What is an object's hash code if hashCode() is not overridden?

Ask Question

If the hashCode() method is not overridden, what will be the result of invoking hashCode() on any object in Java?

java object hashcode

edited Jan 31 '13 at 0:39

Peter O.
19.4k 9 56 66

asked Feb 10 '10 at 15:02

java_geek
6,326 23 70 92

2 check out System.identityHashCode() it gives you the default hashcode - the one that would have been returned if you had not overridden the method. – Ustaman Sangat Dec 14 '11 at 15:58

11 Answers

Typically, hashCode() just returns the object's address in memory if you don't override it.

From 1:

> As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the JavaTM programming language.)

edited Nov 13 '14 at 20:47

assiegee
82 1 10

answered Feb 10 '10 at 15:06

Hans W
3,282 15 20

16 I have strong objection to "converting the internal address" and I have been wondering if Sun/Oracle will remove that

line from their javadocs. Internal address of the object cannot be guaranteed to remain unchanged in the JVM, whose garbage collector might move it around during heap compaction. – Ustaman Sangat Dec 14 '11 at 15:13

Does that mean they are unique all the time? – ernesto May 22 '15 at 12:44

15 The JavaDoc quote is correct, but the answer is not. Object's address is not used for many years. – Tagir Valeev Sep 8 '15 at 7:01

Despite that Javadoc, this answer is wrong. – Raedwald Dec 5 '16 at 6:02

1 @ernesto No they are not uniques all the time. I tested it on my JVM and found that after 120,000 objects, i start seeing duplicate hashcode. You can read more here stackoverflow.com/questions/40931088/… – Sameer Dec 5 '16 at 13:17

In HotSpot JVM by default on the first invocation of non-overloaded `Object.hashCode` or `System.identityHashCode` a random number is generated and stored in the object header. The consequent calls to `Object.hashCode` or `System.identityHashCode` just extract this value from the header. By default it has nothing in common with object content or object location, just random number. This behavior is controlled by `-XX:hashCode=n` HotSpot JVM option which has the following possible values:

- 0: use global random generator. This is default setting in Java 7. It has the disadvantage that concurrent calls from multiple threads may cause a race condition which will result in generating the same hashCode for different objects. Also in highly-concurrent environment delays are possible due to contention (using the same memory region from different CPU cores).
- 5: use some thread-local xor-shift random generator which is free from the previous disadvantages. This is default setting in Java 8.
- 1: use object pointer mixed with some random value which is changed on the "stop-the-world" events, so between stop-the-world events (like garbage collection) generated hashCodes are stable (for testing/debugging purposes)
- 2: use always `1` (for testing/debugging purposes)
- 3: use autoincrementing numbers (for testing/debugging purposes, also global counter is used, thus contention and race conditions are possible)

- 4: use object pointer trimmed to 32 bit if necessary (for testing/debugging purposes)

Note that even if you set `-XX:hashCode=4`, the hashCode will not always point to the object address. Object may be moved later, but hashCode will stay the same. Also object addresses are poorly distributed (if your application uses not so much memory, most objects will be located close to each other), so you may end up having unbalanced hash tables if you use this option.

answered Sep 8 '15 at 9:55

Tagir Valeev

62.6k 10 137 223

The implementation of `hashCode()` may differ from class to class but the contract for `hashCode()` is very specific and stated clearly and explicitly in the Javadocs:

> Returns a hash code value for the object. This method is supported for the benefit of hashtables such as those provided by java.util.Hashtable.
>
> The general contract of hashCode is:
>
> - Whenever it 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.
>
> As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the JavaTM programming language.)

