Q1)

# Why do I need to override the equals and hashCode methods in Java?

ans: 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.

example:

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 MyClass {

private final String importantField;

private final String anotherField;

public MyClass(final String equalField, final String anotherField) {

this.importantField = equalField;

this.anotherField = anotherField;

}

@Override

public int hashCode() {

final int prime = 31;

int result = 1;

result = prime \* result

+ ((importantField == null) ? 0 : importantField.hashCode());

return result;

}

@Override

public boolean equals(final Object obj) {

if (this == obj)

return true;

if (obj == null)

return false;

if (getClass() != obj.getClass())

return false;

final MyClass other = (MyClass) obj;

if (importantField == null) {

if (other.importantField != null)

return false;

} else if (!importantField.equals(other.importantField))

return false;

return true;

}

}

Imagine you have this

MyClass first = new MyClass("a","first");

MyClass second = new MyClass("a","first");

# Override only equals:

======================

If only equals is overriden, then when you call myMap.put(first,someValue) first will hash to some bucket

and when you call myMap.put(second,someOtherValue) it will hash to some other bucket (as they have

a different hashCode). So, although they are equal, as they don't hash to the same bucket,

the map can't realize it and both of them stay in the map.

Although it is not necessary to override equals() if we override hashCode(),

let's see what would happen in this particular case where we know that two objects of MyClass are equal

if their importantField is equal but we do not override equals().

# Override only hashCode:

=======================

If you only override hashCode then when you call myMap.put(first,someValue) it takes first,

calculates its hashCode and stores it in a given bucket.

Then when you call myMap.put(second,someOtherValue) it should replace first with second as per the

Map Documentation because they are equal (according to the business requirement).

But the problem is that equals was not redefined, so when the map hashes second and

iterates through the bucket looking if there is an object k such that second.equals(k) is true it

won't find any as second.equals(first) will be false.

Note:

You need to override hashCode if your class overrides equals but reverse is not true.

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:

Find the right bucket (using hashCode())

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 hashcodes 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 hashcodes don't reflect that.

So one more time: If two objects are equal, their hashcodes must be equal as well.

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.

from Effective Java, by Joshua Bloch

# Identity is not equality.

===========================

In order to identity differences between two objects we need to override equals method.

Identity is not equality.

equals operator == test identity.

equals(Object obj) method compares equality test(i.e. we need to tell equality by overriding the method)

Why do I need to override the equals and hashCode methods in Java?

First we have to understand the use of equals method.

In order to identity differences between two objects we need to override equals method.

For example:

Customer customer1=new Customer("peter");

Customer customer2=customer1;

customer1.equals(customer2); // returns true by JVM. i.e. both are refering same Object

------------------------------

Customer customer1=new Customer("peter");

Customer customer2=new Customer("peter");

customer1.equals(customer2); //return false by JVM i.e. we have two different peter customers.

------------------------------

Now I have overriden Customer class equals method as follows:

@Override

public boolean equals(Object obj) {

if (this == obj) // it checks references

return true;

if (obj == null) // checks null

return false;

if (getClass() != obj.getClass()) // both object are instances of same class or not

return false;

Customer other = (Customer) obj;

if (name == null) {

if (other.name != null)

return false;

} else if (!name.equals(other.name)) // it again using bulit in String object equals to identify the difference

return false;

return true;

}

Customer customer1=new Customer("peter");

Customer customer2=new Customer("peter");

Insteady identify the Object equality by JVM, we can do it by overring equals method.

customer1.equals(customer2); // returns true by our own logic

# Now hashCode method can understand easily.

==========================================

hashCode produces integer in order to store object in data structures like HashMap, HashSet.

Assume we have override equals method of Customer as above,

customer1.equals(customer2); // returns true by our own logic

While working with data structure when we store object in buckets(bucket is a fancy name for folder).

If we use built-in hash technique, for above two customers it generates two different hashcode.

So we are storing the same identical object in two different places.

To avoid this kind of issues we should override the hashCode method also based on the following principles.

un-equal instances may have same hashcode.

equal instances should return same hashcode.

# Q2) Why we override equals() method ?

Ans:

In Java we can not overload how operators like ==, +=, -+ behave. They are behaving a certain way.

Java doesn't supports operator overloading because it's just a choice made by its creators

who wanted to keep the language more simple. ... Operator overloading allows you to do something extra

than what for it is expected for. Java only allows arithmetic operations on elementary numeric types

So let's focus on the operator == for our case here.

