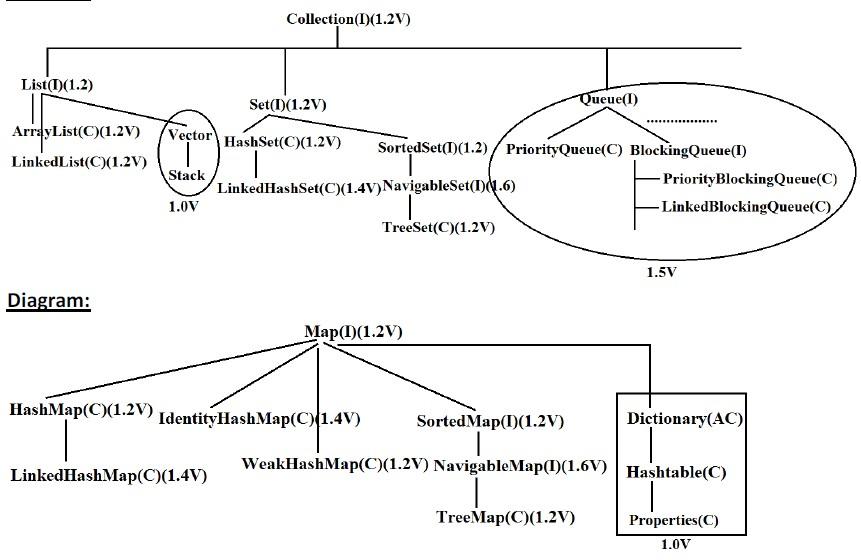
# Explain Collections hierarchy?

**What are Collection related features in Java 8?**

Java 8 has brought major changes in the Collection API. Some of the changes are:

1. Java Stream API for collection classes for supporting sequential as well as parallel processing

2. Iterable interface is extended with forEach() default method that we can use to iterate over a collection. It is very helpful when used with lambda expressions because it’s argument Consumer is a function interface.

3. Miscellaneous Collection API improvements such as forEachRemaining(Consumer action)method in Iterator interface, Map replaceAll(), compute(), merge() methods.

**What is the Java Collections API? List down its advantages?**

Collections are used in every programming language and the 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 the Collections Framework that groups 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 packages.

Some of the benefits of collections framework are;

**Benefits of Collections Framework:**

1. Improves program quality and speed

2. Increases the chances of reusability of software

3. Decreases programming effort.

**What is the 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.

# [What is the difference between Set and List?](http://stackoverflow.com/questions/1035008/what-is-the-difference-between-set-and-list)

|  |  |
| --- | --- |
| **List** | **Set** |
| Ordered Collection | Unordered Collection |
| Duplicate Allowed | Duplicate Not Allowed |
| List allows any number of null values | Set can have only a single null value at most. |
| List interface has one legacy class called [Vector](http://beginnersbook.com/2013/12/vector-in-java/). | Set interface does not have any legacy class. |
| [ListIterator](http://beginnersbook.com/2014/06/listiterator-in-java-with-examples/) can be used to traverse a List in both the directions(forward and backward) | [ListIterator](http://beginnersbook.com/2014/06/listiterator-in-java-with-examples/) cannot be used to traverse a Set. |

**Difference between List, Set and Map?**

|  |  |  |
| --- | --- | --- |
| **List** | **Set** | **Map** |
| List allows duplicate elements. Any number of duplicate elements can be inserted into the list without affecting the same existing values and their indexes. | Set doesn’t allow duplicates. Set and all of the classes which implements Set interface should have unique elements. | Map stored the elements as key & value pair. Map doesn’t allow duplicate keys while it allows duplicate values. |
| List allows any number of null values. | Set allows single null value at most. | Map can have single null key at most and any number of null values. |
| List: ArrayList, LinkedList etc. | Set: HashSet, LinkedHashSet, TreeSet, SortedSet etc. | Map: HashMap, TreeMap, WeakHashMap, LinkedHashMap, IdentityHashMap etc. |
| List and all of its implementation classes maintains the insertion order. | Set doesn’t maintain any order; still few of its classes sort the elements in an order such as LinkedHashSet maintains the elements in insertion order. | Map also doesn’t stores the elements in an order,  however few of its classes does the same. For e.g. TreeMap sorts the map in the ascending order of keys and LinkedHashMap sorts the elements in the insertion order, the order in which the elements got added to the LinkedHashMap. |

**Difference between Array and ArrayList in Java.**

|  |  |
| --- | --- |
| **Array** | **ArrayList** |
| Stores primitive data types and also objects | Stores only objects |
| Defined in Java language itself as a fundamental data structure | Belongs to collections framework |
| Fixed size | Growable and resizable. Elements can be added or removed |
| Stores similar data of one type | Can store heterogeneous data types |
| It is not a class | It is a class with many methods |
| Cannot be synchronized | Can be obtained a synchronized version |
| Elements retrieved with for loop | Can be retrieved with for loop and iterators |
| Elements accessible with index number | Accessing methods like get() etc. are available |

