Popular HashMap and ConcurrentHashMap Interview Questions

June 14, 2013 by Lokesh Gupta

1. Arrange the following in the ascending order (performance):

- HashMap , HashTable , ConcurrentHashMap and Collections.SynchronizedMap
- HashTable < Collections.SynchronizedMap < ConcurrentHashMap < HashMap

HashMap

How HashMap works in Java ?

- This is one of the most important question for java developers. HashMap works on the principle of Hashing .

- It is a data structure which allows us to store object and retrieve it in constant time `O(1)` provided we know the key. In hashing, hash functions are used to link key and value in HashMap. Objects are stored by calling `put(key, value)` method of HashMap and retrieved by calling `get(key)` method. When we call put method, hashcode() method of the key object is called so that hash function of the map can find a bucket location to store value object, which is actually an index of the internal array, known as the table. HashMap internally stores mapping in the form of **`Map.Entry`** object which contains both key and value object. When you want to retrieve the object, you call the get() method and again pass the key object. This time again key object generate same hash code (it's mandatory for it to do so to retrieve the object and that's why HashMap keys are immutable e.g. String) and we end up at same bucket location. If there is only one object then it is returned and that's your value object which you have stored earlier. Things get little tricky when collisions occur. It's easy to answer this question if you have read good books on data structure and algorithms like this one. If you know how hash table data structure works then this is a piece of cake.
- Since the internal array of HashMap is of fixed size, and if you keep storing objects, at some point of time hash function will return same bucket location for two different keys, this is called collision in HashMap. In this case, a linked list is formed at that bucket location and a new entry is stored as next node.
- If we try to retrieve an object from this linked list, we need an extra check to search correct value, this is done by equals() method. Since each node contains an entry, HashMap keeps comparing entry's key object with the passed key using equals() and when it return true, Map returns the corresponding value.
- Since searching inlined list is O(n) operation, in worst case hash collision reduce a map to linked list. This issue is recently addressed in Java 8 by replacing linked list to the tree to search in `O(logN)` time. By the way, you can easily verify how HashMap works by looking at the code of `HashMap.java` in your Eclipse IDE if you know how to attach source code of JDK in Eclipse.
- How HashMap works in Java or sometimes how does get method work in HashMap is a very common question on Java interviews nowadays. Almost everybody who worked in Java knows about HashMap, where to use HashMap and difference between Hashtable and HashMap then why this interview question becomes so special? Because of the depth it offers.

“How HashMap works in java”, “How get and put method of HashMap work internally”.

HashMap in Java works on hashing principle. It is a data structure which allows us to store object and retrieve it in constant time `O(1)` provided we know the key. In hashing, hash functions are used to link key and value in HashMap. Objects are stored by calling `put(key, value)` method of HashMap and retrieved by calling `get(key)` method. When we call put method, hashcode() method of the key object is called so that hash function of the map can find a bucket location to store value object, which is actually an index of the internal array, known as the table. HashMap internally stores mapping in the form of **`Map.Entry`** object which contains both key and value object. When you want to retrieve the object, you call the get() method and again pass the key object. This time again key object generate same hash code (it's mandatory for it to do so to retrieve the object and that's why HashMap keys are immutable e.g. String) and we end up at same bucket location. If there is only one object then it is returned and that's your value object which you have stored earlier. Things get little tricky when collisions occur. It's easy to answer this question if you have read good books on data structure and algorithms like this one. If you know how hash table data structure works then this is a piece of cake.

Since the internal array of HashMap is of fixed size, and if you keep storing objects, at some point of time hash function will return same bucket location for two different keys, this is called collision in HashMap. In this case, a linked list is formed at that bucket location and a new entry is stored as next node.

-

2. How does get method of HashMap work in Java?

HashMap is based upon hash table data structure and uses `hashCode()` method to calculate hash code to find the bucket location on underlying array and `equals()` method to search the object in the same bucket in case of a collision.

3. Which Two Methods HashMap key Object Should Implement?

HashMap is based upon hash table data structure, any object which you want to use as key for HashMap or any other hash based collection e.g. **HashTable**, or ConcurrentHashMap must implement `equals()` and `hashCode()` method.

- `hashCode()` is used to find the bucket location i.e. index of underlying array
- `equals()` method is used to find the right object in linked list stored in the bucket in case of a collision.
- from Java 8, `HashMap` also started using tree data structure to store the object in case of collision to reduce worst case performance of HashMap from `O(n)` to `O(logN)`.

4. Which Two Methods should a HashMap key Object Implement?

- override equals() and hashcode()
- IDE based code generation
- Given that IDEs like NetBeans and Eclipse do such a good job of automatically creating things like getters/setters, constructors, etc., it's no surprise that they can be used to generate equals/hashCode methods as well. Unfortunately, they are not perfect, which prompted me to write this post in the first place.
- When you are in NetBeans and press Ctrl+I, the IDE provides a popup menu with options for methods that it can automatically generate for you.

- When you choose the equals() and hashCode() option, you are presented with the following screen (where the variables will differ depending on your class, obviously).

- After checking all of the checkboxes and pressing generate, the IDE inserts the following two snippets of code:

```
@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final Point other = (Point) obj;
    if (this.x != other.x) {
        return false;
    }
    if (this.y != other.y) {
        return false;
    }
    return true;
}

@Override
public int hashCode() {
    int hash = 3;
    hash = 97 * hash + this.x;
    hash = 97 * hash + this.y;
    return hash;
}
```

5. You need to override equals() and hashcode() methods of a class whose objects you want to use as Key in a hashmap...

- This is required because hashmap uses these 2 methods to retrieve the stored values...
- A hashmap is a datastructure which stores key value pairs of data in array fashion. Lets say a[], where each element in 'a' is a key value pair.
- Also each index in the above array can be linked list thereby having more than one values at one index.
- Now why is a hashmap used? If we have to search among a large array then searching through each if them will not be efficient, so what hash technique tells us that lets pre process the array with some logic and group the elements based on that logic i.e. Hashing
- eg: we have array 1,2,3,4,5,6,7,8,9,10,11 and we apply a hash function mod 10 so 1,11 will be grouped in together. So if we had to search for 11 in previous array then we would have to iterate the complete array but when we group it we limit our scope of iteration thereby improving speed. That datastructure used to store all the above information can be thought of as a 2d array for simplicity
- Now apart from the above hashmap also tells that it wont add any Duplicates in it. And this is the main reason why we have to override the equals and hashcode
- so what put does is that it will first generate the hashcode for the given key to decide which index the value should go in.if nothing is present at that index then the new value will be added over there, if something is already present over there then the new value should be added after the end of the linked list at that index. but remember no duplicates should be added as per the desired behavior of the hashmap. so lets say you have two Integer objects aa=11,bb=11. as every object derived from the object class, the default implementation for comparing two objects is that it compares the reference and not values inside the object. So in the above case both though semantically equal will fail the equality test, and possibility that two objects which same hashcode and same values will exists thereby creating duplicates. If we override then we could avoid adding duplicates.

```
import java.util.HashMap;


public class Employee {

String name:
String mobile:
public Employee(String name,String mobile) {
   this.name=name:
   this.mobile=mobile;
}

@Override
public int hashCode() {
   String str=this.name;
   Integer sum=0;
   for(int i=0;i<str.length();i++){
      sum=sum+str.charAt(i);
   }
   return sum;

}
@Override
public boolean equals(Object obj) {
    Employee emp=(Employee)obj;
   if(this.mobile.equalsIgnoreCase(emp.mobile)){
      return true;
   }else{
     return false;
   }
```

6. When will you need to override equals() ?

The default implementation of Object's equals() is

```
public boolean equals(Object obj) {

      return (this == obj);

}
```

which means two objects will be considered equal only if they have the same memory address which will be true only if you are comparing an object with itself.

But you might want to consider two objects the same if they have the same value for one or more of their properties (

So you will override `equals()` in these situations and you would give your own conditions for equality.

7. Which interface does `java.util.Hashtable` implement? hashmap…hashtable…=map

A. Java.util.Map

B. Java.util.List

C. Java.util.HashTable

D. Java.util.Collection

Correct Answer: Option **A**

Explanation:

Hash table based implementation of the `Map` interface.

8. Why Should an Object Used As the Key should be IMMUTABLE?

- key should be **IMMUTABLE** so that `hashCode()` method always return the same value.
- Since hash code returned by `hashCode()` method depends on upon the content of object i.e. values of member variables.
- If an object is mutable than those values can change and so is the hash code. If the same object returns different hash code once you inserted the value in HashMap, you will end up searching in different bucket location and will not able to retrieve the object. That's why a key object should be IMMUTABLE. It's not a rule enforced by the compiler but you should take care of it as an experienced programmer

9. How to design a good key for HashMap

- 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.
- 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.

10. Which methods do you need to override to use any object as key in HashMap ?

- To use any object as key in HashMap , it needs to implement equals() and hashCode() method .

11. When do you override hashcode and equals() ?

- Whenever necessary especially if you want to do equality check or want to use your object as key in HashMap.

12. What will be the problem if you don't override hashcode() method ?

- You will not be able to recover your object from hash Map if that is used as key in HashMap.