How operator == works.

It checks if 2 references that we compare point to the same instance in memory.

Operator == will resolve to true only if those 2 references represent the same instance in memory.

So now let's consider the following example

public class Person {

private Integer age;

private String name;

..getters, setters, constructors

}

So let's say that in your program you have built 2 Person objects on different places and

you wish to compare them.

Person person1 = new Person("Mike", 34);

Person person2 = new Person("Mike", 34);

System.out.println ( person1 == person2 ); --> will print false!

Those 2 objects from business perspective look the same right? For JVM they are not the same.

Since they are both created with new keyword those instances are located in different segments in memory.

Therefore the operator == will return false

But if we can not override the == operator how can we say to JVM that we want those 2 objects

to be treated as same. There comes the .equals() method in play.

You can override equals() to check if some objects have same values for specific fields to be

considered equal.

You can select which fields you want to be compared. If we say that 2 Person objects will be the same

if and only if they have the same age and same name, then the IDE will create something like the

following for automatic generation of equals()

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

Person person = (Person) o;

return age == person.age &&

name.equals(person.name);

}

Let's go back to our previous example

Person person1 = new Person("Mike", 34);

Person person2 = new Person("Mike", 34);

System.out.println ( person1 == person2 ); --> will print false!

System.out.println ( person1.equals(person2) ); --> will print true!

So we can not overload == operator to compare objects the way we want but Java gave us another way,

the equals() method, which we can override as we want.

Keep in mind however, if we don't provide our custom version of .equals() (aka override)

in our class then the predefined .equals() from Object class and == operator will behave exactly the same.

Default equals() method which is inherited from Object will check whether both compared instances

are the same in memory!

Why we override equals() method

In Java we can not overload how operators like ==, +=, -+ behave. They are behaving a certain way. So let's focus on the operator == for our case here.

How operator == works.

It checks if 2 references that we compare point to the same instance in memory. Operator == will resolve to true only if those 2 references represent the same instance in memory.

So now let's consider the following example

public class Person {

private Integer age;

private String name;

..getters, setters, constructors

}

So let's say that in your program you have built 2 Person objects on different places and you wish to compare them.

Person person1 = new Person("Mike", 34);

Person person2 = new Person("Mike", 34);

System.out.println ( person1 == person2 ); --> will print false!

Those 2 objects from business perspective look the same right? For JVM they are not the same. Since they are both created with new keyword those instances are located in different segments in memory. Therefore the operator == will return false

But if we can not override the == operator how can we say to JVM that we want those 2 objects to be treated as same. There comes the .equals() method in play.

You can override equals() to check if some objects have same values for specific fields to be considered equal.

You can select which fields you want to be compared. If we say that 2 Person objects will be the same if and only if they have the same age and same name, then the IDE will create something like the following for automatic generation of equals()

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

Person person = (Person) o;

return age == person.age &&

name.equals(person.name);

}

Let's go back to our previous example

Person person1 = new Person("Mike", 34);

Person person2 = new Person("Mike", 34);

System.out.println ( person1 == person2 ); --> will print false!

System.out.println ( person1.equals(person2) ); --> will print true!

So we can not overload == operator to compare objects the way we want but Java gave us another way,

the equals() method, which we can override as we want.

Keep in mind however, if we don't provide our custom version of .equals() (aka override) in our class

then the predefined .equals() from Object class and == operator will behave exactly the same.

Default equals() method which is inherited from Object will check whether both compared instances

are the same in memory!

# Q3) Why we override hashCode() method?

===============================================

Some Data Structures in java like HashSet, HashMap store their elements based on a hash function which

is applied on those elements. The hashing function is the hashCode()

If we have a choice of overriding .equals() method then we must also have a choice of overriding

HashCode () method. There is a reason for that.

Default implementation of hashCode() which is inherited from Object considers all objects

in memory unique!

Let's get back to those hash data structures. There is a rule for those data structures.

HashSet cannot contain duplicate values and HashMap cannot contain duplicate keys

HashSet is implemented with a HashMap behind the scenes where each value of a HashSet

is stored as a key in a HashMap.

