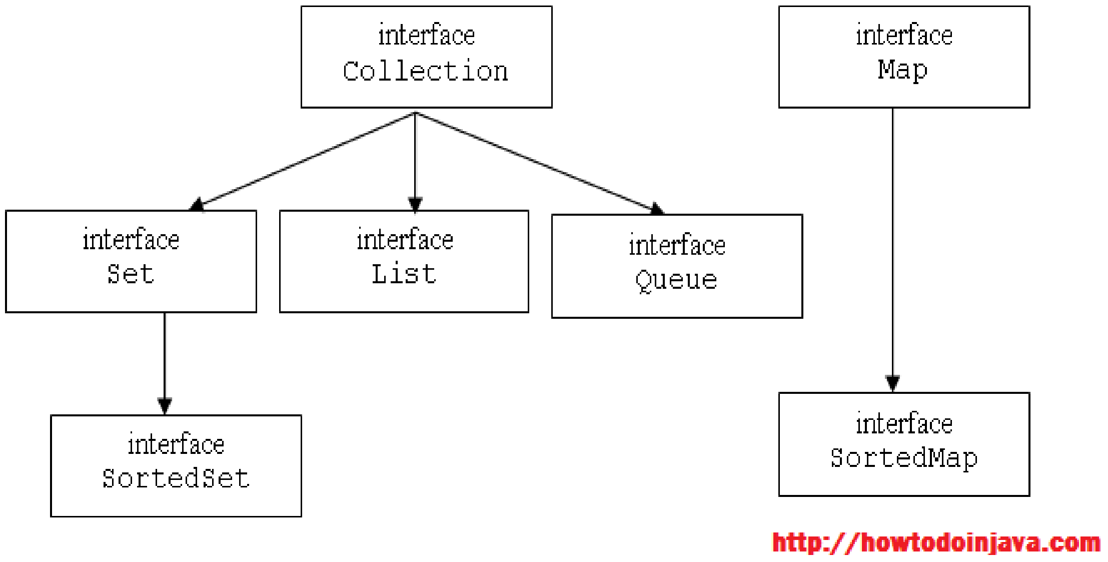
General question

1) What is the Java Collection framework? List down its advantages?

By definition, a collection is **an object that represents a group of objects**. Like in set theory, a set is group of elements. Easy enough !!
Prior to JDK 1.2, JDK has some utility classes such as Vector and HashTable, but there was no concept of Collection framework. Later from JDK 1.2 onwards, JDK felt the need of having a consistent support for reusable data structures. Finally, the collections framework was designed and developed primarily by Joshua Bloch, and was **introduced in JDK 1.2**.
Its most **noticeable advantages** can be listed as:

- Reduced programming effort due to ready to use code
- Increased performance because of high-performance implementations of data structures and algorithms
- Provides interoperability between unrelated APIs by establishing a common language to pass collections back and forth
- Easy to learn APIs by learning only some top level interfaces and supported operations

2) Explain Collection’s hierarchy?



Java Collection Hierarchy

As shown in above image, collection framework has one interface at top i.e. **Collection**. It is **extended by Set, List and Queue interfaces**. Then there are loads of other classes in these 3 branches which we will learn in following questions.
Remember the signature of Collection interface. It will help you in many question.

```
public interface Collection extends Iterable {
//method definitions
}
```

Framework also consist of Map interface, which is part of collection framework. but it does not extend Collection interface. We will see the reason in 4th question in this question bank.

3) Why Collection interface does not extend Cloneable and Serializable interface?

Well, simplest answer is "**there is no need to do it**". Extending an interface simply means that you are creating a subtype of interface, in other words a more specialized behavior and Collection interface is not expected to do what Cloneable and Serializable interfaces do.
Another reason is that not everybody will have a reason to have Cloneable collection because if it has very large data, then every **unnecessary clone operation will consume a big memory**. Beginners might use it without knowing the consequences.

Another reason is that **Cloneable and Serializable are very specialized behavior** and so should be implemented only when required. For example, many concrete classes in collection implement these interfaces. So if you want this feature. use these collection classes otherwise use their alternative classes.

4) Why Map interface does not extend Collection interface?

A good answer to this interview question is "**because they are incompatible**". Collection has a method add(Object o). Map can not have such method because it need key-value pair. There are other reasons also such as Map supports keySet, valueSet etc. Collection classes does not have such views.
Due to such big differences, Collection interface was not used in Map interface, and it was build in separate hierarchy.