# Difference between ArrayList and LinkedList in Java.

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| ArrayList is better for storing and accessing data. | LinkedList is better for manipulating data. |
| Insertion, Deletion Slow | Insertion, Deletion Fast |
| Take Less memory | Take more memory |
| ArrayList internally uses dynamic array to store the elements. | LinkedList internally uses doubly linked list to store the elements. |

**Difference between Arraylist and Vector.**

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| ArrayList is not synchronized. | Vector is synchronized. |
| ArrayList increments 50% of current array size if number of element exceeds from its capacity. | Vector increments 100% means doubles the array size if total number of element exceeds than its capacity. |
| ArrayList is not a legacy class; it is introduced in JDK 1.2. | Vector is a legacy class. |
| ArrayList is fast because it is non-synchronized. | Vector is slow because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object. |
| ArrayList uses Iterator interface to traverse the elements. | Vector uses Enumeration interface to traverse the elements. But it can use Iterator also. |

[**Difference between enumeration and iterator**](http://www.instanceofjava.com/2014/12/difference-between-enumeration-and.html)**.**

|  |  |
| --- | --- |
| **Enumeration** | **Iterator** |
| Enumeration interface implemented in java 1.0 version. So Enumeration is legacy interface. | Iterator is implemented on all Java collection classes. |
| Enumeration uses elements() method. | Iterator uses iterator() method. |
| Enumeration can traverse in forward direction only. | Iterator can traverse in forward direction only. |
| Enumeration having methods like hasMoreElement(), nextElement(). | Iterator having methods like hasNext(), next(), remove(). |

**Difference between Iterator and ListIterator.**

|  |  |
| --- | --- |
| **Iterator** | **ListIterator** |
| Iterator is used for traversing List and Set both. | We can use ListIterator to traverse List only |
| We can traverse in only forward direction using Iterator | Using ListIterator, we can traverse a List in both the directions (forward and Backward). |
| We cannot obtain indexes while using Iterator | We can obtain indexes at any point of time while traversing a list using ListIterator. The methods nextIndex() and previousIndex() are used for this purpose. |
| We cannot add element to collection while traversing it using Iterator, it throws ConcurrentModificationException when you try to do it. | We can add element at any point of time while traversing a list using ListIterator. |
| We cannot replace the existing element value when using Iterator. | By using set(E e) method of ListIterator we can replace the last element returned by next() or previous() methods. |
| Methods of Iterator:  hasNext(),next(),remove() | Methods of ListIterator:  add(E e),hasNext(),hasPrevious(),next()  nextIndex(),previous(),previousIndex(),remove(),set(E e) |

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

|  |  |
| --- | --- |
| **HashSet** | **TreeSet** |
| You can Store any type of data. | Similar Type of Object will be stored. |
| HashSet does not maintain any order of elements. | TreeSet elements are sorted in ascending order by default. |
| HashSet can store null object. | TreeSet does not allow null object. If one try to store null object in TreeSet object, it will throw Null Pointer Exception. |
| HashSet is much faster | TreeSet is much Slower. |
| only add() and remove() methods exist | Apart add and remove methods, also exist overloaded subSet(), headSet() and tailSet() |
| Due to higher performance, it should be preferred (when sorting is not much needed) | Should be preferred only when sorting order is required |

**Difference between HashMap and Hashtable.**

|  |  |
| --- | --- |
| **HashMap** | **Hashtable** |
| HashMap is non synchronized. It is not-thread safe and can't be shared between many threads without proper synchronization code. | Hashtable is synchronized. It is thread-safe and can be shared with many threads. |
| HashMap allows one null key and multiple null values**.** | Hashtabledoesn't allow any null key or value**.** |
| HashMap is a new class introduced in JDK 1.2**.** | Hashtable is a legacy class**.** |
| We can make the HashMap as synchronized by calling this code  Map m=Collections.synchronizedMap(hashMap); | Hashtable is internally synchronized and can't be unsynchronized. |
| HashMap is traversed by Iterator. | Hashtable istraversed by Enumerator and Iterator**.** |
| Iterator in HashMap is fail-fast. | Enumerator in Hashtable is not fail-fast**.** |
| HashMap inherits AbstractMap class. | Hashtable inherits Dictionary class. |
| HashMap stores key, value pairs and **it does not allow duplicate keys**. If the key is duplicate then the old key is replaced with the new value. |  |

