we need to sort an array of Objects, we can use `Arrays.sort()`. If we need to sort a list of objects, we can use `Collections.sort()`. Both these classes have overloaded sort() methods for natural sorting (using Comparable) or sorting based on criteria (using Comparator). Collections internally uses Arrays sorting method, so both of them have same performance except that Collections take sometime to convert list to array.

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

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

- **How can we create a synchronized collection from given collection?**

We can use `Collections.synchronizedCollection(Collection c)` to get a synchronized (thread-safe) collection backed by the specified collection.

- **What are common algorithms implemented in Collections Framework?**

Java Collections Framework provides algorithm implementations that are commonly used such as sorting and searching. Collections class contain these method implementations. Most of these algorithms work on List but some of them are applicable for all kinds of collections. Some of them are sorting, searching, shuffling, min-max values.

- **What is Big-O notation? Give some examples?**

The Big-O notation describes the performance of an algorithm in terms of number of elements in a data structure. Since Collection classes are actually data structures, we usually tend to use Big-O notation to chose the collection implementation to use based on time, memory and performance.Example 1: ArrayList `get(index i)` is a constant-time operation and doesn't depend on the number of elements in the list. So it's performance in Big-O notation is O(1).
Example 2: A linear search on array or list performance is O(n) because we need to search through entire list of elements to find the element.

- **What are best practices related to Java Collections Framework?**

- Chosing the right type of collection based on the need, for example if size is fixed, we might want to use Array over ArrayList. If we have to iterate over the Map in order of insertion, we need to use TreeMap. If we don't want duplicates, we should use Set.
- Some collection classes allows to specify the initial capacity, so if we have an estimate of number of elements we will store, we can use it to avoid rehashing or resizing.
- Write program in terms of interfaces not implementations, it allows us to change the implementation easily at later point of time.
- Always use Generics for type-safety and avoid ClassCastException at runtime.
- Use immutable classes provided by JDK as key in Map to avoid implementation of hashCode() and equals() for our custom class.
- Use Collections utility class as much as possible for algorithms or to get read-only, synchronized or empty collections rather than writing own implementation. It will enhance code-reuse with greater stability and low maintainability.

I will keep on adding more questions on java collections framework as and when I found them, if you found it useful please share it with others too, it motivates me in writing more like these.

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.

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.

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.

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.

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.

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.

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.

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.

How can we make Hashmap synchronized?

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

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.

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.

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.

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)

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.

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.

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.

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().

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.

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.

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.

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.

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.

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.

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.

Can you think of a questions which is not part of this post? Please don't forget to share it with me in comments section & I will try to include it in the list.