**TreeMap:**

TreeMap class implements Map interface similar to HashMap class. The main difference between them is that HashMap is an unordered collection while TreeMap is sorted in the ascending order of its keys.

**TreeMap is unsynchronized collection class which means it is not suitable for thread-safe operations until unless synchronized explicitly.**

**A simple Example:**import java.util.TreeMap;

import java.util.Set;

import java.util.Iterator;

import java.util.Map;

public class Details {

public static void main(String args[]) {

/\* This is how to declare TreeMap \*/

TreeMap<Integer, String> tmap =

new TreeMap<Integer, String>();

/\*Adding elements to TreeMap\*/

tmap.put(1, "Data1");

tmap.put(23, "Data2");

tmap.put(70, "Data3");

tmap.put(4, "Data4");

tmap.put(2, "Data5");

/\* Display content using Iterator\*/

Set set = tmap.entrySet();

Iterator iterator = set.iterator();

while(iterator.hasNext()) {

Map.Entry mentry = (Map.Entry)iterator.next();

System.out.print("key is: "+ mentry.getKey() + " & Value is: ");

System.out.println(mentry.getValue());

}

}

}

**Now, the difference b/w HashMap and HashTable:**since, both are implemented using hashtable (open addressing)

1. Hashtable’s main methods are synchronized (no, that does not mean it is efficient. Since, synchronized used intrinsive locking) whereas HashMap’s methods are not.
2. HashMap allows one null key and multiple null values whereas Hashtable doesn’t allow any null key or value.
3. HashMap is generally preferred over HashTable if thread synchronization is not needed

**Why HashTable doesn’t allow null and HashMap does?**

To successfully store and retrieve objects from a HashTable, the objects used as keys must implement the hashCode method and the equals method. Since null is not an object, it can’t implement these methods. HashMap is an advanced version and improvement on the Hashtable. HashMap was created later.

**HashMap:  
  
Constructors:**

HashMap provides 4 constructors and access modifier of each is public:

**HashMap() :** It is the default constructor which creates an instance of HashMap with initial capacity 16 and load factor 0.75.

(what does that mean? When 13th element (because, at most 12 elements are there) is inserted in the HashMap the size of the HashMap will become 32 and all elements will be implicitly rehashed.

**HashMap(int initial capacity) :** It creates a HashMap instance with specified initial capacity and load factor 0.75.

**HashMap(int initial capacity, float loadFactor) :** It creates a HashMap instance with specified initial capacity and specified load factor.

**HashMap(Map map) :** It creates instance of HashMapwith same mappings as specified map.

**Other Functions: (Important)**  
**void clear()**

Removes all of the mappings from this map.

**boolean containsKey(Object key)**

Returns true if this map contains a mapping for the specified key.

**boolean containsValue(Object value)**

Returns true if this map maps one or more keys to the specified value.

**Set<Map.Entry<K,V>> entrySet()**

Returns a Set view of the mappings contained in this map.

**V get(Object key)**

Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

**boolean isEmpty()**

Returns true if this map contains no key-value mappings.

**Set<K> keySet()**

Returns a Set view of the keys contained in this map.

**V put(K key, V value)**

Associates the specified value with the specified key in this map.

void

**public Set<K> keySet()**

Returns a Set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator's own remove operation), the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Set.remove, removeAll, retainAll, and clear operations. It does not support the add or addAll operations.

**Basic Example:**

// Java program to illustrate

// Java.util.HashMap

import java.util.HashMap;

import java.util.Map;

public class GFG

{

public static void main(String[] args)

{

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

print(map);

map.put("vishal", 10);

map.put("sachin", 30);

map.put("vaibhav", 20);

System.out.println("Size of map is:- " + map.size());

print(map);

if (map.containsKey("vishal"))

{

Integer a = map.get("vishal");

System.out.println("value for key \"vishal\" is:- " + a);

}

map.clear();

print(map);

}

public static void print(Map<String, Integer> map)

{

if (map.isEmpty())

{

System.out.println("map is empty");

}

else

{

System.out.println(map);

}

}

}

**Example Of Using KeySet:**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");

// get keyset value from map

Set keyset = newmap.keySet();