**Difference between HashMap and TreeMap.**

|  |  |
| --- | --- |
| **HashMap** | **TreeMap** |
| HashMap does not maintain any order. | TreeMap elements are sorted according to the natural ordering of its elements. |
| The Key Can is different type. | The Key should be in same type. |
| HashMap can store one null key and many null values. | TreeMap cannot contain null keys but may contain many null values. |
| Internal HashMap implementation use Hashing. | TreeMap internally uses Red-Black tree implementation. |
| HashMap is Fast. | TreeMap is slow in Comparison. |
| For Comparison uses equals() method | For Comparison uses CompareTo()/Compare() method |

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

|  |  |
| --- | --- |
| **Fail-Fast Iterators** | **Fail-Safe Iterators** |
| Fail-Fast iterators doesn’t allow modifications of a collection while iterating over it. | Fail-Safe iterators allow modifications of a collection while iterating over it. |
| These iterators throw *ConcurrentModificationException* if a collection is modified while iterating over it. | These iterators don’t throw any exceptions if a collection is modified while iterating over it. |
| They use original collection to traverse over the elements of the collection. | They use copy of the original collection to traverse over the elements of the collection. |
| These iterators don’t require extra memory. | These iterators require extra memory to clone the collection. |
| Ex : Iterators returned by *ArrayList*, *Vector*,*HashMap*,HashSet | CopyOnWriteArrayList, ConcurrentHashMap |

**Difference between Comparator vs Comparable in Java.**

|  |  |
| --- | --- |
| Comparable | Comparator |
| Comparable is found in **java.lang** package. | Comparator is found in **java.util** package. |
| Following abstract method available  public int compareTo(object obj) | Following abstract method available  public int compare(objectobj1,Object obj2)  public boolean equals(Object obj1) |
| One Object you can use using this keywords and another object specified as argument. | You need to define a separate class by implementing this interface and you need to provide two object as the argument to compare() method that will be compared. |
| Comparable provides **single sorting sequence**. In other words, we can sort the collection on the basis of single element such as id or name or price etc. | Comparator provides **multiple sorting sequence**. In other words, we can sort the collection on the basis of multiple elements such as id, name and price etc. |
| Comparable **affects the original class** i.e. actual class is modified. | Comparator **doesn't affect the original class** i.e. actual class is not modified. |
| We can sort the list elements of Comparable type by Collections.sort(List) method. | We can sort the list elements of Comparator type by **Collections.sort(List,Comparator)**  method. |

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

* **Collection** : The root interface of Java Collections Framework.
* **Collections** : A utility class that is a member of the Java Collections Framework.

**What is the difference between stack and queue?**

* Stack is a LIFO (last in first out) data structure.
* Queue is a FIFO (first in first out) data structure

**What is the difference between Synchronized Collection and Concurrent Collection?**

**ConcurrentHashMap**

You should use ConcurrentHashMap when you need very high concurrency in your project.

It is thread safe without synchronizing the whole map.

Reads can happen very fast while write is done with a lock.

There is no locking at the object level.

The locking is at a much finer granularity at a hashmap bucket level.

ConcurrentHashMap doesn’t throw a ConcurrentModificationException if one thread tries to modify it while another is iterating over it.

ConcurrentHashMap uses multitude of locks.

**SynchronizedHashMap**

Synchronization at Object level.

Every read/write operation needs to acquire lock.

Locking the entire collection is a performance overhead.

This essentially gives access to only one thread to the entire map & blocks all the other threads.

It may cause contention.

SynchronizedHashMap returns Iterator, which fails-fast on concurrent modification.

# Difference between [ConcurrentHashMap and Hashtable](http://stackoverflow.com/questions/12646404/concurrenthashmap-and-hashtable-in-java) .

ConcurrentHashMap uses multiple buckets to store data. This avoids read locks and greatly improves performance over a HashTable. Both are thread safe, but there are obvious performance wins with ConcurrentHashMap.

When you read from a ConcurrentHashMap using get(), there are no locks, contrary to the HashTable for which all operations are simply synchronized. HashTable was released in old versions of Java whereas ConcurrentHashMap is a java 5+ thing.

HashMap is the best thing to use in a single threaded application.

Hashtable is belongs to Collection framework ConcurrentHashMap belongs to Executor framework.

Hashtable uses single lock for whole data. ConcurrentHashMap uses multiple locks on Segment level (16 by default) instead of object level i.e. whole Map.

ConcurrentHashMap Locking is applied only for updates. In case of of retrievals, it allows full concurrency, retrievals reflect the results of the most recently completed update operations. So Reads can happen very fast while write is done with a lock.

ConcurrentHashMap doesn’t throw a ConcurrentModificationException if one thread tries to modify it while another is iterating over it and does not allow null values.

ConcurrentHashMap returns Iterator, which fails-safe(i.e. iterator will make a copy of the internal data structure ) on concurrent modification.