# So we have to understand how a HashMap works.

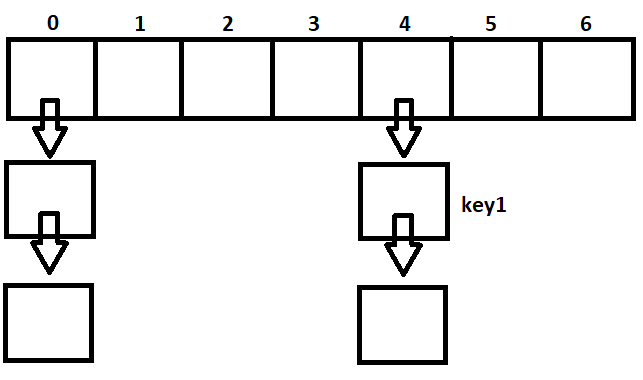
In a simple way a HashMap is a native array that has some buckets. Each bucket has a linkedList.

In that linkedList our keys are stored. HashMap locates the correct linkedList for each key by applying

hashCode() method and after that it iterates through all elements of that linkedList and

applies equals() method on each of these elements to check if that element is already contained there.

No duplicate keys are allowed.



When we put something inside a HashMap, the key is stored in one of those linkedLists. In which linkedList that key will be stored is shown by the result of hashCode() method on that key. So if key1.hashCode() has as a result 4, then that key1 will be stored on the 4th bucket of the array, in the linkedList that exists there.

By default hashCode() method returns a different result for each different instance. If we have the default equals() which behaves like == which considers all instances in memory as different objects we don't have any problem.

But in our previous example we said we want Person instances to be considered equal if their ages and names match.

# Why we override equals() method

In Java we can not overload how operators like ==, +=, -+ behave. They are behaving a certain way. So let's focus on the operator == for our case here.

How operator == works.

It checks if 2 references that we compare point to the same instance in memory. Operator == will resolve to true only if those 2 references represent the same instance in memory.

So now let's consider the following example

public class Person {

private Integer age;

private String name;

..getters, setters, constructors

}

So let's say that in your program you have built 2 Person objects on different places and you wish to compare them.

Person person1 = new Person("Mike", 34);

Person person2 = new Person("Mike", 34);

System.out.println ( person1 == person2 ); --> will print false!

Those 2 objects from business perspective look the same right? For JVM they are not the same. Since they are both created with new keyword those instances are located in different segments in memory. Therefore the operator == will return **false**

But if we can not override the == operator how can we say to JVM that we want those 2 objects to be treated as same. There comes the .equals() method in play.

You can override equals() to check if some objects have same values for specific fields to be considered equal.

You can select which fields you want to be compared. If we say that 2 Person objects will be the same if and only if they have the same age and same name, then the IDE will create something like the following for automatic generation of equals()

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

Person person = (Person) o;

return age == person.age &&

name.equals(person.name);

}

Let's go back to our previous example

Person person1 = new Person("Mike", 34);

Person person2 = new Person("Mike", 34);

System.out.println ( person1 == person2 ); --> will print false!

System.out.println ( person1.equals(person2) ); --> will print true!

So we can not overload == operator to compare objects the way we want but Java gave us another way, the equals() method, which we can override as we want.

**Keep in mind** however, if we don't provide our custom version of .equals() (aka override) in our class then the predefined .equals() from Object class and == operator will behave exactly the same.

Default equals() method which is inherited from Object will check whether both compared instances are the same in memory!

# Why we override hashCode() method

Some Data Structures in java like HashSet, HashMap store their elements based on a hash function which is applied on those elements. The hashing function is the hashCode()

If we have a choice of overriding .equals() method then we must also have a choice of overriding hashCode() method. There is a reason for that.

Default implementation of hashCode() which is inherited from Object considers all objects in memory unique!

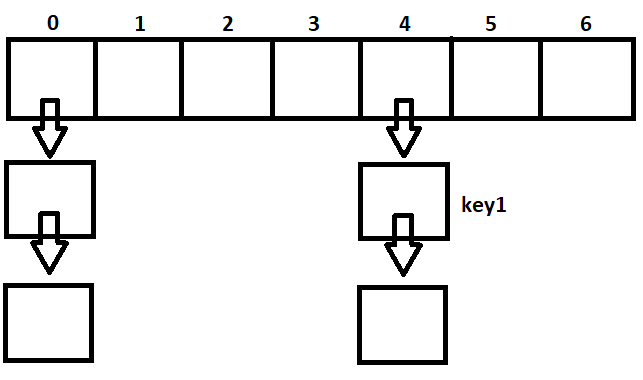
Let's get back to those hash data structures. There is a rule for those data structures.

