Can one use an Employee class as a key in a HashMap?

hashCode() -HashMap provides put(key, value) for storing and get(key) method for retrieving Values from HashMap. When put() method is used to store (Key, Value) pair, HashMap implementation calls hashcode on Key object to calculate a hash that is used to find a bucket where Entry object will be stored. When get() method is used to retrieve value, again key object is used to calculate a hash which is used then to find a bucket where that particular key is stored.

equals() - equals() method is used to compare objects for equality. In case of HashMap key object is used for comparison, also using equals() method Map knows how to handle hashing collision (hashing collision means more than one key having the same hash value, thus assigned to the same bucket. In that case objects are stored in a linked list, refer figure for more clarity. Where hashCode method helps in finding the bucket where that key is stored, equals method helps in finding the right key as there may be more than one key-value pair stored in a single bucket.

Answer to your question is yes, objects of custom classes can be used as a key in a HashMap. But in order to retrieve the value object back from the map without failure, there are certain guidelines that need to be followed.

1)Custom class should follow the contract between hashCode() and equals().

The contract states that:

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.

Yes, you should override equals and hashcode, for the proper functioning of the code otherwise you won't be able to get the value of the key which you have inserted in the map.

e.g

map.put(new Object() , "value") ;

when you want to get that value ,

map.get(new Object()) ; // This will always return null

Because with new Object() - new hashcode will be generated and it will not point to the expected bucket number on which value is saved, and if eventually bucket number comes to be same - it won't be able to match hashcode and even equals so it always return NULL .

```
package com.java.demo.map;

import java.util.HashMap;

public class TestMutableKey
{
   public static void main(String[] args)
   {
      //Create a HashMap with mutable key
      HashMap<Account, String> map = new HashMap<Account, String>();

      //Create key 1
      Account a1 = new Account(1);
      a1.setHolderName("A_ONE");
      //Create key 2
      Account a2 = new Account(2);
      a2.setHolderName("A_TWO");

      //Put mutable key and value in map
      map.put(a1, a1.getHolderName());
      map.put(a2, a2.getHolderName());

      //Change the keys state so hash map should be calculated again
      a1.setHolderName("Defaulter");
      a2.setHolderName("Bankrupt");

      //Success !! We are able to get back the values
      System.out.println(map.get(a1)); //Prints A_ONE
      System.out.println(map.get(a2)); //Prints A_TWO

      //Try with newly created key with same account number
      Account a3 = new Account(1);
      a3.setHolderName("A_THREE");

      //Success !! We are still able to get back the value for account number 1
      System.out.println(map.get(a3)); //Prints A_ONE
   }
}
```

Evaluation Parameter- overriding hashCode ,overriding equals ,the behavior if hashCode returns a constant (including get and put time complexity), the behavior if hashCode is not imlemented but equals is .

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.

Let's try to understand it with an example of what would happen if we override `equals()` without overriding `hashCode()` and attempt to use a `Map`.

Say we have a class like this and that two objects of `MyClass` are equal if their `importantField` is equal (with `hashCode()` and `equals()` generated by eclipse)

```
public class Employee {

    String name;
    int age;

    public Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == this)
            return true;
        if (!(obj instanceof Employee))
            return false;
        Employee employee = (Employee) obj;
        return employee.getAge() == this.getAge()
                && employee.getName() == this.getName();
    }
```

```
    // commented
    /*  @Override
        public int hashCode() {
            int result=17;
            result=31*result+age;
            result=31*result+(name!=null ? name.hashCode():0);
            return result;
        }
     */
}
```

Now create a class, insert Employee object into a HashSet and test whether that object is present or not.

```
public class ClientTest {
    public static void main(String[] args) {
        Employee employee = new Employee("rajeev", 24);
        Employee employee1 = new Employee("rajeev", 25);
        Employee employee2 = new Employee("rajeev", 24);

        HashSet<Employee> employees = new HashSet<Employee>();
        employees.add(employee);
        System.out.println(employees.contains(employee2));
        System.out.println("employee.hashCode():  " + employee.hashCode()
        + "  employee2.hashCode():" + employee2.hashCode());
    }
}
```

It will print the following:

```
false
employee.hashCode():  321755204  employee2.hashCode():375890482
```

Collections such as HashMap and HashSet use a *hashcode* value of an object to determine how it should be stored inside a collection, and the *hashcode* is used again in order to locate the object in its collection.