// check key set values

System.out.println("Key set values are: " + keyset);

}

}

**Example of Using EntrySet:**

import java.util.HashMap;

import java.util.Map.Entry;

import java.util.Set;

public class MyHashMapEntrySet {

public static void main(String a[]){

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

//add key-value pair to hashmap

hm.put("first", "FIRST INSERTED");

hm.put("second", "SECOND INSERTED");

hm.put("third","THIRD INSERTED");

System.out.println(hm);

//getting value for the given key from hashmap

Set<Entry<String, String>> entires = hm.entrySet();

for(Entry<String,String> ent:entires){

System.out.println(ent.getKey()+" ==> "+ent.getValue());

}

}

**ConcurrentHashMap:**

**The Difference b/w ConcurrentHashMap and HashMap:**

1. The ConcurrentHashMap class provides a concurrent version of the standard HashMap. This is an improvement on the synchronizedMap functionality provided in the Collections class.

Unlike Hashtable and Synchronized Map, it never locks whole Map, instead it divides the map in segments and locking is done on those. It perform better if number of reader threads are greater than number of writer threads.

ConcurrentHashMap by default is separated into 16 regions and locks are applied. This default number can be set while initializing a ConcurrentHashMap instance. When setting data in a particular segment, the lock for that segment is obtained. This means that two updates can still simultaneously execute safely if they each affect separate buckets, thus minimizing lock contention and so maximizing performance.

1. ConcurrentHashMap also does not throw a ConcurrentModificationException

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

1. In HashMap, null values are allowed for key and values, whereas in ConcurrentHashMap null value is not allowed for key and value, otherwise we will get Run-time exception saying**NullPointerException.**

**Using Hashmap:**

// Java program to illustrate

// HashMap drawbacks

import java.util.HashMap;

class HashMapDemo extends Thread

{

static HashMap<Integer,String> l=new HashMap<Integer,String>();

public void run()

{

// Child thread trying to add

// new element in the object

l.put(103,"D");

try

{

Thread.sleep(1000);

}

catch(InterruptedException e)

{

System.out.println("Child Thread going to add element");

}

}

public static void main(String[] args) throws InterruptedException

{

l.put(100,"A");

l.put(101,"B");

l.put(102,"C");

HashMapDemo t=new HashMapDemo();

t.start();

for (Object o : l.entrySet())

{

Object s=o;

System.out.println(s);

Thread.sleep(1000);

}

System.out.println(l);

}

}

**Output:**

100=A

Exception in thread "main" java.util.ConcurrentModificationException

**Why?**

Because, the main thread as well as the thread created, are trying to access the HashMap simultaneously.

**Using ConcurrentHashMap:**

// Java program to illustrate

// HashMap drawbacks

import java.util.HashMap;

import java.util.concurrent.\*;

class HashMapDemo extends Thread

{

static ConcurrentHashMap<Integer,String> l =

new ConcurrentHashMap<Integer,String>();

public void run()

{

// Child add new element in the object

l.put(103,"D");

try

{

Thread.sleep(2000);

}

catch(InterruptedException e)

{

System.out.println("Child Thread going to add element");

}

}

public static void main(String[] args) throws InterruptedException

{

l.put(100,"A");

l.put(101,"B");

l.put(102,"C");

HashMapDemo t=new HashMapDemo();

t.start();

for (Object o : l.entrySet())

{

Object s=o;

System.out.println(s);

Thread.sleep(1000);

}

System.out.println(l);

}

}

**The output is:**

100=A

101=B

102=C

103=D

{100=A, 101=B, 102=C, 103=D}

**hashCode and equals In Java**

**And Why should we overload them in case of user defined object in case of the object is going to become a key in any of the Hash based collection?**

Because, in case of failure to do so, your Hash based collection can behave in a unpredictable behaviour.

Before going into deep, you must know about Shallow Comparison and Deep Comparison.

**Equals:**

**Shallow Comparison and Deep Comparison:**

**Shallow comparison:** The default implementation of equals method is defined in Java.lang.Object class which simply checks if two Object references (say x and y) refer to the same Object. i.e. It checks if x == y. Since Object class has no data members that define its state, it is also known as shallow comparison.

