Map Interface:

#. A map defines mapping from key to values, the key-value pair is called an Entry.

#. A Map does not allow duplicate keys that means key are unique. Each key maps to one value, which is called **Singled Value Map**. Both the keys and the value must be the object.

#. A Map is not a Collection and the interface does not extend the Collection interface.

#. Map is useful if you have to search, update or delete elements on the basis of key.

Important Methods of Map Interface:

Method	Description
Object put(Object key, Object value)	It is used to insert an entry in this map.
void putAll(Map map)	It is used to insert the specified map in this map.
Object remove(Object key)	It is used to delete an entry for the specified key.
Public in size()	Return the size of the Map
Public Collection value()	@return a collection view of the values contained in this map
Object get(Object key)	It is used to return the value for the specified key.
boolean containsKey(Object key)	It is used to search the specified key from this map.
Set keySet()	It is used to return the Set view containing all the keys.
Set entrySet()	It is used to return the Set view containing all the keys and values.
Public boolean isEmpty()	Returns true is empty
public void clear()	Removes all of the mappings from this map (optional operation). The map will be empty after this call returns. @throws UnsupportedOperationException if the <tt>clear</tt> operation is not supported by this map
boolean containsValue	@param value value whose presence in this map is to be tested. @return <tt>true</tt> if this map maps one or more keys to the specified value. @throws ClassCastException if the value is of an inappropriate type for this map.

Entry Interface:

#. Map interface has nested interface that name is **Entry**

Methods of Map.Entry interface

Method	Description
Object getKey()	It is used to obtain key.
Object getValue()	It is used to obtain value.
V setValue(V value)	
boolean equals(Object o);	
int hashCode();	
public static <K extends Comparable<? super K>, V> <Map.Entry<K,V>> **comparingByKey**()	
public static <K, V extends Comparable<? super V>> **Comparator**<Map.Entry<K,V>> comparingByValue()	
public static <K, V> Comparator<Map.Entry<K, V>> **comparingByKey**(Comparator<? super K> cmp)	
public static <K, V> Comparator<Map.Entry<K, V>> **comparingByValue**(Comparator<? super V> cmp)	

HashMap Class:

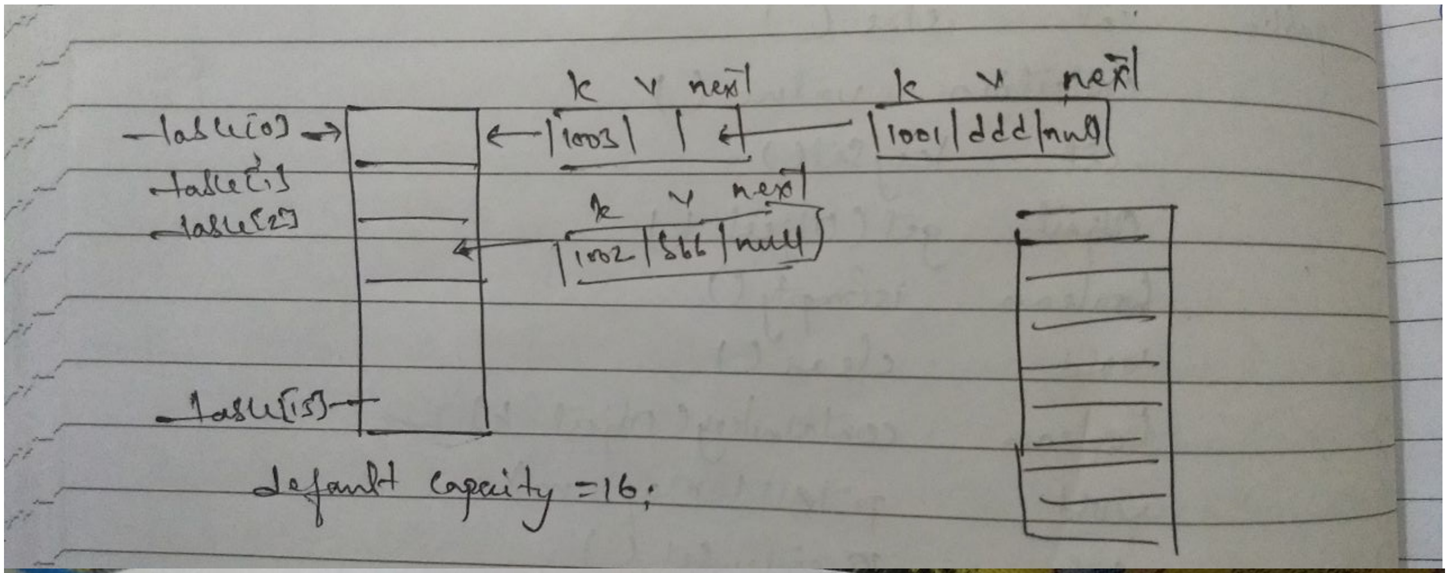
Java HashMap class implements the map interface by using a hashtable. It inherits AbstractMap class and implements Map interface.