ConcurrentHashMap uses a database shards logic(Segment<K, V>[] segments) , i.e. divide the data into shards(segments) than puts locks on each shard(segment) instead of putting single lock for whole data(Map).

**When to use List, Set and Map in Java?**

1) If you do not want to have duplicate values in the database then Set should be your first choice as all of its classes do not allow duplicates.

2) If there is a need of frequent search operations based on the index values then List (ArrayList) is a better choice.

3) If there is a need of maintaining the insertion order then also the List is a preferred collection interface.

4) If the requirement is to have the key & value mappings in the database then Map is your best bet.

**How to make a collection read only?**

Use following methods:

Collections.unmodifiableList(list);

Collections.unmodifiableSet(set);

Collections.unmodifiableMap(map);

These methods takes collection parameter and return a new read-only collection with same elements as in original collection.

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

Fail-fast Iterators fail as soon as they realized that structure of Collection has been changed since iteration has begun. Structural changes means adding, removing or updating any element from collection while one thread is Iterating over that collection.

Fail-fast behavior is implemented by keeping a modification count and if iteration thread realizes the change in modification count it throws ConcurrentModificationException.

**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.

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

**//Using for-each loop**

for(String obj : strList){

System.out.println(obj);

}

**//Using iterator**

Iterator<String> it = strList.iterator();

while(it.hasNext()){

String obj = it.next();

System.out.println(obj);

}

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

**How to reverse the list?**

This question is just like above to test your knowledge of Collections utility class. Use it reverse() method to reverse the list.

**Collections.reverse(list);**

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

1) Your application can live without Random access. Because if you need nth element in LinkedList you need to first traverse up to nth element O(n) and than you get data from that node.

2) Your application is more insertion and deletion driver and you insert or remove more than retrieval. Since insertion or removal doesn't involve resizing its much faster than ArrayList.

That’s all on difference between ArrayList and LinkedList in Java. Use ArrayList in Java for all there situation where you need a non-synchronized index based access. ArrayList is fast and easy to use, just try to minimize array resizing by constructing arraylist with proper initial size.

**How to convert an array of String to ArrayList?**

This is more of a programmatic question which is seen at beginner level. The intent is to check the knowledge of applicant in Collection utility classes. For now, lets learn that there are two utility classes in Collection framework which are mostly seen in interviews i.e. Collections and Arrays.

Collections class provides some static functions to perform specific operations on collection types. And Arrays provide utility functions to be performed on array types.

**//String array**

String[] words = {"ace", "boom", "crew", "dog", "eon"};

**//Use Arrays utility class**

List wordList = Arrays.asList(words);

**//Now you can iterate over the list**

Please not that this function is not specific to String class, it will return List of element of any type, of which the array is. e.g.

//String array

Integer[] nums = {1,2,3,4};

//Use Arrays utility class

List numsList = Arrays.asList(nums);

**Why we use Set interface? What are main classes implementing Set interface?**

It models the mathematical set in set theory. Set interface is like List interface but with some differences. First, it is not ordered collection. So no ordering is preserved while adding or removing elements. The main feature it does provide is “uniqueness of elements“. It does not support duplicate elements.

Set also adds a stronger contract on the behaviour of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ. Two Set instances are equal if they contain the same elements.

Based on above reasons, it does not have operations based on indexes of elements like List. It only has methods which are inherited by Collection interface.

Main classes implementing Set interface are : EnumSet, HashSet, LinkedHashSet, TreeSet.

**How HashSet store elements?**

You must know that HashMap store key-value pairs, with one condition i.e. keys will be unique. HashSet uses Map’s this feature to ensure uniqueness of elements. In HashSet class, a map declaration is as below:

private transient HashMap<E,Object> map;

//This is added as value for each key

private static final Object PRESENT = new Object();

So when you store a element in HashSet, it stores the element as key in map and “PRESENT” object as value. (See declaration above).

public boolean add(E e) {

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

}

**Can a null element added to a TreeSet or HashSet?**

As you see, There is no null check in add() method in previous question. And HashMap also allows one null key, so one “null” is allowed in HashSet.

TreeSet uses the same concept as HashSet for internal logic, but uses NavigableMap for storing the elements.

private transient NavigableMap<E,Object> m;

// Dummy value to associate with an Object in the backing Map

private static final Object PRESENT = new Object();

NavigableMap is subtype of SortedMap which does not allow null keys. So essentially, TreeSet also does not support null keys. It will throw NullPointerException if you try to add null element in TreeSet.

**Why we use Map interface? What are main classes implementing Map interface?**

Map interface is a special type of collection which is used to store key-value pairs. It does not extend Collection interface for this reason. This interface provides methods to add, remove, search or iterate over various views of Map.

Main classes implementing Map interface are: HashMap, Hashtable, EnumMap, IdentityHashMap, LinkedHashMap and Properties.