List interface related

5) Why we use List interface? What are main classes implementing List interface?

A java list is a **"ordered" collection of elements**. This ordering is a **zero based index**. It does not care about duplicates. Apart from methods defined in Collection interface, it does **have its own methods** also which are largely to manipulate the collection **based on index location of element**. These methods can be grouped as search, get, iteration and range view. All above operations support index locations.
The main classes implementing List interface are: **Stack, Vector, ArrayList and LinkedList**. Read more about them in java documentation.

6) 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 = {&quot;ace&quot;, &quot;boom&quot;, &quot;crew&quot;, &quot;dog&quot;, &q
//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);
```

7) 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);
```

Set interface related

8) 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 behavior 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**. Read more on related java documentation.

9) 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;
}
```

I will highly suggest you to read this post: **How HashMap works in java?** This post will help you in answering all the HashMap related questions very easily.

10) 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.

Map interface related

11) 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.**

12) 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.

13) Explain ConcurrentHashMap? How it works?

Taking from java docs:
A hash table supporting full concurrency of retrievals and adjustable expected concurrency for updates. This class obeys the same functional specification as Hashtable, and includes versions of methods corresponding to each method of Hashtable. However, even though all operations are thread-safe, retrieval operations do not entail locking, and there is not any support for locking the entire table in a way that prevents all access. This class is fully interoperable with Hashtable in programs that rely on its thread safety but not on its synchronization details.

14) How hashmap works?

The **most important question** which is most likely to be seen in every level of job interviews. You must be very clear on this topic., not only because it is most asked question but also it will open up your mind in further questions related to collection APIs. Answer to this question is very large and you should read it my post: **How HashMap works?** For now, lets remember that HashMap works **on principle of Hashing**. A map by definition is : “An object that maps keys to values”. To store such structure, **it uses an inner class Entry**:

```
static class Entry implements Map.Entry
{
final K key;
V value;
Entry next;
final int hash;
...//More code goes here
}
```

Here key and value variables are used to store key-value pairs. Whole entry object is stored in an array.

