# How do I prove that Object.hashCode() can produce similar hash code for two different objects in Java?



Had a discussion with an interviewer regarding internal implementation of Java Hashmaps and how it would behave if we overrode equals() but not the HashCode() method for an Employee object.

He told me that hashCode for two different objects would never be the same for the default object.hashCode() implementation, unless we overrode the hashCode() ourselves.

Also, I was told that one bucket can only have unique Hashcode in it, and objects with same hashcodes went in one bucket. Which I know contradicts point one. Duh!

From what I remembered, i told him that Java Hashcode contracts says that two different objects may have the same hashcode().

According to my interviewer, the default object.hashcode() never have the same hashcode() for two different objects, Is this true?

Is it even remotely possible to write a code that demonstrates this. From what I understand, Object.hashcode() can produce 2^30 unique values, how does one produce a collision, with such low possibility of collision to demonstrate that two different objects can get the same hashcode() with the Object classes method.

Or is he right, with the default Object.HashCode() implementation, we will never have a collision i.e two different objects can never have the same HashCode. If so, why do so many java manuals don't explicitly say so.

How can I write some code to demonstrate this? Because on demonstrating this, I can also prove that a bucket in a hashmap can contain different HashCodes(I tried to show him the debugger where the

hashMap was expanded but he told me that this is just logical Implementation and not the internal algo?)

java algorithm collections hashcode

edited Dec 6 '16 at 17:35

asked Dec 2 '16 at 11:22

Sameer

**181** 1 4 21

1 @Kayaman In case of strings, the hashcode() method is unique as it takes into account char at each index and then computes a hash, guaranteeing that different objects with same strings get the same hashcode which may extend to different strings in soem cases, this question is about cases where hashcode() is not overidden and how the default implementation behaves. please remove duplicate tag. – Sameer Dec 2 '16 at 11:32

1 A 32-bit hashcode in a 64-bit heap cannot be guaranteed unique. – Kayaman Dec 2 '16 at 11:39

2 The Pigeonhole princriple states *if `n` items are put into `m` containers, with `n > m`, then at least one container must contain more than one item.* – Elliott Frisch Dec 2 '16 at 11:42

3 Your interviewer is wrong - thanks to the birthday paradox the number of elements needed to achieve a 50% probability of a collision in the 2^31 space of possible hashcodes is only of the order of 2^16 – Alnitak Dec 2 '16 at 13:13

1 Sorry, you probably didn't get the job. Then again, if the interviewer doesn't realize that his expected answer is wrong, then maybe you didn't want to work there anyway. :-/ Here is a +1 for consolation. :-) – Stuart Marks Dec 2 '16 at 16:21

## 4 Answers

2^30 unique values sounds like a lot but the birthday problem means we don't need many objects to get a collision.

The following program works for me in about a second and gives a collision between objects 196 and 121949. I suspect it will heavily depend on your system configuration, compiler version etc.

As you can see from the implementation of the `Hashable` class, every one is guarenteed to be unique and yet there are still collisions.

```
class HashCollider
{
    static class Hashable
    {
```

```
        private static int curr_id = 0;
        public  final  int id;

        Hashable()
        {
            id = curr_id++;
        }
    }

    public static void main(String[] args)
    {
        final int NUM_OBJS = 200000; // birthday problem suggests
                                     // this will be plenty

        Hashable objs[] = new Hashable[NUM_OBJS];
        for (int i = 0; i < NUM_OBJS; ++i) objs[i] = new Hashable();

        for (int i = 0; i < NUM_OBJS; ++i)
        {
            for (int j = i + 1; j < NUM_OBJS; ++j)
            {
                if (objs[i].hashCode() == objs[j].hashCode())
                {
                    System.out.println("Objects with IDs " + objs[i].id
                                       + " and " + objs[j].id + " collided.");
                    System.exit(0);
                }
            }
        }

        System.out.println("No collision");
    }
}
```

answered Dec 2 '16 at 12:24

Michael

**14.4k** 6 29 62

Thanks! I see that up unto 120000, there are no collisions but post that they start appearing, in a sample set of 200,000 objects, I found 9 collisions. All between 120,000 and 200.000 – Sameer Dec 2 '16 at 16:08

I tried to extend the same code to show we can get collisions on hashmap but I am getting strange behaviour, can you have a look here stackoverflow.com/questions/41635231/… – Sameer Jan 13 '17 at 13:02

If you have a large enough heap (assuming 64 bit address space) and objects are small enough (the smallest object size on a 64 bit JVM is 8 bytes), then you will be able to represent more than 2^32 objects that are reachable at the same time. At that point, the objects' identity hashcodes cannot be unique.

