How is hashCode() calculated in Java

What value is `hashCode()` method is returning in java?

I read that it is an memory reference of an object... When I print hash value for `new Integer(1)` it is 1; for `String("a")` is 97.

I am confused: is it ASCII or what type of value is?

java hashcode

edited May 7 '15 at 12:00
nbro
5,124 8 42 85

asked Mar 11 '10 at 18:32
Jothi
5,489 18 55 85

8 Answers

A hashcode is an integer value that represents the state of the object upon which it was called. That is why an `Integer` that is set to 1 will return a hashcode of "1" because an `Integer's` hashcode and its value are the same thing. A character's hashcode is equal to it's ASCII character code. If you write a custom type you are responsible for creating a good `hashCode` implementation that will best represent the state of the current instance.

answered Mar 11 '10 at 18:33
Andrew Hare
262k 48 560 591

The value returned by `hashCode()` is by no means guaranteed to be the memory address of the object. I'm not sure of the implementation in the `Object` class, but keep in mind most classes will override `hashCode()` such that two instances that are semantically equivalent (but are not the same instance) will hash to the same value. This is especially important if the classes may be used within another data structure, such as Set, that relies on `hashCode` being consistent with `equals` .

There is no `hashCode()` that uniquely identifies an instance of an object no matter what. If you want a hashcode based on the underlying pointer (e.g. in Sun's implementation), use `System.identityHashCode()` - this will delegate to the default `hashCode` method regardless of whether it has been overridden.

Nevertheless, even `System.identityHashCode()` can return the same hash for multiple objects. See the comments for an explanation, but here is an example program that continuously generates objects until it finds two with the same `System.identityHashCode()` . When I run it, it quickly finds two `System.identityHashCode()` s that match, on average after adding about 86,000 Long wrapper objects (and Integer wrappers for the key) to a map.

```
public static void main(String[] args) {
    Map<Integer,Long> map = new HashMap<>();
    Random generator = new Random();
    Collection<Integer> counts = new LinkedList<>();

    Long object = generator.nextLong();
    // We use the identityHashCode as the key into the map
    // This makes it easier to check if any other objects
    // have the same key.
    int hash = System.identityHashCode(object);
    while (!map.containsKey(hash)) {
        map.put(hash, object);
        object = generator.nextLong();
        hash = System.identityHashCode(object);
    }
    System.out.println("Identical maps for size:  " + map.size());
    System.out.println("First object value: " + object);
    System.out.println("Second object value: " + map.get(hash));
    System.out.println("First object identityHash:  " +
System.identityHashCode(object));
    System.out.println("Second object identityHash: " +
System.identityHashCode(map.get(hash)));
}
```

Example output:

```
Identical maps for size:  105822
First object value: 7446391633043190962
Second object value: -8143651927768852586
First object identityHash:  2134400190
Second object identityHash: 2134400190
```

edited Oct 14 '15 at 20:04
Josiah Yoder
798 1 10 25

answered Mar 11 '10 at 18:35
danben
57.2k 15 107 131

2 Any particular reason this was modded down? If something here is incorrect I would love a correction, but to downvote without explanation adds nothing to the discussion. – danben Mar 11 '10 at 20:09

6 A few years ago, Ted Neward explained in blogs.tedneward.com/2008/07/16/… how the OpenJDK implemented Object.hashCode(). The OpenJDK derives the hash code from the object address, but caches this value and returns it to subsequent callers in case the object moves in memory and its address changes. After briefly reviewing the latest code, I found that the implementation seems not to have changed since Neward wrote his article. – Derek Mahar Apr 27 '10 at 17:35

This would seem to support my answer. – danben Apr 27 '10 at 20:27

1 Wouldn't that mean that a moved object hashcode could hit another new object's hashcode? – devoured elysium Mar 8 '12 at 20:26

7 Yes: it could "hit" another new object's hashcode. Hashcodes are not to be considered unique. The "hashcode" is designed to "narrow down" uniqueness (for Hashtables), but you must always follow it with "equals". – ChrisCantrell Nov 12 '12 at 0:09

If you want to know how they are implmented, I suggest you read the source. If you are using an IDE you can just + on a method you are interested in and see how a method is implemented. If you cannot do that, you can google for the source.