```
/**
* The table, re-sized as necessary. Length MUST Always be a power of two.
*/
transient Entry[] table;
```

The index of array is calculated on basis on hashcode of Key object. Read more of linked topic.

15) 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**.
Read more : **How to design a good key for HashMap?**

16) 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.

17) 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.

Tell the difference questions

18) Difference between Set and List?

The most noticeable differences are :

- Set is unordered collection where List is ordered collection based on zero based index.
- List allow duplicate elements but Set does not allow duplicates.

- List does not prevent inserting null elements (as many you like), but Set will allow only one null element.

19) Difference between List and Map?

Perhaps most easy question. **List is collection of elements where as map is collection of key-value pairs**. There is actually lots of differences which originate from first statement. They have **separate top level interface, separate set of generic methods, different supported methods and different views of collection**.
I will take much time hear as answer to this question is enough as first difference only.

20) Difference between HashMap and HashTable?

There are several differences between HashMap and Hashtable in Java:

- Hashtable is synchronized, whereas HashMap is not.
- Hashtable does not allow null keys or values. HashMap allows one null key and any number of null values.
- The third significant difference between HashMap vs Hashtable is that Iterator in the HashMap is a fail-fast iterator while the enumerator for the Hashtable is not.

21) Difference between Vector and ArrayList?

Lets note down the differences:

- All the methods of Vector is synchronized. But, the methods of ArrayList is not synchronized.
- Vector is a Legacy class added in first release of JDK. ArrayList was part of JDK 1.2, when collection framework was introduced in java.
- By default, Vector doubles the size of its array when it is re-sized internally. But, ArrayList increases by half of its size when it is re-sized.

22) Difference between Iterator and Enumeration?

Iterators differ from enumerations in three ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with its remove() method. You can not add/remove elements from a collection when using enumerator.
- Enumeration is available in legacy classes i.e Vector/Stack etc. whereas Iterator is available in all modern collection classes.
- Another minor difference is that Iterator has improved method names e.g. Enumeration.hasMoreElement() has become Iterator.hasNext(), Enumeration.nextElement() has become Iterator.next() etc.

23) Difference between HashMap and HashSet?

HashMap is collection of key-value pairs whereas HashSet is un-ordered collection of unique elements. That's it. No need to describe further.

24) Difference between Iterator and ListIterator?

There are three Differences are there:

- We can use Iterator to traverse Set and List and also Map type of Objects. But List Iterator can be used to traverse for List type Objects, but not for Set type of Objects.
- By using Iterator we can retrieve the elements from Collection Object in forward direction only whereas List Iterator, which allows you to traverse in either directions using hasPrevious() and previous() methods.
- ListIterator allows you modify the list using add() remove() methods. Using Iterator you can not add, only remove the elements.

25) Difference between TreeSet and SortedSet?

SortedSet is an interface which TreeSet implements. That' it !!

26) Difference between ArrayList and LinkedList?

- LinkedList store elements within a doubly-linked list data structure. ArrayList store elements within a dynamically resizing array.
- LinkedList allows for constant-time insertions or removals, but only sequential access of elements. In other words, you can walk the list forwards or backwards, but grabbing an element in the middle takes time proportional to the size of the list. ArrayLists, on the other hand, allow random access, so you can grab any element in constant time. But adding or removing from anywhere but the end requires shifting all the latter elements over, either to make an opening or fill the gap.
- LinkedList has more memory overhead than ArrayList because in ArrayList each index only holds actual object (data) but in case of LinkedList each node holds both data and address of next and previous node.

More questions

27) 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.

28) 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.

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

The sole purpose of an Iterator is to enumerate through a collection. All collections contain the add() method to serve your purpose. There would be no point in adding to an Iterator because the **collection may or may not be ordered**. And **add() method can not have same implementation for ordered and unordered collections**.

30) What are different ways to iterate over a list?

You can iterate over a list using following ways:

- Iterator loop
- For loop
- For loop (Advance)
- While loop

Read more : http://www.mkyong.com/java/how-do-loop-iterate-a-list-in-java/

31) 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.

32) What is difference between fail-fast and fail-safe?

You have understood fail-fast in previous question. **Fail-safe iterators** are just opposite to fail-fast. **They never fail if you modify the underlying collection on which they are**

iterating, because they work on clone of Collection instead of original collection and that’s why they are called as fail-safe iterator.
Iterator of CopyOnWriteArrayList is an example of fail-safe Iterator also iterator written by ConcurrentHashMap keySet is also fail-safe iterator and never throw ConcurrentModificationException.

33) 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.

Please note that last two approaches will cause a performance hit.

34) 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.

35) Which collection classes provide random access of it’s elements?

ArrayList, HashMap, TreeMap, Hashtable classes provide random access to it’s elements.

36) 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.

Read the example usage of blocking queue in post : **How to use blocking queue?**

37) What is Queue and Stack, list down 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.

38) What is Comparable and Comparator interface?

In java. all collection which have feature of automatic sorting, uses compare methods to ensure the correct sorting of elements. For example classes which use sorting are TreeSet, TreeMap etc.

To sort the data elements a class needs to implement Comparator or Comparable interface. That's why all Wrapper classes like Integer,Double and String class implements Comparable interface.
Comparable helps in preserving default natural sorting, whereas Comparator helps in sorting the elements in some special required sorting pattern. The instance of comparator if passed usually as collection's constructor argument in supporting collections.