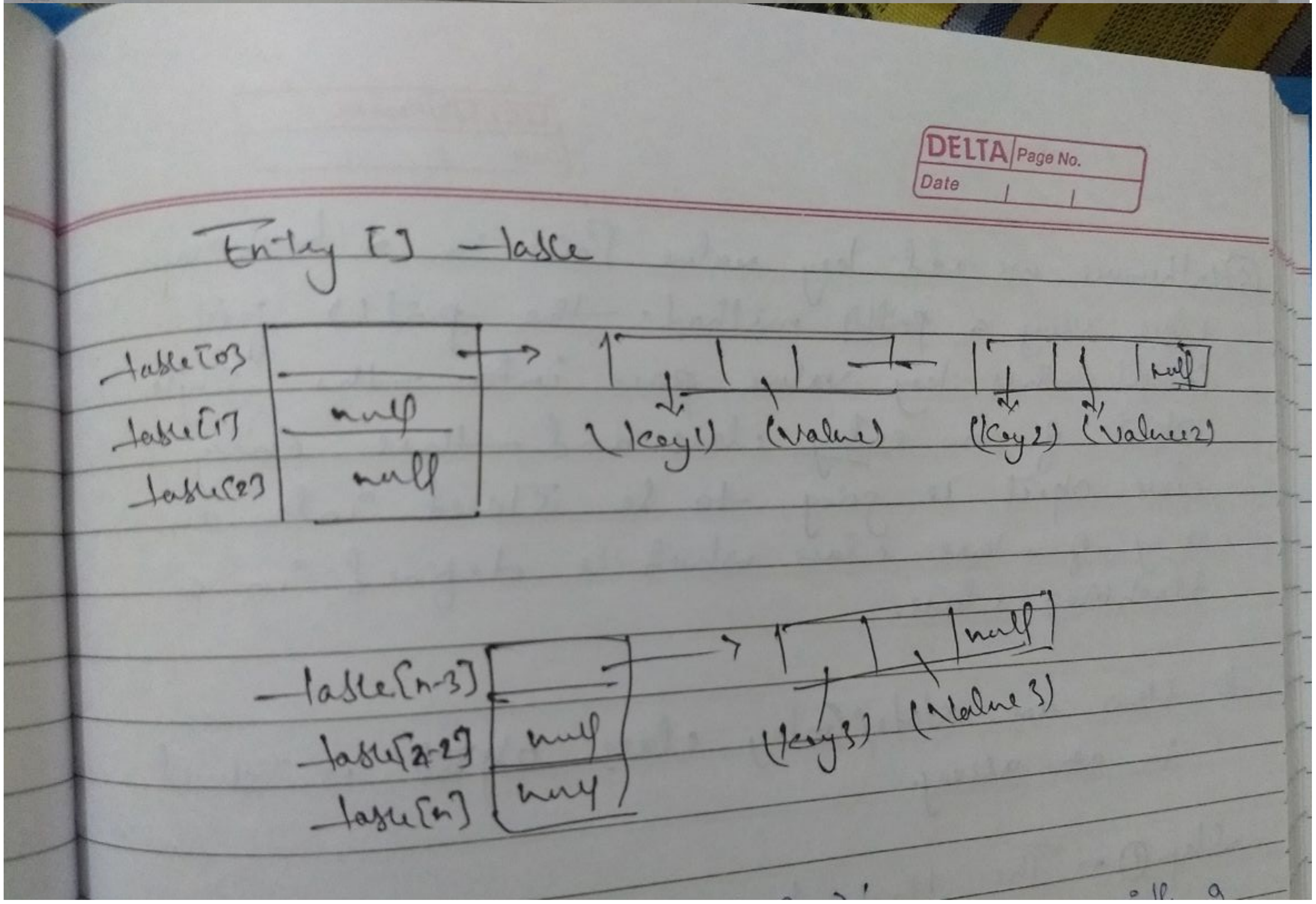
Let's see the declaration for java.util.HashMap class.

```
public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>,

Cloneable, Serializable
```