**What are IdentityHashMap and WeakHashMap?**

**IdentityHashMap** is similar to HashMap except that it uses reference equality when comparing elements. IdentityHashMap class is not a widely used Map implementation. While this class implements the Map interface, it intentionally violates Map’s general contract, which mandates the use of the equals() method when comparing objects. IdentityHashMap is designed for use only in the rare cases wherein reference-equality semantics are required.

**WeakHashMap** is an implementation of the Map interface that stores only weak references to its keys. Storing only weak references allows a key-value pair to be garbage collected when its key is no longer referenced outside of the WeakHashMap. This class is intended primarily for use with key objects whose equals methods test for object identity using the == operator. Once such a key is discarded it can never be recreated, so it is impossible to do a look-up of that key in a WeakHashMap at some later time and be surprised that its entry has been removed.

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

Map interface provides 3 views of key-values pairs stored in it:

key set view

value set view

entry set view

All the views can be navigated using iterators.

**How to make a collection thread safe?**

Use below methods:

Collections.synchronizedList(list);

Collections.synchronizedSet(set);

Collections.synchronizedMap(map);

Above methods take collection as parameter and return same type of collection which are synchronized and thread safe.

**How to design a good key for hashmap?**

Another good question usually followed up after answering how hashmap works. Well, the most important constraint is you must be able to fetch the value object back in future. Otherwise, there is no use of having such a data structure. If you understand the working of hashmap, you will find it largely depends on hashCode() and equals() method of Key objects.

So a good key object must provide same hashCode() again and again, no matter how many times it is fetched. Similarly, same keys must return true when compare with equals() method and different keys must return false.

For this reason, immutable classes are considered best candidate for HashMap keys.

**How HashSet Works Internally In Java?**

Whenever you insert an element into HashSet using add() method, it actually creates an entry in the internally backing HashMap object with element you have specified as it’s key and constant called “PRESENT” as it’s value. This “PRESENT” is defined in the HashSet class as below.

// Dummy value to associate with an Object in the backing Map

private static final Object PRESENT = new Object();

Let’s have a look at add() method of HashSet class.

public boolean add(E e)

{

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

}

You can notice that, add() method of HashSet class internally calls put() method of backing HashMap object by passing the element you have specified as a key and constant “PRESENT” as it’s value.

remove() method also works in the same manner.

**public boolean remove(Object o)**

**{**

**return map.remove(o)==PRESENT;**

**}**

Let’s see one example of HashSet and how it maintains HashMap internally.

public class HashSetExample {

public static void main(String[] args){

//Creating One HashSet object

HashSet<String> set = new HashSet<String>();

//Adding elements to HashSet

set.add("RED");

set.add("GREEN");

set.add("BLUE");

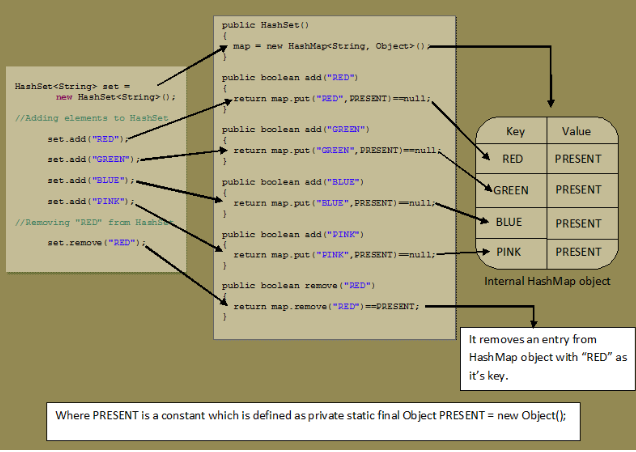
set.add("PINK");

//Removing "RED" from HashSet

set.remove("RED");