For example, Integer.hashCode() is implemented as

```
public int hashCode() {
    return value;
}
```

and String.hashCode()

```
public int hashCode() {
    int h = hash;
    if (h == 0) {
        int off = offset;
        char val[] = value;
        int len = count;

        for (int i = 0; i < len; i++) {
            h = 31*h + val[off++];
        }
        hash = h;
    }
    return h;
}
```

answered Mar 13 '10 at 6:49

Peter Lawrey
421k 54 522 890

I've planned to answer precisely the same way; +1 – incarnate Mar 13 '10 at 7:05

@Peter Lawrey and how can I see the Object hashCode implementation – Java Geek Oct 7 '13 at 9:43

@naroji Its in the OpenJDK. Unfortunately there are multiple strategies and its not clear which one is used. – Peter Lawrey Oct 7 '13 at 13:39

I went to OpenJDK Object class, but there also it is defined as native.. hg.openjdk.java.net/jdk7/jdk7/jdk/file/tip/src/share/classes/… – Java Geek Oct 8 '13 at 4:10

> I read that it is an memory reference of an object..

No. `Object.hashCode()` used to return a memory address about 14 years ago. Not since.

> what type of value is

What it is depends entirely on what class you're talking about and whether or not it has overridden `Object.hashCode().

edited Aug 4 '16 at 12:17

answered Mar 13 '10 at 1:26

EJP

251k 22 197 335

1 This does not provide an answer to the question. To critique or request clarification from an author, leave a comment below their post. – chŝdk May 24 '15 at 13:59

@chsdk Rubbish. It refutes an assertion made in the question. 'No' is an answer. NB I am not requesting clarification from anybody either. – EJP Aug 4 '16 at 12:18

this was an automatic comment, after a flag as a "not an answer", which was unclear and unexplained before your last edit 2 hrs ago. – chŝdk Aug 4 '16 at 15:05

1 @chsdk I am unable to discern anything either 'unclear' or 'unexplained' about 'used to' and 'not since'. How the comment arrived here is of no concern. – EJP Apr 20 '17 at 3:10

hmmm...we are in 2017 and now it is 21 years ago !! – Not a bug Jun 16 '17 at 13:18

The `hashCode()` method is often used for identifying an object. I think the `Object` implementation returns the pointer (not a real pointer but a unique id or something like that) of the object. But most classes override the method. Like the `String` class. Two String objects have not the same pointer but they are equal:

```
new String("a").hashCode() == new String("a").hashCode()
```

I think the most common use for `hashCode()` is in `Hashtable`, `HashSet`, etc..

Java API Object hashCode()

Edit: (due to a recent downvote and based on an article I read about JVM parameters)

With the JVM parameter `-XX:hashCode` you can change the way how the hashCode is calculated (see the Issue 222 of the Java Specialists' Newsletter).

> HashCode==0: Simply returns random numbers with no relation to where in memory the object is found. As far as I can make out, the global read-write of the seed is not optimal for systems with lots of processors.
>
> HashCode==1: Counts up the hash code values, not sure at what value they start, but it seems quite high.
>
> HashCode==2: Always returns the exact same identity hash code of 1. This can be used to test code that relies on object identity. The reason why JavaChampionTest returned Kirk's URL in the example

> above is that all objects were returning the same hash code.
>
> HashCode==3: Counts up the hash code values, starting from zero. It does not look to be thread safe, so multiple threads could generate objects with the same hash code.
>
> HashCode==4: This seems to have some relation to the memory location at which the object was created.
>
> HashCode>=5: This is the default algorithm for Java 8 and has a per-thread seed. It uses Marsaglia's xor-shift scheme to produce pseudo-random numbers.

edited Mar 31 '15 at 12:11

answered Mar 11 '10 at 18:48
alexvetter
1,558 1 13 38

If I am reading the code correctly, in Java 8 & 9 the default strategy is 5. – Stephen C Sep 1 '17 at 22:56

Object.hashCode(), if memory serves correctly (check the JavaDoc for java.lang.Object), is implementation-dependent, and will change depending on the object (the Sun JVM derives the value from the value of the reference to the object).