HashSet can not contain duplicate values and HashMap can not contain duplicate keys

*HashSet is implemented with a HashMap behind the scenes where each value of a HashSet is stored as a key in a HashMap.*

So we have to understand how a HashMap works.

In a simple way a HashMap is a native array that has some buckets. Each bucket has a linkedList. In that linkedList our keys are stored. HashMap locates the correct linkedList for each key by applying hashCode() method and after that it iterates through all elements of that linkedList and applies equals() method on each of these elements to check if that element is already contained there. No duplicate keys are allowed.

[](https://i.stack.imgur.com/aMwel.png)

When we put something inside a HashMap, the key is stored in one of those linkedLists. In which linkedList that key will be stored is shown by the result of hashCode() method on that key. So if key1.hashCode() has as a result 4, then that key1 will be stored on the 4th bucket of the array, in the linkedList that exists there.

By default hashCode() method returns a different result for each different instance. If we have the default equals() which behaves like == which considers all instances in memory as different objects we don't have any problem.

But in our previous example we said we want Person instances to be considered equal if their ages and names match.

Person person1 = new Person("Mike", 34);

Person person2 = new Person("Mike", 34);

System.out.println ( person1.equals(person2) ); --> will print true!

Now let's create a map to store those instances as keys with some string as pair value

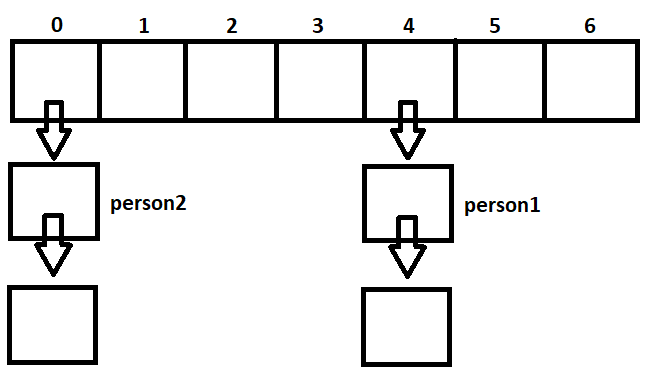
Map<Person, String> map = new HashMap();

map.put(person1, "1");

map.put(person2, "2");

In Person class we have not overridden the hashCode method but we have overridden equals method. Since the default hashCode provides different results for different java instances person1.hashCode() and person2.hashCode() have big chances of having different results.

Our map might end with those persons in different linkedLists.



This is against the logic of a HashMap

**A HashMap is not allowed to have multiple equal keys!**

But ours now has and the reason is that the default hashCode() which was inherited from Object Class was not enough. Not after we have overridden the equals() method on Person Class.

That is the reason why we must override hashCode() method after we have overridden equals method.

Now let's fix that. Let's override our hashCode() method to consider the same fields that equals() considers, namely age, name

public class Person {

private Integer age;

private String name;

..getters, setters, constructors

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

Person person = (Person) o;

return age == person.age &&

name.equals(person.name);

}

@Override

public int hashCode() {

return Objects.hash(name, age);

}

}

Now let's try again to save those keys in our HashMap

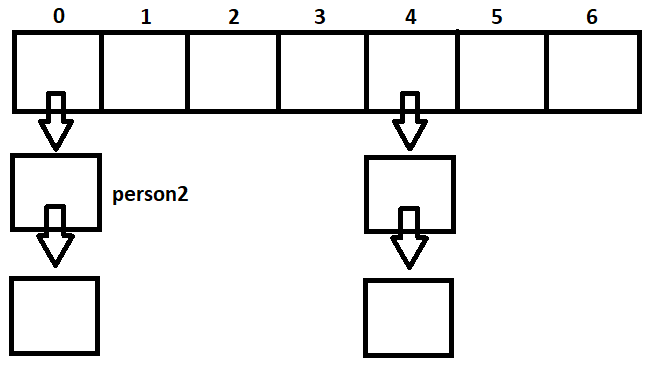
Map<Person, String> map = new HashMap();

map.put(person1, "1");

map.put(person2, "2");

person1.hashCode() and person2.hashCode() will definitely be the same. Let's say it is 0.

HashMap will go to bucket 0 and in that LinkedList will save the person1 as key with the value "1". For the second put HashMap is intelligent enough and when it goes again to bucket 0 to save person2 key with value "2" it will see that another equal key already exists there. So it will overwrite the previous key. So in the end only person2 key will exist in our HashMap.