}

}

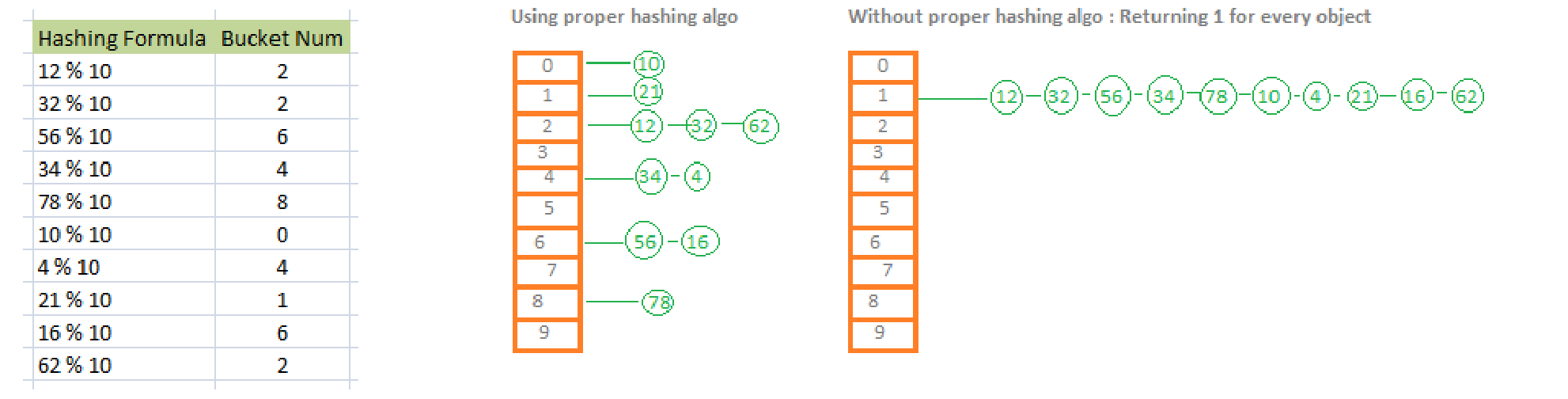
See the below picture how above program works internally. You can observe that internal HashMap object contains elements of HashSet as keys and constant “PRESENT” as their value. In the same manner, all methods of HashSet class process internally backing HashMap object to get the desired result. If you know how HashMap works, it will be easy for you to understand how HashSet works. You go through the source code of HashSet class once, you will get a clear picture about how HashSet works internally in Java.

**How is HashMap actually implemented in java ?**

This one of the java internals questions which has become common these days, While some people say why do we need to bother about it, haven’t we got a java API for that; there are few who have really dug up API’s getting insights over the internals. We’re gonna be the second type now : Before we dig up,

* **HashMap** works on the principal of hashing.
* **Map.Entry interface** - This interface gives a map entry (key-value pair). HashMap in Java stores both key and value object, in bucket, as Entry object which implements this nested interface Map.Entry.
* **hashCode()** -HashMap provides put(key, value) for **storing** and get(key) method for **retrieving** Values from HashMap. When put() method is used to store (Key, Value) pair, HashMap implementation **calls hashcode** on Key object to calculate a hash that is used to find a bucket where Entry object will be stored. When get() method is used to retrieve value, again key object is used to calculate a hash which is used then to find a bucket where that particular key is stored.
* **equals()** - equals() method is used to **compare objects for equality**. In case of HashMap key object is used for comparison, also using equals() method Map knows how to handle **hashing collision** (hashing collision means more than one key having the same hash value, thus assigned to the same bucket. In that case objects are stored in a linked list.  
  Where hashCode method helps in finding the bucket where that key is stored, equals method helps in finding the right key as there may be more than one key-value pair stored in a single bucket.

**Lets understand little Hashing from basics**. I have 10 numbers : 12, 32, 56, 34, 78, 10, 4, 21, 16, 62. (consider these numbers as hashcodes returned from your hashCode method). And I have a hashing function mod 10, to distribute my numbers in hashmap. The hashing function uses the numbers to give us the bucket number where my object must fall. And the hashing function has a pre defined number of buckets. So since my hashing function is mod 10, so the number of buckets which I want to use is also 10. And we put values in the bucket using our hashing algo ‘mod 10’. The point where two values give the same bucket number is known as a Hash Collision. e.g. 12%10 and 32%10, both result is equal to 2. This is a **hash collision**. There are number of ways to resolve hash collision, a common solution is by using LinkedList of elements for each bucket. Every time a new element comes in, its hashcode would be used to calculate a bucket number, and if the bucket is initially empty the object is directly put in the bucket, or else a hash collision occurs and the element is added to the end of the linked list. The below diagram shows the calculated bucket number for each item, and puts them in buckets. Now you can see the advantage of the Hashing where we have distributed the values somewhat evenly, and for searching an element we just have to go to the bucket number and search the value among the values in that particular bucket. Also, every node in the linked list keeps both the key and value inside each node. The key inside node would be helpful for retrieving values from the HashMap.

****

The above diagram also shows an poor hashcode implementation, where I would always return a constant value from my hashcode method, and when hashing algo is applied on the hashcode a collision occurs for every element, hence all elements fall in a single bucket in a very long linked list. **Now, How java implements Hashing ?**

For putting an element into the map via the ‘*put(key, value)*’ method, java gets the hashCode() of the key, and applies its hash algo over the hashcode to figure out the bucket number the *value* object has to fall into. Now java knows where the value has to fall. Now it needs to find if the key is already present in the Map, whether it has to insert new element into map or update/replace with old value. So it goes to the bucket to check if it has any element in it. If there is no element present in the bucket, the element is directly added to the bucket, else if there is an element present, java would now have to traverse through the LinkedList to find if the key is alersdy present in it. It uses the equals() method on *key* of the elemts to find that out. If it gets a matched key it replaces the node of the LinkedList, else it would make another enty in the end of the LinkedList. For getting an element from the map, using the *get(key)* on the object, it again gets the hashcode and applies the hash algo to get the bucket number, there it uses the equals() method to find the node with the matching key, and return the corresponding value.