#. HashMap class has nested static class **Node** (before 1.8 “Entry”) that implements **Map.Entry** interface.

```
class HashMap{

    Entry table[];

    class Node implements Map.Entry{
        Entry next;
        Object key;
        Object value;

        // Constructor
        Entry(Object k, Object v){
            key= key;
            value = value;
        }

        HashMap(){
            table = new Entry[16];
        }

        put(){
            Entry e = new Entry(k,v);
            int hc = e.key.hashtable();
            int index = hc%16;
            e.key.equals(e.key);
        }
    }
}
```





Important Points of HashMap:

#. HashMap class has one nested class with a name Node (Entry before JDK 1.8) which implements a one interface called Map.Entry interface.

#. Map interface is also having a one nested interface called Node (Entry) which implements one interface called Map.Entry and this interface has got three methods.

- Public Object getKey()
- Public Object getValue()
- Public int hashCode()

#. Whenever we create the object of HashMap class using the default constructor then One Node/Entry class array will be created of size 16 which is known as capacity of HashMap.

#. Each index of this array is called bucket.

#. And each bucket is a Linked List.

#. Whenever we add key-value pair in a HasMap class using a put() method, the put() method first convert this key-value pair into the single object of an Node/Entry class and this class object is going to be stored into an array of this class which is defined in a HashMap class.

IQ: How a Node/Entry class object is stored in an array?

Step1: The hash-code of the key class object is first calculated.

Step2: Calculating the remainder of this hashcode by dividing it via total number of bucket or capacity.

Step3: This remainder is stored as an index of buckets.

Step4: Once the bucket is divided, then the equals() method is called on key class object and then passes existing key class object into this method one by one.

If the key Match is found then the Node/Entry class object is not added into the bucket but value of the key will be replaced by new value. Otherwise this Node/Entry class object s added in a bucker as first node of Linked List which is maintained by a bucket.

Few More Important Points:

#1. The default size of an array is 16 (always power of 2) and the load factor is 0.75. Load factor means whenever the size of the HashMap reaches to 75% of its current size. I.e. it will doubles the hashcode of existing data structures elements.

#2. Hence to avoid rehashing of the data structure (as it degrade the performance) as elements grow it is the best practice to explicitly give the size of the HashMap while creating it, never give the capacity too high and load factor too low.

#3. Since java is multi-threaded, it is very possible that more than one thread might be using same HashMap and then they both realize the need for resize the HashMap at the same time which leads to race condition.

?. What is race condition with respect to HashMap?

When two or more threads see the need for resizing the same HashMap, they might end up adding the elements of old bucket to the new bucket simultaneously and hence might lead to infinite loops. In case collision, i.e. when there are different keys with same hashcode, internally we use single linked list to store the element, the every new element at the head of the linked list to avoid tail traversing and hence at the time of resizing the entire sequence of objects in linked list gets reversed, during which there are chances of infinite loop.

For ex: Let’s assume these are 3 keys with same hashcode and hence stored in Linked List inside a bucket below format is in object value (Current_Initial Structure: 1(100,200) -> 2 (200, 300) -> (300, null) -> 2(200, 100) -> (100, null)

When Thread-2 starts resizing, it’s again start with first element by placing it at the head.

1(100, 300) -> 3(300, 200) -> 2(200, 100) => which becomes a infinite loop for next instruction and thread hangs here.

#4. HashMap has a static class named **Node/Entry** which implements **Map.Entry** interface. The Entry class looks like:

```
static class Node/Entry implements Map.Entry{
    final Object key;
    Object value;
    final int hash;
    Entry next;

    Entry/Node(int i, Object obj, Object obj1, Entry entry){
        value = obj1;
        next = entry;
        key = obj;
        hash = i;
    }

    // Other methods
}
```

Every time we insert into HashMap using put () method, a new Node/Entry object is created (Not true in some cases. If key is already exist, then it just replace the value.). Map internally used two data structures to manage/store data.

HashMap Class

The HashMap class used a **HashTable** to implement the Map interface. The HashMap class is not thread-safe and permits one null key and null value. The HashMap class is roughly equivalent to HashTable except that it is Un-Synchronized and permits null. This class make no guarantee as to the order of the Map.