**Deep Comparison:** Suppose a class provides its own implementation of equals() method in order to compare the Objects of that class w.r.t state of the Objects. That means data members (i.e. fields) of Objects are to be compared with one another. Such Comparison based on data members is known as deep comparison.

**Now, if you don’t define equals method for your user defined object (before using it as hash based collection’s key, only shallow comparison will be taken place. There will be no deep comparison.**

**Now, even in case of references, (we are not going into deep comparison)** equals should follow the following conditions:

**Reflexive :** for any reference value a, a.equals(a) should return true.

**Symmetric :** for any reference values a and b, if a.equals(b) should return true then b.equals(a) must return true.

**Transitive :** for any reference values a, b, and c, if a.equals(b) returns true and b.equals(c) returns true, then a.equals(c) should return true.

**Consistent :** for any reference values a and b, multiple invocations of a.equals(b) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.  
  
  
**hashCode()**

It returns the hashcode value as an Integer. Hashcode value is mostly used in hashing based collections like HashMap, HashSet, HashTable….etc. This method must be overridden in every class which overrides equals() method.

**Syntax :**

public int hashCode()

// This method returns the hash code value

// for the object on which this method is invoked.

The general contract of hashCode is:

During the execution of the application, if hashCode() is invoked more than once on the same Object then it must consistently return the same Integer value, provided no information used in equals(Object) comparison on the Object is modified. It is not necessary that this Integer value to be remained same from one execution of the application to another execution of the same application.

If two Objects are equal, according to the the equals(Object) method, then hashCode() method must produce the same Integer on each of the two Objects.

If two Objects are unequal, according to the the equals(Object) method, It is not necessary the Integer value produced by hashCode() method on each of the two Objects will be distinct. It can be same but producing the distinct Integer on each of the two Objects is better for improving the performance of hashing based Collections like HashMap, HashTable…etc.

**Now, coming back to the main question  
  
Why to Override equals(Object) and hashCode() method ?**

**Case 1: Overriding both equals(Object) and hashCode() method**

You must override hashCode() in every class that overrides equals(). Failure to do so will result in a violation of the general contract for Object.hashCode(), which will prevent your class from functioning properly in conjunction with all hash-based collections, including HashMap, HashSet, and Hashtable. (-Joshua Bloch)

Here is the contract, from the java.lang.Object specialization:

Whenever it(hashcode) is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.

If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.

It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

// Java program to illustrate

// overriding of equals and

// hashcode methods

import java.io.\*;

import java.util.\*;

class Geek

{

String name;

int id;

Geek(String name, int id)

{

this.name = name;

this.id = id;

}

@Override

public boolean equals(Object obj)

{

// if both the object references are

// referring to the same object.

if(this == obj)

return true;

// it checks if the argument is of the

// type Geek by comparing the classes

// of the passed argument and this object.

// if(!(obj instanceof Geek)) return false; ---> avoid.

if(obj == null || obj.getClass()!= this.getClass())

return false;

// type casting of the argument.

Geek geek = (Geek) obj;

// comparing the state of argument with

// the state of 'this' Object.

return (geek.name == this.name && geek.id == this.id);

}

@Override

public int hashCode()

{

// We are returning the Geek\_id

// as a hashcode value.

// we can also return some

// other calculated value or may

// be memory address of the

// Object on which it is invoked.

// it depends on how you implement

// hashCode() method.

return this.id;

}

}

// Driver code

class GFG

{

public static void main (String[] args)

{

// creating two Objects with

// same state

Geek g1 = new Geek("aditya", 1);

Geek g2 = new Geek("aditya", 1);

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

map.put(g1, "CSE");

map.put(g2, "IT");

for(Geek geek : map.keySet())

{

System.out.println(map.get(geek).toString());

}

}

}

**Output:**

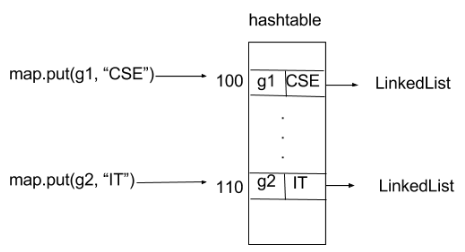
IT

In this case we override both methods properly.