Hashing retrieval is a two-step process:

1. Find the right bucket (using hashCode())
2. Search the bucket for the right element (using equals())

Here is a small example on why we should overrride equals() and hashcode().

Consider an Employee class which has two fields: age and name.

```
public class Employee {

    String name;
    int age;
```

```
    public Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == this)
            return true;
        if (!(obj instanceof Employee))
            return false;
        Employee employee = (Employee) obj;
        return employee.getAge() == this.getAge()
                && employee.getName() == this.getName();
    }

    // commented
    /*  @Override
        public int hashCode() {
            int result=17;
            result=31*result+age;
            result=31*result+(name!=null ? name.hashCode():0);
            return result;
        }
     */
}
```

Now create a class, insert Employee object into a HashSet and test whether that object is present or not.

```
public class ClientTest {
    public static void main(String[] args) {
        Employee employee = new Employee("rajeev", 24);
        Employee employee1 = new Employee("rajeev", 25);
        Employee employee2 = new Employee("rajeev", 24);

        HashSet<Employee> employees = new HashSet<Employee>();
        employees.add(employee);
        System.out.println(employees.contains(employee2));
        System.out.println("employee.hashCode():  " + employee.hashCode()
        + "  employee2.hashCode():" + employee2.hashCode());
    }
}
```

It will print the following:

```
false
employee.hashCode():  321755204  employee2.hashCode():375890482
```

Now uncomment hashcode() method , execute the same and the output would be:

```
true
employee.hashCode():  -938387308  employee2.hashCode():-938387308
```

Now can you see why if two objects are considered equal, their *hashcode*s must also be equal? Otherwise, you'd never be able to find the object since the default *hashcode* method in class Object virtually always comes up with a unique number for each object, even if the equals() method is overridden in such a way that two or more objects are considered equal. It doesn't matter how equal the objects are if their *hashcode*s don't reflect that. So one more time: If two objects are equal, their *hashcode*s must be equal as well.0

You are using a class from a library (say Student). You have a list of Student objects. You need to sort this list based on first name. How will you do it? Constraint: (You do not have the ability to change the source code of the Student class)

You can implement your own Comparator without extending the class, and use it as a parameter for Collections.sort:

```
public class MyComparator implements Comparator<MyClass> {
   @Override
   int compate (MyClass a, MyClass b) {
      // null handling removed for clarity's sake
      // add it if you need it
      return a.getName().compareTo(b.getName());
   }
}
```

And now just use it:

```
List<MyClass> myList = ...;
Collections.sort (myList, new MyComparator());
```

You will have to implement your own comparator and use Collections.sort for your collection. Something like this:

```
class CustomComparator implements Comparator<YourObject> {

   @Override
   public int compare(YourObject o1, YourObject o2) {
     return o1.getName().compareTo(o2.getName());
   }
}
```

Then use the Comparator as

```
Collections.sort(yourCollection,new CustomComparator());
```

3.Consider a class Person with two attributes - String name and List<String> degrees. How will you make this class immutable. Ask what are the advantanges of immutable classes

Evaluation Parameter- Make all fields private and final,Remove the setter methods, Return either a new copy of degrees or an unmodifiableList in the the getter method, In the constructor, make a copy of the List argument passed, Make the class final, Knows the benefits of immutable class in multi-threaded programs and as a good design practice**.**

Q :: Consider a class A with a synchronized method

class A {

public void synchronized m1() {Thread.sleep(5000);}

}

We create two objects of this class - o1 and o2. We call o1.m1() on one thread and o2.m1() on another thread, at the same time. What will be the behaviour?

Follow up with - how will you force these calls to execute one after the other

1. Understands object level synchronization. These two calls do not block each other.

2. Suggests a solution using a shared lock object or class level synchronization.

• Both methods m1() and m2() are synchronized.
• Thus, for a given object, several threads cannot access its method simultaneously.
• Consider the first case :
• One thread wants to execute o1.m1().
• Another thread wants to execute o1.m2().
• First possibility:
• Thread 1 is executed. Entering the function, it aquires the lock on o1.
• From that point, even if Thread 2 wants to execute, it will be blocked outside of m2().
• Thread 1 executes o1.m1() until completion, then releases the lock on o1.
• Thread 2 now can execute o1.m2().
• Second possibility:
• Same as before, but Thread 2 is executed first, locks Thread 1 outside of m1(), finishes o1.m2(), releases the lock and Thread 1 can execute o1.m1().