13. Does not overriding hashcode() method has any performance implication ?

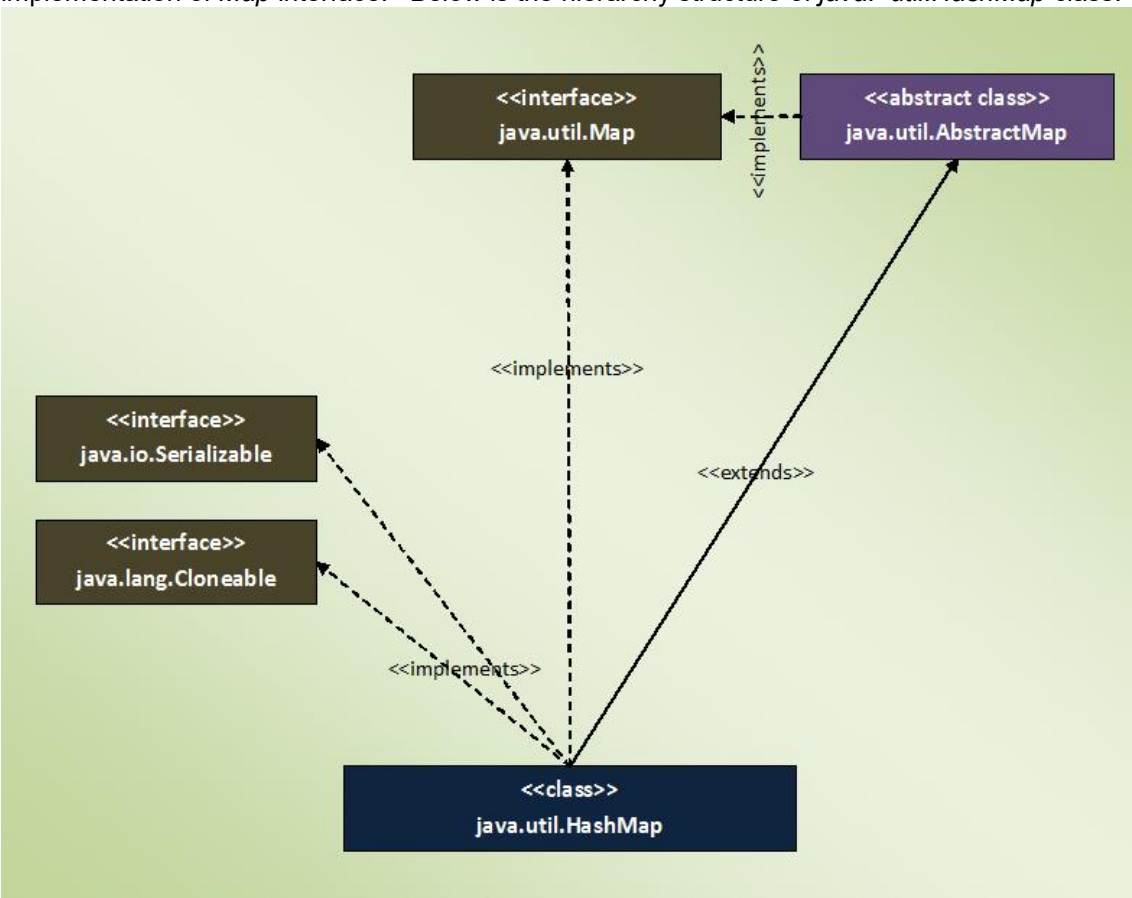
- poor hashcode function will result in frequent collision in HashMap which eventually increase time for adding an object into Hash Map.

pramodbablad December 8, 2015 1

The *java. util.HashMap* is a popular implementation of *Map* interface which holds the data as key-value pairs. *HashMap* extends *AbstractMap* class and implements *Cloneable*and *Serializable* interfaces. In this article, we will discuss about hierarchy of HashMap, properties of HashMap and some important methods of HashMap in java.

14. What is the Hierarchy Of HashMap In Java :

As already said, *HashMap* extends *AbstractMap* class and implements *Cloneable* and *Serializable* interfaces. *AbstractMap* is an abstract class which provides skeletal implementation of *Map* interface. Below is the hierarchy structure of *java. util.HashMap* class.



15. What is the Properties Of HashMap In Java :

1) *HashMap* holds the data in the form of key-value pairs where each key is associated with one value.
2) *HashMap* doesn't allow duplicate keys. But it can have duplicate values.
3) *HashMap* can have multiple null values and only one null key.
4) *HashMap* is not synchronized. To get the synchronized *HashMap*, use *Collections.synchronizedMap()* method.
5) *HashMap* maintains no order.
6) *HashMap* gives constant time performance for the operations like *get()* and *put()* methods.
7) Default initial capacity of *HashMap* is 16.

16. What are the Important Methods Of HashMap In Java :

1) public V put(K key, V value)
This method inserts specified key-value mapping in the map. If map already has a mapping for the specified key, then it rewrites that value with new value.

2) public void putAll(Map m)
This method copies all of the mappings of the map *m* to this map.

3) public V get(Object key)
This method returns the value associated with a specified key.

4) public int size()
This method returns the number of key-value pairs in this map.

5) public boolean isEmpty()
This method checks whether this map is empty or not.

6) public boolean containsKey(Object key)
This method checks whether this map contains the mapping for the specified key.

7) public boolean containsValue(Object value)
This method checks whether this map has one or more keys mapping to the specified value.

8) public V remove(Object key)
This method removes the mapping for the specified key.

9) public void clear()
This method removes all the mappings from this map.

10) public Set<K> keySet()
This method returns the Set view of the keys in the map.

11) public Collection<V> values()
This method returns Collection view of the values in the map.

12) public Set<Map.Entry<K, V>> entrySet()
This method returns the Set view of all the mappings in this map.

13) public V putIfAbsent(K key, V value)
This method maps the given value with specified key if this key is currently not associated with a value or mapped to a null.

13) public boolean remove(Object key, Object value)
This method removes the entry for the specified key if this key is currently mapped to a specified value.

14) public boolean replace(K key, V oldValue, V newValue)
This method replaces the oldValue of the specified key with newValue if the key is currently mapped to oldValue.

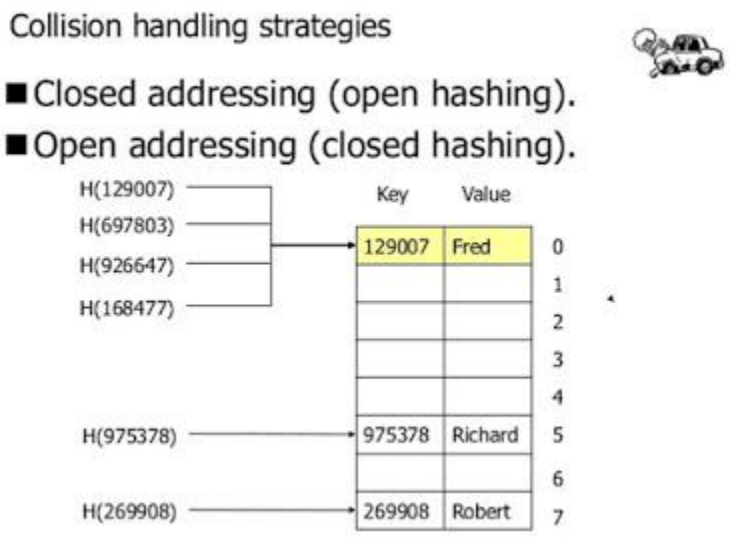
15) public V replace(K key, V value)
This method replaces the current value of the specified key with new value.
(Reference : HashMap Java 8 Documentation)

17. Do you Know how HashMap works in Java or How does get () method of HashMap works in Java

HashMap works on the principle of hashing, we have `put(key, value)` and `get(key)` method for storing and retrieving Objects from HashMap. When we pass Key and Value object to `put()` method on Java HashMap, HashMap implementation calls hashCode method on Key object and applies returned hashcode into its own hashing function to find a bucket location for storing Entry object, important point to mention is that HashMap in Java stores both key and value object as `Map.Entry` in a bucket which is essential to understand the retrieving logic.

18. When does collision occur in HashMap

- equals() and hashCode() contract that two unequal objects in Java can have same hashcode
- hash function will return same bucket location for two different keys, this is called collision
- two different key objects using hash function which is hashCode() of key object and equals() will return same values.



Summary

1) HashMap handles collision by using linked list to store map entries ended up in same array location or bucket location.

2) From Java 8 onwards,

HashMap, ConcurrentHashMap, and LinkedHashMap

will use the balanced tree in place of linked list to handle frequently hash collisions.

The idea is to switch to the balanced tree once the number of items in a hash bucket grows beyond a certain threshold. This will improve the worst case get() method performance from O(n) to O(log n).

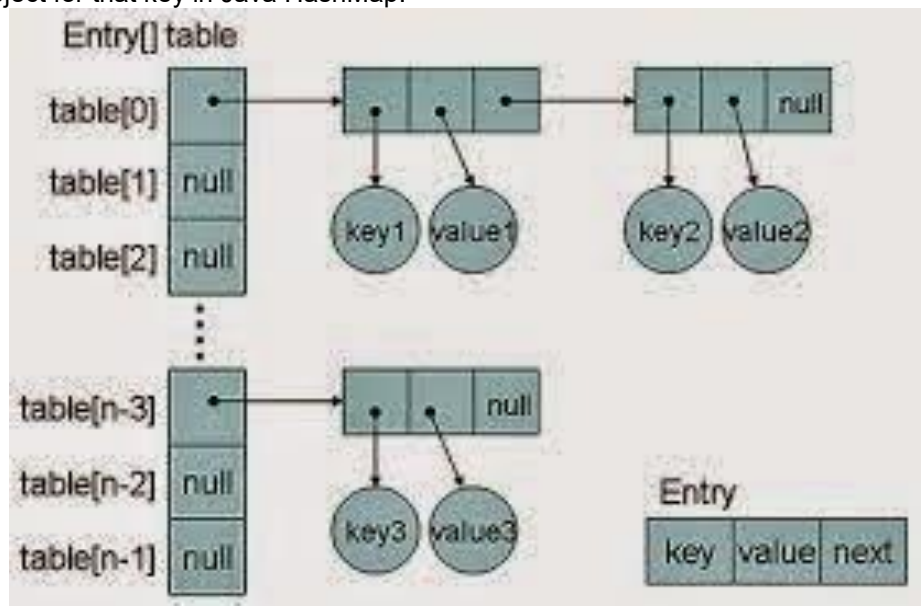
5) Apart from Hashtable, WeakHashMap and IdentityHashMap will also continue to use the linked list for handling collision even in the case of frequent collisions.

19. How will you retrieve Value object if two Keys will have the same hashcode?

Interviewee will say we will call `get()` method and then HashMap uses Key Object's hashcode to find out bucket location and retrieves Value object but then you need to remind him that there are two Value objects are stored in

same bucket , so they will say about traversal in LinkedList until we find the value object ,

then you ask *how do you identify value object because you don't have value object to compare* ,Until they know that after finding bucket location, we will **call keys.equals() method** to identify a correct node in LinkedList and return associated value object for that key in Java HashMap.



20. "What happens On HashMap in Java if the size of the HashMap exceeds a given threshold defined by load factor ?".

If the size of the Map exceeds a given threshold defined by load-factor e.g. if the load factor is `.75` it will act to re-size the map once it filled 75%. Similar to other collection classes like ArrayList, Java HashMap re-size itself by creating a new bucket array of size twice of the previous size of HashMap and then start putting every old element into that new bucket array. This process is called `rehashing` because it also applies the hash function to find new bucket location.

21. "do you see any problem with resizing of HashMap in Java"

hint about multiple thread accessing the Java `HashMap` and potentially looking for **race condition on HashMap in Java**.

So the answer is Y there is potential race condition exists while resizing `HashMap` in Java, if two thread at the same time found that now `HashMap` needs resizing and they both try to resizing. on the process of resizing of `HashMap` in Java, the element in the bucket which is stored in linked list get reversed in order during their migration to new bucket because Java HashMap doesn't append the new element at tail instead it append new element at the head *to avoid tail traversing*. If race condition happens then you will end up with an infinite loop. Though this point, you can potentially argue that what the hell makes you think to use HashMap in multi-threaded environment to interviewer :)

22. Why String, Integer and other wrapper classes are considered good keys?

`String`, `Integer` and other wrapper classes are natural candidates of `HashMap` key, and `String` is most frequently used key as well because String is immutable and final, and overrides `equals` and `hashcode()` method. Other wrapper class also shares similar property. Immutability is required, in order to prevent changes on fields used to calculate `hashCode()` because if key object returns different hashCode during insertion and retrieval than it won't be possible to get an object from `HashMap`.