When we call map.put(g1, “CSE”); it will hash to some bucket location and when we call map.put(g2, “IT”);, it will generates same hashcode value (same as g1) and replace first value by second value because while iterating over same bucket it found a k such that k.equals(g2) is true, means searching key already exist. So, it replaces old value of that key by new value.

**Case 2: Overriding only the equals(Object) method**

If we only override equals(Object) method, when we call map.put(g1, “CSE”); it will hash to some bucket location and when we call map.put(g2, “IT”); it will hash to some other bucket location because of different hashcode value as hashCode() method has not been overridden.

****

// Java program to illustrate

// Overriding only the equals(Object) method

import java.io.\*;

import java.util.\*;

class Geek

{

String name;

int id;

Geek(String name, int id)

{

this.name = name;

this.id = id;

}

@Override

public boolean equals(Object obj)

{

// if both the object references are

// referring to the same object.

if(this == obj)

return true;

// it checks if the argument is of the

// type Geek by comparing the classes

// of the passed argument and this object.

// if(!(obj instanceof Geek)) return false; ---> avoid.

if(obj == null || obj.getClass()!= this.getClass())

return false;

// type casting of the argument.

Geek geek = (Geek) obj;

// comparing the state of argument with

// the state of 'this' Object.

return (geek.name == this.name && geek.id == this.id);

}

}

class GFG

{

public static void main (String[] args)

{

// creating two Objects with

// same state

Geek g1 = new Geek("aditya", 1);

Geek g2 = new Geek("aditya", 1);

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

map.put(g1, "CSE");

map.put(g2, "IT");

for(Geek geek : map.keySet())

{

System.out.println(map.get(geek).toString());

}

}

}

**Output:**

CSE

IT

**Case 3: Overriding only hashCode() method**

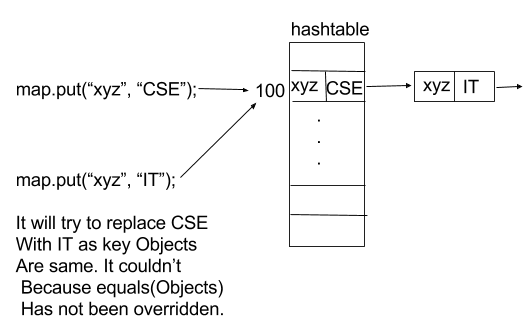
Consider another example of map :

Map map = new HashMap();

map.put(“xyz”, “CSE”);

map.put(“xyz”, “IT”);

When we call map.put(“xyz”, “CSE”); it will generate hashcode value and stores it to the bucket location that is specified with this address (hashcode value). And when we call map.put(“xyz”, “IT”); it generates same hashcode value as previous entry since key objects are same and hashCode() method has been overridden. So it should replace first with second as per rule. But it didn’t. Reason is, when it iterate through that bucket and seeks to find k such that k.equals(“xyz”) i.e. if searching key already exist. But it fails to find because equals(Object ) method has not been overridden. It is violation of rule of hashing.



// Java program to illustrate

// Overriding only hashCode() method

import java.io.\*;

import java.util.\*;

class Geek

{

String name;

int id;

Geek(String name, int id)

{

this.name = name;

this.id = id;

}

@Override

public int hashCode()

{

// We are returning the Geek\_id

// as a hashcode value.

// we can also return some

// other calculated value or may

// be memory address of the

// Object on which it is invoked.

// it depends on how you implement

// hashCode() method.

return this.id;

}

}

class GFG

{

public static void main (String[] args)

{

// creating two Objects with

// same state

Geek g1 = new Geek("aditya", 1);

Geek g2 = new Geek("aditya", 1);

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

map.put(g1, "CSE");

map.put(g2, "IT");

for(Geek geek : map.keySet())

{

System.out.println(map.get(geek).toString());

}

}

}

**Output:**

CSE  
IT