**When to use HashMap or TreeMap?**

HashMap is well known class and all of us know that. So, I will leave this part by saying that it is used to store key-value pairs and allows to perform many operations on such collection of pairs.

TreeMap is special form of HashMap. It maintains the ordering of keys which is missing in HashMap class. This ordering is by default “natural ordering”. The default ordering can be override by providing an instance of Comparator class, whose compare method will be used to maintain ordering of keys.

Please note that all keys inserted into the map must implement the Comparable interface (this is necessary to decide the ordering). Furthermore, all such keys must be mutually comparable: k1.compareTo(k2) must not throw a ClassCastException for any keys k1 and k2 in the map. If the user attempts to put a key into the map that violates this constraint (for example, the user attempts to put a string key into a map whose keys are integers), the put(Object key, Object value) call will throw a ClassCastException.

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

You should first try to find another alternative iterator which are fail-safe. For example if you are using List and you can use ListIterator. If it is legacy collection, you can use enumeration.

If above options are not possible then you can use one of three changes:

If you are using JDK1.5 or higher then you can use ConcurrentHashMap and CopyOnWriteArrayList classes. It is the recommended approach.

You can convert the list to an array and then iterate on the array.

You can lock the list while iterating by putting it in a synchronized block.

**What is UnsupportedOperationException?**

This exception is thrown on invoked methods which are not supported by actual collection type.

For example, if you make a read-only list list using “Collections.unmodifiableList(list)” and then call add() or remove() method, what should happen. It should clearly throw UnsupportedOperationException.

**What is BlockingQueue?**

A 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.

BlockingQueue methods come in four forms: one throws an exception, the second returns a special value (either null or false, depending on the operation), the third blocks the current thread indefinitely until the operation can succeed, and the fourth blocks for only a given maximum time limit before giving up.

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

A collection designed for holding elements prior to processing. Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations.

Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner.

Stack is also a form of Queue but one difference, it is LIFO (last-in-first-out).

Whatever the ordering used, the head of the queue is that element which would be removed by a call to remove() or poll(). Also note that Stack and Vector are both synchronized.

Usage: Use a queue if you want to process a stream of incoming items in the order that they are received.Good for work lists and handling requests.

Use a stack if you want to push and pop from the top of the stack only. Good for recursive algorithms.

**What are Collections and Arrays class?**

Collections and Arrays classes are special utility classes to support collection framework core classes. They provide utility functions to get read-only/ synchronized collections, sort the collection on various ways etc.

Arrays also helps array of objects to convert in collection objects. Arrays also have some functions which helps in copying or working in part of array objects.

**What do you need to do to use a custom object as a key in Collection classes like Map or Set?**

The answer is: If you are using any custom object in Map as key, you need to override equals() and hashCode() method, and make sure they follow their contract. On the other hand if you are storing a custom object in Sorted Collection e.g. SortedSet or SortedMap, you also need to make sure that your equals() method is consistent to compareTo() method, otherwise that collection will not follow there contacts e.g. Set may allow duplicates.

**What is NavigableMap in Java? What is a benefit over Map?**

NavigableMap Map was added in Java 1.6, it adds navigation capability to Map data structure. It provides methods like lowerKey() to get keys which is less than specified key, floorKey() to return keys which is less than or equal to specified key, ceilingKey() to get keys which is greater than or equal to specified key and higherKey() to return keys which is greater specified key from a Map. It also provide similar methods to get entries e.g. lowerEntry(), floorEntry(), ceilingEntry() and higherEntry(). Apart from navigation methods, it also provides utilities to create sub-Map e.g. creating a Map from entries of an exsiting Map like tailMap, headMap and subMap. headMap() method returns a NavigableMap whose keys are less than specified, tailMap() returns a NavigableMap whose keys are greater than the specified and subMap() gives a NavigableMap between a range, specified by toKey to fromKey.

**Which one you will prefer between Array and ArrayList for Storing object and why?**

Though ArrayList is also backed up by array, it offers some usability advantage over array in Java. Array is fixed length data structure, once created you can not change it's length. On the other hand, ArrayList is dynamic, it automatically allocate a new array and copies content of old array, when it resize. Another reason of using ArrayList over Array is support of Generics. Array doesn't support Generics, and if you store an Integer object on a String array, you will only going to know about it at runtime, when it throws ArrayStoreException. On the other hand, if you use ArrayList, compiler and IDE will catch those error on the spot. So if you know size in advance and you don't need re-sizing than use array, otherwise use ArrayList.