Immutability is best as it offers other advantages as well like thread-safety, If you can keep your hashCode same

by only making certain fields final, then you go for that as well. Since `equals()` and `hashCode()` method is used during retrieval of value object from `HashMap` Javarevisited blog

23. Can we use any custom object as a key in HashMap?

This is an extension of previous questions. Of course you can use any `Object` as key in Java `HashMap` provided it follows `equals and hashCode contract` and its hashCode should not vary once the object is inserted into Map. If the custom object is Immutable than this will be already taken care because you can not change it once created.

24. Give a Java HashMap Example :

```
import java.  util.HashMap;
import java.  util.Iterator;
import java.  util.Set;

public class JavaHashMapExample
{
    public static void main(String[] args)
    {
        //Defining the HashMap

        HashMap<String, Double> map = new HashMap<String, Double>();

        //Adding some elements to HashMap

        map.put("Ashwin", 87.55);

        map.put("Bharat", 95.65);

        map.put("Chetan", 68.13);

        map.put("Dhanjay", 74.23);

        map.put("Kartik", 65.42);

        //HashMap can have one null key and multiple null values

        map.put(null, null);

        map.put("Sandesh", null);

        //Getting the size of the map

        System.out.println("Size Of The Map : "+map.size());

        System.out.println("-----------------");

        //Displaying the elements

        System.out.println("The elements are :");

        Set set = map.keySet();

        Iterator keySetIterator = set.iterator();

        while (keySetIterator.hasNext())
        {
            Object key = keySetIterator.next();

            System.out.println(key+"  : "+map.get(key));
        }

        System.out.println("-----------------");
```

```
        //Checking the map for a particular key/value

        System.out.println("Does this map has Chetan as key?
"+map.containsKey("Chetan"));

        System.out.println("Does this map has 74.23 as value?
"+map.containsValue(74.23));

        System.out.println("-----------------");

        //Removing an element from the map

        System.out.println("Value removed from the map : "+map.remove("Kartik"));
    }
}
```

Output :

Size Of The Map : 7

———————

The elements are :

null : null

Ashwin : 87.55

Dhanjay : 74.23

Chetan : 68.13

Bharat : 95.65

Kartik : 65.42

Sandesh : null

———————

Does this map has Chetan as key? true

Does this map has 74.23 as value? true

———————

Value removed from the map : 65.4

25. Creating HashMap with default initial capacity and load factor

```
HashMap<String, Integer> map1 = new HashMap<String, Integer>();
```

26. Creating HashMap with 30 as initial capacity

```
HashMap<String, Integer> map2 = new HashMap<String, Integer>(30);
```

27. Creating HashMap with 30 as initial capacity and 0.5 as load factor

```
HashMap<String, Integer> map3 = new HashMap<String, Integer>(30, 0.5f);
```

28. Creating HashMap by copying another HashMap

```
HashMap<String, Integer> map4 = new HashMap<String, Integer>(map1);
```

29. How do you add key-value pairs to HashMap?

By using put() and putAll() methods. put() method adds key-value pair one by one where as putAll() method copies all key-value pairs from one HashMap to another HashMap.

```
import java.  util.HashMap;
import java.  util.Map.Entry;
import java.  util.Set;

public class JavaHashMapPrograms
{
    public static void main(String[] args)
    {
        //Creating HashMap with default initial capacity and load factor

        HashMap<String, Integer> map = new HashMap<String, Integer>();

        //Inserting key-value pairs to map using put() method

        map.put("ONE", 1);

        map.put("TWO", 2);

        map.put("THREE", 3);

        map.put("FOUR", 4);

        map.put("FIVE", 5);

        //Printing key-value pairs

        Set<Entry<String, Integer>> entrySet = map.entrySet();

        for (Entry<String, Integer> entry : entrySet)
        {
            System.out.println(entry.getKey()+" : "+entry.getValue());
        }

        System.out.println("-------------------------");

        //Creating another HashMap

        HashMap<String, Integer> anotherMap = new HashMap<String, Integer>();

        //Inserting key-value pairs to anotherMap using put() method

        anotherMap.put("SIX", 6);

        anotherMap.put("SEVEN", 7);

        //Inserting key-value pairs of map to anotherMap using putAll() method

        anotherMap.putAll(map);

        //Printing key-value pairs of anotherMap

        entrySet = anotherMap.entrySet();

        for (Entry<String, Integer> entry : entrySet)
        {
            System.out.println(entry.getKey()+" : "+entry.getValue());
        }
    }
}
```

Output :

FIVE : 5
ONE : 1

FOUR : 4
TWO : 2
THREE : 3

FIVE : 5
SIX : 6
ONE : 1
FOUR : 4
TWO : 2
SEVEN : 7
THREE :

30. How do you add given key-value pair to HashMap if and only if it is not present in the HashMap?

Using putIfAbsent() method.

```
import java.  util.HashMap;
import java.  util.Map.Entry;
import java.  util.Set;

public class HashMapExampleThree
{
    public static void main(String[] args)
    {
        //Creating HashMap with default initial capacity and load factor

        HashMap<String, Integer> map = new HashMap<String, Integer>();

        //Adding key-value pairs

        map.put("ONE", 1);

        map.put("TWO", 2);

        map.put("THREE", 3);

        map.put("FOUR", 4);

        //Adds key-value pair 'ONE-111' only if it is not present in map

        map.putIfAbsent("ONE", 111);

        //Adds key-value pair 'FIVE-5' only if it is not present in map

        map.putIfAbsent("FIVE", 5);

        //Printing key-value pairs of map

        Set<Entry<String, Integer>> entrySet = map.entrySet();

        for (Entry<String, Integer> entry : entrySet)
        {
            System.out.println(entry.getKey()+" : "+entry.getValue());
        }
    }
}
```

Output :

FIVE : 5
ONE : 1
FOUR : 4
TWO : 2

THREE :

31. How do you retrieve a value associated with a given key from the HashMap?

Using get() method.

```
import java.  util.HashMap;

public class HashMapExampleFour
{
    public static void main(String[] args)
    {
        //Creating HashMap with default initial capacity and load factor

        HashMap<String, Integer> map = new HashMap<String, Integer>();

        //Adding key-value pairs to HashMap

        map.put("ONE", 1);

        map.put("TWO", 2);

        map.put("THREE", 3);

        map.put("FOUR", 4);

        //Retrieving a value associated with key 'TWO'

        int value = map.get("TWO");

        System.out.println(value);       //Output :     }
}
```

32. How do you check whether a particular key/value exist in a HashMap?

Using containsKey() and containsValue() methods.

```
import java.  util.HashMap;

public class ExampleFive
{
    public static void main(String[] args)
    {
        //Creating the HashMap

        HashMap<Integer, Double> map = new HashMap<Integer, Double>();

        //Adding key-value pairs to HashMap

        map.put(1, 1.1);

        map.put(2, 2.2);

        map.put(3, 3.3);

        map.put(4, 4.4);

        //Checking whether key '3' exist in map

        System.out.println(map.containsKey(3));      //Output : true

        //Checking whether value '3.3' exist in map

        System.out.println(map.containsValue(3.3));   //Output : true
```

```
    }
}
```

33. How do you find out the number of key-value mappings present in a HashMap?

Using size() method.

```
import java.  util.HashMap;

public class JavaHashMapPrograms
{
    public static void main(String[] args)
    {
        //Creating the HashMap

        HashMap<Integer, Double> map = new HashMap<Integer, Double>();

        //Adding key-value pairs to HashMap

        map.put(111, 111.111);

        map.put(222, 222.222);

        map.put(333, 333.333);

        map.put(444, 444.444);

        map.put(555, 555.555);

        //Retrieving the number of key-value pairs present in map

        System.out.println(map.size());      //Output :     }
}
```

34. How do you remove all key-value pairs from a HashMap? OR How do you clear the HashMap for reuse?

using clear() method.

```
import java.  util.HashMap;

public class JavaHashMapExampleSeven
{
    public static void main(String[] args)
    {
        //Creating the HashMap

        HashMap<Integer, Double> map = new HashMap<Integer, Double>();

        //Adding key-value pairs to HashMap

        map.put(111, 111.111);

        map.put(222, 222.222);

        map.put(333, 333.333);

        map.put(444, 444.444);

        map.put(555, 555.555);

        //Retrieving the number of key-value pairs

        System.out.println(map.size());      //Output :
```

```
        //Clearing the map

        map.clear();

        //Checking the number of key-value pairs after clearing the map

        System.out.println(map.size());      //Output :     }
}
```

35. How do you retrieve all keys present in a HashMap?

keySet() method returns all keys present in a HashMap in the form of Set.

```
import java.  util.HashMap;
import java.  util.Set;

public class JavaHashMapExample
{
    public static void main(String[] args)
    {
        //Creating the HashMap

        HashMap<Integer, String> map = new HashMap<Integer, String>();

        //Adding key-value pairs to HashMap

        map.put(1, "AAA");

        map.put(2, "BBB");

        map.put(3, "CCC");

        map.put(4, "DDD");

        map.put(5, "EEE");

        //Retrieving the Key Set

        Set<Integer> keySet = map.keySet();

        for (Integer key : keySet)
        {
            System.out.println(key);
        }
    }
}
```

Output :

1

2

3

4

36. How do you retrieve all the values present in a HashMap?

Using values() method. This method returns Collection view of all the values present in a HashMap.

```
import java.  util.Collection;
import java.  util.HashMap;

public class HashMapExampleNine
{
    public static void main(String[] args)
```

```
    {
        //Creating the HashMap

        HashMap<Integer, String> map = new HashMap<Integer, String>();

        //Adding key-value pairs to HashMap

        map.put(1, "AAA");

        map.put(2, "BBB");

        map.put(3, "CCC");

        map.put(4, "DDD");

        map.put(5, "EEE");

        //Retrieving the Collection view of values present in map

        Collection<String> values = map.values();

        for (String value : values)
        {
            System.out.println(value);
        }
    }
}
```

Output :

AAA
BBB
CCC
DDD
EEE

37. How do you retrieve all key-value pairs present in a HashMap?

entrySet() method returns all key-value pairs present in a HashMap in the form of Set.

```
import java.  util.HashMap;
import java.  util.Map.Entry;
import java.  util.Set;

public class JavaHashMapPrograms
{
    public static void main(String[] args)
    {
        //Creating the HashMap

        HashMap<String, String> map = new HashMap<String, String>();

        //Adding key-value pairs to HashMap

        map.put("ONE", "AAA");

        map.put("TWO", "BBB");

        map.put("THREE", "CCC");

        map.put("FOUR", "DDD");

        map.put("FIVE", "EEE");

        //Retrieving the Set consists of all key-value pairs in map
```

```
        Set<Entry<String, String>> keyValueSet = map.entrySet();

        for (Entry<String, String> entry : keyValueSet)
        {
            System.out.println(entry.getKey()+" : "+entry.getValue());
        }
    }
}
```