HashMap can be synchronized by using a method.

```
Map m = Collections.synchronizedMap(Map m) Hashmap());
```

Constructor of HashMap:

There are four types of constructor in HashMap as shown below:

#1. HashMap(): It is a default constructor.

#2. HashMap(Map m): It initialized the HashMap by using the elements M.

#. HashMap(int capasity): It initialized the capacity of the HashMap to capacity. The default initial capacity of HashMap will be 16 and load factor will be 0.75. Load factor represents at what level HashMap should be doubled.

#4. HashMap (int capacity, float loadFactor): It initialized both capacity and load factory by using it arguments.

Tips: If we want store multiple value with one key then we can store array as value.

There are many new methods in HashMap introduced in Java 8.

1. **public V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)**: If the specified key is not already associated with a value (or is mapped to null), this method attempts to compute its value using the given mapping function and enters it into the HashMap unless Null.

2. **public V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)**: If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.

3. **public V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)**: This HashMap method attempts to compute a mapping for the specified key and its current mapped value.

4. **public void forEach(BiConsumer<? super K, ? super V> action)**: This method performs the given action for each entry in this map.

5. **public V getOrDefault(Object key, V defaultValue)**: Same as get except that defaultValue is returned if no mapping found for the specified key.

6. **public V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)**: If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value. Otherwise, replaces the associated value with the results of the given remapping function, or removes if the result is null.

7. **public V putIfAbsent(K key, V value)**: If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.

8. **public boolean remove(Object key, Object value)**: Removes the entry for the specified key only if it is currently mapped to the specified value.

9. **public boolean replace(K key, V oldValue, V newValue)**: Replaces the entry for the specified key only if currently mapped to the specified value.
10. **public V replace(K key, V value)**: Replaces the entry for the specified key only if it is currently mapped to some value.

11. **public void replaceAll(BiFunction<? super K, ? super V, ? extends V> function)**: Replaces each entry's value with the result of invoking the given function on that entry.

Java HashMap Load Factor

Load Factor is used to figure out when HashMap will be rehashed and bucket size will be increased. Default value of bucket or capacity is 16 and load factor is 0.75. Threshold for rehashing is calculated by multiplying capacity and load factor. So default threshold value will be 12. So when the HashMap will have more than 12 mappings, it will be rehashed and number of bins will be increased to next of power 2 i.e 32. Note that HashMap capacity is always power of 2.

Default load factor of 0.75 provides good tradeoff between space and time complexity. But you can set it to different values based on your requirement. If you want to save space, then you can increase it's value to 0.80 or 0.90 but then get/put operations will take more time.

Java HashMap keySet

Java HashMap keySet method returns the Set view of keys in the HashMap. This Set view is backed by HashMap and any changes in HashMap is reflected in Set and vice versa. Below is a simple program demonstrating HashMap keySet examples and what is the way to go if you want a keySet not backed by map.

```
public class HashMapKeySetExample {

	public static void main(String[] args) {

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

		map.put("1", "1");

		map.put("2", "2");

		map.put("3", "3");


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

		System.out.println(keySet);
```

```
        map.put("4", "4");
        System.out.println(keySet); // keySet is backed by Map

        keySet.remove("1");
        System.out.println(map); // map is also modified

        keySet = new HashSet<>(map.keySet()); // copies the key to new Set
        map.put("5", "5");
        System.out.println(keySet); // keySet is not modified
    }
}
```

Output of the above program will make it clear that keySet is backed by map.

```
[1, 2, 3]
[1, 2, 3, 4]
{2=2, 3=3, 4=4}
[2, 3, 4]
```

Java HashMap values

Java HashMap values method returns a Collection view of the values in the Map. This collection is backed by HashMap, so any changes in HashMap will reflect in values collection and vice versa. A simple example below confirms this behaviour of HashMap values collection.

There are few more things about HashMap that are available in JournalDev website. I read it to inderstand the HashMap better.

LinkedHashMap

public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V>

LinkedHashMap is a subclass of the **HashMap** class, the element of LinkedHashMap are **ordered** by default. The entries of a LinkedJHashMap are in key **insertion** order, that is order in which the keys are inserted in the Map.

Access order means the order we retrieve the element. It remembers the order and, if we next time retrieve the elements then it will retrieve in that remembered order. To tell him to remember the access order we have to use it's 4th constructor. It has 3 constructors same as HashMap.

LinkedHashMap has one extra 4th constructor that remember the access order.

The important points about Java LinkedHashMap class are:

- A LinkedHashMap contains values based on the key.
- It contains only unique elements.
- It may have one null key and multiple null values.
- It is same as HashMap instead maintains insertion order.