39) What are Collections and Arrays classes?

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.

HashMap and ConcurrentHashMap interview questions

1) How to design a good key for HashMap

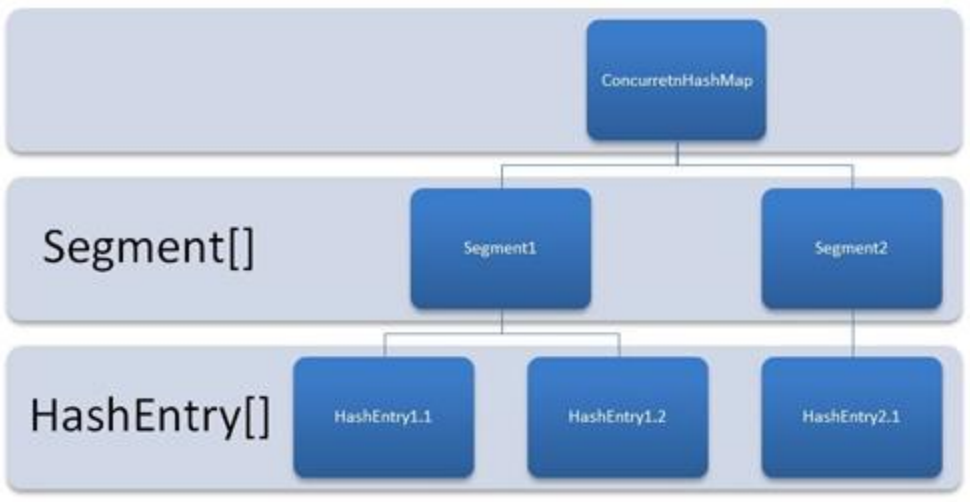
The very basic need for designing a good key is that "we should be able to retrieve the value object back from the map without failure", right?? Otherwise no matter how fancy data structure you build, it will be of no use. To decide that we have created a good key, we MUST know that "**how HashMap works?**". I will leave, how hashmap works, part on you to read from linked post, but in summary it works on principle of Hashing.
Key's hash code is used primarily in conjunction to its equals() method, for putting a key in map and then searching it back from map. So if hash code of key object changes after we have put a key-value pair in map, then its almost impossible to fetch the value object back from map. It is a case of memory leak. To avoid this, map **keys should be immutable**. These are few things to **create an immutable of class**.
This is the main reason why immutable classes like String, Integer or other wrapper classes are a good key object candidate.

But remember that **immutability is recommended and not mandatory**. If you want to make a mutable object as key in hashmap, then you have to make sure that **state change for key object does not change the hash code of object**. This can be done by overriding the hashCode() method. Also, key class must honor the **hashCode() and equals() methods contract** to avoid the undesired and surprising behavior on run time. Read more about this contract in linked post.
A more detailed information is available in **here**.

2) Difference between HashMap and ConcurrentHashMap

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

In concurrentHashMap, the **difference lies in internal structure to store these key-value pairs**. ConcurrentHashMap has an addition concept of segments. It will be easier to understand it you think of one segment equal to one HashMap [conceptually]. A concurrentHashMap is divided into number of segments [default 16] on initialization. ConcurrentHashMap allows similar number (16) of threads to access these segments concurrently so that each thread work on a specific segment during high concurrency. This way, if when your key-value pair is stored in segment 10; code does not need to block other 15 segments additionally. This structure provides a very high level of concurrency.



ConcurrentHashMap Internal Structure

In other words, **ConcurrentHashMap uses a multitude of locks, each lock controls one segment of the map**. When setting data in a particular segment, the lock for that segment is obtained. So essentially **update operations are synchronized**.
When getting data, a volatile read is used without any synchronization. If the volatile read results in a miss, then the lock for that segment is obtained and entry is again searched in synchronized block.
I will go further deeper into this concept in my coming post. I will suggest you to subscribe email updates to get notified.

3) Difference between HashMap and Collections.synchronizedMap(HashMap)

It’s easy question, right !! HashMap is non-synchronized and Collections.synchronizedMap() returns a wrapped instance of HashMap which has all get, put methods synchronized.

Essentially, **Collections.synchronizedMap() returns the reference of internally created inner-class “SynchronizedMap”**, which contains key-value pairs of input HashMap, passed as argument.
This instance of inner class has nothing to do with original parameter HashMap instance and is completely independent.

4) Difference between ConcurrentHashMap and Collections.synchronizedMap(HashMap)

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

As stated above, ConcurrentHashMap is consist of internal segments which can be viewed as independent HashMaps, conceptually. All such segments can be locked by separate threads in high concurrent executions. In this way, **multiple threads can get/put key-value pairs from ConcurrentHashMap without blocking/waiting for each other**.
In Collections.synchronizedMap(), we get a synchronized version of HashMap and **it is accessed in blocking manner**. This means if multiple threads try to access synchronizedMap at same time, they will be allowed to get/put key-value pairs one at a time in synchronized manner.

5) Difference between HashMap and HashTable

It is also very easy question. The major difference is that **HashTable is synchronized and HashMap is not**.
If asked for other reasons, tell them, **HashTable is legacy class** (part of JDK 1.0) which was promoted into collections framework by implementing Map interface later. It still has some **extra features like Enumerator** with it, which HashMap lacks.
Another minor reason can be: **HashMap supports null key** (mapped to zero bucket), HashTable does not support null keys and throws NullPointerException on such attempt.

6) Difference between HashTable and Collections.synchronized(HashMap)

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

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

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

7) Impact of random/fixed hashcode() value for key

The impact of both cases (fixed hashcode or random hashcode for keys) will have same result and that is "**unexpected behavior**". The very basic need of hashcode in HashMap is to identify the bucket location where to put the key-value pair, and from where it has to be retrieved.
If the hashcode of key object changes every time, the exact location of key-value pair will be calculated different, every time. This way, one object stored in HashMap will be lost forever and there will be very minimum possibility to get it back from map.

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

8) Using HashMap in non-synchronized code in multi-threaded application

In normal cases, it **can leave the hashmap in inconsistent state** where key-value pairs added and retrieved can be different. Apart from this, other surprising behavior like NullPointerException can come into picture.
In worst case, **It can cause infinite loop**. YES. You got it right. It can cause infinite loop. What did you asked, How?? Well, here is the reason.
HashMap has the concept of rehashing when it reaches to its upper limit of size. This rehashing is the process of creating a new memory area, and copying all the already present key-value pairs in new memory are. Lets say Thread A tried to put a key-value pair in map and then rehashing started. At the same time, thread B came and started manipulating the buckets using put operation.

Here while rehashing process, there are chances to generate the cyclic dependency where one element in linked list [in any bucket] can point to any previous node in same bucket. This will result in infinite loop, because rehashing code contains a "while(true) { //get next node; }" block and in cyclic dependency it will run infinite.

To watch closely, look art source code of transfer method which is used in rehashing:

```
public Object get(Object key) {

    Object k = maskNull(key);

    int hash = hash(k);

    int i = indexFor(hash, table.length);

    Entry e = table[i];


    //While true is always a bad practice and cause infinite loops


    while (true) {

        if (e == null)

            return e;

        if (e.hash == hash && eq(k, e.key))

            return e.value;

        e = e.next;

    }

}
```

Difference between HashMap and a Hashtable

1) `Hashtable` **is synchronized** (i.e. methods defined inside `Hashtable`), whereas `HashMap` **is not**. If you want to make a`HashMap` thread-safe, use `Collections.synchronizedMap(map)` or `ConcurrentHashMap` class. Methods inside `HashTable` are defined synchronized as below:

```
public synchronized boolean contains(Object obj){ ... }

public synchronized boolean containsKey(Object obj){ ... }

public synchronized Object get(Object obj){ ... }

public synchronized Object put(Object obj, Object obj1){ ... }

public synchronized Object remove(Object obj){ ... }
```

2) Hashtable does not allow null keys or values. `HashMap` **allows one null key** (other null keys will simply overwrite first null key) and **any number of null values**.

```
Hashtable<String, String> hashTable = new Hashtable<String, String>();

hashTable.put(null, "value");

//OR

hashTable.put("key", null);



Output:



Exception in thread "main" java.lang.NullPointerException

    at java.util.Hashtable.hash(Unknown Source)

    at java.util.Hashtable.put(Unknown Source)

    at test.core.MapExamples.main(MapExamples.java:12)
```

3) `Hashtable` **is legacy class** and was not part of the initial Java Collections Framework (later it was included in JDK 1.2).`HashMap` **is part of Collections since it's birth**. Also note that `Hashtable` extends the `Dictionary` class, which as the Javadocs state, is obsolete and has been replaced by the `Map` interface in newer JDK versions.

```
//HashTable is defined as

public class Hashtable extends Dictionary implements Map, Cloneable, Serializable {}

//HashMap is defined as

public class HashMap extends AbstractMap implements Map, Cloneable, Serializable {}
```

4) `Iterator` in the `HashMap` **is fail-fast** and throw `ConcurrentModificationException` if any other Thread modifies the map structurally by adding or removing any element except Iterator's own ***remove()*** method. But this is not a guaranteed behavior and will be done by JVM on best effort. The **enumerator for the Hashtable is not fail-fast**.

```
HashMap<String, String> hashMap = new HashMap<String, String>();

hashMap.put("key1", "value1");

hashMap.put("key2", "value2");

hashMap.put("key3", "value3");

hashMap.put("key4", "value4");
```

```
Iterator<String> iterator = hashMap.keySet().iterator();

while(iterator.hasNext()){

    iterator.next();

    iterator.remove();

    System.out.println(hashMap);

}


Output:


{key3=value3, key2=value2, key1=value1}

{key2=value2, key1=value1}

{key1=value1}

{}
```

5) Finally, Map fixes a minor deficiency in the `Hashtable` interface. `Hashtable` has a method called "**contains()**", which returns true if the `Hashtable` contains a given value. Given its name, you may expect this method to return true if the`Hashtable` contained a given key, because the key is the primary access mechanism for a `Hashtable`.
The `Map` interface eliminates this source of confusion by renaming the method to "**containsValue()**" and "**containsKey()**".