Output :

FIVE : EEE

ONE : AAA

FOUR : DDD

TWO : BBB

THREE : CCC

38. How do you remove a key-value pair from the HashMap?

remove

map.remove("ONE");

```
mport java.  util.HashMap;
import java.  util.Map.Entry;
import java.  util.Set;

public class MainClass
{
    public static void main(String[] args)
    {
        //Creating the HashMap

        HashMap<String, String> map = new HashMap<String, String>();

        //Adding key-value pairs to HashMap

        map.put("ONE", "AAA");

        map.put("TWO", "BBB");

        map.put("THREE", "CCC");

        map.put("FOUR", "DDD");

        map.put("FIVE", "EEE");

        //Printing key-value pairs

        System.out.println("HashMap Before Remove :");

        Set<Entry<String, String>> keyValueSet = map.entrySet();

        for (Entry<String, String> entry : keyValueSet)
        {
            System.out.println(entry.getKey()+" : "+entry.getValue());
        }

        System.out.println("---------------------");

        //Removing the mapping for the key 'ONE'

        map.remove("ONE");

        System.out.println("HashMap After Remove :");
```

```
        for (Entry<String, String> entry : keyValueSet)
        {
            System.out.println(entry.getKey()+" : "+entry.getValue());
        }
    }
}
```

Output :

HashMap Before Remove :

FIVE : EEE

ONE : AAA

FOUR : DDD

TWO : BBB

THREE : CCC

———————

HashMap After Remove :

FIVE : EEE

FOUR : DDD

TWO : BBB

THREE : CCC

39. How do you remove a key-value pair from a HashMap if and only if the specified key is currently mapped to given value?

Another version of remove() method which takes two arguments – one is key and another one is value, removes the mapping for the specified key only if it is currently mapped to given value.

```
import java.  util.HashMap;
import java.  util.Map.Entry;
import java.  util.Set;

public class JavaHashMapExample
{
    public static void main(String[] args)
    {
        //Creating the HashMap

        HashMap<String, String> map = new HashMap<String, String>();

        //Adding key-value pairs to HashMap

        map.put("ONE", "AAA");

        map.put("TWO", "BBB");

        map.put("THREE", "CCC");

        map.put("FOUR", "DDD");

        map.put("FIVE", "EEE");

        //Printing Key-value pairs

        System.out.println("HashMap Before Remove :");

        Set<Entry<String, String>> keyValueSet = map.entrySet();

        for (Entry<String, String> entry : keyValueSet)
        {
```

```
            System.out.println(entry.getKey()+" : "+entry.getValue());
        }

        System.out.println("------------------");

        //Removes the mapping for the key 'ONE' only if it is currently mapped to
'CCC'

        map.remove("ONE", "CCC");

        //Removes the mapping for the key 'FIVE' only if it is currently mapped to
'EEE'

        map.remove("FIVE", "EEE");

        System.out.println("HashMap After Remove :");

        for (Entry<String, String> entry : keyValueSet)
        {
            System.out.println(entry.getKey()+" : "+entry.getValue());
        }
    }
}
```

Output :

HashMap Before Remove :

FIVE : EEE

ONE : AAA

FOUR : DDD

TWO : BBB

THREE : CCC

———————

HashMap After Remove :

ONE : AAA

FOUR : DDD

TWO : BBB

THREE : CCC

40. How do you replace a value associated with a given key in the HashMap?

replace() method replaces the value associated with the specified key if the key is currently mapped to some value.

```
import java.  util.HashMap;
import java.  util.Map.Entry;
import java.  util.Set;

public class JavaHashMapPrograms
{
    public static void main(String[] args)
    {
        //Creating the HashMap

        HashMap<String, String> map = new HashMap<String, String>();

        //Adding key-value pairs to HashMap

        map.put("ONE", "AAA");
```

```
        map.put("TWO", "BBB");

        map.put("THREE", "CCC");

        map.put("FOUR", "DDD");

        map.put("FIVE", "EEE");

        //Printing Key-value pairs

        System.out.println("HashMap Before Replace :");

        Set<Entry<String, String>> keyValueSet = map.entrySet();

        for (Entry<String, String> entry : keyValueSet)
        {
            System.out.println(entry.getKey()+" : "+entry.getValue());
        }

        System.out.println("------------------");

        //Replacing the value associated with 'THREE' to '333'

        map.replace("THREE", "333");

        System.out.println("HashMap After Replace :");

        for (Entry<String, String> entry : keyValueSet)
        {
            System.out.println(entry.getKey()+" : "+entry.getValue());
        }
    }
}
```

Output :

HashMap Before Replace :
FIVE : EEE
ONE : AAA
FOUR : DDD
TWO : BBB
THREE : CCC

HashMap After Replace :
FIVE : EEE
ONE : AAA
FOUR : DDD
TWO : BBB

41. How do you replace a value associated with the given key if and only if it is currently mapped to given value?

Another version of replace() method which takes three arguments, replaces the value associated with the given key only if it is currently mapped to given value.

```
import java.  util.HashMap;
import java.  util.Map.Entry;
import java.  util.Set;

public class JavaHashMapExample
{
```

```
    public static void main(String[] args)
    {
        //Creating the HashMap

        HashMap<String, String> map = new HashMap<String, String>();

        //Adding key-value pairs to HashMap

        map.put("ONE", "AAA");

        map.put("TWO", "BBB");

        map.put("THREE", "CCC");

        map.put("FOUR", "DDD");

        map.put("FIVE", "EEE");

        //Printing Key-value pairs

        System.out.println("HashMap Before Replace :");

        Set<Entry<String, String>> keyValueSet = map.entrySet();

        for (Entry<String, String> entry : keyValueSet)
        {
            System.out.println(entry.getKey()+" : "+entry.getValue());
        }

        System.out.println("------------------");

        //Replacing the value associated with 'FOUR' to '444' only if it is
currently mapped to 'DDD'

        map.replace("FOUR", "DDD", "444");

        System.out.println("HashMap After Replace :");

        for (Entry<String, String> entry : keyValueSet)
        {
            System.out.println(entry.getKey()+" : "+entry.getValue());
        }
    }
}
```

Output :

HashMap Before Replace :

FIVE : EEE

ONE : AAA

FOUR : DDD

TWO : BBB

THREE : CCC

HashMap After Replace :

FIVE : EEE

ONE : AAA

FOUR : 444

TWO : BBB

THREE : CCC

42. What is the number of key mappings in this example

```
class MyKeys {
Integer key;
MyKeys(Integer k) {
key = k;
}
public boolean equals(Object o) {
return ((MyKeys) o).key == this.key;
}
}
Map m = new HashMap();
MyKeys m1 = new MyKeys(1);
MyKeys m2 = new MyKeys(2);
MyKeys m3 = new MyKeys(1);
MyKeys m4 = new MyKeys(new Integer(2));
m.put(m1, "car");
m.put(m2, "boat");
m.put(m3, "plane");
m.put(m4, "bus");
System.out.print(m.size()); //number of key-value mappings in this map
```

What is the result?

A) 2

B) 3

C) 4

D) Compilation fails.

43. What is the number of key mappings in 2nd example

```
package com.tutorialspoint;

import java.util.*;

public class HashMapDemo {
   public static void main(String args[]) {
   // create hash map
   HashMap newmap = new HashMap();

   // populate hash map
   newmap.put(1, "tutorials");
   newmap.put(2, "point");
   newmap.put(3, "is best");

   System.out.println("Size of the map: "+ newmap.size());
   }
}
```

Let us compile and run the above program, this will produce the following result.

```
Size of the map: 3

public class MyClass {
     public static void main(String[] args) {
        class MyKeys {
            Integer key;
            MyKeys(Integer k) {
               key = k;
            }
            public boolean equals(Object o) {
                return ((MyKeys) o).key == this.key;
            }
        }
        Map m = new HashMap();
        MyKeys m1 = new MyKeys(1);
```

```
        MyKeys m2 = new MyKeys(2);
        MyKeys m3 = new MyKeys(1);
        MyKeys m4 = new MyKeys(new Integer(2));
        m.put(m1, "car");
        m.put(m2, "boat");
        m.put(m3, "plane");
        m.put(m4, "bus");
        System.out.println(m.size());
    }

}
run = 4
```

44. How null key is handled in HashMap? Since equals() and hashCode() are used to store and retrieve values, how does it work in case of the null key?

The null key is handled specially in HashMap, there are two separate methods for that `putForNullKey(V value)` and `getForNullKey()`. Later is offloaded version of get() to look up null keys. Null keys always map to index 0. This null case is split out into separate methods for the sake of performance in the two most commonly used operations (get and put), but incorporated with conditionals in others. In short, `equals()` and `hashcode()` method are not used in case of null keys in HashMap.

here is how nulls are retrieved from HashMap

```
private V getForNullKey() {
        if (size == 0) {
            return null;
        }
        for (Entry<K,V> e = table[0]; e != null; e = e.next) {
            if (e.key == null)
                return e.value;
        }
        return null;
    }
```

In terms of usage, Java HashMap is very versatile and I have mostly used HashMap as cache in an electronic trading application I have worked. Since finance domain used Java heavily and due to performance reason we need caching HashMap and ConcurrentHashMap comes as very handy there. You can also check following articles from Javarevisited to learn more about HashMap and Hashtable in Java:

45. HashMap Changes in JDK 1.7 and JDK 1.8

- reduce memory consumption. Due to this
- empty Map are lazily initialized and will cost you less memory.
- Earlier, when you create HashMap e.g. new HashMap() it automatically creates an array of default length e.g. 16.
- From JDK 1.8 onwards HashMap has introduced an improved strategy to deal with high collision rate. Since a poor hash function e.g. which always return location of same bucket, can turn a HashMap into linked list, i.e. converting get() method to perform in O(n) instead of O(1) and someone can take advantage of this fact, Java now internally replace linked list to a binary true once certain threshold is breached. This ensures performance or order O(log(n)) even in the worst case where a hash function is not distributing keys properly. javarevisited.blogspot.com

Set (I) TreeSet implements NavigableSet extednds SortedSet extends Set

Ex. example that accumulates a `Collection` of names into a `TreeSet`:

```
Set<String> set = people.stream()
.map(Person::getName)
```

- `.collect(Collectors.toCollection(TreeSet::new));`

Program for TreeSet:

```
package.com.instanceofjava;import.java.util.TreeSet;
Public
class testSet2 {

   public static void main(String args[]){
      TreeSet treeSet=new treeSet();
      treeSet.add("Sindhu");
      treeSet.add("Sindhu");
      treeSet.add("Indhu");
      treeSet.add("Bindhu');
            Iterator it=treeSet.iterator();
            while(it.hasNext()){
               System.out.println(it.next);
            }
   }
}

Output:

   Bindhu Indhu Sindhu
```

46. Difference between HashMap and HashTable : Popular Interview Question in Java with

- HashMap is not SYNCHRONIZED
- HashTable or hashmap faster.
- **HashTable** includes five point namely Synchronization,Null keys and values,Iterating values , Fail fast iterator ,Performance,Superclass .