However, you don't need a monstrous heap. If you create a large enough pool of objects (e.g. in a large array) and randomly delete and recreate them, it is (I think) guaranteed that you will get a hashcode collision ... if you continue doing this long enough.

- The default algorithm for hashcode in older versions of Java is based on the address of the object *when hashcode is first called*. If the garbage collector moves an object, and another one is created at the original address of the first one, and identityHashCode is called, then the two objects will have the same identity hashcode.
- The current (Java 8) default algorithm uses a PRNG. The "birthday paradox" formula will tell you the probability that one object's identity hashcode is the same as one more of the other's.

The `-XXhashCode=n` option that @BastianJ mentioned has the following behavior:

- hashCode == 0: Returns a freshly generated pseudo-random number
- hashCode == 1: XORs the object address with a pseudo-random number that changes occasionally.
- hashCode == 2: The hashCode is 1! (Hence @BastianJ's "cheat" answer.)
- hashCode == 3: The hashcode is an ascending sequence number.
- hashCode == 4: the bottom 32 bits of the object address
- hashCode >= 5: This is the default algorithm for Java 8. It uses Marsaglia's xor-shift PRNG with a thread specific seed.

If you have downloaded the OpenJDK Java 8 source code, you will find the implementation in `hotspot/src/share/vm/runtime/synchronizer.cp`. Look for the `get_next_hash()` method.

So that is another way to prove it. Show him the source code!

edited Dec 2 '16 at 13:11

answered Dec 2 '16 at 12:00

Stephen C

**486k** 66 536 882

I tried to extend the same code to show we can get collisions on hashmap but I am getting strange behaviour, can you have a look here stackoverflow.com/questions/41635231/… – Sameer Jan 13 '17 at 13:03

Use Oracle JVM and set -XX:hashCode=2. If I remember corretly, this chooses the Default implementation to be "constant 1". Just for the purpose of proving you're right.

answered Dec 2 '16 at 12:00

Bastian J

**128** 5

Yea ... but that is cheating :-) – Stephen C Dec 2 '16 at 12:11

2 Kinda, but it makes clear that there is no such thing as "the default implementation". It depends on the JVM. You could also write your own spec-compliant JVM that only returns 0 or 1 as hashCode by default. There is absolutely *no* assertions you should make about the implementation of hashCode() except that is does not Change for the same object. Depending on the JVM, there are different implementations, and in case of HotSpot, there are many. – Bastian J Dec 2 '16 at 12:55

I have little to add to Michael's answer (+1) except a bit of code golfing and statistics.

The Wikipedia article on the Birthday problem that Michael linked to has a nice table of the number of events necessary to get a collision, with a desired probability, given a value space of a particular size. For example, Java's `hashCode` has 32 bits, giving a value space of 4 billion. To get a collision with a probability of 50%, about 77,000 events are necessary.

Here's a simple way to find two instances of `Object` that have the same `hashCode`:

```
static int findCollision() {
    Map<Integer,Object> map = new HashMap<>();
    Object n, o;

    do {
        n = new Object();
```

```
        o = map.put(n.hashCode(), n);
    } while (o == null);

    assert n != o && n.hashCode() == o.hashCode();
    return map.size() + 1;
}
```

This returns the number of attempts it took to get a collision. I ran this a bunch of times and generated some statistics:

```
    System.out.println(
        IntStream.generate(HashCollisions::findCollision)
                 .limit(1000)
                 .summaryStatistics());
```

```
IntSummaryStatistics{count=1000, sum=59023718, min=635, average=59023.718000,
max=167347}
```

This seems quite in line with the numbers from the Wikipedia table. Incidentally, this took only about 10 seconds to run on my laptop, so this is far from a pathological case.

You were right in the first place, but it bears repeating: hash codes are not unique!

edited May 23 '17 at 10:29

Community ♦

**1** 1

answered Dec 2 '16 at 16:35

Stuart Marks

**70.8k** 23 127 190

I tried to extend the same code to show we can get collisions on hashmap but I am getting strange behaviour, can you have a look here stackoverflow.com/questions/41635231/… – Sameer Jan 13 '17 at 13:03