**Can we replace Hashtable with ConcurrentHashMap?**

Yes we can replace Hashtable with ConcurrentHashMap and that's what suggested in Java documentation of ConcurrentHashMap. but you need to be careful with code which relies on locking behavior of Hashtable. Since Hashtable locks whole Map instead of a portion of Map, compound operations like if(Hashtable.get(key) == null) put(key, value) works in Hashtable but not in concurrentHashMap. instead of this use putIfAbsent() method of ConcurrentHashMap

**What is CopyOnWriteArrayList, how it is different than ArrayList and Vector?**

CopyOnWriteArrayList is new List implementation introduced in Java 1.5 which provides better concurrent access than Synchronized List. better concurrency is achieved by Copying ArrayList over each write and replace with original instead of locking. Also CopyOnWriteArrayList doesn't throw any ConcurrentModification Exception. Its different than ArrayList because its thread-safe and ArrayList is not thread-safe and it's different than Vector in terms of Concurrency. CopyOnWriteArrayList provides better Concurrency by reducing contention among readers and writers. Here is a nice table which compares performance of three of popular List implementation ArrayList, LinkedList and CopyOnWriteArrayList in Java:

Java questions from Collection framework

**Why ListIterator has added() method but Iterator doesn't or Why to add() method is declared in ListIterator and not on Iterator.**

ListIterator has added() method because of its ability to traverse or iterate in both direction of the collection. it maintains two pointers in terms of previous and next call and in a position to add a new element without affecting current iteration.

**When does ConcurrentModificationException occur on iteration?**

When you remove object using Collection's or List's remove method e.g. remove(Object element) or remove(int index), instead of Iterator's remove() method than ConcurrentModificationException occurs. As per Iterator's contract, if it detect any structural change in Collection e.g. adding or removing of the element, once Iterator begins, it can throw ConcurrentModificationException. Here are some tips to avoid ConcurrentModification in Java.

advanced Java interview questions with answers

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

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

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

Contract of hashCode() and equals() method.

**How to make collection immutable in java?**

Unmodifiable collections are usually read-only views (wrappers) of other collections. You can't add, remove or clear them, but the underlying collection can change.

Immutable collections can't be changed at all - they don't wrap another collection - they have their own elements.

Here's a quote from guava's ImmutableList Unlike

**Collections.unmodifiableList(java.util.List<? extends T>),**

which is a view of a separate collection that can still change, an instance of ImmutableList contains its own private data and will never change.

# [Immutable vs Unmodifiable collection](http://stackoverflow.com/questions/8892350/immutable-vs-unmodifiable-collection).

An unmodifiable collection is often a wrapper around a modifiable collection *which other code may still have access to*. So while *you* can't make any changes to it if you only have a reference to the unmodifiable collection, you can't rely on the contents not changing.

An *immutable* collection guarantees that *nothing* can change the collection any more. If it wraps a modifiable collection, it makes sure that no other code has access to that modifiable collection. Note that although no code can change which objects the collection contains references to, the objects themselves may still be mutable - creating an immutable collection of StringBuilder doesn't somehow "freeze" those objects.

Basically, the difference is about whether other code may be able to change the collection behind your back.

**When to use ConcurrentHashMap in Java**

1. ConcurrentHashMap allows concurrent read and thread-safe update operation.

2. During the update operation, ConcurrentHashMap only locks a portion of Map instead of whole Map.

3. The concurrent update is achieved by internally dividing Map into the small portion which is defined by concurrency level.

4. Choose concurrency level carefully as a significantly higher number can be a waste of time and space and the lower number may introduce thread contention in case writers over number concurrency level.

5. All operations of ConcurrentHashMap are thread-safe.

6. Since ConcurrentHashMap implementation doesn't lock whole Map, there is chance of read overlapping with update operations like put() and remove(). In that case result returned by get() method will reflect most recently completed operation from there start.

7. Iterator returned by ConcurrentHashMap is weekly consistent, fail-safe and never throw ConcurrentModificationException. In Java.

8. ConcurrentHashMap doesn't allow null as key or value.

9. You can use ConcurrentHashMap in place of Hashtable but with caution as CHM doesn't lock whole Map.

10. During putAll() and clear() operations, the concurrent read may only reflect insertion or deletion of some entries.

That’s all on What is ConcurrentHashMap in Java and when to use it. We have also seen little bit about internal working of ConcurrentHashMap and how it achieves it’s thread-safety and better performance over Hashtable and synchronized Map. Use ConcurrentHashMap in Java program, when there will be more reader than writers and it’s a good choice for creating cache in Java as well.