47. What is default size of ArrayList and HashMap in Java?

As of Java 7 now, default size of ArrayList is 10 and default capacity of HashMap is 16, it must be power of 2. Here is code snippet from ArrayList and HashMap class :

```
// from ArrayList.java JDK 1.7
private static final int DEFAULT_CAPACITY = 10;
//from HashMap.java JDK 7
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
```

48. Is it possible for two unequal objects to have the same hashcode?

- Y, two unequal objects can have same hashcode that's why collision happen in a hashmap.
- the equal hashcode contract only says that two equal objects must have the same hashcode it doesn't say anything about the unequal object.

49. Can two equal object have the different hash code?

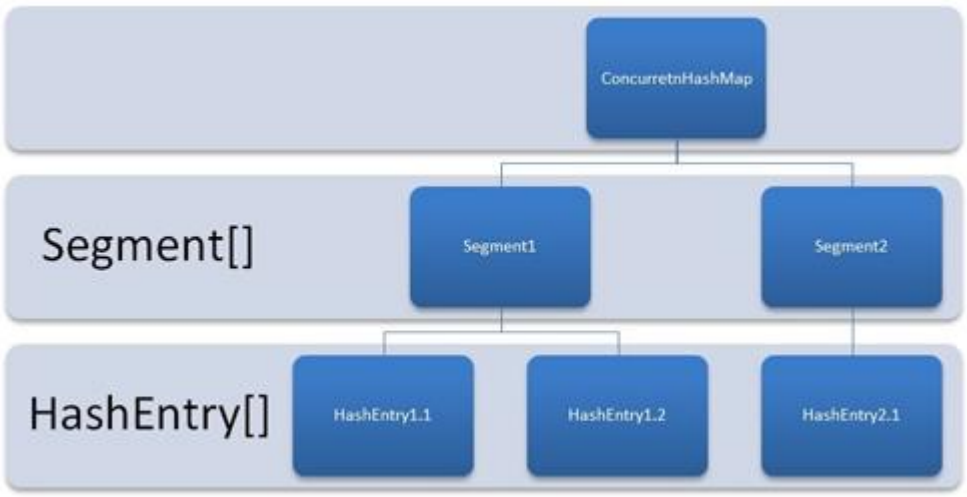
- No, thats not possible according to hash code contract.

50. Can we use random numbers in the hashcode() method?

- No, because hashcode of an object should be always same. See the answer to learning more about things to remember while overriding hashCode() method in Java.

HashMap and ConcurrentHashMap

51. 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.
- 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. - What is Read-Write Lock?
  - ReadWrite Lock is an implementation of *lock stripping* technique, where two separate locks are used for read and write operation. Since read operation doesn't modify the state of the object, it's safe to allow multiple thread access to a shared object for reading without locking, and by splitting one lock into read and write lock, you can easily do that.
  - Java provides an implementation of read-write lock in the form of `ReentrantReadWritLock` class in the java.util.concurrent.lock package. This is worth looking before you decide to write your own read-write locking implementation.

52. Does ConcurrentHashMap in Java Use The ReadWrite Lock?

- Also, the current implementation of `java.util.ConcurrentHashMap` doesn't use the `ReadWriteLock`, instead, it divides the Map into several segments and locks them separately using different locks. This means any given time, *only a portion of the ConcurrentHashMap is locked*, instead of the whole Map.

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

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

54. 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.

55. 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 &amp;amp;&amp;amp; eq(k, e.key))
            return e.value;
        e = e.next;
    }
}
```

HashSet

56. HashSet is implemented in Java with HashMap, How does it uses Hashing ?

- for hashing you need both key and value and there is no key for store it in a bucket, then how exactly `HashSet` store element internally.
- Well, `HashSet` is built on top of HashMap.
- it uses a HashMap with same values for all keys, as shown below :

```
private transient HashMap map;
// Dummy value to associate with an Object in the backing Map
private static final Object PRESENT = new Object();
```

When you call add() method of HashSet, it put entry in HashMap :

```
public boolean add(E e) {
  return map.put(e, PRESENT)==null;
}
```

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

- If you are using any custom object in Map as key, you need to override `equals()` and `hashCode()` method, and make sure they follow there 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 those collection will not follow there contacts e.g. Set may allow duplicates.

58. Program for HashSet:

```
package.com.instanceofjava;
import.java.util.HashSet;

public class A{
   public static void main(String args[]){

      HashSet hashset=new HashSet();

      hashset.add("Indhu");
      hashset.add("Indhu");
      hashset.add("Sindhu");
      hashset.add("swathi");
      hashset.add(null);
      hashset.add(null);
      hashset.add("Lavs");
      Iterator it=hashset.iterator();
      while(it.hasNext()){
         System.out.println(it.next);
      }
   }
}
Output:
   Indhu
   null
   Lavs
   Sindhu
   swathi
```

59. Constructors of HashSet:

1.HashSet( )

- Creates an empty HashSet object with default initial capacity 16. Fill ratio or load factor 0.75

2.HashSet(Collection obj)

- This constructor initializes the hash set by using the elements of the collection **obj**.

3.HashSet(int capacity)

- Creates an empty HashSet object with given capacity.

4.HashSet(int capacity, float fillRatio)

- This constructor initializes both the capacity and the fill ratio (also called load capacity) of the hash set from its arguments (fill ratio 0.1 to 1.0)

60. HashSet implements Set Example

The following code segment demonstrates how to perform many `Collection` operations on `HashSet`, a basic collection that implements the `Set` interface. A `HashSet` is simply a set that doesn't allow duplicate elements and doesn't order or position its elements. The code shows how you create a basic collection and add, remove, and test for elements. Because `Vector` now supports the `Collection` interface, you can also execute this code on a vector, which you can test by changing the `HashSet` declaration and constructor to a `Vector`.

```
import java.util.collections.*;
public class CollectionTest {
   // Statics
   public static void main( String [] args ) {
      System.out.println( "Collection Test" );
      // Create a collection
      HashSet collection = new HashSet();
      // Adding
      String dog1 = "Max", dog2 = "Bailey", dog3 = "Harriet";
      collection.add( dog1 );
      collection.add( dog2 );
      collection.add( dog3 );
      // Sizing
      System.out.println( "Collection created" +
            ", size=" + collection.size() +
            ", isEmpty=" + collection.isEmpty() );
      // Containment
      System.out.println( "Collection contains " + dog3 +
            ": " + collection.contains( dog3 ) );
      // Iteration. Iterator supports hasNext, next, remove
      System.out.println( "Collection iteration (unsorted):" );
      Iterator iterator = collection.iterator();
      while ( iterator.hasNext() )
         System.out.println( "   " + iterator.next() );
      // Removing
      collection.remove( dog1 );
      collection.clear();
   }
}
```

The following `program` prints out all distinct words in its argument list.

The first uses JDK 8 aggregate operations.

```
import java.util.*;
import java.util.stream.*;

public class FindDups {
    public static void main(String[] args) {
        Set<String> distinctWords = Arrays.asList(args).stream()
      .collect(Collectors.toSet());
```

```
        System.out.println(distinctWords.size()+
                           " distinct words: " +
                           distinctWords);
    }
}
```

The second uses the for-each construct.

```
Using the for-each Construct:
import java.util.*;

public class FindDups {
    public static void main(String[] args) {
        Set<String> s = new HashSet<String>();
        for (String a : args)
            s.add(a);
            System.out.println(s.size() + " distinct words: " + s);
    }
}
```

Now run either version of the program.

```
java FindDups i came i saw i left
```

The following output is produced:

```
4 distinct words: [left, came, saw, i]
```

Note that the code always refers to the `Collection` by its INTERFACE type (`Set`) rather than by its implementation type. This is a *strongly* recommended programming practice because it gives you the flexibility to change implementations merely by changing the constructor. If either of the variables used to store a collection or the parameters used to pass it around are declared to be of the `Collection`'s implementation type rather than its INTERFACE type, *all* such variables and parameters must be changed in order to change its implementation type.

61. Difference between HashMap and HashTable

1. **Superclass and Legacy :**

HashTable is a subclass of Dictionary class which is now obsolete in Jdk 1.7 ,so ,it is not used anymore. It is better off externally synchronizing a HashMap or using a ConcurrentMap implementation (e.g ConcurrentHashMap).HashMap is the subclass of the AbstractMap class. Although **HashTable** and HashMap has different superclasses but they both are implementations of the *"Map"* abstract data type.

Example of HashMap and **HashTable**

```
import java.util.HashTable;
public class HashMapHashTableExample {

    public static void main(String[] args) {

   HashTable<String,String> HashTableobj = new HashTable<String, String>();
   HashTableobj.put("Alive is ", "awesome");
   HashTableobj.put("Love", "yourself");
   System.out.println("HashTable object output :"+ HashTableobj);



   HashMap hashmapobj = new HashMap();
   hashmapobj.put("Alive is ", "awesome");
   hashmapobj.put("Love", "yourself");
   System.out.println("HashMap object output :"+hashmapobj);