```
public boolean containsKey(Object obj) {...}

public boolean containsValue(Object obj) {...}
```

Suggestion regarding usage of HashMap vs Hashtable

There is hardly any job which HashMap or it's related classes
(i.e. `LinkedHashMap` or `ConcurrentHashMap`) can not do which HashTable does. So, there

is no good reason to use Hashtable in new code you write. **Always prefer to use HashMap over HashTable**.
It’s really hard to go deeper and deeper inside this list. As soon as you are able to understand above differences, you are good to use both the classes (infact only HashMap you should use). For me, analyzing beyond above points is simply waste of time. So, I am stopping here.

Best practices for using ConcurrentHashMap

The ConcurrentHashMap is very similar to the HashMap class, except that ConcurrentHashMap offers internally maintained concurrency. It means you do not need to have synchronized blocks when accessing ConcurrentHashMap in multithreaded application.

```
//Initialize ConcurrentHashMap instance
ConcurrentHashMap<String, Integer> m = new ConcurrentHashMap<String, Integer>();

//Print all values stored in ConcurrentHashMap instance
for each (Entry<String, Integer> e : m.entrySet())
{
system.out.println(e.getKey()+"="+e.getValue());
}
```

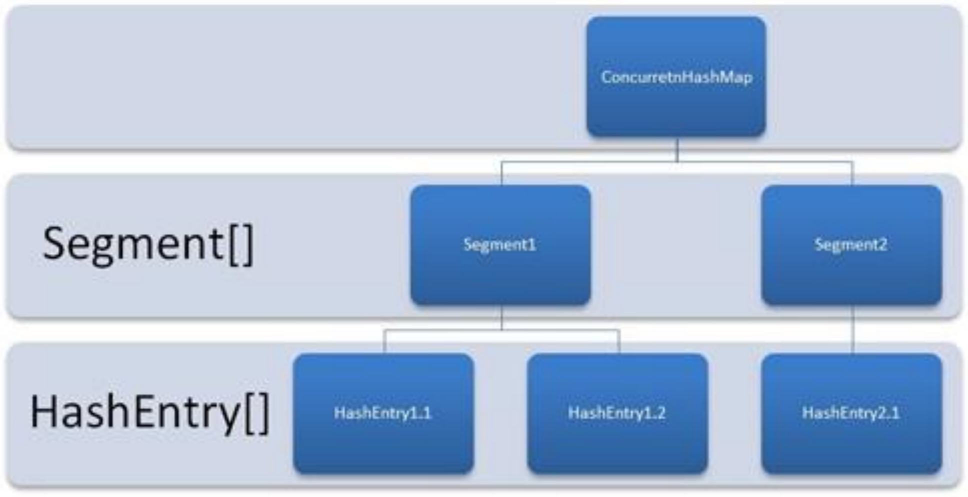
Above code is reasonably valid in multi-threaded environment in your application. The reason, I am saying “reasonably valid” is that, above code yet provides thread safety, still it can decrease the performance of application. And ConcurrentHashMap was introduced to improve the performance while ensuring thread safety, right??

So, what is that we are missing here??

To understand that we need to understand the internal working of ConcurrentHashMap class. And the best way to start is look at the constructor arguments. Fully parametrized constructor of ConcurrentHashMap takes 3 parameters, initialCapacity, loadFactor and concurrencyLevel.

1) initialCapacity
2) loadFactor
3) concurrencyLevel

First two are fairly simple as their name implies but last one is tricky part. This denotes the number of shards. It is used to divide the ConcurrentHashMap internally into this number of partitions and equal number of threads are created to maintain thread safety maintained at shard level.



ConcurrentHashMap

The default value of "concurrencyLevel" is 16. It means 16 **shards** whenever we create an instance of ConcurrentHashMap using default constructor, before even adding first key-value pair. It also means the creation of instances for various inner classes like ConcurrentHashMap$Segment, ConcurrentHashMap$HashEntry[] and ReentrantLock$NonfairSync.
In most cases in normal application, a single shard is able to handle multiple threads with reasonable count of key-value pairs. And performance will be also optimal. Having multiple shards just makes the things complex internally and introduces a lot of un-necessary objects for garbage collection, and all this for no performance improvement.

The extra objects created per concurrent hashmap using default constructor are normally in ratio of 1 to 50 i.e. for 100 such instance of ConcurrentHashMap, there will be 5000 extra objects created.

Based on above, I will suggest to **use the constructor parameters wisely to reduce the number of unnecessary objects and improving the performance**.
A good approach can be having initialization like this:

```
ConcurrentHashMap<String, Integer> instance =
new ConcurrentHashMap<String, Integer>(16, 0.9f, 1);
```

An initial capacity of 16 ensures a reasonably good number of elements before resizing happens. Load factor of 0.9 ensures a dense packaging inside ConcurrentHashMap which will optimize memory use. And concurrencyLevel set to 1 will ensure that only one shard is created and maintained.

Please note that if you are working on very high concurrent application with very high frequency of updates in ConcurrentHashMap, you should consider increasing the concurrencyLevel more than 1, but again it should be a well calculated number to get the best results.

How hashmap works in java

Single statement answer

If anybody asks me to describe "**How HashMap works?**", I simply answer: "**On principle of Hashing**". As simple as it is. Now before answering it, one must be very sure to know at least basics of Hashing. Right??

What is Hashing

Hashing in its simplest form, is a way to assigning a unique code for any variable/object after applying any formula/algorithm on its properties. A true Hashing function must follow this rule:

Hash function should return the same hash code each and every time, when function is applied on same or equal objects. In other words, two equal objects must produce same hash code consistently.

All objects in java inherit a default implementation of hashCode() function defined in Object class. This function produce hash code by typically converting the internal address of the object into an integer, thus producing different hash codes for all different objects.

Read more here : Working with hashCode() and equals() methods in java

A little about Entry class

A map by definition is : **"An object that maps keys to values"**. Very easy.. right?
So, there must be some mechanism in HashMap to store this key value pair. Answer is YES. HashMap has an inner class Entry, which looks like this:

```
static class Entry<K ,V> implements Map.Entry<K ,V>

{

    final K key;

    V value;

    Entry<K ,V> next;

    final int hash;

    ...//More code goes here

}
```

Surely Entry class has key and value mapping stored as attributes. Key has been marked as final and two more fields are there: next and hash. We will try to understand the need of these fields as we go forward.

What put() method actually does

Before going into put() method's implementation, it is very important to learn that instances of Entry class are stored in an array. HashMap class defines this variable as:

```
/**
 * The table, resized as necessary. Length MUST Always be a power of two.
 */
transient Entry[] table;
```

Now look at code implementation of put() method:

```
/**
* Associates the specified value with the specified key in this map. If the
* map previously contained a mapping for the key, the old value is
* replaced.
*
* @param key
*            key with which the specified value is to be associated
* @param value
*            value to be associated with the specified key
* @return the previous value associated with <tt>key</tt>, or <tt>null</tt>
*         if there was no mapping for <tt>key</tt>. (A <tt>null</tt> return
*         can also indicate that the map previously associated
*         <tt>null</tt> with <tt>key</tt>.)
*/
public V put(K key, V value) {
    if (key == null)
    return putForNullKey(value);
    int hash = hash(key.hashCode());
    int i = indexFor(hash, table.length);
    for (Entry<K , V> e = table[i]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;

            e.value = value;

            e.recordAccess(this);

            return oldValue;

        }

    }


    modCount++;

    addEntry(hash, key, value, i);

    return null;