`hashCode()` is closely tied to `equals()` and if you override `equals()`, you should also override `hashCode()`.

edited Nov 17 '15 at 4:22

JeffC
10.8k 3 14 32

answered Feb 10 '10 at 15:10

Asaph
119k 18 156 171

1 "hashCode() is closely tied to equals() and if you implement one, you should implement the other" is not entirely correct. You only need to override hashCode if equals is overridden. It is technically valid to override hashCode without overriding equals. – Steve Kuo Feb 11 '10 at 3:09

@Steve Kuo: Fair enough. I re-worded the last sentence based on your comment. – Asaph Feb 11 '10 at 4:54

If hashcode is not overriden you will call Object's hashcode, here is an excerpt from its javadoc:

> As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the JavaTM programming language.)

answered Feb 10 '10 at 17:47

pgras
10.5k 1 28 40

---

the default hashcode implementation gives the internal address of the object in the jvm, as a 32 bits integer. Thus, two different (in memory) objects will have different hashcodes.

This is consistent with the default implementation of equals. If you want to override equals for your objects, you will have to adapt hashCode so that they are consistent.

See http://www.ibm.com/developerworks/java/library/j-jtp05273.html for a good overview.

answered Feb 10 '10 at 15:09

tonio
8,624 1 36 52

You should try to implement the hash code so that different objects will give different results. I don't think there is a standard way of doing this.

Read this article for some information.

answered Feb 10 '10 at 15:06

kgiannakakis
85.1k 19 141 183

A hashcode is useful for storing an object in a collection, such as a hashset. By allowing an Object to define a Hashcode as something unique it allows the algorithm of the HashSet to work effectively.

Object itself uses the Object's address in memory, which is very unique, but may not be very useful if two different objects (for example two identical strings) should be considered the same, even if they are duplicated in memory.

answered Feb 10 '10 at 15:09

Yishai
68.8k 19 156 235

Two objects with different hash code must not be equal with regard to equals()

`a.hashCode() != b.hashCode()` must imply `!a.equals(b)`

However, two objects that are not equal with regard to equals() can have the same hash code. Storing these objects in a set or map will become less efficient if many objects have the same hash code.

answered Feb 10 '10 at 15:11

ndeuma
469 4 11

Not really an answer but adding to my earlier comment

> internal address of the object cannot be guaranteed to remain unchanged in the JVM, whose garbage collector might move it around during heap compaction.

I tried to do something like this:

```
public static void main(String[] args) {
    final Object object = new Object();
    while (true) {
        int hash = object.hashCode();
        int x = 0;
        Runtime r = Runtime.getRuntime();
        List<Object> list = new LinkedList<Object>();
        while (r.freeMemory() / (double) r.totalMemory() > 0.3) {
            Object p = new Object();
            list.add(p);
            x += object.hashCode();//ensure optimizer or JIT won't remove this
        }
        System.out.println(x);
        list.clear();
        r.gc();
        if (object.hashCode() != hash) {
            System.out.println("Voila!");
            break;
        }
    }
}
```

But the hashcode indeed doesn't change... can someone tell me how Sun's JDK actually implements Obect.hashcode?

edited Dec 14 '11 at 15:56

answered Dec 14 '11 at 15:47

Ustaman Sangat

1,085 9 23

OpenJDK hg.openjdk.java.net/jdk6/jdk6-gate/jdk/file/tip/src/share/… has it as a native function. I would like to see the native function implementation...stackoverflow.com/questions/410756/…. Anyone? – Ustaman Sangat Dec 14 '11 at 15:54

Maybe the default hashcode method stores the first value and never changes it. – Ustaman Sangat Dec 14 '11 at 18:28

returns 6 digit hex number. This is usually the memory location of the slot where the object is addressed.

From an algorithmic per-se, I guess JDK does double hashing (native implementation) which is one of the best hashing functions for open addressing. This double hashing scheme highly reduces the possibility of collisions.

The following post will give a supportive idea -

Java - HashMap confusion about collision handling and the get() method

edited May 23 '17 at 12:17

Community ♦

1 1

answered Oct 17 '12 at 16:24

Larsen

317 2 9

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 collection`s, `including HashMap, HashSet, and Hashtable`.

answered Nov 30 '14 at 17:44

anish

2,504 8 36 84