Programs

Concurrent HashMap

Java **1.5** has introduced **java.util.concurrent** package with **Collection classes** implementations that allow you to modify your collection objects at runtime.

ConcurrentHashMap is the class that is similar to HashMap but works fine when you try to modify your map at runtime.

Let’s understand this via a Java program.

```
import java.util.HashMap;

import java.util.Iterator;

import java.util.Map;

import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapExample {

	public static void main(String[] args) {

		//ConcurrentHashMap

		Map<String,String> myMap = new ConcurrentHashMap<String,String>();

		myMap.put("1", "1");

		myMap.put("2", "1");

		myMap.put("3", "1");

		myMap.put("4", "1");

		myMap.put("5", "1");

		myMap.put("6", "1");

		System.out.println("ConcurrentHashMap before iterator: "+myMap);

		Iterator<String> it = myMap.keySet().iterator();


		while(it.hasNext()){

			String key = it.next();

			if(key.equals("3")) myMap.put(key+"new", "new3");

		}

		System.out.println("ConcurrentHashMap after iterator: "+myMap);


		//HashMap

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

		myMap.put("1", "1");
```

```
          myMap.put("2", "1");
          myMap.put("3", "1");
          myMap.put("4", "1");
          myMap.put("5", "1");
          myMap.put("6", "1");
          System.out.println("HashMap before iterator: "+myMap);
          Iterator<String> it1 = myMap.keySet().iterator();

          while(it1.hasNext()){
               String key = it1.next();
               if(key.equals("3")) myMap.put(key+"new", "new3");
          }
          System.out.println("HashMap after iterator: "+myMap);
     }
}
```

When we try to run the above class, output is

```
ConcurrentHashMap before iterator: {1=1, 5=1, 6=1, 3=1, 4=1, 2=1}
ConcurrentHashMap after iterator: {1=1, 3new=new3, 5=1, 6=1, 3=1, 4=1, 2=1}
HashMap before iterator: {3=1, 2=1, 1=1, 6=1, 5=1, 4=1}
Exception in thread "main" java.util.ConcurrentModificationException
      at java.util.HashMap$HashIterator.nextEntry(HashMap.java:793)
      at java.util.HashMap$KeyIterator.next(HashMap.java:828)
      at com.test.ConcurrentHashMapExample.main(ConcurrentHashMapExample.java:44)
```

Looking at the output, its clear that **ConcurrentHashMap** takes care of any new entry in the map whereas HashMap throws ConcurrentModificationException.

Lets look at the exception stack trace closely. The statement that has thrown Exception is:

String key = it1.next();

It means that the new entry got inserted in the HashMap but Iterator is failing. Actually Iterator on Collection objects are **fail-fast** i.e any modification in the structure or the number of entry in the collection object will trigger this exception thrown by iterator.

So How does iterator knows that there has been some modification in the HashMap. We have taken the set of keys from HashMap once and then iterating over it.

HashMap contains a variable to count the number of modifications and iterator use it when you call its next() function to get the next entry.

`HashMap.java`

```
/**
     * The number of times this HashMap has been structurally modified
     * Structural modifications are those that change the number of mappings in
     * the HashMap or otherwise modify its internal structure (e.g.,
     * rehash).  This field is used to make iterators on Collection-views of
     * the HashMap fail-fast.  (See ConcurrentModificationException).
     */
    transient volatile int modCount;
```

Now to prove above point, lets change the code a little bit to come out of the iterator loop when we insert the new entry. All we need to do is add a break statement after the put call.

```
I                        f(key.equals("3")){
                             myMap.put(key+"new", "new3");
                             break;
                         }
```

Now execute the modified code and the output will be:

```
ConcurrentHashMap before iterator: {1=1, 5=1, 6=1, 3=1, 4=1, 2=1}
ConcurrentHashMap after iterator: {1=1, 3new=new3, 5=1, 6=1, 3=1, 4=1, 2=1}
HashMap before iterator: {3=1, 2=1, 1=1, 6=1, 5=1, 4=1}
HashMap after iterator: {3=1, 2=1, 1=1, 3new=new3, 6=1, 5=1, 4=1}
```

Finally, what if we won’t add a new entry but update the existing key-value pair. Will it throw exception?

Change the code in the original program and check yourself.

```
            myMap.put(key+"new", "new3");
            myMap.put(key, "new3");
```

TreeMap