Given a List of integers (List<Integer>), write code in Java 8 style to get the sum of the squares of all the odd numbers in the array.
Evaluation Parameter 1. Used stream() correctly.
2. Used filter() correctly.
3. Used map() correctly.
4. Used sum() or reduce() correctly.

```
list.stream()
            .filter(i -> i % 2 !=0)
            .mapToInt(i -> i)
            .sum();
```

What is the difference between HashMap and ConcurrentHashMap

HashMap is the Class which is under Traditional Collection and ConcurrentHashMap is a Class which is under Concurrent Collections, apart from this there are various differences between them which are:

- HashMap is non-Synchronized in nature i.e. HashMap is not Thread-safe whereas ConcurrentHashMap is Thread-safe in nature.
- HashMap performance is relatively high because it is non-synchronized in nature and any number of threads can perform simultaneously. But ConcurrentHashMap performance is low sometimes because sometimes Threads are required to wait on ConcurrentHashMap.
- While one thread is Iterating the HashMap object, if other thread try to add/modify the contents of Object then we will get Run-time exception saying **ConcurrentModificationException**.Whereas In ConcurrentHashMap we wont get any exception while performing any modification at the time of Iteration.
  Using HashMap

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

- HashMap is introduced in JDK 1.2 whereas ConcurrentHashMap is introduced by SUN Microsystem in JDK 1.5.

For your needs, use ConcurrentHashMap. It allows concurrent modification of the Map from several threads without the need to block them. Collections.synchronizedMap(map) creates a blocking Map which will degrade performance, albeit ensure consistency (if used properly).

Use the second option if you need to ensure data consistency, and each thread needs to have an up-to-date view of the map. Use the first if performance is critical, and each thread only inserts data to the map, with reads happening less frequently.

Synchronized Map:

Synchronized Map is also not very different than Hashtable and provides similar performance in concurrent Java programs. Only difference between Hashtable and SynchronizedMap is that SynchronizedMap is not a legacy and you can wrap any Map to create it's synchronized version by using Collections.synchronizedMap() method.

ConcurrentHashMap:

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.

ConcurrentHashMap doesn't throw a ConcurrentModificationException

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

Difference between synchornizedMap and ConcurrentHashMap

Collections.synchornizedMap(HashMap) will return a collection which is almost equivalent to Hashtable, where every modification operation on Map is locked on Map object while in case of ConcurrentHashMap, thread-safety is achieved by dividing whole Map into different partition based upon concurrency level and only locking particular portion instead of locking whole Map.

ConcurrentHashMap does not allow null keys or null values while synchronized HashMap allows one null keys.

In ConcurrentHashMap, the lock is applied to a segment instead of an entire Map. Each segment manages its own internal hash table. The lock is applied only for update operations. Collections.synchronizedMap(Map) synchronizes the entire map.

8.Explain what the following command does on Unix:
chmod 764 file1

1. Knows that this command changes the access permissions on the file..
2. Can explain the meaning of 764 - read,write and execute permission for the owner, read and write permission for the group and read permission for everyone.

764' absolute code says the following:
Owner can read, write and execute.
Usergroup can read and write.
World can only read

We have a table called BookAuthor. It has two columns Book and Author, Book being unique column. Write a query to find the names of the authors who have written more than 10 books.

You could use a subquery, which will find all book ids which have more than two authors:

```
SELECT c.book_id
  FROM book_author c
  GROUP BY c.book_id
  HAVING count(c.book_id) >= 10
```

... and then use this in your main query, to get all authors & books from your joined tables whos book id appears in the subquery:

```
SELECT ba.author_id, a.name AS author_name, b.title
FROM
    book_author ba JOIN book b ON ba.book_id = b.book_id
    JOIN author a ON ba.author_id = a.author_id
WHERE
    b.book_id IN (SELECT c.book_id
                  FROM book_author c
                  GROUP BY c.book_id
                  HAVING count(c.book_id) >= 2);
```