 }
}
```

Output :

HashTable object output :{Love=yourself, Alive is =awesome}

HashMap object output :{Alive is =awesome, Love=yourself}

62. Which collection class allows you to access its elements by associating a key with an element's value, and provides synchronization?

A. java. util.SortedMap

B. java. util.TreeMap

C. java. util.TreeSet

D. java. util.Hashtable

Correct Answer: Option **D**

Explanation:

`Hashtable` is the only class listed that provides synchronized methods. If you need synchronization great; otherwise, use `HashMap`, it's faster.

63. When to use HashMap and HashTable?

1. Single Threaded Application

- HashMap should be preferred over **HashTable** for the non-threaded applications. In simple words , use

HashMap in un**SYNCHRONIZED** or single threaded applications .

2. Multi Threaded Application
   - We should avoid using **HashTable**, as the class is now obsolete in latest Jdk 1.8 . Oracle has provided a better replacement of **HashTable** named ConcurrentHashMap. For multithreaded application prefer ConcurrentHashMap instead of **HashTable**.
   -

64. What is the difference between HashSet and TreeSet ?

- Main differences between HashSet and TreeSet are :
- a. HashSet maintains the inserted elements in random order while TreeSet maintains elements in the sorted order
- b. HashSet can store null object while TreeSet can not store null object.

65. What is the difference between Fail-fast iterator and Fail-safe iterator ? - Advanced

- Fail-fast throw ConcurrentModificationException while Fail-safe does not.
- **Fail-fast does not clone the original collection** list of objects while
- **Fail-safe creates a copy of the original collection list of objects**.

66. When do you use ConcurrentHashMap in Java?

- ConcurrentHashMap is better suited for situation where you have multiple readers and one
- Writer or fewer writers since Map gets locked only during write operation. If you have equal number of reader and writer than ConcurrentHashMap will perform in line of HashTable or SYNCHRONIZED HashMap.

67. 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 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

68. Why ConcurrentHashMap is faster than HashTable in Java?

- `ConcurrentHashMap` is introduced as alternative of **HashTable** in Java 5; it is faster because of its design. ConcurrentHashMap divides whole map into different segments and only lock a particular segment during update operation, instead of **HashTable**, which locks whole Map.
- `ConcurrentHashMap` also provides lock free read which is not possible in **HashTable**, because of this and lock striping; ConcurrentHashMap is faster than **HashTable**, especially when number of reader is more than number of writers. In order to better answer this popular Java concurrency interview questions, I suggest reading my post about internal working of ConcurrentHashMap in Java.

69. What will be the problem if you don't override hashcode () method?

- If you don't override equals method, than contract between equals and hashcode will not work, according to which, two object which are equal by equals() must have same hashcode. In this case, other object may return different hashcode and will be stored on that location, which breaks invariant of HashMap class, because they are not supposed to allow duplicate keys. When you add object using put() method, it iterate through all Map.Entry object present in that bucket location, and update value of previous mapping, if Map already contains that key. This will not work if hashcode is not overridden.

70. Does not overriding hashcode () method has any performance implication?

- This is a good question and opens to all, as per my knowledge a poor hashcode function will result in frequent collision in HashMap, which eventually increase time for adding an object into Hash Map.

71. What's wrong using HashMap in multithreaded environment? When get () method go to infinite loop?

- Well nothing is wrong, it depending upon how you use. For example if you initialize the HashMap just by one thread and then all threads are only reading from it, then it's perfectly fine. One example of this is a Map, which contains configuration properties. Real problem starts when at-least one of those thread is updating HashMap i.e. adding, changing or removing any key value pair. Since put () operation can cause re-sizing and which can further lead to infinite loop, that's why either you should use **HashTable** or ConcurrentHashMap, later is better.

72. How do you use a custom object as key in Collection classes like HashMap ?

- If one is using the custom object as key then one needs to override equals() and hashCode() method
- and one also need to fulfill the contract.
- If you want to store the custom object in the SortedCollections like SortedMap then one needs to make sure that equals() method is consistent to the compareTo() method. If inconsistent , then collection will not follow their contracts ,that is , Sets may allow duplicate elements.

73. Give example of a HashSet and TreeSet

```
HashSet<String> hashSet = new HashSet<String>();
hashSet.add("Java");
hashSet.add(null);

TreeSet<String> treeSet = new TreeSet<String>();
treeSet.add("C++");
treeSet.add(null); //Java.lang.NullPointerException
Output:
Exception in thread "main" java.lang.NullPointerException
   at java.util.TreeMap.put(TreeMap.java:5
   at java.util.TreeSet.add(TreeSet.java:2
   at test.CollectionTest.main(CollectionTest.java:
Java Result: 1
```

-

LinkedHashSet

74. What is a LinkedHashSet

- Uses LinkedHashMap internally
- LinkedHashSet is the **HashTable** and linked list implementation of the Set **INTERFACE** with predictable iteration order.
- The linked list defines the **iteration ordering, which is the order in which elements were inserted into the set.** Insertion order is not affected if an element is re-inserted into the set.

75. Why we need LinkedHashSet when we already have the HashSet and TreeSet ?

- HashSet and TreeSet classes were added in jdk 1.2 while LinkedHashSet was added to the jdk in java version 1.4
- HashSet provides constant time performance for basic operations like (add, remove and contains) method but **elements are in chaotic ordering i.e unordered.**
- In TreeSet elements are naturally sorted but **there is increased cost associated with it .**
- So , LinkedHashSet is added in jdk 1.4 to maintain ordering of the elements without incurring increased cost.

76. What is Initial capacity and load factor?

- The ***capacity*** is the number of buckets(used to store key and value) in the Hash table ,
- ***initial capacity*** is simply the capacity at the time Hash table is created.
- The ***load factor*** is a measure of how full the Hash table is allowed to get before its capacity is automatically increased.

77. What is the CONSTRUCTOR of LinkedHashSet

- depends on above two parameters *initial capacity* and *load factor* .
- There are four **CONSTRUCTOR**s present in the LinkedHashSet class .
- All **CONSTRUCTOR**s have the same below pattern :

```
    // CONSTRUCTOR 1
public LinkedHashSet (int initialCapacity , float loadFactor)
    {
   super(initialCapacity , loadFactor , true);
    }
```

- *Note :* If initialCapacity or loadFactor parameter value is missing during LinkedHashSet object creation , then default value of initialCapacity or loadFactor is used .
- Default value for initialCapacity : 16 ,
- Default value for loadFactor : 0.75f

78. What is a overloaded CONSTRUCTORof a LinkedHashSet

loadFactor is missing in the LinkedHashSet **CONSTRUCTOR** argument. So during super() call , we use the default value of the loadFactor(0.75f).

```
    // CONSTRUCTOR 2
public LinkedHashSet (int initialCapacity)
    {
   super(initialCapacity , 0.75f , true);
    }
```

check the below overloaded **CONSTRUCTOR** , initialCapacity and loadFactor both are missing in the LinkedHashSet **CONSTRUCTOR** argument. So during super() call , we use the default value of both initialCapacity(and

loadFactor(0.75f).

```
    // CONSTRUCTOR 3
public LinkedHashSet ()
    {
   super(16 , 0.75f , true);
    }
```

below is the last overloaded CONSTRUCTOR which uses Collection in the LinkedHashSet **CONSTRUCTOR** argument. So during super() call , we use the default value of loadFactor(0.75f).

```
    // CONSTRUCTOR 4
public LinkedHashSet (Collection c)
    {
   super(Math.max(2*c.size() ,, 0.75f , true);
    }
```

Note : Since LinkedHashSet extends HashSet class.

Above all the 4 **CONSTRUCTOR**s are calling the super class (i.e HashSet) **CONSTRUCTOR** , given below

```
public HashSet (int initialCapacity , float loadFactor , boolean dummy)
    {
 1.  map = new LinkedHashMap<>(initialCapacity , loadFactor);
    }
```

79. In the above HashSet CONSTRUCTOR , there are two main points to notice :

a. We are using extra boolean parameter *dummy* . It is used to distinguish other int, float **CONSTRUCTOR**s present in the HashSet class.

b. Internally it is creating a LinkedHashMap object passing the initialCapacity and loadFactor as parameters.

80. How LinkedHashSet Maintains Unique Elements ?

```
public class HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, java.io.Serializable
{
    private transient HashMap<E,Object> map;

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

    private static final Object PRESENT = new Object();



    public HashSet(int initialCapacity , float loadFactor , boolean dummy) {
         map = new LinkedHashMap<>(initialCapacity , loadFactor);
    }

    // SOME CODE ,i.e Other methods in Hash Set


    public boolean add(E e) {
   return map.put(e, PRESENT)==null;
    }



    // SOME CODE ,i.e Other methods in Hash Set
}
```

- So , we are achieving uniqueness in LinkedHashSet,internally in java through LinkedHashMap . Whenever you create an object of LinkedHashSet it will indirectly create an object of LinkedHashMap as you can see in the italic lines of HashSet **CONSTRUCTOR**.
- As we know in LinkedHashMap each key is unique . So what we do in the LinkedHashSet is that we pass the argument in the add(Elemene E) that is E as a key in the LinkedHashMap . Now we need to associate some

value to the key , so what Java apis developer did is to pass the Dummy value that is (new Object ()) which is referred by Object reference PRESENT .

- So , actually when you are adding a line in LinkedHashSet like linkedhashset.add(what java does internally is that it will put that element E here 5 as a key in the LinkedHashMap(created during LinkedHashSet object creation) and some dummy value that is Object's object is passed as a value to the key .
- Since LinkedHashMap put(Key k , Value v) method does not have its own implementation . LinkedHashMap put(Key k , Value v) method uses HashMap put(Key k , Value v) method.
- Now if you see the code of the HashMap put(Key k,Value v) method , you will find something like this

```
 public V put(K key, V value) {
//Some code
}
```

- The main point to notice in above code is that put (key,value) will return
  1. null , if key is unique and added to the map
  2. Old Value of the key , if key is duplicate
- So , in LinkedHashSet add() method , we check the return value of map.put(key,value) method with null value i.e.

```
  public boolean add(E e) {
 return map.put(e, PRESENT)==null;
 }
```

- So , if map.put(key,value) returns null ,then

```
map.put(e, PRESENT)==null will return true and element is added to the
LinkedHashSet.
```

- So , if map.put(key,value) returns old value of the key ,then

```
map.put(e, PRESENT)==null will return false and element is  not added to the
LinkedHashSet .
```

81. How LinkedHashSet Maintains Insertion Order ?

- LinkedHashSet differs from HashSet because it maintains the insertion order .
- LinkedHashSet implementation differs from HashSet in that it maintains a doubly-linked list running through all of its entries
- LinkedHashSet internally uses LinkedHashMap to add elements to its object.

82. What is Entry object?

- LinkedHashMap consists of a static inner class named as Entry . **Each object of Entry represents a key,value pair**. The key K in the Entry object is the value which needs to be added to the LinkedHashSet object. The value V in the Entry object is any dummy object called PRESENT.
- Insertion Order of the LinkedHashMap is maintained by two Entry fields head and tail , which stores the head and tail of the doubly linked list.

```
transient LinkedHashMap.Entry  head;

transient LinkedHashMap.Entry  tail;
```

- For double linked list we need to maintain the previous and next Entry objects for each Entry object .
- Entry fields *before* and *after* are used to store the references to the previous and next Entry objects .