}
```

Lets note down the steps one by one:

1) First of all, key object is checked for null. If key is null, value is stored in table[0] position. Because hash code for null is always 0.

2) Then on next step, a hash value is calculated using key's hash code by calling its hashCode() method. This hash value is used to calculate index in array for storing Entry object. JDK designers well assumed that there might be some poorly written hashCode() functions that can return very high or low hash code value. To solve this issue, they introduced another **hash()**function, and passed the object's hash code to this hash() function to bring hash value in range of array index size.
3) Now *indexFor(hash, table.length)* function is called to calculate exact index position for storing the Entry object.
4) Here comes the main part. Now, as we know that two unequal objects can have same hash code value, how two different objects will be stored in same array location [**called bucket**].
Answer is LinkedList. If you remember, Entry class had an attribute **"next"**. This attribute always points to next object in chain. This is exactly the behavior of LinkedList. So, in case of collision, Entry objects are stored in LinkedList form. When an Entry object needs to be stored in particular index, HashMap checks whether there is already an entry?? If there is no entry already present, Entry object is stored in this location.

If there is already an object sitting on calculated index, its next attribute is checked. If it is null, and current Entry object becomes next node in LinkedList. If next variable is not null, procedure is followed until next is evaluated as null.

What if we add the another value object with same key as entered before. Logically, it should replace the old value. How it is done? Well, after determining the index position of Entry object, while iterating over LinkedList on calculated index, HashMap calls equals method on key object for each Entry object. All these Entry objects in LinkedList will have similar hash code but equals() method will test for true equality.
If **key.equals(k)** will be true then both keys are treated as same key object. This will cause the replacing of value object inside Entry object only.
In this way, HashMap ensure the uniqueness of keys.

How get() methods works internally

Now we have got the idea, how key-value pairs are stored in HashMap. Next big question is : what happens when an object is passed in get method of HashMap? How the value object is determined?

Answer we already should know that the way key uniqueness is determined in put() method , same logic is applied in get() method also. The moment HashMap identify exact match for the key object passed as argument, it simply returns the value object stored in current Entry object.

If no match is found, get() method returns null.

Let have a look at code:

```
/**

* Returns the value to which the specified key is mapped, or {@code null}

* if this map contains no mapping for the key.

*

* <p>
* More formally, if this map contains a mapping from a key {@code k} to a
* value {@code v} such that {@code (key==null ? k==null :
* key.equals(k))}, then this method returns {@code v}; otherwise it returns
* {@code null}. (There can be at most one such mapping.)
*
* </p><p>
* A return value of {@code null} does not <i>necessarily</i> indicate that
* the map contains no mapping for the key; it's also possible that the map
* explicitly maps the key to {@code null}. The {@link #containsKey
* containsKey} operation may be used to distinguish these two cases.
*
* @see #put(Object, Object)
*/
public V get(Object key) {
    if (key == null)
    return getForNullKey();
    int hash = hash(key.hashCode());
    for (Entry<K , V> e = table[indexFor(hash, table.length)]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
            return e.value;
    }
    return null;
}
```

Above code is same as put() method till *if (e.hash == hash && ((k = e.key) == key || key.equals(k)))*, after this simply value object is returned.

Key Notes

1. Data structure to store Entry objects is an array named **table** of type Entry.
2. A particular index location in array is referred as bucket, because it can hold the first element of a LinkedList of Entry objects.
3. Key object's hashCode() is required to calculate the index location of Entry object.
4. Key object's equals() method is used to maintain uniqueness of Keys in map.
5. Value object's hashCode() and equals() method are not used in HashMap's get() and put() methods.
6. Hash code for null keys is always zero, and such Entry object is always stored in zero index in Entry[].

[Update] Improvements in Java 8

As part of the work for JEP 180, there is a performance improvement for HashMap objects where there are lots of collisions in the keys by using balanced trees rather than linked lists to store map entries. The principal idea is that **once the number of items in a hash bucket grows beyond a certain threshold, that bucket will switch from using a linked list of entries to a balanced tree. In the case of high hash collisions, this will improve worst-case performance from O(n) to O(log n)**.
Basically when a bucket becomes too big (**currently: TREEIFY_THRESHOLD = 8**), HashMap dynamically replaces it with an ad-hoc implementation of tree map. This way rather than having pessimistic O(n) we get much better O(log n).
Bins (elements or nodes) of TreeNodes may be traversed and used like any others, but additionally support faster lookup when overpopulated. However, since the vast majority of bins in normal use are not overpopulated, checking for existence of tree bins may be delayed in the course of table methods.

Tree bins (i.e., bins whose elements are all TreeNodes) are ordered primarily by hashCode, but in the case of ties, if two elements are of the same "`class C implements Comparable<C>`", type then their `compareTo()` method is used for ordering.
Because TreeNodes are about twice the size of regular nodes, we use them only when bins contain enough nodes. And when they become too small (due to removal or

resizing) they are converted back to plain bins (**currently: UNTREEIFY_THRESHOLD** = **6**). In usages with well-distributed user hashCodes, tree bins are rarely used.