Now we are aligned with the rule of Hash Map that says no multiple equal keys are allowed!

**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.

**I have successfully implemented equals() and it is working great.So why are they asking to override hashCode() as well?**

Well.As long as you don't use "Hash" based Collections on your user-defined class,it is fine. But some time in the future you might want to use HashMap or HashSet and if you don't override and "correctly implement" hashCode(), these Hash based collection won't work as intended.

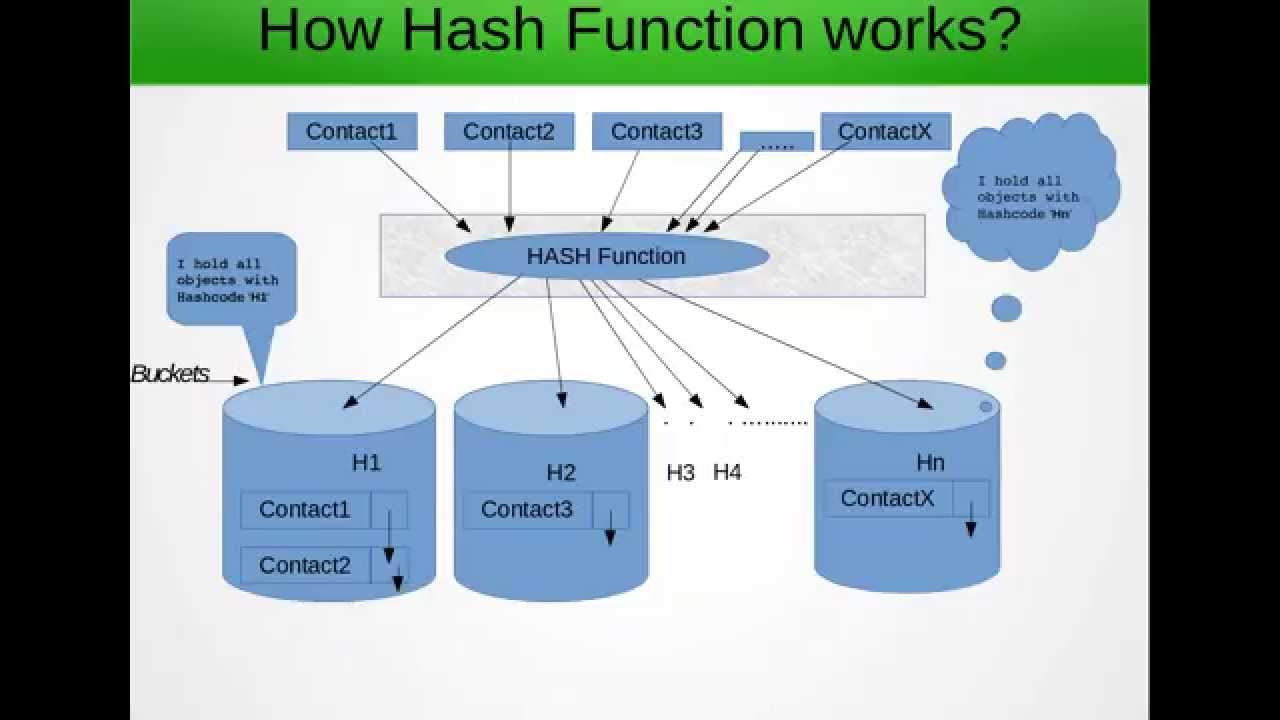
# Equals and Hashcode methods in Java

They are methods of java.lang.Object class which is the super class of all the classes (custom classes as well and others defined in java API).

Implementation:

public boolean equals(Object obj)

public int hashCode()



# **public boolean equals(Object obj)**

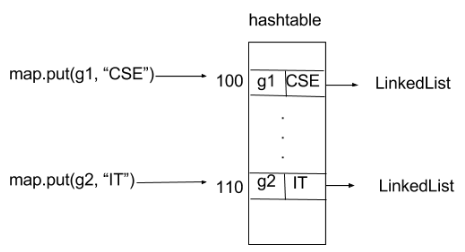
This method simply checks if two object references x and y refer to the same object. i.e. It checks if x == y.

It is reflexive: for any reference value x, x.equals(x) should return true.

It is symmetric: for any reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.

It is transitive: for any reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.

It is consistent: for any reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.



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());

        }

    }

}

**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