Note that if you are implementing any nontrivial object, and want to correctly store them in a HashMap or HashSet, you MUST override hashCode() and equals(). hashCode() can do whatever you like (it's entirely legal, but suboptimal to have it return 1.), but it's vital that if your equals() method returns true, then the value returned by hashCode() for both objects are equal.

Confusion and lack of understanding of hashCode() and equals() is a big source of bugs. Make sure that you thoroughly familiarize yourself with the JavaDocs for Object.hashCode() and Object.equals(), and I guarantee that the time spent will pay for itself.

answered Mar 11 '10 at 18:54
Ben Fowler
170 7

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 Java™ programming language.)

https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#hashCode--

answered Jan 5 at 17:46

Sam
42 2 8

```
public static int murmur3_32(int paramInt1, char[] paramArrayOfChar, int
paramInt2, int paramInt3) {
/* 121 */     int i = paramInt1;
/*     */
/* 123 */     int j = paramInt2;
/* 124 */     int k = paramInt3;
/*     */
/*     */     int m;
/* 127 */     while (k >= 2) {
/* 128 */       m = paramArrayOfChar[(j++)] & 0xFFFF | paramArrayOfChar[(j++)] <<
'\020';
/*     */
/* 130 */       k -= 2;
/*     */
/* 132 */       m *= -862048943;
/* 133 */       m = Integer.rotateLeft(m, 15);
/* 134 */       m *= 461845907;
/*     */
/* 136 */       i ^= m;
/* 137 */       i = Integer.rotateLeft(i, 13);
/* 138 */       i = i * 5 + -430675100;
/*     */     }
/*     */
/*     */
/*     */
/* 143 */     if (k > 0) {
/* 144 */       m = paramArrayOfChar[j];
/*     */
/* 146 */       m *= -862048943;
/* 147 */       m = Integer.rotateLeft(m, 15);
/* 148 */       m *= 461845907;
/* 149 */       i ^= m;
/*     */     }
/*     */
/*     */
/*     */
/* 154 */     i ^= paramInt3 * 2;
/*     */
/*     */
/* 157 */     i ^= i >>> 16;
/* 158 */     i *= -2048144789;
/* 159 */     i ^= i >>> 13;
/* 160 */     i *= -1028477387;
/* 161 */     i ^= i >>> 16;
/*     */
/* 163 */     return i;
/*     */   }
```

If you really curious to learn then go through this code available in Hashing.class ;

Here first param **HASHING_SEED** is calculated based on below code

```
 {
   long nanos = System.nanoTime();
   long now = System.currentTimeMillis();
   int SEED_MATERIAL[] = {
           System.identityHashCode(String.class),
           System.identityHashCode(System.class),
           (int) (nanos >>> 32),
           (int) nanos,
           (int) (now >>> 32),
           (int) now,
           (int) (System.nanoTime() >>> 2)
   };

   // Use murmur3 to scramble the seeding material.
   // Inline implementation to avoid loading classes
   int h1 = 0;

   // body
   for (int k1 : SEED_MATERIAL) {
       k1 *= 0xcc9e2d51;
       k1 = (k1 << 15) | (k1 >>> 17);
       k1 *= 0x1b873593;

       h1 ^= k1;
       h1 = (h1 << 13) | (h1 >>> 19);
       h1 = h1 * 5 + 0xe6546b64;
   }

   // tail (always empty, as body is always 32-bit chunks)

   // finalization
```

```
    h1 ^= SEED_MATERIAL.length * 4;

    // finalization mix force all bits of a hash block to avalanche
    h1 ^= h1 >>> 16;
    h1 *= 0x85ebca6b;
    h1 ^= h1 >>> 13;
    h1 *= 0xc2b2ae35;
    h1 ^= h1 >>> 16;

    HASHING_SEED = h1;
}
```

the second param is char array of String , third is always '0' and fourth one is char array length.

And the above calculation is just for String hash code.

For all integer, its hash code will be its integer value. For char(up to two letter) it will be ASCII code.

answered May 25 '17 at 18:37
arjun kumar mehta
33 6

I don't know where this code comes from but it has has nothing to do with how Java's `String::hashCode()` is calculated. Or any other standard Java type's hashcode ... AFAIK. The *real* code for `String::hashCode()` is here: grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/… – Stephen C Sep 1 '17 at 22:30