```
static class Entry extends HashMap.Node {

    Entry before, after ;
    Entry( int hash , K key , V value , Node next ) {
super(hash,key,value,next);
    }
```

```
}
```

https://blog.udemy.com/java-interview-questions/

83. Why you need to override hashcode, when you override equals in Java?

- Because equals have code contract mandates to override equals and hashcode together .since many container class like HashMap or HashSet depends on hashcode and equals contract.

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

1. 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.
2. 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.
3. Contract of hashCode() and equals() method
    - If object1.equals(object, then object1.hashCode() == object2.hashCode() should always be true.
    - If object1.hashCode() == object2.hashCode() is true does not guarantee object1.equals(object2)

85. What is IdentityHashMap ?

- IdentityHashMap
- IdentityHashMap is a class present in java.util package. It implements the Map **INTERFACE** with a hash table , using reference equality instead of object equality when comparing keys and values.In other words , in IdentityHashMap two keys k1 and k2 are considered equal if only if (k1==k2).
- IdentityHashMap is not synchronized.
- Iterators returned by the iterator() method are fail-fast , hence , will throw ConcurrentModificationException.

86. What is WeakHashMap ?

- **HashTable** based implementation of Map **INTERFACE** with weak keys. An entry in WeakHashMap will automatically be removed when its key is no longer in ordinary use. More precisely the presence of a mapping for a given key will not prevent the key from being discarded by the garbage collector.
- It permits null keys and null values.
- Like most collection classes this class is not synchronized. A **SYNCHRONIZED** WeakHashMap may be constructed using the Collections.synchronizedMap() method.
- Iterators returned by the iterator() method are fail-fast , hence , will throw ConcurrentModificationException.

87. Which Two Methods should a HashMap key Object Implement? What happens during a hash collision?

- Use Hashmap or Vector Not **HashTable** LinkedList
- c) Always use **INTERFACE** to a represent and access a collection e.g. use List to store ArrayList, Map to store HashMap and so on.

88. Difference between HashMap and HashTable : Popular Interview Question in Java with

- HashMap is not SYNCHRONIZED
- HashTable or hashmap faster.
- What could go wrong is that if he placed another follow-up question like how hashMap works in Java or can you replace HashTable with ConcurrentHashMap etc.

- **HashTable** includes five point namely Synchronization,Null keys and values,Iterating values , Fail fast iterator ,Performance,Superclass .

- What is default size of ArrayList and HashMap in Java?

  As of Java 7 now, default size of ArrayList is 10 and default capacity of HashMap is 16, it must be power of 2. Here is code snippet from ArrayList and HashMap class :

```
// from ArrayList.java JDK 1.7

private static final int DEFAULT_CAPACITY = 10;

//from HashMap.java JDK 7

static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
```

89. Is it possible for two unequal objects to have the same hashcode?

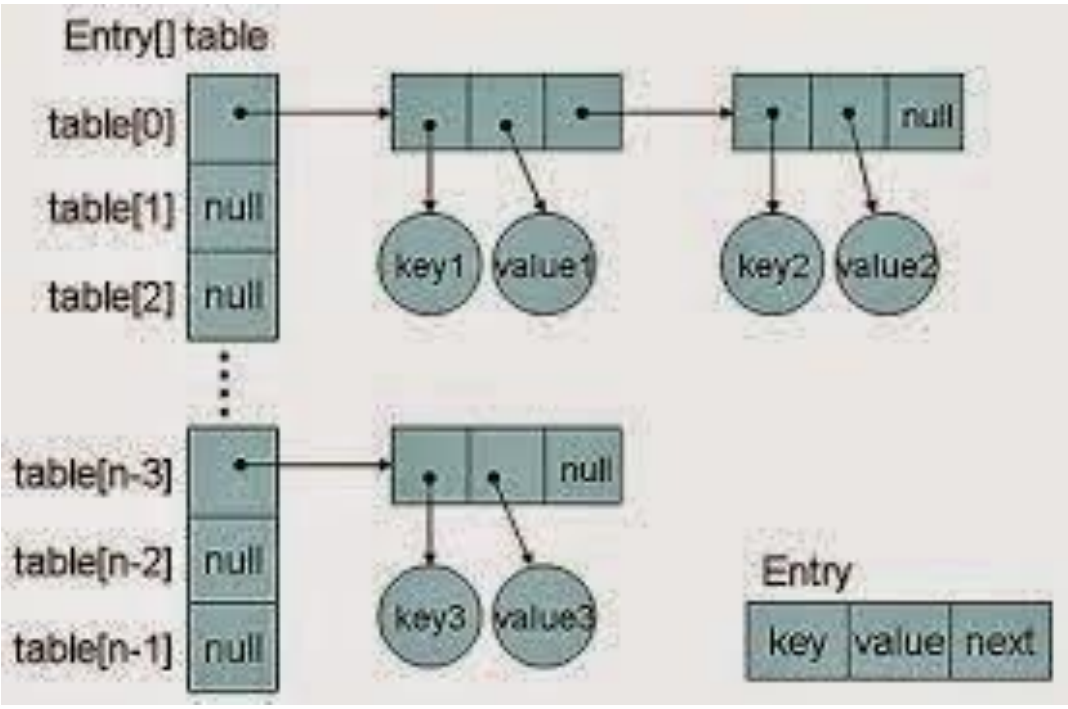
- Yes, two unequal objects can have same hashcode that's why collision happen in a hashmap.
- the equal hashcode contract only says that two equal objects must have the same hashcode it doesn't say anything about the unequal object.
- But you can have two unequal objects in Java can have same hashcode. equals() and hashCode() contract Some will give up at this point and few will move ahead and say "Since hashcode is same, bucket location would be same and collision will occur in HashMap Since `HashMap` uses LinkedList to store object, this entry (object of `Map.Entry` comprise key and value) will be stored in LinkedList.
- Great this answer make sense though there are many collision resolution methods available like linear probing and chaining, this is simplest and HashMap in Java does follow this. But story does not end here and interviewer asks

90. How will you retrieve Value object if two Keys will have the same hashcode?

- Interviewee will say we will call `get()` method and then HashMap uses Key Object's hashcode to find out bucket location and retrieves Value object but then you need to remind him that there are two Value objects are stored in same bucket , so they will say about traversal in LinkedList until we find the value object , then you ask

91. how do you identify value object because you don't have value object to compare ,

- Until they know that HashMap stores both Key and Value in `LinkedList` node or as `Map.Entry` they won't be able to resolve this issue and will try and fail.
- But those bunch of people who remember this key information will say that after finding bucket location, we will **call keys.equals() method** to identify a correct node in LinkedList and return associated value object for that key in Java HashMap. Perfect this is the correct answer.
- In many cases interviewee fails at this stage because they get confused between hashCode() and equals() or keys and values object in Java HashMap which is pretty obvious because they are dealing with the `hashcode()` in all previous questions and `equals()` come in picture only in case of retrieving value object from HashMap in Java. Some good developer point out here that using immutable,final object with proper `equals()` and `hashcode()` implementation would act as perfect Java HashMap keys and **improve the performance of Java HashMap by reducing collision**. Immutability *also allows caching their hashcode of different keys* which makes overall retrieval process very fast and suggest that String and various wrapper classes e.g. Integer very good keys in Java HashMap.



92. "What happens On HashMap in Java if the size of the HashMap exceeds a given threshold defined by load factor ?".

- Until you know how HashMap works exactly you won't be able to answer this question. If the size of the Map exceeds a given threshold defined by load-factor e.g. if the load factor is `.75` it will act to re-size the map once it filled 75%. Similar to other collection classes like ArrayList, Java HashMap re-size itself by creating a new bucket array of size twice of the previous size of HashMap and then start putting every old element into that new bucket array. This process is called `rehashing` because it also applies the hash function to find new bucket location.

93. If you manage to answer this question on HashMap in Java you will be greeted by "do you see any problem with resizing of HashMap in Java" ,

- you might not be able to pick the context and then he will try to give you hint about multiple thread accessing the Java `HashMap` and potentially looking for **race condition on HashMap in Java**.
- So the answer is Yes there is potential race condition exists while resizing `HashMap` in Java, if two thread at the same time found that now `HashMap` needs resizing and they both try to resizing. on the process of resizing of `HashMap` in Java, the element in the bucket which is stored in linked list get reversed in order during their migration to new bucket because Java HashMap doesn't append the new element at tail instead it append new element at the head *to avoid tail traversing*. If race condition happens then you will end up with an infinite loop. Though this point, you can potentially argue that what the hell makes you think to use HashMap in multi-threaded environment to interviewer :)

94. Can we use any custom object as a key in HashMap?

- This is an extension of previous questions. Of course you can use any Object as key in Java HashMap provided it follows equals and hashCode contract and its hashCode should not vary once the object is inserted into Map. If the custom object is Immutable than this will be already taken care because you can not change it once created.

95. How null key is handled in HashMap? Since equals() and hashCode() are used to store and retrieve

values, how does it work in case of the null key?

- The null key is handled specially in HashMap, there are two separate methods for that `putForNullKey(V value)` and `getForNullKey()`. Later is offloaded version of get() to look up null keys. Null keys always map to index 0. This null case is split out into separate methods for the sake of performance in the two most commonly used operations (get and put), but incorporated with conditionals in others. In short, `equals()` and `hashcode()` method are not used in case of null keys in HashMap.

  here is how nulls are retrieved from HashMap

```
private V getForNullKey() {

 if (size == 0) {
     return null;
 }
 for (Entry<K,V> e = table[0]; e != null; e = e.next) {
     if (e.key == null)
         return e.value;
 }
 return null;
}
```

96. Can two equal object have the different hash code?

- No, thats not possible according to hash code contract.

97. Can we use random numbers in the hashcode() method?

- No, because hashcode of an object should be always same. See the answer to learning more about things to remember while overriding hashCode() method in Java.

• **HashMap** ○ Implements Map<K,V>	• Essentially an unsynchronized `Hashtable` that supports `null`keys and values. • Fastest updates (key/values); allows one null key, many null values.
• **Hashtable** ○ **Implements Map<K,V>**	• Like a slower HashMap (as with Vector, due to its synchronized methods). • No null values or null keys allowed
• **LinkedHashMap** ○ **Implements Map<K,V>**	• Predictable iteration order. • Runs nearly as fast as HashMap. • Allows one null key, many null values.

-

98. Which class does not override the `equals()` and `hashCode()` methods, inheriting them directly from class Object?

A. java. lang.String

B. java. lang.Double

C. java. lang.StringBuffer

D. java. lang.Character

Correct Answer: Option **C**

Explanation:

`java. lang.StringBuffer` is the only class in the list that uses the default methods provided by class `Object`.

99. Which of the following statements about the `hashcode()` method are incorrect?

4. The value returned by `hashcode()` is used in some collection classes to help locate objects.
5. ~~The `hashcode()` method is required to return a positive `int` value.~~
6. ~~The `hashcode()` method in the `String` class is the one inherited from `Object`.~~
7. Two new empty String objects will produce identical hashcodes.

A. 1 and

B. 2 and 3

C. 3 and

D. 1 and

Correct Answer: Option **B**

Explanation:

(2) is an incorrect statement because there is no such requirement.

(3) is an incorrect statement and therefore a correct answer because the hashcode for a string is computed from the characters in the string.

100. assuming that the `equals()` and `hashCode()` methods are properly implemented, if the output is "`x = 1111`", which of the following statements will always be true?

```
x = 0;
if (x1.hashCode() != x2.hashCode() )  x = x + 1;
if (x3.equals(x4) )  x = x + 10;
if (!x5.equals(x6) ) x = x + 100;
if (x7.hashCode() == x8.hashCode() )  x = x + 1000;
System.out.println("x = " + x);
```

A. x2.equals(x1)

B. x3.hashCode() == x4.hashCode()

C. x5.hashCode() != x6.hashCode()

D. x8.equals(x7)

Correct Answer: Option **B**

Explanation:

By contract, if two objects are equivalent according to the `equals()` method, then the `hashCode()` method

must evaluate them to be ==.
Option A is incorrect because if the `hashCode()` values are not equal, the two objects must not be equal.
Option C is incorrect because if `equals()` is not true there is no guarantee of any result from `hashCode()`.
Option D is incorrect because `hashCode()` will often return == even if the two objects do not evaluate to `equals()` being true.

101. What two statements are true about properly overridden `hashCode()` and `equals()` methods?

~~8. `hashCode()` doesn't have to be overridden if `equals()` is.~~
~~9. `equals()` doesn't have to be overridden if `hashCode()` is.~~
10. `hashCode()` can always return the same value, regardless of the object that invoked it.
11. `equals()` can be true even if it's comparing different objects.

A. 1 and

B. 2 and

C. 3 and 4

D. 1 and

Correct Answer: Option **C**

Explanation:

(3) and (4) are correct.

(1) and (2) are incorrect because by contract `hashCode()` and `equals()` can't be overridden unless both are overridden.

LinkedHashMap

102. Map - LinkedHashMap extends HashMap implements Map(I) Example

```
package.com.instanceofjava;import.java.util.LinkedHashMap;
public class testSet {
   public static void main(String args[]){

      LinkedHasMap linkedHashMap=new LinkedHasMap();

      linkedHashMap.add('Indhu');
            linkedHashMap.add("Indhu");
            linkedHashMap.add("Sindhu");
            linkedHashMap.add("Bindhu");
            linkedHashMap.add(null);
            linkedHashMap.add(null);

            Iterator it=linkedHashMap.iterator();
            while(it.hasNext()){
               System.out.println(it.next);
            }
   }
}

Output:
```

```
Indhu Sindhu Bindhu null
```

ConcurrentHashMap

103. Can we use ConcurrentHashMap in place of Hashtable?

- This is another question which getting popular due to increasing popularity of ConcurrentHashMap. Since we know Hashtable is synchronized but ConcurrentHashMap provides better concurrency by only locking portion of map determined by concurrency level. ConcurrentHashMap is certainly introduced as Hashtable and can be used in place of it, but Hashtable provides stronger thread-safety than ConcurrentHashMap. See my post difference between Hashtable and ConcurrentHashMap for more details.
- Personally, I like this question because of its depth and number of concept it touches indirectly if you look at questions asked during interview this HashMap questions has verified

  The concept of hashing

  Collision resolution in HashMap

  Use of equals () and hashCode () and their importance in HashMap?

  The benefit of the immutable object?

  Race condition on HashMap in Java

  Resizing of Java HashMap

equals and hashcode method

104. Which two method you need to implement for key Object in HashMap ?

- In order to use any object as Key in HashMap, it must implements equals and hashcode method in Java.

105. How to write a hashCode?

```
class Test1
{
    public int value;
    public int hashCode() { return 42; }
}
class Test2
{
    public int value;
    public int hashcode() { return (int)(value^5); }
}
```

a. `class Test1` will not compile.

b. The `Test1 hashCode()` method is more efficient than the `Test2 hashCode()` method.

c. The `Test1 hashCode()` method is less efficient than the `Test2 hashCode()` method.

d. `class Test2` will not compile.

Correct Answer: Option **C**

Explanation:

The so-called "hashing algorithm" implemented by `class Test1` will always return the same value, 42, which is legal but which will place all of the hash table entries into a single bucket, the most inefficient setup possible.

Option A and D are incorrect because these classes are legal.

Option B is incorrect based on the logic described above.

```
@Override
 public int hashCode() {
      int result = 0;
      result = (int) (value / 11);
      return result;
 }

 @Override
 public boolean equals(Object o) {
     if ((o instanceof Crunchify) && (((Crunchify) o).getValue() == this.value)) {
         return true;
     } else {
         return false;
     }
 }
```

In my opinion better way to **override both equals and hashcode method** should be left to IDe. I have seen Netbeans and Eclipse and found that both has excellent support of generating *code for equals and hashcode* and there implementations seems to follow all best practice and requirement e.g. null check , instanceof check etc and also frees you to remember how to compute hashcode for different data-types.

Read more: http://javarevisited.blogspot.com/2011/10/override-hashcode-in-java-example.html#ixzz4yCB3ewh5

Synchronized HashMap, SynchronizedMap

106. How do you get synchronized HashMap in java?

Using Collections.synchronizedMap() method.

```
import java.  util.Collections;
import java.  util.HashMap;
import java.  util.Map;

public class JavaHashMapPrograms
{
    public static void main(String[] args)
    {
        //Creating the HashMap

        HashMap<String, Integer> map = new HashMap<String, Integer>();

        //Getting synchronized Map

        Map<String, Integer> syncMap = Collections.synchronizedMap(map);
    }
}
```

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

- IHashMap 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.

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

- 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.

LinkedHashMap

109. Map - LinkedHashMap extends HashMap implements Map(I) Example

HashMapStructure.java(main class)

```
import java.util.HashMap;
import java.util.Iterator;

public class HashMapStructure {

    /**
     * @author Arpit Mandliya
     */
    public static void main(String[] args) {

        Country india=new Country("India",1000);
        Country japan=new Country("Japan",10000);

        Country france=new Country("France",2000);
        Country russia=new Country("Russia",20000);

        HashMap<country,string> countryCapitalMap=new HashMap<country,string>();
        countryCapitalMap.put(india,"Delhi");
        countryCapitalMap.put(japan,"Tokyo");
        countryCapitalMap.put(france,"Paris");
        countryCapitalMap.put(russia,"Moscow");

        Iterator<country>
countryCapitalIter=countryCapitalMap.keySet().iterator();//put debug point at this
line
        while(countryCapitalIter.hasNext())
        {
            Country countryObj=countryCapitalIter.next();
            String capital=countryCapitalMap.get(countryObj);
            System.out.println(countryObj.getName()+"----"+capital);
            }
        }
```

```
}
</country></country,string></country,string>
```

110. Map - LinkedHashMap extends HashMap implements Map(I) Example

Array Sort

```
import java.util.Arrays;
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class TestComparable {
   public static void main(String[] args) {
      // Sort and search an "array" of Strings
      String[] array = {"Hello", "hello", "Hi", "HI"};

      // Use the Comparable defined in the String class
      Arrays.sort(array);
      System.out.println(Arrays.toString(array));  // [HI, Hello, Hi, hello]

      // Try binary search - the array must be sorted
      System.out.println(Arrays.binarySearch(array, "Hello")); // 1
      System.out.println(Arrays.binarySearch(array, "HELLO")); // -1 (insertion at
index 0)

      // Sort and search a "List" of Integers
      List<Integer> lst = new ArrayList<Integer>();
      lst.add(22);  // auto-box
      lst.add(11);
      lst.add(44);
      lst.add(33);
      Collections.sort(lst);    // Use the Comparable of Integer class
      System.out.println(lst);  // [11, 22, 33, 44]
      System.out.println(Collections.binarySearch(lst, 22)); // 1
      System.out.println(Collections.binarySearch(lst, 35)); // -4 (insertion at
index 3)
   }
}
```

111. Map - LinkedHashMap extends HashMap implements Map(I) Example

In this example, instead of using the default `Comparable`, we define our customized `Comparator` for `String`s and `Integer`s.

```
import java.util.Arrays;
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class TestComparator {

   // Define a Comparator<String> to order strings in case-insensitive manner
   public static class StringComparator implements Comparator<String> {
      @Override
      public int compare(String s1, String s2) {
         return s1.compareToIgnoreCase(s2);
      }
   }

   // Define a Comparator<Integer> to order Integers based on the least significant
```

```
digit
   public static class IntegerComparator implements Comparator<Integer> {
      @Override
      public int compare(Integer s1, Integer s2) {
         return s1%10 - s2%10;
      }
   }

   public static void main(String[] args) {
      // Use a customized Comparator for Strings
      Comparator<String> compStr = new StringComparator();

      // Sort and search an "array" of Strings
      String[] array = {"Hello", "Hi", "HI", "hello"};
      Arrays.sort(array, compStr);
      System.out.println(Arrays.toString(array));  // [Hello, hello, Hi, HI]
      System.out.println(Arrays.binarySearch(array, "Hello", compStr)); // 1
      System.out.println(Arrays.binarySearch(array, "HELLO", compStr)); // 1 (case-
insensitive)

      // Use a customized Comparator for Integers
      Comparator<Integer> compInt = new IntegerComparator();

      // Sort and search a "List" of Integers
      List<Integer> lst = new ArrayList<Integer>();
      lst.add(42);  // auto-box
      lst.add(21);
      lst.add(34);
      lst.add(13);
      Collections.sort(lst, compInt);
      System.out.println(lst);  // [21, 42, 13, 34]
      System.out.println(Collections.binarySearch(lst, 22, compInt)); // 1
      System.out.println(Collections.binarySearch(lst, 35, compInt)); // -5
(insertion at index 4)
   }
}
```

112. Map(I) - LinkedHashMap extends HashMap implements Map(I) Example

```
import com.sun.java.util.collections.*;
public class HashMapTest {
   // Statics
   public static void main( String [] args ) {
      System.out.println( "Collection HashMap Test" );
      HashMap collection1 = new HashMap();
      // Test the Collection interface
      System.out.println( "Collection 1 created, size=" + collection1.size() +
         ", isEmpty=" + collection1.isEmpty() );
      // Adding
      collection1.put( new String( "Harriet" ), new String( "Bone" ) );
      collection1.put( new String( "Bailey" ),  new String( "Big Chair" ) );
      collection1.put( new String( "Max" ),     new String( "Tennis Ball" ) );
      System.out.println( "Collection 1 populated, size=" + collection1.size() +
         ", isEmpty=" + collection1.isEmpty() );
      // Test Containment/Access
      String key = new String( "Harriet" );
      if ( collection1.containsKey( key ) )
         System.out.println( "Collection 1 access, key=" + key + ", value=" +
            (String) collection1.get( key ) );
      // Test iteration of keys and values
      Set keys = collection1.keySet();
      System.out.println( "Collection 1 iteration (unsorted), collection contains
keys:" );
      Iterator iterator = keys.iterator();
```

```
      while ( iterator.hasNext() )
         System.out.println( "   " + iterator.next() );
      collection1.clear();
      System.out.println( "Collection 1 cleared, size=" + collection1.size() +
         ", isEmpty=" + collection1.isEmpty() );
   }
}
```